Функциональное программирование

Программируй на HASKELL

Языки программирования зачастую отличаются лишь отдельными элементами: несколько ключевых слов, библиотек или платформенных решений. Haskell формирует абсолютно новую точку зрения. По мнению пионера программного обеспечения Алана Кея, смена парадигмы может дать 80 баллов IQ, и разработчики на Haskell соглашаются с исключительными преимуществами мышления в стиле Haskell: функционального подхода с ориентацией на типобезопасность, математическую определённость и многое другое.

Эта книга проведёт вас через короткие уроки, примеры и упражнения, разработанные так, чтобы вы смогли прочувствовать Haskell. В книге вы найдёте кристально ясные иллюстрации и легко сможете попрактиковаться. Вы будете писать и тестировать дюжины интересных программ, а также погрузитесь в различные модули и библиотеки. В итоге перед вами откроется новая перспектива в программировании и возможность использовать Haskell в реальном мире (80 баллов IQ не гарантируются).

В этой книге:

- **мышление в стиле Haskell;**
- **основы функционального программирования;**
- **программирование на типах;**
- **приложения на Haskell в реальных проектах.**

«Доступное и тщательное введение в Haskell и функциональное программирование. Эта книга изменит то, как вы думаете о программировании, в лучшую сторону.»
— Макаранд Дешпандэ, SAS R&D

«Я думал, что Haskell сложен для изучения. Благодаря этой книге выяснилось, что это не так.»
— Миккель Арентофт, Danske Bank

Уилл Курт сейчас работает аналитиком данных. Он ведет блог на www.countbayesie.com, где объясняет анализ данных обычным людям.

Интернет-магазин:
www.dmkpress.com
Книга – почтой:
e-mail: orders@alians-kniga.ru
Оптовая продажа:
«Альянс-книга»
Тел./факс: (499) 782-3889
e-mail: books@alians-kniga.ru

ДМК издательство www.дмк.рф

MANNING

ISBN 978-5-97060-694-0

Функциональное программирование

Уилл Курт

Программируй на HASKELL

Программируй на HASKELL

Уилл Курт

Программируй на Haskell

Москва, 2019

Get Programming with Haskell

ЛАНЬ

Will Kurt

ЛАНЬ

Manning Publications

Программируй на Haskell

Уилл Курт

Перевод с английского
Я. О. Касюлевича, А. А. Романовского и С. Д. Степаненко,
под ред. В. Н. Брагилевского

Москва, 2019

УДК 004.432.42 Haskell
ББК 32.973.28-018.1
К93

К93 **Уилл Курт**
Программируй на Haskell / пер. с англ. Я. О. Касюлевича, А. А. Романовского и С. Д. Степаненко; под ред. В. Н. Брагилевского. – М.: ДМК Пресс, 2019. — 648 с.: ил.
ISBN 978-5-97060-694-0

Языки программирования зачастую отличаются лишь отдельными элементами — несколько ключевых слов, библиотек или платформенных решений. Haskell формирует абсолютно новую точку зрения. По мнению пионера программного обеспечения Алана Кэя, смена перспективы может дать 80 баллов IQ, и разработчики на Haskell соглашаются с исключительными преимуществами мышления в стиле Haskell: функционального мышления с ориентацией на типобезопасность, математическую определённость и многое другое. В этой практической книге вы будете учиться именно этому.

«Программируй на Haskell» проведёт вас через короткие уроки, примеры и упражнения, разработанные так, чтобы вы смогли прочувствовать Haskell. В ней вы найдёте кристально ясные иллюстрации и легко сможете практиковаться под её руководством. Вы будете писать и тестировать дюжины интересных программ, а также погрузитесь в различные модули и библиотеки. Вы получите новую перспективу в программировании и возможность использовать Haskell в реальном мире (80 баллов IQ не гарантируются).

Написано для читателей, который уже знают хотя бы один язык программирования.

УДК 004.432.42 Haskell
ББ 32.973.28-018.1

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок всё равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несёт ответственности за возможные ошибки, связанные с использованием книги.

ISBN 9781617293764 (англ.)
ISBN 978-5-97060-694-0 (рус.)

© 2018 Will Kurt
© Перевод на русский язык, оформление, ДМК Пресс, 2019

Лизе и Арчеру, источникам бесконечной поддержки и вдохновения

Оглавление

Предисловие 13

Благодарности 15

Об этой книге 17

Об авторе 21

Урок 1. Начало работы с Haskell 22
1.1. Добро пожаловать в мир Haskell 22
1.2. Компилятор GHC языка Haskell 23
1.3. Взаимодействие с Haskell — GHCi 25
1.4. Написание кода на Haskell и работа с ним 28

Модуль 1. Основания функционального программирования 34

Урок 2. Функции и функциональное программирование 36
2.1. Функции 37
2.2. Функциональное программирование 38
2.3. Функциональное программирование на практике 39

Урок 3. Лямбда-функции и лексическая область видимости 46
3.1. Лямбда-функции 47
3.2. Пишем свой аналог блока where 48
3.3. От лямбда-функций к let: изменяемые переменные! 51
3.4. Лямбда-функции и области видимости на практике 53

Урок 4. Функции как значения первого класса 57
4.1. Функции как аргументы 58
4.2. Возвращаем функции 63

Урок 5. Замыкания и частичное применение функций 67

5.1. Замыкания — создание функций функциями 68
5.2. Пример: генерация URL для API 69
5.3. Собираем всё вместе 75

Урок 6. Списки **78**
6.1. Анатомия списков 79
6.2. Списки и ленивые вычисления 82
6.3. Основные функции на списках 84

Урок 7. Правила рекурсии и сопоставление с образцом **90**
7.1. Рекурсия 91
7.2. Правила рекурсии 92
7.3. Ваша первая рекурсивная функция: наибольший общий делитель 94

Урок 8. Написание рекурсивных функций **99**
8.1. Обзор: правила рекурсии 100
8.2. Рекурсия на списках 100
8.3. Патологическая рекурсия: функция Аккермана и гипотеза Коллатца 103

Урок 9. Функции высшего порядка **109**
9.1. Использование map 110
9.2. Обобщение вычислений с помощью map 111
9.3. Фильтрация списка 113
9.4. Свёртка списка 114

Урок 10. Итоговый проект: функциональное объектно-ориентированное программирование и роботы! **119**
10.1. Объект с одним свойством: кружка кофе 120
10.2. Более сложные объекты: создаём боевых роботов! 124
10.3. Почему важно программировать без состояний 128
10.4. Типы — объекты и многое другое! 130

Модуль 2. Введение в типы 132

Урок 11. Основы системы типов **134**
11.1. Типы в Haskell 135
11.2. Типы функций 138
11.3. Типовые переменные 143

Урок 12. Создание пользовательских типов **148**
12.1. Использование синонимов типов 149
12.2. Создание новых типов 151
12.3. Использование синтаксиса записей 156

Урок 13. Классы типов **161**
13.1. Дальнейшее исследование типов 162
13.2. Классы типов 163
13.3. Преимущества классов типов 164
13.4. Определение класса типов 164
13.5. Основные классы типов 166
13.6. Порождение экземпляров классов типов 169

Урок 14. Использование классов типов **172**
14.1. Тип, требующий классы 173
14.2. Реализация Show 173
14.3. Классы типов и полиморфизм 174
14.4. Реализации методов по умолчанию и минимально полные определения 176
14.5. Реализация Ord 178
14.6. Порождать или нет? 180
14.7. Классы типов для более сложных типов 182
14.8. Схема классов типов 184

Урок 15. Итоговый проект: секретные сообщения! **186**
15.1. Шифры для начинающих: ROT13 186
15.2. XOR: магия криптографии! 194
15.3. Представление значений как битов 196
15.4. Одноразовый блокнот 199
15.5. Класс Cipher 201

Модуль 3. Программирование в типах 205

Урок 16. Создание типов с помощью «И» и «ИЛИ» **207**
16.1. Типы-произведения — объявление типов с помощью «И» . . 208
16.2. Типы-суммы — объявление типов с помощью «ИЛИ» 213
16.3. Собираем книжный магазин 216

Урок 17. Проектирование композицией: полугруппы и моноиды **220**
17.1. Введение в композицию: комбинирование функций 221
17.2. Комбинирование схожих типов: полугруппы 222

17.3. Композиция с нейтральным элементом: моноиды 226
17.4. Комбинирование элементов моноида функцией mconcat . . 228

Урок 18. Параметризованные типы **235**
18.1. Типы, которые принимают аргументы 236
18.2. Типы с более чем одним параметром 241

Урок 19. Тип Maybe: работа с отсутствующими значениями **248**
19.1. Maybe: возможность отсутствия значения как тип 249
19.2. Проблема с null 251
19.3. Вычисления с Maybe 253
19.4. Назад в лабораторию! Снова вычисления с Maybe 255

Урок 20. Итоговый проект: временные ряды **260**
20.1. Данные и тип для их представления 261
20.2. Сшивание временных рядов 265
20.3. Вычисления на временных рядах 270
20.4. Преобразование временных рядов 273
20.5. Скользящее среднее 275

Модуль 4. Ввод и вывод в Haskell 279

Урок 21. «Привет, мир!» — введение в ввод-вывод **282**
21.1. Типы IO: работаем с «грязным» миром 283
21.2. Do-нотация 288
21.3. Пример: вычисление стоимости пиццы 290

Урок 22. Командная строка и ленивый ввод-вывод **295**
22.1. Энергичное взаимодействие с командной строкой 296
22.2. Взаимодействие с ленивым вводом-выводом 301

Урок 23. Работа с типом Text и Юникодом **307**
23.1. Тип Text 308
23.2. Использование Data.Text 309
23.3. Тип Text и Юникод 315
23.4. Ввод-вывод для Text 317

Урок 24. Работа с файлами **319**
24.1. Открытие и закрытие файлов 320
24.2. Простые средства ввода-вывода 323
24.3. Проблемы ленивого ввода-вывода 325

24.4. Строгий ввод-вывод 328

Урок 25. Работа с двоичными данными **331**
25.1. Обработка двоичных данных с помощью ByteString 332
25.2. Добавление помех на изображение JPEG 334
25.3. ByteString, Char8 и Юникод 343

Урок 26. Итоговый проект: обработка двоичных файлов и книжных данных **346**
26.1. Работа с книжными данными 348
26.2. Работа с MARC-записями 351
26.3. Собираем всё вместе 362

Модуль 5. Работа с типами в контексте 365

Урок 27. Класс типов Functor **369**
27.1. Пример: вычисление с Maybe 370
27.2. Класс типов Functor и вызов функций в контексте 372
27.3. Функторы повсюду! 374

Урок 28. Приступаем к аппликативным функторам: функции в контексте **382**
28.1. Расчёт расстояния между городами 383
28.2. Операция <*> и частичное применение в контексте 387
28.3. Использование <*> для данных в контексте 393

Урок 29. Списки как контекст: углубляемся в аппликативные вычисления **397**
29.1. Представляем класс типов Applicative 398
29.2. Контейнеры и контексты 401
29.3. Список как контекст 403

Урок 30. Введение в класс типов Monad **412**
30.1. Ограничения Applicative и Functor 413
30.2. Операция (>>=) 419
30.3. Класс типов Monad 421

Урок 31. Облегчение работы с монадами с помощью do-нотации **427**
31.1. Возвращаемся к do-нотации 428
31.2. Использование кода в разных контекстах и do-нотация . . . 431
31.3. Контекст списка — обработка списка кандидатов 436

Урок 32. Монада списка и генераторы списков **442**
32.1. Построение списков при помощи монады 443
32.2. Генераторы списков 447
32.3. Монады: больше, чем просто списки 449

Урок 33. Итоговый проект: SQL-подобные запросы в Haskell **451**
33.1. Начало работы 452
33.2. Простые запросы на списках: select и where 455
33.3. Соединение типов данных Course и Teacher 457
33.4. Построение интерфейса HINQ и тестовые запросы 459
33.5. Определение типа HINQ для запросов 461
33.6. Выполнение HINQ-запросов 462

Модуль 6. Организация кода и сборка проектов 468

Урок 34. Организация кода на Haskell с помощью модулей **469**
34.1. Что случится, если использовать имя из Prelude? 470
34.2. Сборка многофайловой программы с помощью модулей . . 473

Урок 35. Сборка проектов при помощи stack **480**
35.1. Создание нового проекта stack 481
35.2. Разбор структуры проекта 482
35.3. Написание кода 485
35.4. Сборка и запуск вашего проекта 487

Урок 36. Тестирование свойств с помощью QuickCheck **490**
36.1. Создание нового проекта 491
36.2. Разные виды тестирования 492
36.3. Тестирование свойств с помощью QuickCheck 497

Урок 37. Итоговый проект: библиотека для простых чисел **504**
37.1. Создание нового проекта 505
37.2. Изменение файлов, созданных по умолчанию 506
37.3. Реализация основных библиотечных функций 507
37.4. Написание тестов для кода 511
37.5. Написание кода факторизации чисел 515

Модуль 7. Применение Haskell на практике 519

Урок 38. Ошибки в Haskell и тип Either **521**

38.1. Функция head, частичные функции и ошибки 522
38.2. Обработка частичных функций с помощью Maybe 526
38.3. Первая встреча с Either 528

Урок 39. Создание HTTP-запросов в Haskell **535**
39.1. Первоначальная настройка проекта 536
39.2. Использование модуля HTTP.Simple 539
39.3. Создание HTTP-запроса 542
39.4. Собираем всё вместе 544

Урок 40. Работа с данными JSON с использованием Aeson **546**
40.1. Первоначальная настройка 548
40.2. Использование библиотеки Aeson 549
40.3. Экземпляры FromJSON и ToJSON для своих типов 551
40.4. Чтение данных, полученных от NOAA 559

Урок 41. Использование баз данных в Haskell **563**
41.1. Первоначальная настройка проекта 564
41.2. Использование SQLite и настройка базы данных 565
41.3. Вставка данных: пользователи и данные об аренде 569
41.4. Чтение данных из БД и класс типов FromRow 571
41.5. Модификация существующих данных 575
41.6. Удаление данных из БД 578
41.7. Собираем всё вместе 578

Урок 42. Эффективные массивы с изменением состояния в Haskell **583**
42.1. Тип UArray и эффективные массивы 585
42.2. Изменение состояния с помощью STUArray 592
42.3. Извлечение значений из контекста ST 595
42.4. Реализация сортировки методом пузырька 597

Послесловие **601**

Примерные решения задач **607**

Предметный указатель **631**

Предисловие

Когда ко мне обратились с идей написать *«Программируй на Haskell»*, я не был уверен, стоит ли мне это делать. В то время моим главным интересом было написание блога Count Bayesie по теории вероятностей. Хотя у меня уже был опыт преподавания как Haskell, так и вообще функционального программирования, с тех пор прошло время и, откровенно говоря, я немного вышел из формы. Мой активный интерес к анализу данных, теории вероятностей и машинному обучению родился из личного разочарования в Haskell. Конечно, язык был красив и мощен, но с помощью нескольких некрасивых строк в R и линейной алгебры я мог выполнять сложный анализ и строить модели, чтобы предсказывать будущее. В Haskell даже ввод/вывод нетривиален! Едва ли я был подходящим евангелистом для написания книги по Haskell.

Затем я вспомнил цитату Д. Д. Сэлинджера из книги «Симор: Введение», где он описывает такую уловку, для того чтобы начать писать:

> Спроси себя как читателя, какую вещь ты хотел бы прочитать больше всего на свете, если бы тебе предложили выбрать что-то по душе? И мне просто не верится, как жутко и вместе с тем как просто будет тогда сделать шаг, о котором я сейчас тебе напишу. Тебе надо будет сесть и без всякого стеснения самому написать такую вещь.

В тот момент я осознал, что именно поэтому я должен написать *Программируй на Haskell*. Есть много хороших книг по Haskell, но ни одна из них

не утоляла мою жажду изучения Haskell. Я всегда хотел прочитать книгу, которая покажет, как решать практические задачи, что зачастую является настоящей болью на Haskell. Я хотел видеть не столько большие промышленного уровня программы, сколько забавные эксперименты, которые позволяют вам исследовать мир с помощью этого впечатляющего языка программирования. Я также всегда хотел прочитать книгу по Haskell, которая разумно коротка и после прочтения позволит мне комфортно заниматься различными весёлыми проектами на Haskell по выходным. Именно такого воплощения книги о Haskell, которую я хотел прочитать, ещё не существовало, и я решил, что написать *«Программируй на Haskell»* было бы неплохо.

Теперь, когда я закончил писать (и читать) эту книгу, я в восторге от того, сколько удовольствия получил. Haskell — это бесконечно интересный язык, в котором всегда есть что-то ещё, чему можно научить. Это сложный для изучения язык, но в этом часть удовольствия. Почти каждая тема в этой книге не будет похожа на то, что вы видели раньше (только если вы не опытный разработчик на Haskell). Радость при изучении Haskell состоит в том, чтобы открываться богатому обучающему опыту. Если вы слишком поспешите в освоении Haskell, то это может оказаться ужасным времяпрепровождением. Однако если вы потратите время на исследования, снова став новичком, то будете вознаграждены.

Благодарности

Написание книги — это огромная работа, и автор — всего лишь один из многих людей, необходимых для обеспечения успеха проекта. Первые, кого я должен поблагодарить, — это люди, оказывавшие мне эмоциональную и интеллектуальную поддержку во время этого большого приключения. Моя жена Лиза и сын Арчер были очень терпеливы к моим долгим часам работы и бесконечно поддерживали меня на протяжении всего пути. Также я должен поблагодарить моих дорогих друзей Ричарда Келли и Хавьера Бенгочея, которые были постоянным источником обратной связи, поддержки и интеллектуальной стимуляции. Этой книги никогда бы не было, если бы мой научный руководитель, Фред Харрис, не дал мне удивительную возможность преподавать Haskell группе восторженных студентов. Я также хотел бы поблагодарить моих коллег в Quick Sprout: Стива Кокса, Иена Мэйна и Хитена Ша, которые вынесли мою бесконечную болтовню о Haskell в течение прошлого года.

Сложно переоценить вклад невероятной команды Manning в эту книгу, помощь оказало больше людей, чем может здесь поместиться. Эта книга была бы тенью того, чем она является, без поддержки моего редактора, Дэна Махарри. Дэн требовал доводить каждую мою хорошую идею до совершенства. Я также должен поблагодарить Эрин Тухей за то, что она была первым человеком, кому пришла безумная идея, что я должен написать книгу по Haskell. Мой технический редактор, Палак Матур, проделал отличную работу, обеспечив лёгкое следование за техническим наполнением этой книги и его понимание. Я также должен поблагодарить Виталия Бра-

гилевского за предоставление ценной обратной связи по улучшению кода в этой книге и Шерон Уилки за её терпеливое редактирование. Наконец, я хотел бы упомянуть рецензентов, которые потратили своё время на прочтение и комментирование этой книги: Александр Мыльцев, Арно Байли, Карлос Айя, Клаудио Родригез, Герман Гонзалез-Моррис, Хемант Капила, Джеймс Анаипакос, Кай Геллиен, Макран Дешпанде, Миккель Арентофт, Никита Дюмин, Питер Хемптон, Ричард Тобиас, Серхио Мартинез, Виктор Татай, Виталий Брагилевский и Юрий Клайман.

Об этой книге

Цель книги *«Программируй на Haskell»* — в том, чтобы дать достаточно полное введение в программирование на языке Haskell, позволяющее вам после её завершения писать нетривиальные, полезные на практике программы. Многие другие книги сильно фокусируются на академических основаниях Haskell, но зачастую оставляют читателей немного озадаченными, когда дело доходит до решения практических задач, совершенно обыденных в других языках. К концу этой книги у вас должно возникнуть стойкое понимание того, что именно делает Haskell интересным как язык программирования, вы также сможете уверенно создавать не совсем игрушечные приложения, которые работают с вводом-выводом, генерируют случайные числа, используют базы данных и в целом выполняют те же вещи, что и программы на других знакомых вам языках программирования.

Кому следует читать эту книгу

Это книга для всех, у кого есть опыт программирования и кто хочет поднять свои навыки программирования и понимания языков программирования на новый уровень. Вы можете прийти к своему заключению относительно практичности Haskell, но существуют две хорошие и вполне прагматичные причины для его изучения.

В первую очередь, даже если вы больше никогда не притронетесь к Haskell, получение навыков программирования на Haskell сделает вас более сильным программистом в целом. Haskell принуждает вас писать безопасный функциональный код, а также аккуратно моделировать ваши задачи. Обучение работе с Haskell научит вас правильнее рассуждать об абстракциях и предотвращать потенциальные ошибки в любых языках про-

граммирования. Не уверен, что мне удастся повстречать разработчика программного обеспечения, который хорошо разбирается в Haskell, но при этом не является программистом уровня выше среднего.

Второе преимущество изучения Haskell — в том, что оно, по сути, сопровождается ускоренным курсом теории языков программирования. Вы вряд ли сможете изучить Haskell на уровне, достаточном для написания нетривиальных программ, обойдясь без значительного объёма знаний о функциональном программировании, ленивых вычислениях и сложных системах типов. Эти основы теории языков программирования не только полезны из академического любопытства, но и служат вполне прагматичным целям. Элементы Haskell постоянно проникают как в новые языки программирования, так и в уже существующие. Знание Haskell и его особенностей поможет вам понимать, чего можно ожидать на горизонтах программирования на годы вперёд.

Как организована эта книга

Структура *«Программируй на Haskell»* может отличаться от многих книг по программированию, которые вы читали раньше. Вместо длинных глав эта книга поделена на короткие и простые для усвоения уроки. Эти уроки объединены в семь модулей, которые покрывают основной материал. Все модули, кроме последнего, заканчиваются итоговыми проектами. Эти итоговые проекты объединяют всё освоенное в модуле с целью разработки расширенного примера. Все уроки содержат упражнения с возможностью быстрой проверки и несложные вопросы, с помощью которых можно убедиться, что вы всё схватываете. В конце каждого урока мы даём несколько более серьёзных задач (их решения приведены в конце книги). Модули покрывают следующее содержание:

- *модуль 1* — этот модуль закладывает основы функционального программирования в целом, а также представляет большинство уникальных характеристик Haskell — после прочтения этого модуля вы будете достаточно знакомы с основами функционального программирования и сможете изучать любой другой функциональный язык, находя при этом материал знакомым;
- *модуль 2* — здесь вы начнёте рассматривать мощную систему типов языка Haskell — этот модуль покрывает базовые типы вроде `Int`, `Char` и `Boolean` и как с их помощью создавать свои типы данных, вы также начнёте рассматривать систему классов типов Haskell, которая позволяет вам использовать одну и ту же функцию с данными различных типов;

- *модуль 3* — когда вы изучите основы типов в Haskell, вы сможете перейти к более абстрактным типам и классам типов, которые делают Haskell таким мощным, вы увидите, как Haskell позволяет комбинировать типы способами, которые невозможны в большинстве других языков программирования, вы изучите классы типов `Monoid` и `Semigroup`, а также увидите, как тип `Maybe` позволяет избавиться от целого класса ошибок в ваших программах;
- *модуль 4* — наконец, вы достаточно изучили Haskell, чтобы обсудить ввод-вывод — этот модуль представляет все основы исполнения ввода-вывода в Haskell и объясняет, что делает его уникальным (и порой сложным), к концу этого модуля вы сможете комфортно писать инструментарий для командной строки, читать и записывать файлы, работать с данными в Юникоде и преобразовывать двоичные данные;
- *модуль 5* — к этому моменту в книге вы уже встречали несколько типов, которые создают *контекст* для других типов. Типы `Maybe` определяют контекст для возможно отсутствующих значений, типы `IO` — это значения, которые имеют контекст использования при вводе-выводе. В этом модуле вы погрузитесь в семейство классов типов, которые необходимы для работы со значениями в контексте: `Functor`, `Applicative` и `Monad`. Хотя у них запугивающие имена, они играют довольно простую роль: применение любой функции в часто используемых контекстах. Несмотря на свою абстрактность эти концепции предлагают вам единый способ работы с типами `Maybe`, `IO` и даже списками;
- *модуль 6* — одна из самых сложных тем позади, время начать думать о написании кода для реального мира. Во-первых, вам надо убедиться, что ваш код правильно организован. Этот модуль начинается с урока по системе модулей в Haskell. Затем в оставшейся части модуля вы будете изучать stack, мощный инструмент для создания и поддержки проектов на Haskell;
- *модуль 7* — мы завершим эту книгу, взглянув на то, чего нам не хватало для работы с Haskell в реальном мире. Этот модуль начинается с обзора обработки ошибок в Haskell, которая отличается от многих других языков. После этого вы взглянете на три практические задачи, реализуемые с помощью Haskell: использование HTTP для создания запросов к REST API, разбор JSON-данных с использованием библиотеки Aeson и разработка приложения, общающегося с базой данных. Вы закончите чтение этой книги задачей, при выборе инструмента для решения которой вряд ли подумали бы о Haskell: эффективные, основанные на массивах алгоритмы с изменяемым состоянием.

Самое сложное в изучении (и преподавании) Haskell — то, что вам нужно пройти довольно большое количество тем, прежде чем вы сможете выполнять даже базовый ввод-вывод. Если ваша цель — понимать и использовать Haskell, то я советую вам читать модули последовательно. Но цель этой книги состоит в том, чтобы вы могли остановиться в нескольких местах, заполучив при этом нечто ценное. Модуль 1 организован так, чтобы предоставить вам крепкий фундамент для изучения любого функционального языка программирования. Будь это Clojure, Scala, F#, Racket или Common Lisp, все они разделяют основные черты, описанные в модуле 1. Если вы уже имеете опыт в функциональном программировании, вы можете пропустить модуль 1, хотя вам всё равно следует обратить внимание на уроки по частичному применению и ленивым вычислениям. К концу модуля 4 вы должны знать Haskell достаточно хорошо, чтобы играться с проектами на выходных. После модуля 5 вы вполне сможете переходить к более сложным темам самостоятельно. Модули 6 и 7 сконцентрированы на использовании Haskell для практических задач.

ЛАНЬ

О коде

В этой книге содержится много примеров кода. Код в этой книге представлен `шрифтом с~фиксированной шириной, таким как этот`, благодаря чему его несложно отличить от обычного текста. Многие фрагменты кода аннотированы цифрами, с помощью которых эти фрагменты потом можно прокомментировать. Более сложные примеры включают в себя стрелки, указывающие на каждый раздел и объясняющие его более детально. При написании кода на Haskell вы будете часто пользоваться REPL для взаимодействия с кодом. Эти секции будут отличаться от остальных, так как они будут иметь текст вида `GHCi>`, что обозначает место, где пользователь вводит код. К сожалению, GHCi по умолчанию отображает строки с кириллическими символами в виде последовательностей кодов, например так:

```
GHCi> "Привет, мир!"
"\1055\1088\1080\1074\1077\1090, \1084\1080\1088!"
```

Во всех таких случаях мы будем оставлять в выводе GHCi кириллицу. При желании вы можете сконфигурировать GHCi так, чтобы кириллица отображалась. Для этого можно установить пакет `unescaping-print`, дальнейшие инструкции приведены в описании пакета на Hackage: `http://hackage.haskell.org/package/unescaping-print`.

Также в тексте будут встречаться отсылки к командной строке, в которых используется символ `$` как обозначение места, где пользователь вводит свои команды.

Об авторе

Уилл Курт работает аналитиком в Bombora. Он имеет формальное образование в компьютерных науках (магистр) и английской литературе (бакалавр), а потому заинтересован в объяснении сложных технических тем настолько просто и доступно, насколько возможно. Он читал раздел курса, посвящённый Haskell, в Университете Невады (Рено) и вёл практические семинары по функциональному программированию. Также он ведёт блог о теории вероятностей на `CountBayesie.com`.

1

Начало работы с Haskell

После прочтения урока 1 вы:

- сможете установить инструменты для разработки на Haskell;
- начнёте использовать GHC и GHCi;
- научитесь пользоваться подсказками по написанию программ.

1.1. Добро пожаловать в мир Haskell

Прежде чем погрузиться в изучение Haskell, вам потребуется познакомиться с базовым инструментарием, который вы будете использовать на протяжении обучения. Этот урок поможет вам приступить к работе с Haskell, начнётся он с загрузки необходимого программного обеспечения для написания, компиляции и запуска программ на Haskell. Затем вы сможете взглянуть на примеры кода и вникнуть в суть программирования на Haskell. После всего этого вы будете готовы к полному погружению!

1.1.1. Haskell Platform

Худшая часть изучения нового языка программирования — это первоначальная установка окружения для разработки. К счастью, и к удивлению, это не проблема для Haskell. Haskell-сообщество собрало цельный и простой для установки пакет полезных инструментов, называемый *Haskell Platform*. Haskell Platform — способ распространения языка программирования «из коробки».

Haskell Platform включает в себя:

- компилятор языка Haskell (GHC);
- интерактивный интерпретатор (GHCi);
- утилиту stack для управления проектами на Haskell;
- набор полезных пакетов.

Дистрибутив Haskell Platform можно скачать по следующему адресу: `www.haskell.org/downloads#platform`. После этого следуйте инструкции по установке для той операционной системы, которую вы предпочитаете. Эта книга написана с расчётом на использование GHC версии 8.0.1 и выше.

1.1.2. Текстовые редакторы

Теперь, когда вы установили Haskell Platform, вы, возможно, полюбопытствуете по поводу того, какой редактор следует использовать. Известно, что Haskell — это язык, который настоятельно призывает *думать перед написанием кода*. В результате программы на Haskell обычно получаются очень лаконичными. Так что, помимо контроля отступов и подсветки синтаксиса, редактор здесь не помощник. Многие разработчики используют Emacs с `haskell-mode`. Но если вы незнакомы с Emacs (или не любите работать с ним), определённо не стоит заниматься изучением Emacs в дополнение к Haskell. Мы рекомендуем найти Haskell-плагин для того редактора, который вы используете чаще всего. Такие минималистичные редакторы, как Pico или Notepad++, прекрасно подойдут для наших целей, да и для большинства развитых IDE существуют нужные плагины.

1.2. Компилятор GHC языка Haskell

Haskell — компилируемый язык, а GHC — причина, по которой Haskell является настолько мощным языком. Основная цель компилятора заключается в переводе исходного кода, доступного людям для понимания, в бинарные инструкции, которые может принять компьютер. В Ruby, например, иной подход, там другая программа читает исходный код и интерпретирует его на лету (это делается с помощью *интерпретатора*). Главное преимущество компилятора перед интерпретатором в том, что компилятор может изменить код до запуска, что позволяет выполнить анализ и оптимизацию написанного вами кода. А благодаря другой конструктивной

особенности Haskell, а именно мощной системе типов, возникла поговорка: «Если это компилируется, то это работает». Несмотря на то что вы будете использовать GHC достаточно часто, не воспринимайте компилятор как должное. Этот удивительный образчик программного обеспечения сам по себе достоин отдельной книги.

Для вызова GHC откройте консоль и наберите `ghc`:

```
$ ghc
```

В этой книге если вы встречаете символ `$`, то это означает, что вам нужно напечатать что-то в командной строке. Конечно, без файлов для компиляции GHC будет возмущаться. Для начала вам нужно сделать простой файл hello.hs. В редакторе, который вы предпочитаете, создайте новый файл hello.hs и введите следующие строки:

Листинг 1.1 Пример первой программы hello.hs

```
--hello.hs мой первый Haskell-файл!
main = do
  putStrLn "Привет, мир!"
```

Закомментированная строка с названием вашего файла

Начало функции `main`

В функции `main` выводится строка `"Hello World!"`

На этой стадии не волнуйтесь по поводу того, что происходит в коде этого листинга. Сейчас ваша главная цель — изучить инструменты, которые вам потребуются, чтобы они не стояли на вашем пути при освоении языка.

Сейчас, когда у вас есть этот тестовый файл, запустите GHC снова, на этот раз указав hello.hs как аргумент:

```
$ ghc hello.hs
[1 of 1] Compiling Main
Linking hello ...
```

Если компиляция завершилась успешно, GHC создаст три файла:

- hello (для Windows hello.exe);
- hello.hi;
- hello.o.

Самый важный файл здесь — hello, являющийся исполняемым. А раз он исполняемый, значит, вы можете его запустить:

```
$ ./hello
Привет, мир!
```

Обратите внимание, что стандартное поведение скомпилированной программы — исполнение действий в `main`. Обычно Haskell-программы, которые вы будете компилировать, требуют наличия функции `main`, играющей роль, аналогичную роли метода `Main` в Java/C#, функций `main` в C/C++ или `__main__` в Python.

Как и большинство инструментов командной строки, GHC поддерживает великое множество необязательных флагов. Например, если вы хотите скомпилировать hello.hs в исполняемый файл helloworld, вы можете указать ключ `-o`:

```
$ ghc hello.hs -o helloworld
[1 of 1] Compiling Main
Linking hello ...
```

Для просмотра более полного списка ключей компиляции вызовите `ghc --help` (в этом случае аргумент с названием входного файла не требуется).

Проверка 1.1. Скопируйте код из hello.hs и скомпилируйте свой собственный исполняемый файл testprogram.

1.3. Взаимодействие с Haskell — GHCi

Одним из наиболее полезных инструментов для написания программ на Haskell является GHCi, интерактивный интерпретатор. Как и GHC, его можно запустить, введя простую команду: `ghci`. После запуска GHCi вы будете встречены новой командной строкой:

```
$ ghci
GHCi>
```

Ответ 1.1. Просто скопируйте код в файл и затем, находясь в том же каталоге, в котором был создан файл, введите следующее:

```
ghc hello.hs -o testprogram
```

В этой книге использование GHCi обозначается с помощью `GHCi>` в начале строк, которые вводите вы, и пустого префикса для строк, которые печатает GHCi. Первая вещь, которой следует научиться при использовании любой программы, которую вы запускаете из консоли, — это то, как из неё выйти! В GHCi для выхода достаточно ввести `:q` в командной строке:

```
$ ghci
GHCi> :q
Leaving GHCi.
```

Работа с GHCi очень похожа на работу с интерпретаторами большинства других интерпретируемых языков программирования, например таких, как Python и Ruby. Его можно использовать как обычный калькулятор:

```
ghci> 1+1
2
```

Вы также можете писать код в GHCi на лету:

```
ghci> x=2+2
ghci> x
4
```

До выхода восьмой версии компилятора в GHCi определения функций и переменных требовалось начинать с ключевого слова `let`. Теперь это необязательно, но во многих примерах кода на Haskell, которые можно найти в Интернете и старых книгах, это ключевое слово ещё встречается:

```
ghci> let f x = x+x
ghci> f 2
4
```

Наиболее значимый способ использования GHCi — взаимодействие с программами, которые вы пишете. Существует два способа загрузки файла в GHCi. Первый — передать имя файла в качестве аргумента `ghci`:

```
$ ghci hello.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
```

Другой — использовать команду `:l` (краткая версия `:load`) во время сеанса работы с GHCi:

```
$ ghci
GHCi> :l hello.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
```

В обоих случаях вы сможете вызвать реализованные функции:

```
ghci> :l hello.hs
ghci> main
Привет, мир!
```

В отличие от компиляции файлов с помощью GHC, вашим файлам не требуется наличие `main` для загрузки в GHCi. Каждый раз, загружая файл, вы переписываете существующие определения функций и переменных. Вы можете постоянно загружать ваш файл, пока работаете с ним и вносите изменения. Haskell является, пожалуй, уникальным языком из-за хорошей поддержки компилятора, а также естественной и простой в использовании интерактивной среды. Если вы пришли к Haskell после Python, Ruby или JavaScript, то будете чувствовать себя как дома, используя GHCi. А если знакомы с компилируемыми языками, такими как Java, C# или C++, то, скорее всего, при написании программ на Haskell будете поражаться, что работаете с компилируемым языком.

Проверка 1.2. Измените вашу первую программу так, чтобы она выводила `Привет, <Ваше имя>!`. Перезагрузите её в GHCi и протестируйте вывод.

Ответ 1.2. Измените файл следующим образом:

```
main = do
  putStrLn "Привет, Уилл!"
```

И загрузите его в GHCi:

```
GHCi> :l hello.hs
GHCi> main
Привет, Уилл!
```

1.4. Написание кода на Haskell и работа с ним

Одной из наиболее разочаровывающих вещей для начинающих программистов на Haskell является то, что базовые операции ввода-вывода довольно сложны. Очень часто при изучении нового языка распространённой практикой является вывод некой информации на каждом шаге выполнения, для того чтобы понять, как работает программа. В Haskell же этот способ отладки несколько затруднён. Обычное дело — столкнуться с ошибкой в программе на Haskell и запутанным сообщением о ней, абсолютно при этом не понимая, как действовать дальше.

Усугубляет эту проблему то, что чудесный компилятор Haskell достаточно строго проверяет корректность вашего кода. Если вы привыкли быстро написать программу, запустить её, а потом по-быстрому исправить сделанные ошибки, то Haskell вас разочарует. Haskell поощряет медитативное сидение и обдумывание встреченных проблем до запуска программы. После того как вы наберётесь опыта, мы уверены, это разочарование превратится в вашу любимую особенность Haskell. Оборотной стороной одержимости корректностью на этапе компиляции является то, что ваши программы будут работать гораздо более предсказуемо, чем вы, возможно, привыкли.

Секрет спокойного написания кода без частых столкновений с ошибками состоит в том, чтобы писать его маленькими кусочками и тестировать эти кусочки в процессе написания. Чтобы продемонстрировать этот способ работы, давайте возьмём сложную программу на Haskell и почистим её так, чтобы стали понятны отдельные фрагменты. Например, давайте разработаем консольное приложение для составления электронных писем с благодарностями читателям от авторов. Ниже находится первая, плохо написанная версия этой программы.

Листинг 1.2 Черновая версия first_prog.hs

```
messyMain :: IO ()
messyMain = do
    putStrLn "Кто получатель этого письма?"
    recipient <- getLine
    putStrLn "Название книги:"
    title <- getLine
    putStrLn "Кто автор этого письма?"
    author <- getLine
    putStrLn ("Дорогой " ++ recipient ++ "!\n"
          ++ "Спасибо за то, что купили \""
```

```
        ++ title ++ "\"!\nC уважением,\n" ++ author)
```

Ключевой момент в этой программе — одна большая цельная функция `messyMain`. Рекомендация писать модульный код универсальна при разработке программного обеспечения, но в Haskell особенно важно писать код, который вы сможете понимать и инспектировать. Эта программа работает, хотя и выглядит неряшливо. Если вы измените `messyMain` на `main`, то сможете её скомпилировать и запустить. Но вы также можете загрузить этот код в GHCi без изменений, при условии что находитесь в том же каталоге, что и first_prog.hs:

```
$ ghci
GHCi> :l first_prog.hs
[1 of 1] Compiling Main    ( first_prog.hs, interpreted)
Ok, modules loaded: Main.
```

Если GHCi вывел «Ok», то становится понятно, что код был успешно скомпилирован и нормально работает. Обратите внимание, что GHCi не волнует наличие `main`. Теперь можно устроить тестовый прогон программы (вводимые пользователем строки выделены полужирным шрифтом):

```
GHCi> messyMain
Кто получатель этого письма?
Читатель
Название книги:
Программируй на Haskell
Кто автор этого письма?
Уилл
Дорогой Читатель!
Спасибо за то, что купили "Программируй на Haskell"!
С уважением,
Уилл
```

Всё нормально, но было бы гораздо удобнее работать с кодом, разбитым на меньшие части. Основная задача — создать текст для письма, но можно заметить, что письмо состоит из трёх частей, связанных вместе: имя получателя, основная часть и подпись. Начнём с того, что сделаем отдельные функции для этих частей. Следующие строки нужно добавлять в файл first_prog.hs. Да и большинство функций и значений, которые вы встретите в книге, предназначаются для тех файлов, с которыми вы работаете в данный момент. Начнём с функции `toPart`:

```
toPart recipient = "Дорогой" ++ recipient ++ "!\n"
```

В данном случае вы легко сможете написать все три функции разом, но лучше работать вдумчиво, тестируя каждую функцию по мере написания. Для проверки загрузим файл в GHCi:

```
GHCi> :l first_prog.hs
[1 of 1] Compiling Main    ( first_prog.hs, interpreted )
Ok, modules loaded: Main.
GHCi> toPart "Читатель"
"ДорогойЧитатель!\n"
GHCi> toPart "Боб Смит"
"ДорогойБоб Смит!\n"
```

Этот метод работы с кодом, когда он сначала набирается в редакторе, а потом перезагружается в GHCi, будет основным при работе с данной книгой. Чтобы избежать повторений, будем предполагать, что команда `:l first_prog.hs` используется всегда, когда это необходимо.

Теперь, когда вы загрузили файл в GHCi, можно заметить небольшую ошибку — пропущенный пробел между словом «Дорогой» и именем получателя. Давайте посмотрим, как её можно исправить.

Листинг 1.3 Исправленная функция toPart

```
toPart recipient = "Дорогой " ++ recipient ++ "!\n"
```

И протестируем в GHCi:

```
GHCi> toPart "Джуд Лоу"
"Дорогой Джуд Лоу!\n"
```

Выглядит нормально. Теперь определим оставшиеся две функции, реализовав их вместе, но не забывая, что лучше всего реализовывать функции по одной, загружать их в GHCi для тестирования и только потом продвигаться дальше.

Листинг 1.4 Определения функций bodyPart и fromPart

```
bodyPart bookTitle = "Спасибо за то, что купили \""
                     ++ bookTitle ++ "\"!\n"
fromPart author = "С уважением,\n" ++ author
```

Эти функции вы также можете протестировать:

```
GHCi> bodyPart "Программируй на Haskell"
"Спасибо за то, что купили \"Программируй на Haskell\"!\n"
GHCi> fromPart "Уилл"
"С уважением,\nУилл"
```

Всё выглядит прекрасно! Теперь потребуется функция, которая свяжет всё воедино.

Листинг 1.5 Определение функции createEmail

```
createEmail recipient bookTitle author = toPart recipient ++
                                         bodyPart bookTitle ++
                                         fromPart author
```

Обратите внимание на то, как выровнены вызовы функций. Пробельные символы в Haskell значимы (хотя и не настолько существенно, как в Python). Любое форматирование отнюдь не случайно: если какие-то секции кода выровнены, то для этого есть причины. Большинство редакторов может автоматически следить за этим, если поставить плагины для работы с Haskell.

Вооружившись всеми написанными функциями, можно протестировать `createEmail`:

```
GHCi> createEmail "Читатель" "Программируй на Haskell" "Уилл"
"Дорогой Читатель!\nСпасибо за то, что купили
          ↳ \"Программируй на Haskell\"!\nС уважением,\nУилл"
```

Функции работают как положено, поэтому теперь можно соединить их в `main`.

Листинг 1.6 Версия first_prog.hs с аккуратной main

```
main = do
    putStrLn "Кто получатель этого письма?"
    recipient <- getLine
    putStrLn "Название книги:"
    title <- getLine
    putStrLn "Кто автор этого письма?"
    author <- getLine
    putStrLn (createEmail recipient title author)
```

Всё готово к компиляции, но всегда лучше сначала протестировать программу в GHCi:

```
GHCi> main
Кто получатель этого письма?
Читатель
Название книги:
Программируй на Haskell
Кто автор этого письма?
```

```
Уилл
Дорогой Читатель!
Спасибо за то, что купили "Программируй на Haskell"!
С уважением,
Уилл
```

Похоже, что все части вместе работают нормально, к тому же нам удалось протестировать их по отдельности и убедиться, что они работают так, как предполагалось. Наконец-то можно скомпилировать получившуюся программу:

```
$ ghc first_prog.hs
[1 of 1] Compiling Main    ( first_prog.hs, first_prog.o )
Linking first_prog ...
$ ./first_prog
Кто получатель этого письма?
Читатель
Название книги:
Программируй на Haskell
Кто автор этого письма?
Уилл
Дорогой Читатель!
Спасибо за то, что купили "Программируй на Haskell"!
С уважением,
Уилл
```

Только что вы написали свою первую рабочую программу на Haskell. Теперь, обладая основными знаниями о процессе разработки, вы можете погрузиться в удивительный мир Haskell!

Итоги

В этом уроке нашей главной целью было приступить к работе с Haskell. Начали мы с установки Haskell Platform, соединяющей в себе инструменты, которые мы будем использовать в данной книге. Эти инструменты включают: GHC, компилятор Haskell; GHCi, интерактивный интерпретатор, и stack, инструмент для сборки программ, который потребуется позже. В оставшейся части урока рассматривались вопросы написания и оформления кода на Haskell, а также способы взаимодействия с ним. Давайте проверим, насколько хорошо вы во всём этом разобрались.

Задача 1.1. Посчитайте в GHCi: 2^{123}.

Задача 1.2. Измените код каждой из функций в файле first_prog.hs, тестируя их в GHCi в процессе модификации, а затем скомпилируйте новую версию программы для генерации шаблонов электронных писем, указав `email` в качестве имени исполняемого файла.

Модуль 1

Основания функционального программирования

Известно два основных подхода к пониманию сущности программирования. Первым, и исторически более распространённым, является представление о том, что программист даёт компьютеру последовательность инструкций, чтобы он исполнял их определённым образом. Эта модель программирования привязывает программиста к устройству отдельного инструмента для программирования, называющегося компьютером. При таком понимании программирования компьютер — это устройство, которое принимает ввод, обращается к памяти, посылает инструкции процессору и, наконец, доводит результат до пользователя. Эта модель компьютера называется архитектурой *фон Неймана* в честь известного математика и физика Джона фон Неймана.

Язык программирования, который лучше всего олицетворяет такой подход мышления о программах, — это язык C. Программа на C принимает данные из стандартного потока ввода, управляемого операционной системой, хранит и извлекает необходимые значения в физической памяти, которая зачастую должна управляться вручную, требует обработки указателей на определённый блок памяти и, наконец, возвращает результат через стандартный поток вывода, также управляемый операционной системой.

Но компьютер, построенный по архитектуре фон Неймана, не единственный способ проводить вычисления. Люди проводят широкий спектр вычислений, которые не имеют ничего общего с размышлениями о распределении памяти и наборах инструкций: сортировка книг на полке, нахождение производной функции в математическом анализе, указания друзьям и так далее. Когда мы пишем код на C, мы программируем с использованием конкретной реализации идеи вычисления. Джон Бэкус, который руководил командой, создавшей Fortran, спросил в своей лекции после вручения премии Тьюринга: «Можно ли освободить программирование от стиля фон Неймана?»

Этот вопрос ведёт к второму подходу к пониманию программирования, который является предметом данной книги. *Функциональное программирование* пытается освободить программирование от стиля фон Неймана. Основания функционального программирования — это абстрактная, математическая идея вычисления, выходящая за пределы конкретной реализации. Это приводит к методу программирования, который зачастую решает задачи через их описание. Фокусируясь на вычислениях, а не компьютерах, функциональное программирование даёт программисту доступ к мощным абстракциям, которые упрощают решения многих проблем.

Цена этого упрощения в том, что начать может быть гораздо сложнее. Идеи функционального программирования зачастую абстрактны, и мы должны начать с формирования самой идеи программирования с первых принципов. Прежде чем мы сможем создавать полезные программы, нам придётся изучить много концепции. Работая с этим модулем, помните, что вы учитесь программировать способом, который выходит за пределы программирования собственно компьютера.

Так же, как C — это почти идеальное олицетворение стиля фон Неймана, Haskell — это чистейший функциональный язык, который вы можете изучить. Как язык Haskell полностью вверяется мечте Бэкуса и не позволяет вам вернуться к более знакомым стилям программирования. Это делает изучение Haskell более сложным, чем изучение многих языков, но при изучении Haskell невозможно не получить глубокое понимание функционального программирования. К концу этого модуля у вас будет достаточно сильная база в функциональном программировании, чтобы понимать основы всех других функциональных языков, вы также сможете подготовиться к путешествию в изучение Haskell.

2

Функции и функциональное программирование

После прочтения урока 2 вы:

- поймёте основную идею функционального программирования;
- научитесь определять простые функции на Haskell;
- сможете объявлять переменные на Haskell;
- узнаете о преимуществах функционального программирования.

Первое, что вам необходимо понять при изучении Haskell, — что такое функциональное программирование. У функционального программирования есть репутация сложной для освоения темы. И хотя это, несомненно, правда, основы функционального программирования удивительно просты. Первое, что вам нужно изучить, — что значит *функция* в функциональном языке программирования. У вас наверняка есть хорошее представление о том, что значит использовать функцию. В этом уроке мы обсудим простые правила, которым должны подчиняться функции в Haskell и которые не только сделают ваш код более простым для рассуждений, но и научат размышлять о написании программ абсолютно по-новому.

Обратите внимание. Вы и ваши друзья хотите заказать пиццу. В меню есть пицца трёх размеров по трём разным ценам:

(1) 45 сантиметров за 600 рублей;
(2) 40 сантиметров за 500 рублей;
(3) 35 сантиметров за 450 рублей.

Какой вариант даст вам больше пиццы за рубль? Как написать функцию, вычисляющую цену пиццы за квадратный сантиметр?

2.1. Функции

Что такое функция? Это важный вопрос, в нём необходимо разобраться, если вы собираетесь изучать функциональное программирование. Поведение функций в Haskell пришло напрямую из математики. В математике мы часто пишем вещи вроде $f(x) = y$, имея в виду, что есть функция f, принимающая один аргумент x и отображающая его на значение y. В математике любой x отображается на единственное y. Если $f(2) = 2\,000\,000$ для заданной функции f, то никогда не будет такого, что $f(2) = 2\,000\,001$.

Вдумчивые читатели могут спросить: «А что насчёт функции квадратного корня? Число 4 имеет два корня, 2 и -2, как тогда `sqrt x` может быть настоящей функцией, если она чётко указывает на два `y`?» Главное, что необходимо понять, — это то, что `x` и `y` необязательно должны быть одним и тем же. Мы можем сказать, что `sqrt x` — положительный корень, тогда `x` и `y` — положительные вещественные числа, что решает проблему. Но мы можем также считать `sqrt x` функцией из положительного вещественного числа в пары вещественных чисел. В таком случае `x` отображается точно на одну пару.

Функции в Haskell работают аналогично математическим. На рис. 2.1 показана функция с именем `simple`.

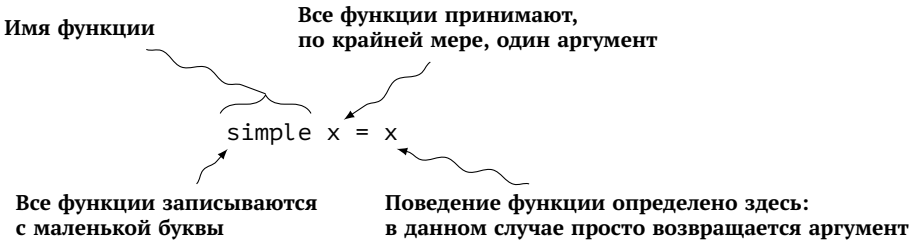


Рис. 2.1: Определение функции `simple`

Функция `simple` принимает один аргумент `x` и возвращает его нетронутым. Заметьте, что, в отличие от многих других языков программирования, в Haskell вам не нужно уточнять, что вы возвращаете значение. В Haskell функции обязаны возвращать значение, так что никогда не нужно указывать это явно. Вы можете загрузить свою простую функцию в GHCi и посмотреть на её поведение. Всё, что вам нужно сделать, чтобы загрузить функцию, — это сохранить её в файл и ввести команду `:load <имя файла>` в GHCi:

```
GHCi> simple 2
2
GHCi> simple "собака"
"собака"
```

> **Примечание.** В этом разделе мы используем GHCi в режиме цикла *чтение-вычисление-печать* (Read-Eval-Print Loop, или REPL) для запуска команд и просмотра результата их выполнения.

Все функции в Haskell следуют трём правилам, которые заставляют их вести себя как функции в математике:

- все функции должны принимать аргумент;
- все функции должны возвращать значение;
- функция возвращает одно и то же значение для одного аргумента.

Третье правило — это часть математического определения функции. Когда правило, которое гласит, что один и тот же аргумент функции должен давать одно и то же значение, применяется к функции в языке программирования, это называется *ссылочной прозрачностью*.

2.2. Функциональное программирование

Если функции — это просто отображения из множества иксов в множество игреков, то какое отношение они имеют к программированию? В 1930-х математик по имени Алонсо Чёрч попытался создать логическую систему, которая содержала бы только функции и переменные (иксы и игреки). Эта система называется *лямбда-исчислением*. В лямбда-исчислении вы выражаете всё через функции: ложь и истина — это функции, и даже все целые числа могут быть представлены как функции.

Изначально целью Чёрча было решение некоторых проблем в математической теории множеств. К сожалению, лямбда-исчисление не решило этих проблем, но из работы Чёрча вышло нечто более интересное. Выяснилось, что лямбда-исчисление является универсальной моделью вычислений, эквивалентной машине Тьюринга!

Такое соотношение между лямбда-исчислением и вычислением называется *тезисом Чёрча—Тьюринга*. Замечательным результатом его открытия является появление математической модели программирования!

Большинство используемых языков программирования представляют собой удивительные произведения инженерии, однако дают мало гарантий относительно поведения программ. Будучи основанным на математике, Haskell способен исключить из вашего кода целые классы ошибок. Пере-

довые исследования в области языков программирования экспериментируют со способами математически доказать, что программа будет делать то, что вы ожидаете. К тому же нематематическая природа большинства языков программирования означает, что абстракции, которые вы можете использовать, ограничены инженерными решениями. Если бы вы могли запрограммировать математику, то могли бы как доказывать утверждения касательно вашего кода, так и иметь доступ к практически безграничным абстракциям математики. В этом и состоит цель функционального программирования: дать программисту всю мощь математики.

Что такое машина Тьюринга?

Машина Тьюринга — это абстрактная модель компьютера, разработанная известным учёным в области информатики Аланом Тьюрингом. С теоретической точки зрения машина Тьюринга полезна, потому что она позволяет рассуждать о том, что может или не может быть вычислено не только на цифровом компьютере, но на любом компьютере вообще. Эта модель также позволяет доказывать эквивалентность вычислительных систем, если каждая из них может имитировать машину Тьюринга. Вы можете использовать её, например, чтобы показать, что нет ничего такого, вычисляемого с помощью Java, что нельзя было бы вычислить и на языке ассемблера.

2.3. Функциональное программирование на практике

Математическая модель программирования имеет множество различных смыслов. Благодаря простым правилам, что функции всегда должны принимать и возвращать значения и всегда должны возвращать одно и то же значение для одного и того же аргумента, Haskell является безопасным языком программирования. Программы безопасны тогда, когда они ведут себя так, как вы ожидаете, и вы легко можете рассуждать об их поведении. Безопасный язык программирования — тот, который заставляет ваши программы вести себя так, как ожидается.

Давайте посмотрим на код, который не является безопасным и нарушает наши простые правила для функций. Предположим, что вы читаете новый код и встречаете строки, которые выглядят следующим образом.

Листинг 2.1 Скрытое состояние в вызовах функции

```
tick()
if(timeToReset){
  reset()
}
```

Этот код явно написан не на Haskell, потому что и `tick`, и `reset` нарушают правила, которые мы установили. Ни одна из функций не принимает аргументов, ни одна из них не возвращает значения. Тогда возникает вопрос: что делают эти функции и чем они отличаются от функций в Haskell? Несложно предположить, что `tick` увеличивает счётчик, а `reset` возвращает счётчик к начальному значению. Даже если мы не совсем правы, эти рассуждения позволяют проиллюстрировать нашу проблему. Если вы не передаёте в функцию аргумент, то вы должны иметь доступ к значению в своём окружении, и если вы не возвращаете значение, вы должны изменять значение в своём окружении. Когда вы изменяете значение в программном окружении, вы меняете состояние программы. Изменение состояния создаёт побочные эффекты, из-за которых ваш код может стать сложным для анализа и небезопасным.

Похоже, что и `tick`, и `reset` обращаются к глобальной переменной (переменной, доступной из любого места программы), что считается плохим решением в любом языке программирования. Но побочные эффекты затрудняют анализ даже самого простого и хорошо написанного кода. Чтобы это увидеть, давайте посмотрим на коллекцию значений `myList` и обратим её содержимое с помощью встроенного функционала. Следующий код работает в Python, Ruby и JavaScript. Понятно ли вам, что там происходит?

Листинг 2.2 Сбивающее с толку поведение стандартных библиотек

```
myList = [1,2,3]
myList.reverse()
newList = myList.Reverse()
```

Какое значение вы ожидаете увидеть в `newList`? Поскольку это работающая программа в Ruby, Python и JavaScript, кажется разумным, что и значение будет одинаковым. Однако вот ответы для всех трёх языков:

```
Ruby -> [3,2,1]
Python -> None
JavaScript -> [1,2,3]
```

Три абсолютно разных ответа для одного кода в разных языках! Функция `reverse` в языках Python и JavaScript имеет побочные эффекты. Так как эти

побочные эффекты отличаются и явно программисту не видны, мы получаем разные ответы. Код в Ruby ведёт себя как код в Haskell, без побочных эффектов. На этом примере вы можете увидеть важность ссылочной прозрачности. В Haskell вы всегда можете увидеть, какие эффекты есть у функций. Когда вы ранее вызывали `reset` и `tick`, изменения, которые они делали, были для вас незаметны. Не посмотрев на исходный код, вы не можете точно узнать, какие и даже как они используют множества других значений. Haskell не позволяет функциям иметь побочные эффекты, что объясняет, почему все функции в Haskell должны принимать аргумент и возвращать значение. Если бы функции в Haskell не всегда возвращали значение, они бы должны были менять скрытое состояние в программе, иначе они были бы бесполезны. Если бы они не принимали аргумента, то они имели бы доступ к чему-то скрытому, что означало бы, что они более не прозрачны.

Небольшое свойство функций в Haskell приводит к коду, который намного более предсказуем. Даже в Ruby программист может использовать побочные эффекты. Пользуясь кодом другого программиста, вы всё ещё не можете предполагать, что происходит, когда вы вызываете функцию или метод. Так как Haskell не позволяет этого, вы можете смотреть на любой код, написанный любым программистом, и рассуждать о его поведении.

Проверка 2.1. Во многих языках используется операция ++, увеличивающая значение переменной на единицу, например `x++`. Как вы думаете, есть ли в Haskell операция или функция, которые так работают?

2.3.1. Переменные

Переменные в Haskell просты. Здесь вы присваиваете значение 2 переменной `x`:

Листинг 2.3 Определение вашей первой переменной

```
x=2
```

Ответ 2.1. Операция ++ используется в языках вроде C++ и не может существовать в Haskell, так как нарушает правила для функций. В частности, результат при каждом вызове `x++` отличается.

Единственная хитрость с переменными в Haskell в том, что на самом деле они вовсе не переменные! Если бы вы попробовали скомпилировать код из следующего листинга в Haskell, то получили бы ошибку.

Листинг 2.4 Переменные не переменны!

```
x=2
x=3
```

Не скомпилируется, так как меняет значение x

Правильнее думать о переменных в Haskell как об объявлениях. Опять-таки, математическое мышление заменяет привычный способ размышления о коде. Проблема в том, что в большинстве языков программирования переприсваивание переменных естественно для решения многих задач. Невозможность изменять переменные также приводит к большей ясности кода. Это правило может казаться строгим, но в награду после каждого вызова функции мир остаётся прежним.

Проверка 2.2. Даже языки, в которых нет операции ++, имеют операцию +=, также используемую для увеличения значения. Например, `x+=2` увеличивает значение `x` на `2`. Вы можете воспринимать += как функцию, которая следует нашим правилам: она принимает значение и возвращает значение. Значит ли это, что в Haskell возможно +=?

Основная польза переменных в программировании в том, что они делают код более понятным и позволяют избегать повторений. Например, представьте, что у вас есть функция для вычисления размера сдачи под названием `calcChange`. Эта функция принимает два аргумента: сколько нужно заплатить и сколько денег получено. Если вам дали достаточно денег, вы возвращаете сдачу. Но если вам дали недостаточно и вы не хотите отдавать отрицательные доллары, то возвращаете 0. Вот один из способов это записать.

Листинг 2.5 Функция calcChange, версия 1

```
calcChange owed given = if given - owed > 0
                        then given - owed
                        else 0
```

Ответ 2.2. Хотя операция += принимает и возвращает значение, как и в случае с ++, при каждом вызове += вы получаете другой результат.

С этой функцией есть как минимум две проблемы:

- несмотря на столь малый размер, её тяжело читать. Каждый раз, когда вы видите выражение `given - owed`, вам нужно думать о том, что происходит. Для чего-либо более сложного, чем вычитание, это было бы неприятно;
- здесь повторяются вычисления! Вычитание — это дешёвая операция, но если бы здесь была более тяжёлая операция, вы бы зря потратили ресурсы.

Haskell решает данные проблемы с помощью блока `where`. Вот предыдущая функция, записанная с его помощью.

Листинг 2.6 Функция calcChange, версия 2

```
calcChange owed given = if change > 0
                        then change
                        else 0
  where change = given - owed
```

given - owed вычисляется единожды и присваивается переменной change

Первое, что должно вас заинтересовать, — это то, как блок `where` обращает привычный порядок записи переменных. В большинстве языков программирования переменные объявляются перед их использованием. Такое соглашение в большинстве языков программирования является отчасти побочным продуктом возможности изменять состояние. Порядок переменных имеет значение, потому что вы всегда можете переприсвоить переменной значение, после того как вы его присвоили в первый раз. В Haskell благодаря ссылочной прозрачности так не получится. Подход Haskell также повышает читабельность: если вы читаете алгоритм, намерения ясны сразу.

Проверка 2.3. Дополните недостающую часть кода в блоке `where`:

```
doublePlusTwo x = doubleX + 2
   where doubleX = __________
```

Ответ 2.3

```
doublePlusTwo x = doubleX + 2
   where doubleX = x*2
```

2.3.2. Переменные, которые всё-таки переменны

Так как перемены — это неизбежная часть жизни, иногда имеет смысл иметь переменные, значение которых можно переприсвоить. Один из этих случаев возникает при работе в GHCi. Вот пример:

```
GHCi> x = 7
GHCi> x
7
GHCi> x = [1,2,3]
GHCi> x
[1,2,3]
```

До 8-й версии компилятора GHC переменные в GHCi должны были предваряться ключевым словом `let`, чтобы обозначить, что они отличны от остальных переменных Haskell. Если хотите, вы всё ещё можете объявлять переменные в GHCi, используя `let`:

```
GHCi> let x = 7
GHCi> x
7
```

Также стоит заметить, что таким же образом можно определять однострочные функции:

```
GHCi> let f x = x^2
GHCi> f 8
64
```

Используемое таким образом выражение `let` вы увидите и в некоторых других контекстах. Это может сбивать с толку, но удобно в первую очередь для того, чтобы немного упростить решение задач реального мира.

Проверка 2.4. Каково итоговое значение переменной `x` в данном коде?

```
GHCi> let x = simple simple
GHCi> let x = 6
```

Ответ 2.4. Так как вы можете переприсваивать переменные, итоговым значением переменной `x` будет 6.

Итоги

Целью этого урока было знакомство с функциональным программированием и написанием на Haskell простейших функций. Вы увидели, что функциональное программирование ставит ограничения на поведение функций. Эти ограничения таковы:

- функция должна принимать аргумент;
- функция должна возвращать значение;
- вызов одной и той же функции с одним и тем же аргументом всегда должен возвращать одно и то же значение.

Эти три правила существенно влияют на то, как вы пишете программы на Haskell. Главное преимущество написания кода в таком стиле — в том, что о ваших программах проще рассуждать, и ведут они себя более предсказуемо. Проверим, разобрались ли вы с этим.

Задача 2.1. Для написания функции `calcChange` мы использовали конструкцию `if then else`. В Haskell все `if`-выражения должны содержать компонент `else`. Почему, согласно нашим трём правилам для функций, `if` нельзя использовать само по себе?

Задача 2.2. Напишите функции с названиями `inc`, `double` и `square`, которые увеличивают, удваивают и возводят в квадрат аргумент `n` соответственно.

Задача 2.3. Напишите функцию, которая принимает значение n. Если n чётное, то функция возвращает $n-2$, а если нечётное, то $3\times n+1$. Чтобы проверить, чётно ли число, вы можете использовать функции `even` (проверка на чётность) или `mod` (взятие остатка от деление).

3

Лямбда-функции и лексическая область видимости

После прочтения урока 3 вы:

- сможете определять лямбда-функции в Haskell;
- научитесь использовать лямбда-функции для определения функций там, где это необходимо;
- разберётесь с устройством областей видимости;
- сможете управлять областями видимости, создавая лямбда-функции.

В этом уроке вы продолжите свой путь по изучению функционального программирования в целом и Haskell в частности, а именно: изучите одну из самых базовых концепций функционального программирования — лямбда-функции. На первый взгляд, идея *лямбда-функции* — функции, не имеющей имени, — кажется очень простой и не слишком интересной. Однако лямбда-функции оказываются очень мощным инструментом как в теории, так и на практике.

Обратите внимание. Допустим, вы экспериментируете с GHCi и хотите быстро вычислить разность между квадратом суммы и суммой квадратов трёх чисел 4, 10 и 22. Для решения этой задачи вы можете написать следующий код:

```
GHCi> (4 + 10 + 22)^2 - (4^2 + 10^2 + 22^2)
```

Во время написания такого выражения легко допустить опечатку, которая приведёт к ошибке при выполнении команды. Также возникают некоторые неудобства при попытке с помощью истории команд GHCi (посредством нажатий на стрелку вверх на клавиатуре можно переходить к предыдущим выполненным) изменить числа, используемые в выражении. Существует ли способ улучшить этот код, не прибегая к явному объявлению новой функции?

3.1. Лямбда-функции

Одна из наиболее важных идей функционального программирования — функция без имени, называемая *лямбда-функцией* (отсюда происходит название *лямбда-исчисление*). Лямбда-функции часто обозначаются с помощью маленькой греческой буквы λ. Другое распространённое название лямбда-функции — *анонимная функция*. Вы можете воспользоваться лямбда-функцией для переопределения функции `simple` из урока 2, только на этот раз — без имени функции. Чтобы сделать это, нужно воспользоваться синтаксисом Haskell для лямбда-функций. Этот синтаксис показан на рис. 3.1.

Рис. 3.1: Функция `simple`, переписанная как лямбда-функция

Лямбда-функция представляет собой функцию, содержащую только самый необходимый минимум: она принимает значение и возвращает значение — вот и всё. Нельзя просто вставить такую функцию в GHCi или в код программы, написанной на языке Haskell, потому что она представляет собой всего лишь выражение и сама по себе ничего не делает. Чтобы это исправить, нам достаточно передать ей некоторый аргумент:

```
GHCi> (\x -> x) 4
4
GHCi> (\x -> x) "привет"
привет
GHCi> (\x -> x) [1,2,3]
[1,2,3]
```

Обратите внимание: каждый раз при использовании лямбда-функции нужно писать её определение. Это вполне логично — у нас ведь нет имени для её вызова! Лямбда-функции полезны, но устроены так, что могут существовать только весьма непродолжительное время. В общем случае, если можно обойтись функцией, имеющей имя, то лучше так и сделать.

Проверка 3.1. Напишите лямбда-функцию, возвращающую результат умножения своего аргумента на два, и проверьте результат её работы с несколькими числами в качестве аргументов.

3.2. Пишем свой аналог блока where

У функционального программирования есть интересная особенность: при желании можно написать с нуля практически всё, что может прийти вам в голову. Таким образом, набравшись опыта в функциональном программировании, вы в большинстве случаев будете иметь глубокое понимание того, как работает та или иная программа. Чтобы продемонстрировать

Ответ 3.1

```
GHCi> (\x -> x*2) 2
4
GHCi> (\x -> x*2) 4
8
```

мощь лямбда-функций, проведём эксперимент: забудем про встроенный в Haskell блок where и посмотрим, удастся ли написать его самостоятельно, с нуля. Давайте немного поразмышляем об этом. На данный момент `where` — единственный известный вам способ хранения переменных внутри функции.

Оказывается, лямбда-функции сами по себе являются чрезвычайно мощным средством для создания переменных на пустом месте. Для начала давайте рассмотрим функцию, использующую блок `where`. Она принимает два числа и возвращает наибольшее из суммы их квадратов и квадрата их суммы. Ниже приведена реализация с использованием where.

Листинг 3.1 Функция sumSquareOrSquareSum, версия 1

```
sumSquareOrSquareSum x y = if sumSquare > squareSum
                           then sumSquare
                           else squareSum
  where sumSquare = x^2 + y^2
        squareSum = (x+y)^2
```

В функции `sumSquareOrSquareSum` мы используем `where` в двух целях: чтобы повысить читаемость кода и сократить количество вычислений (хотя на самом деле во многих случаях Haskell устранит повторяющиеся вызовы функций даже без введения переменных). Можно обойтись без `where` и переменных, но это приведёт к повторным вычислениям одних и тех же значений, а также сделает код менее читаемым, что можно увидеть ниже:

```
sumSquareOrSquareSum x y = if (x^2 + y^2) > ((x+y)^2)
                           then (x^2 + y^2)
                           else (x+y)^2
```

Наша функция относительно проста, но без `where` и переменных она выглядит ужасно! Одно из возможных решений этой проблемы состоит в разбиении функции на две части: вспомогательная функция под названием `body` будет выполнять основную работу исходной функции по сравнению значений, а новая реализация `sumSquareOrSquareSum` будет вычислять значения `sumSquare` и `squareSum`, а затем передавать их в качестве аргументов функции `body`. Ниже представлен код функции `body`:

```
body sumSquare squareSum = if sumSquare > squareSum
                           then sumSquare
                           else squareSum
```

Теперь функции sumSquareOrSquareSum остаётся вычислить два значения sumSquare и squareSum и передать их в body:

```
sumSquareOrSquareSum x y = body (x^2 + y^2) ((x+y)^2)
```

Это решает проблему, но добавляет некоторые трудности — например, нам пришлось определять новую, «промежуточную» функцию body. Эта функция настолько проста, что было бы хорошо иметь возможность не тратить время на её отдельное определение. Нам нужно каким-то образом избавиться от именованной функции body, а это — превосходная задача для лямбда-функции! Для начала давайте взглянем на лямбда-функцию, соответствующую body:

```
body = (\sumSquare squareSum ->
          if sumSquare > squareSum
          then sumSquare
          else squareSum)
```

Теперь, если подставить эту лямбда-функцию в приведённое выше определение sumSquareOrSquareSum, мы получим выражение, приведённое на рис. 3.2.

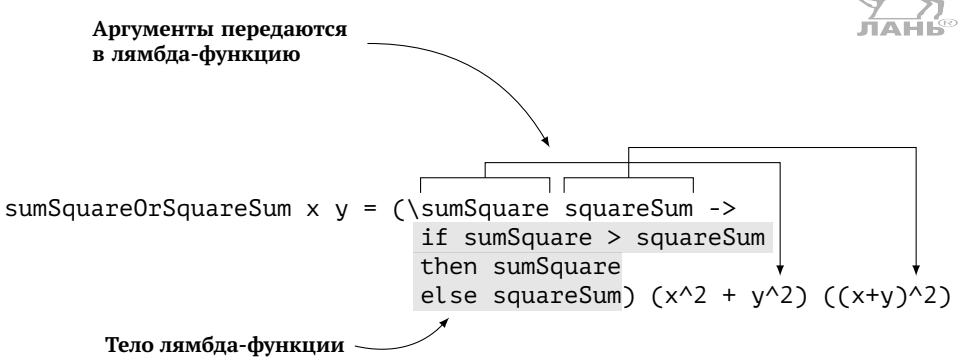


Рис. 3.2: Реализация sumSquareOrSquareSum через лямбда-функции

Эта реализация всё ещё не так хороша, как реализация с использованием блока where (в первую очередь поэтому блок where и существует в Haskell), но всё же гораздо лучше, чем предыдущая. Что более важно: мы реализовали идею переменных практически с нуля!

Проверка 3.2. Перепишите реализацию следующей функции, использовав лямбда-функцию вместо `where`:

```
doubleDouble x = dubs*2
  where dubs = x*2
```

3.3. От лямбда-функций к let: изменяемые переменные!

Лямбда-функции хуже читаются, нежели блоки `where`, но зато они — более мощное средство языка! Блок `where` делает код более лёгким для понимания, но синтаксически он неотделим от функции, внутри которой используется. Нельзя просто взять и вытащить блок `where` из функции. При использовании лямбда-выражений таких ограничений не возникает. В предыдущем разделе мы просто вставили лямбда-функцию в нужное место, а при необходимости могли бы с такой же лёгкостью её оттуда вырезать и поместить в другое место. Лямбда-функция представляет собой *выражение* — фрагмент кода, который может существовать сам по себе.

В Haskell имеется альтернатива блокам `where` — выражение `let`. Выражение `let` позволяет совмещать читаемость кода, обеспечиваемую блоками `where`, и мощь лямбда-функций. На рис. 3.3 приведён пример реализации функции `sumSquareOrSquareSum` с использованием `let`.

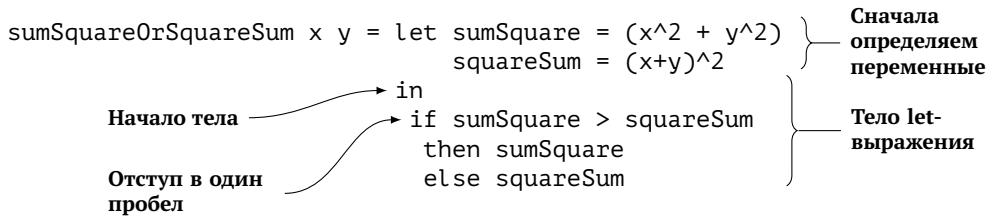


Рис. 3.3: Реализация `sumSquareOrSquareSum` с помощью `let`

Выбор между `where` и `let` в Haskell является, как правило, делом вкуса.

Ответ 3.2

```
doubleDouble x = (\dubs -> dubs*2) (x*2)
```

К данному моменту читателю должно быть ясно, что лямбда-функции сами по себе являются очень мощным средством языка. Чтобы закрепить эту информацию, давайте попробуем сделать кое-что такое, что Haskell делать обычно не позволяет: перезапишем значения переменных! В следующем примере для повышения читаемости кода вместо лямбда-функций будут использованы выражения `let`. В функциональном программировании, как правило, нет смысла специально перезаписывать значения переменных, но чтобы показать, что это (в некотором смысле) возможно, на следующем листинге представлена функция `overwrite`, которая принимает переменную `x` и несколько раз перезаписывает её значение.

Листинг 3.2 Функция overwrite

```
overwrite x = let x = 2
              in let x = 3
               in let x = 4
                in x
```

Код этой (довольно бесполезной) функции напоминает переопределение переменных в `GHCi`:

```
GHCi> let x = 2
GHCi> x
2
GHCi> let x = 3
GHCi> x
3
```

Функция `overwrite` проливает свет на то, каким образом в `GHCi` обеспечивается возможность переопределять переменные, избегая при этом нарушения правил функционального программирования.

Проверка 3.3. Перепишите реализацию функции `overwrite`, используя только лямбда-выражения.

Ответ 3.3

```
overwrite x = (\x ->
                (\x ->
                  (\x -> x) 4) 3) 2
```

Вот и всё. Теперь вы, при желании, можете с помощью неименованных функций (они же — лямбда-функции) переопределять переменные, как и в любом другом языке программирования.

3.4. Лямбда-функции и области видимости на практике

Примеры использования лямбда-функций с `let` и `where` могут изначально показаться слишком научными или надуманными, однако они составляют основу одного из самых важных архитектурных шаблонов языка JavaScript. JavaScript отлично поддерживает работу с лямбда-функциями; функция, эквивалентная `\x -> x`, в JavaScript выглядит следующим образом:

```
function(x){
    return x;
}
```

Изначально JavaScript предназначался только для того, чтобы добавить веб-сайтам немного дополнительной функциональности. Поэтому неудачные архитектурные решения, принятые разработчиками этого языка, привели к трудностям с поддержкой объёмных и сложных участков кода. Один из самых серьёзных недостатков JavaScript состоит в отсутствии поддержки пространств имён и модулей. Так, при создании своей функции `length` разработчику приходится надеяться на то, что он случайно не переопределит функцию `length`, реализация которой находится в одной из множества используемых в проекте библиотек. Помимо этого, JavaScript позволяет очень легко случайно определять глобальные переменные. Чтобы это продемонстрировать, рассмотрим функцию под названием `libraryAdd`, представив, что она находится в одной из используемых вами сторонних библиотек.

```
var libraryAdd = function(a, b){
    c = a + b;
    return c;
}
```

Ой! Разработчик забыл про ключевое слово `var`, случайно создав глобальную переменную!

В этой простой функции есть большая проблема: переменная `c` была случайно объявлена глобальной! Насколько это опасно? Ниже приведён пример проблемы, к которой может привести такая ошибка:

```
var a = 2;
var b = 3;
var c = a + b;
var d = libraryAdd(10, 20);
console.log(c);
```

Внутри функции `libraryAdd` происходит изменение глобальной переменной `c`, но это совершенно не очевидно

Будет выведено число 30 вместо 5!

Несмотря на то что в своём коде вы всё сделали правильно, после вызова функции `libraryAdd` переменная `c` имеет значение 30! Это происходит из-за того, что в JavaScript нет пространств имён, поэтому, когда функция `libraryAdd` выполняет присваивание значения переменной `c`, она выполняет поиск этой переменной или создаёт новую глобальную переменную, если найти её не удалось. Не разобравшись глубоко в чужом коде на JavaScript, вы никогда не сможете обнаружить эту ошибку!

Для решения данной проблемы разработчики на JavaScript используют лямбда-функции. Поместив свой код в лямбда-функцию и немедленно её вызвав, можно обеспечить безопасность этого кода. Этот шаблон написания кода называется *немедленно вызываемые функции* (англ. *immediately invoked function expression*, IIFE). С использованием IIFE код будет выглядеть следующим образом:

```
(function(){
 var a = 2;
 var b = 3;
 var c = a + b;
 var d = libraryAdd(10, 20);
 console.log(c);
})()
```

Определение лямбда-функции

Теперь вызов `libraryAdd` не опасен

Выводится правильное значение 5

Хорошо, что у проблемы нашлось решение! IIFE работает по тому же принципу, что и наш пример с заменой `where` на лямбда-функцию. Каждый раз при создании функции, именованной или неименованной, создаётся новая область видимости, представляющая собой контекст, в котором определены переменные. При использовании переменной программа выполняет её поиск в ближайшей области видимости; если нужной переменной там не оказалось, поиск выполняется в области видимости, внешней по отношению к той, в которой только что производился поиск. Такой способ поиска переменной называется *лексической областью видимости*. В Haskell, как и в Javascript, используется лексическая область видимости, поэтому IIFE и наши лямбда-функции работают схожим образом. На рис. 3.4 представлен пример определения переменной и трёх функций, использующих лексическую область видимости для преобразования своих аргументов.

Рис. 3.4: Использование области видимости в функциях `add1`, `add2`, `add3`

Вызывая функции `add1`, `add2`, `add3` с одним и тем же аргументом, можно увидеть, насколько различными получаются результаты их работы:

```
GHCi> add1 1
5
GHCi> add2 1
4
GHCi> add3 1
3
```

В 5-м уроке вы узнаете, что возможность использования неименованных функций для создания областей видимости на лету позволяет делать гораздо более серьёзные вещи, чем описанные в этом разделе.

Итоги

Целью этого урока было научить вас лямбда-функциям. Идея лямбда-функции проста: это функция, не имеющая имени. Однако эта идея является фундаментальной для функционального программирования. Помимо того что лямбда-функции являются краеугольным камнем теории функционального программирования, они очень полезны на практике. Самое очевидное преимущество лямбда-функций состоит в том, что они позволяют создавать функции на лету. Ещё более мощным их свойством является возможность при необходимости создавать области видимости. Давайте удостоверимся, что вы всё поняли.

Задача 3.1. Попрактикуйтесь в написании лямбда-функций, переписав каждую функцию из урока 3 как лямбда-выражение.

Задача 3.2. Использование выражений `let` и лямбда-функций — технически не совсем одно и то же. Например, попытка запуска следующего кода приведёт к ошибке:

```
counter x = let x = x + 1
            in
             let x = x + 1
             in
              x
```

Чтобы доказать, что `let` и лямбда-функции — не одно и то же, перепишите функцию `counter` точно как в примере выше, но с использованием вложенных лямбда-функций вместо `let`. (*Подсказка*: Начните с конца.)

4

Функции как значения первого класса

После прочтения урока 4 вы:

- поймёте идею функций как значений первого класса;
- будете использовать функции в качестве аргументов других функций;
- сможете выносить вычисление из функции;
- научитесь возвращать функцию как значение.

Несмотря на то что функциональное программирование долгое время имело репутацию излишней академичности, всё больше элементов функционального стиля программирования появляются во многих популярных языках программирования. Наиболее распространённый из таких элементов — это функции как *значения первого класса*, то есть функции, которые могут передаваться в качестве параметров аналогично любым другим значениям. Ещё десяток лет назад эта идея шокировала бы многих программистов, но сегодня большинство языков программирования поддерживают и широко её используют. Если вы когда-нибудь устанавливали обработчик события в JavaScript или передавали пользовательскую логику сортировки в метод `sort` в языке вроде Python, то вы уже использовали функции как значения первого класса.

Обратите внимание. Допустим, вы хотите создать сайт, который позволяет сравнивать цены различных товаров на других сайтах вроде Amazon и eBay. У вас уже есть функция, которая возвращает URL необходимого товара, но вам надо написать код для каждого сайта, который будет определять, как извлечь цену с веб-страницы. Одно из решений может заключаться в написании отдельной функции для каждого сайта:

```
getAmazonPrice url

getEbayPrice url

getWalmartPrice url
```

Всё хорошо, но эти функции будут иметь слишком много общего в логике работы (например, преобразование цены в строковом виде $1,999.99 к числовому типу вроде 1999.99). Есть ли способ полностью отделить логику, извлекающую цену из HTML-страницы, и передавать её в общую функцию `getPrice`?

4.1. Функции как аргументы

Идея функций как значений первого класса состоит в том, что функции не отличаются от других данных, используемых в программе. Функции могут использоваться как аргументы и возвращаться как значения из других функций. Это на удивление мощное свойство языка программирования. Оно позволяет обобщить любые повторяющиеся вычисления в вашем коде вплоть до написания функций, которые пишут другие функции.

Допустим, у вас есть функция `ifEvenInc`, которая увеличивает число `n` на единицу, если оно чётное, оставляя его неизменным в противном случае, как показывает следующий листинг:

Листинг 4.1 Функция ifEvenInc

```
ifEvenInc n = if even n
              then n+1
              else n
```

Позже вы узнаёте, что вам нужны ещё две функции, `ifEvenDouble` и `ifEvenSquare`, которые удваивают и возводят значение в квадрат соответственно, как показано далее. Это простые для написания функции, если

вы знаете, как написать `ifEvenInc`.

Листинг 4.2 Функции ifEvenDouble и ifEvenSquare

```
ifEvenDouble n = if even n
                 then n*2
                 else n

ifEvenSquare n = if even n
                 then n^2
                 else n
```

Хотя эти функции нетрудно написать, все они практически идентичны. Единственное отличие — это поведение в прибавлении единицы, удваивании или возведении в квадрат. То, что мы здесь обнаружили, называется общим вычислительным шаблоном, который можно обобщить. Главное, что необходимо для реализации желаемого поведения, — это возможность передать функцию как аргумент.

Давайте покажем это на примере функции `ifEven`, которая принимает функцию и число как аргументы. Если это число чётное, то `ifEven` применяет функцию к этому числу.

Листинг 4.3 Функция ifEven

```
ifEven myFunction x = if even x
                      then myFunction x
                      else x
```

Мы также можем вынести увеличение на единицу, удвоение и возведение в квадрат в три отдельные функции:

```
inc n = n + 1

double n = n * 2

square n = n^2
```

Теперь давайте посмотрим, как пересоздать предыдущие определения, используя функции как значения первого класса:

```
ifEvenInc n = ifEven inc n

ifEvenDouble n = ifEven double n

ifEvenSquare n = ifEven square n
```

Теперь вы легко сможете добавлять новые функции вроде `ifEvenCube` или `ifEvenNegate`.

Функции и приоритет операций

В этом уроке вы уже видели примеры функций и операций. Например, `inc` — это функция, а `+` — операция. При написании кода на Haskell необходимо знать, что вызовы функций всегда вычисляются перед операциями. Что это значит? Рассмотрим пример в GHCi:

```
GHCi> 1 + 2 * 3
7
```

Как и в большинстве языков программирования, * обладает большим приоритетом, чем +, таким образом, умножение 2 и 3 и прибавление единицы даёт 7. Теперь посмотрим, что произойдёт, если заменить `1+` на `inc`:

```
GHCi> inc 2 * 3
9
```

Результат отличается, потому что функции всегда имеют приоритет над операциями. Это значит, что сначала вычисляется `inc 2`, а затем результат умножается на 3. Это верно и для функций нескольких аргументов:

```
GHCi> add x y = x + y
GHCi> add 1 2 * 3
9
```

Основное преимущество соответствующего правила в том, что вы можете избежать использования большого количества ненужных скобок.

4.1.1. Лямбда-функции как аргументы

Именование функций — обычно хорошая идея, но вы также можете использовать лямбда-функции, чтобы быстро добавить код для передачи в функцию. Если вы хотите удвоить значение, то можете быстро набросать лямбда-функцию:

```
GHCi> ifEven (\x -> x*2) 6
12
```

Несмотря на то что именованные функции более предпочтительны, зачастую достаточно простой функциональности.

Проверка 4.1. Напишите лямбда-функцию для возведения `x` в куб и передайте её в `ifEven`.

4.1.2. Пример — пользовательская сортировка

Передачу функций в другие функции можно использовать на практике, например для сортировки. Допустим, у вас есть список имён и фамилий. В этом примере каждое имя представлено как кортеж. *Кортеж* — это тип наподобие списка, но он может содержать значения различных типов и имеет фиксированный размер. Вот пример имени в кортеже:

```
author = ("Уилл","Курт")
```

Есть две полезные функции для работы с кортежами из двух элементов (или *парами*), `fst` и `snd`, которые возвращают первый и второй элементы кортежа соответственно:

```
GHCi> fst author
"Уилл"
GHCi> snd author
"Курт"
```

Теперь предположим, что у вас есть список имён, который вы хотите отсортировать. Вот набор имён, представленный в виде списка кортежей.

Листинг 4.4 Имена

```
names = [("Иэн","Кертис"),
         ("Бернард","Самнер"),
         ("Питер","Хук"),
         ("Стивен","Моррис")]
```

Вы хотите упорядочить имена. К счастью, в Haskell есть встроенная функция `sort`. Чтобы ею воспользоваться, необходимо импортировать модуль `Data.List`. Сделать это довольно просто, достаточно добавить вверху файла, с которым вы работаете, следующее объявление:

Ответ 4.1

```
GHCi> ifEven (\x -> x^3) 4
```

```
import Data.List
```

Вы также *можете импортировать модуль в GHCi*. Если загрузить файл со списком `names` и вашим импортом, то можно увидеть, что `sort` в Haskell каким-то образом угадал, как отсортировать эти кортежи:

```
GHCi> sort names
[("Бернард","Самнер"),("Иэн","Кертис"),("Питер","Хук"),
↳ ("Стивен","Моррис")]
```

Неплохо, учитывая, что Haskell понятия не имеет, что вы пытаетесь сделать! К сожалению, обычно хочется сортировать не по имени. Для решения этой проблемы можно использовать функцию `sortBy` из модуля `Data.List`. Вам нужно снабдить `sortBy` другой функцией, которая будет сравнивать имена в кортежах. После того как вы объясните, как сравнивать два элемента, остальное уже будет понятно. Для этого напишем функцию `compareLastNames`. Эта функция принимает два аргумента, `name1` и `name2`, и возвращает `GT`, `LT` или `EQ`, специальные значения, означающие *больше чем*, *меньше чем* и *равно*. Во многих языках программирования вы бы возвращали `True` или `False` или 1, -1 или 0.

Листинг 4.5 Функция compareLastNames

```
compareLastNames name1 name2 = if lastName1 > lastName2
                               then GT
                               else
                                 if lastName1 < lastName2
                                 then LT
                                 else EQ
  where lastName1 = snd name1
        lastName2 = snd name2
```

Теперь можно вернуться в GHCi и использовать `sortBy` с вашей собственной сортировкой:

```
GHCi> sortBy compareLastNames names
[("Иэн","Кертис"),("Стивен","Моррис"),("Бернард","Самнер"),
↳ ("Питер","Хук")]
```

Это намного лучше! Такие языки программирования, как JavaScript, Ruby и Python, поддерживают похожее использование функций как значений первого класса для задания критерия сортировки, так что эта техника, скорее всего, хорошо знакома многим программистам.

Проверка 4.2. В функции `compareLastNames` мы не обрабатывали случай, в котором две фамилии совпадают, а имена отличаются. Модифицируйте её таким образом, чтобы она сравнивала имена, и используйте новую реализацию для исправления сортировки списка имён.

4.2. Возвращаем функции

ЛАНЬ

Мы обсудили передачу функций в качестве аргументов, но это только половина идеи функций как значений первого класса. Функции также возвращают значения, значит, чтобы функции были по-настоящему значениями первого класса, иногда имеет смысл возвращать их из других функций в качестве результата. Как всегда, первый вопрос должен быть следующим: почему мне вообще захочется вернуть функцию? Одна хорошая причина состоит в том, что вы можете захотеть выбирать между некоторыми функциями в зависимости от переданных параметров.

Предположим, вы хотите организовать Секретное общество современных алхимиков и вам нужно рассылать письма членам по абонентским ящикам в разных региональных отделениях. Отделения есть в трёх городах: Сан-Франциско, Рино и Нью-Йорке. Вот их почтовые адреса:

ЛАНЬ

- а/я 1234, Сан-Франциско, штат Калифорния, 94111;
- а/я 789, Нью-Йорк, штат Нью-Йорк, 10013;
- а/я 456, Рино, штат Невада, 89523.

Ответ 4.2

```
compareLastNames name1 name2 =
   if lastName1 > lastName2
   then GT
   else if lastName1 < lastName2
        then LT
        else if firstName1 > firstName2
             then GT
             else if firstName1 < firstName2
                  then LT
                  else EQ
  where lastName1 = snd name1
        lastName2 = snd name2
        firstName1 = fst name1
        firstName2 = fst name2
```

Нужно реализовать функцию, которая принимала бы кортеж с именем члена общества (как в примере с сортировкой) и адресом отделения, и формировала бы полный почтовый адрес. Первый подход к реализации этой функции может выглядеть следующим образом. Единственное, что нам нужно представить и чего вы раньше не видели, — это операция ++, используемая для конкатенации строк (и списков).

Листинг 4.6 Функция addressLetter, версия 1

```
addressLetter name location = nameText ++ " - " ++ location
  where nameText = (fst name) ++ " " ++ (snd name)
```

Чтобы воспользоваться этой функцией, необходимо передать ей кортеж с именем и адрес:

```
GHCi> addressLetter ("Боб","Смит") "А/я 1234, Сан-Франциско,
                                    ↳ штат Калифорния, 94111"
"Боб Смит - А/я 1234, Сан-Франциско, штат Калифорния, 94111"
```

Это хорошее решение. Вы также можете легко использовать переменные для хранения адресов, и тогда менее вероятны ошибки (да и печатать меньше). Всё готово для рассылки писем!

Разослав первую партию писем, вы получили несколько жалоб и запросов от местных отделений:

- в Сан-Франциско добавили новый адрес для абонентов с фамилией на букву Л и далее по алфавиту, письма им теперь нужно отправлять на а/я 1010, Сан-Франциско, Калифорния, 94109;
- отделение в Нью-Йорке по загадочным причинам, которыми они почему-то не делятся, хочет, чтобы после имени стояло двоеточие;
- Рино требует, чтобы в адресах для большей секретности использовались только фамилии.

К сожалению, теперь ясно, что для каждого из отделений нужна отдельная функция.

Листинг 4.7 Функции sfOffice, nyOffice, renoOffice

```
sfOffice name =
    if lastName < "Л"
    then nameText ++
         " - А/я 1234, Сан-Франциско, штат Калифорния, 94111"
    else nameText ++
         " - А/я 1010, Сан-Франциско, штат Калифорния, 94109"
  where lastName = snd name
        nameText = (fst name) ++ " " ++ lastName
```

```
nyOffice = nameText ++
           ": А/я 789, Нью-Йорк, штат Нью-Йорк, 10013"
  where nameText = (fst name) ++ " " ++ (snd name)

renoOffice = nameText ++ " - А/я 456, Рино, штат Невада, 89523"
  where nameText = snd name
```

Теперь вопрос в том, как использовать эти функции с `addressLetter`. Например, можно переписать `addressLetter`, чтобы она принимала в качестве аргумента функцию вместо почтового адреса. Проблема в том, что функция `addressLetter` будет частью большого веб-приложения, и вы бы всё-таки предпочли передавать именно строковый параметр. Хочется иметь функцию, которая будет принимать строку с информацией о местоположении и возвращать необходимую для формирования полного адреса функцию. Вместо связки выражений `if-then-else` здесь можно использовать выражение `case`.

Листинг 4.8 Функция getLocationFunction

```
getLocationFunction location =
  case location of
    "ny" -> nyOffice
    "sf" -> sfOffice
    "reno" -> renoOffice
    _  -> (\name ->
          (fst name) ++ " " ++ (snd name))
```

case смотрит на значение location

Если location — "ny", то возвращаем nyOffice

Если location — "sf", то возвращаем sfOffice

Если location — "reno", то возвращаем renoOffice

В остальных случаях (символ _) возвращаем общее решение

Выражение `case` выглядит довольно простым, единственную сложность может представлять нижнее подчёркивание (_) в самом конце. Вы хотите описать ситуацию, когда введённая строка отличается от указанных вариантов местоположения. В Haskell символ _ часто используется как обозначение для произвольного значения. Мы обсудим его подробнее в следующем уроке. В этом случае, если пользователь передаёт неправильное местоположение, мы по-быстрому описываем лямбда-функцию, которая превратит кортеж с именем в строку. Теперь у нас есть единая функция, которая будет возвращать нужную функцию всякий раз, когда она понадобится. Наконец, можно переписать функцию `addressLetter`.

Листинг 4.9 Функция addressLetter, версия 2

```
addressLetter name location = locationFunction name
  where locationFunction = getlocationFunction location
```

В GHCi можно проверить, что ваша функция работает так, как и ожидалось:

```
GHCi> addressLetter ("Боб","Смит") "ny"
"Боб Смит: А/я 789, Нью-Йорк, штат Нью-Йорк, 10013"
GHCi> addressLetter ("Боб","Джонс") "ny"
"Боб Джонс: А/я 789, Нью-Йорк, штат Нью-Йорк, 10013"
GHCi> addressLetter ("Дейзи","Смит") "sf"
"Дейзи Смит - А/я 1010, Сан-Франциско, штат Калифорния, 94109"
GHCi> addressLetter ("Боб","Смит") "reno"
"Смит - А/я 456, Рино, штат Невада, 89523"
GHCi> addressLetter ("Боб","Смит") "la"
"Боб Смит"
```

Теперь, когда вы выделили каждую из функций, необходимых для генерации адресов, вы можете легко добавлять новые правила, когда они будут приходить из отделений общества. В этом примере возвращение функций в качестве значений помогло сделать код проще для понимания и расширения. Это был простой пример того, как можно использовать возвращение функций в качестве значений; мы всего лишь автоматизировали порядок передачи функций внутри программы.

Итоги

В этом уроке нашей целью было изучение функций как значений первого класса. Haskell позволяет передавать функции как аргументы и возвращать их в качестве результатов. Функции как значения первого класса — это очень мощный инструмент, потому что он делает возможным выделение конкретных вычислений в виде функций. Мощь функций как значений первого класса доказана их широким распространением в большинстве современных языков программирования. Посмотрим, как вы с этим разобрались.

Задача 4.1. Всё, что может быть сравнено (например, `[Char]`, который мы использовали для имён в кортежах), в Haskell можно сравнить с помощью функции `compare`. Функция `compare` возвращает как раз `GT`, `LT` или `EQ`. Перепишите `compareLastNames` с использованием `compare`.

Задача 4.2. Определите новую функцию формирования полного почтового адреса для Вашингтона, округ Колумбия, и добавьте её в реализацию функции `getLocationFunction`. Согласно требованиям отделения в округе Колумбия, любому имени должен предшествовать текст *Многоуважаемый(ая)*.

Замыкания и частичное применение функций

После прочтения урока 5 вы:

- сможете захватывать значения в лямбда-выражениях;
- научитесь использовать замыкания для создания новых функций;
- сумеете упрощать этот процесс с помощью частичного применения.

В этом уроке вы изучите последний ключевой элемент функционального программирования: замыкания. *Замыкания* представляют собой логическое продолжение лямбда-функций и функций как значений первого класса. Комбинируя лямбда-функции и функции как значения первого класса для создания замыканий, мы можем создавать функции динамически. Эта концепция оказывается очень мощной, хотя и одновременно требующей определённых усилий для понимания и привыкания. Благодаря возможности частичного применения функций в Haskell довольно удобно работать с замыканиями. К концу этого урока вы увидите, как частичное применение функций упрощает порой запутанную работу с замыканиями.

Обратите внимание. В предыдущем уроке вы научились передавать программную логику в функции с помощью функций как значений первого класса. Например, у вас может быть функция `getPrice`, которая принимает URL и функцию извлечения цены с конкретного сайта:

```
getPrice amazonExtractor url
```

Эта функция полезна, но что, если вам нужно извлечь цену по тысяче разных URL, используя для каждого `amazonExtractor`? Можно ли зафиксировать этот аргумент на лету, чтобы во все последующие вызовы функции передавать только параметр `url`?

5.1. Замыкания — создание функций функциями

В уроке 4 вы определили функцию под названием `ifEven` (листинг 4.3). Благодаря передаче функции в качестве аргумента `ifEven` вам удалось получить обобщённый вычислительный шаблон. Затем вы создали функции `ifEvenInc`, `ifEvenDouble` и `ifEvenSquare`.

Листинг 5.1 ifEvenInc, ifEvenDouble, ifEvenSquare

```
ifEvenInc n = ifEven inc n
ifEvenDouble n = ifEven double n
ifEvenSquare n = ifEven square n
```

Использование функций в качестве аргументов помогает сделать код чище. Но заметьте, что вы опять пишете код по одному и тому же шаблону! Приведённые выше определения идентичны, за исключением функции, передаваемой в `ifEven`. Нам нужна функция, создающая функции вида `ifEvenX`. Для решения этой проблемы можно описать возвращающую функции функцию `genIfEven`, как показано на рис. 5.1.

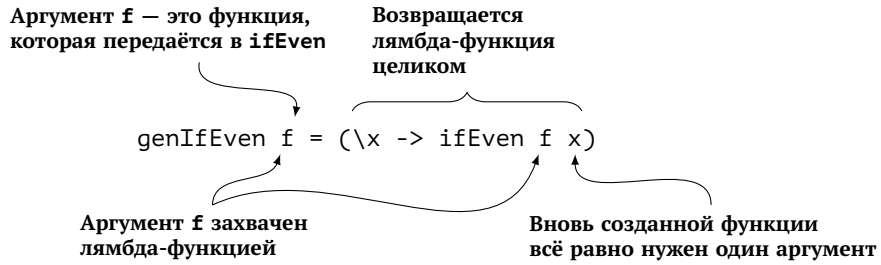


Рис. 5.1: Функция `genIfEven` позволяет создавать функции вида `ifEvenX`

Теперь вы передаёте функцию в качестве параметра и возвращаете лямбда-функцию. Функция `f`, которая подаётся на вход, захватывается лямбда-функцией. Когда вы захватываете некоторое значение в лямбда-функцию, это называется *замыканием*.

Даже в этом маленьком примере может быть сложно понять, что именно происходит. Чтобы лучше разобраться, давайте посмотрим, как создать вашу `ifEvenInc` с помощью `genIfEven`, соответствующий процесс проиллюстрирован на рис. 5.2.

Теперь давайте перейдём к примеру реальной задачи на использование замыканий: мы будем генерировать URL для применения с некоторым API.

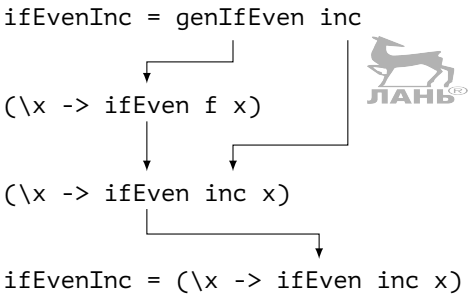


Рис. 5.2: Создание `ifEvenInc` с помощью замыкания

Проверка 5.1. Напишите функцию `genIfEvenX`, которая создаёт замыкание с аргументом `x` и возвращает новую функцию, позволяющую пользователю передать ей функцию в качестве аргумента и применить её к `x`, при условии что `x` является чётным.

5.2. Пример: генерация URL для API

Один из самых распространённых способов получить данные — обращение к API в стиле REST с помощью HTTP-запроса. Самый простой тип запроса — GET-запрос, при использовании которого все параметры, которые вам нужно передать на сервер, закодированы в URL. В этом примере для каждого запроса вам потребуются следующие данные:

- имя хоста;
- имя запрашиваемого ресурса;
- ID запрашиваемого ресурса;
- ваш ключ к API.

Ответ 5.1

```
ifEven f x = if even x
             then f x
             else x
genIfEvenX x = (\f -> ifEven f x)
```

На рис. 5.3 показан пример URL.

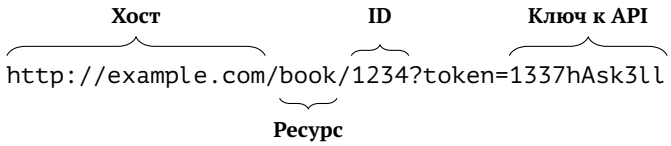


Рис. 5.3: Составные части URL

Построить URL из этих компонентов довольно просто. Ниже приведена соответствующая функция `getRequestUrl`.

Листинг 5.2 Функция getRequestUrl

```
getRequestUrl host apiKey resource id =
   host ++ "/" ++ resource ++ "/" ++ id ++ "?token=" ++ apiKey
```

Кое-что в этой функции может показаться вам странным: порядок аргументов не совпадает с порядком их использования, то есть с их расположением в самом URL. *Всякий раз, когда вы хотите использовать замыкание (а в Haskell это случается довольно часто), нужно располагать аргументы функции в порядке от наиболее общего к наименее общему.* В данном случае каждый хост может иметь несколько ключей к API, каждый ключ к API будет использоваться для различных ресурсов, а каждый ресурс будет иметь несколько ассоциированных с ним ID. Та же самая идея применима и к функции `ifEven`: передаваемая в неё функция будет работать с большим количеством различных входных данных, поэтому она считается более общей и должна стоять на первом месте в списке аргументов.

Теперь, когда у нас есть базовая функция для генерации запросов, посмотрим, как она работает:

```
GHCi> getRequestUrl "http://example.com" "1337hAsk3ll" "book"
                                                    ↳ "1234"
"http://example.com/book/1234?token=1337hAsk3ll"
```

Отлично! Это хорошее, достаточно обобщённое решение; ваша команда в целом будет отправлять множество запросов к разным хостам — поэтому важно не углубляться в частности. Почти каждый программист из команды будет фокусироваться на данных, получаемых от небольшого числа хостов. Впрочем, заставлять программистов каждый раз вручную вводить `http://example.com`, чтобы выполнить запрос, — не очень разумный подход, который к тому же может привести к ошибкам. Здесь больше подойдёт функция, которая позволит программисту создать собственный URL-генератор (функцию для сборки URL по нужным параметрам). Это

можно сделать с помощью замыкания. Ваш генератор будет выглядеть примерно как тот, что представлен на рис. 5.4.

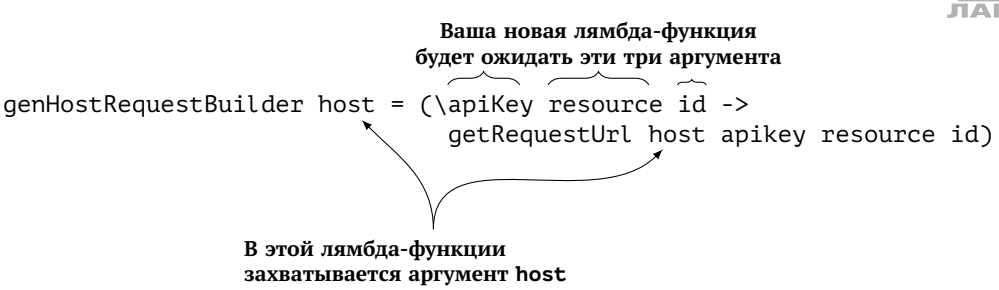


Рис. 5.4: Захват значения `host` в замыкании

Листинг 5.3 URL-генератор exampleUrlBuilder, версия 1

```
exampleUrlBuilder = genHostRequestBuilder "http://example.com"
```

Когда вы передаёте значение example.com в функцию, вы создаёте новую неименованную функцию, которая захватывает имя хоста и принимает на вход три оставшихся аргумента. Определяя `exampleUrlBuilder`, вы даёте имя этой анонимной функции. Теперь каждый раз, когда у вас появится необходимость начать отправлять запросы к новому URL, вы можете с лёгкостью создать для этого специальную функцию. Загрузите эту функцию в GHCi и посмотрите, как она упрощает ваш код:

```
GHCi> exampleUrlBuilder "1337hAsk3ll" "book" "1234"
"http://example.com/book/1234?token=1337hAsk3ll"
```

Если посмотреть на параметр `apiKey`, то становится ясно, что здесь возникает аналогичная ситуация. Каждый раз передавать ваш ключ к API при вызове `exampleUrlBuilder` всё же утомительно, потому что вы, скорее всего, будете использовать всего один или два различных ключа. Разумеется, для решения этой проблемы можно воспользоваться ещё одним замыканием! На этот раз вам нужно передать в генератор два аргумента: функцию `exampleUrlBuilder` и ваш `apiKey`.

Листинг 5.4 Функция genApiRequestBuilder

```
genApiRequestBuilder hostBuilder apiKey =
                (\resource id -> hostBuilder apiKey resource id)
```

Здесь интересно то, что вы комбинируете использование функций в качестве аргументов и возвращаемых значений. Внутри вашего замыка-

ния находится копия конкретной функции, которая понадобится вам в будущем, а также ключ к API, который необходимо зафиксировать. Наконец, вы можете создать функцию, которая сильно упрощает построение URL-запроса.

Листинг 5.5 Функция myExampleUrlBuilder, версия 1

```
myExampleUrlBuilder = genApiRequestBuilder exampleUrlBuilder
                                           "1337hAsk3ll"
```

А затем вы можете использовать её для быстрого создания URL с различными комбинациями запрашиваемых ресурсов и ID:

```
GHCi> myExampleUrlBuilder "book" "1234"
"http://example.com/book/1234?token=1337hAsk3ll"
```

Проверка 5.2. Напишите версию функции `genApiRequestBuilder`, которая принимает ещё и `resource` в качестве аргумента.

5.2.1. Частичное применение: упрощаем замыкания

Замыкания являются мощным средством языка. Однако использование лямбда-функций для создания замыканий делает чтение кода и рассуждение о них более сложными, чем следовало бы. Более того, все замыкания, которые вам пока что приходилось писать, следуют одному и тому же шаблону: зафиксировать некоторые параметры, принимаемые функцией, и создать новую функцию, принимающую остальные. Допустим, у вас есть функция `add4`, и складывающая значения четырёх переменных:

```
add4 a b c d = a + b + c + d
```

Теперь вы хотите создать функцию `addXto3`, которая принимает аргумент `x` и возвращает замыкание, принимающее остальные три:

```
addXto3 x = (\b c d -> add4 x b c d)
```

Ответ 5.2

```
genApiRequestBuilder hostBuilder apiKey resource =
                     (\id -> hostBuilder apiKey resource id)
```

Явное использование лямбда-выражения затрудняет понимание сути происходящего. А что, если вы захотите создать функцию `addXYto2`?

```
addXYto2 x y = (\c d -> add4 x y c d)
```

Хотя аргументов всего четыре, даже эту тривиальную функцию непросто осознать. Лямбда-функции — мощная и полезная концепция, но они определённо могут сделать слишком громоздкими определения функций, которые могли бы выглядеть гораздо более аккуратно.

У языка Haskell есть интересная особенность, позволяющая справляться с этой проблемой. Что произойдёт, если вы вызовете функцию `add4` меньше, чем с четырьмя аргументами? Ответ, кажущийся очевидным: будет выброшено исключение. Это — *не то*, что происходит в Haskell. Можно определить значение `mystery` в GHCi с помощью `add4` и одного аргумента:

```
GHCi> mystery = add4 3
```

Запустив этот код, вы обнаружите, что ошибки не происходит. Haskell создал для вас новую функцию:

```
GHCi> mystery 2 3 4
12
GHCi> mystery 5 6 7
21
```

Функция `mystery` складывает число 3 с тремя остальными аргументами, которые вы ей передаёте. В Haskell при вызове любой функции с количеством аргументов меньше требуемого вы получаете новую функцию, принимающую оставшиеся аргументы. Эта особенность языка называется *частичным применением*. Функция `mystery` делает то же, что и `addXto3` при передаче ей числа 3 в качестве аргумента. Помимо того что частичное применение функции спасло вас от использования лямбда-функции, вам также не нужно определять неуклюже названную `addXto3`! Вы столь же легко можете воспроизвести поведение функции `addXYto2`:

```
GHCi> anotherMystery = add4 2 3
GHCi> anotherMystery 1 2
8
GHCi> anotherMystery 4 5
14
```

Если использование замыканий показалось вам непонятным, то вам повезло! Благодаря частичному применению вам редко придётся писать или думать о замыканиях в Haskell явно. Вся работа таких функций, как

genHostRequestBuilder или genApiRequestBuilder, встроена в язык и может быть заменена отбрасыванием ненужных вам аргументов.

Листинг 5.6 exampleUrlBuilder и myExampleUrlBuilder, версия 2

```
exampleUrlBuilder = getRequestUrl "http://example.com"
myExampleUrlBuilder = exampleUrlBuilder "1337hAsk3ll"
```

В некоторых случаях в Haskell вам, возможно, всё же захочется использовать лямбда-функции для создания замыканий, но частичное применение функций используется гораздо чаще. На рис. 5.5 проиллюстрирован процесс частичного применения функции.

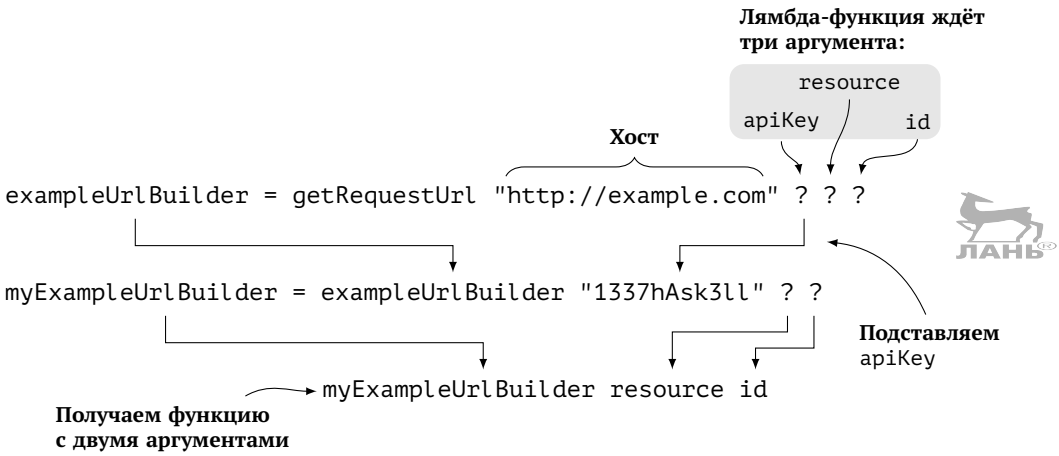


Рис. 5.5: Визуальное представление частичного применения функции

Проверка 5.3. Напишите функцию генерации URL для конкретных данных: хост http://example.com, ключ 1337hAsk3ll и ресурс book. Эта функция должна принимать в качестве аргумента только ID.

Ответ 5.3

```
exampleBuilder = getRequestUrl "http://example.com"
                               "1337hAsk3ll"
                               "book"
```

5.3. Собираем всё вместе

Частичное применение является причиной, по которой мы ввели правило о том, что аргументы должны располагаться в порядке от самого общего к наименее общему. При частичном применении аргументы применяются по порядку от первого к последнему. Мы нарушили это правило при определении функции `addressLetter` из урока 4 (листинг 4.6):

```
addressLetter name location = locationFunction name
  where locationFunction = getLocationFunction location
```

В этой функции аргумент `name` стоит перед аргументом `location`. Гораздо разумнее было бы создать функцию `addressLetterNY`, которая принимает имя, чем функцию `addressLetterBobSmith`, которая писала бы письма всем Бобам Смитам в мире. Переписать функцию не всегда возможно, особенно если вы используете функции из другой библиотеки, поэтому лучше исправить эту проблему созданием похожей частично применённой версии, как в следующем листинге.

Листинг 5.7 Функция addressLetterV2

```
addressLetterV2 location name = addressLetter name location
```

Это хорошее решение для одиночного случая исправления имеющейся функции `addressLetter`. Но что, если вы унаследовали кодовую базу, в которой множество библиотечных функций имеет аналогичную ошибку, только с двумя аргументами? Было бы здорово найти общее решение этой проблемы, а не писать отдельные исправления для каждого случая. Скомбинировав концепции, изученные к данному моменту, вы можете сделать это одной простой функцией. Вам нужно создать функцию под названием `flipBinaryArgs`, которая будет принимать функцию, изменять порядок следования аргументов и возвращать функцию, не изменившуюся в остальном. Чтобы это сделать, вам понадобятся: лямбда-функция, функции как значения первого класса и замыкание. Всё это можно совместить в одной строке кода на Haskell, как показано на рис. 5.6.

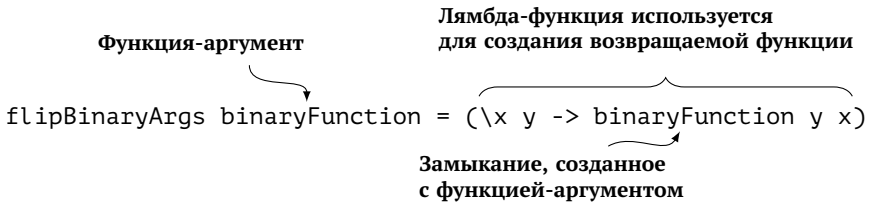


Рис. 5.6: Функция `flipBinaryArgs`

Теперь можно переписать функцию addressLetterV2 с использованием flipBinaryArgs, определив затем функцию addressLetterNY:

```
addressLetterV2 = flipBinaryArgs addressLetter
addressLetterNY = addressLetterV2 "ny"
```

Протестировать эти функции можно в GHCi:

```
GHCi> addressLetterNY ("Боб", "Смит")
Боб Смит: А/я 789, Нью-Йорк, штат Нью-Йорк, 10013
```

Функция flipBinaryArgs полезна не только для исправления кода, не соответствующего нашему правилу о порядке аргументов. Многие бинарные функции имеют естественный порядок аргументов, например деление. Полезный трюк в Haskell состоит в возможности использования любой инфиксной операции (такой как +, /, -, *) в качестве префиксной функции посредством взятия операции в скобки:

```
GHCi> 2 + 3
5
GHCi> (+) 2 3
5
GHCi> 10 / 2
5.0
GHCi> (/) 10 2
5.0
```

В делении и вычитании порядок аргументов важен. Несмотря на наличие естественного порядка аргументов, нетрудно понять, что может возникнуть необходимость в создании замыкания по второму аргументу. В этих случаях вы можете использовать flipBinaryArgs. Более того, функция flipBinaryArgs настолько полезна, что в Haskell уже есть такая функция, и называется она просто flip.

Проверка 5.4. Используя flip и частичное применение, определите subtract2, вычитающую число 2 из любого переданного ей числа.

Ответ 5.4

```
subtract2 = flip (-) 2
```

Итоги

Целью этого урока было изучение такого важного элемента функционального стиля программирования, как замыкания. Вооружившись лямбда-функциями, функциями как значениями первого класса и замыканиями, вы овладели всем необходимым для занятий функциональным программированием. Замыкания в комбинации с лямбда-функциями и функциями как значениями первого класса дают вам очень мощный инструмент. С помощью замыканий вы можете легко создавать новые функции на лету. Вы также разобрались с тем, как частичное применение облегчает работу с замыканиями. После того как вы привыкнете использовать частичное применение, вы иногда можете забыть, что вообще работаете с замыканиями! Давайте проверим, всё ли вам понятно.

Задача 5.1. Теперь, когда вы знаете о частичном применении функций, вам больше не нужно писать функции типа `genIfEvenX`. Переопределите `ifEvenInc`, `ifEvenDouble` и `ifEvenSquare`, используя `ifEven` и частичное применение.

Задача 5.2. Даже если бы в Haskell не было частичного применения, вы могли бы придумать нечто похожее. Используя ту же идею, что и в определении функции `flipBinaryArgs` (рис. 5.6), определите новую функцию `binaryPartialApplication`, которая принимает бинарную функцию вместе с одним её аргументом и возвращает функцию, принимающую пропущенный аргумент.

6

Списки

После прочтения урока 6 вы:

- научитесь выявлять компоненты списка;
- узнаете, как создавать списки;
- осознаете роль списков в функциональном программировании;
- разберётесь с основами ленивых вычислений.

Массивы в некотором смысле являются основной структурой данных в языке C. Если вы правильно понимаете суть массивов в C, то наверняка знаете, как работает распределение памяти, как на компьютере хранятся данные, что такое указатели и арифметика указателей. Для Haskell (и функционального программирования в целом) основной структурой данных является список. Даже когда вы приблизитесь к некоторым более продвинутым темам этой книги вроде функторов и монад, простой список будет по-прежнему оставаться самым полезным примером.

В данном уроке мы введём вас в эту удивительно важную структуру данных. Вы изучите основы устройства списков, их компоненты и способы формирования, а также узнаете некоторые предоставляемые Haskell функции на списках. Наконец, вы взглянете на ещё одну уникальную черту Haskell — ленивые вычисления. Ленивые вычисления настолько мощны, что позволяют представлять и работать со списками бесконечной длины! Если вы впоследствии застрянете на какой-то теме, то почти всегда полезно вернуться к спискам и посмотреть, не подскажут ли они что-нибудь.

Обратите внимание. Допустим, вы работаете в компании со штатом в 10 000 сотрудников, и некоторые из них хотят играть после работы в команде по софтболу. Всего в компании пять команд, названных по цветам, которые хотелось бы использовать для приписывания сотрудников к командам:

```
teams = ["красный","жёлтый","оранжевый","синий","зелёный"]
```

У вас есть список сотрудников и вы хотите приписывать их к верной команде настолько равномерно, насколько возможно. Каким простым способом вы можете использовать функции на списках в Haskell, чтобы решить эту задачу?

6.1. Анатомия списков

Списки — это единственная наиболее важная структура данных в функциональном программировании. Одна из главных причин состоит в том, что списки по своей природе рекурсивны. Список — это либо пустой список, либо элемент, за которым следует список. Разделение списка на компоненты и формирование нового списка образуют базовый инструментарий для многих приёмов функционального программирования. При разделении списка главными компонентами оказываются голова (`head`) и хвост (`tail`). Случай пустого списка обозначается как `[]`. Функция `head` возвращает первый элемент списка:

```
GHCi> head [1,2,3]
1
GHCi> head [[1,2],[3,4],[5,6]]
[1,2]
```

Функция `tail` возвращает остальную часть списка, идущую после головы:

```
GHCi> tail [1,2,3]
[2,3]
GHCi> tail [3]
[]
```

Хвост списка с одним элементом — это пустой список `[]`, что можно воспринимать как маркер конца. Пустой список отличается от остальных

списков, поскольку у него нет ни головы, ни хвоста. Вызов `head` или `tail` на пустом списке даст ошибку. Если вы внимательно подумаете об идее разделения списка на голову и хвост, то сразу поймёте рекурсивную природу обработки списков: голова — это элемент, а хвост — другой список. Просто представьте себе, что вы надрываете первый элемент в списке покупок, как это показано на рис. 6.1.

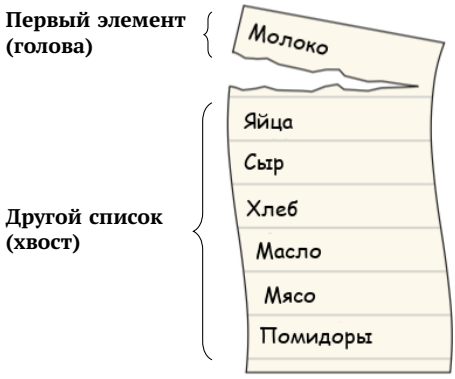


Рис. 6.1: Список составлен из первого элемента и хвоста списка

Вы можете разбить список на кусочки, но в этом мало пользы, если не сможете собрать его обратно! В функциональном программировании построение списков так же важно, как и разделение на компоненты. Чтобы построить список, вам нужна всего одна функция, она же операция `(:)`. Мы будем называть эту операцию *конструированием списка*, так как символ `:` в предложении выглядит несколько странно.

Чтобы сконструировать список, вы должны взять значение и присоединить его к другому списку. Самый простой способ создать список — присоединить значение к пустому списку:

```
GHCi> 1:[]
[1]
```

Под капотом все списки в Haskell представлены как набор операций конструирования, а запись вида `[...]` — это синтаксический сахар (элемент синтаксиса языка программирования, созданный исключительно для упрощения чтения):

```
GHCi> 1:2:3:4:[]
[1,2,3,4]
```

```
GHCi> (1,2):(3,4):(5,6):[]
[(1,2),(3,4),(5,6)]
```

Обратите внимание, что все эти списки заканчиваются пустым списком `[]`. По определению, список — это всегда значение, присоединённое к другому списку (который может быть пустым). Вы можете присоединить значение к началу уже существующего списка, если хотите:

```
GHCi> 1:[2,3,4]
[1,2,3,4]
```

Стоит заметить, что строки, которые вы уже видели, сами по себе являются синтаксическим сахаром для списка символов (обозначаемых одинарными кавычками, а не двойными, как строки):

```
GHCi>['п','р','и','в','е','т']
"привет"
GHCi> 'п':'р':'и':'в':'е':'т':[]
"привет"
```

Важно помнить, что в Haskell все элементы списка должны быть одного типа. Например, вы можете сконструировать список из буквы `'п'` и строки `"ривет"`, потому что `"ривет"` — это список символов, а `'п'` (в одинарных кавычках) — одиночный символ:

```
GHCi> 'п':"ривет"
"привет"
```

Но вы не можете сконструировать список из `"п"` и `"ривет"`, потому что `"п"` — список из одного символа, а значения в `"ривет"` — это отдельные символы. Это становится более очевидно, когда вы убираете синтаксический сахар.

Листинг 6.1 Конструирование списков из символов и строк

```
GHCi> "п":"ривет"                ← Ошибка!
GHCi> ['п']:['р','и','в','е','т']   ← Тот же код без одного слоя синтаксического сахара
GHCi>  'п':[]:'р':'и':'в':'е':''т":[]   ← Синтаксический сахар убран полностью
```

Если вы хотите соединить два списка, то вам нужно *сконкатенировать* их с помощью операции ++. Вы уже видели её в уроке 3, где мы конкатенировали строки текста, но, учитывая, что строки — это просто списки, она сработает с любыми списками:

```
GHCi> "п" ++ "ривет"
"привет"
GHCi> [1] ++ [2,3,4]
[1,2,3,4]
```

Конструирование списков важно понять, так как это неотъемлемая часть написания рекурсивных функций на списках. Многие более сложные действия в функциональном программировании включают построение списков, разбиение на части или их комбинации.

6.2. Списки и ленивые вычисления

Поскольку списки так важны в Haskell, существует множество способов быстро их генерировать. Вот несколько примеров:

```
GHCi> [1 .. 10]
[1,2,3,4,5,6,7,8,9,10]
```

Создаём список чисел от 1 до 10

```
GHCi> [1,3 .. 10]
[1,3,5,7,9]
```

Создаём список нечётных чисел с шагом как от 1 до 3

```
GHCi> [1, 1.5 .. 5]
[1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0]
```

Создаём список чисел, увеличивающихся на 0.5

```
GHCi> [1,0 .. -10]
[1,0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]
```

Создаём список чисел в порядке убывания

Это полезно, но не слишком интересно. Во многих языках программирования есть функция `range`, которая работает похожим образом. Что произойдёт, если вы забудете указать верхнюю границу вашего диапазона?

```
GHCi> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12 ..
```

Только что сгенерировался бесконечный список! Это здорово, но быстро забивает экран и не выглядит слишком полезным. Интересно, что можно сохранить этот список в переменной и даже использовать в функциях:

```
simple x = x
longList = [1 .. ]
stillLongList = simple longList
```

Шокирует то, что такой код компилируется абсолютно нормально. Вы определили бесконечный список и даже использовали его в функции. Почему Haskell не завис в попытках вычислить бесконечно длинный список? Haskell использует особую форму вычислений, которые называются *ленивыми*. В ленивых вычислениях код не вычисляется до того момента, пока он не потребуется. В случае `longList` ни одно из значений списка не было нужно для получения результата.

Ленивые вычисления имеют преимущества и недостатки. Легко увидеть некоторые преимущества. Во-первых, вы получаете эффективное вычисление, так как любой код, который вам не нужен, никогда не будет вычислен. Другое преимущество в том, что вы можете определять и использовать интересные структуры данных вроде бесконечных списков. Это может оказаться полезным для ряда практических задач. Недостатки ленивых вычислений менее очевидны. Самый существенный состоит в том, что трудно анализировать производительность кода. В этом тривиальном примере легко увидеть, что аргумент, переданный в `simple`, не будет вычислен, но даже чуть большая сложность кода делает всё менее очевидным. Ещё большая проблема состоит в том, что вы легко можете собрать большой набор невычисленных функций, которые было бы гораздо дешевле хранить как значения.

Проверка 6.1. Правда или ложь: можно скомпилировать и выполнить программу с переменной

```
backwardsInfinity = reverse [1..]
```

Ответ 6.1 Правда. Даже несмотря на то, что вы обращаете бесконечный список, вы никогда не вызываете этот код, а значит, этот список никогда не вычислится. Однако если вы загрузите код с определением `backwardsInfinity` в GHCi и наберёте:

```
GHCi> backwardsInfinity
```

то у вас будут проблемы, так как программа попытается вычислить это значение, чтобы напечатать получившийся результат.

6.3. Основные функции на списках

Важность списков такова, что разработчики языка Haskell встроили широкий спектр полезных функций для обработки списков в модуль стандартной библиотеки, который называется `Prelude`. Вы уже видели функции `head` и `tail`, а также операции `:` и `++`, которые позволяют разделять списки на компоненты и собирать их обратно. Есть много других функций на списках, которые при программировании на Haskell будут часто встречаться, так что стоит с ними познакомиться.

6.3.1. Операция !!

Если вы хотите получить доступ к отдельному элементу списка, то можете использовать операцию `!!`. Эта операция принимает на вход список и число, а возвращает элемент списка, находящийся на соответствующей позиции списка. Индексация списков в Haskell начинается с 0. Если вы попытаетесь получить элемент за границами диапазона, то получите сообщение об ошибке выполнения операции `!!` с текстом, что индекс слишком велик:

```
GHCi> [1,2,3] !! 0
1
GHCi> "щенки" !! 3
'к'
GHCi> [1..10] !! 11
*** Exception: Prelude.!!: index too large
```

Как уже говорилось в уроке 5, любая инфиксная операция (операция, размещённая между двумя значениями, вроде +) может также использоваться как префиксная, если обернуть её в скобки:

```
GHCi> (!!) [1,2,3] 0
1
```

Использование префиксной записи способно упростить многие вещи вроде частичного применения. Префиксная запись также полезна для использования операций как аргументов к другим функциям. Вы также можете использовать частичное применение с инфиксной операцией; для этого просто нужно обернуть выражение в скобки:

```
GHCi> paExample1 = (!!) "собака"
GHCi> paExample1 2
'б'
```

```
GHCi> paExample2 = ("собака" !!)
GHCi> paExample2 2
'б'
```

Обратите внимание, что в `paExample2` частичное применение работает для инфиксной бинарной операции. Чтобы выполнить частичное применение на бинарной операции (результат называется *сечением*), необходимо заключить выражение в скобки. Если включить только один аргумент справа, функция будет ждать левый аргумент. Если включить только аргумент слева, вы получите функцию, которая будет ждать аргумент справа. В следующем примере создаётся частичное применение с правым аргументом:

```
GHCi> paExample3 = (!! 2)
GHCi> paExample3 "собака"
'б'
```

При использовании сечений важно помнить, что скобки в них обязательны.

6.3.2. Функция length

Функция `length` очевидна, она возвращает длину списка!

```
GHCi> length [1..20]
20
GHCi> length [(10,20),(1,2),(15,16)]
3
GHCi> length "Программируй на Haskell"
23
```

6.3.3. Функция reverse

Ничего удивительного, reverse разворачивает элементы списка в обратном порядке:

```
GHCi> reverse [1,2,3]
[3,2,1]
GHCi> reverse " сыр"
"рыс"
```

Функцию `reverse` можно использовать для создания простой проверки палиндромов, как показано в следующем листинге.

Листинг 6.2 Функция isPalindrome

```
isPalindrome word = word == reverse word
```

```
GHCi> isPalindrome "сыр"
False
GHCi> isPalindrome "топот"
True
GHCi> isPalindrome [1,2,3]
False
GHCi> isPalindrome [1,2,1]
True
```

6.3.4. Функция elem

Функция `elem` принимает на вход значение и список и проверяет, имеется ли это значение в списке:

```
GHCi> elem 13 [0,13 .. 100]
True
GHCi> elem 'п' " сыр"
False
```

Функцией `elem` для большей читабельности удобно пользоваться как инфиксной операцией. Любая бинарная функция может использоваться как инфиксная операция, если заключить её имя в обратные кавычки (`). Например, функция `respond` из следующего листинга возвращает ответ в зависимости от наличия в строке восклицательного знака.

Листинг 6.3 Функция respond

```
respond phrase = if '!' `elem` phrase
                 then "Ого!"
                 else "Ну, ок"

GHCi> respond "привет"
"Ну, ок"
GHCi> respond "привет!"
"Ого!"
```

Добавляет ли инфиксная запись `elem` читабельности — вопрос спорный, но в реальном коде вы часто будете встречать инфиксную запись бинарных функций.

6.3.5. Функции take и drop

Функция `take` принимает число `n` и список в качестве аргументов и возвращает первые `n` элементов списка:

```
GHCi> take 5 [2,4..100]
[2,4,6,8,10]
GHCi> take 3 "замечательно"
"зам"
```

Если вы запросите больше значений, чем есть в списке, `take` вернёт всё, что сможет, без ошибки:

```
GHCi> take 1000000 [1]
[1]
```

Функция `take` лучше всего работает в комбинации с другими функциями на списках. Например, вы можете использовать `take` и `reverse`, чтобы получить `n` последних элементов списка.

Листинг 6.4 Функция takeLast

```
takeLast n aList = reverse (take n (reverse aList))
GHCi> takeLast 10 [1..100]
[91,92,93,94,95,96,97,98,99,100]
```

Функция `drop` похожа на `take`, она опускает первые `n` элементов списка:

```
GHCi> drop 2 [1,2,3,4,5]
[3,4,5]
GHCi> drop 6 "Очень здорово"
"здорово"
```

6.3.6. Функция zip

Функция `zip` используется, когда вы хотите скомбинировать два списка в пары. Аргументами для `zip` выступают два списка. Если один из списков короче, то `zip` остановится, когда один из списков будет пуст:

```
GHCi> zip [1,2,3] [2,4,6]
[(1,2),(2,4),(3,6)]
GHCi> zip "пёс" "кролик"
[('п','к'),('ё','р'),('с','о')]
GHCi> zip ['a' .. 'f'] [1 .. ]
[('a',1),('b',2),('c',3),('d',4),('e',5),('f',6)]
```

6.3.7. Функция cycle

Функция `cycle` особенно интересна, потому что она использует ленивые вычисления для создания бесконечного списка. Получив на вход

список, cycle бесконечно его повторяет. Это может показаться бесполезным, но пригождается на удивление часто. Например, в численных расчётах иногда возникает необходимость в списке из *n* единиц. Написать такую функцию с помощью cycle несложно.

Листинг 6.5 Функция ones

```
ones n = take n (cycle [1])

GHCi> ones 2
[1,1]
GHCi> ones 4
[1,1,1,1]
```

Функция cycle может быть очень полезна для распределения элементов списка по группам. Представьте, что вы хотите распределить список файлов для отправки их на n серверов, или просто разделить сотрудников на n команд. Общее решение — написать функцию assignToGroups, которая принимает количество групп и список, после чего циклически распределяет элементы по группам.

Листинг 6.6 Функция assignToGroups

```
assignToGroups n aList = zip groups aList
    where groups = cycle [1..n]

GHCi> assignToGroups 3 ["файл1.txt","файл2.txt","файл3.txt"
                      ↳ ,"файл4.txt","файл5.txt","файл6.txt"
                      ↳ ,"файл7.txt","файл8.txt","файл9.txt"]
[(1,"файл1.txt"),(2,"файл2.txt"),(3,"файл3.txt"),
↳ (1,"файл4.txt"),(2,"файл5.txt"),(3,"файл6.txt"),
↳ (1,"файл7.txt"),(2,"файл8.txt"),(3,"файл9.txt")]

GHCi> assignToGroups 2 ["Боб","Кэт","Сью","Джоан","Майк"]
[(1,"Боб"),(2,"Кэт"),(1,"Сью"),(2,"Джоан"),(1,"Майк")]
```

Это только некоторые функции из широкого спектра функций для обработки списков, которые предлагает Haskell. Не все функции на списках включены в стандартный модуль Prelude. Большинство функций, включая те, что автоматически включены в Prelude, находятся в модуле Data.List. Исчерпывающий список функций этого модуля можно найти в документации: https://hackage.haskell.org/package/base/docs/Data-List.html.

Итоги

Целью этого урока было детальное изучение строения списка. Вы узнали, что список состоит из головы и хвоста. Мы также прошли множество самых распространённых функций на списках. Давайте проверим, как вы с этим разобрались.

Задача 6.1. В Haskell есть функция `repeat`, которая принимает значение и бесконечно его повторяет. Используя функции, которые вы изучили, реализуйте свою версию `repeat`.

Задача 6.2. Напишите функцию `subseq`, которая будет принимать три аргумента: начальную позицию, конечную позицию и список. Функция должна возвращать подпоследовательность от начальной позиции до конечной. Например:

```
GHCi> subseq 2 5 [1 .. 10]
[3,4,5]
GHCi> subseq 4 8 "это щенок"
"щенок"
```

Задача 6.3. Напишите функцию `inFirstHalf`, которая возвращает `True`, если элемент находится в первой половине списка, и `False` в противном случае.

Правила рекурсии и сопоставление с образцом

После прочтения урока 7 вы:

- поймёте определение рекурсивной функции;
- узнаете правила написания рекурсивных функций;
- разберётесь с примерами определений рекурсивных функций;
- научитесь использовать сопоставление с образцом для решения задач на рекурсию.

Одна из первых проблем при написании прикладного кода в функциональном стиле состоит в невозможности изменять состояние, у вас также нет циклов, опирающихся на изменение состояния, вроде `for`, `while` или `until`. Все итерационные задачи должны решаться через рекурсию. Для многих программистов эта мысль сама по себе ужасна, так как рекурсия обычно навевает воспоминания о вызывавших головную боль решениях. К счастью, чтобы упростить процесс написания рекурсивного кода, достаточно следовать нескольким простым правилам. Кроме того, как частичное применение упрощало работу с замыканиями, так и имеющееся в Haskell *сопоставление с образцом* упрощает рассуждения относительно рекурсивных функций.

Обратите внимание. В предыдущем уроке вы изучили функцию `take`, которая позволяет взять из списка первые `n` элементов:

```
GHCi> take 3 [1,2,3,4]
[1,2,3]
```

Как бы вы написали свою версию `take`?

7.1. Рекурсия

Вообще говоря, нечто *рекурсивно*, если оно определяется через само себя. Обычно это приводит к головной боли, так как программисты начинают представлять раскручивание бесконечного цикла рекурсивных определений. Рекурсия не должна вызывать головную боль, и зачастую она куда более естественна, чем другие программные формы итерирования. Списки — это рекурсивная структура данных, определённая как пустой список или элемент и другой список. При работе со списками не возникает головной боли, не требуются и упражнения ментальной гимнастики. Рекурсивные функции — это функции, которые используют сами себя в своём определении. Ну да, звучит страшновато.

Но если вы будете думать о рекурсивных функциях, как описаниях рекурсивных процессов, это понятие вдруг окажется вполне обыденным. Почти любая человеческая деятельность — это рекурсивный процесс! Например, мытьё посуды. Если в раковине нет посуды, то вы закончили её мыть, но если там есть тарелка, то вы берёте её, моете и ставите на полку. И так далее, пока не закончите.

Проверка 7.1. Опишите что-нибудь, что вы делаете ежедневно, как рекурсивный процесс.

Ответ 7.1. Когда я пишу урок для этой книги, я пишу текст, а затем:

(1) получаю правки от редактора;

(2) принимаю или отклоняю эти изменения, делаю свою правки;

(3) отправляю текст редактору;

(4) если редактор счастлив, я закончил! Иначе возвращаюсь к шагу 1.

7.2. Правила рекурсии

Сложности с рекурсией возникают, когда вы начинаете записывать рекурсивные процессы. Даже в случае списка или алгоритма мытья посуды написать их с нуля оказывается гораздо сложнее, чем просто осознать факт рекурсивности. Секрет написания рекурсивных функций в том, чтобы *не думать о рекурсии*! Если слишком много думать о рекурсии, то заболит голова. Писать рекурсивные функции нужно, руководствуясь следующими простыми правилами:

(1) определите конечную цель (или цели);

(2) определите, что происходит, когда цель достигнута;

(3) перечислите все имеющиеся возможности;

(4) определите, какие действия следует повторять;

(5) убедитесь, что каждая из возможностей приближает к цели.

7.2.1. Правило 1: определите конечную цель (или цели)

Обычно рекурсивные процессы завершаются. Как выглядит этот конец? В случае списков конец — это пустой список, в случае мытья посуды — это пустая раковина. После того как вы распознаете рекурсивность процесса, первым шагом к его реализации в виде рекурсивной функции будет выяснение, чем же вы должны закончить. Иногда бывает, что цель не одна. Телемаркетологу для завершения работы бывает необходимо обзвонить 100 человек или же сделать 5 продаж. В этом случае цель — сделать 100 звонков или совершить 5 продаж.

7.2.2. Правило 2: определите, что происходит, когда цель достигнута

Для каждой установленной в правиле 1 цели необходимо определить, каким будет результат. В случае мытья посуды результатом будет то, что вы помоете посуду. Функции должны вернуть значение, значит, нужно определить, какое значение должно возвращаться в конечном итоге. Типичная проблема возникает, когда программисты начинают думают о результате как о конце длинного рекурсивного процесса. Обычно в этом нет необходимости, и это слишком сложно. Зачастую ответ очевиден, если задать вопрос: «Что произойдёт, если я вызову функцию на итоговом значении?» Например, итоговое состояние последовательности чисел Фибоначчи — это 1, ведь по определению `fib 1 = 1`. В более простом примере на-

хождения количества книг в вашем шкафу путём подсчёта книг на каждой полке конечное состояние — это когда больше не остаётся полок. Количество книг, если полки закончились, — 0.

7.2.3. Правило 3: перечислите все имеющиеся возможности

Если вы пока не в конечном состоянии, то чем заняться? Звучит так, будто у вас тут много работы, но, как правило, для прихода в конечное состояние имеется только одна или две возможности. Если у вас нет пустого списка, то у вас есть список, в котором что-то есть. Если раковина не пуста, то у вас есть раковина с посудой. В случае телемаркетолога, если вы ещё не обзвонили 100 человек или не сделали 5 продаж, то у вас есть две возможности — вы можете позвонить и что-то продать либо позвонить и ничего не продать.

7.2.4. Правило 4: определите, какие действия следует повторять

Это правило почти идентично правилу 2, только здесь вам нужно отыскать повторяемые действия. Не накручивайте и не пробуйте развернуть рекурсию. В случае списка вы можете взять элемент и взглянуть на хвост. При мытье посуды вы моете тарелку, ставите её сушиться и опять смотрите в раковину. Телемаркетолог либо делает звонок, оформляет продажу и повторяет, либо фиксирует, что звонок был сделан без продажи, после чего снова повторяет.

7.2.5. Правило 5: убедитесь, что каждая из возможностей приближает к цели

Это важное правило! Для каждого из действий, найденных в правиле 4, вам необходимо спросить себя, «приближает ли это действие меня к цели?» Если вы будете всё время брать хвост от списка, то получите пустой список. Если вы продолжите убирать посуду из раковины, то получите пустую раковину. Оформление продаж или фиксация звонков в итоге приведёт какой-то из счётчиков к цели. Но предположим, что вы хотите бросать монетку, пока не выпадет орёл. Цель в том, чтобы получить орла; как только вы его получите, вы остановитесь. Одна из возможностей — выпадение решки, если выпала решка, то вы бросаете монетку снова. К сожалению, очередной бросок не гарантирует, что вы когда-либо получите орла! По статистике вы должны прийти к завершению, так что на практике всё должно быть в порядке, но это функция, запускать которую потенциально опасно

(представьте, что вместо монетки вы используете что-то с ещё меньшим шансом на успех).

7.3. Ваша первая рекурсивная функция: наибольший общий делитель

Чтобы познакомиться с рекурсией, начнём с одного из старейших численных алгоритмов — алгоритма Евклида. Это удивительно простой метод для вычисления наибольшего общего делителя (НОД) двух чисел. Если вы забыли, наибольший общий делитель двух чисел — это самое большое число, которое делит каждое из них. Например, НОД чисел 20 и 16 — это 4, потому что 4 — это самое большое число, которое делит и 20, и 16. Для 10 и 100 НОД равен 10. Евклид описал алгоритм в своей книге *Начала* (написанной примерно в 300 году до н. э.). Вот его краткое изложение:

(1) вы начинаете с двумя числами, a и b;

(2) если вы делите a на b и остаток равен 0, ясно, что b — НОД;

(3) в противном случае вы меняете значение a, присваивая значение b (b становится новым a). Вы также меняете значение b на остаток из шага 2 (новое b — это остаток изначального a, поделённого на изначальное b);

(4) продолжаете, пока у a/b не будет остатка.

Рассмотрим пример:

(1) $a = 20$, $b = 16$;

(2) $a/b = 20/16 = 1$, в остатке 4;

(3) $a = 16$, $b = 4$;

(4) $a/b = 4$, в остатке 0;

(5) НОД $= b = 4$.

Для реализации этого алгоритма в коде необходимо начать с выявления цели (правило 1). Цель в том, чтобы не было остатка от a/b. В коде для реализации этой идеи вы можете использовать функцию взятия остатка. В Haskell ваша цель выражается следующим образом:

```
a `mod` b == 0
```

Следующий вопрос, на который необходимо ответить, — это что нужно возвращать, когда вы доходите до конечного состояния (правило 2)? Если

a/b не имеет остатка, b должно делить a нацело, таким образом, b — НОД. Получаем поведение при достижении цели:

```
if a `mod` b == 0
then b ....
```

Далее необходимо отыскать все способы, которыми можно приближаться к цели, если вы её ещё не достигли (правило 3). Для нашей задачи есть только одна возможность — остаток не равен 0. Если остаток не равен 0, то нужно повторять алгоритм, в котором b — это новое a, а b — это остаток, a `` `mod` `` b (правило 4):

```
else gcd b (a `mod` b)
```

Теперь вы можете собрать всё это в рекурсивную реализацию алгоритма Евклида, как показано далее:

Листинг 7.1 Функция myGCD

```
myGCD a b = if remainder == 0
            then b
            else myGCD b remainder
  where remainder = a `mod` b
```

Наконец, вы должны убедиться, что всякий раз приближаетесь к цели (правило 5). Используя остаток, вы всегда будете уменьшать b, в худшем случае (a и b — простые) вы просто дойдёте до 1 для a и b. Это рассуждение подтверждает, что алгоритм должен завершиться. Руководствуясь правилами для написания рекурсивных функций, вы избежали размышлений о бесконечно закручивающейся рекурсии!

Проверка 7.2. Имеет ли значение для функции `myGCD`, что верно, $a > b$ или $a < b$?

Ответ 7.2. Не имеет значения, в случае $a < b$ добавляется всего один шаг. Например, `20 `mod` 50` — это 20, значит, следующим вызовом будет `myGCD 50 20`, что всего на один шаг больше, чем если сразу вызвать `myGCD 50 20`.

В примере myGCD имеется только две возможности: либо мы пришли к цели, либо процесс повторяется. Эта ситуация отлично укладывается в выражение `if then else`. Нетрудно догадаться, что если вам встретится более сложная функция, то можно натолкнуться на большие и очень большие выражения `if then else` или даже использование `case`. В Haskell имеется удивительная возможность, называющаяся *сопоставлением с образцом*, которая позволяет удобным образом смотреть на значения, переданные функции в качестве аргументов, и действовать соответствующим образом. Для примера давайте реализуем функцию `sayAmount`, которая будет возвращать `"один"` для 1, `"два"` для 2 и `"много"` для всего остального. Во-первых, давайте попробуем реализовать её с помощью `case`:

Листинг 7.2 Функция sayAmount, версия 1

```
sayAmount n = case n of
    1 -> "один"
    2 -> "два"
    n -> "много"
```

Версия с использованием сопоставления с образцом выглядит как три отдельных определения, каждое для одного значения аргумента:

Листинг 7.3 Функция sayAmount, версия 2

```
sayAmount 1 = "один"
sayAmount 2 = "два"
sayAmount n = "много"
```

Сопоставление с образцом, как и `case`, рассматривает варианты по порядку, так что если бы вы поместили `sayAmount n` в начало перечня вариантов, то вызов `sayAmount` всегда возвращал бы `"много"`.

Используя сопоставление с образцом, важно понимать, что оно может только смотреть на аргументы, но не может выполнять над ними какие бы то ни было вычисления. Например, сопоставляя с образцом, вы не сможете проверить, меньше ли `n`, чем 0, или нет. Даже с таким ограничением сопоставление с образцом выглядит очень мощным инструментом. Вы можете использовать сопоставление с образцом, чтобы проверять, пуст ли список, сопоставляя с `[]`:

```
isEmpty [] = True
isEmpty aList = False
```

В Haskell имеется стандартная практика использовать _ как произвольное для значений, которые вас не интересуют. В `isEmpty` вы не используете параметр `aList`, поэтому обычно этот аргумент записывают так:

```
isEmpty [] = True
isEmpty _ = False
```

На списках можно делать и более сложное сопоставление с образцом. В Haskell популярно соглашение об использовании имени `x` для представления одиночного значения и `xs` — для целого списка (хотя мы для большей читабельности часто будем это соглашение игнорировать). Вы можете определить свою версию функции `head` следующим образом:

```
myHead (x:xs) = x
```

Чтобы лучше разобраться с тем, что делает Haskell, когда дело доходит до сопоставления с образцом, посмотрите на рис. 7.1, в котором иллюстрируется сопоставление с образцом для списка.

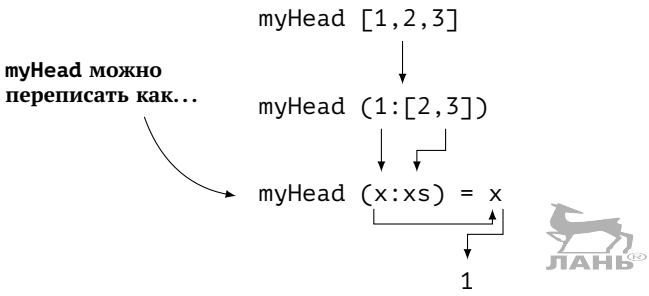


Рис. 7.1: Визуализация сопоставления с образцом в функции `myHead`

Как и в реальной версии `head` в Haskell, у вас не обрабатывается случай пустого списка, у которого нет головы. В этом случае можно выбросить ошибку, воспользовавшись функцией `error`.

Листинг 7.4 Функция myHead

```
myHead (x:xs) = x
myHead [] = error "Голова у пустого списка отсутствует"
```

Так как вы хотите думать о рекурсии просто как о списке целей и возможных действий, сопоставление с образцом становится незаменимым средством для безболезненного написания рекурсивного кода. Идея в том, что мы начинаем думать образцами. Когда бы вы ни писали рекурсивные функции, вы можете разделить определения так, чтобы иметь дело только с конечными целями, всегда определяемыми в первую очередь, а затем со всеми имеющимися возможностями одновременно. Зачастую это приводит к более коротким определениям функций, но, что более важно, так

становится проще рассуждать о сути каждого шага. Сопоставление с образцом — это отличный способ смягчить боль и симптомы рекурсии.

Проверка 7.3. Дополните это определение функции `myTail`, используя сопоставление с образцом, и убедитесь, что используете _ там, где значение не требуется:

```
myTail (<заполните здесь>) = xs
```

Итоги

Целью этого урока было изучение способа написания рекурсивных функций. Пока вы в этом деле неопытны, задача зачастую кажется более сложной, чем она есть на самом деле. Вот основные правила, которые должны помочь вам, если вы застряли:

(1) определите конечную цель (или цели);

(2) определите, что происходит, когда цель достигнута;

(3) перечислите все имеющиеся возможности;

(4) определите, какие действия следует повторять;

(5) убедитесь, что каждая из возможностей приближает к цели.

Давайте посмотрим, как вы это поняли.

Задача 7.1. Функция `tail` в Haskell возвращает ошибку, когда вызывается на пустом списке. Измените реализацию `myTail` так, чтобы она обрабатывала пустой список, возвращая пустой список.

Задача 7.2. Перепишите myGCD, используя сопоставление с образцом.

Ответ 7.3

```
myTail (_:xs) = xs
```

8

Написание рекурсивных функций

После прочтения урока 8 вы:

- познакомитесь с общими схемами применения правил рекурсии;
- поймёте, как применять рекурсию для обработки списков;
- научитесь измерять время работы функций в GHCi;
- сможете рассуждать о пограничных ситуациях в случаях применения пяти правил рекурсии.

Лучший способ научиться работать с рекурсией — практика, практика и ещё раз практика! Чтобы помочь вам лучше разобраться с применением правил рекурсии из предыдущего урока, в этом мы рассмотрим несколько рекурсивных функций. В процессе изучения материала урока вы начнёте подмечать некоторые схемы, повторяющиеся при решении задач, связанных с рекурсией. В связи с тем, что Haskell не позволяет вам «жульничать» посредством использования итерации с сохранением промежуточного состояния, почти весь код, который вы пишете на Haskell, в некоторой степени использует рекурсию (хотя зачастую она является неявной). Поэтому вы быстро приспособитесь к использованию рекурсивных функций и решению задач в рекурсивном стиле.

Обратите внимание. В предыдущем уроке вам предлагалось обдумать собственную реализацию функции `take`. На этот раз давайте рассмотрим функцию `drop`:

```
GHCi> drop 3 [1, 2, 3, 4]
[4]
```

Напишите свою реализацию функции `drop` и подумайте, чем эта функция похожа и чем отличается от `take`.

8.1. Обзор: правила рекурсии

В предыдущем уроке вы узнали о правилах написания рекурсивных функций. Для удобства они приведены ниже:

(1) определите конечную цель (или цели);

(2) определите, что происходит, когда цель достигнута;

(3) перечислите все имеющиеся возможности;

(4) определите, какие действия следует повторять;

(5) убедитесь, что каждая из возможностей приближает к цели.

Чтобы лучше разобраться с этими правилами, в этом уроке вы изучите большое количество примеров. Вы также будете активно использовать сопоставление с образцом для максимально простого рекурсивного решения задач.

8.2. Рекурсия на списках

В 6-м уроке мы говорили о важности списков в функциональном программировании и обсудили некоторые облегчающие их обработку функции, включённые в стандартный модуль `Prelude`. Сейчас вы ещё раз рассмотрите некоторые из них, но на этот раз вы должны будете самостоятельно их реализовать. Это упражнение должно помочь вам научиться думать рекурсивно при решении реальных задач, а также получить более глубокое понимание устройства некоторых из базовых функций в Haskell.

8.2.1. Реализация функции length

Вычисление длины списка — один из самых простых и прямолинейных примеров рекурсивных функций для работы со списками. С помощью сопоставления с образцом нашу задачу можно легко разобрать на части.

Наша цель — получение пустого списка (правило 1). Большинство рекурсивных функций для списков имеют пустой список в качестве конечной цели. Что должно произойти при достижении этой цели (правило 2)? Как известно, длина пустого списка равна 0, потому что в нём ничего нет. Таким образом, мы описали цель:

```
myLength [] = 0
```

Затем нужно рассмотреть различные возможные случаи (правило 3). Есть всего один вариант: непустой список. Если список не пуст, он точно содержит один элемент. Чтобы получить длину непустого списка, нужно прибавить 1 к длине хвоста (`tail`) этого списка (правило 4):

```
myLength xs = 1 + myLength (tail xs)
```

Перед тем как заявить, что задача решена, вы должны подумать, приближает ли описанный шаг к цели (правило 5). Понятно, что если вы будете продолжать брать хвост (конечного) списка, вы, в конце концов, достигнете `[]`. Других возможностей здесь нет, а каждое состояние, не являющееся целью, приближает вас к цели — дело сделано!

Листинг 8.1 Функция myLength

```
myLength [] = 0
myLength xs = 1 + myLength (tail xs)
```

Проверка 8.1. Перепишите функцию `myLength` с помощью сопоставления с образцом и без явного вызова функции `tail`.

Ответ 8.1

```
myLength [] = 0
myLength (x:xs) = 1 + myLength xs
```

8.2.2. Реализация функции take

Функция `take` интересна по двум причинам: `take` принимает два аргумента, `n` и `список`, и оказывается, что у `take` есть две цели! Как и в большинстве случаев при работе со списками, `take` завершается на пустом списке `[]`. Как было замечено ранее, в отличие от `tail` и `head`, функция `take` без проблем работает с пустым списком и возвращает столько элементов, сколько возможно. Другое условие, при котором `take` завершается, — когда `n = 0`. В обоих конечных состояниях действия функции `take` совпадают. Если необходимы `n` элементов из пустого списка — это `[]`; 0 элементов из любого списка — так же `[]`. Код выглядит следующим образом:

```
myTake _ [] = []
myTake 0 _ = []
```

Единственный вариант, когда конечная цель не достигнута, возникает, когда одновременно `n` больше нуля и список не пуст. При реализации функции `length` вам надо было заботиться только о том, чтобы разобрать список на части, но в случае с `myTake` вам нужно ещё и вернуть список, который придётся строить на ходу. Из каких элементов состоит ваш новый список? Давайте подумаем об этом на примере: `take 3 [1, 2, 3, 4, 5]`:

(1) вам нужен первый элемент `1`, добавленный к `take 2 [2, 3, 4, 5]`;

(2) затем вам нужен следующий элемент `2`, добавленный в начало списка `take 1 [3, 4, 5]`;

(3) затем вам нужно число `3`, добавленное в начало `take 0 [4, 5]`;

(4) при 0 вы достигли цели, возвращается `[]`;

(5) в итоге получается `1:2:3:[]`, то есть, `[1, 2, 3]`.

Соответствующий код выглядит следующим образом:

```
myTake n (x:xs) = x:rest
  where rest = myTake (n - 1) xs
```

Наконец, вы задаёте вопрос: «Приближает ли рекурсивный вызов меня к цели?» В этом случае так и есть: уменьшение `n` в итоге приводит к 0, а взятие хвоста от списка в итоге приводит к `[]`.

Листинг 8.2 myTake

```
myTake _ [] = []
myTake 0 _ = []
myTake n (x:xs) = x:rest
  where rest = myTake (n - 1) xs
```

8.2.3. Реализация функции cycle

Функция `cycle` является самой интересной для реализации функцией обработки списков, причём далеко не многие языки программирования позволяют её вообще реализовать. Функция `cycle` принимает список и бесконечно его повторяет. Это возможно только благодаря ленивости вычислений, которая есть далеко не во всех языках. С точки зрения наших правил построения рекурсивных функций интересно, что функция `cycle` не имеет конечной цели. К счастью, рекурсия без конечной цели даже в Haskell является довольно редким явлением. Тем не менее разбор этого примера даёт неплохое представление о комбинировании рекурсии и ленивых вычислений.

Вам вновь нужно строить список. Давайте начнём с конечного списка. Базовое поведение функции, которое вам нужно реализовать, состоит в том, чтобы вернуть переданный в качестве аргумента список с первым элементом, добавленным в его конец:

```
finiteCycle (first:rest) = first:rest ++ [first]
```

Функция `finiteCycle` на самом деле не создаёт цикл; она возвращает исходный список с одним добавленным в конец элементом. Чтобы получить цикл, вы должны циклически повторить поведение функции для отрезка списка `rest:[first]`.

Листинг 8.3 Функция myCycle

```
myCycle (first:rest) = first : myCycle (rest++[first])
```

Даже с нашими правилами в качестве руководства к пониманию рекурсивных функций рекурсия может быть головной болью. Ключ к решению рекурсивных задач: не торопясь, обдумать цели и порассуждать о процессе вычислений. Положительная сторона рекурсивных задач состоит в том, что их решения часто представляют собой всего лишь несколько строк кода. Попрактиковавшись, вы увидите, что схем написания рекурсивных функций на самом деле довольно мало.

8.3. Патологическая рекурсия: функция Аккермана и гипотеза Коллатца

В этом разделе вы рассмотрите две интересные математические функции, демонстрирующие некоторые пределы наших пяти правил рекурсии.

8.3.1. Функция Аккермана

Функция Аккермана принимает два аргумента: m и n. При отсылках к математическому определению функции будем в целях экономии места использовать для неё обозначение $A(m, n)$. Функция Аккермана определяется в соответствии со следующими тремя правилами:

- если $m = 0$, вернуть $n + 1$;
- если $n = 0$, вернуть $A(m - 1, 1)$;
- если $m \neq 0$ и $n \neq 0$, вернуть $A(m - 1, A(m, n - 1))$.

Теперь посмотрим, как реализовать это определение в Haskell с использованием наших правил. Во-первых, цель достигается при $m = 0$, причём функция возвращает в этом случае $n + 1$. Это легко сделать, используя сопоставление с образцом (правила 1 и 2):

```
ackermann 0 n = n + 1
```

Далее у вас есть два варианта: либо n может быть равно 0, либо обе переменные n и m могут быть ненулевыми. В определении функции сказано, что делать в этих случаях (правила 3 и 4):

```
ackermann m 0 = ackermann (m - 1) 1
ackermann m n = ackermann (m - 1) (ackermann m (n - 1))
```

И, наконец, продвигаетесь ли вы по этим двум веткам к конечной цели (правило 5)? При $n = 0$ да, так как уменьшая m, вы в итоге получите $m = 0$. Это верно и для последнего случая. Даже если вы дважды вызываете `ackermann`, первая m уменьшается до 0, в то время как во втором вызове до нуля уменьшается n, тем самым действительно приближая вас к цели!

Всё выглядит здорово до тех пор, пока вы не попробуете запустить код. Давайте загрузим функцию в GHCi и замерим время вызова функций, воспользовавшись командой `:set +s`.

```
GHCi> :set +s
GHCi> ackermann 3 3
61
(0.01 secs)
GHCi> ackermann 3 8
2045
(3.15 secs)
GHCi> ackermann 3 9
4093
(12.97 secs)
```

Из-за того, что ваш рекурсивный вызов функции делает вложенные рекурсивные вызовы к той же функции, временные затраты на выполнение кода растут подобно взрыву! Даже следуя правилам построения рекурсии, вы можете угодить в серьёзные неприятности с функцией Аккермана.

8.3.2. Гипотеза Коллатца

Гипотеза Коллатца представляет собой увлекательную математическую задачу. Она включает в себя определение рекурсивного процесса по заданному числу n:

- если `n = 1`, процесс останавливается;
- если n — чётное, процесс продолжается для `n/2`;
- если n — нечётное, процесс продолжается для `n*2 + 1`.

Давайте напишем функцию `collatz`, реализующую описанные выше вычисления. Единственная проблема с ней состоит в том, что, согласно данному выше определению, `collatz` будет всегда завершаться со значением `n = 1`. Чтобы сделать её более интересной, давайте записывать длину пути от заданного числа до 1. Так, например, при вызове `collatz 5` вы должны проделать следующий путь:

```
5 -> 16 -> 8 -> 4 -> 2 -> 1
```

В этом случае вызов `collatz 5` должен возвращать 6.

Перейдём к написанию кода. Для начала вы определяете свою цель (правило 1): это просто случай, когда `n = 1`. Что должно произойти, когда вы достигнете цели (правило 2)? Вам нужно просто вернуть единицу, так как мы считаем, что это — один шаг. Проще всего выразить этот шаг сопоставлением с образцом:

```
collatz 1 = 1
```

Теперь вы должны перечислить имеющиеся возможности (правило 3). В данном случае их две: (1) n не равно 1 и является чётным или (2) n не равно 1 и является нечётным. Чтобы отнести входные данные к первому или второму случаю, нужно произвести вычисления, а потому здесь не подойдёт сопоставление с образцом:

```
collatz n = if even n
                then ...
                else ...
```

Вы почти справились! Следующим шагом будет описание происходящего в различных случаях (правило 4). Это легко сделать, потому что описания уже есть в формулировке гипотезы. Не забывайте, что вам также нужно сохранять информацию о текущей длине пути. Получается, что нужно прибавить 1 к результату следующего вызова `collatz`.

Листинг 8.4 collatz

```
collatz 1 = 1
collatz n =
   if even n
   then 1 + collatz (n `div` 2)
   else 1 + collatz (n*3 + 1)
```

Вот и всё — функция готова. Можно немного поиграть с ней в GHCi:

```
GHCi> collatz 9
20
GHCi> collatz 999
50
GHCi> collatz 92
18
GHCi> collatz 91
93
```

Можно даже попросить GHCi вычислить множество значений разом:

```
GHCi> map collatz [100 .. 120]
[26,26,26,88,13,39,13,101,114,114,114,70,21,13,34,34,21,21,34,
↳ 34,21]
```

Кажется, вы забыли проверить, что рекурсия в каждом из случаев приближает вас к цели (правило 5). Первый из возможных случаев, когда n — чётное, не представляет проблем. Если n является чётным, вы делите его на два; продолжая так делать, вы в итоге достигнете 1. Но в случае с нечётным n выражение `n*3 + 1` не выглядит так, будто приближает вас к цели. Вполне возможно, что такое увеличение нечётных чисел в комбинации с уменьшением чётных будет всегда приводить к единице. К несчастью, вы не знаете этого наверняка. Никто не знает! Гипотеза Коллатца состоит в том, что ваша функция `collatz` всегда завершается, но доказательства истинности этого утверждения нет. Если вы встретите число, на котором GHCi зависнет, попытавшись вычислить значение функции `collatz`, запишите его; возможно, это приведёт к важному математическому открытию!

Функция `collatz` интересным образом нарушает наши правила. Это не обязательно означает, что эту функцию нужно выбросить. В принципе,

вы можете протестировать её на большом диапазоне значений (рис. 8.1), и тогда эту функцию можно будет спокойно использовать в разработке программного обеспечения. Тем не менее важно отмечать нарушение 5-го правила, так как это нарушение может быть очень опасным и приводить к никогда не завершающимся функциям.

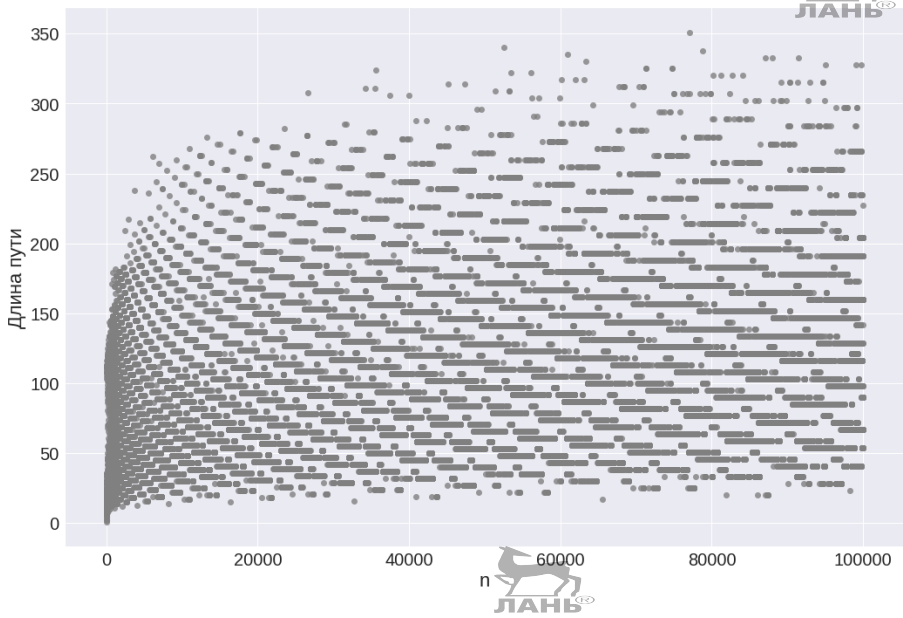


Рис. 8.1: Гипотеза Коллатца: длины путей до достижения единицы

Итоги

В этом уроке нашей целью было закрепление правил рекурсии, изученных вами в предыдущем уроке. С практикой и удержанием в голове правил рекурсии написание рекурсивного кода становится гораздо более естественным. Вы также узнали, что существуют граничные случаи: ваш код может удовлетворять правилам рекурсии, но всё же быть небезопасным при запуске, а может не подходить под правила рекурсии, но быть вполне применимым для практических целей. Давайте проверим, как вы разобрались с этим материалом.

Задача 8.1. Напишите собственную реализацию функции `reverse`, инвертирующей список.

Задача 8.2. Вычисление чисел Фибоначчи — возможно, самая распространённая задача для написания рекурсивной функции. Самое прямолинейное определение выглядит следующим образом:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Как и в случае с функцией Аккермана, время работы этой реализации очень быстро возрастает в связи с повторяющимися рекурсивными вызовами. Однако, в отличие от функции Аккермана, существует гораздо более эффективный способ вычисления n-го числа Фибоначчи. Напишите функцию `fastFib`, вычисляющую 1000-е число Фибоначчи практически мгновенно. Подсказка: `fastFib` принимает три аргумента: `n1`, `n2` и `counter`. Чтобы вычислить 1000-е число Фибоначчи, вы должны вызвать `fastFib 1 1 1000`, а для 5-го — `fastFib 1 1 5`.

9

Функции высшего порядка

После прочтения урока 9 вы:

- поймёте идею функций высшего порядка;
- научитесь использовать функции `map`, `filter` и `foldl`, избегая явного написания рекурсивных функций;
- сможете писать функции высшего порядка самостоятельно.

В предыдущем уроке мы обсуждали широкое множество рекурсивных функций. И хотя с практикой писать рекурсивный код становится проще, многие функции пишутся по одним и тем же схемам. Поэтому возникает желание обобщить эти схемы в виде небольшого количества широко используемых функций, которые не требуют от вас даже думать о рекурсии. Практический ответ на вызов написания рекурсивного кода в том, что вы обычно используете эти уже существующие функции, их обычно называют функциями высшего порядка.

С технической точки зрения *функция высшего порядка* — это *любая функция, которая принимает другую функцию в качестве аргумента*. Обычно, когда упоминаются функции высшего порядка, на ум приходит некая их конкретная группа, и почти все они используются для обобщения основных схем рекурсивного кода. В этом уроке мы рассмотрим функции высшего порядка, которые делают написание рекурсивных функций намного проще. Настоящее излечение головных болей от рекурсии — это абстракция!

Обратите внимание. Рассмотрим функции `add3ToAll` и `mul3ByAll`, которые добавляют 3 к каждому элементу и умножают на 3 каждый элемент списка соответственно:

```
add3ToAll [] = []
add3ToAll (x:xs) = (3 + x) : add3ToAll xs

mul3ByAll [] = []
mul3ByAll (x:xs) = (3 * x) : mul3ByAll xs
```

Обе эти функции легки для написания и понимания, к тому же они имеют практически идентичную структуру. Теперь представьте функцию `squareAll`, которая возводит каждый элемент списка в квадрат. Функция `squareAll` имеет такую же базовую структуру. Вероятно, вы можете придумать бесконечно много различных функций, которые реализуют ту же схему. Можете ли вы придумать, как использовать функции как значения первого класса, чтобы переписать эти примеры, используя новую функцию, которая принимает в качестве аргументов функцию и список и которая может быть использована для определения и `add3ToAll`, и `mul3ByAll`?

9.1. Использование map

Трудно переоценить, насколько важна функция `map` для функционального программирования и для Haskell. Функция `map` принимает другую функцию и список в качестве аргументов и применяет эту функцию к каждому элементу в списке:

```
GHCi> map reverse ["собака", "кот", "лось"]
["акабос","ток","ьсол"]
GHCi> map head ["собака", "кот", "лось"]
"скл"
GHCi> map (take 4) ["тыква", "пирог", "арахисовое масло"]
["тыкв","пиро","арах"]
```

Как правило, первое впечатление у программистов состоит в том, что `map` — это просто более простая версия цикла `for`. Сравните оба эти подхода на примере добавления восклицательного знака к названиям животных в списке в JavaScript (который поддерживает и `map`, и `for`), как показано в следующем листинге:

Листинг 9.1 Пример функции map в JavaScript

```
var animals = ["собака","кот","лось"]

// цикл for
for(i = 0; i < animals.length; i++){
        animals[i] = animals[i] + "!"
}

//функция map
var addBang = function(s){return s+"!"}
animals = animals.map(addBang)
```

Даже в языке, который не требует соблюдения функционального стиля программирования так строго, как Haskell, функция `map` имеет несколько преимуществ. Для начала, так как вы передаёте именованную функцию, вы точно знаете, что в ней происходит. В этом тривиальном примере это не так важно, но тело цикла `for` может оказаться довольно сложным. Если функция хорошо названа, увидеть, что происходит в коде, гораздо проще. Вы также можете позднее захотеть изменить поведение map (например, добавляя вопросительные знаки функцией `addQuestion`), и вам всего лишь нужно будет изменить аргумент.

Читабельность кода также улучшилась, потому что `map` — это особый вид итерации. Вы знаете, что на выходе вы получите новый список такого же размера, как исходный. Преимущество может быть неочевидным, когда вы только познакомились с `map` и функциями высшего порядка на списках. Когда вы начнёте ориентироваться в приёмах функционального программирования, то будете думать именно в терминах преобразования элементов списка, отказавшись от гораздо более общей формы перебора набора значений, выражаемой циклом `for`.

9.2. Обобщение вычислений с помощью map

Основная причина применения функций как значений первого класса, а следовательно, и наличия функций высшего порядка — это возможность обобщения схем программного кода. Чтобы с этим разобраться, давайте посмотрим на то, как работает `map`. Оказывается, `map` имеет только поверхностное сходство с `for`, а под капотом не имеет с ним ничего общего. Чтобы выяснить, как работает `map`, давайте разберём две простые задачи, которые решаются с помощью `map`, после чего реализуем их так, будто `map` не существует (как и функций в роли значений первого класса, если на то пошло).

Возьмём пример с добавлением восклицательных знаков из нашего примера на JavaScript и функцию `squareAll`, которая возводит в квадрат все элементы списка. Вот что мы пытаемся сделать:

```
GHCi> map (++"!") ["поезд","самолёт","лодка"]
["поезд!","самолёт!","лодка!"]
GHCi> map (^2) [1,2,3] [1,4,9]
[1,4,9]
```

Начнём с `addBang` (добавление восклицательных знаков). Как обычно, первым делом следует себя спросить: «Какова моя конечная цель?» Так как вы идёте по списку, то закончите, когда дойдёте до `[]`. Следующий вопрос: «Что я буду делать в конце?» Вы пытаетесь добавить восклицательный знак к каждому слову в списке, и когда слов не остаётся, логично вернуть пустой список. Другая подсказка в том, что вы строите список, а значит, стоит вернуть пустой. Если ваша рекурсивная функция возвращает список, то он завершается пустым списком. Итак, для конечной цели у вас есть простое определение:

```
addBang [] = []
```

Другая возможность состоит в том, что у нас непустой список. В этом случае хочется взять голову списка и применить `addBang` ко всему, что осталось:

```
addBang (x:xs) = (x ++ "!") : addBang xs
```

Приближаемся ли мы к цели? Да, потому что берём хвост списка, что в итоге приведёт нас к пустому списку.

Функция `squareAll` следует похожей схеме. Она заканчивается на пустом списке, и единственное другое возможное значение аргумента — это непустой список. В случае непустого списка возводим голову в квадрат и продолжаем дальше:

```
squareAll [] = []
squareAll (x:xs) = x^2 : squareAll xs
```

Если вы пойдёте ещё дальше и уберёте конкатенацию и функцию возведения в квадрат, заменяя их на произвольную функцию `f`, то получите в точности определение `map`!

Листинг 9.2 Функция myMap

```
myMap f [] = []
myMap f (x:xs) = (f x) : myMap f xs
```

Если бы у нас не было функции map, то пришлось бы снова и снова писать функции в соответствии с этой схемой. Не то чтобы эта явная рекурсия слишком сложна, но часто писать и читать такое было бы гораздо менее приятно. Если рекурсия всё-таки кажется вам сложной, то хорошая новость состоит в том, что все достаточно часто используемые схемы рекурсии уже давно реализованы в виде функций. Писать рекурсивные функции на практике приходится довольно редко. Правда, поскольку все основные схемы рекурсии уже оформлены как функции высшего порядка, когда сталкиваешься с действительно сложной рекурсивной задачей, то обычно приходится серьёзно подумать.

9.3. Фильтрация списка

Другой важной функцией высшего порядка для работы со списками является filter. Функция filter выглядит и ведёт себя схоже с map, принимая функцию и список в качестве аргументов и возвращая список. Разница в том, что в filter должна передаваться функция, которая возвращает True или False. Функция filter оставляет только те элементы списка, которые проходят тест:

```
GHCi> filter even [1,2,3,4]
[2,4]
GHCi> filter (\(x:xs) -> x == 'a') ["арбуз","банан","авокадо"]
["арбуз","авокадо"]
```

Использовать filter несложно, и это полезный инструмент. Самое интересное в filter — это схема рекурсии, которую она обобщает. Как и в случае map, цель filter — пустой список. Функция filter отличается тем, что в процессе работы со списком возникают две возможности: непустой список, в котором первый элемент проходит проверку, и непустой список, в котором первый элемент её не проходит. Единственное отличие при их обработке в том, что когда тест проваливается, элемент к списку не присоединяется.

Листинг 9.3 Функция myFilter

```
myFilter test [] = []
myFilter test (x:xs) = if test x
                       then x:myFilter test xs
                       else myFilter test xs
```

Проходит ли голова списка проверку?

Если проходит, то присоединяем её к результату обработки хвоста

В противном случае просто обрабатываем хвост списка

Проверка 9.1. Реализуйте функцию `remove`, которая убирает элементы, проходящие проверку.

9.4. Свёртка списка

Функция `foldl` (`l` значит левая, скоро мы это объясним) принимает список и сводит его к одному значению. Функция принимает три аргумента: бинарную функцию, начальное значение и список. Самое простое применение `foldl` — это суммирование элементов списка:

```
GHCi> foldl (+) 0 [1,2,3,4]
10
```

Функция `foldl`, вероятно, наименее очевидная из уже рассмотренных функций высшего порядка. Она работает следующим образом: первый аргумент (бинарная функция) применяется к начальному значению и голове списка. Результат применения — это новое начальное значение. На рис. 9.1 проиллюстрирован весь процесс.

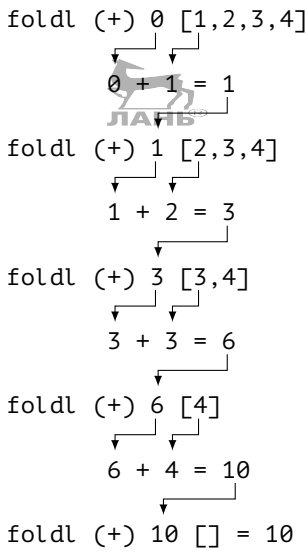


Рис. 9.1: Визуализация `foldl (+)`

Свёртка полезна, но определённо требуется практика, чтобы к ней привыкнуть. Вы можете построить функцию `concatAll`, которая соединяет все строки в списке:

```
concatAll xs = foldl (++) "" xs
```

Ответ 9.1

```
remove test [] = []
remove test (x:xs) = if test x
                     then remove test xs
                     else x:remove test xs
```

Проверка 9.2. Напишите функцию `myProduct`, вычисляющую произведение элементов числового списка.

Довольно часто `map` и `foldl` используются вместе. Например, вы можете написать `sumOfSquares`, которая возводит в квадрат каждый элемент списка, а затем считает их сумму:

```
sumOfSquares xs = foldl (+) 0 (map (^2) xs)
```

Возможно, самое интересное применение `foldl` — это инвертирование списка. Чтобы это сделать, вам понадобится аналогичная операции `(:)` функция `rcons`, но принимающая аргументы в обратном порядке.

Листинг 9.4 Функция myReverse

```
rcons xs y = y:xs
myReverse xs = foldl rcons [] xs
```

Работу этой свёртки тоже для большей ясности стоит визуализировать (см. рис. 9.2). Заметьте, что возвращаемое здесь функцией `foldl` «одно» значение — это целый список!

```
foldl rcons [] [1,2,3]
              |
              v
            1:[]
              |
              v
foldl rcons [1] [2,3]
              |
              v
            2:[1]
              |
              v
foldl rcons [2,1] [3]
              |
              v
            3:[2,1]
              |
              v
foldl rcons [3,2,1] [] = [3,2,1]
```

Рис. 9.2: Визуализация `foldl rcons`

Реализация `foldl` чуть хитрее, чем у тех функций, которые вы уже видели. Ещё раз, ваша цель — прийти к пустому списку `[]`. Но что вы должны вернуть? Так как начальное значение будет обновляться после каждого вызова бинарной функции, оно будет содержать итоговое значение вычисления. При достижении конца списка, вы возвращаете значение `init` на тот момент:

```
myFoldl f init [] = init
```

Ответ 9.2

```
myProduct xs = foldl (*) 1 xs
```

Теперь остаётся только одна возможность — непустой список. Передаёте начальное значение и голову списка бинарной функции, которая создаёт новое значение `init`. Затем вызываете `myFoldl` на оставшейся части списка с новым значением `init`.

Листинг 9.5 myFoldl

```
myFoldl f init [] = init
myFoldl f init (x:xs) = myFoldl f newInit xs
  where newInit = f init x
```

Проверка 9.3. Правда или ложь: основной шаг `myFoldl` завершается.

Остался вопрос: почему эта свёртка *левая*? Оказывается, что другой способ сворачивать список значений в одно. Альтернатива для `foldl` — это `foldr`, где `r` означает *правая*. Если вы посмотрите на определение `myFoldr`, то увидите, чем оно отличается.

Листинг 9.6 myFoldr

```
myFoldr f init [] = init
myFoldr f init (x:xs) = f x rightResult
  where rightResult = myFoldr f init xs
```

Причина, почему она называется *правой* свёрткой, — в том, что у бинарной функции есть два аргумента: левый и правый. Левая свёртка сворачивает список в левый аргумент, а правая — в правый.

Между `foldl` и `foldr` есть разница и в производительности, и в порядке вычислений. На данном этапе обучения важно знать, что эти функции дают разный результат, если важен порядок применения. Для сложения разницы нет, и эти функции ведут себя одинаково:

```
GHCi> foldl (+) 0 [1,2,3,4]
10
GHCi> foldr (+) 0 [1,2,3,4]
10
```

Ответ 9.3. Правда: так как вы всегда рекурсивно работаете с остающейся частью списка, он неминуемо уменьшается, пока не станет пустым (при условии что он не бесконечный).

Но для вычитания порядок имеет значение:

```
GHCi> foldl (-) 0 [1,2,3,4]
-10
GHCi> foldr (-) 0 [1,2,3,4]
-2
```

При изучении Haskell `foldl` для сворачивания списков предпочтительнее, потому что её поведение интуитивно понятнее. Понимание разницы между `foldl` и `foldr` — это хороший знак, что вы освоили рекурсию.

Виды свёрток

Семейство функций свёртки — несомненно, самое хитрое из представленных здесь функций высшего порядка. Существует ещё одна полезная функция свёртки `foldl'` (обратите внимание на штрих), находящаяся в модуле `Data.List`. Приведём несколько советов, помогающих решить, какую из свёрток лучше использовать в каждом конкретном случае:

- `foldl` — это самая интуитивно понятная из свёрток, но обычно у неё ужасная производительность и она не может применяться к бесконечным спискам;
- `foldl'` — это неленивая версия `foldl`, которая зачастую более эффективна;
- `foldr` обычно более эффективна, чем `foldl`, и только она работает с бесконечными списками.

При изучении Haskell нет необходимости сразу изучать все виды свёрток. Скорее всего, вы наткнётесь на проблемы с `foldl` при написании более сложного кода на Haskell.

Итоги

В этом уроке нашей целью было познакомить вас с семейством функций, которые упрощают работу с рекурсией. Многие рекурсивные задачи можно решить с помощью `map`, `filter` и `foldl`. Когда вы встречаете рекурсивную задачу, первый вопрос, который вы должны задать: можете ли вы решить её с помощью этих трёх функций. Давайте проверим, как вы это всё поняли.

Задача 9.1. Воспользуйтесь функциями `filter` и `length`, чтобы переопределить функцию `elem`.

Задача 9.2. Функция `isPalindrome` из урока 6 не обрабатывает предложения с заглавными буквами и пробелами. Используйте `map` и `filter`, чтобы убедиться, что предложение «А роза упала на лапу Азора» распознаётся как палиндром.

Задача 9.3. В математике гармонической последовательностью называют сумму вида $1/2 + 1/3 + 1/4 \ldots$. Напишите функцию `harmonic`, которая принимает аргумент `n` и вычисляет сумму этой последовательности до заданного `n`. Убедитесь, что используете ленивые вычисления.

ЛАНЬ

10

Итоговый проект: функциональное объектно-ориентированное программирование и роботы!

Итоговый проект включает в себя:

- применение приёмов функционального программирования для создания объектов;
- создание взаимодействующих друг с другом объектов;
- представление состояния объектов в функциональном стиле.

Довольно распространено заблуждение о некоторой противоположности объектно-ориентированного и функционального программирования. На самом деле это совсем не так. Многие функциональные языки программирования, включая Common Lisp, R, F#, OCaml и Scala, поддерживают некоторые формы объектно-ориентированного программирования. В этом модуле вы узнали, что функции можно использовать для выполнения любых вычислений. Поэтому вполне разумно предположить, что с помощью средств функционального языка можно запрограммировать простую объектно-ориентированную систему!

Это ваш первый итоговый проект. В ходе его реализации вы узнаете, как можно использовать функциональное программирование для воспроизведения основных элементов архитектуры ООП-языков. Вы начнёте с простого объекта «кружка», а затем займётесь моделированием боевых роботов!

Думайте как программист

В Haskell не используются объекты — так почему же вы должны тратить время на реализацию ООП с нуля? Основная причина состоит в том, что это поможет вам осознать мощь функциональных средств программирования, которые вы изучали до сих пор. Если вы поймёте, как реализовать ООП, используя замыкания, лямбда-функции и функции как значения первого класса, то по-настоящему достигнете функционального просветления.

10.1. Объект с одним свойством: кружка кофе

Для начала давайте смоделируем кружку кофе. Весь написанный в этом разделе код нужно сохранять в файле с именем cup.hs. В этом примере кружка будет иметь всего одно свойство: объём жидкости (в миллилитрах), находящейся в ней в данный момент. Вам нужен некоторый контейнер, хранящий это значение и обеспечивающий к нему доступ. Этот контейнер будет играть роль вашего простого объекта. К счастью, в уроке 5 вы открыли для себя удобное средство для хранения значений в функциях: замыкания, давайте им воспользуемся.

Вы можете определить функцию cup, которая принимает объём жидкости и возвращает замыкание, хранящее это значение:

```
cup ml = \_ -> ml
```

В связи с тем, что мы можем использовать функции как значения первого класса, вы можете обращаться со значением, хранящимся внутри замыкания, как с обычными данными. Теперь вы можете передавать эту информацию как объект. Но этого, очевидно, не достаточно — ведь тот факт, что вы сохранили объём, не даёт вам возможности делать что-то интересное. Что вам нужно, так это возможность передавать сообщения этому внутреннему значению кружки.

Чтобы передать сообщение своему объекту, вы будете использовать функции как значения первого класса. Передаваемое сообщение может каким-либо образом воздействовать на внутреннее свойство объекта.

Обратите внимание: вы будете использовать подход к передаче сообщений объекту, несколько отличающийся от традиционного способа вызо-

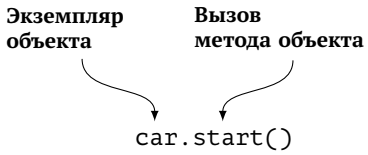


Рис. 10.1: Подход к ООП, основанный на вызове методов

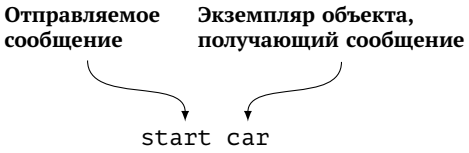


Рис. 10.2: Подход к ООП, основанный на передаче сообщений (часто используется в функциональных языках программирования)

ва методов объектов. Вызов метода обычно имеет вид «объект.действие», как представлено на рис. 10.1.

Ваш подход будет представлять собой обратную идею — отправку сообщения объекту, как показано на рис. 10.2. Эта менее распространённая форма записи используется в Common Lisp Object System (CLOS) и в объектной системе S3 языка R.

10.1.1. Создание конструктора

Самый распространённый способ создания экземпляра объекта — вызов специального метода под названием *конструктор*. Для создания конструктора своего объекта вам нужно будет сделать только одно — предоставить способ отправки ему сообщений. Указав в своём замыкании один именованный аргумент, вы можете добавить возможность передавать сообщения объекту.

Листинг 10.1 Конструктор для базового объекта cup

```
cup ml = \message -> message ml
```

Теперь у вас есть простой конструктор, который может создавать экземпляры объекта. Обратите внимание: вы сделали это с помощью лямбда-функции, замыкания и функций как значений первого класса! Вы можете создать экземпляр своего объекта `cup` в GHCi:

```
GHCi> aCup = cup 180
```

Вы даже можете определить в файле cup.hs пол-литровую кружку кофе!

Листинг 10.2 Объект coffeeCup

```
coffeeCup = cup 500
```

10.1.2. Добавление к объекту функций-аксессоров

Вы сохранили своё значение в объекте, но теперь вам нужно, чтобы с этим объектом можно было делать что-то полезное. Давайте определим простые сообщения для получения и установки значений в объекте. В первую очередь вам нужно иметь возможность получить текущий объём кофе в кружке. Для этого нужно создать сообщение `getMl`, которое принимает объект `cup` и возвращает текущий объём жидкости в кружке.

Листинг 10.3 Сообщение getMl

```
getMl aCup = aCup (\ml -> ml)
```

Чтобы использовать эту функцию, нужно передать это сообщение в объект:

```
GHCi> getMl coffeeCup
500
```

Теперь вам нужно кое-что посложнее. Самая полезная вещь, которую можно сделать с кружкой, — выпить из неё! Выпивание жидкости из кружки меняет состояние объекта. Но как вы собираетесь сделать это в Haskell?! Легко: достаточно создать за кадром новый объект. Ваше сообщение для установки объёма жидкости должно вернуть новый объект с соответствующим образом изменённым внутренним состоянием.

Листинг 10.4 Обновление состояния сообщением drink

```
drink aCup mlDrank = cup (ml - mlDrank)
  where ml = getMl aCup
```

Теперь можно отхлебнуть кофе прямо в GHCi:

```
GHCi> afterASip = drink coffeeCup 30
GHCi> getMl afterASip
470
GHCi> afterTwoSips = drink afterASip 30
GHCi> getMl afterTwoSips
440
```

```
GHCi> afterGulp = drink afterTwoSips 120
GHCi> getMl afterGulp
320
```

В этом определении есть небольшая ошибка: у вас есть возможность выпить больше кофе, чем может быть в кружке. Проблема состоит в том, что ваше сообщение `drink` допускает получение в кружке отрицательного объёма жидкости. К счастью, можно переписать `drink` так, чтобы минимальный объём кофе в кружке составлял 0 миллилитров.

ЛАНЬ

Листинг 10.5 Улучшение определения drink

```
drink aCup mlDrank = if mlDiff >= 0
                     then cup mlDiff
                     else cup 0
  where ml = getMl aCup
        mlDiff = ml - mlDrank
```

С этим улучшением ваша кружка никогда не попадёт в кофейный долг:

```
GHCi> afterBigGulp = drink coffeeCup 1000
GHCi> getMl afterBigGulp
0
```

Следует добавить ещё одно вспомогательное сообщение для проверки, пуста ли кружка.

ЛАНЬ

Листинг 10.6 Определение isEmpty

```
isEmpty aCup = getMl aCup == 0
```

Из-за необходимости постоянно отслеживать изменения в состоянии объекта слишком большое количество глотков из кружки может сделать ваш код слишком объёмным. К счастью, вам на помощь приходит функция `foldl`. В уроке 9 мы обсудили, что `foldl` является функцией высшего порядка, принимающей функцию, начальное значение и список, которая сворачивает их в одно значение. Ниже приведён пример использования `foldl`, моделирующий пять глотков из кружки.

Листинг 10.7 Моделирование множества глотков с помощью foldl

```
afterManySips = foldl drink coffeeCup [30, 30, 30, 30, 30]
```

Можно убедиться, что это работает:

```
GHCi> getMl afterManySips
350
```

10.2. Более сложные объекты: создаём боевых роботов!

До сих пор вы моделировали базовые составляющие объекта. Вы научились сохранять информацию об объекте при помощи конструктора. Затем освоили взаимодействие с объектом с помощью функций-аксессоров. Владея основами представления объектов, вы можете создать кое-что поинтереснее. Давайте соберём несколько боевых роботов!

У вашего робота будет несколько основных свойств:

- имя;
- сила атаки;
- количество урона, которое робот способен выдержать.

Для работы с этими свойствами вам потребуется кое-что более сложное, чем то, что вы использовали раньше. Вы можете передавать три значения в замыкание, но такой подход сделает работу со свойствами запутанной. Вместо этого лучше использовать кортеж значений, представляющих свойства вашего робота. Например, `("Боб", 10, 100)` — робот по имени Боб, имеющий силу атаки 10 и способный выдержать 100 очков повреждений.

Вместо того чтобы отправлять сообщение единственному значению, вы будете отправлять сообщение этой коллекции атрибутов. Обратите внимание: для облегчения доступа к значениям будет использовано сопоставление с образцом.

Листинг 10.8 Конструктор robot

```
robot (name, attack, hp) = \message ->
                               message (name, attack, hp)
```

О всех объектах можно думать как о коллекции атрибутов, которой передаются сообщения. В следующем модуле вы изучите систему типов в Haskell, предоставляющую гораздо более широкие возможности абстрагирования от данных. Впрочем, даже после этого идея использования кортежа в качестве простейшей подходящей структуры данных сохранится.

Вы можете создать экземпляр робота следующим способом:

```
killerRobot = robot ("убийца", 25, 200)
```

Чтобы этот объект был полезным, вам стоит добавить несколько функций-аксессоров для облегчения работы с его свойствами. Для начала со-

здайте вспомогательную функцию, которая позволит легко получать доступ к разным компонентам кортежа по их именам. Первые две функции будут работать так же, как `fst` и `snd` для пары значений (они использовались в уроке 4).

Листинг 10.9 Вспомогательные функции name, attack и hp

```
name (n, _, _) = n
attack (_, a, _) = a
hp (_, _, hp) = hp
```

С помощью этих функций реализовать геттеры будет несложно.

Листинг 10.10 Функции-аксессоры getName, getAttack и getHP

```
getName aRobot = aRobot name
getAttack aRobot = aRobot attack
getHP aRobot = aRobot hp
```

Благодаря функциям-аксессорам вам больше не нужно помнить о порядке расположения значений в кортеже:

```
GHCi> getAttack killerRobot
25
GHCi> getHP killerRobot
200
```

В связи с тем, что в этот раз ваш объект более сложен, вам нужно написать функции-сеттеры, позволяющие устанавливать значения свойств объекта. В каждом случае нужно будет возвращать новый экземпляр вашего робота.

Листинг 10.11 Функции setName, setAttack и setHp

```
setName aRobot newName = aRobot (\(n, a, h) ->
                                   robot (newName, a, h))

setAttack aRobot newAttack = aRobot (\(n, a, h) ->
                                      robot (n, newAttack, h))

setHP aRobot newHP = aRobot (\(n, a, h) ->
                               robot (n, a, newHP))
```

Обратите внимание: поскольку состояние объекта никогда не меняется, то вы теперь можете не только устанавливать значения, но и эмулировать поведение объектно-ориентированного программирования, основанного на прототипах.

ООП на прототипах

Объектно-ориентированные языки программирования, основанные на прототипах, такие как JavaScript, позволяют создавать новые экземпляры объектов, просто изменяя объект-прототип, понятие класса при этом не используется. Прототипы в JavaScript часто являются источником недопонимания. В данном случае, как вы могли увидеть, клонирование и изменение объектов для создания новых являются естественным результатом применения приёмов функционального программирования. В Haskell вы можете создавать новые объекты посредством изменения копий старых, уже существующих:

```
nicerRobot = setName killerRobot "котёнок"
gentlerRobot = setAttack killerRobot 5
softerRobot = setHP killerRobot 50
```

Было бы удобно иметь функцию для вывода на экран информации о роботе. Давайте определим сообщение `printRobot`, которое будет работать как метод `toString` в других языках.

Листинг 10.12 Определение сообщения printRobot

```
printRobot aRobot = aRobot (\(n, a, h) ->
                              n ++
                              " атака:" ++ (show a) ++
                              " здоровье:" ++ (show h))
```

Это облегчит изучение внутренностей ваших объектов в GHCi:

```
GHCi> printRobot killerRobot
"убийца атака:25 здоровье:200"
GHCi> printRobot nicerRobot
"котёнок атака:25 здоровье:200"
GHCi> printRobot gentlerRobot
"убийца атака:5 здоровье:200"
GHCi> printRobot softerRobot
"убийца атака:25 здоровье:50"
```

10.2.1. Передача сообщений между объектами

Самое интересное в моделировании боевых роботов— это битвы! Первым делом нужно научиться отправлять роботу сообщение о нанесении урона. Оно будет работать так же, как сообщение drink в примере с кружкой

(листинги 10.4 и 10.5). В данном случае вам нужно получить все атрибуты робота, а не только `ml`.

Листинг 10.13 Реализация функции damage

```
damage aRobot attackDamage =
         aRobot (\(n, a, h) -> robot (n, a, h - attackDamage))
```

С помощью сообщения `damage` вы можете сказать роботу, что ему нанесён урон:

```
GHCi> afterHit = damage killerRobot 90
GHCi> getHP afterHit
110
```

Пришло время сражаться! Сейчас вы впервые реализуете взаимодействие объектов друг с другом, то есть по-настоящему займётесь ООП. Ваше сообщение `fight` будет эквивалентно следующему в стиле ООП:

```
robotOne.fight(robotTwo)
```

Сообщение `fight` будет передавать урон от атакующего к защищающемуся; дополнительно вы должны запретить атаковать роботам, не имеющим жизни:

Листинг 10.14 Определение fight

```
fight aRobot defender = damage defender attack
  where attack = if getHP aRobot > 10
                 then getAttack aRobot
                 else 0
```

Теперь вам нужен оппонент для битвы с `killerRobot`:

```
gentleGiant = robot ("Мистер Дружелюбный", 10, 300)
```

Давайте устроим бой в три раунда:

```
gentleGiantRound1 = fight killerRobot gentleGiant
killerRobotRound1 = fight gentleGiant killerRobot
gentleGiantRound2 = fight killerRobotRound1 gentleGiantRound1
killerRobotRound2 = fight gentleGiantRound1 killerRobotRound1
gentleGiantRound3 = fight killerRobotRound2 gentleGiantRound2
killerRobotRound3 = fight gentleGiantRound2 killerRobotRound2
```

После окончания боя вы можете посмотреть, как они справились:

```
GHCi> printRobot gentleGiantRound3
"Мистер Дружелюбный атака:10 здоровье:225"
GHCi> printRobot killerRobotRound3
"убийца атака:25 здоровье:170"
```

10.3. Почему важно программировать без состояний

На данный момент вам удалось создать некоторое разумное приближение объектно-ориентированной системы. Конечно, пришлось проделать некоторую часть работы по явному сохранению состояния объектов вручную. Хоть это и работает, разве не проще было бы для решения этих проблем просто иметь изменяемое состояние? Скрытое состояние сделало бы этот код чище, но в связи с наличием состояния могут возникнуть некоторые серьёзные проблемы. Давайте рассмотрим другое сражение, чтобы оценить издержки, связанные со скрытым состоянием:

```
fastRobot = robot ("быстрый", 15, 40)
slowRobot = robot ("тормоз", 20, 30)
```

Теперь устроим ещё один бой из трёх раундов.

Листинг 10.15 Трёхраундовый бой с одновременными атаками

```
fastRobotRound1 = fight slowRobot fastRobot
slowRobotRound1 = fight fastRobot slowRobot
fastRobotRound2 = fight slowRobotRound1 fastRobotRound1
slowRobotRound2 = fight fastRobotRound1 slowRobotRound1
fastRobotRound3 = fight slowRobotRound2 fastRobotRound2
slowRobotRound3 = fight fastRobotRound2 slowRobotRound2
```

Вы можете проверить результаты в GHCi:

```
GHCi> printRobot fastRobotRound3
"быстрый атака:15 здоровье:0"
GHCi> printRobot slowRobotRound3
"тормоз атака:20 здоровье:0"
```

Кто должен победить? Из-за нашего способа изменения значений роботы атаковали в один и тот же момент. Судя по именам роботов, желаемым результатом должна быть победа быстрого робота. Быстрый робот должен нанести смертельный удар до того, как успеет ударить медленный, а медленный, таким образом, не должен иметь возможности ударить.

Вы можете легко это исправить, так как имеете полный контроль над изменением состояния.

Листинг 10.16 Изменение приоритета атак

```
slowRobotRound1 = fight fastRobot slowRobot
fastRobotRound1 = fight slowRobotRound1 fastRobot
fastRobotRound2 = fight fastRobotRound1 slowRobotRound1
fastRobotRound2 = fight slowRobotRound2 fastRobotRound1
fastRobotRound2 = fight fastRobotRound2 slowRobotRound2
fastRobotRound2 = fight slowRobotRound3 fastRobotRound2
```

В этом примере вы можете убедиться, что атакующая версия медленного робота обновлена после того, как быстрый робот нанёс удар:

```
GHCi> printRobot fastRobotRound3
"быстрый атака:15 здоровье:20"
GHCi> printRobot slowRobotRound3
"тормоз атака:20 здоровье:-15"
```

Как и ожидалось, быстрый робот одержал победу в матче.

В связи с тем, что в функциональном программировании у вас нет состояния, вы имеете полный контроль над тем, как происходит вычисление. Сравните это с ООП, использующим состояния. Ниже приведён один раунд с объектами, хранящими состояния:

```
fastRobot.fight(slowRobot)
slowRobot.fight(fastRobot)
```

Но что, если ваш код исполняется в другом порядке, например:

```
slowRobot.fight(fastRobot)
fastRobot.fight(slowRobot)
```

Результат получится совершенно другим!

В случае с последовательно исполняемым кодом это совсем не проблема. Но представьте, что вы используете асинхронный, конкурентный или параллельный код. В этом случае вы можете не иметь контроля над порядком исполнения операций! Более того, управление приоритетом сражений оказалось бы гораздо более сложным, если бы вы захотели гарантировать, что `fastRobot` всегда бьёт первым.

В качестве мысленного упражнения подумайте, как сделать так, чтобы `fastRobot` всегда бил первым `slowRobot`-а, даже если вы не знаете, какой код будет выполнен раньше: `fastRobot.fight` или `slowRobot.fight`.

Теперь представьте, какое количество дополнительного кода вам понадобится, чтобы решить эту задачу в случае с тремя раундами, если возможно, что код 3-го раунда будет выполнен раньше, чем код первого или второго. Если вам уже приходилось писать низкоуровневый параллельный код, вы, наверное, представляете себе сложность управления состоянием в этом окружении. Хотите — верьте, хотите — нет, в Haskell решена проблема происхождения 3-го раунда перед 2-м. Вы удивитесь, но в Haskell не важен порядок следования этих функций! Можно перемешать строки приведённого выше кода и получить точно такой же результат!

Листинг 10.17 Порядок в коде на Hakell значения не имеет

```
fastRobotRound3 = fight slowRobotRound3 fastRobotRound2
fastRobotRound2 = fight slowRobotRound2 fastRobotRound1
fastRobotRound1 = fight slowRobotRound1 fastRobot
slowRobotRound2 = fight fastRobotRound1 slowRobotRound1
slowRobotRound3 = fight fastRobotRound2 slowRobotRound2
slowRobotRound1 = fight fastRobot slowRobot
```

Результаты в GHCi не изменятся:

```
GHCi> printRobot fastRobotRound3
"быстрый атака:15 здоровье:20"
GHCi> printRobot slowRobotRound3
"тормоз атака:20 здоровье:-15"
```

Любые ошибки, возникающие из-за порядка написания функций, происходят в Haskell гораздо реже, чем в ООП-языках. Благодаря возможности точно определять, когда и как моделируется состояние, нет никаких загадок, связанных с исполнением кода. Мы специально сделали код более подробным, чем ему стоило быть, только для того, чтобы было понятнее, насколько хорошо вы контролируете состояние. Битвы роботов могут происходить в любом порядке, а результаты от этого не изменятся.

10.4. Типы — объекты и многое другое!

Haskell не является объектно-ориентированным языком. Вся функциональность, реализованная здесь с нуля, уже существует в гораздо более мощной форме благодаря системе типов в Haskell. Многие идеи, использованные в этом разделе, возникнут снова, но вместо попыток создать объекты вы будете создавать типы. Типы в Haskell могут воспроизвести всё то по-

ведение, которое вы моделировали здесь, но их плюсом является способность компилятора Haskell «рассуждать» о типах гораздо глубже, чем о ваших самодельных объектах. В связи с возможностью проверки типов код, созданный с использованием мощной системы типов, часто оказывается более надёжным и предсказуемым. Преимущества функционального программирования, которые вы увидели на данный момент, сильно возрастают в комбинации с системой типов в Haskell.

Итоги

В ходе выполнения итогового проекта вы:

- увидели, что функциональный и объектно-ориентированный стили программирования не противопоставлены друг другу;
- реализовали объектно-ориентированную систему с помощью средств функционального программирования, изученных в этом модуле;
- использовали замыкания и лямбда-функции для создания и представления объектов;
- отправляли сообщения объектам с помощью функций как значений первого класса;
- управляли состоянием в функциональном стиле, открывая тем самым возможности точного контроля над исполнением программы.

Расширение проекта. Ниже представлены некоторые идеи простых упражнений, которые вы можете выполнить самостоятельно, чтобы расширить итоговый проект.

Во-первых, используя `map` на списке роботов, получите количество жизни каждого робота в списке.

Во-вторых, напишите функцию `threeRoundFight`, принимающую на вход двух роботов, заставляющую их драться в течение трёх раундов и возвращающую победителя. Чтобы избежать использования множества переменных для хранения состояний роботов, используйте серию вложенных лямбда-функций, перезаписывая `robotA` и `robotB`.

В-третьих, создайте список из трёх роботов. Затем создайте четвёртого робота. Используйте частичное применение функций для создания замыкания с методом fight, чтобы четвёртый робот мог подраться со всеми тремя роботами сразу с помощью `map`. Наконец, используйте `map` для получения оставшегося количества жизни остальных роботов.

Модуль 2

Введение в типы

Почти в каждом языке программирования поддерживается то или иное понятие типов. Типы важны, потому что они определяют виды вычислений, допустимых на определённых данных. Возьмём, например, текст `"привет"` и число 6. Допустим, вы хотите их сложить:

```
"привет"+6
```

Даже тот, у кого нет опыта программирования, найдёт этот вопрос интересным, потому что не понятно, как это сделать. Два очевидных ответа:

- сгенерировать ошибку;
- соединить эти значения самым разумным образом: `"привет6"`.

Чтобы прийти к любому из вариантов, вам необходим способ отслеживать типы ваших данных, а также тип ожидаемого результата вычислений. Обычно мы говорим, что значение `"привет"` является строкой (тип `String`), а значение 6 — целое число (`Int`).

Такие языки программирования, как Ruby, Python или JavaScript, используют *динамическую типизацию*. При динамической типизации все решения вроде тех, что мы должны были принять в случае с `"привет"` и 6, принимаются во время работы программы. Преимущество динамической типизации для программиста — это в основном большая гибкость языка программирования и отсутствие необходимости вручную следить за типами. Опасность динамической типизации в ошибках, которые происходят во время работы программы. Например, у вас есть следующее выражение на Python:

```
def call_on_xmas():
    "санта получает "+10
```

Этот код приведёт к ошибке, так как Python требует, чтобы число 10 было приведено к строковому типу перед прибавлением к строковому литералу. Однако, как вы можете догадаться, эта функция не будет вызываться до Рождества! Если данная ошибка проберётся в систему, это будет означать ужасное Рождество с исправлением необычной проблемы. Решение в том, чтобы заранее организовать обширное модульное тестирование и гарантировать, что такие ошибки не могут проскользнуть. Это несколько снижает преимущество отсутствия необходимости явно указывать типы при написании кода.

В языках вроде Java, C++ и C# используется статическая типизация. Со статической типизацией проблемы вроде `"привет"+6` решаются на этапе компиляции. Если возникает ошибка, программа не компилируется. Очевидное преимущество статической типизации в том, что целый класс ошибок не может попасть в работающие программы. Недостаток обычно в том, что языки со статической типизацией требуют, чтобы программист собственноручно указывал множество типов. Типовые аннотации необходимы для каждой функции и метода, и у всех переменных должен быть объявлен тип.

Haskell — это язык со статической типизацией, но он определённо не похож на другие языки со статической типизацией, которые вы уже видели. Все ваши ранее написанные функции и переменные вообще не имели указаний относительно типов, потому что Haskell активно использует *вывод типов*. Компилятор Haskell умён, и он в состоянии разобраться, какие типы вы используете, на основании того, с какими аргументами и как вызываются функции.

Система типов Haskell невероятно мощна и по меньшей мере играет ту же роль в обеспечении уникального места Haskell, что и его приверженность чистому функциональному программированию. Вы приступаете к изучению способов моделирования данных и определения своих типов и классов типов.

11

Основы системы типов

После прочтения урока 11 вы:

- разберётесь с самыми простыми типами языка Haskell, включая `Int`, `String` и `Double`;
- научитесь читать типовые аннотации функций;
- сможете использовать простые типовые переменные.

Этот урок познакомит вас с одним из мощнейших аспектов Haskell — его системой типов. Основы функционального программирования, рассмотренные в предыдущих уроках, являются общими для всех функциональных языков от Lisp до Scala. А вот система типов Haskell выделяет его на фоне других языков программирования. Наше введение в эту тематику начнётся в данном уроке с самых базовых понятий.

Обратите внимание. Вам требуется создать простую функцию для вычисления среднего арифметического списка чисел. Самое очевидное решение — посчитать сумму всех элементов списка и разделить её на количество элементов этого списка:

```
myAverage aList = sum aList / length aList
```

Но эта простая реализация не будет работать. Как исправить функцию для вычисления среднего арифметического числового списка?

11.1. Типы в Haskell

Вас это может удивить, но Haskell — статически типизированный язык. Другие распространённые статически типизированные языки: C++, C#, Java. В них и в большинстве других языков со статической типизацией программисты обременены необходимостью следить за типовыми аннотациями. До сих пор вам не требовалось записывать какую-либо информацию о типах функций. Оказывается, что Haskell делает это за вас! Haskell с помощью механизма *вывода типов* автоматически определяет типы всех значений во время компиляции, исходя из того, как эти значения используются. Но вы не обязаны полагаться на Haskell при определении типов. На рис. 11.1 показана переменная, которой мы явно указываем тип `Int`.

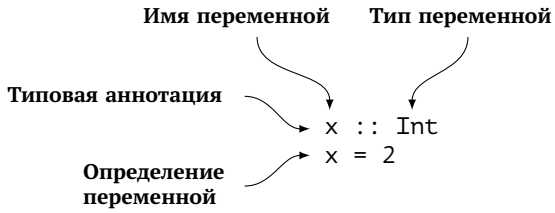


Рис. 11.1: Типовая аннотация

Имена всех типов в Haskell начинаются с заглавной буквы, для того чтобы отличать их от функций (их имена, в свою очередь, начинаются со строчной буквы). Тип `Int`, пожалуй, является одним из самых обычных и широко распространённых типов в программировании. Тип `Int` отражает способ, которым компьютер интерпретирует числа, представленные ограниченным количеством бит (обычно с помощью 32 или 64 бит). Так как эти числа хранятся с помощью фиксированного количества бит, они ограничены некими предельными значениями, которые они могут принимать. Например, если вы загрузите код с определением `x` в GHCi, то сможете проделать несколько простых операций, демонстрирующих ограничения этого типа:

```
x :: Int
x = 2

GHCi> x*2000
4000
GHCi> x^2000
0
```

Как видите, Haskell обрабатывает выход за границы `Int`, возвращая 0. Типы, обладающие свойством иметь максимальное и минимальное значения, называются *ограниченными* (*bounded*). Мы коснёмся ограниченных типов в уроке 13.

Тип `Int` — пример того, как типы обычно трактуются в традиционных языках программирования. Он выступает в качестве метки, которая сообщает компьютеру, как читать и воспринимать некую область памяти. Типы в Haskell несут больше смысла. Они предоставляют информацию о том, как значения этого типа могут вести себя и как организованы данные. Хорошим примером того, как в Haskell следует относиться к типам, является тип `Integer`. Давайте посмотрим, как определить новую переменную `y` типа `Integer`.

Листинг 11.1 Тип Integer

```
y :: Integer
y = 2
```

Повторив предыдущие вычисления, вы можете заметить отличие от `Int`:

```
GHCi> y*2000
4000
GHCi> y^2000
114813069527425452423283320117768198402231770208869520047764273682576626139237031385665948631650626991844596463898746277344711896086305533142593135616665318539129989145312280000688779148240044871428926990063486244781615463646388363947317026040466353970904996558162398808944629605623311649536164221970332681344168908984458505602379484807914058900934776500429002716706625830522008132236281291761267883317206598995396418127021779858404042159853183251540889433902091920554957783589672039160081957216630582755380425583726015528348786419432054508915275783882625175435528800822842770817965453762184851149029376
```

Как вы видите, `Integer` лучше соответствует тому, что называется целыми числами в математике. В отличие от `Int`, тип `Integer` не ограничен какими-то предельными значениями, зависящими от способа хранения.

Haskell поддерживает все типы, привычные вам по другим языкам программирования. Вот несколько примеров.

Листинг 11.2 Традиционные типы Char, Double и Bool

```
letter :: Char
letter = 'a'
```

```
interestRate :: Double
interestRate = 0.375

isFun :: Bool
isFun = True
```

Другой важный пример — тип списка.

Листинг 11.3 Тип списка

```
values :: [Int]
values = [1,2,3]

testScores :: [Double]
testScores = [0.99,0.7,0.8]

letters :: [Char]
letters = ['a','b','c']
```

Список символов — то же, что и обычная строка:

```
GHCi> letters == "abc"
True
```

Чтобы упростить работу, в Haskell есть синоним типа `[Char]` — тип `String`. Соответствующие типовые аннотации обозначают абсолютно одинаковые вещи:

```
aPet :: [Char]
aPet = "кошка"

anotherPet :: String
anotherPet = "собака"
```

Другой важный тип — кортеж. Вы успели немного с ним познакомиться в уроке 4. Пока вы не задумывались о типах, кортежи не сильно отличались от списков, но на самом деле они чуть сложнее. Два основных различия: кортежи имеют фиксированную длину, а также могут состоять из значений разных типов. Например, `[Char]` является строкой произвольной длины, а `(Char)` — кортеж, содержащий ровно *один* символ. Вот несколько примеров работы с кортежами.

Листинг 11.4 Тип кортежа

```
ageAndHeight ::(Int,Int)
ageAndHeight = (34,74)
```

```
firstLastMiddle :: (String,String,Char)
firstLastMiddle = ("Оскар","Гроуш",'Д')

streetAddress :: (Int,String)
streetAddress = (123,"ул. Счастья")
```

Кортежи очень удобны при моделировании простых типов данных.

11.2. Типы функций

У функций тоже есть типовые аннотации, в которых для разделения типов аргумента и возвращаемого значения используется стрелка ->. На рис. 11.2 представлен пример типовой аннотации для функции double.

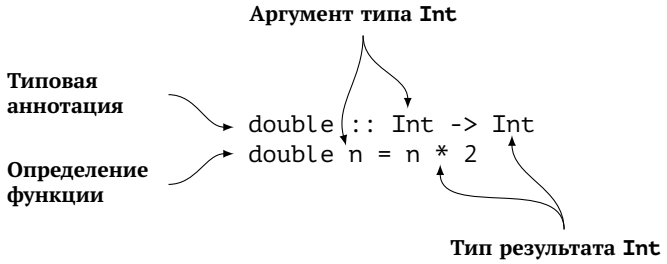


Рис. 11.2: Определение функции double с явным указанием типа

Вы могли бы легко заменить Int на Double, Integer или любой другой числовой тип. В уроке 12 мы рассмотрим классы типов, которые позволят обобщить работу с разными числовыми типами.

Вполне можно возвращать Int, если функция удваивает аргумент, однако это не так в случае деления на 2. Если вы хотите написать функцию half, которая определена на значениях типа Int, то возвращать эта функция должна Double. Выпишем тип такой функции.

Листинг 11.5 Преобразование одного типа в другой функцией half

```
half :: Int -> Double
```

Первая попытка её определить оказывается ошибочной:

```
half n = n/2          <-- Неправильно!
```

Проблема заключается в том, что вы пытаетесь применить операцию деления к значению типа Int, что, вообще говоря, в Haskell запрещено, к тому же в типе функции указано, что возвращаемое значение должно иметь

тип `Double`. Так что для начала потребуется преобразовать `Int` к `Double`. Большинство языков программирования позволяет приводить значения одного типа к значениям другого типа. Приведение позволяет принудительно поменять тип значения. Именно поэтому приведение переменных часто воспринимается как попытка катить квадратное. В Haskell нет каких-то общих правил для автоматического приведения типов, вместо этого делается ставка на функции, которые корректно преобразуют типы значений. В нашем случае вы можете использовать функцию `fromIntegral`:

```
half n = (fromIntegral n) / 2
```

Тут происходит смена типа аргумента `n` с `Int` на более общий. Проницательный читатель может поинтересоваться: «Почему не требуется использовать `fromIntegral` со значением 2?» Во многих языках программирования, если вы захотите использовать какой-то числовой литерал как значение типа `Double`, вам потребуется добавить к нему десятичный разделитель. Например, в Python и Ruby $5/2$ равно 2, а $5/2.0 = 2.5$. Haskell относится к подобным преобразованиям одновременно более строго и гибко. Строго, так как в Haskell, в отличие от тех же Python и Ruby, не существует понятия «неявное приведение типов», а гибко, так как все числовые литералы являются *полиморфными*: их тип определяется компилятором, исходя из того, как они используются. К примеру, если вы используете GHCi в качестве калькулятора, вам вряд ли потребуется заботиться о типах чисел:

```
GHCi> 5/2
2.5
```

Проверка 11.1. В Haskell есть функция `div`, которая выполняет целочисленное деление (возвращает только целые числа). Реализуйте функцию `halve`, используя `div`, и напишите её типовую аннотацию.

Ответ 11.1

```
halve :: Integer -> Integer
halve value = value `div` 2
```

11.2.1. Функции преобразования к строкам и из строк

Наверное, самыми распространёнными преобразованиями типов являются функции, которые переводят некие значения в строки и наоборот. В Haskell для этого есть две полезные функции: `show` и `read`. В уроках 12 и 13 рассказывается о том, как они устроены, а сейчас давайте просто посмотрим на несколько примеров. Функция `show` работает достаточно предсказуемо:

```
GHCi> show 6
"6"
GHCi> show 'c'
"'c'"
GHCi> show 6.0
"6.0"
```

Проверка 11.2. Реализуйте функцию `printDouble`, которая принимает `Int` и возвращает удвоенное число в виде строки.

Функция `read` преобразует строку к значению другого типа, но действует хитрее, чем `show`. Например, как Haskell без типовой аннотации может понять, что делать в такой ситуации?

```
z = read "6"
```

Невозможно определить, требуется ли считать из строки `Int`, `Integer` или `Double`. А если уж мы не можем понять, что делать, то Haskell тем более не справится с такой задачей. В данном случае механизм вывода типов не поможет, так что придётся прибегнуть к другим способам. Например, если переменная `z` где-то используется, то Haskell, скорее всего, сможет определить, как к ней относиться:

```
q = z / 2
```

Ответ 11.2

```
printDouble :: Int -> String
printDouble value = show (value*2)
```

Теперь компилятор может воспринимать z как Double, даже если в строковом представлении не было дробной части. Другим решением является явное указание типа.

Листинг 11.6 Пример чтения значения из строки

```
anotherNumber :: Int
anotherNumber = read "6"
```

Хотя в первом модуле мы и не пользовались типовыми аннотациями, обычно всё-таки следует их указывать, так как они здорово помогают при рассуждениях о том, что именно вы пишите. Эти небольшие подсказки помогают компилятору распознавать, что требуется от read, а вам они помогут понятнее описывать свои идеи. Существует ещё один распространённый способ показать компилятору, какой тип вам нужен, — вы можете явно указать, какой тип должна возвращать функция. Чаще всего это будет требоваться при работе с GHCi, но иногда данный метод поможет избавиться от двусмысленности в коде:

```
GHCi> read "6" :: Int
6
GHCi> read "6" :: Double
6.0
```

11.2.2. Функции нескольких аргументов

До сих пор многие встречавшиеся вам типовые аннотации были достаточно простыми. При освоении Haskell у новичков часто бывают трудности с типами функций нескольких аргументов. Предположим, вы хотите написать функцию, которая принимает номер дома, улицу и название города, а возвращает кортеж, содержащий полный адрес. Тип и определение такой функции представлены на рис. 11.3.

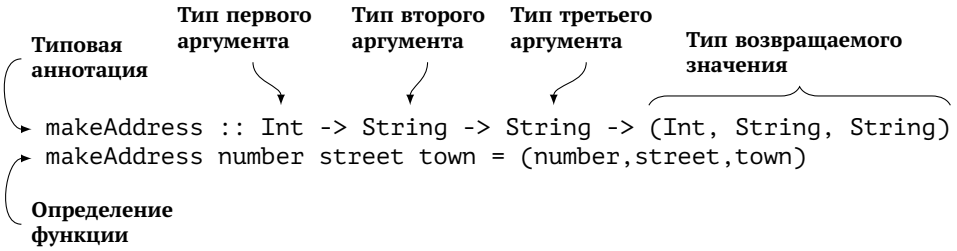


Рис. 11.3: Типовая аннотация и определение функции makeAddress

Наибольшую путаницу вносит то, что нет чёткого разделения между типами аргументов и типом возвращаемого значения. Простой способ запомнить порядок типов: последний отражает то, что функция возвращает. Возникает вопрос: почему типовые аннотации выглядят именно так? Причина скрывается за кулисами: на самом деле все функции в Haskell принимают только один аргумент. Если переписать `makeAddress` с помощью вложенных лямбда-функций, как показано на рис. 11.4, вы увидите функцию нескольких аргументов так, как её воспринимает Haskell.

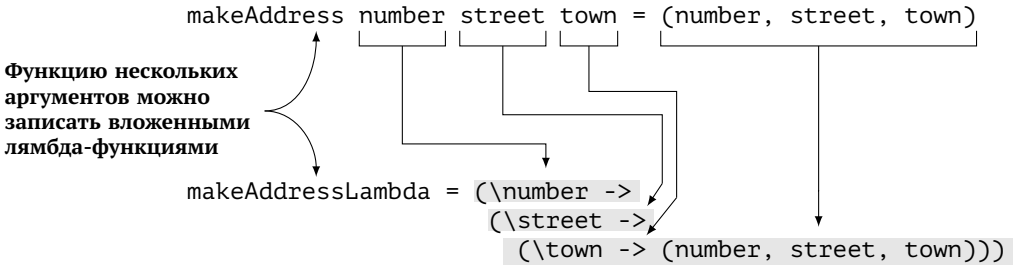


Рис. 11.4: Представление `makeAddress` в виде функции одного аргумента

Переписанную функцию можно вызвать следующим образом:

```
GHCi> (((makeAddressLambda 123) "ул. Счастья") "Хаскельтаун")
(123,"ул. Счастья","Хаскельтаун")
```

В такой записи каждая функция возвращает функцию, ожидающую следующий аргумент. Это может казаться странным, пока вы не поймёте, что именно так работает частичное применение. Вы можете вызвать функцию `makeAddress` таким же образом и получите аналогичные результаты:

```
GHCi> (((makeAddress 123) "ул. Счастья") "Хаскельтаун")
(123,"ул. Счастья","Хаскельтаун")
```

Также оказывается, что из-за специфики применения аргументов вторую версию `makeAddress` можно использовать точно так же, как и обычную функцию:

```
GHCi> makeAddressLambda 123 "ул. Счастья" "Хаскельтаун"
(123,"ул. Счастья","Хаскельтаун")
```

Надеюсь, этот пример помог вам получить представление о типовых аннотациях функций нескольких аргументов и частичном применении!

Проверка 11.3. Выпишите типы функций, полученных путём последовательных частичных применений аргументов к `makeAddress`.

11.2.3. Типы функций высшего порядка

Как мы отмечали в уроке 4, функции могут возвращать другие функции и принимать их в качестве аргументов. При записи типов таких функций тип функций-аргументов нужно заключать в скобки. К примеру, перепишем `ifEven` с типовой аннотацией:

Листинг 11.7 Типовая аннотация функции высшего порядка ifEven

```
ifEven :: (Int -> Int) -> Int -> Int
ifEven f n = if even n
             then f n
             else n
```

11.3. Типовые переменные

Мы рассмотрели несколько обычных типов и то, как они используются в функциях. Но как поступить с функцией `simple`, которая возвращает любое переданное ей значение? Эта функция может принимать аргумент любого типа. На данном этапе изучения вы, возможно, могли бы реализовать семейство функций `simple`, по одной для каждого типа.

Ответ 11.3. Начнём с исходного варианта:

```
makeAddress :: Int -> String -> String -> (Int,String,String)
```

После первого применения тип становится следующим:

```
String -> String -> (Int,String,String)
```

Подставим первую строку:

```
((makeAddress 123) "ул. Счастья")
```

Тип полученной функции:

```
String -> (Int,String,String)
```

Если же подставить все аргументы, получаем тип результата:

```
(((makeAddress 123) "ул. Счастья") "Хаскельтаун")
(Int,String,String)
```

Листинг 11.8 Функции simpleInt и simpleChar

```
simpleInt :: Int -> Int
simpleInt n = n

simpleChar :: Char -> Char
simpleChar c = c
```

Но это нелепо, и мы определённо можем сделать лучше, так как компилятор мог вывести тип функции `simple` самостоятельно. Для решения этой и других подобных проблем в Haskell существуют *типовые переменные*. Строчные буквы в типовых аннотациях обозначают позиции, в которые могут быть подставлены любые типы. Воспользуемся этой возможностью и напишем типовую аннотацию функции `simple`.

Листинг 11.9 Использование типовых переменных: функция simple

```
simple :: a -> a
simple x = x
```

Типовые переменные — действительно просто переменные, используемые для типов. Они работают абсолютно аналогично обычным переменным, но вместо значений представляют типы. Когда вы используете функции с типовыми переменными, Haskell подставляет конкретные типы вместо переменных, как показано на рис. 11.5.

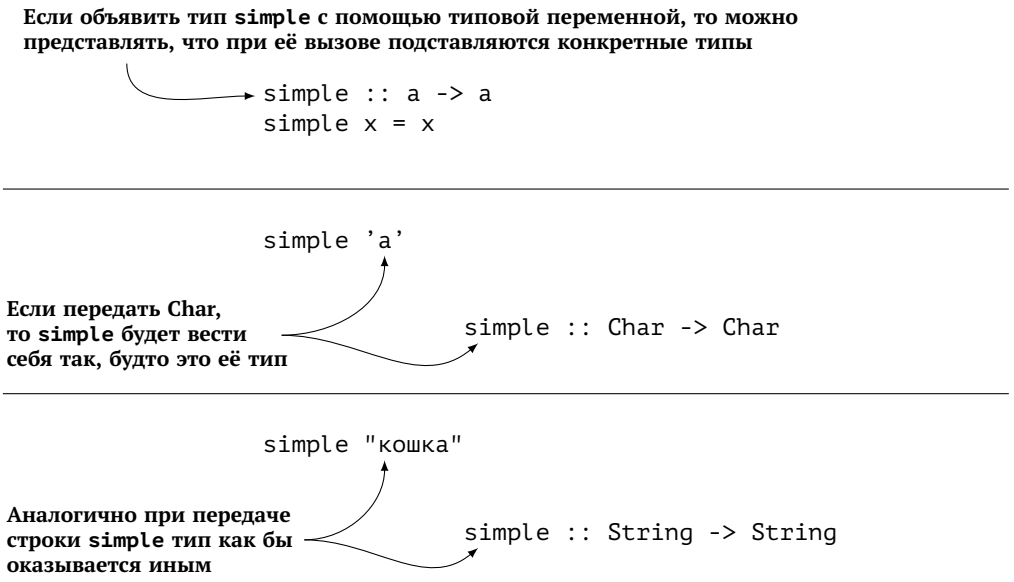


Рис. 11.5: Визуализация подстановки типов на место типовых переменных

Типовые аннотации могут содержать больше одной переменной. Однако, хотя функции и могут быть переданы всевозможные значения подставленного типа, на место одноимённых типовых переменных должны подставляться одинаковые типы. Это утверждение можно продемонстрировать на примере функции, которая принимает три аргумента разных типов, а возвращает кортеж, состоящий из трёх этих значений.

Листинг 11.10 Разные типовые переменные в типовой аннотации

```
makeTriple :: a -> b -> c -> (a,b,c)
makeTriple x y z = (x,y,z)
```

Мы использовали разные типовые переменные по той же причине, по которой для обычных переменных используют различные имена: они могут содержать разные значения. В случае `makeTriple` можно представить гипотетическую ситуацию, в которой этой функции будут переданы значения типов `String`, `Char` и снова `String`:

```
nameTriple = makeTriple "Оскар" 'Г' "Гроуш"
```

В этом примере типовую аннотацию данной функции можно представить себе следующим образом:

```
makeTriple :: String -> Char -> String -> (String,Char,String)
```

Обратите внимание на то, что определения `makeTriple` и `makeAddress` почти идентичны, однако их типы различны, так как в `makeTriple` мы использовали типовые переменные, а значит, эта функция может использоваться для более широкого круга задач, нежели `makeAddress`. Например, `makeTriple` можно запросто использовать для замены `makeAddress`. Это, впрочем, не означает, что `makeAddress` — бесполезная функция, так как более конкретный тип `makeAddress` позволяет строить более уверенные предположения о том, как эта функция себя ведёт. К тому же компилятор не позволит создать адрес, в котором вместо числа типа `Int` по недоразумению использовалось значение типа `String`.

Как и в случае с обычными переменными, разные имена означают не то, что типовые переменные обязаны представлять разные типы, а то, что они могут это делать. Допустим, вы сравниваете типы двух неизвестных вам функций `f1` и `f2`:

```
f1 :: a -> a
f2 :: a -> b
```

Вы точно можете сказать, что `f2` — функция с более широким набором возможных выходных значений, а `f1` всего лишь меняет значение некого типа, сохраняя этот тип: `Int -> Int`, `Char -> Char` и так далее. Функция `f2` при этом может представлять более крупное семейство функций: `Int -> Char`, `Int -> Int`, `Int -> Bool`, `Char -> Int`, `Char -> Bool` и многие другие.

Проверка 11.4. Типовая аннотация `map` выглядит так:

```
map :: (a -> b) -> [a] -> [b]
```

Почему она не может выглядеть следующим образом?

```
map :: (a -> a) -> [a] -> [a]
```

Подсказка: попробуйте подставить в `myMap show [1,2,3,4]` конкретные типы на место типовых переменных.

Итоги

В этом уроке нашей целью было научить вас основам потрясающей системы типов Haskell. Вы увидели, что в Haskell есть множество стандартных типов, знакомых по другим языкам программирования, например `Int`, `Char`, `Bool` и `String`. Несмотря на наличие в Haskell мощной системы типов, вы смогли пройти предыдущие уроки, не указывая явно типы функций и значений, и всё это благодаря механизму вывода типов, который позволяет Haskell определять требуемые типы, исходя из того, как

Ответ 11.4. Тип `map :: (a -> a) -> [a] -> [a]` показывает, что `map` всегда возвращает значения исходного типа. В этом случае вы не сможете выполнить

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

так как `show` возвращает `String`, который никак не согласуется с типом аргумента. Настоящая сила `map` — не в обработке всех элементов списка, а в том, что эта функция может преобразовать все элементы списка в значения другого типа.

ЛАНЬ

они используются. Но, даже учитывая, что Haskell в большинстве случаев может обойтись без использования типовых аннотаций, их написание может существенно помочь при работе с кодом. С этого момента и до конца книги большая часть наших бесед будет сводиться к рассуждениям о типах. Давайте посмотрим, как вы в этом разобрались.

Задача 11.1. Каков тип функции `filter`? Как он отличается от типа `map`?

Задача 11.2. Использование функций `tail` и `head` на пустых списках приводит к ошибке. Можно реализовать версию `tail`, которая возвращает пустой список, если исходный список был пуст. Сможете ли вы реализовать подобный вариант `head`? Для ответа на этот вопрос посмотрите внимательнее на типы функций `head` и `tail`.

ЛАНЬ

Задача 11.3. Давайте вспомним функцию `myFoldl` из урока 9.

```
myFoldl f init [] = init
myFoldl f init (x:xs) = myFoldl f newInit xs
  where newInit = f init x
```

Каков тип этой функции? Замечание: тип `foldl` другой.

12

Создание пользовательских типов

После прочтения урока 12 вы:

- начнёте определять синонимы типов для повышения ясности кода;
- будете создавать свои собственные типы данных;
- сможете создавать типы из других типов;
- научитесь использовать синтаксис записей для сложных типов.

В предыдущем уроке вы научились использовать базовые типы языка Haskell. Теперь пришло время начать создавать ваши собственные типы. Определение типов в Haskell является более важной частью разработки, чем в большинстве других (даже статически типизированных) языках программирования, так как практически любая задача, которую вы будете решать, сводится к используемым типам. Даже при работе с уже существующим типом часто бывает удобно его переименовать ради повышения читаемости кода. Рассмотрим, например, следующую типовую аннотацию:

```
areaOfCircle :: Double -> Double
```

С этим типом всё в порядке, но представьте, что вместо него вы увидели бы что-то такое:

```
areaOfCircle :: Diameter -> Area
```

С такой типовой аннотацией вы точно знаете, аргументы какого типа ожидает функция и каков её результат.

Вы также научитесь самостоятельно описывать более сложные типы. Создание типов данных в Haskell настолько же важно, насколько создание классов в объектно-ориентированных языках программирования.

Обратите внимание. Допустим, вы хотите написать функцию, работающую с музыкальными альбомами. Альбом характеризуется следующими свойствами (и их типами): исполнитель (`String`), название альбома (`String`), год выхода (`Int`) и список треков (`[String]`). Единственный известный вам на данный момент способ хранения всех этих данных — кортеж. К сожалению, это выглядит несколько неуклюже и затрудняет процесс получения хранимой информации (поскольку требует сопоставления с образцом для каждого атрибута). Есть ли способ сделать это лучше?

12.1. Использование синонимов типов

В 11-м уроке мы упоминали, что в Haskell вместо типа `[Char]` вы можете писать `String`. С точки зрения Haskell это два названия одного и того же типа. Когда у вас есть два имени для одного типа, это называется *синонимом типа*. Синонимы типов очень полезны, потому что они значительно упрощают чтение типовых аннотаций. Например, у вас может быть функция для написания врачебных отчётов. Функция `patientInfo` принимает имя, фамилию, возраст и вес и используется для составления коротких описаний пациентов.

Листинг 12.1 Определение функции patientInfo

```
patientInfo :: String -> String -> Int -> Int -> String
patientInfo fname lname age height = name ++ " " ++ ageHeight
  where name = lname ++ ", " ++ fname
        ageHeight = "(Возраст: " ++ show age ++
                    "; рост: " ++ show height ++ "см)"
```

Вы можете воспользоваться этой функцией в GHCi:

```
GHCi> patientInfo "Джон" "Доу" 43 188
"Доу, Джон (Возраст: 43; рост: 188см)"
GHCi> patientInfo "Джейн" Смит" 25 156
"Смит, Джейн (Возраст: 25; рост: 156см)"
```

Если предположить, что `patientInfo` является частью более объёмного заявления о пациенте, то, скорее всего, имя, фамилия, возраст и вес будут

часто использоваться. Типовые аннотации в Haskell гораздо больше помогают программисту, нежели компилятору. Вам не нужно создавать совершенно новый тип для каждого из этих значений, но при быстром просмотре кода постоянно видеть `String` и `Int` не очень удобно. По аналогии с синонимом типа `String` для `[Char]` вы можете захотеть создать синонимы типов для некоторых свойств ваших пациентов.

Haskell позволяет определять новые синонимы типов с помощью ключевого слова `type`. Приведём код для создания нужных вам синонимов:

Листинг 12.2 Синонимы типов: FirstName, LastName, Age и Height

```
type FirstName = String
type LastName = String
type Age = Int
type Height = Int
```

Теперь можно переписать изначальную типовую аннотацию:

```
patientInfo :: FirstName -> LastName -> Age -> Height -> String
```

Роль синонимов типов не ограничивается простым переименованием. Например, гораздо удобнее хранить имена пациентов в кортеже. Вы можете с помощью одного типа описать кортеж, содержащий имя и фамилию.

Листинг 12.3 Синоним типа PatientName

```
type PatientName = (String, String)
```

Теперь можно определить вспомогательные функции для получения имени и фамилии пациента.

Листинг 12.4 Получение доступа к компонентам типа PatientName

```
firstName :: PatientName -> String
firstName patient = fst patient

lastName :: PatientName -> String
lastName patient = snd patient
```

И протестировать полученный код в GHCi:

```
GHCi> testPatient = ("Джон", "Доу")
GHCi> firstName testPatient
"Джон"
GHCi> lastName testPatient
"Доу"
```

Проверка 12.1. Перепишите функцию `patientInfo` для работы с типом `PatientName`, сократив общее количество аргументов с четырёх до трёх.

12.2. Создание новых типов

Теперь давайте добавим пол пациента, который может быть женским или мужским. Вы могли бы использовать для этого строку, а именно константы `"мужской"` и `"женский"`, или `Int`, или даже `Bool`. Во многих других языках программирования вы, скорее всего, так и сделали бы. Но ни один из этих типов не выглядит идеально подходящим, и можно легко себе представить возможные ошибки, связанные с использованием таких решений. В Haskell вы должны, насколько это возможно, использовать мощь системы типов. В данной ситуации лучше всего создать новый тип. Новый тип может быть создан с помощью ключевого слова `data`, как показано на рис. 12.1.

Рис. 12.1: Определение типа `Sex`

Ответ 12.1

```
patientInfoV2 :: PatientName -> Int -> Int -> String
patientInfoV2 (fname, lname) age height = name ++ " " ++
                                          ageHeight
  where name = lname ++ ", " ++ fname
        ageHeight = "(Возраст: " ++ show age ++
                    "; рост: " ++ show height ++ "см)"
```

Рассмотрим основные компоненты определения типа. Ключевое слово `data` говорит Haskell, что вы определяете новый тип. Слово `Sex` представляет собой конструктор типа. В данном случае конструктор является именем типа, но в следующих уроках вы увидите, что конструкторы типов могут принимать аргументы. `Male` и `Female` являются конструкторами данных. *Конструктор данных* используется для создания конкретных экземпляров типа данных. Разделяя конструкторы данных с помощью символа `|`, вы говорите: «Значение типа `Sex` может быть экземпляром `Male` или `Female`».

Оказывается, что именно таким образом объявлен тип `Bool` в Haskell:

```
data Bool = True | False
```

Почему бы просто не использовать Bool для объявления синонима типа? Во-первых, у вас есть собственные, более читаемые конструкторы данных. Это облегчает использование таких средств языка, как сопоставление с образцом. Ниже приведена функция, возвращающая один символ в соответствии с полом пациента.

Листинг 12.5 Определение функции sexInitial

```
sexInitial :: Sex -> Char
sexInitial Male = 'M'
sexInitial Female = 'F'
```

Если бы вы воспользовались синонимом типа, вам пришлось бы использовать здесь `True` и `False`, что негативно сказалось бы на читаемости кода. Более важным является тот факт, что теперь компилятор может проверить, что вы всегда используете правильные типы. Любая возможная ошибка, связанная со случайным использованием `Bool` несовместимым с вашим типом `Sex` образом, теперь будет выявлена компилятором.

Сейчас давайте смоделируем тип крови пациента, что является более сложной задачей, нежели моделирование пола. Когда речь идёт о типе крови, вы можете сказать: «у него первая положительная» или «у неё четвёртая отрицательная». Компоненты названия типов крови «первая» или «четвёртая» называются *группами крови*[1]. Существует четыре вида групп крови: первая, вторая, третья и четвёртая. Распределение между ними происходит в соответствии с наличием в крови определённых семейств антител. Компонент названия «положительная» или «отрицательная» относится к резус-фактору человека и обозначает наличие или отсутствие в крови

[1] В английском языке группам крови с первой по четвёртую соответствуют названия O, A, B и AB соответственно, в коде будут использоваться именно они. — *Прим. пер.*

определённого антигена. Несовпадение между антителами и антигенами при переливании крови может привести к летальному исходу.

Чтобы смоделировать тип крови, вы могли бы сделать то же, что и с полом человека, перечислив все возможные конструкторы данных (`APos | ANeg | BPos ...`). Но, учитывая, что у вас есть два возможных резус-фактора для каждой из четырёх групп крови, вы получите целых восемь возможных конструкторов! Лучшим решением будет начать с определения типов для резус-фактора и группы крови по отдельности.

Тип для резус-фактора будет выглядеть почти как тип для пола.

Листинг 12.6 Определение типа RhType

```
data RhType = Pos | Neg
```

У типа для группы крови будет четыре конструктора данных.

Листинг 12.7 Определение типа ABOType

```
data ABOType = A | B | AB | O
```

Наконец, вы можете определить тип крови — `BloodType`. Как было отмечено раньше, тип крови — это `ABOType` и `RhType`. Именно так вы его и определите, как показано на рис. 12.2.

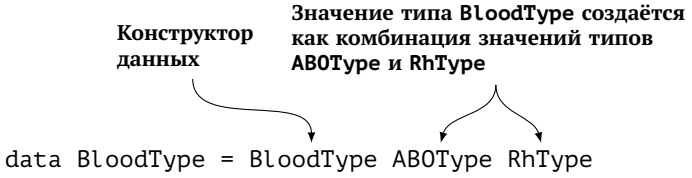


Рис. 12.2: Комбинирование `ABOType` и `RhType` для создания `BloodType`

Обратите внимание: в данном случае имя конструктора данных совпадает с именем конструктора типа. Это не обязательно должно быть так, но в данном случае имеет смысл так сделать. Ваш конструктор данных должен комбинировать значения типов `ABOType` и `RhType`. Вы можете прочесть этот конструктор данных следующим образом: `BloodType` — это `ABOType` вместе с `RhType`.

Теперь вы можете создавать данные типа `BloodType`:

```
patient1BT :: BloodType
patient1BT = BloodType A Pos
```

```
patient2BT :: BloodType
patient2BT = BloodType O Neg

patient3BT :: BloodType
patient3BT = BloodType AB Pos
```

Было бы здорово иметь возможность печатать эти значения. В 13-м уроке рассматривается более хороший способ это сделать, но сейчас давайте просто напишем функции `showRh`, `showABO` и `showBloodType`. С помощью сопоставления с образцом для вашего нового типа это не составит труда!

Листинг 12.8 Вывод на экран значений пользовательских типов

```
showRh :: RhType -> String
showRh Pos = "+"
showRh Neg = "-"

showABO :: ABOType -> String
showABO A = "A"
showABO B = "B"
showABO AB = "AB"
showABO O = "O"

showBloodType :: BloodType -> String
showBloodType (BloodType abo rh) = showABO abo ++ showRh rh
```

Обратите внимание, как легко с помощью сопоставления с образцом на последнем шаге можно извлечь `ABOType` и `RhType` из `BloodType`.

Самый интересный вопрос, который можно задать о типе крови: «Может ли один пациент быть донором для другого?» Совместимость типов крови определяется следующими правилами:

- пациент с первой группой крови (O) может быть донором для всех групп крови;
- пациент со второй группой крови (A) может быть донором для второй (A) и четвёртой (AB);
- пациент с третьей группой крови (B) может быть донором для третьей (B) и четвёртой (AB);
- пациент с четвёртой группой крови (AB) может быть донором только для четвёртой (AB).

(Обратите внимание: в данном случае нас не заботит резус-фактор.)

Вам нужна функция canDonateTo для определения возможности донорства одного BloodType по отношению к другому.

Листинг 12.9 Определение функции canDonateTo

```
canDonateTo :: BloodType -> BloodType -> Bool
canDonateTo (BloodType O _) _ = True   <-- Универсальный донор
canDonateTo _ (BloodType AB _) = True  <-- Универсальный получатель
canDonateTo (BloodType A _) (BloodType A _) = True
canDonateTo (BloodType B _) (BloodType B _) = True
canDonateTo _ _ = False   <-- Во всех остальных случаях
```

Ниже приведены примеры в GHCi:

```
GHCi> canDonateTo patient1BT patient2BT
False
GHCi> canDonateTo patient2BT patient1BT
True
GHCi> canDonateTo patient2BT patient3BT
True
GHCi> canDonateTo patient1BT patient3BT
True
GHCi> canDonateTo patient3BT patient1BT
False
```

Теперь было бы неплохо немного изменить обработку имён пациентов. Было бы полезно иметь возможность моделировать необязательное поле «второе имя» (middle name). На данный момент у вас есть синоним типа PatientName, представляющий собой кортеж из имени и фамилии. Вы могли бы использовать знания, полученные при определении типов Sex и BloodType для создания более удобного типа Name. Давайте добавим синоним типа MiddleName и используем его для создания более сложного типа для имён.

Листинг 12.10 Поддержка различных имён: MiddleName и Name

```
type MiddleName = String
data Name = Name FirstName LastName
          | NameWithMiddle FirstName MiddleName LastName
```

Это определение Name можно прочесть следующим образом: Name представляет собой либо имя и фамилию, либо имя, второе имя и фамилию. Вы можете использовать сопоставление с образцом для создания функции showName, работающей с любым из конструкторов.

Листинг 12.11 Отображение нескольких конструкторов: showName

```
showName :: Name -> String
showName (Name f l) = f ++ " " ++ l
showName (NameWithMiddle f m l) = f ++ " " ++ m ++ " " ++ l
```

Теперь парочка примеров:

```
name1 = Name "Джером" "Сэлинджер"
name2 = NameWithMiddle "Джером" "Дэвид" "Сэлинджер"
```

И посмотрим на их поведение в GHCi:

```
GHCi> showName name1
"Джером Сэлинджер"
GHCi> showName name2
"Джером Дэвид Сэлинджер"
```

Теперь у вас есть гораздо более гибкий тип `Name`.

12.3. Использование синтаксиса записей

В начале этого урока вы передавали в функцию `patientInfo` четыре аргумента:

```
patientInfo :: String -> String -> Int -> Int -> String
patientInfo fname lname age height = name ++ " " ++ ageHeight
  where name = lname ++ ", " ++ fname
        ageHeight = "(Возраст: " ++ show age ++
                    "; рост: " ++ show height ++ "см)"
```

В определении этой функции мы пытались зафиксировать идею передачи данных пациента в функцию, но у нас не было средств для того, чтобы выразить эту информацию в Haskell компактно. Теперь, когда вы больше знаете о типах, вы должны быть способны создать тип `Patient`, хранящий всю эту и некоторую другую информацию. Это позволит вам сохранить время, избавив от необходимости передавать в функцию большое количество аргументов всякий раз, когда нужно решить задачу, связанную с общей информацией о пациенте.

Первый шаг в моделировании типа пациента — перечислить все детали, которые нужно учитывать, а также определить типы для их представления:

- имя: `Name`;
- пол: `Sex`;
- возраст (в годах): `Int`;
- рост (в сантиметрах): `Int`;
- вес (в килограммах): `Int`;
- тип крови: `BloodType`.

Теперь вы можете использовать ключевое слово `data` для создания нового типа, представляющего эту информацию, по аналогии с тем, что делалось с типом крови.

Листинг 12.12 Тип Patient, версия 1

```
data Patient = Patient Name Sex Int Int Int BloodType
```

Вы получили компактное представление всех шести атрибутов пациента. Это здорово, так как теперь можно выполнять всевозможные вычисления, связанные с пациентом, без необходимости передавать в функции длинные списки аргументов. Давайте создадим пациента из нашего первого примера:

```
johnDoe :: Patient
johnDoe = Patient (Name "Джон" "Доу") Male 43 188 92
                                          (BloodType AB Pos)
```

Проверка 12.2. Создайте пациента по имени Джейн Элизабет Смит, используя любые нравящиеся вам (но разумные) значения.

Создание таким способом новых данных отлично работает для `Sex` и `BloodType`, но определённо вызывает некоторые затруднения для данных с таким большим количеством свойств. На ранних этапах вы можете

Ответ 12.2

```
janeESmith :: Patient
janeESmith = Patient
               (NameWithMiddle "Джейн" "Элизабет" "Смит")
               Female 28 157 64
               (BloodType O Neg)
```

немного улучшить ситуацию с помощью синонимов типов. Но даже если бы определение типа `Patient` было более читаемым, сам такой тип не слишком удобен. Отвернитесь на секунду от страницы и попробуйте вспомнить порядок значений. Можно легко себе представить некоторые дополнительные значения, которые вы могли бы добавить к определению пациента, ещё больше затруднив работу с этим типом данных.

С этим представлением пациентов связана ещё одна неприятность. Разумно ожидать иметь возможность получать компоненты данных о пациенте по отдельности. Вы можете достичь этого, реализовав функции для получения каждого из значений с помощью сопоставления с образцом.

Листинг 12.13 Функции getName, getAge, getBloodType

```
getName :: Patient -> Name
getName (Patient n _ _ _ _ _) = n

getAge :: Patient -> Int
getAge (Patient _ _ a _ _ _) = a

getBloodType :: Patient -> BloodType
getBloodType (Patient _ _ _ _ _ bt) = bt
```

Сопоставление с образцом сильно облегчает написание таких функций, но необходимость писать все шесть раздражает. Представьте себе, что ваше финальное определение типа `Patient` будет содержать 12 значений! С самого начала придётся проделать много работы, что не вписывается в стиль Haskell. К счастью, в Haskell есть отличное решение этой проблемы. Вы можете определять такие типы данных, как `Patient`, с помощью *синтаксиса записей*. Определение нового типа данных с помощью синтаксиса записей значительно облегчает понимание того, какими типами представлены различные свойства типа данных.

Листинг 12.14 Тип Patient, версия 2 (синтаксис записей)

```
data Patient = Patient { name :: Name
                       , sex :: Sex
                       , age :: Int
                       , height :: Int
                       , weight :: Int
                       , bloodType :: BloodType}
```

Первый плюс в пользу синтаксиса записей состоит в том, что ваше определение типа теперь гораздо более читаемо и доступно для пониманию. Следующий большой плюс состоит в значительном облегчении созда-

ния данных типа Patient. Вы можете задать значение каждого свойства с помощью имени — порядок больше не имеет значения!

```
jackieSmith :: Patient
jackieSmith = Patient { name = Name "Джеки" "Смит"
                      , age = 43
                      , sex = Female
                      , height = 157
                      , weight = 52
                      , bloodType = BloodType O Neg }
```

Более того, вам не нужно писать свои геттеры — каждое поле в синтаксисе записей автоматически создаёт функцию для доступа к этому значению из записи:

```
GHCi> height jackieSmith
157
GHCi> showBloodType (bloodType jackieSmith)
"O-"
```

Проверка 12.3. Выведите имя Джеки Смит.

Синтаксис записей позволяет обновлять значения отдельных полей, передавая новое значение в фигурных скобках сразу после имени записи. Допустим, вы должны обновить возраст Джеки Смит в связи с её днём рождения. С помощью синтаксиса записей вы можете сделать это следующим образом:

Листинг 12.15 Обновление поля записи jackieSmith

```
jackieSmithUpdated = jackieSmith { age = 44 }
```

Вы всё ещё находитесь в чистом функциональном мире, а потому будет создан новый пациент и его полю age будет присвоено новое значение.

Ответ 12.3

```
showName (name jackieSmith)
```

Итоги

ЛАНЬ

Целью этого урока было научить вас основам создания типов. Вы начали с синонимов типов, позволяющих изменять имена существующих типов. Синонимы типов облегчают понимание кода во время чтения типовых аннотаций. Затем вы научились создавать свои собственные типы, комбинируя существующие с помощью ключевого слова `data`. Наконец, вы узнали, как синтаксис записей упрощает доступ к отдельным компонентам типа данных. Давайте удостоверимся, что вы всё поняли.

Задача 12.1. Напишите аналогичную `canDonateTo` функцию, принимающую в качестве аргументов двух пациентов вместо двух значений типа `BloodType`.

Задача 12.2. Реализуйте функцию `patientSummary`, работающую с вашим типом `Patient`. Эта функция должна возвращать строку следующего вида:

```
**************
Имя пациента: Смит, Джон
Пол: мужской
Возраст: 46
Рост: 185см
Вес: 95кг
Тип крови: B+
**************
```

ЛАНЬ

Если понадобится, создавайте вспомогательные функции.

13 Классы типов

После прочтения урока 13 вы:

- разберётесь с основами классов типов;
- научитесь читать определения классов типов;
- сможете использовать основные классы типов, такие как `Num`, `Show`, `Ord` и `Bounded`.

В этом уроке мы начнём разбираться с важной абстракцией системы типов Haskell — классами типов. Классы типов позволяют вам группировать типы на основе общего поведения. На первый взгляд, классы типов похожи на интерфейсы во многих объектно-ориентированных языках программирования. Класс типов определяет, какие функции должен поддерживать тип, как интерфейс определяет, какие методы должен поддерживать класс. Но классы типов играют более важную роль в Haskell, чем интерфейсы в Java и C#. Главное отличие в том, что по мере погружения в Haskell оказывается, что классы типов требуют выражения мыслей разработчика в соответствии с более мощными формами абстракции. Во многом классы типов — сердце программирования на Haskell.

Обратите внимание. Вы уже несколько раз писали функцию `inc`, которая увеличивает значение на единицу. Но возможно ли написать функцию инкремента, которая работала бы с данными из широкого диапазона числовых типов? Удручает, что в модуле 1 вы могли это сделать без явного указания типов. Как можно написать типовую аннотацию функции `inc`, которая будет работать на всех числах?

13.1. Дальнейшее исследование типов

На данный момент вы уже повидали некоторое множество типовых аннотаций и даже строили свои собственные нетривиальные типы. Один из лучших способов изучать различные типы в Haskell — это команда GHCi `:t` (сокращение от `:type`), которая позволяет узнать тип нужной вам функции. Когда вы впервые написали `simple`, вы сделали это без типовой аннотации:

```
simple x = x
```

Если бы вы захотели узнать тип этой функции, то могли бы загрузить её в GHCi и воспользоваться командой `:t`:

```
GHCi> :t simple
simple :: t -> t
```

То же самое можно сделать и с версией `simple`, записанной как лямбда-функция:

```
GHCi> :t (\x -> x)
(\x -> x) :: r -> r
```

Проверка 13.1. Определите тип:

```
aList = ["кот","собака","мышь"]
```

Если вы начнёте исследовать типы таким образом, то почти сразу встретите вещи, которых ещё не встречали. Посмотрим, к примеру, на сложение:

```
GHCi> :t (+)
(+) :: Num a => a -> a -> a
```

После всего того времени, что мы смотрели на типы, простое сложение так нас запутывает! Самая большая загадка — часть с `Num a =>`.

Ответ 13.1

```
aList = ["кот","собака","мышь"]
GHCi> :t aList
aList :: [[Char]]
```

13.2. Классы типов

То, что вы здесь встретили, — ваш первый класс типа! *Классы типов* в Haskell — это способ описания групп типов, которые ведут себя схожим образом. Если вы знакомы с Java или C#, то классы типов могут напомнить вам интерфейсы. Когда вы видите `Num a`, то можно сказать, что речь идёт о неком типе `a` класса `Num`. Но что это значит — быть частью класса типов `Num`? `Num` — это класс типов, обобщающий понятия числа. Все сущности класса `Num` реализуют определённую на них функцию `(+)`. Также в этом классе типов есть и другие функции. Один из самых полезных инструментов GHCi — это команда `:info`, которая предоставляет информацию о типах и классах типов. Если вы выполните `:info` на `Num`, то получите следующий (сокращённый) вывод:

Листинг 13.1 Определение класса типов Num

```
GHCi> :info Num
class Num a where
    (+) :: a -> a -> a
    (-) :: a -> a -> a
    (*) :: a -> a -> a
    negate :: a -> a
    abs :: a -> a
    signum :: a -> a
```

Команда `:info` показывает определение класса типов. Определение — это список функций, которые класс должен реализовывать, вместе с их типами. Семейство функций, которые описывают число, — это `+`, `-`, `*`, `negate`, `abs` и `signum` (возвращает знак числа). Каждая типовая аннотация использует одну и ту же типовую переменную `a` для всех аргументов и результата. Ни одна из этих функций не может вернуть значение отличного от аргумента типа. Нельзя сложить два значения `Int` и получить `Double`.

Проверка 13.2. Почему деление не включено в список функций, необходимых для `Num`?

Ответ 13.2. Потому что деление (`/`) не определено для всех типов `Num`.

13.3. Преимущества классов типов

Почему вам вообще нужны классы типов? До этого момента все функции, которые вы определяли в Haskell, работали только для одного конкретного набора типов. Без классов типов вам понадобились бы разные имена для каждой функции, которая работает с различными типами. У вас есть типовые переменные, но они слишком гибкие. Например, вы определяете `myAdd` со следующей типовой аннотацией:

```
myAdd :: a -> a -> a
```

Тогда вам нужна возможность вручную проверять, что вы складываете только те типы, которые имеет смысл складывать (что, вообще говоря, в Haskell невозможно).

Классы типов также позволяют вам определять функции на множестве типов, о которых вы даже не могли подумать. Допустим, вы хотели написать функцию `addThenDouble` следующим образом:

Листинг 13.2 Использование классов типов: addThenDouble

```
addThenDouble :: Num a => a -> a -> a
addThenDouble x y = (x + y)*2
```

Так как вы используете класс типов `Num`, этот код будет автоматически работать не только на `Int` и `Double`, но и со всем, что любой программист написал и реализовал для класса типов `Num`. Если вы будете работать с библиотекой римских цифр, то эта функция будет с ними работать при условии, что её автор реализовал класс типов `Num`!

13.4. Определение класса типов

Вывод, который вы получили в GHCi для `Num`, — это буквально определение класса типов. Определения классов типов имеют структуру, изображённую на рис. 13.1.

В определении `Num` вы видите множество типовых переменных. Почти все функции, требуемые в определениях любого класса типов, будут выражаться в терминах типовых переменных, потому что вы буквально описываете целый класс типов. Когда вы определяете класс типов, вы делаете это именно так, потому что не хотите, чтобы ваши функции привязывались к одному типу. Один из способов трактовки классов типов — думать

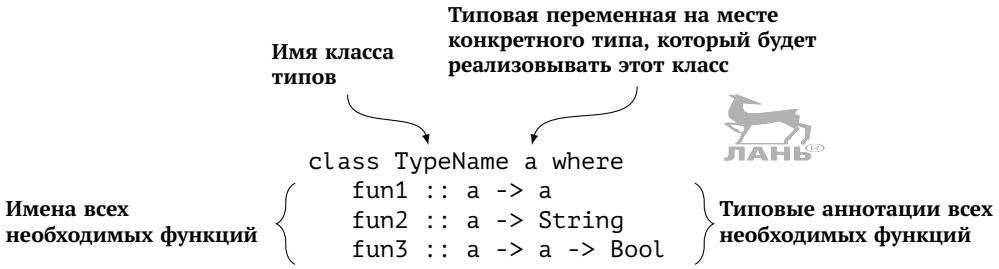


Рис. 13.1: Структура определения класса типов

о них как об ограничениях на наборы типов, которым может соответствовать типовая переменная.

Для закрепления идеи давайте напишем свой простой класс типов. При изучении Haskell будет удобно иметь класс типов `Describable`. Любой тип, являющийся экземпляром `Describable`, может дать своё описание на русском языке. Итак, нужна только одна функция `describe`. Какой бы тип мы ни взяли, если он принадлежит к `Describable`, вызов `describe` на значении этого типа расскажет вам всё об этом типе. Например, если бы тип `Bool` принадлежал классу типов `Describable`, вы бы ожидали чего-то вроде:

```
GHCi> describe True
"Значение типа Bool, True - это обратное значение к False"
GHCi> describe False
"Значение типа Bool, False - это обратное значение к True"
```

На данном этапе мы не будем беспокоиться о реализации класса типов (вы это сделаете в следующем уроке), пока мы его только определим. Вы знаете, что требуется только одна функция `describe`. Единственное, о чем вам нужно подумать, — это типовая аннотация функции. В каждом случае аргумент функции — это значение типа, который реализует наш класс типов, а результат — строка. Значит, вам нужно использовать типовую переменную для первого типа и строку для возвращаемого значения. Вы можете собрать всё это вместе и получить следующий класс типов:

Листинг 13.3 Определение собственного класса типов Describable

```
class Describable a where
   describe :: a -> String
```

Вот и всё! Если хочется, с использованием этого класса можно построить массу инструментов для генерации автоматической документации по коду или даже генерировать учебные пособия.

13.5. Основные классы типов

Для вашего удобства Haskell определяет множество классов типов, о которых вы узнаете из этой книги. В этом разделе мы разберём четыре самых простых: `Ord`, `Eq`, `Bounded` и `Show`.

13.5.1. Классы типов Ord и Eq

Давайте взглянем на простую операцию *больше* (`>`):

```
GHCi> :t (>)
(>) :: Ord a => a -> a -> Bool
```

Вот и новый класс типов, `Ord`! Эта типовая аннотация говорит: «Возьмите два элемента одного типа, который реализует `Ord`, и верните `Bool`». Класс типов `Ord` соответствует всем сущностям вселенной, которые могут быть сравнены и упорядочены. Числа могут быть сравнимы, а также строки и многие другие вещи. Вот список функций, который определяет `Ord`:

Листинг 13.4 Класс Ord требует реализации класса Eq

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

Разумеется, Haskell должен всё усложнять. Заметьте, что прямо в определении класса типов есть другой класс типов! В этом случае это класс `Eq`. Прежде чем разбираться с `Ord`, вы должны взглянуть на `Eq`.

Листинг 13.5 Класс типов Eq обобщает идею равенства

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Класс типов `Eq` требует реализовать только две функции: `==` и `/=`. Если вы можете сказать, что два значения некоторого типа равны или не равны, то этот тип принадлежит классу типов `Eq`. Это объясняет, почему класс

типов Ord включает в своё определение Eq. Прежде чем утверждать, что нечто упорядочено, должна быть возможность сказать, что значения этого типа могут быть равны. Обратное, однако, неверно. Мы можем описать многие вещи, сказав «эти две вещи равны», но не «это лучше, чем то». Вы можете любить ванильное мороженое больше, чем шоколадное, а я могу любить шоколадное больше, чем ванильное. Вы и я можем согласиться, что два рожка ванильного мороженого одинаковы, но мы не можем согласиться о порядке шоколадного и ванильного рожков. Так что если бы вы создали тип Icecream, вы могли бы реализовать Eq, но не Ord.

13.5.2. Класс типов Bounded

В уроке 11 мы отметили разницу между типами Int и Integer. Оказывается, что эта разница также отражается классами типов. Команда :info была полезна для изучения классов типов, но она также полезна для изучения типов. Если вы используете :info на Int, то получите список всех классов типов, членом которых является Int:

```
GHCi> :info Int
data Int = GHC.Types.I# GHC.Prim.Int# -- Defined in 'GHC.Types'
instance Bounded Int -- Defined in 'GHC.Enum'
instance Enum Int -- Defined in 'GHC.Enum'
instance Eq Int -- Defined in 'GHC.Classes'
instance Integral Int -- Defined in 'GHC.Real'
instance Num Int -- Defined in 'GHC.Num'
instance Ord Int -- Defined in 'GHC.Classes'
instance Read Int -- Defined in 'GHC.Read'
instance Real Int -- Defined in 'GHC.Real'
instance Show Int -- Defined in 'GHC.Show'
```

Вы можете сделать то же самое для типа Integer, и если сделаете, то увидите только одно отличие между двумя этими типами. Тип Int имеет экземпляр класса типов Bounded, а Integer — нет. Знание классов типов, связанных с типом, может значительно помочь вам понять, как ведёт себя тип. Класс Bounded — это ещё один простой класс типов (большинство таковы), который требует реализовать только две функции.

Листинг 13.6 Класс типов Bounded содержит значения, а не функции

```
class Bounded a where
  minBound :: a
  maxBound :: a
```

Члены класса Bounded должны давать способ получить их верхнюю и нижнюю границы. Интересно то, что minBound и maxBound — это значения, а не функции! Они не принимают аргументов, а просто являются значениями того типа, которому принадлежат. И Char, и Int принадлежат классу типов Bounded, так что вам никогда не придётся угадывать верхнюю и нижнюю границы для этих типов:

```
GHCi> minBound :: Int
-9223372036854775808
GHCi> maxBound :: Int
9223372036854775807
GHCi> minBound :: Char
'\NUL'
GHCi> maxBound :: Char
'\1114111'
```

13.5.3. Класс типов Show

Мы упомянули функции show и read в уроке 11. Show и Read — это очень полезные классы типов, они определяют функции show и read соответственно. Кроме нескольких случаев для специфических типов, Show реализует только одну важную функцию: show.

Листинг 13.7 Определение класса типов Show

```
class Show a where
  show :: a -> String
```

Функция show переводит значение в String. Любой тип, который реализует Show, может быть напечатан. Вы использовали show гораздо чаще, чем думаете. Каждый раз, когда значение печатается в GHCi, оно печатается благодаря наличию экземпляра класса типов Show. В качестве контрпримера давайте определим ваш тип Icecream, но не реализуем show.

Листинг 13.8 Определение типа Icecream

```
data Icecream = Chocolate | Vanilla
```

Тип Icecream почти идентичен Bool, но Bool реализует Show. Посмотрим, что произойдёт, если вы напечатаете их конструкторы в GHCi:

```
GHCi> True
True
GHCi> False
False
```

```
GHCi> Chocolate
<interactive>:404
:1:
No instance for (Show Icecream) arising from a use of ‘print’
In a stmt of an interactive GHCi command: print it
```

Вы получите ошибку, потому что Haskell понятия не имеет, как представить ваши конструкторы данных в виде строк. Каждое значение, которое вы видели напечатанным в GHCi, было там благодаря классу Show.

13.6. Порождение экземпляров классов типов

Немного раздражает, что для вашего класса типов Icecream нужно реализовать show. В конце концов, Icecream почти как Bool, так почему Haskell не может быть достаточно умным и сделать то же самое, что делает для Bool? Всё, что происходит с Bool, — печатается конструктор данных. Так получилось, что Haskell это умеет! Когда вы определяете тип, Haskell может постараться автоматически породить экземпляр класса типов. Вот синтаксис определения типа Icecream с порождением Show.

Листинг 13.9 Тип Icecream с порождением класса типов Show

```
data Icecream = Chocolate | Vanilla deriving (Show)
```

Теперь вы можете вернуться в GHCi и убедиться, что всё работает:

```
GHCi> Chocolate
Chocolate
GHCi> Vanilla
Vanilla
```

Многие широко используемые классы типов имеют разумную реализацию по умолчанию. Вы также можете добавить класс Eq:

```
data Icecream = Chocolate | Vanilla deriving (Show, Eq)
```

И, воспользовавшись GHCi, увидеть, равны ли два вкуса мороженого:

```
GHCi> Vanilla == Vanilla
True
GHCi> Chocolate == Vanilla
False
GHCi> Chocolate /= Vanilla
True
```

В следующем уроке вы более пристально посмотрите на то, как реализовать свой класс типов, так как Haskell не всегда может угадывать ваши истинные намерения.

Проверка 13.3. Проверьте, какой вкус мороженого лучший по мнению Haskell, породив экземпляр класса типов `Ord`.

Итоги

В этом уроке нашей целью было научить вас основам классов типов. Все классы типов, которые мы обсуждали, должны быть знакомы пользователям объектно-ориентированных языков вроде Java и C#, которые поддерживают интерфейсы. Классы типов, которые вы видели, позволяют легко применять одну функцию ко множеству типов. Это делает проверку на равенство, сортировку данных и перевод данных в строку намного проще. Более того, вы увидели, что Haskell в некоторых случаях может автоматически реализовывать классы типов при использовании ключевого слова `deriving`. Давайте посмотрим, как вы это всё поняли.

Задача 13.1. Если вы запускали примеры с `:info`, то наверняка заметили, что несколько раз встречался тип `Word`. Не пользуясь сторонними ресурсами, примените `:info`, чтобы исследовать тип `Word` с соответствующими ему классами типов и самостоятельно с ним разобраться. Чем он отличается от `Int`?

Задача 13.2. Мы не обсудили класс типов `Enum`. Посмотрите, пользуясь `:info`, на его определение и методы. Обратите внимание, что `Int` реализует и `Enum`, и `Bounded`. Далее, по следующему определению функции `inc`:

```
inc :: Int -> Int
inc x = x + 1
```

и функции `succ`, которая определяется в `Enum`, выясните, в чём отличие между `inc` и `succ` для `Int`?

Ответ 13.3. Если вы добавите `Ord` к списку `deriving` в вашем определении `Icecream`, то Haskell по умолчанию будет учитывать порядок конструкторов данных. Так что `Vanilla` будет больше, чем `Chocolate`.

Задача 13.3. Напишите следующую функцию, которая работает как `succ` на типах из `Bounded`, но не может быть вызвана на значении неограниченного типа данных без ошибки. Эта функция будет работать как `inc` в предыдущем примере, но на большем спектре типов, включая типы, которые не являются членами `Num`:

```
cycleSucc :: (Bounded a, Enum a, ? a) => a -> a
cycleSucc n = ?
```

Ваше определение будет включать функции и значения из `Bounded` и `Enum`, а также некоторого неизвестного класса типов. Запишите, откуда пришли эти три (или больше) функции или значения.

ЛАНЬ

14

Использование классов типов

После прочтения урока 14 вы:

- сможете реализовывать классы типов для собственных типов;
- разберётесь с идеей полиморфизма в Haskell;
- узнаете, когда стоит использовать `deriving`;
- научитесь искать документацию при помощи Hackage и Hoogle.

В уроке 13 вы впервые взглянули на классы типов, которые объединяют типы в Haskell по их общему поведению. В этом уроке вы глубже посмотрите на то, как определять их экземпляры для своих собственных типов. Это позволит вам создавать новые типы, к которым сразу можно будет применять существующие функции из широкого спектра.

Обратите внимание. У вас есть тип данных с конструкторами, обозначающими штаты Новой Англии:

```
data NewEngland = ME | VT | NH | MA | RI | CT
```

Вы хотите иметь возможность выводить их полные названия, используя `Show`. Выводить аббревиатуры с помощью функции `show`, порождённой автоматически, нетрудно, но нет очевидного способа создать свою версию `show`. Можно ли реализовать возможность вывода полного названия штата, используя `show`?

14.1. Тип, требующий классы

Начнём с моделирования шестигранного кубика. Хорошей стандартной реализацией был бы тип наподобие `Bool`, только с шестью значениями вместо двух. Давайте назовём конструкторы последовательно от `S1` до `S6` соответственно граням кубика.

Листинг 14.1 Определение типа SixSidedDie

```
data SixSidedDie = S1 | S2 | S3 | S4 | S5 | S6
```

Затем вы хотите реализовать несколько полезных классов типов. Возможно, `Show` — самый важный для реализации класс, потому что вы почти всегда хотите иметь лёгкий способ выводить значения вашего типа, особенно в GHCi. В уроке 13 мы упомянули, что можно добавить ключевое слово `deriving`, чтобы автоматически создать экземпляр класса. Можно было бы определить `SixSidedDie` таким образом и на этом закончить.

Листинг 14.2 Тип SixSidedDie с порождением экземпляра Show

```
data SixSidedDie = S1 | S2 | S3 | S4 | S5 | S6 deriving (Show)
```

Если бы вы использовали этот тип в GHCi, то получали бы простую текстовую версию конструкторов данных при их вводе:

```
GHCi> S1
S1
GHCi> S2
S2
GHCi> S3
S3
```

Это немного скучно, потому что вы просто печатаете конструкторы данных, которые, конечно, имеют смысл с точки зрения реализации, но не слишком читабельны. Давайте вместо этого выводить словесные описания на русском!

14.2. Реализация Show

Для этого потребуется реализовать ваш первый класс типов — `Show`. Есть только одна функция (в случае классов типов мы называем их *методами*), которую вам нужно реализовать, — `show`. Вот как это делается.

Листинг 14.3 Создание экземпляра Show для типа SixSidedDie

```
instance Show SixSidedDie where
  show S1 = "один"
  show S2 = "два"
  show S3 = "три"
  show S4 = "четыре"
  show S5 = "пять"
  show S6 = "шесть"
```

Вот и всё! Вы можете вернуться в GHCi и получить куда более интересный вывод, чем с `deriving`:

```
GHCi> S1
один
GHCi> S2
два
GHCi> S6
шесть
```

Проверка 14.1. Перепишите определение `show` так, чтобы она выводила римские числа 1–6.

14.3. Классы типов и полиморфизм

Возникает вопрос: почему нужно реализовывать `show` именно так? Почему вам нужно объявлять экземпляр класса типа? На удивление, если

Ответ 14.1

```
data SixSidedDie = S1 | S2 | S3 | S4 | S5 | S6

instance Show SixSidedDie where
  show S1 = "I"
  show S2 = "II"
  show S3 = "III"
  show S4 = "IV"
  show S5 = "V"
  show S6 = "VI
```

вы уберёте объявление экземпляра (instance), следующий код отлично скомпилируется:

Листинг 14.4 Неверная попытка реализовать show для SixSidedDie

```
show :: SixSidedDie -> String
show S1 = "один"
show S2 = "два"
show S3 = "три"
show S4 = "четыре"
show S5 = "пять"
show S6 = "шесть"
```

ЛАНЬ

К сожалению, если вы загрузите этот код в GHCi, то получите две проблемы. Во-первых, GHCi больше не сможет выводить конструкторы типов по умолчанию. Во-вторых, даже если вы вручную используете show, то получите сообщение об ошибке, гласящее, что компилятор видит несколько реализаций функции show:

```
"Ambiguous occurrence 'show'"
```

Вы ещё не изучали систему модулей в Haskell, но эта ошибка связана именно с ней. Дело в том, что определение, которое вы написали для show, конфликтует с тем, что имеется в классе типов. Реальная проблема становится ещё более очевидной, если вы создадите тип TwoSidedDie и попытаетесь написать show для него:

Листинг 14.5 Проблема определения show для TwoSidedDie

```
data TwoSidedDie = One | Two

show :: TwoSidedDie -> String
show One = "один"
show Two = "два"
```

Ошибка, которую вы получите на этот раз, означает попытку определить одну и ту же функцию несколько раз:

```
Multiple declarations of 'show'
```

Истинная проблема в том, что по умолчанию вы хотели бы иметь несколько вариантов поведения show в зависимости от того типа, с которым работаете. То, что вам нужно, называется *полиморфизм*. Полиморфизм означает, что одна и та же функция ведёт себя по-разному в зависимости

от типа данных, с которым работает. Полиморфизм важен как в объектно-ориентированном программировании, так и в Haskell. Эквивалентом `show` является метод `toString` из ООП, общий для всех классов, объектам которых может понадобиться строковое представление. Классы типов — это способ использования полиморфизма в Haskell, как показано на рис. 14.1.

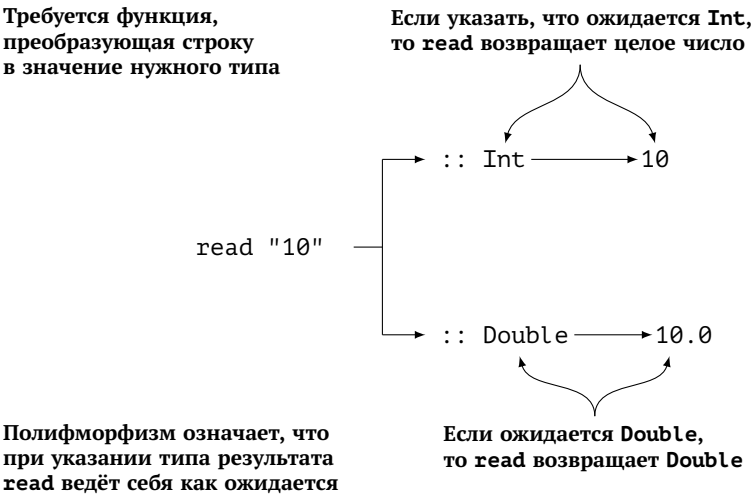


Рис. 14.1: Визуализация полиморфизма для функции `read`

14.4. Реализации методов по умолчанию и минимально полные определения

Теперь, когда вы можете выводить любые строки для конструкторов значений типа `SixSidedDie`, будет полезно определить равенство кубиков. Значит, вам необходимо дать реализацию класса `Eq`. Это полезно ещё и потому, что `Eq` требуется для `Ord`. Мы слегка коснулись этой связи между классами в уроке 13, не давая ей названия. Мы говорим, что `Eq` — это суперкласс `Ord`, то есть что любой экземпляр `Ord` будет также экземпляром `Eq`. В конечном счёте вы захотите сравнивать конструкторы данных `SixSidedDie`, что означает необходимость реализовать `Ord`, а значит, вам сначала нужно реализовать `Eq`. Используя команду `:info` в GHCi, вы можете получить определение класса для `Eq`:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Вам нужно реализовать только два метода: проверки на равенство (==) и неравенство (/=). Учитывая, насколько умён Haskell, кажется, что тут от нас требуют больше работы, чем необходимо. В конце концов, если вы знаете определение (==), то определение (/=) — это *не* (==). Конечно, могут быть некоторые исключения, но в подавляющем большинстве случаев если вы знаете одно, то можете определить другое.

Оказывается, что Haskell тоже так умеет. Классы типов могут иметь реализации методов *по умолчанию*. Если вы определите (==), то Haskell может выяснить, что значит (/=), без какой-либо помощи.

Листинг 14.6 Реализация экземпляра Eq для SixSidedDie

```
instance Eq SixSidedDie where
  (==) S6 S6 = True
  (==) S5 S5 = True
  (==) S4 S4 = True
  (==) S3 S3 = True
  (==) S2 S2 = True
  (==) S1 S1 = True
  (==) _ _ = False
```

В GHCi можно убедиться, что (/=) работает автоматически!

```
GHCi> S6 == S6
True
GHCi> S6 == S5
False
GHCi> S5 == S6
False
GHCi> S5 /= S6
True
GHCi> S6 /= S6
False
```

Это полезно, но как же можно было понять, какие методы следует реализовывать? Команда :info — это отличный источник информации, но не полная документация. Источник более полной информации — это *Hackage*, централизованная библиотека пакетов Haskell. Hackage можно найти в сети по адресу https://hackage.haskell.org. Если вы перейдёте на страницу Eq в Hackage (https://hackage.haskell.org/package/base/docs/Data-Eq.html), то получите гораздо больше информации об этом классе (возможно, даже больше, чем хотели!). Для наших целей самой важной частью является раздел «Минимально полное определение». В случае Eq вы обнаружите следующее:

```
(==) | (/=)
```

Это гораздо более полезно! Для реализации класса типов `Eq` вам достаточно определить либо (==), либо (/=). Как и в определениях данных, | означает *или*. Если вы предоставите любой из этих вариантов, Haskell проделает остальную работу за вас.

Hackage и Hoogle

Хотя Hackage и является главным хранилищем информации о Haskell, искать информацию по нужным вам типам может быть чрезвычайно неудобно. Как решение, поиск по Hackage может осуществляться с помощью замечательного интерфейса по названием Hoogle. Hoogle можно найти по адресу `www.haskell.org/hoogle`. Hoogle позволяет искать по типам и типовым аннотациям. Например, если вы будете искать `a -> String`, то получите `show` и множество других функций. Одного Hoogle достаточно, чтобы вы полюбили систему типов Haskell.

Проверка 14.2. Используйте Hoogle для поиска информации о классе типов `RealFrac`. Каково его минимально полное определение?

14.5. Реализация Ord

Одно из самых важных свойств игрального кубика состоит в том, что у его граней есть порядок. Класс типов `Ord` определяет набор полезных функций для сравнения значения типа:

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
```

Ответ 14.2. Перейдите по адресу `http://hackage.haskell.org/package/base/docs/Prelude.html#t:RealFrac`. Минимально полное определение должно содержать реализацию функции `properFraction`.

```
(>) :: a -> a -> Bool
(>=) :: a -> a -> Bool
max :: a -> a -> a
min :: a -> a -> a
```

К счастью, на Hackage вы можете узнать, что достаточно реализовать только метод `compare`. Метод `compare` принимает два значения некоторого типа и возвращает значение типа `Ordering`. Это тип, который мы уже видели в уроке 4. `Ordering` аналогичен `Bool`, только в нём три конструктора данных. Вот его определение:

```
data Ordering = LT | EQ | GT
```

Далее вы можете увидеть начальную часть определения `compare`.

Листинг 14.7 Частичное определение compare для SixSidedDie

```
instance Ord SixSidedDie where
   compare S6 S6 = EQ
   compare S6 _ = GT
   compare _ S6 = LT
   compare S5 S5 = EQ
   compare S5 _ = GT
   compare _ S5 =  LT
```

Даже с грамотным использованием сопоставления с образцом написание этого определения потребует много работы. Подумайте, насколько большим будет такое определение для 60-гранного кубика!

Проверка 14.3. Выпишите строки определения `compare` для `S4`.

Ответ 14.3

```
compare S4 S4 = EQ
compare _ S4 = LT
compare S4 _ = GT
```

Обратите внимание: благодаря последовательности применения сопоставления с образцом случаи `compare S5 S4` и `compare S6 S4` уже будут сопоставлены ранее.

14.6. Порождать или нет?

На данный момент экземпляры всех классов, которые вы видели, можно было автоматически *породить*. Это значит, что вы можете использовать ключевое слово `deriving` для генерации их реализации для вашего определения типа. Языки программирования обычно предлагают реализации по умолчанию для вещей вроде метода `.equals` (которые зачастую слишком тривиальны, чтобы приносить пользу). Вопрос в следующем: как сильно вы должны полагаться на Haskell в порождении классов типов, или же лучше реализовывать их самостоятельно?

Давайте взглянем на `Ord`. В этом случае разумнее использовать автоматическое порождение посредством `deriving (Ord)`, это работает гораздо лучше в случае простых типов. Стандартное поведение при определении `Ord` состоит в использовании порядка следования конструкторов данных. Рассмотрим, к примеру, следующий листинг.

Листинг 14.8 Порождение класса типов Ord

```
data Test1 = AA | ZZ deriving (Eq, Ord)
data Test2 = ZZZ | AAA deriving (Eq, Ord)
```

В GHCi вы можете увидеть следующее:

```
GHCi> AA < ZZ
True
GHCi> AA > ZZ
False
GHCi> AAA > ZZZ
True
GHCi> AAA < ZZZ
False
```

Проверка 14.4. Перепишите SixSidedDie, породив и `Eq`, и `Ord`.

Ответ 14.4

```
data SixSidedDie = S1 | S2 | S3 | S4 | S5 | S6
  deriving (Show,Eq,Ord)
```

В случае Ord использование deriving спасает вас от написания излишнего и потенциально ошибочного кода.

Ещё более полезный случай для использования deriving возникает при работе с классом типов Enum. Данный класс типов позволяет представить грани кубика как пронумерованный список констант. Это отражает наш естественный образ мыслей относительно игральных кубиков, так что будет полезно. Вот определение:

```
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
```

Опять-таки, вы спасены, потому что вам нужно реализовать только два метода: toEnum и fromEnum. Эти методы будут переводить значения вашего типа в Int и обратно. Вот реализация.

Листинг 14.9 Реализация Enum для SixSidedDie (с ошибками)

```
instance Enum SixSidedDie where
  toEnum 0 = S1
  toEnum 1 = S2
  toEnum 2 = S3
  toEnum 3 = S4
  toEnum 4 = S5
  toEnum 5 = S6
  toEnum _ = error "Такого значения нет"

  fromEnum S1 = 0
  fromEnum S2 = 1
  fromEnum S3 = 2
  fromEnum S4 = 3
  fromEnum S5 = 4
  fromEnum S6 = 5
```

Здесь вы можете увидеть некоторые практические преимущества Enum. Для начала вы можете генерировать списки с элементами из SixSidedDie так же, как с Int и Char:

```
GHCi> [S1 .. S6]
[один,два,три,четыре,пять,шесть]
```

```
GHCi> [S2,S4 .. S6]
[два,четыре,шесть]
GHCi> [S4 .. S6]
[четыре,пять,шесть]
```

Это здорово, но что, если случайно создать бесконечный список?

```
GHCi> [S1..]
[один,два,три,четыре,пять,шесть***
                              ↳ Exception: Такого значения нет
```

Ага! Вы получили ошибку, так как не обработали случай отсутствующего значения. Но если бы вы породили экземпляр класса типов, это не было бы проблемой:

```
data SixSidedDie = S1 | S2 | S3 | S4 | S5 | S6 deriving (Enum)

GHCi> [S1..]
[один,два,три,четыре,пять,шесть]
```

Haskell иногда попахивает магией, особенно когда дело доходит до порождения экземпляров классов типов. В общем, если у вас нет хорошей причины определить экземпляр класса типов самостоятельно, порождение не только проще, но зачастую и лучше.

14.7. Классы типов для более сложных типов

В уроке 4 мы показывали, что вы можете использовать функции как значения первого класса, чтобы правильно сортировать списки кортежей имён.

Листинг 14.10 Использование синонима типов для Name

```
type Name = (String,String)

names :: [Name]
names = [("Иэн","Кертис"),
         ("Бернард","Самнер"),
         ("Питер","Хук"),
         ("Стивен","Моррис")]
```

Как вы, возможно, помните, проблема заключалась в порядке сортировки:

```
GHCi> import Data.List
GHCi> sort names
[("Бернард","Самнер"),("Иэн","Кертис"),("Питер","Хук"),
↳ ("Стивен","Моррис")]
```

Теперь ясно, что для кортежей был порождён экземпляр класса `Ord`, потому что они всё-таки сортируются. К сожалению, они отсортированы не так, как хотелось бы, то есть сначала по фамилии, а затем по имени. В уроке 4 вы использовали функцию как значение первого класса и передавали её в `sortBy`, но делать это больше одного раза раздражает. Не лучше ли реализовать свой экземпляр `Ord` для `Name`?

Листинг 14.11 Попытка реализовать Ord для синонима типа

```
instance Ord Name where
  compare (f1,l1) (f2,l2) = compare (l1,f1) (l2,f2)
```

Но когда вы попробуете загрузить этот код, вы получите ошибку! Проблема — с точки зрения Haskell тип `Name` идентичен `(String, String)` и, как вы видели, Haskell уже умеет их сортировать. Чтобы решить эту проблему, нужно определить новый тип данных. Например, с помощью ключевого слова `data`.

Листинг 14.12 Определение нового типа Name

```
data Name = Name (String, String) deriving (Show, Eq)
```

Здесь необходимость конструкторов данных становится ясна. Тем самым мы сообщаем Haskell, что «этот кортеж отличается от других». Теперь, когда у вас есть этот тип, вы можете реализовать свой `Ord`.

Листинг 14.13 Верная реализация Ord для типа Name

```
instance Ord Name where
  compare (Name (f1,l1)) (Name (f2,l2)) =
                                        compare (l1,f1) (l2,f2)
```

Обратите внимание, что вы можете использовать тот факт, что Haskell порождает экземпляр `Ord` для кортежа `(String, String)`, и это позволяет реализовать собственный `compare` намного проще.

```
names :: [Name]
names = [ Name ("Иэн","Кертис")
        , Name ("Бернард","Самнер")
        , Name ("Питер","Хук")
        , Name ("Стивен","Моррис")]
```

Теперь ваши имена сортируются так, как ожидается:

```
GHCi> import Data.List
GHCi> sort names
[Name ("Иэн","Кертис"),Name ("Стивен","Моррис"),
↳ Name ("Бернард","Самнер"),Name ("Питер","Хук")]
```

Создание типов с помощью newtype

С нашим определением `Name` мы наталкиваемся на интересный случай, когда хотелось бы использовать синоним типа, но нужно определять полноценный тип данных, чтобы сделать данный тип экземпляром класса типов. Haskell имеет более предпочтительный метод для решения такой задачи, предлагая использовать ключевое слово `newtype`. Вот пример определения `Name` с использованием `newtype`:

```
newtype Name = Name (String, String) deriving (Show, Eq)
```

В подобных случаях использование `newtype` более эффективно, чем использование `data`. Любой тип, определённый через `newtype`, может также быть определён через `data`. Но обратное неверно. Типы, определённые через `newtype`, могут иметь только один конструктор данных и одно значение в нём (в случае `Name` внутри один кортеж). В большинстве случаев, когда вам будет нужен конструктор для создания чего-то более мощного, нежели синоним типа, объявление `newtype` — это предпочтительный вариант.

Для простоты в этой книге мы будем придерживаться создания типов через `data`.

14.8. Схема классов типов

На рис. 14.2 демонстрируются классы типов, определённые в стандартной библиотеке Haskell. Стрелки от одного класса к другому связывают суперклассы и классы. В этом модуле мы обсудили большинство базовых классов типов. В модуле 3 вы начнёте исследовать различные особенности классов типов и их отличия от интерфейсов. В модуле 5 вы взглянете на семейство классов типов — `Functor`, `Applicative` и `Monad`, которые позволяют моделировать контекст вычислений. Хотя последняя группа довольно сложная для изучения, она открывает доступ к самым мощным, постоянно используемым в Haskell абстракциям.

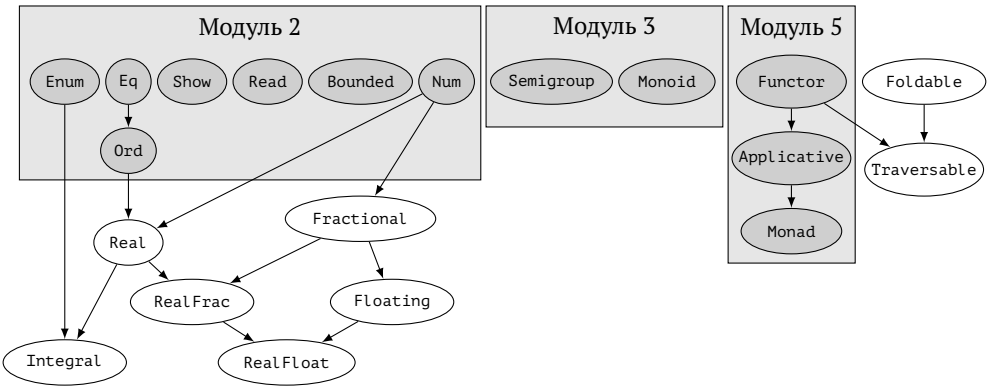


Рис. 14.2: Схема классов типов

Итоги

В этом уроке нашей целью было погружение в классы типов в Haskell. Вы изучили, как читать определения классов типов и как создавать экземпляры классов типов, не ограничиваясь использованием ключевого слова `deriving`. Вы также узнали, когда лучше использовать `deriving`, а когда стоит писать свои экземпляры классов типов. Давайте посмотрим, как вы это поняли.

Задача 14.1. Обратите внимание, что `Enum` не требует ни `Eq`, ни `Ord`, хотя отображает типы на `Int` (который реализует `Eq` и `Ord`). Игнорируя тот факт, что вы легко можете использовать `deriving` для `Eq` и `Ord`, примените сгенерированную реализацию `Enum`, чтобы упростить ручное определение экземпляров `Eq` и `Ord`.

Задача 14.2. Определите пятигранную кость (тип `FiveSidedDie`). Затем определите класс типов `Die` и хотя бы один метод, который был бы полезен для игральной кости. Также включите суперклассы, которые, по вашему мнению, будут полезны для кости. Наконец, сделайте ваш `FiveSidedDie` его экземпляром.

15

Итоговый проект: секретные сообщения!

Этот итоговый проект включает в себя:

- изучение основ криптографии;
- использование простейших типов для моделирования данных;
- практическое использование `Enum` и `Bounded`;
- написание и создание экземпляров собственного класса `Cipher`.

Все обожают тайно друг с другом переписываться. В этом итоговом проекте вы используете свои знания о типах и классах типов, чтобы реализовать несколько примеров шифров. *Шифр* в криптографии — это средство кодирования сообщения с целью, чтобы другие не могли его прочитать. Шифры — это основание криптографии, к тому же они просто забавны, с ними интересно поиграть. Сперва вы взглянете на простой для реализации и взлома шифр, затем узнаете чуть больше об основах шифрования символов и, наконец, создадите невзламываемый шифр!

15.1. Шифры для начинающих: ROT13

Большинство людей открывают для себя криптографию в начальной школе, когда пытаются отправлять тайные сообщения своим друзьям. Типичный способ зашифровать текст, на который натыкаются многие школьники, — это метод ROT13. ROT — это сокращение от *rotation* (*от англ. поворот*), а 13 — количество букв, на которые вы сдвигаетесь относительно исходной. Метод ROT13 работает переводом каждой буквы в предложении

на 13 букв вперёд. Так, например, a — первая буква в алфавите, а через 13 букв от неё в английском алфавите стоит буква n. Значит, при таком кодировании буква a будет заменена на n.

Давайте попробуем отправить с помощью ROT13 секретное сообщение, чтобы убедиться, что вы поняли его идею. Будем отправлять сообщение *Jean-Paul likes Simone* (Жан-Поль любит Симону). В этом примере заглавные и строчные буквы для нас равны, а пробелы и специальные символы мы будем игнорировать. Лучший способ визуализировать ROT13 — это кольцо декодирования, изображённое на рис. 15.1.

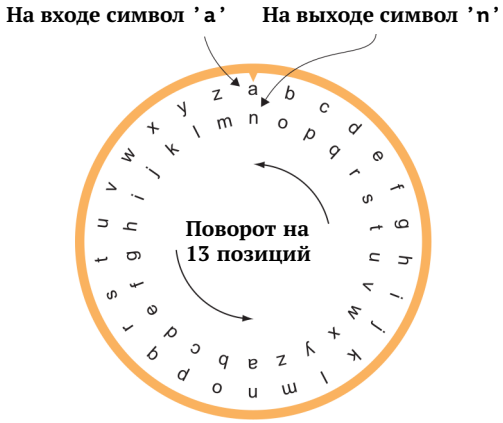


Рис. 15.1: Кольцо декодирования — лучший способ визуализации ROT13

Используя кольцо декодирования, вы можете перевести своё сообщение. Буква *j* отражается на букву *w*, *e* на *r* и так далее. В итоге вы получите *Wrna-Chny yvxrf Fvzbar*.

Причина использования числа 13 состоит в том, что в английском алфавите 26 букв, так что двукратное применение ROT13 вернёт исходный текст. Применение ROT13 к *n* даст *a*. Ещё более интересно, что кодирование *Wrna-Chny yvxrf Fvzbar* даст *Jean-Paul likes Simone*. Симметрия обычна и естественна для большинства криптографических систем.

15.1.1. Реализация своего шифра ROT

Теперь, когда вы увидели, как работает ROT13, давайте сделаем похожий шифр для типа `Char` в Haskell, чтобы вы могли кодировать и декодировать строки. В предыдущем примере вы использовали 13, потому что у вас было 26 символов и вам было удобно делать пол-оборота кольца деко-

дирования, чтобы закодировать сообщение. Вы уже встретились с проблемой, когда вам пришлось игнорировать пробелы и специальные символы в предложении *Jean-Paul likes Simone*. В идеале, хотелось бы делать поворот на *N* символов в любом заданном алфавите. Вы хотите получить общую функцию `rotN`, которая может поворачивать любой алфавит с *N* элементами. Давайте посмотрим, как создать простой четырёхсимвольный алфавит, который можно использовать для экспериментов.

Листинг 15.1 Определение четырёхсимвольного алфавита

```
data FourLetterAlphabet = L1 | L2 | L3 | L4
  deriving (Show,Enum,Bounded)
```

Важно отметить, какие классы типов вы порождаете и почему:

- вы добавляете порождение `Show`, чтобы с типом было проще работать в GHCi;
- вы добавляете порождение `Enum`, потому что это позволит автоматически переводить конструкторы данных в значения типа `Int`. Возможность переводить символ в `Int` позволяет использовать для вращения букв простейшую математику. Функцию `fromEnum` можно использовать для перевода символов в `Int`, а `toEnum` — чтобы переводить их из `Int` обратно в буквы;
- наконец, вы порождаете экземпляр класса `Bounded`, потому что он даёт значения `maxBound` и `minBound`, которые помогут узнать, как далеко нужно вращать алфавит.

Теперь, когда у вас есть тип алфавита, с которым вы можете работать, давайте подумаем, как будет работать сам шифр.

15.1.2. Алгоритм rotN

Вот как будет работать функция `rotN`:

(1) ей передаётся размер алфавита и буква, которую следует повернуть;
(2) для отыскания середины алфавита используется функция `div`, которая отличается от `/` тем, что её результат для значений типа `Int` — целое число, то есть `4 `div` 2` — это 2, `5 `div` 2` — это тоже 2; результат функции `div` показывает, как далеко следует повернуть букву;
(3) для выполнения поворота к целочисленному значению буквы (в представлении `Enum`) добавляется половина длины алфавита. Разумеется, для половины значений `Enum` прибавление половины длины алфавита даст значение, выходящее за границы алфавита. Чтобы этого не про-

изошло, будем брать остаток от деления вычисленного ранее смещения на длину алфавита;

(4) наконец, используется `toEnum`, чтобы перевести представление буквы в `Int` обратно в букву.

Этот алгоритм `rotN` будет работать с любым типом, который является членом и `Bounded`, и `Enum`.

Листинг 15.2 Функция rotN для работы с произвольным алфавитом

```
rotN :: (Bounded a, Enum a) => Int -> a -> a
rotN alphabetSize c = toEnum rotation
  where halfAlphabet = n `div` 2
        offset = fromEnum c + halfAlphabet
        rotation = offset `mod` alphabetSize
```

Находим середину алфавита

Используем середину для вычисления смещения

Остаёмся в границах алфавита

Вы можете попробовать этот алгоритм в GHCi:

```
GHCi> rotN 4 L1
L3
GHCi> rotN 4 L2
L4
GHCi> rotN 4 L3
L1
GHCi> rotN 4 L4
L2
```

Заметим, что тип `Bool` также является членом `Enum` и `Bounded`, а значит, функция `rotN` также будет работать для поворота значений `Bool`. Благодаря классам типов можно кодировать любой тип, реализующий `Enum` и `Bounded`.

Теперь вы можете использовать `rotN`, чтобы вращать значения `Char`. Единственное, что вам нужно выяснить, — насколько их много. Вы можете начать с использования значения `maxBound`, которое требуется классом типов `Bounded`. Используя `maxBound`, вы можете получить наибольшее значение `Char`. Теперь вы можете перевести его в `Int`, используя функцию `fromEnum`, которая определена в `Enum`. Вот код получения номера самого большого из `Char`.

Листинг 15.3 Определение кода наибольшего значения Char

```
largestCharNumber :: Int
largestCharNumber = fromEnum (maxBound :: Char)
```

Нужно добавить `:: Char`, чтобы `maxBound` вычислилось для нужного типа

Значение Int для наименьшего Char — это 0 (вы можете его вычислить, используя аналогичный трюк с minBound). Из-за этого размер алфавита из Char равен largestCharNumber+1. Если вы почитаете о классе типов Enum на Hackage, то увидите, что определение подразумевает, что перечисление начинается с 0 и заканчивается на n-1 (напоминаем, что Hackage, https://hackage.haskell.org — это онлайн-источник полных определений классов типов в Haskell). Из-за этого вам обычно безопаснее подразумевать, что в любом алфавите всегда maxBound+1 элементов. Если вы захотите написать специфичную для Char функцию rotN, она будет выглядеть следующим образом.

Листинг 15.4 Поворот для Char

```
rotChar :: Char -> Char
rotChar charToEncrypt = rotN sizeOfAlphabet charToEncrypt
  where sizeOfAlphabet = 1 + fromEnum (maxBound :: Char)
```

15.1.3. ROT для кодирования строки

У вас уже есть метод для вращения любого одиночного символа любого типа из Enum и Bounded. Но вы хотите кодировать и декодировать сообщения. Сообщения в любом алфавите — это списки букв. Предположим, что у вас есть сообщение в вашем алфавите FourLetterAlphabet, которое вы хотели бы отправить.

Листинг 15.5 Сообщение в четырёхбуквенном алфавите

```
message :: [FourLetterAlphabet]
message = [L1,L3,L4,L1,L1,L2]
```

Чтобы закодировать это сообщение, следует применить соответствующую функцию rotN к каждой букве в этом списке. Лучший способ применить функцию к каждому элементу списка — это map! Далее вы можете увидеть кодировщик для четырёхсимвольного алфавита.

Листинг 15.6 Определение fourLetterEncoder с помощью map

```
fourLetterAlphabetEncoder :: [FourLetterAlphabet] ->
                             [FourLetterAlphabet]
fourLetterEncoder vals = map rot4l vals
  where
    alphaSize = 1 + fromEnum (maxBound :: FourLetterAlphabet)
    rot4l = rotN alphaSize
```

Вот пример вашего закодированного сообщения в GHCi:

```
GHCi> fourLetterEncoder message
[L3,L1,L2,L3,L3,L4]
```

Следующий шаг состоит в декодировании вашего сообщения с помощью `rotN`. Как вы можете помнить из первого раздела, шифр ROT13 — симметричный: чтобы декодировать сообщение в ROT13, вам нужно применить к нему ROT13 ещё раз. Похоже, что у вас есть простое решение: вы можете применить ту же самую функцию `rotN`. Но эта симметрия сработает, только если у вас чётное количество букв в алфавите.

15.1.4. Проблема с декодированием алфавитов нечётной длины

Проблема возникает при декодировании алфавитов нечётной длины, потому что вы делаете целочисленное деление и всегда округляете вниз. В качестве иллюстрации здесь используется `ThreeLetterAlphabet`, соответствующее секретное сообщение и декодер.

Листинг 15.7 Кодирование в трёхсимвольном алфавите

```
data ThreeLetterAlphabet = Alpha
                         | Beta
                         | Kappa deriving (Show,Enum,Bounded)

threeLetterMessage :: [ThreeLetterAlphabet]
threeLetterMessage = [Alpha,Alpha,Beta,Alpha,Kappa]

threeLetterEncoder :: [ThreeLetterAlphabet] ->
                      [ThreeLetterAlphabet]
threeLetterEncoder vals =  map rot3l vals
  where
    alphaSize = 1 + fromEnum (maxBound :: ThreeLetterAlphabet)
    rot3l = rotN alphaSize
```

Теперь вы можете сравнить в GHCi, что происходит при попытке кодирования и декодирования одной и той же функцией для каждого из алфавитов:

```
GHCi> fourLetterEncoder fourLetterMessage
[L3,L1,L2,L3,L3,L4]
GHCi> fourLetterEncoder (fourLetterEncoder fourLetterMessage)
[L1,L3,L4,L1,L1,L2]
GHCi> threeLetterMessage
```

```
[Alpha,Alpha,Beta,Alpha,Kappa]
GHCi> threeLetterEncoder threeLetterMessage
[Beta,Beta,Kappa,Beta,Alpha]
GHCi> threeLetterEncoder (threeLetterEncoder
                                        ↳ threeLetterMessage)
[Kappa,Kappa,Alpha,Kappa,Beta]
```

Как вы можете увидеть, в случае алфавита нечётной длины ваш декодер не симметричен. Для решения этой проблемы вам нужно создать похожую на `rotN` функцию, которая будет добавлять к смещению 1, если в алфавите нечётное количество символов.

Листинг 15.8 Декодер для алфавитов нечётной длины

```
rotNdecoder :: (Bounded a, Enum a) => Int -> a -> a
rotNdecoder n c = toEnum rotation
  where halfN = n `div` 2
        offset = if even n
                 then fromEnum c + halfN
                 else 1 + fromEnum c + halfN
        rotation = offset `mod` n
```

С помощью функции `ronNdecoder` вы можете построить ещё более добротный декодер.

Листинг 15.9 Декодер для трёхбуквенного алфавита

```
threeLetterDecoder :: [ThreeLetterAlphabet] ->
                      [ThreeLetterAlphabet]
threeLetterDecoder vals =  map rot3ldecoder vals
  where
    alphaSize = 1 + fromEnum (maxBound :: ThreeLetterAlphabet)
    rot3ldecoder = rotNdecoder alphaSize
```

В GHCi можете убедиться, что это работает:

```
GHCi> threeLetterMessage
[Alpha,Alpha,Beta,Alpha,Kappa]
GHCi> threeLetterEncoder threeLetterMessage
[Beta,Beta,Kappa,Beta,Alpha]
GHCi> threeLetterDecoder (threeLetterEncoder
                                        ↳ threeLetterMessage)
[Alpha,Alpha,Beta,Alpha,Kappa]
```

Наконец, вы можете собрать всё это вместе, чтобы создать действительно работающие `rotEncoder` и `rotDecoder`, которыми можно кодировать и декодировать строки. Это будет работать независимо от длины алфавита, чётной или нечётной.

Листинг 15.10 Функции rotEncoder и rotDecoder

```
rotEncoder :: String -> String
rotEncoder text = map rotChar text
  where alphaSize = 1 + fromEnum (maxBound :: Char)
        rotChar = rotN alphaSize

rotDecoder :: String -> String
rotDecoder text =  map rotCharDecoder text
  where alphaSize = 1 + fromEnum (maxBound :: Char)
        rotCharDecoder = rotNdecoder alphaSize

threeLetterEncoder :: [ThreeLetterAlphabet] ->
                      [ThreeLetterAlphabet]
threeLetterEncoder vals =  map rot3l vals
  where
    alphaSize = 1 + fromEnum (maxBound :: ThreeLetterAlphabet)
    rot3l = rotN alphaSize

threeLetterDecoder :: [ThreeLetterAlphabet] ->
                      [ThreeLetterAlphabet]
threeLetterDecoder vals =  map rot3ldecoder vals
  where
    alphaSize = 1 + fromEnum (maxBound :: ThreeLetterAlphabet)
    rot3ldecoder = rotNdecoder alphaSize

fourLetterAlphabetEncoder :: [FourLetterAlphabet] ->
                             [FourLetterAlphabet]
fourLetterEncoder vals = map rot4l vals
  where
    alphaSize = 1 + fromEnum (maxBound :: FourLetterAlphabet)
    rot4l = rotN alphaSize

fourLetterDecoder :: [FourLetterAlphabet] ->
                     [FourLetterAlphabet]
fourLetterDecoder vals =  map rot4ldecoder vals
  where
    alphaSize = 1 + fromEnum (maxBound :: ThreeLetterAlphabet)
    rot4ldecoder = rotNdecoder alphaSize
```

Поэкспериментируем с кодированием и декодированием:

```
GHCi> rotEncoder "hi"
"\557160\557161"
GHCi> rotDecoder(rotEncoder "hi")
"hi"
GHCi> rotEncoder "Jean-Paul likes Simone"
```

```
"\557130\557157\557153\55....
GHCi> rotDecoder (rotEncoder "Jean-Paul likes Simone")
"Jean-Paul likes Simone"
```

ROT13 — едва ли надёжный метод отправки сообщений. Из-за того, что каждая буква всегда кодируется одинаково, в закодированном сообщении легко найти повторяющиеся последовательности символов, которые позволяют его декодировать. Далее вы увидите более криптографически надёжный подход к отправке секретных сообщений.

15.2. XOR: магия криптографии!

Прежде чем вы реализуете гораздо более сильный шифр, вам нужно немного узнать о криптографии. К счастью, вам достаточно узнать об одной простой бинарной операции — `XOR`. Операция `XOR` (сокращение от *exclusive or*, исключающее «или») аналогична обычному логическому `OR`, но её результат ложен для случая, когда оба значения истинны. Таблица 15.1 показывает значения `XOR` для пар булевых значений.

Первое значение	Второе значение	Результат
`false`	`false`	`false`
`true`	`false`	`true`
`false`	`true`	`true`
`true`	`true`	`false`

Таблица 15.1: Определение операции XOR

`XOR` широко применяется в криптографии, потому что у него есть два важных свойства. Первое заключается в том, что, как и ROT13, `XOR` симметричен. `XOR` двух списков значений `Bool` вернёт новый список значений `Bool`. `XOR` новых списков вернёт исходные. На рис. 15.2 приведён пример.

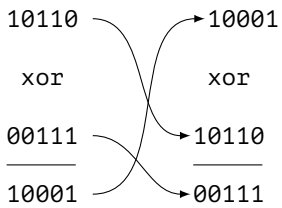


Рис. 15.2: Симметричная природа XOR

Другим полезным свойством XOR является то, что при заданном равномерном распределении значений `True` и `False` (на практике это строки битов) результат XOR будет равномерно распределён вне зависимости от распределения значений True и False в исходном тексте. На практике, если вы возьмёте неслучайную битовую строку вроде текста и сделаете её `XOR` со случайной, то результат будет казаться наблюдателю случайным шумом. Это свойство исправляет основную проблему ROT13. Хотя текст, закодированный через ROT13, неразборчив при первом рассмотрении, повторения символов в выходном тексте такие же, как и в исходном, а значит, их легко декодировать. При правильном применении к данным `XOR` результат неотличим от шума. Вы можете легко визуализировать это свойство, сравнив применение `rotN` к изображению и применение `XOR` с шумом (см. рис. 15.3). Подразумевается, что серые пиксели соответствуют `False`, а чёрные — `True`.

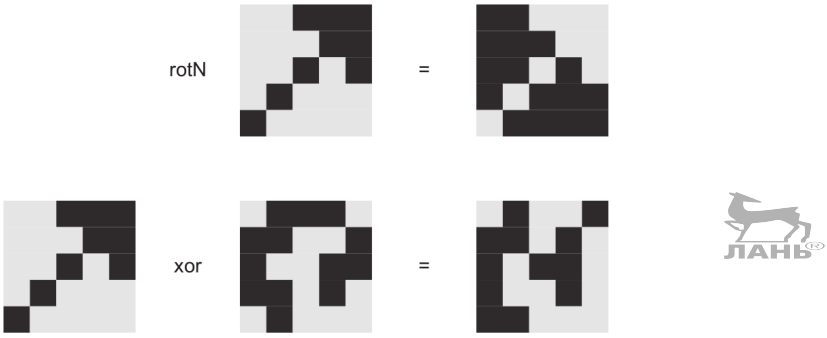


Рис. 15.3: Сравнение кодирования `rotN` и XOR на примере картинки

Давайте определим простую функцию `xor`. Вы начнёте с создания вспомогательной функции `xorBool`, которая будет действовать на двух значениях `Bool`.

> **Примечание.** Модуль `Data.Bool` в Haskell не содержит функции `xor`, которая совпадала бы с определяемой здесь `xorBool`.

Листинг 15.11 Функция xorBool как основа для xor

```
xorBool :: Bool -> Bool -> Bool
xorBool value1 value2 = (value1 || value2) &&
                        (not (value1 && value2))
```

Для вашей версии `xor` главная цель — в простом применении `XOR` к двум спискам значений `Bool`. Внутри, в окончательной версии функ-

ции xor, вы захотите работать с парами, потому что легко скрепить два списка вместе, а затем обработать список пар. В качестве подготовки к этому определим функцию xorPair, которая будет работать с парой булевых значений.

Листинг 15.12 xorPair для применения xor к паре Bool

```
xorPair :: (Bool,Bool) -> Bool
xorPair (v1,v2) = xorBool v1 v2
```

Наконец, вы можете собрать всё это вместе в функцию xor, которая работает на двух списках булевых значений.

Листинг 15.13 Наконец, завершенная функция xor

```
xor :: [Bool] -> [Bool] -> [Bool]
xor list1 list2 = map xorPair (zip list1 list2)
```

Вооружившись функцией xor, необходимо выяснить, как применять её к двум строкам.

15.3. Представление значений как битов

В криптографии мы зачастую рассматриваем не списки значений Bool, а потоки битов. Чтобы упростить рассуждения о коде далее в этом разделе, создадим полезный синоним типа — Bits.

> **Примечание.** В модуле 4 вы увидите, как Haskell представляет различные значения в виде битов, пока же создадим свою систему представления.

Листинг 15.14 Синоним типа Bits

```
type Bits = [Bool]
```

Ваша конечная цель — кодировать строки текста, но для этого нужно переводить их в биты. Вы можете начать с перевода значений Int в биты, потому что каждое значение Char может быть переведено в Int. При условии, что у вас есть значение Int, всё, что вам нужно сделать, — перевести десятичное число в поток битов, который совпадает с исходным числом в двоичной системе счисления.

Вы можете перевести число в десятичной системе счисления в двоичную, рекурсивно деля его на 2. Если у вас нет остатка, то вы добавляете

к списку битов `False` (или 0), в противном случае добавляете 1. Вы останавливаетесь, когда число равно 0 или 1. Определим функцию `intToBits'` (обратите внимание на одинарную кавычку). Кавычка в названии этой функции свидетельствует, что она будет служить как вспомогательная к результирующей функции `intToBits`.

Листинг 15.15 Функция intToBits' запускает перевод Int в Bool

```
intToBits' :: Int -> Bits
intToBits' 0 = [False]
intToBits' 1 = [True]
intToBits' n = if (remainder == 0)
               then False : intToBits' nextVal
               else True : intToBits' nextVal
where remainder = n `mod` 2
      nextVal = n `div` 2
```

Если вы попробуете этот код на хорошо известных степенях двойки, то увидите, что данный код имеет небольшую проблему:

```
GHCi> intToBits' 2
[False,True]
GHCi> intToBits' 8
[False,False,False,True]
```

Алгоритм работает хорошо, за исключением того, что число выше перевёрнуто! В итоговой версии `intToBits` (без ') нам нужно будет перевернуть вывод `intToBits'`. Более того, вы хотели бы, чтобы все ваши списки `Bits` были одного размера. Сейчас `intToBits 0` вернёт список из одного значения `Bool`, в то время как `intToBits maxBound` вернёт список с 63 значениями `Bool`! Чтобы решить данную проблему, вам нужно убедиться, что вы добавляете к началу дополнительные значения `False`, чтобы список был равен по длине самому длинному из кодируемых значений. Для этого мы сначала посчитаем значение `maxBits`.

Листинг 15.16 Значение maxBits и итоговая функция intToBits

```
maxBits :: Int
maxBits = length (intToBits' maxBound)

intToBits :: Int -> Bits
intToBits n = leadingFalses ++ reversedBits
    where reversedBits = reverse  (intToBits' n)
          missingBits = maxBits - (length reversedBits)
          leadingFalses = take missingBits (cycle [False])
```

Наконец, вы можете переводить одно значение `Char` в `Bits`.

Листинг 15.17 Перевод значений Char в Bits

```
charToBits :: Char -> Bits
charToBits char = intToBits (fromEnum char)
```

С `charToBits` у вас есть базовые инструменты для создания более криптографически надёжных сообщений! Единственная проблема в том, что вы хотели бы переводить биты обратно в значения `Char`. К счастью, это не очень сложно. Начнём с функции `bitsToInts`. Создаёте список индексов для каждого бита, и если бит равен `True`, то прибавляете к нему число 2 в степени индекса. Например, двоичное число 101 в десятичной системе — это $1 * 2^2 + 0 * 2^1 + 1 * 2^0$. Так как, кроме 1 и 0, в двоичной системе ничего нет, то берём сумму ненулевых степеней. Вы могли бы использовать `if then else`, но куда более в стиле Haskell применить `filter`.

Листинг 15.18 Функция bitsToInt и переход от Bits к Int

```
bitsToInt :: Bits -> Int
bitsToInt bits = sum (map (\x -> 2^(snd x)) trueLocations)
  where size = length bits
        indices = [size-1,size-2 .. 0]
        trueLocations = filter (\x -> fst x == True)
                        (zip bits indices)
```

Вы можете убедиться, что эта функция работает, в GHCi:

```
GHCi> bitsToInt (intToBits 32)
32
GHCi> bitsToInt (intToBits maxBound)
9223372036854775807
```

> **Примечание.** Есть один источник возможных ошибок в обработке перевода целых чисел в биты: у вас нет обработки отрицательных значений. Это нормально в данном случае, потому что вы используете `intToBits` и обратную к ней функцию только как средство работы с `Char` и `Enum`. Все значения `Char` и `Enum` находятся между 0 и `maxBound`, а значит, вы никогда не встретите отрицательное число на практике. В уроке 38 мы разберём случаи, подобные этому, подробнее.

Последняя функция, которая вам нужна для работы с битами, — это `bitsToChar`, которая может быть определена с помощью `toEnum`.

Листинг 15.19 Функция bitsToChar

```
bitsToChar :: Bits -> Char
bitsToChar bits = toEnum (bitsToInt bits)
```

В GHCi вы можете увидеть, что это также работает:

```
GHCi> bitsToChar (charToBits 'a')
'a'
GHCi> bitsToChar (charToBits maxBound)
'\1114111'
GHCi> bitsToChar (charToBits minBound)
'\NUL'
```

Теперь вы можете собрать все эти кусочки вместе, чтобы создать гораздо более надёжную систему отправки секретных сообщений!

15.4. Одноразовый блокнот

Функция `xor` позволит нам перейти от ненадёжного шифра `ROT13` к невзламываемым одноразовым блокнотам! *Одноразовый блокнот* — это невероятно важный криптографический инструмент; при правильной реализации он не может быть взломан. Идея одноразового блокнота проста. Вы начинаете со своим текстом и текстом, который хотя бы настолько же длинный, что и исходный. По традиции второй текст писался в блокноте, отсюда и слово *блокнот* в названии шифра. Одноразовый блокнот невозможно взломать при условии, что текст в блокноте достаточно случаен и, как говорится в названии, блокнот используется только один раз.

15.4.1. Реализация одноразового блокнота

Давайте начнём с простого блокнота.

Листинг 15.20 Простой блокнот

```
myPad :: String
myPad = "Shhhhhh"
```

У вас имеется текст, который вы хотите зашифровать.

Листинг 15.21 Исходный текст

```
myPlainText :: String
myPlainText = "Haskell"
```

Для того чтобы зашифровать `myPlainText`, вы переводите блокнот и исходный текст в биты, а затем делаете `xor` результатов.

Листинг 15.22 Применение блокнота к строке: функция apllyOTP'

```
applyOTP' :: String -> String -> [Bits]
applyOTP' pad plaintext =  map (\pair ->
                                  (fst pair) `xor` (snd pair))
                               (zip padBits plaintextBits)
where padBits = map charToBits pad
      plaintextBits = map charToBits plaintext
```

Разумеется, `applyOTP'` возвращает только список битов, тогда как нам нужна строка. Окончательная версия `applyOTP` примет вывод `applyOTP'` и преобразует его в строку.

Листинг 15.23 Кодирование строки блокнотом: функция applyOTP

```
applyOTP :: String -> String -> String
applyOTP pad plaintext = map bitsToChar bitList
  where bitList = applyOTP' pad plaintext
```

Теперь можно зашифровать ваш текст:

```
GHCi> applyOTP myPad myPlainText
"\ESC\t\ESC\ETX\r\EOT\EOT"
```

Первое, что мы должны отметить, — никогда не стоит разворачивать собственную криптографическую систему! Ясно, что наш простой одноразовый блокнот на базе `XOR` выдаёт одинаковые символьные последовательности при применении одних и тех же букв друг к другу. Дело в том, что блокнот не особенно хорош, учитывая, сколько букв в нём повторяется. Не забывайте, что `xor` даёт равномерно случайный вывод при условии, что одно из значений `xorBool` равномерно случайно. Ясно, что ваш исходный текст и блокнот тоже не слишком случайны. Если бы блокнот был случайным и взломщик не знал его, то шифр был бы невзламываемым!

Интересно то, что вы можете декодировать свой текст так же, как закодировали его. Используя частичное применение (передача в функцию меньшего числа аргументов, чем она требует, чтобы получить функцию, которая ожидает остальных аргументов, что рассматривалось в уроке 5), вы можете создать кодер/декодер.

Листинг 15.24 Функция encoderDecoder и частичное применение

```
encoderDecoder :: String -> String
encoderDecoder = applyOTP myPad
```

Ваш кодер/декодер будет работать с любым текстом, который короче вашего блокнота.

```
GHCi> encoderDecoder "book"
"1\a\a\ETX"
GHCi> encoderDecoder "1\a\a\ETX"
"book"
```

ЛАНЬ

С реализацией одноразового блокнота у нас появился неплохой способ отправлять зашифрованные сообщения, получше, чем ROT13, с которого всё начиналось. Главное ограничение в том, чтобы блокнот был достаточно случайным, и важно: используйте его только один раз!

15.5. Класс Cipher

ЛАНЬ

Теперь, когда у вас есть два шифра для кодирования сообщений, было бы здорово написать класс типов, который обобщает идею кодирования и декодирования сообщений. Это позволяет вам создать общий интерфейс для новых шифров, которые вы можете написать в будущем, а также упростит работу с `rotEncoder` и `applyOTP`. Определим класс типов `Cipher` с двумя методами: `encode` и `decode`.

Листинг 15.25 Класс Cipher для обобщения операций с шифрами

```
class Cipher a where
  encode :: a -> String -> String
  decode :: a -> String -> String
```

Но каковы типы шифров? Пока что у вас нет ничего, что выглядело бы как цельный тип, только алгоритмы для преобразования строк. Можно начать с определения простого типа для шифра ROT13.

Листинг 15.26 Тип данных ROT

```
data Rot = Rot
```

Чем может быть полезен одинокий конструктор типа и данных? Используя этот простой тип и определив для него реализацию класса `Cipher`, вы можете выразить кодирование ROTN более общими функциями `encode` и `decode`.

Листинг 15.27 Экземпляр класса Cipher для типа ROT

```
instance Cipher Rot where
   encode Rot text = rotEncoder text
   decode Rot text = rotDecoder text
```

Чтобы закодировать текст в ROT13, вы передаёте конструктор данных `Rot` и ваш текст:

ЛАНЬ

```
GHCi> encode Rot "Haskell"
"\557128\557153\557171\557163\557157\557164\557164"
GHCi> decode Rot "\557128\557153\557171\557163\557157\557164
                                                    ↳ \557164"
"Haskell"
```

ЛАНЬ

Далее вам нужно создать тип для одноразового блокнота. Это немного сложнее, потому что одноразовый блокнот требует дополнительного аргумента. Можно указать этот аргумент в определении типа. Определим тип данных `OneTimePad`, который принимает значение типа `String`, служащее блокнотом.

Листинг 15.28 Тип данных OnetimePad

```
data OneTimePad = OTP String
```

Теперь делаем тип `OneTimePad` экземпляром класса `Cipher`.

Листинг 15.29 Экземпляр класса Cipher для типа OneTimePad

```
instance Cipher OneTimePad where
    encode (OTP pad) text = applyOTP pad text
    decode (OTP pad) text = applyOTP pad text
```

Теперь остаётся создать значение типа данных `OneTimePad`, но что использовать в качестве блокнота? Пока то, что вы используете, длиннее обычного текста, всё должно быть в порядке, но как получить что-то, достаточно длинное для любого потенциально возможного текста? Вы можете решить эту задачу при помощи ленивых вычислений и создать бесконечный список всех символов, повторяющийся бесконечно!

Листинг 15.30 Ленивые вычисления и бесконечный блокнот

```
myOTP :: OneTimePad
myOTP = OTP (cycle [minBound .. maxBound])
```

Теперь вы можете кодировать и декодировать строки любой длины:

```
GHCi> encode myOTP "Learn Haskell"
"Ldcqj\%Nf{bog‘"
GHCi> decode myOTP "Ldcqj\%Nf{bog‘"
"Learn Haskell"
GHCi> encode myOTP "this is a longer sentence, I hope
                                                    ↳ it encodes"
"tikp$lu’i)fdbjk}0bw}‘pxt}5:R<uqoE\SOHKW\EOT@HDGMOX"
GHCi> decode myOTP "tikp$lu’i)fdbjk}0bw}‘pxt}5:R<uqoE\SOHKW
                                                  ↳ \EOT@HDGMOX"
"this is a longer sentence, I hope it encodes"
```

С классом `Cipher` у вас есть отличный интерфейс для работы с любой системой передачи секретных сообщений, о которой вы можете подумать. Но помните: никогда, абсолютно никогда не используйте свою криптографию на практике.

Итоги

В этом итоговом проекте вы:

- использовали классы типов `Enum` и `Bounded` для создания собственного шифра `rotN`;
- разобрались с использованием `XOR` для кодирования потоков значений `Bool`;
- применили синонимы типов, позволяющие трактовать `[Bool]` как набор битов `Bits`. В процессе перевода из `Char` в `Bits` и из `Bits` в `Char` вы исследовали, как различные типы работают вместе и как переводить один тип данных в другой. Вы комбинировали эти знания для создания мощного криптографического инструмента, одноразового блокнота;
- определили класс типов `Cipher` как интерфейс для различных методов шифрования и сделали его экземпляры для двух типов, соответствующих различным алгоритмам.

Расширение проекта

Проблема одноразового блокнота заключается в самом одноразовом блокноте. Он должен быть хотя бы таким же длинным, как исходное сообщение, и вы можете использовать его только один раз. Решение состоит

в генерации вашего собственного одноразового блокнота по заданному начальному значению. Как это может быть сделано? Всё, что вам нужно, — это генератор псевдослучайных чисел (ГПСЧ). Получая начальное значение, ГПСЧ выдаёт случайное число. Сгенерировав поток значений `Int`, вы можете использовать `intToBits` для создания всех значений, требуемых для `xor`. Таким образом, ГПСЧ может использовать одно число, чтобы генерировать бесконечный одноразовый блокнот. Шифрование сообщения при помощи результата ГПСЧ называется *потоковым шифрованием*.

Вот код на Haskell для простого ГПСЧ. Данный алгоритм называется *линейным конгруэнтным генератором*.

Листинг 15.31 Линейный конгруэнтный ГПСЧ

```
prng :: Int -> Int -> Int -> Int -> Int
prng a b maxNumber seed = (a*seed + b) `mod` maxNumber
```

Аргументы `a` и `b` — это параметры инициализации, которые помогают достичь случайности, `maxNumber` определяет верхнюю границу числа, которое может быть сгенерировано, и, наконец, задаётся начальное значение. Далее вы можете увидеть пример использования частичного применения для создания примера ГПСЧ для чисел, меньших 100.

Листинг 15.32 Функция examplePRNG

```
examplePRNG :: Int -> Int
examplePRNG  = prng 1337 7 100
```

Вот вы генерируете случайные числа в GHCi:

```
GHCi> examplePRNG 12345
72
GHCi> examplePRNG 72
71
GHCi> examplePRNG 71
34
GHCi> examplePRNG 34
65
```

Чтобы разобраться с этим вопросом, воспользуйтесь ГПСЧ для создания типа `StreamCipher`, который может быть экземпляром класса типов `Cipher`. Запомните: никогда не используйте свою криптографию в реальном мире! Считайте, что её можно применять только для передачи простеньких записок в начальной школе.

Модуль 3

Программирование в типах

Типы в Haskell предлагают вам особый подход к написанию программ. В модуле 1 вы познакомились с общими идеями функционального программирования. Если это было ваше первое знакомство с функциональным программированием, скорее всего, вы увидели абсолютно новый способ мышления о написании кода и решении задач. Система типов в Haskell столь мощна, что можно подходить к ней как ко второму языку программирования, работающему поверх того, что вы изучили в модуле 1.

Но что значит думать о программировании в типах? Когда вы смотрите на типовую аннотацию функции вроде `a -> b`, то можете рассматривать соответствующий тип как описание некоторого преобразования. Допустим, у вас есть тип `CoffeeBeans -> CofeeGrounds`, то есть вы хотите преобразовывать кофейные зёрна в молотый кофе, какую *функцию* мы могли бы описать таким типом? Ясно, что мы говорим о том, чтобы перемолоть зёрна. А как насчёт типа `CoffeeGrounds -> Water -> Coffee`? Мы тут явно берём молотый кофе, добавляем воду и варим готовый к употреблению кофе! Итак, типы в Haskell позволяют интерпретировать программы как последовательности преобразований.

Вы можете думать о *преобразованиях* как о более абстрактном уровне рассуждения о функциях. Решая задачи в Haskell, вы можете подходить к ним как к последовательностям абстрактных преобразований. Например, у вас есть большой текстовый документ, и вам нужно найти все числа в этом документе, а затем сложить их вместе. Как вы будете решать эту задачу в Haskell? Начнём с того, что документ может быть представлен как `String`:

```
type Document = String
```

Затем вам нужна функция, которая разобьёт вашу большую строку на фрагменты, чтобы вы могли отыскать числа:

```
Document -> [String]
```

После этого можно приступать к поиску чисел. Следующая функция примет ваш список строк и функцию, которая проверяет, является ли строка числом, и возвращает только те строки, которые соответствуют числам:

```
[String] -> (String -> Bool) -> [String]
```

Теперь у вас есть числа, но их нужно преобразовать из строк в целые:

```
[String] -> [Integer]
```

Наконец, вам нужно взять список целых и посчитать сумму:

```
[Integer] -> Integer
```

Конец! Думая о типах преобразований, которые вы собираетесь сделать, вы смогли спроектировать целую программу как одну функцию.

Хотя этот пример показывает, как типы позволяют проектировать программы, мы только начали изучать систему типов Haskell и пока не знаем, что она может предложить. В этом модуле вы начнёте исследовать более мощные свойства типов Haskell. Вы будете комбинировать типы способами, невозможными в других языках, увидите, как типы сами могут принимать аргументы, и разберётесь с примерами того, как правильно используемые типы могут исключить из кода целые классы ошибок. Чем больше вы изучаете Haskell, тем чаще будете замечать, что сначала программируете в типах и лишь потом для прояснения деталей пишете функции.

16

Создание типов с помощью «И» и «ИЛИ»

После прочтения урока 16 вы:

- разберётесь с устройством типов-произведений в различных языках программирования;
- научитесь использовать типы-суммы как новый способ моделирования задач;
- сможете выйти за пределы иерархического подхода к проектированию программ.

В этом уроке вы познакомитесь с несколькими уже рассмотренными нами типами немного поближе. Это нужно для того, чтобы вы больше узнали о том, что делает типы в Haskell уникальными и как проектировать программы с их помощью. Большинство изученных вами на данный момент типов являются алгебраическими. *Алгебраические типы данных* — любые типы, которые можно моделировать с помощью комбинирования других типов. К счастью, способов создания таких типов всего два. Вы можете комбинировать типы с помощью «И» (например, имя — это строка `String` «И» ещё одна строка `String`) или с помощью «ИЛИ» (например, `Bool` — это конструктор данных `True` «ИЛИ» конструктор данных `False`). Типы, созданные из комбинации других типов с помощью «И», называются *типами-произведениями*. Типы, скомбинированные при помощи «ИЛИ», называются *типами-суммами*.

Обратите внимание. Вам нужно написать код для работы с меню завтраков в местном кафе. Варианты завтраков состоят из одного или нескольких гарниров, выбранного мяса и основного блюда. Ниже представлены соответствующие типы данных:

```
data BreakfastSide = Toast | Biscuit | Homefries | Fruit
  deriving Show
data BreakfastMeat = Sausage | Bacon | Ham deriving Show
data BreakfastMain = Egg | Pancake | Waffle deriving Show
```

Вы должны определить тип `BreakfastSpecial` для представления комбинаций этих видов еды, доступных для выбора клиентом. Ниже представлены возможные комбинации:

- детский завтрак — одно основное блюдо и один гарнир;
- обычный завтрак — одно основное блюдо, один вид мяса и один гарнир;
- завтрак дровосека — два основных блюда, два вида мяса и три гарнира.

Как бы вы создали один тип, допускающий эти и только эти комбинации для завтрака из всех возможных?

16.1. Типы-произведения — объявление типов с помощью «И»

Типы-произведения определяются посредством комбинирования двух или более существующих типов с помощью «И». Некоторые распространённые примеры:

- дробь может быть представлена числителем (`Integer`) «И» знаменателем (ещё один `Integer`);
- адрес в пределах города — это название улицы (`String`), «И» число (`Int`) для номера дома, «И» число (`Int`) для номера квартиры;
- почтовый адрес может быть адресом в пределах города, «И» городом (`String`), «И»-областью (`String`), «И» почтовым индексом (`Int`).

Хотя название «типы-произведения» для этого способа комбинирования типов может звучать чересчур сложно, это наиболее распространён-

ный способ определения типов во всех языках программирования. Большинство языков программирования поддерживает типы-произведения. Простейший пример — структура в C. Ниже приведён пример структуры в C для книги и автора.

Листинг 16.1 Структуры в C как типы-произведения

```
struct author_name {
  char *first_name;
  char *last_name;
};

struct book {
  author_name author;
  char *isbn;
  char *title;
  int year_published;
  double price;
}
```

В этом примере вы можете увидеть, что тип `author_name` представляет собой комбинацию двух строк (для тех, кто незнаком с языком C: `char*` в C представляет массив символов). Тип `book` состоит из комбинации `author_name`, двух строк, целого и вещественного чисел. Структуры `author_name` и `book` созданы посредством комбинирования других типов с помощью «И». Структуры в C являются предшественниками похожих типов практически во всех языках программирования, включая классы и JSON. В Haskell наш пример с книгой выглядел бы следующим образом:

Листинг 16.2 Реализация структур author_name и book в Haskell

```
data AuthorName = AuthorName String String
data Book = Book Author String String Int Double
```

Для реализации версии `book`, похожей на свою родственницу из C, предпочтительнее использовать синтаксис записей.

Листинг 16.3 Синтаксис записей для реализации Book

```
data Book = Book {
       author :: AuthorName
     , isbn   :: String
     , title  :: String
     , year   :: Int
     , price  :: Double}
```

Типы Book и AuthorName являются примерами типов-произведений и имеют аналоги почти в каждом современном языке программирования. Интересно, что в большинстве языков программирования комбинирование типов с помощью «И» — единственный способ создания новых типов.

Проверка 16.1. Перепишите AuthorName, воспользовавшись синтаксисом записей.

16.1.1. Проклятие типов-произведений: иерархическая архитектура

Создание новых типов только с помощью комбинирования существующих приводит к интересной модели архитектуры программного обеспечения. В связи с ограничением, состоящим в том, что вы можете расширить идею, только добавив к ней что-то, вы ограничиваетесь архитектурой «сверху вниз», начиная с самого абстрактного представления типа, какое вы можете себе представить. Это основа проектирования программного обеспечения в терминах иерархии классов.

В качестве примера предположим, что вы пишете код на Java и хотите начать моделировать данные для книжного магазина. Вы начинаете с Book, приведённой в предыдущем примере (в предположении, что класс Author уже существует).

Листинг 16.4 Первая попытка определения класса Book в Java

```
public class Book {
    Author author;
    String isbn;
    String title;
    int yearPublished;
    double price;
}
```

Ответ 16.1

```
data AuthorName = AuthorName {
     firstName :: String
   , lastName  :: String
}
```

Это решение работает хорошо до тех пор, пока вы не захотите продавать в книжном магазине ещё и виниловые пластинки. Ваше базовое определение `VinylRecord` выглядит так:

Листинг 16.5 Добавление класса VinylRecord в Java

```
public class VinylRecord {
    String artist;
    String title;
    int yearPublished;
    double price;
}
```

Класс `VinylRecord` похож на `Book`, но различий хватает, чтобы начались трудности. Для начала вы не можете переиспользовать тип `Author`, потому что не все исполнители имеют имена; иногда исполнитель — это группа, а не один человек. Например, вы могли бы использовать тип `Author` для *Эллиотта Смита*, но не для *The Smiths*. В традиционной иерархической архитектуре нет хорошего решения проблемы несовпадения `Author` и `artist` (в следующем разделе вы узнаете, как справиться с этой проблемой в Haskell). Другая проблема состоит в том, что у виниловых пластинок нет номеров ISBN.

Возникает серьёзная проблема, если вам нужно создать единый тип для представления виниловых пластинок и книг, чтобы реализовать поиск в каталоге. Из-за того, что вы можете соединять типы только с помощью «И», вам нужно разработать абстракцию, описывающую всё общее, что есть у пластинок и книг. Затем вы должны будете реализовать различающиеся части в отдельных классах. Это основная идея наследования. Далее вы создадите класс `StoreItem`, являющийся родительским по отношению к `VinylRecord` и `Book`. Ниже приведён соответствующий код.

Листинг 16.6 Класс StoreItem — общий предок Book и VinylRecord

```
public class StoreItem {
    String title;
    int yearPublished;
    double price;
}

public class Book extends StoreItem {
    Author author;
    String isbn;
}
```

```
public class VinylRecord extends StoreItem {
    String artist;
}
```

Это решение работает нормально. Теперь вы можете написать остальной код для работы с объектами класса `StoreItem` и использовать условные выражения для работы с `Book` и `VinylRecord`. Но давайте представим, что вам понадобятся ещё и коллекционные фигурки. Ниже представлен класс для них.

Листинг 16.7 Класс CollectibleToy в Java

```
public class CollectibleToy {
    String name;
    String description;
    double price;
}
```

Чтобы всё работало, вам нужно снова переписать весь код! Теперь класс `StoreItem` может содержать только свойство `price`, потому что это единственное значение, являющееся общим для всех товаров. Общие значения для `VinylRecord` и `Book` нужно вернуть в соответствующие классы. В качестве альтернативного решения вы могли бы создать новый класс, являющийся наследником `StoreItem` и предком `VinylRecord` и `Book`. Что насчёт свойства `name` класса `CollectibleToy`? Отличается ли оно от `title`? Возможно, вместо всего этого вам стоит создать интерфейс для всех ваших товаров? Идея здесь состоит в том, что даже в относительно простых случаях использование только лишь типов-произведений при разработке программной архитектуры может быстро приводить к значительным затруднениям.

Проверка 16.2. Предположим, у вас есть типы `Car` и `Spoiler`. Как бы вы представили `SportsCar` как `Car` со `Spoiler`?

В теории создание иерархий объектов представляет собой элегантное решение и позволяет абстрактно зафиксировать всевозможные взаимосвязи, существующие в мире. На практике же даже создание тривиальных

Ответ 16.2

```
data SportsCar = SportsCar Car Spoiler
```

иерархий объектов наполнено архитектурными трудностями. Корень всех этих проблем состоит в том, что единственный способ комбинирования типов в большинстве языков программирования — это «И». Это вынуждает вас начинать со сложной абстракции и спускаться вниз. К сожалению, в реальной жизни встречается множество пограничных случаев, что приводит к значительно большим трудностям, чем вам хотелось бы.

16.2. Типы-суммы — объявление типов с помощью «ИЛИ»

Типы-суммы — удивительно мощное средство языка, хотя они всего лишь предоставляют возможность комбинировать два типа с помощью «ИЛИ». Примеры комбинирования типов с помощью «ИЛИ»:

- игральная кость может быть шестигранной, двадцатигранной или...;
- статья за авторством одного человека (`String`) или группы людей (`[String]`);
- список является пустым (`[]`) или элементом, присоединённым к хвосту списка (`a:[a]`).

Самый простой тип-сумма — `Bool`.

Листинг 16.8 Распространённый тип-сумма: Bool

```
data Bool = False | True
```

Экземпляром типа `Bool` может быть конструктор данных `False` или конструктор данных `True`. Из-за этого может возникнуть ошибочное представление о том, что типы-суммы являются просто способом создавать типы-перечисления в Haskell, которые есть во многих других языках программирования. Но вы уже встречались со случаем, когда типы-суммы могут быть использованы гораздо более мощным способом, например в уроке 12 при определении двух типов имён.

Листинг 16.9 Тип-сумма для моделирования имён

```
type FirstName = String
type LastName = String
type MiddleName = String

data Name = Name FirstName LastName
          | NameWithMiddle FirstName MiddleName LastName
```

В этом примере вы можете использовать два типа конструкторов, а именно: `FirstName`, состоящий из двух строк, или `NameWithMiddle`, состоящий из трёх строк. Здесь использование «ИЛИ» между двумя типами позволяет вам выразительно описать значения типов. Добавление «ИЛИ» к инструментам, которые вы можете использовать для комбинирования типов, открывает перед вами мир возможностей Haskell, не доступных в языках программирования без типов-сумм. Чтобы увидеть, насколько мощными могут быть типы-суммы, давайте разберёмся с некоторыми проблемами из предыдущего раздела.

Интересным местом для начала является разница между автором и исполнителем. В нашем примере вам нужно два типа, потому что предполагается, что имя автора для каждой книги может быть представлено как имя и фамилия, в то время как исполнитель, записывающий песни, может быть представлен именем человека или названием группы. Решить эту проблему с помощью типов-произведений трудно. Но с помощью типов-сумм справиться с этим довольно легко. Для начала вы можете создать тип `Creator`, экземпляром которого может быть либо `Author`, либо `Artist` (вы определите их немного позже).

Листинг 16.10 Тип Creator: либо Author, либо Artist

```
data Creator = AuthorCreator Author | ArtistCreator Artist
```

У вас уже есть тип `Name`, так что вы можете определить `Author` с помощью `Name`:

Листинг 16.11 Тип Author

```
data Author = Author Name
```

С исполнителем всё немного сложнее; как мы уже отметили, `Artist` может быть именем человека или названием группы. Чтобы решить эту проблему, вы можете воспользоваться ещё одним типом-суммой!

Листинг 16.12 Исполнитель может быть либо Person, либо Band

```
data Artist = Person Name | Band String
```

Это хорошее решение, но что, если возникнет один из тех пограничных случаев, которые постоянно встречаются в реальной жизни? Например, что насчёт авторов вроде Г. Ф. Лавкрафта? Вы могли бы заставить себя написать *Говард Филлипс Лавкрафт*, но зачем специально ограничивать себя фиксированной моделью данных? Она должна быть гибкой. Вы можете легко это исправить, добавив к `Name` ещё один конструктор данных.

Листинг 16.13 Расширение типа Name для работы с Г. Ф. Лавкрафтом

```
data Name = Name FirstName LastName
          | NameWithMiddle FirstName MiddleName LastName
          | TwoInitialsWithLast Char Char LastName
```

Обратите внимание, что все типы `Artist`, `Author` и, как следствие, `Creator` зависят от определения `Name`. Но вам достаточно изменить только само определение `Name` без необходимости волноваться об определениях других типов, использующих `Name`. В то же время вы всё ещё можете извлекать выгоду из переиспользования кода, так как типы `Artist` и `Author` получают преимущество благодаря определению `Name` только в одном месте. В качестве примера ниже приведён наш Г. Ф. Лавкрафт типа `Creator`.

Листинг 16.14 Объявление Г. Ф. Лавкрафта типом Creator

```
hpLovecraft :: Creator
hpLovecraft = AuthorCreator
              (Author
                (TwoInitialsWithLast 'H' 'P' "Lovecraft"))
```

Возможно, конструкторы данных в этом примере излишне подробны, на практике вы скорее станете использовать функции, которые будут абстрагироваться от большей части этих деталей. Теперь сравните это решение с тем, которое вы могли бы использовать при иерархической архитектуре, требуемой для типов-произведений. С точки зрения иерархической архитектуры вам бы понадобился родительский класс `Name` с единственным свойством «фамилия» (так как это единственное свойство, разделяемое всеми тремя типами). Затем вам бы понадобились отдельные подклассы для каждого из трёх используемых вами конструкторов данных. Но даже тогда имя вроде *Василий К.* с инициалом вместо фамилии полностью сломало бы вашу модель. Это легко исправить при помощи типов-сумм.

Листинг 16.15 Расширение Name для представления *Василия К.*

```
data Name = Name FirstName LastName
    | NameWithMiddle FirstName MiddleName LastName
    | TwoInitialsWithLast Char Char LastName
    | FirstNameWithInit FirstName Char
```

Единственное решение с точки зрения типов-произведений состоит в создании класса `Name` с растущим списком полей, которые не всегда будут использоваться:

```
public class Name {
    String firstName;
    String lastName;
    String middleName;
    char firstInitial;
    char middleInitial;
    char lastInitial;
}
```

Для работы с таким классом потребуется много дополнительного кода для обеспечения того, что всё работает правильно. Помимо этого, у вас нет гарантий того, что `Name` находится в корректном состоянии. Что, если всем этим свойствам заданы значения? Java не может на уровне типов проверить, что объект типа `Name` удовлетворяет ограничениям, наложенным вами на имена. В Haskell вы знаете, что существуют только явно определённые вами типы.

16.3. Собираем книжный магазин

Теперь давайте вернёмся к нашей задаче с книжным магазином и посмотрим, как мышление в терминах типов-сумм может здесь помочь. С вашим мощным типом `Creator` вы сможете переписать `Book`:

Листинг 16.16 Определение типа Book с помощью Creator

```
data Book = Book {
      author    :: Creator
    , isbn      :: String
    , bookTitle :: String
    , bookYear  :: Int
    , bookPrice :: Double
}
```

Вы также можете определить тип `VinylRecord`:

Листинг 16.17 Тип VinylRecord

```
data VinylRecord = VinylRecord {
      artist        :: Creator
    , recordTitle   :: String
    , recordYear    :: Int
    , recordPrice   :: Double
}
```

Почему не просто price?

Внимательный читатель мог заметить, что `Book` и `VinylRecord` имеют собственные уникальные названия для цены. Почему бы не сделать работу с этими типами более единообразной и не использовать имя `price` вместо `bookPrice` и `recordPrice`? Проблема связана не с ограничениями типов-сумм, а с ограничениями возможностей работы Haskell с синтаксисом записей. Вы можете вспомнить, что без синтаксиса записей вы бы определили тип книги следующим образом:

```
data Book = Book Creator String String Int Double
```

Синтаксис записей позволяет автоматизировать создание функций следующего вида:

```
price :: Book -> Double
price (Book _ _ _ _ val) = val
```

Проблема состоит в том, что использование одинаковых имён функций для `Book` и `VinylRecord` означает определение конфликтующих функций в одном пространстве имён!

Этот недостаток Haskell раздражает, и смириться с ним мне самому было непросто. Возможные пути обхода данной проблемы мы обсудим позже. Если вы считаете, что это кошмар, знайте: вы не одиноки.

Теперь вы можете тривиальным образом определить тип `StoreItem`.

Листинг 16.18 Тип StoreItem — это либо Book, либо VinylRecord

```
data StoreItem = BookItem Book | RecordItem VinylRecord
```

Но мы опять забыли про `CollectibleToy`. Благодаря типам-суммам вы можете легко расширить свой тип `StoreItem` новыми данными.

Листинг 16.19 Добавление типа CollectibleToy

```
data CollectibleToy = CollectibleToy {
     name :: String
   , description :: String
   , toyPrice :: Double
}
```

Для исправления StoreItem достаточно просто добавить ещё одно «ИЛИ».

Листинг 16.20 Включение CollectibleToy в StoreItem

```
data StoreItem = BookItem Book
               | RecordItem VinylRecord
               | ToyItem CollectibleToy
```

Наконец, мы продемонстрируем создание функций для работы со всеми этими типами, написав функцию price для получения цены любого из них.

Листинг 16.21 Функция price для вычисления цены StoreItem

```
price :: StoreItem -> Double
price (BookItem book) = bookPrice book
price (RecordItem record) = recordPrice record
price (ToyItem toy) = toyPrice toy
```

Проверка 16.3. Предположим, что для типа Creator определён экземпляр класса Show. Напишите функцию madeBy, имеющую тип StoreItem -> String и определяющую (или, по крайней мере, пытающуюся определить) производителя StoreItem.

Типы-суммы позволяют вам использовать типы гораздо более выразительным способом, в то же время сохраняя возможность удобно создавать группы похожих типов.

Итоги

Целью этого урока было научить вас двум способам создания новых типов из уже существующих. Первый способ — типы-произведения. Типы-

Ответ 16.3

```
madeBy :: StoreItem -> String
madeBy (BookItem book) = show (author book)
madeBy (RecordItem record) = show (artist record)
madeBy _ = "unknown"
```

произведения комбинируют типы с помощью «И», соединяя два или более типа для создания нового типа. Типы-произведения поддерживаются практически во всех языках программирования, но иногда под другими названиями. Другой способ комбинирования типов — использование «ИЛИ». Типы-суммы гораздо менее распространены, чем типы-произведения. Проблема использования только типов-произведений состоит в том, что вам приходится мыслить в терминах иерархических абстракций. Типы-суммы являются мощным средством языка, позволяющим вам определять новые типы гораздо более выразительным способом. Давайте удостоверимся, что вы всё поняли.

Задача 16.1. Чтобы ещё больше усложнить структуру вашего магазина, добавьте в него брошюры. У брошюр будет название, описание и контактная информация об организации, предоставляющей брошюру. Создайте для брошюр тип `Pamphlet` и добавьте его в `StoreItem`. Измените функцию `price` так, чтобы она работала с `Pamphlet`.

Задача 16.2. Создайте тип `Shape`, включающий в себя следующие формы: `Circle`, `Square` и `Rectangle`. Затем напишите функции вычисления периметра и площади `Shape`.

17

Проектирование композицией: полугруппы и моноиды

После прочтения урока 17 вы:

- сможете создавать новые функции с помощью композиции функций;
- будете использовать `Semigroup` для смешивания цветов;
- научитесь использовать охранные выражения;
- сможете решать задачи на вероятность с помощью `Monoid`.

В предыдущем уроке вы узнали, почему типы-суммы позволяют вам мыслить за пределами типичных иерархических шаблонов проектирования программ на большинстве языков программирования. Другим важным пунктом, в котором Haskell отклоняется от традиционной архитектуры ПО, является идея композиции. *Композиция* означает, что вы можете создать что-то новое, скомбинировав две похожие сущности.

Что значит *скомбинировать* две вещи? Например, вы можете получить новый список конкатенацией двух списков, вы можете скомбинировать документы и получить новый документ, и вы можете смешать два цвета, чтобы получить новый цвет. Во многих языках программирования каждый из этих типов комбинирования имел бы свою собственную уникальную операцию или функцию. По аналогии с тем, как почти каждый язык программирования предоставляет стандартный способ преобразования значения типа в строку, Haskell предоставляет стандартный способ комбинирования элементов одного типа.

Обратите внимание. До сих пор для комбинирования нескольких строк вы использовали операцию ++. Это может быть утомительно для больших строк:

```
"это" ++ " " ++ "немного" ++ " " ++ "слишком"
```

Есть ли способ справиться с этим получше?

17.1. Введение в композицию: комбинирование функций

Перед тем как погрузиться в комбинирование типов, давайте рассмотрим кое-что более фундаментальное: комбинирование функций. Специальная функция высшего порядка под названием *композиция*, записывающаяся как `(.)` — точка, принимает две функции в качестве аргументов. Использование композиции для функций очень удобно для комбинирования функций на лету читаемым образом. Ниже приведены некоторые примеры функций, которые могут быть легко выражены с помощью композиции функций.

Листинг 17.1 Использование композиции для создания функций

```
myLast :: [a] -> a
myLast = head . reverse

myMin :: Ord a => [a] -> a
myMin = head . sort

myMax :: Ord a => [a] -> a
myMax = myLast . sort

myAll :: (a -> Bool) -> [a] -> Bool
myAll testFunc = (foldr (&) True) . (map testFunc)
```

Для использования `sort` требуется подключить модуль `Data.List`

Функция `myAll` проверяет, что предикат возвращает `True` для всех элементов списка

Во многих случаях, когда вы можете использовать лямбда-функцию для быстрого создания функции, функция композиции была бы более эффективным и читаемым способом достичь той же цели.

Проверка 17.1. Реализуйте функцию `myAny` с помощью функции композиции. Эта функция должна проверять, что предикат возвращает `True` хотя бы для одного из значений в списке.

17.2. Комбинирование схожих типов: полугруппы

Чтобы глубже изучить композицию, давайте рассмотрим на удивление простой класс типов под названием `Semigroup`. Чтобы это сделать, вам нужно импортировать модуль `Data.Semigroup` (в верхней строке своего файла).

Класс `Semigroup` содержит всего один нужный вам метод — операцию `<>`. Вы можете думать о ней как о комбинировании элементов одного типа. Вы можете тривиальным образом реализовать `Semigroup` для типа `Integer`, определив `<>` как `+`.

Листинг 17.2 Реализация Semigroup для типа Integer

```
instance Semigroup Integer where
  (<>) x y = x + y
```

Эта операция может показаться слишком очевидной, но важно понять, что она означает. Так выглядит типовая аннотация для `(<>)`:

```
(<>) :: Semigroup a => a -> a -> a
```

Этот простой тип является сердцем идеи композиции; вы можете взять две похожие сущности и скомбинировать их для получения новой сущности того же типа.

Ответ 17.1

```
myAny :: (a -> Bool) -> [a] -> Bool
myAny testFunc = (foldr (||) False) . (map testFunc)
```

Пример использования:

```
GHCi> myAny even [1, 2, 3]
True
```

Проверка 17.2. Можно ли использовать `(/)`, чтобы сделать `Int` представителем `Semigroup`?

17.2.1. Полугруппа цветов

Изначально может показаться, что эта концепция полезна только в математике, но на самом деле мы все знакомы с этой идеей с ранних лет. Самый известный пример данной концепции — смешивание цветов. Как вам должно быть известно с детства, мы можем комбинировать основные цвета для получения новых, например:

- голубой и жёлтый дают зелёный;
- красный и жёлтый дают оранжевый;
- голубой и красный дают фиолетовый.

Вы легко можете представить задачу смешивания цветов с помощью типов. Для начала вам нужен простой тип-сумма для цветов.

Листинг 17.3 Определение типа Color

```
data Color = Red | Yellow | Blue | Green | Purple
           | Orange | Brown deriving (Show, Eq)
```

Теперь вы можете реализовать `Semigroup` для типа `Color`.

Листинг 17.4 Реализация Semigroup для Color, версия 1

```
instance Semigroup Color where
   (<>) Red Blue = Purple
   (<>) Blue Red = Purple
   (<>) Yellow Blue = Green
   (<>) Blue Yellow = Green
   (<>) Yellow Red = Orange
   (<>) Red Yellow = Orange
   (<>) a b = if a == b
              then a
              else Brown
```

Ответ 17.2. Нет, потому что деление не всегда возвращает значение типа `Int`, тем самым нарушая правило.

Теперь вы можете поиграть с цветами, прямо как когда в детстве окунали свои пальцы в краску!

```
GHCi> Red <> Yellow
Orange
GHCi> Red <> Blue
Purple
GHCi> Green <> Purple
Brown
```

Всё чудесно, но при смешивании более чем двух цветов вы сталкиваетесь с интересной проблемой. Вам нужно, чтобы смешивание цветов было ассоциативным. *Ассоциативность* означает, что порядок, в котором вы применяете операцию <>, не имеет значения. Для чисел, например, это означает, что $1 + (2 + 3) = (1 + 2) + 3$. Как вы можете увидеть, ваша операция смешивания цветов определённо не является ассоциативной:

```
GHCi> (Green <> Blue) <> Yellow
Brown
GHCi> Green <> (Blue <> Yellow)
Green
```

Ассоциативность не только соответствует нашим интуитивным представлениям (смешивание цветов в любом порядке должно давать один и тот же цвет), но и формально требуется классом типов `Semigroup`. Этот момент может оказаться довольно неожиданным, когда речь пойдёт о более сложных классах типов, которые мы будем рассматривать в этом модуле. Многие из них имеют *законы классов типов*, требующие определённого поведения. К сожалению, компилятор Haskell не может обеспечить их соблюдение. Хорошей практикой является внимательное чтение документации на Hackage (`https://hackage.haskell.org/`) всякий раз, когда вы самостоятельно реализуете нетривиальный класс типов.

17.2.2. Реализация ассоциативности и охранные выражения

Вы можете решить эту проблему, сделав так, что если некоторый цвет используется для создания другого составного цвета, то их комбинация по-прежнему возвращает тот же составной цвет. Таким образом, фиолетовый плюс красный всё ещё даёт фиолетовый. Вы можете подойти к этой проблеме, выписав большое количество правил сопоставления с образцом для всех возможных вариантов. Но это решение было бы довольно объёмным. Вместо этого вы воспользуетесь средством Haskell под названием

охранные выражения. *Охранные выражения* работают почти как сопоставление с образцом, но они позволяют вам производить некоторые вычисления на аргументах, которые вы собираетесь анализировать. На рис. 17.1 приведён пример функции, использующей охранные выражения.

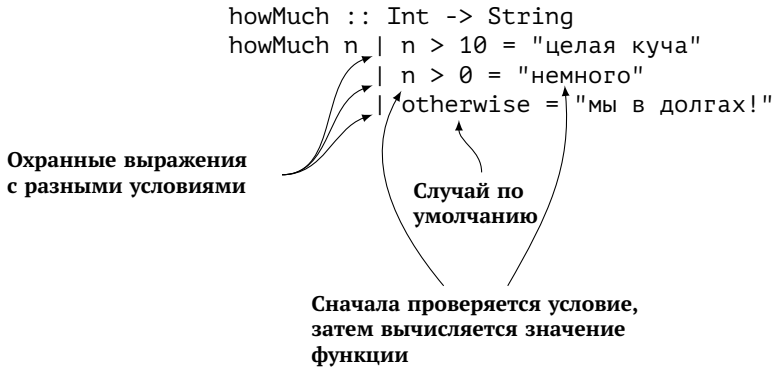


Рис. 17.1: Использование охранных выражений в функции `howMuch`

Воспользовавшись охранными выражениями, вы можете переписать реализацию `Semigroup` для `Color` в соответствии с законами классов типов для полугрупп.

Листинг 17.5 Ассоциативность в реализации Semigroup для Color

```
instance Semigroup Color where
   (<>) Red Blue = Purple
   (<>) Blue Red = Purple
   (<>) Yellow Blue = Green
   (<>) Blue Yellow = Green
   (<>) Yellow Red = Orange
   (<>) Red Yellow = Orange
   (<>) a b
         | a == b = a
         | all (`elem` [Red, Blue, Purple]) [a, b] = Purple
         | all (`elem` [Blue, Yellow, Green]) [a, b] = Green
         | all (`elem` [Red, Yellow, Orange]) [a, b] = Orange
         | otherwise = Brown
```

Можете проверить, что проблема решена:

```
GHCi> (Green <> Blue) <> Yellow
Green
GHCi> Green <> (Blue <> Yellow)
Green
```

Законы классов типов важны, потому что при написании любого кода, использующего экземпляр класса типов, программисты предполагают, что эти законы выполняются.

Проверка 17.3. Поддерживает ли ваша реализация `Semigroup` для типа `Integer` ассоциативность?

В реальном мире имеется множество способов создать новую сущность из двух. Представьте себе следующие примеры композиции:

- комбинирование двух SQL-запросов с созданием нового SQL-запроса;
- комбинирование двух фрагментов HTML для создания нового;
- комбинирование двух форм для создания новой формы.

17.3. Композиция с нейтральным элементом: моноиды

Другим классом типов, похожим на `Semigroup`, является `Monoid`. Единственное существенное различие между `Semigroup` и `Monoid` состоит в том, что `Monoid` требует наличия в типе нейтрального элемента. Нейтральность элемента означает, что `x <> id = x` и `id <> x = x` (обозначение `id` происходит от *identity*). По отношению к сложению целых чисел нейтральным элементом будет 0. Но в своём текущем виде ваш тип `Color` не имеет нейтрального элемента. Наличие нейтрального элемента может показаться незначительной деталью, но оно значительно увеличивает мощь типа, предоставляя вам возможность использовать функцию `fold` для лёгкого комбинирования списков значений этого типа.

Класс типов `Monoid` также интересен тем, что демонстрирует неприятную проблему в эволюции классов типов в Haskell. Логически вы могли бы предположить, что определение `Monoid` выглядит следующим образом:

Листинг 17.6 Разумное определение Monoid

```
class Semigroup a => Monoid a where
   identity :: a
```

Ответ 17.3. Да, потому что сложение целых чисел действительно ассоциативно: $1 + (2 + 3) = (1 + 2) + 3$.

В конце концов, `Monoid` должен быть подклассом `Semigroup`, потому что он представляет собой `Semigroup` с `identity`. Однако `Monoid` появился в Haskell раньше, чем `Semigroup`, и официально не является его подклассом. Вместо этого определение `Monoid` выглядит несколько запутанно:

Листинг 17.7 Настоящее определение Monoid

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> [a]
```

Почему `mempty`, а не `identity`? Почему `mappend` вместо `<>`? Эти странности в именах связаны с тем, что класс типов `Monoid` был добавлен в язык до `Semigroup`. Самый распространённый моноид — это список. Пустой список является нейтральным элементом для списков, а операция `++` (конкатенация) — это операция `<>` для списков. Странные имена методов `Monoid` — это просто буква `m` (от слова `Monoid`), добавленная к обычным функциям для списков: `empty`, `append` и `concat`. Вы можете сравнить все три способа выполнения одной и той же операции со списком:

```
GHCi> [1, 2, 3] ++ []
[1, 2, 3]
GHCi> [1, 2, 3] <> []
[1, 2, 3]
GHCi> [1, 2, 3] `mappend` []
[1, 2, 3]
```

Обратите внимание: `mappend` имеет точно такой же тип, что и `<>`.

Проверка 17.4. Если вы реализуете `mappend`/`<>` для `Integer` с умножением вместо сложения, каким будет ваше значение `mempty`?

Ответ 17.4. 1, потому что $x * 1 = x$.

17.4. Комбинирование элементов моноида функцией mconcat

Самый простой способ увидеть мощь нейтрального элемента состоит в том, чтобы изучить последний метод в определении `Monoid` — функцию `mconcat`. Для определения экземпляра `Monoid` достаточно реализаций `mempty` и `mappend`.

Реализовав эти две функции, вы бесплатно получаете `mconcat`. Если вы посмотрите на её типовую аннотацию, то сможете получить хорошее представление о том, что она делает:

```
mconcat :: Monoid a => [a] -> [a]
```

Функция `mconcat` принимает список элементов моноида и комбинирует их, возвращая один элемент. Лучший способ разобраться с `mconcat` — взять список списков и посмотреть, что произойдёт при её применении. Чтобы было попроще, вы будете использовать строки, так как они являются просто списками символов:

```
GHCi> mconcat ["в", " этом", " есть", " смысл?"]
"в этом есть смысл?"
```

Отличная черта `mconcat` состоит в том, что как только вы определили `mempty` и `mappend`, Haskell может автоматически доопределить `mconcat`. Это связано с тем, что определение `mconcat` основывается на функциях `foldr` (урок 9), `mappend` и `mempty`. Это определение выглядит следующим образом:

```
mconcat = foldr mappend mempty
```

У любого метода из класса типов может быть определение по умолчанию, при условии что для реализации достаточно других методов класса.

17.4.1. Законы моноидов

Как и у `Semigroup`, у класса типов `Monoid` есть законы. Вот они:

(1) `mappend mempty x = x` — если вспомнить, что для списков `mappend` — то же самое, что и `(++)`, а `mempty` — то же, что и `[]`, этот закон интуитивно означает, что `[] ++ [1, 2, 3] = [1, 2, 3]`;

(2) `mappend x mempty = x` — то же, что и в первом, но наоборот, то есть в случае списков `[1, 2, 3] ++ [] = [1, 2, 3]`;

(3) `mappend x (mappend y z) = mappend (mappend x y) z` — ассоциативность, и снова для списков это выглядит довольно очевидно:

```
[1] ++ ([2] ++ [3]) = ([1] ++ [2]) ++ [3]
```

Так как этот закон имеет место для `Semigroup` и если `mappend` уже реализована как `<>`, то разумно предполагать выполнение этого закона;

(4) `mconcat = foldr mappend mempty` — это наше определение `mconcat`.

Обратите внимание: причина, по которой `mconcat` использует `foldr` вместо `foldl`, состоит в том, что `foldr` может работать с бесконечными списками, в то время как `foldl` требует полного вычисления.

17.4.2. Моноиды на практике: построение таблиц вероятностей

Теперь давайте посмотрим на более практическую задачу, которую вы можете решить с помощью моноидов. Вам нужно иметь возможность создавать таблицы вероятностей событий и иметь удобный способ их комбинировать. Для начала посмотрите на простую таблицу бросания монеты. У вас есть всего два события: выпадение орла или решки. В табл. 17.1 представлены вероятности этих событий.

Событие	**Вероятность**
Орёл	0.5
Решка	0.5

Таблица 17.1: Вероятность выпадения орла и решки

У вас есть список строк, представляющих события, и список вещественных чисел, представляющих вероятности:

Листинг 17.8 Синонимы типов Events и Probs

```
type Events = [String]
type Probs = [Double]
```

Таблица вероятностей — это просто список пар событий и вероятностей.

Листинг 17.9 Тип данных PTable

```
data PTable = PTable Events Probs
```

Теперь вам нужна функция для создания PTable. Эта функция будет простым конструктором, но также она будет обеспечивать равенство единице суммы всех вероятностей. Это легко реализовать, разделив каждую вероятность на сумму всех вероятностей.

Листинг 17.10 Создание корректной таблицы вероятностей

```
createPTable :: Events -> Probs -> PTable
createPTable events probs = PTable events normalizedProbs
  where totalProbs = sum probs
        normalizedProbs = map (\x -> x/totalProbs) probs
```

Вам не следует уходить слишком далеко, не сделав PTable экземпляром класса типов Show. Для начала вы должны создать простую функцию, печатающую одну строку таблицы.

Листинг 17.11 Создание строки для пары событие-вероятность

```
showPair :: String -> Double -> String
showPair event prob = mconcat [event, "|", show prob, "\n"]
```

Обратите внимание: для лёгкого комбинирования списков вы можете использовать mconcat. Раньше вы использовали операцию ++ для объединения строк. Оказывается, что mconcat не только позволяет меньше писать, но также предоставляет более удобную возможность комбинирования строк. Это связано с тем, что в Haskell есть некоторые другие типы (обсуждаемые в модуле 4), поддерживающие mconcat, но не ++.

Чтобы сделать PTable экземпляром класса Show, вам потребуется воспользоваться функцией zipWith, передав ей в качестве параметра функцию showPair. С функцией zipWith вы встречаетесь здесь впервые. Она «пристёгивает» списки друг к другу, применяя к их элементам определённую функцию. Ниже приведён пример суммирования двух списков:

```
GHCi> zipWith (+) [1, 2, 3] [4, 5, 6]
[5, 7, 9]
```

Листинг 17.12 Реализация экземпляра класса Show для PTable

```
instance Show PTable where
   show (PTable events probs) = mconcat pairs
     where pairs = zipWith showPair events probs
```

Вы можете проверить в GHCi реализацию нужного поведения:

```
GHCi> createPTable ["орёл", "решка"] [0.5, 0.5]
орёл|0.5
решка|0.5
```

Вам нужно с помощью класса типов `Monoid` иметь возможность комбинировать два (или более) элемента типа `PTable`. Например, если у вас есть две монетки, вы должны получить следующий результат:

```
орёл-орёл|0.25
орёл-решка|0.25
решка-орёл|0.25
решка-решка|0.25
```

Для этого нужно сгенерировать комбинации всех событий и всех вероятностей. Это называется *декартовым произведением*. Вы начнёте с обобщённой функции для получения декартова произведения двух списков. Функция `cartCombine` принимает три аргумента: функцию для комбинирования элементов списков и два списка.

Листинг 17.13 Вычисление декартова произведения списков

```
cartCombine :: (a -> b -> c) -> [a] -> [b] -> [c]
cartCombine func l1 l2 = zipWith func newL1 cycledL2
  where
    nToAdd = length l2
    repeatedL1 = map (take nToAdd . repeat) l1
    newL1 = mconcat repeatedL1
    cycledL2 = cycle l2
```

Нужно повторить каждый элемент первого списка для каждого элемента второго

Предыдущая строка даёт список списков, которые теперь нужно объединить

Создаём nToAdd копий каждого элемента l1

Зацикливаем второй список,что позволяет комбинировать их посредством zipWith

Теперь ваши функции для комбинирования элементов и вероятностей являются частными случаями `cartCombine`.

Листинг 17.14 Функции combineEvents и combineProbs

```
combineEvents :: Events -> Events -> Events
combineEvents e1 e2 = cartCombine combiner e1 e2
  where
    combiner = (\x y -> mconcat [x, "-", y])

combineProbs :: Probs -> Probs -> Probs
combineProbs p1 p2 = cartCombine (*) p1 p2
```

Разделяем названия комбинируемых событий дефисом

Перемножаем вероятности комбинируемых событий

Теперь с помощью функций combineEvents и combineProbs вы можете сделать PTable экземпляром Semigroup.

Листинг 17.15 Делаем PTable экземпляром Semigroup

```
instance Semigroup PTable where
  (<>) ptable1 (PTable [] []) = ptable1
  (<>) (PTable [] []) ptable2 = ptable2
  (<>) (PTable e1 p1) (PTable e2 p2) =
                                createPTable newEvents newProbs
    where newEvents = combineEvents e1 e2
          newProbs = combineProbs p1 p2
```

Нужно обработать случай пустой PTable

Наконец, вы можете реализовать экземпляр Monoid. Вам известно, что в этом классе типов mappend и <> означают одно и то же. Всё, что вам нужно, — определить нейтральный элемент. В данном случае им оказывается пустая таблица PTable [] []. Так выглядит ваш экземпляр Monoid для PTable:

Листинг 17.16 Делаем PTable экземпляром Monoid

```
instance Monoid PTable where
  mempty = PTable [] []
  mappend = (<>)
```

Не забывайте: вы получаете мощь mconcat совершенно бесплатно!

Чтобы увидеть, как это работает, давайте попробуем создать две таблицы вероятностей. Первая для классической монеты, а вторая — для рулетки с цветами и различными вероятностями выпадения каждого цвета.

Листинг 17.17 Примеры рулетки и монеты в PTable

```
coin :: PTable
coin = createPTable ["орёл", "решка"] [0.5, 0.5]

spinner :: PTable
spinner = createPTable ["красный", "синий", "зелёный"]
                       [0.1, 0.2, 0.7]
```

Если вы хотите узнать вероятность одновременного выпадения *решки* и *синего* цвета, то можете воспользоваться операцией <>:

```
GHCi> coin <> spinner
орёл-красный|5.0e-2
```

```
орёл-синий|0.1
орёл-зелёный|0.35
решка-красный|5.0e-2
решка-синий|0.1
решка-зелёный|0.35
```

По этому выводу вы можете увидеть, что вероятность выпадения *решки* и *синего* цвета равна 0.1, или 10%.

Что насчёт выпадения орла три раза подряд? Вы можете воспользоваться `mconcat` и упростить себе эту задачу:

```
GHCi> mconcat [coin, coin, coin]
орёл-орёл-орёл|0.125
орёл-орёл-решка|0.125
орёл-решка-орёл|0.125
орёл-решка-решка|0.125
решка-орёл-орёл|0.125
решка-орёл-решка|0.125
решка-решка-орёл|0.125
решка-решка-решка|0.125
```

В данном случае все исходы имеют одинаковую вероятность 12.5%.

Изначально идея «комбинирования сущностей» может показаться чересчур абстрактной. Но как только вы начинаете рассматривать задачи в терминах моноидов, становится заметно, как часто возникают задачи, связанные с комбинированием. Моноиды отлично демонстрируют мощь мышления в типах при написании кода.

Итоги

Целью этого урока было представить вам два интересных класса типов: `Semigroup` и `Monoid`. Хотя оба класса имеют достаточно странные имена, они играют относительно простую роль. `Monoid` и `Semigroup` позволяют вам комбинировать элементы типа, создавая новый элемент. Эта идея абстракции с помощью композиции является важной для Haskell. Единственное отличие `Monoid` от `Semigroup` состоит в том, что `Monoid` требует от вас определения нейтрального элемента. `Monoid` и `Semigroup` также представляют собой отличное введение в абстрактное мышление, используемое в более абстрактных классах типов. Здесь начинаются заметные различия в философии классов типов Haskell и интерфейсов в большинстве ООП-языков. Давайте проверим, что вы всё поняли.

Задача 17.1. Ваша текущая реализация `Color` не содержит нейтрального элемента. Перепишите свой код так, чтобы у `Color` был нейтральный элемент, а затем сделайте `Color` представителем класса `Monoid`.

Задача 17.2. Если бы ваши типы `Events` и `Probs` были самостоятельными типами данных, а не просто синонимами типов, вы могли бы сделать их экземплярами классов типов `Semigroup` и `Monoid` с операциями `combineEvents` и `combineProbs` в качестве <> соответственно. Перепишите эти типы данных и реализуйте экземпляры `Semigroup` и `Monoid`.

18

Параметризованные типы

После прочтения урока 18 вы:

- научитесь использовать параметризованные типы для создания произвольных типов данных;
- разберётесь с видами типов;
- сможете писать код, используя для поиска значений тип `Data.Map`.

В этом модуле мы уже обсудили, как типы можно почти как данные складывать и умножать. Как и функции, типы тоже могут принимать аргументы. Они это делают, используя в своих определениях типовые переменные (так что их аргументы — это всегда другие типы). Типы, определённые с использованием параметров, называются *параметризованными*. Параметризованные типы выполняют в Haskell важную роль, потому что они позволяют определять произвольные структуры данных, работающие с широким спектром существующих данных.

Обратите внимание. Допустим, вы хотите создать тип, представляющий пару из двух значений одного типа. Например, это может быть пара значений `Double`, означающих широту и долготу, или пара имён встречающихся друг с другом людей, или пара вершин графа, соединённых дугой. Вы не хотите использовать стандартные кортежи, потому что хотите быть уверены, что данные в вашей паре точно одного типа. Как вы можете это сделать?

18.1. Типы, которые принимают аргументы

Если вы знакомы с дженериками в языках вроде C# и Java, то параметризованные типы поначалу будут казаться знакомыми. Как и дженерики в C# и Java, параметризованные типы позволяют вам создавать «контейнеры», которые могут содержать значения других типов. Например, тип `List<String>` представляет собой список, содержащий только строки, а `KeyValuePair<int, string>` представляет пару значений, в которой `int` служит ключом к `string`. Обычно вы используете дженерики для ограничения типов значений, которые тип контейнера может принимать, чтобы с ним было проще работать. В Haskell верно то же самое.

Самый простой параметризованный тип, который вы можете определить, — это `Box`, служащий контейнером для любых других типов. Тип `Box` похож на функцию `simple`, но для параметризованных типов. На рис. 18.1 приведено его определение.

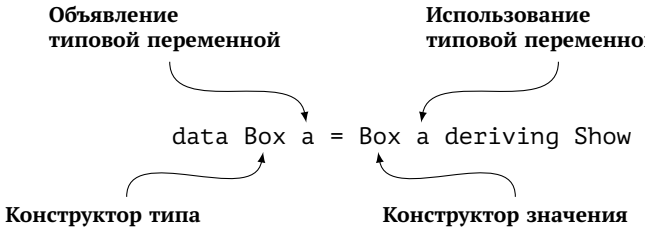


Рис. 18.1: Определение параметризованного типа `Box`

Тип `Box` — это абстрактный контейнер, который может хранить любые другие типы. Как только вы положите тип внутрь `Box`, тип `Box` примет конкретное значение. Вы можете использовать GHCi для изучения вопроса:

```
GHCi> n = 6 :: Int
GHCi> :t Box n
Box n :: Box Int
GHCi> word = "строка"
GHCi> :t Box word
Box word :: Box [Char]
GHCi> f x = x
GHCi> :t Box f
Box f :: Box (t -> t)
GHCi> otherBox = Box n
GHCi> :t Box otherBox
Box otherBox :: Box (Box Int)
```

Вы также можете определять простые функции для `Box` вроде `wrap` и `unwrap`, чтобы класть предметы в контейнер или доставать их из него.

Листинг 18.1 Определения функций wrap и unwrap для Box

```
wrap :: a -> Box a
wrap x = Box x

unwrap :: Box a -> a
unwrap (Box x) = x
```

Обратите внимание, что обе эти функции не знают точного типа `Box`, но могут работать с ним.

Проверка 18.1. Каков тип `wrap (Box 'a')`?

18.1.1. Более полезный параметризованный тип: Triple

Как и функция `simple`, тип `Box` слишком примитивен, чтобы быть хоть немного полезным. Гораздо более полезным является контейнер `Triple`, который мы определим как три одинаковых значения.

Листинг 18.2 Определение типа Triple

```
data Triple a = Triple a a a deriving Show
```

Стоит отметить, что `Triple` — это не то же самое, что кортеж из трёх элементов `(a,b,c)`. Кортежи в Haskell могут содержать значения разных типов. В типе `Triple` значения должны быть одного типа. Есть много случаев на практике, когда значения обладают этим свойством. Например, точки трёхмерного пространства могут рассматриваться как `Triple` типа `Double`.

Листинг 18.3 Точка в трёхмерном пространстве как Triple

```
type Point3D = Triple Double
aPoint :: Point3D
aPoint = Triple 0.1 53.2 12.3
```

Ответ 18.1

```
Box (Box Char)
```

Имена людей могут быть представлены как `Triple` типа `String`.

Листинг 18.4 Использование Triple для определения типа имён

```
type FullName = Triple String

aPerson :: FullName
aPerson = Triple "Говард" "Филлипс" "Лавкрафт"
```

Подобным образом инициалы — это `Triple` типа `Char`.

Листинг 18.5 Тип Triple для определения типа инициалов

```
type Initials = Triple Char

initials :: Initials
initials = Triple 'H' 'P' 'L'
```

Теперь, когда у вас есть модель для однородного `Triple`, вы можете писать функции, которые будут работать сразу для всех приведённых выше примеров. Первое, что вам нужно сделать, — это создать способ получать доступ к значениям в `Triple`. Оказывается, что `fst` и `snd` определены только на кортежах с двумя значениями, в больших кортежах нет возможности получать доступ к их значениям.

Листинг 18.6 Функции-аксессоры для типа Triple

```
first :: Triple a -> a
first (Triple x _ _) = x

second :: Triple a -> a
second (Triple _ x _ ) = x

third :: Triple a -> a
third (Triple _ _ x) = x
```

Вы также легко можете превратить ваш `Triple` в список.

Листинг 18.7 Определение функции toList на Triple

```
toList :: Triple a -> [a]
toList (Triple x y z) = [x,y,z]
```

Наконец, вы можете создать простой инструмент для преобразования любого `Triple` в `Triple` того же типа.

Листинг 18.8 Функция для преобразования Triple

```
transform :: (a -> a) -> Triple a -> Triple a
transform f (Triple x y z) = Triple (f x) (f y) (f z)
```

Такой тип преобразования чрезвычайно полезен. Теперь вы можете умножить все компоненты точки в трёхмерном пространстве на константное значение:

ЛАНЬ

```
GHCi> transform (* 3) aPoint
Triple 0.3 159.6 36.9
```

Или перевернуть все буквы в имени человека:

```
GHCi> transform reverse aPerson
Triple "дравоГ" "спиллиФ" "тфарквaЛ"
```

Или вы можете импортировать `Data.Char` и перевести инициалы в нижний регистр:

ЛАНЬ

```
GHCi> transform toLower initials
Triple 'h' 'p' 'l'
```

Комбинируя последнее преобразование с `toList`, вы можете получить строку с инициалами в нижнем регистре:

```
GHCi> toList (transform toLower initials)
"hpl"
```

Проверка 18.2. В чём отличие между функциями `transform` и `map` для списков? (Подсказка: ещё раз посмотрите на тип функции `map`.)

Ответ 18.2. Функция `transform` не позволяет изменить тип, применив её, к примеру, с функцией `a -> b`. Функция `map` для списков эту возможность поддерживает.

18.1.2. Списки

Самый распространённый параметризованный тип — это `List`. Тип `List` интересен благодаря своему конструктору, отличному от тех, которые были у большинства типов, которые вы видели. Как вы знаете, для создания списка и размещения в нём значений мы используем квадратные скобки. Это сделано для удобства, но делает поиск информации о списках более сложным, чем о типах, которые имеют более традиционные конструкторы типов. В GHCi вы можете получить больше информации о списках, используя команду `:info []`. На рис. 18.2 показано формальное определение типа списка.

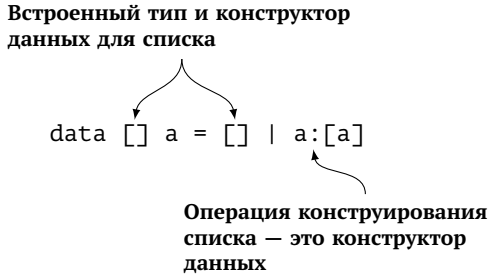


Рис. 18.2: Определение списка

Поражает, что это полная вполне работающая реализация списка! Если вы когда-либо писали связные списки в других языках программирования, это может стать для вас сюрпризом. Чтобы лучше это понять, давайте перепишем список самостоятельно. Специальное использование квадратных скобок — это встроенный синтаксис, повторить его нам не удастся. Мы также не сможем воспользоваться конструктором данных `(:)`. Поэтому воспользуемся словами `List`, `Cons` и `Empty`. Вот определение.

Листинг 18.9 Определение вашего собственного списка

```
data List a = Empty | Cons a (List a) deriving Show
```

Обратите внимание, что определение `List` рекурсивно! На естественном языке вы можете читать это определение следующим образом: «Список типа `a` либо пуст, либо конструируется из значения типа `a` с другим списком типа `a`». Может быть, тяжело поверить в то, что это определение типа является завершённым определением вашей структуры данных `List`! Однако ниже приведено доказательство, эти списки действительно идентичны.

Листинг 18.10 Сравнение вашего списка со встроенным

```
builtinEx1 :: [Int]
builtinEx1 = 1:2:3:[]

ourListEx1 :: List Int
ourListEx1 = Cons 1 (Cons 2 (Cons 3 Empty))

builtinEx2 :: [Char]
builtinEx2 = 'к':'о':'т':[]

ourListEx2 :: List Char
ourListEx2 = Cons 'к' (Cons 'о' (Cons 'т' Empty))
```

В качестве финальной демонстрации вы можете реализовать для своего списка функцию `map` и использовать её в GHCi.

Листинг 18.11 Определяем ourMap для своего списка

```
ourMap :: (a -> b) -> List a -> List b
ourMap _ Empty = Empty
ourMap func (Cons a rest)  = Cons (func a) (ourMap func rest)

GHCi> ourMap (*2) ourListEx1
Cons 2 (Cons 4 (Cons 6 Empty))
```

Теперь вы знаете, что когда на собеседовании при приёме на работу вас попросят реализовать связный список, вашим первым вопросом должен быть «Могу ли я сделать это на Haskell?».

18.2. Типы с более чем одним параметром

Как и функции, типы могут принимать больше одного аргумента. Важно помнить, что если типовых параметров больше, чем один, то тип может быть контейнером для более чем одного типа. Это отличается от хранения нескольких значений одного типа, как в `Triple`.

18.2.1. Кортежи

Кортежи — это самый широко используемый многопараметрический тип в Haskell и единственный многопараметрический тип, который вы видели на данный момент. Как и списки, кортежи используют встроенный

конструктор (). Если вы используете :info на кортеже, вам нужно использовать (), в которых будет по одной запятой для каждого из n-1 элемента кортежа. Например, если вы хотите определение кортежа из двух элементов, вам нужно напечатать в GHCi :info (,). Вот встроенное определение:

Листинг 18.12 Определение кортежа

```
data (,) a b = (,) a b
```

Обратите внимание, что определение кортежа из двух элементов содержит две типовые переменные. Как мы уже упоминали, это позволяет кортежу содержать значения двух типов. В языках с динамической типизацией вроде Python, Ruby и JavaScript обычные списки могут содержать значения различных типов. Важно понимать, что кортежи в Haskell не такие же, как списки в этих языках. Причина в том, что после того, как вы создали ваш тип, он принимает конкретные значения. Это заметно лучше всего, если вы попробуете создать список кортежей. Допустим, у вас есть система инвентарного учёта, которая отслеживает предметы и их количество.

Листинг 18.13 Исследуем типы кортежей

```
itemCount1, itemCount2, itemCount3 :: (String,Int)
itemCount1 = ("Стёрки",25)
itemCount2 = ("Карандаши",25)
itemCount3 = ("Ручки",13)
```

Вы можете создать список для отслеживания инвентаря:

Листинг 18.14 Список элементов инвентаря

```
itemInventory :: [(String,Int)]
itemInventory = [itemCount1, itemCount2, itemCount3]
```

Обратите внимание, что здесь указан конкретный тип: (String, Int).

Проверка 18.3. А что, если добавить в инвентарь ("Бумага",12.4)?

Ответ 18.3. Это приведёт к ошибке, так как ваши пары имеют тип (String, Int), а ("Бумага",12.4) имеет тип (String,Double).

18.2.2. Виды: типы типов

Типы в Haskell похожи на функции и данные ещё и тем, что они тоже имеют собственные типы! Тип типа называется его *видом* (*kind*). Виды ожидаемо довольно абстрактны. Но они встретятся, когда вы погрузитесь в более продвинутые классы типов, рассматриваемые в модуле 5 (`Functor`, `Applicative` и `Monad`). Вид типа отражает количество параметров, принимаемых типом, обозначая их звёздочками (`*`). Тип, который не принимает параметров, имеет вид `*`, типы, принимающие один параметр, имеют вид `* -> *`, типы с тремя параметрами — `* -> * -> *` и так далее.

Команда GHCi `:kind` позволяет смотреть виды типов, в которых вы сомневаетесь (убедитесь, что импортировали модуль `Data.Map`):

```
GHCi> :kind Int
Int :: *
GHCi> :kind Triple
Triple :: * -> *
GHCi> :kind []
[] :: * -> *
GHCi> :kind (,)
(,) :: * -> * -> *
GHCi> :kind Map.Map
Map.Map :: * -> * -> *
```

Отметим, что конкретные типы имеют отличные от своих неконкретных аналогов виды:

```
GHCi> :kind [Int]
[Int] :: *
GHCi> :kind Triple Char
Triple Char :: *
```

Поначалу виды могут показаться абстрактной чепухой. Но понимание видов может быть полезно при создании экземпляров классов типов `Functor` и `Monad` (которые мы рассмотрим в модуле 5).

Проверка 18.4. Каков вид `(,,)`?

Ответ 18.4. `(,,) :: * -> * -> * -> *`

18.2.3. Модуль Data.Map

Ещё один полезный параметризованный тип Haskell — это `Map` (не путайте с функцией `map`). Чтобы им воспользоваться, вам нужно импортировать модуль `Data.Map`. Так как имена некоторых функций модуля `Data.Map` совпадают с именами из `Prelude`, вам нужно сделать квалифицированный импорт (детали приведены на рис. 18.3).

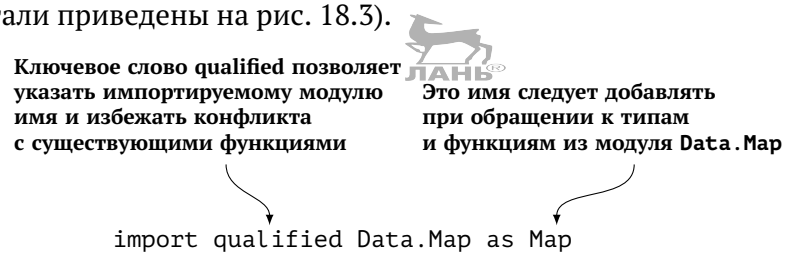


Рис. 18.3: Использование квалифицированного импорта

Выполненный таким образом квалифицированный импорт предполагает, чтобы каждая функция и тип из этого модуля предварялись префиксом `Map`. Тип `Map` позволяет вам искать значения при помощи ключей. Во многих других языках этот тип называется `Dictionary` (словарь). Типовые параметры `Map` — это типы ключей и значений. В отличие от списков и кортежей, реализация `Map` нетривиальна. Проще всего разобраться с этим типом на конкретном примере.

Допустим, вы безумный учёный и у вас есть список чисел, которые соответствуют различным органам, используемым для создания отвратительных монстров. Вы можете по-быстрому начать с определения типа-суммы для соответствующих частей тела.

Листинг 18.15 Тип данных Organ

```
data Organ = Heart | Brain | Kidney | Spleen
  deriving (Show, Eq)
```

Допустим, в ваших запасниках имеются следующие органы (повторы — это нормально, селезёнок вечно не хватает!):

Листинг 18.16 Пример списка органов

```
organs :: [Organ]
organs = [Heart,Heart,Brain,Spleen,Spleen,Kidney]
```

Каждый орган размещён в пронумерованном ящике, и позже его оттуда можно будет достать. На каждый ящик нанесён его номер. Так как мы

хотим использовать ящики для поиска предметов, номер каждого из них должен быть уникален . Более того, какой бы идентификатор вы не использовали, он должен принадлежать к классу `Ord`. Если у шкафчиков не будет порядка, организовать эффективный поиск органов будет сложно!

Словари и хеш-таблицы

Словари похожи на другую структуру данных — хеш-таблицы. Обе они позволяют искать значения по ключам. Основное отличие между этими структурами состоит в способе поиска значений. В хеш-таблице функция превращает ключ в индекс массива, в котором хранится значение. Это позволяет быстро искать значения, но требует большого объёма памяти, необходимого для предотвращения коллизий. Словарь ищет значения при помощи бинарного дерева поиска. Это медленнее хеш-таблицы, но всё же достаточно быстро. Словарь ищет значения при помощи ключей, принадлежащих классу `Ord`, чтобы можно было сравнивать два ключа и эффективно находить их в дереве.

Вот список идентификаторов (не в каждом ящике что-то есть, так что в списке имеются промежутки).

Листинг 18.17 Список идентификаторов ящиков с органами

```
ids :: [Int]
ids = [2,7,13,14,21,24]
```

Теперь у вас есть вся необходимая информация для построения `Map`! Он будет служить каталогом ящиков, так что вы легко сможете увидеть, какие предметы лежат в нужном вам ящике.

Проще всего строить `Map` функцией `fromList`. Используя в GHCi команду `:t`, вы можете узнать её тип (мы его приводим на рис. 18.4).

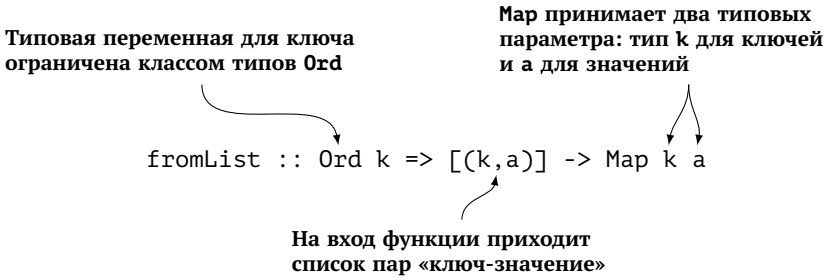


Рис. 18.4: Функция `fromList` для построения `Map`

Теперь вы можете рассмотреть типовые параметры Map: k и a. Заметьте, тип ключа k должен принадлежать к классу Ord. Это ограничение необходимо из-за внутренних способов хранения и поиска ключей.

Также важно заметить, что функция fromList ожидает список кортежей, которые представляют ключи и значения. Вы можете переписать ваши списки следующим образом:

Листинг 18.18 Пары органов и идентификаторов, версия 1

```
pairs = [(2,Heart),(7,Heart),(13,Brain) ...
```

Правда, для больших списков это будет очень неудобно! Вместо этого вы можете использовать полезную функцию zip из урока 6. Функция zip принимает два списка и возвращает список пар.

Листинг 18.19 Создание списка пар органов с идентификаторами

```
organPairs :: [(Int,Organ)]
organPairs = zip ids organs
```

Теперь у вас есть всё необходимое для создания каталога органов.

Листинг 18.20 Создание каталога органов

```
organCatalog :: Map.Map Int Organ
organCatalog = Map.fromList organPairs
```

Наконец-то вы можете искать предметы при помощи Map.lookup. Если вы попробуете сделать это в GHCi, то получите интересный результат:

```
GHCi> Map.lookup 7 organCatalog
Just Heart
```

Вы получаете Heart, как и ожидалось, но оно предваряется конструктором данных Just. Если вы посмотрите на типовую аннотацию Map.lookup, то увидите следующее:

Листинг 18.21 Типовая аннотация функции Map.lookup

```
Map.lookup :: Ord k => k -> Map.Map k a -> Maybe a
```

Оказывается, функция Map.lookup возвращает новый параметризованный тип Maybe. Тип Maybe — это простой, но крайне мощный тип, который является предметом нашего следующего урока!

Итоги

В этом уроке нашей целью было рассказать вам о параметризованных типах. Параметризованные типы — это типы, которые принимают один или больше аргументов (как дженерики во многих объектно-ориентированных языках). Самый частый пример параметризованного типа — это список, который может содержать элементы любого типа. Параметризованные типы могут принимать любое число типов в качестве аргументов. Количество типов, которое параметризованный тип принимает в качестве аргументов, определено его видом. `Map` — это параметризованный тип, который принимает два аргумента — один для типа ключей, а другой для типа значений. Давайте посмотрим, как вы это поняли.

Задача 18.1. Реализуйте аналогичные `map` функции `tripleMap` и `boxMap` для типов `Box` и `Triple`.

Задача 18.2. Модифицируйте тип `Organ` так, чтобы он мог быть использован в качестве ключа. Затем постройте `Map` под названием `organInventory`, где сам орган будет ключом, а количество таких органов в `organCatalog` — значением.

19

Тип Maybe: работа с отсутствующими значениями

После прочтения урока 19 вы:

- разберётесь с тем, как работает тип Maybe;
- начнёте использовать тип Maybe для работы с отсутствующими значениями;
- сможете проектировать программы с использованием типа Maybe.

Как классы типов зачастую гораздо более абстрактны, нежели интерфейсы в ООП, так и параметризованные типы играют гораздо большую роль, чем типы-дженерики в большинстве языков. В этом уроке представлен важный параметризованный класс типов `Maybe`. В отличие от `List` или `Map`, представляющих контейнеры для значений, `Maybe` — первый из увиденных вами типов, представляющих *контекст* значения. Тип `Maybe` представляет значения, которые могут отсутствовать. В большинстве языков программирования отсутствующее значение представляют с помощью значения `null`. Посредством использования контекста, представляющего значение, которое может отсутствовать, тип `Maybe` позволяет вам писать гораздо более безопасный код. Благодаря типу `Maybe` ошибки, связанные с `null`-значениями, систематически из Haskell-программ удаляются.

Обратите внимание. Предположим, есть список покупок в виде `Map`, содержащий названия и соответствующие им количества товаров:

```
groceries :: Map.Map String Int
groceries = Map.fromList [("Молоко", 1), ("Конфеты", 10),
                          ("Сыр", 2)]
```

Вы случайно выполнили поиск по ключу «МОЛОКО» вместо «Молоко». Какого поведения вам стоит ожидать от своего `Map` и как вы можете обработать такую ошибку, чтобы ваши программы гарантированно работали без ошибок даже при отсутствии значений в `Map`?

19.1. Maybe: возможность отсутствия значения как тип

В конце 18-го урока вы работали на безумного учёного, собирающего коллекцию человеческих органов. С целью облегчения поиска для хранения списка органов использовался тип `Map`. Давайте продолжим работу над этим упражнением. Ниже приведён важный код из предыдущего урока с добавленной реализацией экземпляра класса `Show` для более приятного вывода названий органов:

```
import qualified Data.Map as Map

data Organ = Heart | Brain | Kidney | Spleen deriving (Eq)

instance Show Organ where
  show Heart = "Сердце"
  show Brain = "Мозги"
  show Kidney = "Почка"
  show Spleen = "Селезёнка"

organs :: [Organ]
organs = [Heart, Heart, Brain, Spleen, Spleen, Kidney]

ids :: [Int]
ids = [2, 7, 13, 14, 21, 24]

organPairs :: [(Int, Organ)]
organPairs = zip ids organs

organCatalog :: Map.Map Int Organ
organCatalog = Map.fromList organPairs
```

В прошлом уроке всё шло хорошо до тех пор, пока вы не решили воспользоваться функцией Map.lookup для поиска значения типа Organ в вашем Map. Попытавшись это сделать, вы столкнулись с новым типом Maybe.

Maybe — простой, но мощный тип. До сих пор все наши параметризованные типы рассматривались как контейнеры. Maybe не таков. Maybe лучше представлять себе как тип в контексте. Контекст в данном случае заключается в том, что значение находящегося в нём типа данных может отсутствовать. Определение типа Maybe выглядит следующим образом:

Листинг 19.1 Определение Maybe

```
data Maybe a = Nothing | Just a
```

Значением типа Maybe может быть Nothing или Just с каким-то значением типа a. Как это понимать? Давайте посмотрим в GHCi, что происходит:

```
GHCi> Map.lookup 13 organCatalog
Just Мозги
```

Когда вы выполняете поиск в каталоге по идентификатору, то получаете конструктор данных Just и значение, которого ожидали для этого идентификатора. Если вы посмотрите на тип этого значения, то получите следующее:

```
Map.lookup 13 organCatalog :: Maybe Organ
```

В определении lookup возвращаемое значение имеет тип Maybe a. В результате применения функции lookup тип возвращаемого значения стал конкретным, а именно Maybe Organ. Тип Maybe Organ говорит: «это значение *может быть* экземпляром типа Organ». А когда это может быть не так? Давайте посмотрим, что произойдёт, если вы запросите значение по идентификатору, которого в данном Map нет:

```
GHCi> Map.lookup 6 organCatalog
Nothing
```

Проверка 19.1. Какой тип имеет Nothing в предыдущем примере?

Ответ 19.1. Тип Nothing — Maybe Organ.

19.2. Проблема с null

Наш каталог органов `organCatalog` не содержит значения для идентификатора 6. В большинстве языков программирования при обращении по ключу, отсутствующему в словаре, происходит одно из двух: вы получаете ошибку или значение null. С обоими этими вариантами связаны серьёзные проблемы.

19.2.1. Обработка отсутствующих значений с помощью ошибок

В случае с выбрасыванием ошибок многие языки программирования не требуют, чтобы вы перехватывали ошибки, которые могут возникнуть. Если программа запрашивает ключ, отсутствующий в словаре, программист должен помнить о необходимости обработать ошибку, чтобы вся программа из-за этого не сломалась. Помимо этого, ошибка должна быть обработана в момент генерации исключения. Это может не казаться большой проблемой, потому что перехват проблемы на корню может быть неплохой идеей. Но представьте, что вы хотите обработать случай отсутствия `Spleen` не так, как случай отсутствия `Heart`. Во время возникновения ошибки отсутствия идентификатора у вас может не быть достаточно информации, чтобы нужным способом обработать разные случаи отсутствия значения.

19.2.2. Возврат значения null

С возвращением `null` может быть связано большее количество проблем. Самое большое затруднение состоит в том, что программисту опять необходимо помнить о необходимости проверки значений на null во всех местах, где null может встретиться. Нет способа, которым программа могла бы заставить программиста выполнять эти проверки. Null-значения также крайне подвержены ошибкам в связи с тем, что они обычно ведут себя не так, как значения, ожидаемые вашей программой. Простой вызов `toString` легко может привести к возникновению в программе ошибки, связанной со значением null. Если вы являетесь разработчиком Java или C#, выражение *null pointer exception* должно быть достаточным аргументом в поддержку сложности работы с null-значениями.

19.2.3. Maybe и отсутствующие значения

`Maybe` решает все эти проблемы умным способом. Когда функция возвращает значение типа `Maybe`, программа не может использовать это зна-

чение, не разобравшись с тем, что именно завёрнуто в Maybe. Отсутствующие значения не могут вызвать ошибку в программе на Haskell, потому что Maybe не оставляет возможности забыть, что значение может быть null. В то же время об этом без крайней необходимости не приходится заботиться программисту. Maybe используется во всех типичных для появления null местах, включая следующие:

- открытие файлов, которые могут не существовать;
- чтение из базы данных, которая может хранить null-значения;
- отправка REST API-запроса к потенциально отсутствующему ресурсу.

Лучший способ проиллюстрировать магию Maybe — с помощью кода. Допустим, вы по-прежнему помощник безумного учёного. Периодически вам нужно производить инвентаризацию, чтобы узнать, каких частей тел не хватает. Вы никак не можете запомнить, что лежит в каком шкафчике или хотя бы лежит ли в шкафчике что-нибудь вообще. Единственный способ проверить все шкафчики — использовать каждый идентификатор в диапазоне от 1 до 50.

Листинг 19.2 Список шкафчиков для каталога органов

```
possibleDrawers :: [Int]
possibleDrawers = [1..50]
```

Теперь вам нужна функция для получения содержимого шкафчика. Приведённая ниже функция отображает список возможных шкафчиков с помощью функции lookup.

Листинг 19.3 Определение getDrawerContents

```
getDrawerContents :: [Int] -> Map.Map Int Organ ->
                                                [Maybe Organ]
getDrawerContents ids catalog = map getContents ids
  where getContents = \id -> Map.lookup id catalog
```

С помощью этой функции вы можете выполнить поиск в каталоге:

Листинг 19.4 Список органов с отсутствующими значениями

```
availableOrgans :: [Maybe Organ]
availableOrgans = getDrawerContents possibleDrawers
                                    organCatalog
```

В языке программирования, выбрасывающем исключения или возвращающем null, ваша программа бы уже «взорвалась». Обратите внима-

ние: тип возвращаемого значения — список, содержащий элементы типа `Maybe Organ`. Вы избежали проблемы с возвращением специального значения null. Что бы вы ни делали с этим списком, до тех пор пока вы явно не разберётесь с возможным отсутствием значений, вам придётся держать данные в типе `Maybe`.

Последнее, что вам нужно сделать, — подсчитать количество определённых органов в вашем списке. Для этого вам нужно разобраться с `Maybe`.

Листинг 19.5 Функция countOrgan для подсчёта количества органов

```
countOrgan :: Organ -> [Maybe Organ] -> Int
countOrgan organ available = length (filter
                                     (\x -> x == Just organ)
                                     available)
```

Интересный момент заключается в том, что вам даже не пришлось «вытаскивать» орган из контекста `Maybe`. `Maybe` реализует класс типов `Eq`, так что вы можете просто сравнить две сущности типа `Maybe Organ`. Вам не только не нужно обрабатывать никакие ошибки, но также, в связи с тем что ваше вычисление нигде явно не работает с несуществующими значениями, вы можете не волноваться об обработке этого случая! Итоговый результат в GHCi выглядит следующим образом:

```
GHCi> countOrgan Brain availableOrgans
1
GHCi> countOrgan Heart availableOrgans
2
```

19.3. Вычисления с Maybe

Было бы полезно иметь возможность напечатать список имеющихся органов `availableOrgans`, чтобы увидеть, что у вас имеется. Типы `Organ` и `Maybe` имеют экземпляры класса `Show`, так что вы можете вывести этот список в GHCi:

```
GHCi> show availableOrgans [Nothing, Just Сердце, Nothing,
↳ Nothing, Nothing, Nothing, Just Сердце, Nothing, Nothing, ...
```

Хотя вы и получили возможность печати автоматически, такой вывод выглядит не очень красиво. Во-первых, вам стоит убрать все значения `Nothing`. Для этого вы можете использовать функцию `filter` и сопоставление с образцом.

Листинг 19.6 Определение isSomething

```
isSomething :: Maybe Organ -> Bool
isSomething Nothing = False
isSomething (Just _) = True
```

Теперь вы можете отфильтровать список и получить только органы, имеющиеся в наличии.

Листинг 19.7 Использование isSomething и filter для очистки вывода

```
justTheOrgans :: [Maybe Organ]
justTheOrgans = filter isSomething availableOrgans
```

В GHCi вы можете увидеть, что стало значительно лучше:

```
GHCi> justTheOrgans
[Just Сердце,Just Сердце,Just Мозги,Just Селезёнка,
↳ Just Селезёнка,Just Почка]
```

Функции isJust и isNothing

Модуль `Data.Maybe` содержит функции `isJust` и `isNothing` для обработки значений в `Just`. `isJust` идентична функции `isSomething`, но работает для всех типов `Maybe`. Импортировав `Data.Maybe`, вы могли бы решить предыдущую задачу следующим образом:

```
justTheOrgans = filter isJust availableOrgans
```

Проблема состоит в том, что у вас всё ещё остались конструкторы данных `Just`, в которые «упаковано» всё остальное. Вы можете избавиться от них с помощью сопоставления с образцом. Давайте напишем функцию `showOrgan`, превращающую `Maybe Organ` в `String`. Она будет производить сопоставление с образцом `Nothing`, даже несмотря на то что использовать его не придётся, так как хорошей привычкой является сопоставление со всеми возможными образцами просто на всякий случай.

Листинг 19.8 Определение showOrgan

```
showOrgan :: Maybe Organ -> String
showOrgan (Just organ) = show organ
showOrgan Nothing = ""
```

Вот несколько примеров использования этой функции в GHCi:

```
GHCi> showOrgan (Just Heart)
"Сердце"
GHCi> showOrgan Nothing
""
```

Теперь вы можете отобразить список `justTheOrgans` с помощью функции `showOrgan`:

Листинг 19.9 Использование showOrgan с map

```
organList :: [String]
organList = map showOrgan justTheOrgans
```

В качестве последнего штриха вы добавите запятые, чтобы список выглядел красивее. Вы можете воспользоваться функцией `intercalate` (красивый синоним `insert`) из модуля `Data.List` (так что вам придётся добавить `import Data.List` в начало своего файла):

```
cleanList :: String
cleanList = intercalate ", " organList

GHCi> cleanList
"Сердце, Сердце, Мозги, Селезёнка, Селезёнка, Почка"
```

Проверка 19.2. Напишите функцию `numOrZero`, которая принимает `Maybe Int` и возвращает 0, если это `Nothing`, и число в противном случае.

19.4. Назад в лабораторию! Снова вычисления с Maybe

Допустим, вам нужно выполнить несколько вычислений со значением в `Maybe`. У безумного учёного есть более интересный проект. Вам дают

Ответ 19.2

```
numOrZero :: Maybe Int -> Int
numOrZero Nothing = 0
numOrZero (Just n) = n
```

идентификатор шкафчика. Вы должны достать из него орган. Затем вам нужно положить этот орган в подходящий контейнер (в ванночку, холодильник или сумку). Наконец, вы должны поставить контейнер в подходящее место. Ниже приведены правила для контейнеров и мест.

Для контейнеров:

- мозги нужно класть в ванночку;
- сердца нужно класть в холодильник;
- селезёнки и почки нужно класть в сумку.

Для мест:

- ванночки и холодильники нужно ставить в лабораторию;
- сумки нужно класть на кухню.

Для начала напишем это всё так, как будто всё идёт хорошо и вам не нужно волноваться о `Maybe`.

Листинг 19.10 Основные функции и типы данных

```
data Container = Vat Organ | Cooler Organ | Bag Organ

instance Show Container where
  show (Vat organ) = show organ ++ " в ванночке"
  show (Cooler organ) = show organ ++ " в холодильнике"
  show (Bag organ) = show organ ++ " в сумке"

data Location = Lab | Kitchen | Bathroom

instance Show Location where
  show Lab = "лаборатория"
  show Kitchen = "кухня"
  show Bathroom = "душевая"

organToContainer :: Organ -> Container
organToContainer Brain = Vat Brain
organToContainer Heart = Cooler Heart
organToContainer organ = Bag organ

placeInLocation :: Container -> (Location, Container)
placeInLocation (Vat a) = (Lab, Vat a)
placeInLocation (Cooler a) = (Lab, Cooler a)
placeInLocation (Bag a) = (Kitchen, Bag a)
```

Функция process будет принимать Organ и помещать его в подходящие контейнер и локацию. Затем функция report будет принимать контейнер и локацию и возвращать отчёт для безумного учёного.

Листинг 19.11 Функции process и report

```
process :: Organ -> (Location, Container)
process organ = placeInLocation (organToContainer organ)

report :: (Location, Container) -> String
report (location, container) =
   show container ++
   " (место: " ++
   show location ++ ")"
```

Эти функции написаны *в предположении, что нет отсутствующих органов*. Вы можете проверить их работу, перед тем как начать приспосабливать их к работе с настоящим каталогом:

```
GHCi> process Brain
(лаборатория,Мозги в ванночке)
GHCi> process Heart
(лаборатория,Сердце в холодильнике)
GHCi> process Spleen
(кухня,Селезёнка в сумке)
GHCi> process Kidney
(кухня,Почка в сумке)
GHCi> report (process Brain)
"Мозги в ванночке (место: лаборатория)"
GHCi> report (process Spleen)
"Селезёнка в сумке (место: кухня)"
```

Вы всё ещё не разобрались с получением Maybe Organ из каталога. В Haskell типы, такие как Maybe, помогают справиться при программировании со множеством ситуаций, в которых могли бы возникнуть проблемы. То, что вы сделали с функцией process, является распространённым шаблоном программирования в Haskell: вы отделили части кода, в которых вам нужно было бы волноваться о некоторой проблеме (например, об отсутствующих значениях), от частей кода, в которых такого нет. В отличие от большинства других языков программирования, значения типа Maybe не могут случайно пробраться в функцию process. Представьте себе, вы можете написать код, в который не могут попасть null-значения.

Теперь давайте соберём всё вместе, чтобы вы могли получать данные из своего каталога. Вам нужно что-то вроде следующей функции, только нужно ещё разобраться с Maybe:

Листинг 19.12 Неработающее определение функции processRequest

```
processRequest :: Int -> Map.Map Int Organ -> String
processRequest id catalog = report (process organ)
  where organ = Map.lookup id catalog
```

Проблема состоит в том, что значение `organ` имеет тип `Maybe Organ`, а функция `process` принимает значение типа `Organ`. Чтобы решить эту проблему, пользуясь известными вам средствами, вы должны скомбинировать функции `report` и `process` в функцию, работающую с `Maybe Organ`.

Листинг 19.13 Функция processAndReport для работы с Maybe Organ

```
processAndReport :: (Maybe Organ) -> String
processAndReport (Just organ) = report (process organ)
processAndReport Nothing = "ошибка, идентификатор не найден"
```

Теперь вы можете использовать эту функцию для обработки запроса.

Листинг 19.14 processRequest с поддержкой Maybe Organ

```
processRequest :: Int -> Map.Map Int Organ -> String
processRequest id catalog = processAndReport organ
  where organ = Map.lookup id catalog
```

Это решение работает хорошо. Как вы можете проверить в GHCi, функция успешно работает с существующими и несуществующими органами.

```
GHCi> processRequest 13 organCatalog
"Мозги в ванночке (место: лаборатория)"
GHCi> processRequest 12 organCatalog
"ошибка, идентификатор не найден"
```

Есть одна небольшая проблема с точки зрения архитектуры. Сейчас ваша функция `processRequest` обрабатывает отчёты с ошибками. В идеале вам бы хотелось, чтобы это делала функция `process`. Но чтобы сделать это, используя только имеющиеся у вас на текущий момент знания, вам бы пришлось переписать `process` так, чтобы она принимала `Maybe`. Вы бы оказались в худшей ситуации, так как больше не смогли бы пользоваться преимуществом функции обработки, при написании которой гарантированно можно не заботиться об отсутствующем значении.

Проверка 19.3. Как бы вы переписали функцию `report`, чтобы она работала с `Maybe (Location, Container)` и обрабатывала случай отсутствия `Organ`?

Итоги

В этом уроке нашей целью было представить вам один из самых интересных параметризованных типов Haskell — тип `Maybe`. Тип `Maybe` позволяет вам моделировать значения, которые могут отсутствовать. Это достигается благодаря использованию двух конструкторов данных `Just` и `Nothing`. Значения, представленные с помощью конструктора данных `Nothing`, отсутствуют. Значения, представленные через конструктор данных `Just a`, могут быть безопасно обработаны с помощью сопоставления с образцом. `Maybe` является отличным примером того, как мощь параметризованных типов позволяет вам писать код, менее подверженный ошибкам. Благодаря типу `Maybe` полностью устранён класс ошибок, связанный с наличием null-значений. Давайте удостоверимся, что вы всё поняли.

Задача 19.1. Реализуйте функцию `emptyDrawers`, принимающую результат вызова функции `geDrawerContents` и возвращающую количество пустых шкафчиков.

Задача 19.2. Реализуйте аналогичную функции `map` функцию под названием `maybeMap`, работающую с типами `Maybe`.

Ответ 19.3

```
report :: Maybe (Location, Container) -> String
report Nothing = "контейнер не найден"
report (Just (location, container)) = show container ++
             " (место: " ++ show location ++ ")"
```

20

Итоговый проект: временные ряды

Этот итоговый проект включает в себя:

- изучение основ анализа временных рядов;
- комбинирование временных рядов при помощи `Monoid` и `Semigroup`;
- решение проблемы повторяющихся значений во временных рядах;
- уход от ошибок, вызванных отсутствующими значениями, с использованием `Maybe`.

В этом итоговом проекте вы будете моделировать временные ряды, разработав для этого с помощью Haskell целый ряд инструментов. Временные ряды довольно просты в теории — это ряд значений и соответствующих им дат. На рис. 20.1 демонстрируются данные по продажам из набора данных Бокса и Дженкинса, которые зачастую используются для демонстрации временных рядов (данные в этом проекте представляют собой подмножество из первых 36 месяцев этих данных).

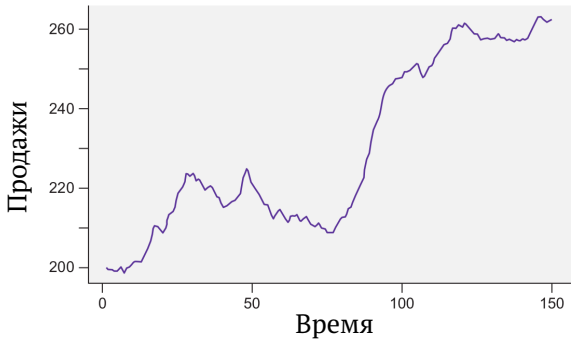


Рис. 20.1: Пример временного ряда продаж (Бокс и Дженкинс, 1976)

Хотя концепция работы с временными рядами проста, на практике они доставляют множество интересных вызовов. Зачастую у вас не хватает данных, иногда нужно совместить несколько неполных наборов данных, а затем эту мешанину проанализировать, для чего довольно часто необходимы другие преобразования, способные привнести в данные смысл. В этом итоговом проекте вы примените подходы, рассматривавшиеся в данном модуле, для создания инструментов обработки данных временных рядов. Вы узнаете, как совмещать несколько временных рядов в один, посчитаете итоговые данные (например, среднее арифметическое) по временным рядам с отсутствующими значениями и закончите преобразованиями данных вроде сглаживания с целью удаления шума.

Весь код этого раздела можно расположить в файле под названием `time_series.hs`. Файл должен импортировать следующие модули (этот текст пишется в начале файла):

Листинг 20.1 Импортируемые модули для `time_series.hs`

```
import Data.List
import qualified Data.Map as Map
import Data.Semigroup
import Data.Maybe
```

20.1. Данные и тип для их представления

Предположим, что вы начали работать в новой компании и получили задание организовать её финансовые данные. У вас есть 36 месяцев (частичных) финансовых данных, в которых необходимо разобраться. Данные содержатся в четырёх файлах, и ни в одном нет полного набора. Так как мы ещё не работали с файлами в Haskell, будем считать, что имеющиеся данные уже считаны и представлены списком кортежей типа `(Int, Double)`.

Листинг 20.2 Данные

```
file1 :: [(Int,Double)]
file1 = [ (1, 200.1), (2,  199.5), (3, 199.4), (4, 198.9)
        , (5,  199.0), (6, 200.2), (9, 200.3), (10, 201.2)
        , (12, 202.9)]

file2 :: [(Int,Double)]
file2 = [ (11, 201.6), (12, 201.5), (13, 201.5), (14, 203.5)
        , (15, 204.9), (16, 207.1), (18, 210.5), (20, 208.8)]
```

```
file3 :: [(Int,Double)]
file3 = [ (10, 201.2), (11, 201.6), (12, 201.5), (13, 201.5)
        , (14, 203.5), (17, 210.5), (24, 215.1), (25, 218.7)]

file4 :: [(Int,Double)]
file4 = [ (26, 219.8), (27, 220.5), (28, 223.8), (29, 222.8)
        , (30, 223.8), (31, 221.7), (32, 222.3), (33, 220.8)
        , (34, 219.4), (35, 220.1), (36, 220.6)]
```

При работе с данными реальных компаний довольно часто имеет место следующая ситуация: данные разбиты на несколько файлов, среди данных в файлах имеются пробелы, также есть перекрывающиеся данные. Вы хотите иметь возможность делать следующее:

- легко сшивать данные разных файлов вместе;
- отслеживать отсутствующие данные;
- выполнять анализ временных рядов, не переживая об ошибках из-за отсутствия значений.

При сшивании временных рядов вы объединяете два временных ряда и получаете один. Это знакомый шаблон, с которым мы уже встречались, обсуждая полугруппы. Вы можете решить задачу сшивания одиночных временных рядов, если сделаете ваш временной ряд экземпляром `Semigroup`. Если вы хотите соединять списки временных рядов, то наверняка захотите реализовать `Monoid` и использовать функцию `mconcat`. Для работы с отсутствующими значениями вы можете использовать преимущества типа `Maybe`. Аккуратно применяя сопоставление с образцом на значениях типа `Maybe`, вы можете реализовать функции на временных рядах и обрабатывать случаи отсутствия значений.

20.1.1. Построение базового типа временного ряда

Для начала вам нужен базовый тип временного ряда. Для простоты будем считать, что все отметки времени принадлежат типу `Int`, который таким образом будет относительным индексом. Не важно, что у нас, 36 месяцев, дней или миллисекунд, — мы всё равно можем обозначать их индексами 1–36. Для значений в рядах вы будете использовать типовый параметр, так как не хотите ограничивать допустимые значения. В этом случае вы будете использовать `Double`, но легко сможете создать временной ряд из `Bool` («Выполнили ли мы план продаж?») или временной ряд из `String` («Кто был лидером по продажам?»). Тип, который вы будете использовать

для представления временного ряда, будет параметризованным типом вида `* -> *`, то есть типом, принимающим один аргумент. Вы также хотите использовать для своих значений тип `Maybe`, поскольку отсутствующие значения — это довольно частая проблема при работе с любыми данными. В анализе данных отсутствующие значения зачастую обозначают как `NA` (от *not available* — не доступно, что похоже на `null` в программном обеспечении). Вот определение типа `TS`.

Листинг 20.3 Определение типа TS

```
data TS a = TS [Int] [Maybe a]
```

Далее мы определим функцию, которая принимает список дат и список значений и создаёт значение типа `TS`. Как и с данными в файлах, будем подразумевать, что при создании значения типа `TS` даты могут не быть идеально непрерывны. При вызове функции `createTS` временная линия насыщается значениями и становится непрерывной. Затем из имеющихся дат и значений создаётся `Map`. После этого мы проходим по полному списку дат и ищем в `Map` соответствующие им значения. Это автоматически создаёт список значений типа `Maybe`, где существующие значения будут представлены `Just a`, тогда как значения `NA` окажутся равны `Nothing`.

Листинг 20.4 Функция createTS и создание значений типа TS

Создаём временной ряд с полной временной линией и списком значений типа `Maybe`, подразумевая, что аргументы могут представлять ограниченное множество возможных значений

Список `completeTimes` содержит все даты от наименьшей, переданной в функцию, до наибольшей

```
createTS :: [Int] -> [a] -> TS a
createTS times values = TS completeTimes extendedValues
  where
    completeTimes = [minimum times .. maximum times]
  timeValueMap = Map.fromList (zip times values)
    extendedValues = map (\v -> Map.lookup v timeValueMap)
                         completeTimes
```

Создаём простой `Map` из имеющихся дат и значений

При поиске по всем датам вы будете получать значения `Just x` для существующих времён и `Nothing` для отсутствующих. Эта строка позаботится о наполнении набора значений, соответствующего полной временной линии (даже если некоторые значения отсутствуют)

Файлы с данными находятся в не совсем подходящем формате для

функции createTS, так что нужно создать вспомогательную функцию, которая поможет раскрыть пары:

Листинг 20.5 Функция fileToTS и перевод данных в значения TS

```
fileToTS :: [(Int,a)] -> TS a
fileToTS tvPairs = createTS times values
  where (times, values) = unzip tvPairs
```

Перед тем как двинуться дальше, было бы неплохо определить удовлетворительный экземпляр Show для объектов типа TS. Сперва вы создадите функцию для вывода пары «время–значение».

Листинг 20.6 Функция showTVPair и пары «время–значение»

```
showTVPair :: Show a => Int -> Maybe a -> String
showTVPair time (Just value) = mconcat [ show time, "|"
                                       , show value, "\n" ]
showTVPair time Nothing = mconcat [show time,"|NA\n"]
```

Теперь вы готовы создать экземпляр Show, используя zipWith и функцию showTVPair.

Листинг 20.7 Экземпляр Show для типа TS

```
instance Show a => Show (TS a) where
    show (TS times values) = mconcat rows
        where rows = zipWith showTVPair times values
```

В GHCi можно посмотреть, как выглядят данные в виде значений типа TS:

```
GHCi> fileToTS file1
1|200.1
2|199.5
3|199.4
4|198.9
5|199.0
6|200.2
7|NA
8|NA
9|200.3
10|201.2
11|NA
12|202.9
```

Теперь можно перевести все файлы в TS.

Листинг 20.8 Приводим все данные к типу TS

```
ts1 :: TS Double
ts1 = fileToTS file1

ts2 :: TS Double
ts2 = fileToTS file2

ts3 :: TS Double
ts3 = fileToTS file3

ts4 :: TS Double
ts4 = fileToTS file4
```

Теперь вы можете конвертировать данные в файлах к базовому типу TS и выводить их на экран, чтобы с ними было проще экспериментировать. Следующая задача — использование Semigroup и Monoid и упрощение с их помощью комбинирования файлов.

20.2. Сшивание временных рядов

Сейчас, когда базовая модель временного ряда готова, вы хотите решать задачу сшивания одиночных временных рядов в один ряд. Подумав над задачей в типах, вы хотите получить следующую типовую аннотацию:

```
TS a -> TS a -> TS a
```

Вам нужна функция, которая принимает два значения TS и возвращает одно. Эта типовая аннотация должна напомнить вам уже знакомую. Если вы посмотрите на операцию <> из Semigroup, то увидите, что это обобщение типа, который вы ищете:

```
(<>) :: Semigroup a => a -> a -> a
```

Это хороший признак, чтобы сделать TS экземпляром Semigroup. Теперь вам необходимо подумать, как именно вы хотите комбинировать два значения TS. Притом что ваш тип TS — это, по сути, два списка, довольно заманчиво решить, что вы можете присоединить два этих списка друг к другу и получить новое значение TS. Но тут есть две проблемы, из-за которых вы не можете просто присоединить два списка, и с ними следует предварительно разобраться. Во-первых, данные из разных файлов могут относиться к разным временным диапазонам. Например, file2 содержит значение

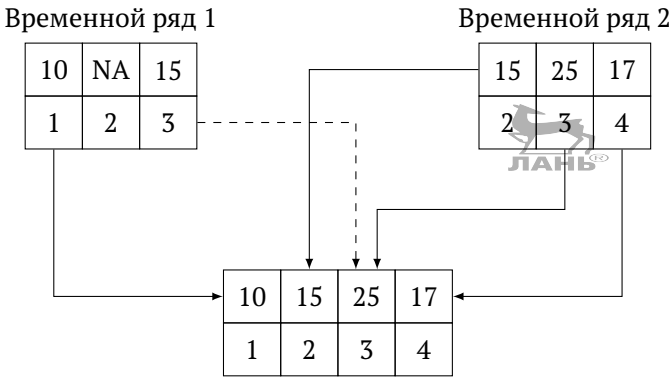


Рис. 20.2: Комбинирование двух временных рядов

для даты 11, а file1 содержит значение для даты 12. Другая проблема в том, что два временных ряда могут иметь конфликтующие значения для одной точки. Первый и второй файлы содержат информацию по дате 12, но они не согласованы. Эту проблему можно решить, дав приоритет второму файлу.

Для решения обеих проблем вы можете использовать `Map`. Вы начнёте с того, что будете брать пары «время–значение» из первого ряда и использовать их для построения `Map`. Затем вы вставите в `Map` пары из второго ряда. Этот подход бесшовно соединит два набора пар и автоматически обработает перезапись повторяющихся значений. На рис. 20.2 показано, как можно объединять два временных ряда.

Важно понимать, что `Map`, который объединит данные из двух временных рядов, будет иметь тип `Map k v`, где `k` — тип ключа, а `v` — тип значения. Но ваши значения в `TS` — это `k` и `Maybe v`. Вам нужна одна маленькая вспомогательная функция, которая позволит вам вставлять значения типа `(k, Maybe v)` в `Map` типа `k v`. Далее вы можете увидеть реализацию такой функции под названием `insertMaybePair`.

Листинг 20.9 Вспомогательная функция для вставки данных в Map

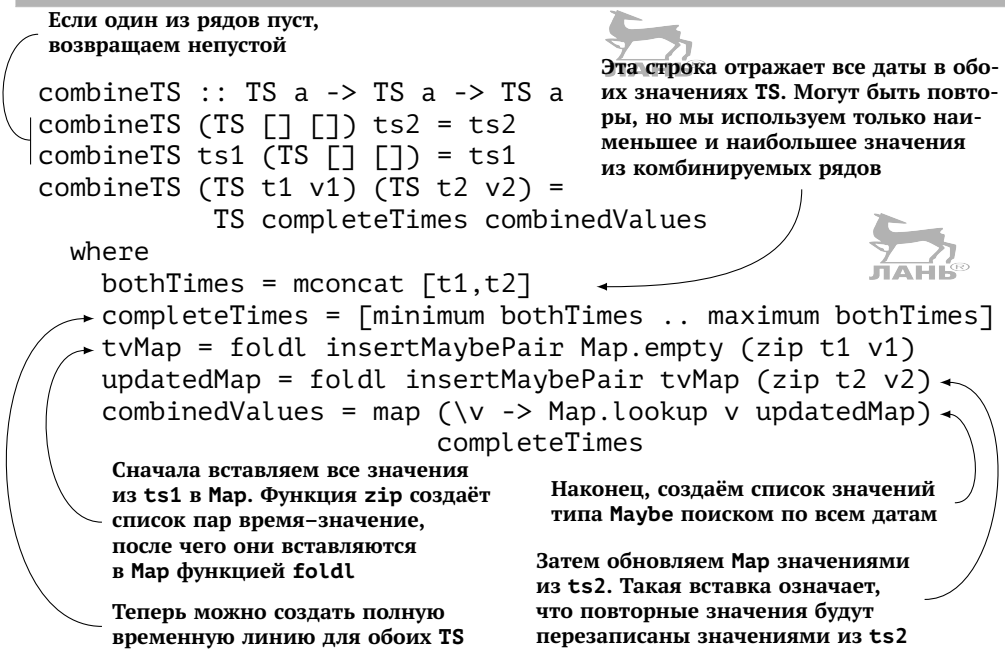
```
insertMaybePair :: Ord k => Map.Map k v -> (k, Maybe v)
                                        -> Map.Map k v
insertMaybePair myMap (_, Nothing) = myMap
insertMaybePair myMap (key, Just value) =
                                  Map.insert key value myMap
```

Так как ваш `Map` состоит из реальных значений, вы можете игнорировать случаи, когда значение `Maybe` отсутствует, возвращая просто `myMap`

Если есть реальное значение, вы извлекаете его из контекста `Just` и помещаете в `Map`

С появлением функции `insertMaybePair` у нас имеются все инструменты для комбинирования двух временных рядов в один. В листинге 20.10 представлена функция `combineTS`, решающая именно эту задачу.

Листинг 20.10 Функция combineTS



Вот как работает функция `combineTS`: во-первых, вам надо обработать случай, когда один (или оба) временной ряд пуст. В этом случае вы возвращаете непустой (или пустой, если пусты оба). Если у вас два непустых значения типа `TS`, вы комбинируете их по всему диапазону дат, которые в них встречаются. Для этого формируете непрерывную временную линию, покрывающую все возможные даты. Затем вы вставляете все существующие значения из первого `TS` в `Map`, используя `insertMaybePair` и сворачивая список пар «значение–время», созданных при помощи `zip`, инициализировав предварительно функцию `foldl` пустым `Map`. После этого вы тем же образом вставляете значения из второго `TS`, только вместо `foldl` с пустым `Map` вы используете `Map`, созданный на предыдущем шаге. Вставка значений из второго ряда после первого гарантирует, что значения второго `TS` будут использованы для всех повторяющихся значений. Наконец, вы ищете все значения в `Map`, собранные из обоих временных рядов, что даёт список значений `Maybe`, как в функции `createTS`.

Функция `combineTS` — это всё, что нужно для реализации `Semigroup`! Вы могли бы поместить всю эту логику прямо в определение `(<>)`. Лич-

но мне показалось, что легче отладить отдельную функцию. Во избежание дублирования кода я бы теперь вставил определение `combineTS` в качестве определения `(<>)`. Но для этого примера давайте просто определим `(<>)` как `combineTS`.

Листинг 20.11 Делаем TS экземпляром Semigroup

```
instance Semigroup (TS a) where
  (<>) = combineTS
```

В GHCi можно убедиться, что теперь вы можете легко объединять два временных ряда!

```
GHCi> ts1 <> ts2
1|200.1
2|199.5
3|199.4
4|198.9
5|199.0
6|200.2
7|NA
8|NA
9|200.3
10|201.2
11|201.6
12|201.5
13|201.5
14|203.5
15|204.9
16|207.1
17|NA
18|210.5
19|NA
20|208.8
```

С `TS` в качестве экземпляра `Semigroup` вы теперь можете объединять временные ряды, автоматически заполняя недостающие значения и перезаписывая повторяющиеся.

20.2.1. Делаем TS экземпляром Monoid

Иметь возможность комбинировать два или более значения типа `TS` при помощи `(<>)` полезно. Но при наличии четырёх уникальных файлов, которые нужно скомбинировать, было бы ещё лучше иметь возможность комбинировать список.

Рассуждая в типах, вы придёте к типовой аннотации `[TS a] -> TS a`, которая описывает необходимое вам поведение. Этот тип должен напоминать конкатенацию списка, которая выполняется функцией `mconcat` типа `Monoid a => [a] -> a`. Единственное, чего нам пока не хватает, — тип `TS` не является экземпляром `Monoid`.

Как обычно, после реализации `Semigroup` вам достаточно добавить нейтральный элемент `mempty`. Без нейтрального элемента вы не сможете автоматически сконкатенировать элементы списка `TS`.

Листинг 20.12 **Делаем TS экземпляром Monoid**

```
instance Monoid (TS a) where
  mempty = TS [] []
  mappend = (<>)
```

Так как вместе с `Monoid` вы бесплатно получаете `mconcat`, то легко можете комбинировать списки `TS`:

```
GHCi> mconcat [ts1, ts2]
1|200.1
2|199.5
3|199.4
4|198.9
5|199.0
6|200.2
7|NA
8|NA
9|200.3
10|201.2
11|201.6
12|201.5
13|201.5
14|203.5
15|204.9
16|207.1
17|NA
18|210.5
19|NA
20|208.8
```

Наконец, вы можете сшить вместе все свои временные ряды в единый временной ряд, который полон настолько, насколько это вообще при имеющихся данных возможно.

Листинг 20.13 Создание единого временного ряда

```
tsAll :: TS Double
tsAll = mconcat [ts1,ts2,ts3,ts4]
```

Чтобы добраться до этого момента, нам понадобилось серьёзно поработать. Зато теперь мы легко можем загружать и объединять временные ряды, полученные из нескольких источников.

20.3. Вычисления на временных рядах

Данные во временных рядах бесполезны, если их нельзя анализировать. Временные ряды в анализе данных в основном используются для выявления основных трендов и временных изменений отслеживаемых значений. Даже простые вопросы о временном ряде могут быть сложными, потому что временные ряды редко показывают аккуратную ровную линию. Давайте начнём анализировать данные временного ряда с подсчёта простой сводной статистики. *Сводная статистика* — это небольшое число значений, которые позволяют описать сложный набор данных. Самая простая сводная статистика для практически любых данных — это среднее значение. В этом разделе вы посмотрите на вычисление среднего значения для имеющихся у нас данных, а также на нахождение наибольшего и наименьшего значений и их суть.

Первым делом нужно вычислить среднее значение временного ряда. Функция `meanTS` будет принимать `TS`, параметризованный типом с ограничением `Real`, и возвращать среднее для значений из `TS` типа `Double`. Класс типов `Real` позволяет использовать функцию `realToFrac`, которая упрощает деление для типов вроде `Integer`. Функция `meanTS` будет возвращать тип `Maybe`, потому что в двух случаях нет значащего результата: в случае пустого временного ряда или когда все его значения равны `Nothing`.

Для начала давайте реализуем функцию, вычисляющую среднее значение числового списка, а затем займёмся `meanTS`.

Листинг 20.14 Вычисление среднего значения числового списка

```
mean :: (Real a) => [a] -> Double
mean xs = total/count
  where total = (realToFrac . sum) xs
        count = (realToFrac . length) xs
```

Теперь обратимся к реализации функции `meanTS`:

```
meanTS :: (Real a) => TS a -> Maybe Double
meanTS (TS _ []) = Nothing
meanTS (TS times values) =
   if all (== Nothing) values
   then Nothing
   else Just avg
 where justVals = filter isJust values
       cleanVals = map fromJust justVals
       avg = mean cleanVals
```

Функция `isJust` происходит из модуля `Data.Maybe`, она проверяет, имеется ли значение в наличии

Функция `fromJust` также находится в `Data.Maybe` и является эквивалентом `(\Just x -> x)`

В GHCi можно проверить средние продажи на отрезке времени:

```
GHCi> meanTS tsAll
Just 210.5966666666667
```

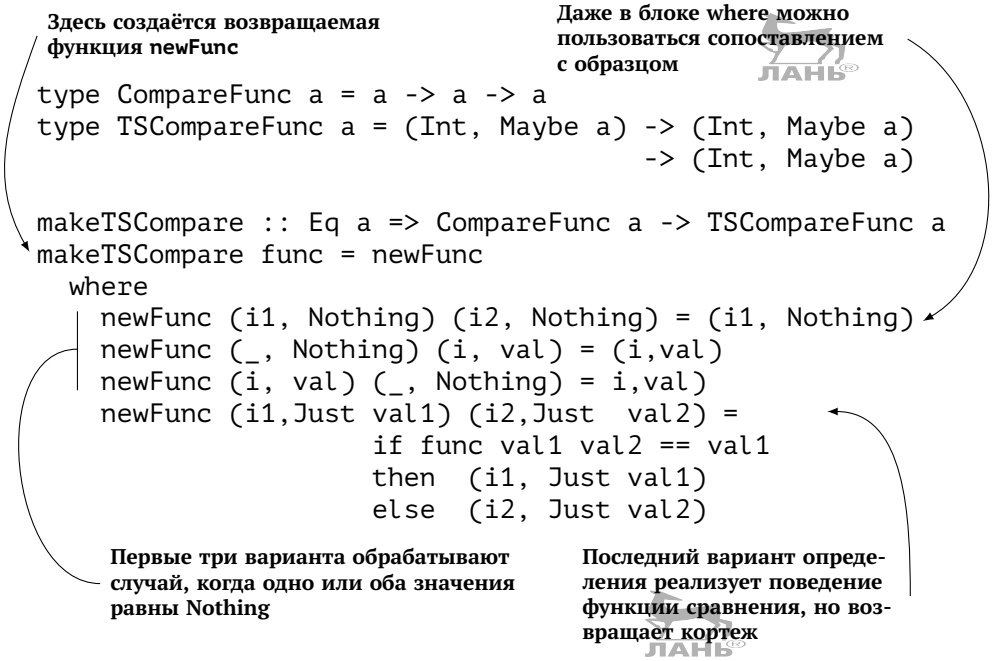
20.3.1. Вычисление значений min и max для ваших временных рядов

Знать минимум и максимум временного ряда тоже полезно. Помимо собственно значений, нам нужны и даты, в которые они достигались. Так как алгоритмы поиска минимума и максимума отличаются только способом сравнения, можно сделать обобщённую функцию `compareTS`, которая будет принимать функцию `(a -> a -> a)` (похоже на `max`, принимающую два значения и определяющую «победителя»). Что интересно, тип функции сравнения совпадает с `(<>)` из `Semigroup`. Однако типа не всегда достаточно, чтобы рассказать всю историю. Обычно мы используем `Semigroup` и `Monoid` для обобщения *комбинирования* двух типов, а не *сравнения*.

Однако вот проблема — функция сравнения имеет тип `(a -> a -> a)`, но сравнивать нужно значения типа `(Int, Maybe a)`. Дело в том, что хочется отслеживать и значение, и дату, но сравнивать нужно только значения. Для простоты напишем функцию `makeTSCompare` (листинг 20.15), которая принимает функцию сравнения типа `(a -> a -> a)` и превращает её в функцию `(Int, Maybe a) -> (Int, Maybe a) -> (Int, Maybe a)`. В результате любые функции вроде `min` и `max` смогут работать с кортежами `(Int, Maybe a)`! Как пример давайте сравним две пары «время–значение» в GHCi:

```
GHCi> makeTSCompare max (3,Just 200) (4,Just 10)
(3,Just 200)
```

Листинг 20.15 Функция makeTSCompare, полезные синонимы типов




```
type CompareFunc a = a -> a -> a
type TSCompareFunc a = (Int, Maybe a) -> (Int, Maybe a)
                                      -> (Int, Maybe a)

makeTSCompare :: Eq a => CompareFunc a -> TSCompareFunc a
makeTSCompare func = newFunc
  where
    newFunc (i1, Nothing) (i2, Nothing) = (i1, Nothing)
    newFunc (_, Nothing) (i, val) = (i,val)
    newFunc (i, val) (_, Nothing) = i,val)
    newFunc (i1,Just val1) (i2,Just  val2) =
                    if func val1 val2 == val1
                    then  (i1, Just val1)
                    else  (i2, Just val2)
```

Функция makeTSCompare позволяет больше не думать об использовании функций сравнения вроде max или min или любых других подобных.

Теперь вы можете построить общую функцию compareTS, которая позволит сравнить все значения в TS.

Листинг 20.16 Сравнение данных во временных рядах

```
compareTS :: Eq a =>  (a -> a -> a) -> TS a
                                    -> Maybe (Int, Maybe a)
compareTS func (TS [] []) = Nothing
compareTS func (TS times values) =
    if all (== Nothing) values
    then Nothing
    else Just best
  where
    pairs = zip times values
    best = foldl (makeTSCompare func) (0, Nothing) pairs
```

Функция compareTS позволяет тривиальным образом определять конкретные функции сравнения данных во временных рядах. Определим, к примеру, max и min.

Листинг 20.17 Определение maxTS и minTS при помощи compareTS

```
minTS :: Ord a => TS a -> Maybe (Int, Maybe a)
minTS = compareTS min

maxTS :: Ord a => TS a -> Maybe (Int, Maybe a)
maxTS = compareTS max
```

Вот примеры в GHCi:

```
GHCi> minTS tsAll
Just (4,Just 198.9)
GHCi> maxTS ts1
Just (12,Just 202.9)
```

Вооружившись несколькими основными инструментами вычисления сводной статистики, можно переходить к более продвинутому анализу данных временных рядов.

20.4. Преобразование временных рядов

Сводная статистика полезна, но её редко хватает для того, чтобы осветить все детали, которые вы хотите знать о временном ряде. В случае месячных данных о продажах вы можете захотеть выяснить, растёт ли ваша компания. Временной ряд — это не просто прямая линия, может быть на удивление сложно ответить на простые вопросы вроде «Как быстро растёт компания?» Наиболее простой подход состоит в слежении за изменениями значений временного ряда во времени. Другая проблема заключается в том, что данные временного ряда могут содержать шум. Для снижения уровня шума применяют *сглаживание*, это может облегчить анализ данных. Обе эти задачи представляют собой преобразования исходных данных с целью извлечения из них интересной информации.

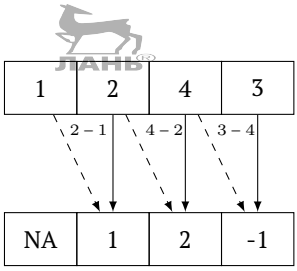


Рис. 20.3: Вычисление разностей

Первое преобразование `TS`, которое мы рассмотрим, — это вычисление разностей. Это преобразование отражает изменения данных по каждому дню. Например, у вас есть значения из рис. 20.3. Заметьте, что новый ряд на одно значение короче исходного. Это происходит, потому что неоткуда брать вычитаемое. Для отражения данного факта мы должны добавлять

к началу результирующего TS значение Nothing. На рис. 20.4 демонстрируется результат применения этого преобразования к данным с рис. 20.1.

Рис. 20.4: Временной ряд продаж после вычисления разностей

Тип преобразования для вычисления разностей можно записать как TS a -> TS a. Это не самое идеальное описание того, каким вы хотите видеть свой финальный результат. Тип TS может принимать любой параметр в качестве типа его значений, но не все значения могут вычитаться друг из друга. На самом деле значение может иметь любой тип, имеющий экземпляр класса типов Num, потому что значения любых таких типов можно друг из друга вычитать: Num a => TS a -> TS a. Пересмотренная типовая аннотация позволяет определить тип, позволяющий выполнить соответствующее преобразование, более точно.

Опять старая проблема: мы работаем со значениями Maybe, но хотим выполнять операции над числами внутри. Начнём с функции diffPair, которая принимает два значения Maybe и выполняет их вычитание.

Листинг 20.18 Типовая аннотация функции diffPair

```
diffPair :: Num a => Maybe a -> Maybe a -> Maybe a
```

Если одно из значений равно Nothing, то нужно возвращать Nothing, иначе вычисляем разность.

Листинг 20.19 Определение функции diffpair

```
diffPair Nothing _ = Nothing
diffPair _ Nothing = Nothing
diffPair (Just x) (Just y) = Just (x - y)
```

Теперь определим функцию требуемого преобразования `diffTS`. Для простоты воспользуемся `zipWith`, которая работает как `zip`, но вместо размещения двух значений в кортеже применяет к ним заданную функцию.

ЛАНЬ

Листинг 20.20 Вычисление разностей во временном ряду

```
diffTS :: Num a => TS a -> TS a
diffTS (TS [] []) = TS [] []
diffTS (TS times values) = TS times (Nothing:diffValues)
  where shiftValues = tail values
        diffValues = zipWith diffPair shiftValues values
```

Функция `diffTS` позволяет посчитать среднее значение изменений в месячных объёмах продаж на протяжении всего промежутка времени:

```
GHCi> meanTS (diffTS tsAll)
Just 0.6076923076923071
```

В среднем продажи каждый месяц росли на 0.6. Здорово, что можно использовать эти инструменты, не беспокоясь об отсутствующих значениях!

ЛАНЬ

20.5. Скользящее среднее

Другое важное преобразование данных во временном ряду — это сглаживание. Зачастую данные имеют необъяснимые пики и провалы, встречается случайный шум, всё это усложняет анализ. Другая проблема заключается в сезонности. Допустим, у вас есть данные по неделям. Ожидаете ли вы, что продажи в воскресенье и вторник будут одинаково хороши? Вы не хотите, чтобы сезонность данных влияла на то, какие выводы вы на их основе делаете.

Лучший способ сглаживания данных — это вычисления скользящего среднего. *Скользящее среднее* похоже на вычисление разностей, но вместо рассмотрения двух чисел вы вычисляете среднее целого диапазона. Скользящее среднее принимает параметр n — количество элементов, по которым будет выполняться сглаживание. Вот пример скользящего среднего по трём элементам для следующих шести чисел:

```
1,2,3,4,3,2

2.000000 3.000000 3.333333 3.000000
```

На рис. 20.5 можно увидеть эффект сглаживания данных из рис. 20.1, к которым применили скользящее среднее по 12 месяцам.

Рис. 20.5: Ряд, сглаженный скользящим средним по 12 месяцам

Обратите внимание, что в процессе сглаживания вы теряете `n/2` значений. При вычислении разностей достаточно было добавить одно `Nothing` в начало данных, в случае скользящего среднего данные обычно «центрируют», добавляя `NA` (или `Nothing` в нашем случае) к обоим концам.

Типовая аннотация функции `movingAverageTS` будет более ограничивающей, нежели у функции `diffTS`. Так как вы знаете, что будете использовать функцию `mean` для вычисления средних значений, то можете взглянуть на её тип и получить подсказку относительно окончательного типа для `movingAverageTS`:

```
mean :: (Real a) => [a] -> Double
```

Так как `mean` будет делать основную работу в вычислении скользящего среднего, ясно, что у нас должно получиться преобразование типа `Real a => TS a` в тип `TS Double`. Вам также понадобится дополнительное значение, уточняющее количество элементов ряда, по которым выполняется сглаживание. Это значит, что тип преобразования должен быть следующим:

```
movingAverageTS :: (Real a) => TS a -> Int -> TS Double
```

Цель ясна, можно начать определять функции, которые к ней приведут.

Для начала упростим реализацию последующих функций, определив функцию скользящего среднего, которая работает со списком значений из временного ряда вида `[Maybe a]`.

Листинг 20.21 Вычисление среднего значения списка Maybe a

```
meanMaybe :: (Real a) => [Maybe a] -> Maybe Double
meanMaybe vals = if any (== Nothing) vals
                 then Nothing
                 else (Just avg)
  where avg = mean (map fromJust vals)
```

Теперь мы готовы реализовать основную логику вычисления скользящего среднего.

Листинг 20.22 Вычисление скользящего среднего списка Maybe a

```
movingAvg :: (Real a) => [Maybe a] -> Int -> [Maybe Double]
movingAvg [] n = []
movingAvg vals n =
     if length nextVals == n
     then meanMaybe nextVals : movingAvg restVals n
     else []
  where nextVals = take n vals
        restVals = tail vals
```

Остаётся только позаботиться о «центрировании» итогового временного ряда значениями `NA`. Воспользуемся для этого целочисленным делением функцией `div` и получим целое значение средней точки.

Листинг 20.23 Вычисление скользящего среднего с центрированием

```
movingAverageTS :: (Real a) => TS a -> Int -> TS Double
movingAverageTS (TS [] []) n= TS [] []
movingAverageTS (TS times values) n = TS times smoothedValues
  where
    ma = movingAvg values n
    nothings = replicate (n `div` 2) Nothing
    smoothedValues = mconcat [nothings,ma,nothings]
```

Функция `replicate` создаёт список с повторяющимися значениями

Функция `movingAverageTS` позволяет сглаживать данные во временных рядах!

Итоги

В этом итоговом проекте вы:

- изучили основные приёмы обработки данных во временных рядах;
- определили тип `TS`, описывающий простейшие временные ряды;
- применили `Maybe` для моделирования значений `NA` в данных;
- использовали `Map` для комбинирования наборов значений;
- применили полугруппы и моноиды к совмещению временных рядов;
- выполняли нетривиальные вычисления в контексте `Maybe`.

Расширение проекта

Никаких границ для крутых вещей, которые можно делать с временными рядами, нет. Простые расширения для этого упражнения включают:

- использование для сглаживания данных медианы вместо среднего;
- вычисление отношений вместо разностей — это позволяет увидеть изменения данных в процентах;
- вычисление стандартного отклонения для данных типа `TS`.

Если вы хотите ещё большего, то наиболее полезное задание — это сложение и вычитание временных рядов. Для каждой из точек в двух `TS` вы должны складывать или вычитать соответствующие значения.

Модуль 4

Ввод и вывод в Haskell

К этому моменту в процессе чтения данной книги вы уже видели множество впечатляющих вещей, которые можно делать в Haskell. Мощь Haskell во многом обеспечивается благодаря чистоте языка и системе типов. Но мы до сих пор избегали разговоров об одной достаточно важной вещи: вводе-выводе.

Вне зависимости от того, что делает ваша программа и на каком языке программирования она написана, ввод-вывод является важной её частью. Это точка соприкосновения вашего кода и реального мира. Так почему же мы до сих пор не касались ввода-вывода в Haskell? Проблема заключается в том, что ввод-вывод неизбежно требует изменений окружающего мира. Рассмотрим, например, считывание пользовательского ввода из командной строки. Каждый раз, когда ваша программа запрашивает ввод пользователя, полученные данные могут быть различными. Но в первом модуле мы довольно долго говорили о том, что функция должна принимать аргумент и возвращать значение, причём одно и то же значение для одного и того же аргумента. Ещё одна проблема со вводом-выводом состоит в том, что, постоянно изменяя окружающий мир, вы вынуждены работать с неким состоянием. Даже если вы просто считываете данные из одного файла и записываете их в другой, ваши программы окажутся бесполезны, если по пути они не изменят состояние внешнего мира. Но, как мы опять же говорили в первом модуле, возможность избежать работы с состоянием является одним из ключевых достоинств языка Haskell.

Так как же Haskell решает эту проблему? Как вы могли догадаться, с помощью типов. В Haskell есть специальный параметризованный тип под названием `IO`. Любое значение, находящееся в контексте `IO`, должно оставаться в этом контексте. Это предотвращает смешивание *чистого* (удовлетворяющего требованиям, описанным нами в модуле 1) кода с кодом, который может таковым не являться.

Для того чтобы лучше в этом разобраться, вы можете сравнить две схожие загадочные функции, написанные на Java и Haskell. Для начала посмотрите на две практически идентичные функции, написанные на Java: `mystery1` и `mystery2`.

Листинг 1 Java-методы mystery1 и mystery2

```
public class Example {
    public static int mystery1(int val1, int val2) {
        int val3 = 3;
        return Math.pow(val1 + val2 + val3, 2);
    }
    public static int mystery2(int val1, int val2) {
        int val3 = 3;
        System.out.print("Введите число:");
        try {
            Scanner in = new Scanner(System.in);
            val3 = in.nextInt();
       } catch (IOException e) {
            e.printStackTrace();
       }
       return Math.pow(val1 + val2 + val3,2);
    }
}
```

Перед вами реализация двух статических методов на Java: `mystery1` и `mystery2`. Они делают одно и то же: принимают два числа, складывают их с «загадочным» третьим числом и возводят результат сложения в квадрат. Важнее всего здесь то, что эти методы в Java имеют одинаковые типы. Но я не думаю, что кто-то станет спорить с тем, что эти методы на самом деле лишь относительно схожи!

Метод `mystery1` является довольно предсказуемым. Всякий раз при передаче в него одних и тех же аргументов вы получите один и тот же результат. В терминах Haskell `mystery1` является *чистой функцией*. Если вы потратите некоторое время на вызовы этой функции с различными аргументами, то сможете разобраться, что она делает.

Метод `mystery2` отличается от своего «собрата». Каждый раз при вызове `mystery2` может возникнуть множество различных ошибок. Помимо этого, каждый вызов `mystery2` с одними и теми же аргументами, вероятно, будет давать вам различные результаты. Возможно, вам вообще не удалось бы угадать, что делает этот метод. Разницу в работе двух методов из этого примера легко заметить, так как `mystery2` вызывает командную строку. Но представьте, что `mystery2` просто считывал бы данные из случайного файла на диске. Вы могли бы вообще никогда не понять, что он делает. Идея загадочных функций, на первый взгляд, может показаться вам надуманной, но на практике постоянно приходится иметь дело с загадочными функциями при работе с унаследованным кодом или сторонними библиотеками: в некоторых случаях вы легко можете угадать, что они делают,

но вы никогда не знаете этого наверняка, не заглянув в исходный код.

Haskell решает данную проблему, заставляя эти две функции иметь разные типовые аннотации. Если в функции используется ввод-вывод, результат функции будет помечен контекстом `IO`, отделить от которого его (результат) будет невозможно. Ниже приведены реализации рассмотренных нами раннее Java-методов в виде функций на Haskell.

Листинг 2 Функции mystery1 и mystery2, переписанные на Haskell

```
mystery1 :: Int -> Int -> Int
mystery1 val1 val2 = (val1 + val2 + val3) ^ 2
  where val3 = 3

mystery2 :: Int -> Int -> IO Int
mystery2 val1 val2 = do
  putStrLn "Введите число:"
  val3Input <- getLine
  let val3 = read val3Input
  return ((val1 + val2 + val3)^2)
```

Каким образом тип `IO` делает ваш код безопаснее? `IO` делает невозможным случайное использование в чистых функциях значений, полученных в результате выполнения операций ввода-вывода. Например, сложение является чистой функцией, так что вы можете сложить результаты двух вызовов `mystery1`:

```
safeValue = (mystery1 2 4) + (mystery1 5 6)
```

Если вы попытаетесь сделать то же с результатами вызовов `mystery2`, получите ошибку компиляции, сообщающую, что складывать значения типа `IO Int` нельзя:

```
unsafeValue = (mystery2 2 4) + (mystery2 2 4)
"No instance for (Num (IO Int)) arising from a use of '+'"
```

Хотя это и делает ваши программы безопаснее, как же вам тогда заставить ваши программы что-либо делать? В этом модуле вы сфокусируетесь на изучении средств языка Haskell, позволяющих вам разделять чистый код и ввод-вывод, в то же время создавая полезные программы, взаимодействующие с реальным миром. После прочтения этого модуля вы научитесь использовать Haskell для решения широкого круга повседневных, настоящих задач, требующих использования ввода-вывода.

21

«Привет, мир!» — введение в ввод-вывод

После прочтения урока 21 вы:

- поймёте, как связаны ввод-вывод в Haskell и типы `IO`;
- сможете использовать do-нотацию для выполнения ввода-вывода;
- начнёте писать программы, взаимодействующие с реальным миром.

В первом уроке вы видели базовый пример программы Hello World. В этом уроке, чтобы лучше разобраться с тем, как работает ввод-вывод в Haskell, мы рассмотрим похожую программу. Ниже приведён пример программы с использованием ввода-вывода, считывающей имя из командной строки и печатающей: `"Привет, <имя>!"`.

Листинг 21.1 Простая программа Hello World

```
helloPerson :: String -> String
helloPerson name = "Привет, " ++ name ++ "!"

main :: IO ()
main = do
   putStrLn "Привет! Как тебя зовут?"
   name <- getLine
   let statement = helloPerson name
   putStrLn statement
```

Даже не зная языка Haskell, вы вполне могли бы прочесть эту программу. К сожалению, теперь, когда вы немного знаете Haskell, этот код, скорее всего, выглядит непонятно! Работа функции `helloPerson` должна быть очевидна, но всё остальное, начиная с `main`, отличается от того, что вам

приходилось видеть до сих пор. У вас наверняка появилось несколько вопросов:

- что это за тип — `IO ()`;
- зачем нужно `do` после `main`;
- возвращает ли `putStrLn` какое-либо значение;
- почему присваивание значений выполняется с помощью `<-` для одних переменных и с помощью `let` для других?

К концу этого урока у вас будут разумные ответы на все эти вопросы, а также, надеюсь, гораздо лучшее понимание основ работы `IO` в Haskell.

Проверка 21.1. В какой строчке считывается пользовательский ввод? Как вы думаете, каков тип этого ввода?

Обратите внимание. Вы можете получить строку пользовательского ввода с помощью функции `getLine`. Но каждый раз при вызове `getLine` эта функция определённо может возвращать разные результаты. Как это может работать, учитывая, что одной из самых важных особенностей языка Haskell является свойство функций возвращать один и тот же результат при одних и тех же входных данных?

21.1. Типы IO: работаем с «грязным» миром

Как часто бывает в Haskell, если вы не совсем понимаете, что происходит, лучше всего будет посмотреть на типы! Первый тип, с которым вам нужно разобраться, — `IO`. В конце предыдущего модуля вы изучали

Ответ 21.1. Считывание ввода происходит в строке, содержащей вызов `getLine`. На данный момент можно считать, что ввод имеет тип `String` (в конце этого урока вы узнаете, что это на самом деле `IO String`).

тип Maybe. Maybe является параметризованным типом (типом, принимающим другой тип в качестве аргумента), представляющим контекст возможного отсутствия значения. IO в Haskell — параметризованный тип, похожий на Maybe. Первая их общая черта — одинаковый вид. Вы можете проверить это, посмотрев на виды типов IO и Maybe:

```
GHCi> :kind Maybe
Maybe :: * -> *
GHCi> :kind IO
IO :: * -> *
```

Ещё одна общая черта Maybe и IO: они оба (в отличие от List или Map) описывают контекст своих параметров, а не контейнер. Контекст для типа IO состоит в том, что значение получено в результате выполнения операции ввода-вывода. Распространёнными примерами таких операций являются чтение ввода пользователя, печать в стандартный поток вывода и чтение из файла.

С помощью типа Maybe вы создаёте контекст, описывающий единственную проблему: иногда значение для программы может отсутствовать. При помощи типа IO вы создаёте контекст для широкого класса ситуаций, которые могут возникнуть при выполнении ввода-вывода. Тип IO подвержен ошибкам, он также по своей сути содержит состояние (запись в файл что-то меняет) и является «грязным» (множественные вызовы getLine вполне могут каждый раз возвращать разные значения в зависимости от ввода пользователя). Хотя эти черты могут казаться проблемой типа IO, они также являются основополагающими для его способа работы. Какой толк от программы, не изменяющей некоторым способом состояние мира? Чтобы сохранить код на Haskell чистым и предсказуемым, вы должны использовать тип IO, обеспечивая контекст для данных, которые могут вести себя не так, как остальной ваш код на Haskell. Действия IO не являются функциями. В программе, приведённой выше, вы видите только одно объявление, использующее тип IO, — типовую аннотацию функции main:

```
main :: IO ()
```

На первый взгляд, () может показаться специальным символом, но на самом деле это просто кортеж из нуля элементов. Мы уже выяснили, что кортежи, представляющие пары элементов, могут быть полезными, но в чём польза кортежа, не содержащего элементов? Ниже приведены похожие типы, построенные с помощью Maybe, чтобы вы могли увидеть, что IO () — это просто IO, параметризованный (), и попробовать понять, почему () может быть полезным:

```
GHCi> :type Just (1, 2)
Just (1, 2) :: (Num t, Num t1) => Maybe (t, t1)
GHCi> :type Just (1)
Just (1) :: Num a => Maybe (a)
GHCi> :type Just ()
Just () :: Maybe ()
```

Тип `Maybe`, параметризованный `()`, бесполезен. Он может иметь всего два значения: `Just ()` и `Nothing`. Но можно сказать, что `Just ()` и есть `Nothing`. Оказывается, что представление отсутствующего значения и есть причина, по которой вам может понадобиться параметризовать тип `IO` пустым кортежем.

Вы можете лучше понять это, подумав о том, что происходит при запуске `main`. Последняя строка кода выглядит следующим образом:

```
putStrLn statement
```

Как вы знаете, этот код печатает `statement`. Каков тип возвращаемого значения функции `putStrLn`? Она отправила сообщение во внешний мир, но не похоже, что обратно должно было прийти что-то полезное. Функция `putStrLn` буквально ничего не возвращает. Так как Haskell нуждается в типе для вашей функции `main`, а `main` ничего не возвращает, вы используете пустой кортеж `()` для параметризации типа `IO`. Так как `()`, по сути, представляет собой «ничто», он является лучшим способом выражения этой идеи в системе типов Haskell.

Хотя вы и удовлетворили систему типов Haskell, кое-что ещё в функции `main` должно быть вам непонятно. В начале книги мы выделили три свойства функций, делающих функциональное программирование предсказуемым и безопасным:

- все функции должны принимать значение;
- все функции должны возвращать значение;
- всякий раз при передаче одного и того же аргумента функция должна возвращать одно и то же значение (*ссылочная прозрачность*).

Функция `main`, очевидно, не возвращает какого-либо осмысленного значения; она просто выполняет *действие*. Оказывается, `main` *не является* функцией, потому что нарушает одно из фундаментальных правил для функций: она не возвращает значение. В связи с этим мы говорим о `main` как о *действии ввода-вывода*. Действия ввода-вывода во многом похожи на функции, за исключением того, что они нарушают как минимум одно из трёх правил, установленных нами для функций в этой книге.

Некоторые из действий ввода-вывода не возвращают значений, некоторые не принимают аргументов, а некоторые — не всегда возвращают одно и то же значение на одних и тех же входных аргументах.

21.1.1. Примеры действий ввода-вывода

Если `main` — не функция, значит, и `putStrLn` функцией не является. Вы можете быстро в этом убедиться, посмотрев на тип `putStrLn`:

```
putStrLn :: String -> IO ()
```

Как вы можете увидеть, функция `putStrLn` возвращает значение типа `IO ()`. Как и `main`, `putStrLn` является действием ввода-вывода, так как нарушает правило о том, что функции должны возвращать значения.

Следующая непонятная функция — `getLine`. Она, очевидно, работает не так, как другие функции, которые вы видели, потому что она не принимает аргументов! Тип `getLine` выглядит следующим образом:

```
getLine :: IO String
```

В отличие от `putStrLn`, которая принимает аргумент и ничего не возвращает, `getLine` не принимает никаких значений, но возвращает значение типа `IO String`. Это означает, что `getLine` нарушает наше правило о том, что все функции должны принимать аргумент. В связи с тем, что `getLine` нарушает правила для функций, она также является действием ввода-вывода.

Теперь давайте рассмотрим более интересный случай. Если вы импортируете `System.Random`, то сможете использовать функцию `randomRIO`, принимающую пару аргументов, представляющих нижнюю и верхнюю границы промежутка, и генерирующую случайное число в этом промежутке. Ниже приведена простая программа под названием roll.hs, использующая `randomRIO` и работающая как бросание игральной кости.

Листинг 21.2 Программа roll.hs для симуляции бросания кости

```
import System.Random

minDie :: Int
minDie = 1

maxDie :: Int
maxDie = 6
```

```
main :: IO ()
main = do
  dieRoll <- randomRIO (minDie, maxDie)
  putStrLn (show dieRoll)
```

Вы можете скомпилировать эту программу с помощью GHC и «бросить» свою кость:

```
$ ghc roll.hs
$ ./roll
2
```

Что насчёт `randomRIO`? Она принимает аргумент (пару из минимального и максимального значений) и возвращает значение (типа IO, параметризованного типом пары) — является ли она функцией? Если вы запустите программу несколько раз, то увидите некоторую проблему:

```
$ ./roll
4
$ ./roll
6
$ ./roll
1
$ ./roll
6
```

Каждый раз при вызове `randomRIO` вы получаете разные результаты, даже если передаёте на вход один и тот же аргумент. Это нарушает правило ссылочной прозрачности. Таким образом, `randomRIO`, как и `getLine` с `putStrLn`, является действием ввода-вывода.

Проверка 21.2. Может ли последней строкой кода в функции `main` быть `getLine`?

Ответ 21.2. Нет, поскольку `main` имеет тип `IO ()`, тогда как `getLine` возвращает `IO String`.

21.1.2. Хранение значений в контексте IO

Интересный момент относительно функции `getLine` состоит в том, что у вас есть полезное возвращаемое значение типа `IO String`. В случае с `Maybe String` контекст заключается в возможном отсутствии значения, а в случае с `IO String` — в том, что значение получено с помощью действия ввода-вывода. В уроке 19 мы обсуждали тот факт, что `Maybe` предотвращает проникновение ошибок, связанных с отсутствием значений, в другой код. Хотя null-значения и вызывают множество ошибок, вы только подумайте, с каким количеством ошибок можно потенциально столкнуться при использовании ввода-вывода!

В связи с тем, что ввод-вывод так опасен и абсолютно непредсказуем, Haskell не позволяет вам использовать эти значения за пределами контекста `IO`. Так, если вы получаете случайное число с помощью функции `randomRIO`, вы не можете использовать его за пределами функции `main` или похожего действия ввода-вывода. Как вы помните, при работе с типом `Maybe` вы могли воспользоваться сопоставлением с образцом, для того чтобы безопасно извлечь значение из контекста. Это связано с тем, что единственное, что может пойти не так со значением типа `Maybe`, — оно будет представлять собой `Nothing`. В случае с вводом-выводом может возникнуть бесконечное множество проблем. Поэтому при работе с данными в контексте `IO` избавиться от контекста нельзя. Изначально это может показаться осложнением. После того как вы познакомитесь с разделением ввода-вывода и остальной логики в Haskell, вы, скорее всего, захотите реализовать то же в других языках программирования (хотя у вас и не будет достаточно мощной для этого системы типов).

21.2. Do-нотация

Невозможность избавиться от контекста `IO` означает, что вам нужен удобный способ выполнения последовательности вычислений в контексте `IO`. В этом состоит предназначение ключевого слова `do`. *Do-нотация* позволяет вам работать с `IO`-типами так, будто они являются совершенно обычными типами.

Интересно, что это разделение также объясняет использование для одних переменных `let`, а для других `<-`. Переменные, значения которым были присвоены с помощью `<-`, позволяют вам работать с типом `IO a` так, будто это значение типа `a`. Выражения `let` используются всякий раз, когда вам нужно создать переменную, *не принадлежащую* типу `IO`.

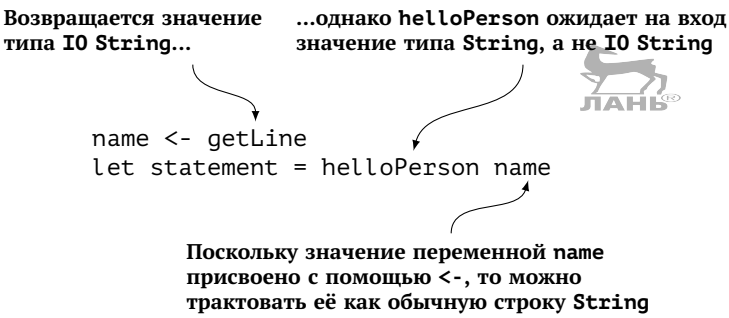


Рис. 21.1: Работа с `IO String` как с обычной `String` в рамках do-нотации

На рис. 21.1 приведены две строки кода функции `main`, в которых используются `<-` и `let`, а также имеются комментарии относительно сути происходящего. Вы уже знаете, что `getLine` возвращает значение типа `IO String`. Тип `name` должен быть `IO String`. Но вы хотите использовать `name` в качестве аргумента для `helloPerson`. Давайте снова посмотрим на тип функции `helloPerson`:

```
helloPerson :: String -> String
```

Функция `helloPerson` работает только с обычными `String`, не с `IO String`. Do-нотация позволяет присвоить значение переменной типа `IO String` с помощью `<-`, чтобы она выглядела как обычная строка (`String`), и затем передавать её в функции, способные работать только со значениями типа `String`.

Ниже снова приведён листинг нашей программы, на этот раз с комментариями, подчёркивающими использование IO-типов и обычных типов.

Листинг 21.3 Разбор do-нотации в программе Hello World

```
helloPerson :: String -> String
helloPerson name = "Привет, " ++ name ++ "!"

main :: IO ()
main = do
   putStrLn "Привет! Как тебя зовут?"
   name <- getLine
   let statement = helloPerson name
   putStrLn statement
```

getLine является действием ввода-вывода и возвращает значение типа IO String

helloPerson является функцией String -> String, но name типа IO String работает благодаря do-нотации

putStrLn является действием ввода-вывода,принимающим обычную String (не IO String)

Мощь этого подхода состоит в том, что вы можете смешивать функции, работающие с безопасными (не имеющими тип `IO`) значениями, и «бесшовно» использовать их с данными в контексте `IO`.

Проверка 21.3. Можете ли вы упростить код, чтобы скомбинировать `helloPerson` и `getLine` следующим образом?

```
let statement = helloPerson getLine
```

21.3. Пример: вычисление стоимости пиццы

Чтобы лучше разобраться с тем, как работает do-нотация, давайте рассмотрим более сложный пример. Вы создадите работающую в командной строке утилиту, которая запрашивает у пользователя размер и цену двух пицц, а затем сообщает, какая из них дешевле в расчёте на квадратный сантиметр. Так как теперь вы можете использовать `IO`, вы наконец сможете создать настоящую исполняемую программу. Назовите файл с исходным кодом pizza.hs. Эта программа будет запрашивать у пользователя стоимость и размер двух пицц, а в ответ сообщать, какая из них лучше в смысле стоимости квадратного сантиметра. Компиляция и запуск программы выглядят следующим образом:

```
$ ghc pizza.hs
$ ./pizza
Введите размер первой пиццы
30
Введите стоимость первой пиццы
150
Введите размер второй пиццы
50
Введите стоимость второй пиццы
220
Пицца размера 50.0 дешевле по цене 0.11204507993669431 за
                                        ↳ квадратный сантиметр
```

Ответ 21.3. Нет, так как `getLine` всё ещё имеет тип `IO String`.

При проектировании программ на Haskell, использующих ввод-вывод, распространённой практикой является написание максимально возможного количества кода без использования `IO`-типов. Это облегчает размышления о задачах и позволяет легко тестировать чистые функции и экспериментировать с ними. Чем больше кода, не использующего контекст `IO`, вы пишете, тем больше кода гарантированно не будет подвержено ошибкам ввода-вывода.

Для начала вам понадобится функция для вычисления стоимости пиццы по её диаметру. В первую очередь нужно вычислить площадь круга по заданному диаметру. Площадь круга равняется $\pi \times r^2$, а радиус равен половине диаметра.

Листинг 21.4 Вычисление площади пиццы по заданному диаметру

```
areaGivenDiameter :: Double -> Double
areaGivenDiameter size = pi * (size / 2)^2
```

Для представления пиццы вы будете использовать пары (размер, стоимость). Можно ввести соответствующий синоним типа:

Листинг 21.5 Синоним типа Pizza

```
type Pizza = (Double, Double)
```

Чтобы вычислить стоимость квадратного сантиметра, вам нужно разделить общую стоимость на площадь.

Листинг 21.6 Вычисление стоимости квадратного сантиметра

```
costPerCm :: Pizza -> Double
costPerCm (size, cost) = cost / areaGivenDiameter size
```

Теперь можно сравнивать две пиццы. Функция `comparePizzas` будет принимать два аргумента типа `Pizza` и возвращать более дешёвую из них.

Листинг 21.7 Сравнение двух пицц

```
comparePizzas :: Pizza -> Pizza -> Pizza
comparePizzas p1 p2 = if costP1 < costP2
                      then p1
                      else p2
  where costP1 = costPerCm p1
        costP2 = costPerCm p2
```

Наконец, вам нужно сообщить пользователю, какая пицца дешевле и какова её цена за квадратный сантиметр.

Листинг 21.8 Описание пиццы

```
describePizza :: Pizza -> String
describePizza (size, cost) = "Пицца размера " ++ show size ++
                             " дешевле по цене " ++
                             show costSqCm ++
                             " за квадратный сантиметр"
  where costSqCm = costPerCm (size, cost)
```

Осталось связать всё это вместе в функции `main`. Но тут возникает ещё одна интересная проблема: `getLine` возвращает `IO String`, но вам нужно, чтобы значения имели тип `Double`. Чтобы разобраться с этим, можно использовать функцию `read`.

Листинг 21.9 Собираем весь код в функции main

```
main :: IO ()
main = do
   putStrLn "Введите размер первой пиццы"
   size1 <- getLine
   putStrLn "Введите стоимость первой пиццы"
   cost1 <- getLine
   putStrLn "Введите размер второй пиццы"
   size2 <- getLine
   putStrLn "Введите стоимость второй пиццы"
   cost2 <- getLine
   let pizza1 = (read size1, read cost1)
   let pizza2 = (read size2, read cost2)
   let betterPizza = comparePizzas pizza1 pizza2
   putStrLn (describePizza betterPizza)
```

Ключевой момент здесь заключается в том, что вам нужно волноваться только о тех частях программы, которые должны быть выполнены в контексте `IO`, то есть тех, что связаны с получением и обработкой ввода.

21.3.1. Первый взгляд на монады: do-нотация для Maybe

Тип `IO` способен работать с do-нотацией в связи с тем, что он принадлежит мощному классу типов под названием `Monad`. Мы обсудим его подробнее в пятом модуле. Do-нотация не связана напрямую с типом `IO` и может быть использована для выполнения вычислений в контексте с любым членом класса `Monad`. Контекст значений в `Maybe` состоит в их возможном отсутствии. Контекст `IO` состоит в том, что вы взаимодействуете с реальным миром и ваши данные могут вести себя не так, как остальная программа на Haskell.

Maybe также является членом класса типов Monad и, таким образом, может работать с do-нотацией. Допустим, вместо того чтобы получать данные о пицце из пользовательского ввода, вам нужно получать эти значения из двух Map: первого, содержащего размеры, и второго, содержащего цены. Ниже приведена практически идентичная написанной вами программа, только с использованием Maybe. Вместо того чтобы получать информацию от пользователя, вы теперь выполняете поиск стоимости пиццы по её идентификатору в Map под названием costData.

Листинг 21.10 Информация о стоимости пиццы в Map

```
costData :: Map.Map Int Double
costData = Map.fromList [(1, 150), (2, 220)]
```

Размеры содержатся в другом Map:

Листинг 21.11 Информация о размерах пиццы в Map

```
sizeData :: Map.Map Int Double
sizeData = Map.fromList [(1, 30), (2, 50)]
```

И наконец, функция maybeMain, выглядящая почти как ваша!

Листинг 21.12 Версия main с использованием Maybe вместо IO

```
maybeMain :: Maybe String
maybeMain = do
   size1 <- Map.lookup 1 sizeData
   cost1 <- Map.lookup 1 costData
   size2 <- Map.lookup 2 sizeData
   cost2 <- Map.lookup 2 costData
   let pizza1 = (size1, cost1)
   let pizza2 = (size2, cost2)
   let betterPizza = comparePizzas pizza1 pizza2
   return (describePizza betterPizza)
```

Единственная новая деталь здесь: добавлена функция return, принимающая значение некоторого типа и помещающая его в контекст, используемый в do-нотации. В этом случае String превращается в Maybe String. Вам не нужно было этого делать в main, потому что putStrLn возвращает значение типа IO (). Можно проверить работу этой функции в GHCi:

```
GHCi> maybeMain
Just "Пицца размера 50.0 дешевле по цене 0.11204507993669431
                                    ↳ за квадратный сантиметр"
```

Если вы уже слышали о монадах и интересовались, чем вызван весь шум вокруг них, то этот пример может вас разочаровать. Класс типов `Monad` позволяет вам писать код, который может работать в самых разнообразных контекстах. Благодаря do-нотации вы можете написать новую программу, сохранив все основные функции оригинальной. В большинстве языков программирования вам бы, скорее всего, пришлось переписывать каждую функцию, чтобы преобразовать программу от работающей с `IO` к работающей с потенциально отсутствующими значениями в словаре. В пятом модуле вы погрузитесь в эту тему гораздо глубже. А пока можно думать о do-нотации как об удобном способе выполнять действия ввода-вывода.

Итоги

Целью этого урока был рассказ о работе с вводом-выводом. Проблема состоит в том, что ввод-вывод требует некоторых особенностей поведения функций, которые мы ранее в этой книге запретили. Ввод-вывод часто меняет состояние внешнего мира и так же часто приводит к тому, что функции (точнее, действия ввода-вывода) возвращают различные результаты каждый раз, когда их вызывают. Haskell справляется с этой проблемой, требуя размещения всей логики ввода-вывода в контексте `IO`. В отличие от типа `Maybe`, вы не можете извлечь значения из контекста `IO` после того, как они туда попали. Чтобы упростить работу с `IO`, в Haskell введена специальная do-нотация, позволяющая писать код так, будто вы работаете не в контексте `IO`. Давайте удостоверимся, что вы всё поняли.

Задача 21.1. Перепишите код листинга 21.1 в код, использующий do-нотацию в контексте `Maybe`. Предполагайте, что весь пользовательский ввод заменён на `Map`, содержащий нужные значения. Пропустите первый вызов `putStrLn` и просто верните приветствие в конце.

Задача 21.2. Напишите программу, запрашивающую у пользователя число `n` и возвращающую `n`-е число Фибоначчи (пример вычисления чисел Фибоначчи можно найти в уроке 8).

22

Командная строка и ленивый ввод-вывод

После прочтения урока 22 вы:

- узнаете, как получать доступ к аргументам командной строки;
- сможете использовать традиционный подход к взаимодействию через ввод-вывод;
- научитесь использовать ленивые вычисления для упрощения работы с вводом-выводом.

Часто при первом знакомстве с вводом-выводом в Haskell люди думают, что ввод-вывод представляет для Haskell некоторое затруднение в связи с тем, что Haskell предназначен для написания чистых программ, а операции ввода-вывода чистыми не являются. Но существует другой взгляд на ввод-вывод, делающий его подходящим именно для Haskell и несколько неуклюжим в других языках программирования. При работе с вводом-выводом в любом языке мы часто говорим о *потоках ввода-вывода*, но что такое поток? Хороший способ представления потока — лениво вычисляемый список символов. STDIN (стандартный поток ввода) последовательно передаёт пользовательский ввод программе до тех пор, пока не достигнет его конца. Но заранее неизвестно, где находится этот конец (теоретически, его может вообще не быть). Это совпадает с нашими представлениями о списках в Haskell с использованием ленивых вычислений.

Такое представление ввода-вывода используется практически во всех языках программирования при чтении из больших файлов. Загрузка большого файла в память перед началом работы с ним зачастую является непрактичной или даже невозможной. Но представьте, что данный большой файл представляет собой текст, присвоенный переменной, которая

является ленивым (вычисляемым только при необходимости) списком. Как вы знаете из предыдущих уроков, ленивые вычисления позволяют работать с бесконечно длинными списками. Размер входных данных не имеет значения, если вы можете работать с ними как с большим списком.

В этом уроке вы рассмотрите простую задачу и несколько способов её решения. Вам нужно будет создать программу, считывающую произвольной длины список чисел, вводимых пользователем, а затем сложить все эти числа и вернуть пользователю результат. В процессе решения вы научитесь писать код, реализующий традиционный ввод-вывод, и сможете использовать ленивые вычисления для упрощения этого решения.

Обратите внимание. Вам нужно написать программу, позволяющую пользователю проверять слова на палиндромность. Можно легко реализовать проверку одного слова, но как сделать так, чтобы пользователь мог вводить произвольный список слов, каждое из которых должно проверяться программой?

22.1. Энергичное взаимодействие с командной строкой

Для начала давайте спроектируем утилиту командной строки, считывающую список введённых пользователем чисел и суммирующую их. Создайте для программы файл под названием sum.hs. В предыдущем уроке вы разобрались со считыванием ввода пользователя и выполнением над ним вычислений. Сложность задачи на этот раз состоит в том, что заранее неизвестно, сколько чисел введёт пользователь.

Возможный способ решения этой задачи — предоставить пользователю возможность ввести значение в качестве аргумента для программы (аргумента командной строки), например:

```
$ ./sum 4
3
5
9
25
42
```

Для получения аргументов командной строки вы можете использовать функцию `getArgs`, которую можно найти в `System.Environment`. Её типовая аннотация выглядит следующим образом:

```
getArgs :: IO [String]
```

ЛАНЬ

То есть вы получаете список строк в контексте IO. Ниже приведён пример использования getArgs в функции main.

Листинг 22.1 Доступ к аргументам командной строки (getArgs)

```
import System.Environment

main :: IO ()
main = do
  args <- getArgs
  ...
```

Чтобы понять, как работает getArgs, можно вывести полученные нами аргументы args. Как вам известно, args — список, а потому вы можете использовать map для прохода по всем значениям. Но есть проблема: вы работаете в контексте do-нотации с типом IO. Вам хотелось бы написать следующий код:

Листинг 22.2 Предполагаемый код для печати args (не работает!)

```
main putStrLn args
```

Но args — не просто список, а putStrLn — не просто функция. Выполнить map над списком в контексте IO можно с помощью специальной версии функции map, работающей со списками в этом контексте (вообще говоря, со списком в любом контексте, являющемся представителем класса типов Monad). Для этого существует специальная функция под названием mapM (M означает Monad).

Листинг 22.3 Попытка исправления: функция mapM (не работает!)

```
main :: IO ()
main = do
  args <- getArgs
  mapM putStrLn args
```

При попытке скомпилировать программу вы снова получите ошибку:

```
Couldn't match type '[()]' with '()'
```

GHC выдаёт ошибку из-за того, что тип main должен быть IO (), но map, как вы помните, возвращает список. Вам нужно просто пройти по элементам списка args и выполнить действия ввода-вывода; результаты этих действий не имеют значения, и вам не нужен список возвращаемых значений.

Справиться с этой проблемой поможет функция под названием `mapM_` (обратите внимание на нижнее подчёркивание). Она работает как `mapM`, но отбрасывает результаты. Обычно, если имя функции в Haskell заканчивается нижним подчёркиванием, это означает, что вы отбрасываете результаты. Внеся небольшие изменения в код, вы получите:

```
main :: IO ()
main = do
  args <- getArgs
  mapM_ putStrLn args
```

Вы можете проверить работу программы, несколько раз её запустив:

```
$ ./sum
$ ./sum 2
2
$ ./sum 2 3 4 5
2
3
4
5
```

Проверка 22.1. Напишите функцию `main`, использующую `mapM`, чтобы трижды вызвать `getLine`, а затем используйте `mapM_`, чтобы напечатать введённые значения (подсказка: вам нужно отбросить аргумент при использовании `mapM` с `getLine`, для достижения этого используйте `(\_ -> ...)`).

Теперь вы можете реализовать логику получения нужного аргумента. Вы также должны обработать случай, когда пользователь не вводит аргумент. Можно рассматривать его как необходимость считать 0 строк. Также обратите внимание, что вы впервые будете использовать функцию `print`.

Ответ 22.1

```
exampleMain :: IO ()
exampleMain = do
   vals <- mapM (\_ -> getLine) [1..3]
   mapM_ putStrLn vals
```

Реализация функции `print` выглядит как `(putStrLn. show)` и облегчает вывод значения любого типа.

Листинг 22.4 Количество строк как аргумент командной строки

```
main :: IO ()
main = do
  args <- getArgs
  let linesToRead = if length args > 0
                    then read (head args)
                    else 0 :: Int
  print linesToRead
```

Теперь, когда вы знаете, сколько строк вам нужно считать, нужно столько раз вызвать `getLine`. В Haskell есть подходящая для этого функция под названием `replicateM`. Она принимает число, обозначающее количество раз, которое вы хотите выполнить действие ввода-вывода, и повторяет соответствующее действие указанное количество раз. Для её использования вам нужно импортировать модуль `Control.Monad`.

Листинг 22.5 Считывание строк в заданном количестве

```
import Control.Monad

main :: IO ()
main = do
  args <- getArgs
  let linesToRead = if length args > 0
                    then read (head args)
                    else 0 :: Int
  numbers <- replicateM linesToRead getLine
  print "здесь будет сумма"
```

У вас почти получилось! Как вы помните, функция `getLine` возвращает `String` в контексте `IO`. Перед тем как вы сможете вычислить сумму всех аргументов, вам нужно преобразовать их к типу `Int`, а затем вернуть сумму списка целых чисел.

Листинг 22.6 Полная реализация программы sum.hs

```
import System.Environment
import Control.Monad

main :: IO ()
main = do
  args <- getArgs
```

```
let linesToRead = if length args > 0
                  then read (head args)
                  else 0 :: Int
numbers <- replicateM linesToRead getLine
let ints = map read numbers :: [Int]
print (sum ints)
```

Работы было много, но зато теперь у вас есть утилита, позволяющая пользователям вводить произвольное количество целых чисел и складывать их:

```
$ ./sum 2
4
59
63
$ ./sum 4
1
2
3
410
416
```

На примере этой простой программы вы изучили целый ряд средств обработки пользовательского ввода. Таблица 22.1 содержит некоторые полезные функции для повторения действий в контексте `IO`.

Функция	**Поведение**
`mapM`	Принимает на вход действие ввода-вывода и обычный список, выполняет действие на каждом элементе списка и возвращает список в контексте `IO`
`mapM_`	Работает как `mapM`, но отбрасывает результат (обратите внимание на `_`)
`replicateM`	Принимает на вход действие ввода-вывода и целое число `n`, повторяет действие `n` раз и возвращает результаты в виде списка в контексте `IO`
`replicateM_`	Работает как `replicateM`, но отбрасывает результат

Таблица 22.1: Функции для повторения действий в контексте `IO`

В следующем разделе вы узнаете, как можно было бы сделать это проще, используя ленивые вычисления.

Проверка 22.2. Напишите собственную версию `replicateM` под названием `myReplicateM`, использующую `mapM` (можете не задумываться о типовой аннотации).

22.2. Взаимодействие с ленивым вводом-выводом

Последняя написанная вами программа работает, но имеет несколько недостатков. Во-первых, вы требуете от пользователя ввода фиксированного числа строк. Таким образом, пользователь заранее должен знать, сколько чисел он собирается ввести. Что, если пользователь в онлайн-режиме считает посетителей музея или работает с выводом другой вашей программы?

Вспомните: основная цель использования типа `IO` состоит в разделении функций, которые неизбежно должны работать с вводом-выводом, и всех остальных. В идеале вам нужно, чтобы максимально возможный объём программной логики находился за пределами `main`. В этой программе вся логика завёрнута в `IO`, что свидетельствует о том, что вы не очень хорошо структурируете свою программу. Отчасти это связано с тем, что большое количество операций ввод-вывода перемешано с основной логикой работы вашей программы.

Основная причина этой проблемы состоит в том, что вы работаете с данными, полученными в результате выполнения действий ввода-вывода, как с последовательностью значений, которую нужно немедленно обработать. Альтернативным подходом является представление потока данных от пользователя в виде списка. Вместо того чтобы думать о каждой части данных как об отдельном взаимодействии с пользователем, вы можете работать со всеми данными от пользователя как со списком символов. При рассмотрении входных данных как списка символов вам будет гораздо легче спроектировать программу, не думая о неудобствах, связанных с вводом-выводом. Чтобы это сделать, вам понадобится специальное дей-

Ответ 22.2

```
myReplicateM :: Monad m => Int -> m a -> m [a]
myReplicateM n func = mapM (\_ -> func) [1 .. n]
```

ствие getContents. Действие getContents позволяет работать с потоком ввода STDIN как со списком символов.

Вы можете воспользоваться getContents вместе с mapM_ и увидеть, каким странным может оказаться результат. Для этого раздела вам понадобится создать файл под названием sum_lazy.hs.

Листинг 22.7 Изучение особенностей ленивого ввода-вывода

```
main :: IO ()
main = do
  userInput <- getContents
  mapM_ print userInput
```

Действие getContents считывает ввод до тех пор, пока не дойдёт до символа конца файла. Для обычного текстового файла это собственно конец файла, а в случае пользовательского ввода его нужно ввести вручную (в большинстве терминалов — посредством нажатия комбинации клавиш Ctrl-D). Перед запуском программы стоит подумать о том, что произойдёт, учитывая ленивость вычислений. В энергичном (не являющемся ленивым) языке программирования вы бы предположили, что вам нужно будет дождаться ручного ввода Ctrl-D, перед тем как ввод будет напечатан на экране. Давайте посмотрим, что произойдёт в Haskell:

```
$ ./sum_lazy
hi
'h'
'i'
'\n'
what?
'w'
'h'
'a'
't'
'?'
'\n'
```

Как видите, благодаря тому что Haskell умеет работать с ленивыми списками, он может обработать ваш текст сразу же, как только вы его ввели! Это значит, что вы можете обрабатывать непрерывное взаимодействие с пользователем интересными способами.

Проверка 22.3. Используя ленивый ввод-вывод, напишите программу, которая принимает ввод пользователя, инвертирует и печатает его на экране.

22.2.1. Думаем о задаче как о ленивом списке

Действие `getContents` позволяет переписать программу, не обращая на этот раз внимания на `IO`, пока не придёт время. Для начала берём список символов, состоящий из чисел и символов перехода на новую строку (`\n`), например:

Листинг 22.8 Тестовые данные как строка входных символов

```
sampleData = ['6', '2', '\n', '2', '1', '\n']
```

Если удастся написать функцию, преобразующую этот список в список целых чисел, задача будет решена! Для строк есть полезная функция, которая облегчит вам работу: функция `lines` позволяет разбить строку по символу перехода на новую строку. Ниже приведён пример её использования в GHCi на тестовых данных:

```
GHCi> lines sampleData
["62", "21"]
```

Модуль `Data.List.Split` содержит более продвинутую, чем `lines`, функцию `splitOn`, разбивающую строку на части по другой заданной строке. Модуль `Data.List.Split` не является частью базового Haskell, но входит в Haskell Platform. Если вы не пользуетесь Haskell Platform, вам придётся его установить. Функция `splitOn` очень полезна при обработке текста. Реализация `lines` с помощью `splitOn` могла бы выглядеть следующим образом:

Ответ 22.3

```
reverser :: IO ()
reverser = do
   input <- getContents
   let reversed = reverse input
   putStrLn reversed
```

Листинг 22.9 Определение myLines с помощью splitOn

```
myLines = splitOn "\n"
```

Теперь остаётся с помощью функции `lines` и многократного применения функции `read` получить список значений типа `Int`. Для этого можно написать функцию `toInts`:

Листинг 22.10 Функция toInts для преобразования Char в Int

```
toInts :: String -> [Int]
toInts = map read . lines
```

Заставить эту функцию работать с `IO` достаточно просто. Для этого нужно применить её к `userInput`, полученному с помощью `getContents`.

Листинг 22.11 Ленивое решение задачи суммирования чисел

```
main :: IO ()
main = do
  userInput <- getContents
  let numbers = toInts userInput
  print (sum numbers)
```

Как вы можете увидеть, финальная версия `main` гораздо короче и чище, чем в предыдущем решении. Теперь вы можете скомпилировать и протестировать свою программу:

```
$ ./sum_lazy
4
234
23
1
3
<ctrl-d>
265
```

Это решение гораздо лучше предыдущего, так как код теперь чище и пользователям больше не нужно думать о том, сколько чисел будет во вводимом ими списке. В этом уроке вы узнали, как структурировать свою программу так, чтобы она была похожа на программы, написанные на большинстве других языков программирования. Вы запрашиваете ввод пользователя, обрабатываете эти данные, а затем (при необходимости) снова запрашиваете ввод. В этой модели вы выполняете *энергичный* ввод-вывод, то есть обрабатываете все данные сразу при получении. Во многих случаях, если вы работаете с пользовательским вводом как

с обычными ленивыми списками символов, вы можете вынести большую часть программной логики за пределы кода, использующего ввод-вывод. В итоге у вас остаётся всего одно место для работы со списком в контексте `IO` — место, где вы впервые его получаете. Благодаря этому остальной ваш код может быть написан в расчёте на обычные списки Haskell.

Проверка 22.4. Напишите программу, возвращающую сумму квадратов введённых пользователем чисел.

Итоги

Нашей целью в этом уроке было представить вам способы написания на Haskell простых приложений, работающих с командной строкой. Самый привычный способ — относиться к вводу-выводу как в любом другом языке программирования. Вы можете использовать do-нотацию для создания процедурных списков действий ввода-вывода и таким образом строить взаимодействие с вводом-выводом. Более интересным подходом, доступным в очень немногих языках, помимо Haskell, является использование преимуществ ленивых вычислений. Благодаря ленивым вычислениям вы можете думать о всём потоке ввода как о лениво вычисляемом списке символов типа `[Char]`. Вы можете существенно упростить свой код посредством написания чистых функций, как будто надо работать с обычным типом `[Char]`. Давайте проверим, что вы всё поняли.

Задача 22.1. Напишите программу `simple_calc.hs`, считывающую простые выражения, включающие сложение и умножение двух чисел. Программа должна вычислять значения выражений, вводимых пользователем (в каждую строку нужно вводить одно выражение).

Ответ 22.4

```
mainSumSquares :: IO ()
mainSumSquares = do
   userInput <- getContents
   let numbers = toInts userInput
   let squares = map (^2) numbers
   print (sum squares)
```

Задача 22.2. Напишите программу, предлагающую пользователю выбрать число от 1 до 15 и затем выводящую известную цитату (выбор цитат предоставляется вам). После вывода цитаты программа должна спрашивать у пользователя о желании увидеть ещё одну. Если пользователь вводит `"n"`, программа останавливается; иначе он получает ещё одну цитату. Программа повторяет вышеописанные действия до тех пор, пока пользователь не введёт `"n"`. Попробуйте использовать ленивые вычисления и работать с пользовательским вводом как со списком, вместо того чтобы рекурсивно вызывать `main` в конце.

ЛАНЬ

23

Работа с типом Text и Юникодом

После прочтения урока 23 вы:

- начнёте использовать тип `Text` для эффективной обработки текста;
- научитесь менять поведение Haskell, пользуясь расширениями;
- узнаете множество функций для обработки текста;
- разберётесь с корректной обработкой текста в Юникоде.

К этому моменту вы уже довольно много пользовались типом `String`. В предыдущем уроке вы видели, что даже можете рассматривать потоки ввода-вывода как ленивые списки типа `Char` (то есть `String`). Тип `String` был нам часто полезен как пример для изучения многих тем. К сожалению, у `String` есть большая проблема — обработка текста, представленного с его помощью, может быть ужасно неэффективной.

С философской точки зрения ничто не может быть более совершенно, чем представление одного из наиболее важных типов в программировании одной из основополагающих структур данных Haskell — списком. Проблема в том, что список — это не лучшая структура данных, чтобы хранить в ней требующие сложной обработки текстовые данные. Детали относительно производительности Haskell выходят за пределы материала этой книги, отметим лишь, что реализация типа `String` как связного списка символов излишне затратна по времени и памяти.

В этом уроке вы взглянете на новый тип `Text`. Вы узнаете, как повысить эффективность обработки текстов, заменив `String` на `Text`. Затем вы узнаете об общих для `String` и `Text` функциях. И наконец, узнаете о возможностях `Text` по обработке Юникода, реализовав функцию, которая подсвечивает искомый текст, причём на санскрите!

Обратите внимание. `String` в Haskell — это частный случай списка. Однако в большинстве языков программирования строковые типы более эффективно хранятся в виде массивов. Есть ли в Haskell способ использовать уже известные инструменты для работы с типом `String`, получив при этом эффективность хранения в виде массива?

23.1. Тип Text

Для работы с текстовыми данными в случае практического и коммерческого программирования на Haskell наиболее предпочтителен тип `Text`. Он становится доступным при подключении модуля `Data.Text`. На практике этот модуль почти всегда импортируется квалифицированно с одной буквой, обычно `T`, в качестве псевдонима:

```
import qualified Data.Text as T
```

Значения типа `Text`, в отличие от `String`, под капотом реализованы в виде массива. Это делает многие строковые операции быстрее и эффективнее по памяти. Другое важное отличие между `Text` и `String` состоит в том, что `Text` *не использует* ленивые вычисления. Ленивые вычисления продемонстрировали свою полезность в предыдущих уроках, однако в задачах реального мира они могут быть источником проблем с производительностью. Тем не менее, если вам очень нужен ленивый текст, вы можете использовать `Data.Text.Lazy`, интерфейс которого совпадает с `Data.Text`.

23.1.1. Когда использовать Text, а когда String

Среди коммерческих разработчиков на Haskell бытует однозначное мнение о предпочтительности `Data.Text` над `String`. Некоторые члены Haskell-сообщества даже утверждают, что стандартный `Prelude` бесполезен для промышленной разработки в силу слишком глубокой зависимости от `String`. Однако при изучении Haskell тип `String` всё же полезен, как минимум по двум причинам. Во-первых, как уже было замечено, многие базовые средства обработки строк доступны непосредственно из `Prelude`. Во-вторых, списки для Haskell — это то же самое, что и массивы для C. Многие концепции Haskell удобно демонстрировать на списках, а строки — это удобные списки. Так что для обучения вы можете свободно пользоваться `String`, но для чего-либо вне упражнений старайтесь применять

Data.Text как можно больше. Далее в этой книге мы продолжим использовать String, но и Data.Text будет появляться довольно часто.

23.2. Использование Data.Text

Первое, что вам нужно узнать, — как пользоваться типом Text. Модуль Data.Text предоставляет две функции pack и unpack, которые могут использоваться для преобразований String -> Text и Text -> String. Выяснить, какая функция что делает, можно по их типовым аннотациям:

```
T.pack :: String -> T.Text
T.unpack :: T.Text -> String
```

Вот некоторые примеры преобразований String в Text и обратно.

Листинг 23.1 Преобразование между типами String и Text

```
firstWord :: String
firstWord = "пессимизм"

secondWord :: T.Text
secondWord = T.pack firstWord

thirdWord :: String
thirdWord = T.unpack secondWord
```

Важно отметить, что эти преобразования вычислительно дороги, так как вам нужно всякий раз проходить по всей строке. Старайтесь избегать конвертирования из Text в String и обратно.

Проверка 23.1. Определите значение fourthWord, преобразовав thirdWord в T.Text.

Ответ 23.1

```
fourthWord :: T.Text
fourthWord = T.pack thirdWord
```

23.2.1. Расширение OverloadedStrings и другие расширения Haskell

В работе с `T.Text` раздражает ошибка при написании следующего кода.

Листинг 23.2 Строковые литералы в определении T.Text

```
myWord :: T.Text
myWord = "собака"
```

Сообщение об ошибке выглядит следующим образом:

```
Couldn't match expected type 'T.Text' with actual type '[Char]'
```

Её причина в том, что литерал `"собака"` имеет тип `String`. Это особенно раздражает, поскольку при работе с числовыми типами такой проблемы нет. Рассмотрим, например, код с определениями числовых значений.

Листинг 23.3 Числовые литералы разных типов

```
myNum1 :: Int
myNum1 = 3

myNum2 :: Integer
myNum2 = 3

myNum3 :: Double
myNum3 = 3
```

Этот код отлично компилируется, хотя здесь один и тот же литерал `3` используется как значение для трёх разных типов.

Ясно, что это не та проблема, от которой спасёт какой-нибудь особо умный программистский приём, и мощь Haskell тут тоже ни при чём. Чтобы решить эту проблему, потребуется фундаментальным образом поменять способ, которым GHC читает ваш файл. На удивление, простое решение всё-таки существует! GHC позволяет использовать *языковые расширения* и с их помощью влиять на то, как работает сам Haskell. Конкретное расширение, которым мы здесь воспользуемся, называется `OverloadedStrings`.

Есть два способа подключить языковое расширение. Первый — указать его во время компиляции в GHC. Для этого используют флаг `-X`, за которым следует имя расширения. Для программы с названием `text.hs` этот способ выглядит следующим образом:

```
$ ghc text.hs -XOverloadedStrings
```

То же самое можно делать и при запуске интерпретатора GHCi.

Проблема в том, что тот, кто использует ваш код (и этим кем-то можете оказаться вы), может позабыть указать этот флаг. Поэтому предпочтительнее другой способ, а именно использование директивы LANGUAGE. Эта директива выглядит следующим образом:

```
{-# LANGUAGE <Extension Name> #-}
```

Полностью файл text.hs, в котором для значений типа Text используются строковые литералы, выглядит следующим образом.

Листинг 23.4 Расширение OverloadedStrings и литералы типа Text

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text as T

aWord :: T.Text
aWord = "Сыр"

main :: IO ()
main = do
  print aWord
```

Директива LANGUAGE позволяет компилировать эту программу без ошибок.

Языковые расширения крайне мощны, они разнятся от практических до экспериментальных. При написании промышленного кода на Haskell некоторые расширения встречаются очень часто и чрезвычайно полезны.

Проверка 23.2. Существует языковое расширение TemplateHaskell. Как бы вы компилировали файл templates.hs с использованием этого расширения? Как бы его включили, используя директиву LANGUAGE?

Ответ 23.2. Запуск компилятора GHC из командной строки:

```
$ ghc templates.hs -XTemplateHaskell
```

Первая строка файла templates.hs:

```
{-# LANGUAGE TemplateHaskell -#}
```

Другие языковые расширения Haskell

Языковые расширения встречаются в коде на Haskell довольно часто. Они позволяют вам использовать возможности языка, которые могут быть недоступны по умолчанию в течение многих лет, если вообще когда-либо будут. `OverloadedStrings` — самое распространённое. Вот ещё несколько, которые вы можете встретить или найти полезными:

- `ViewPatterns` — более продвинутое сопоставление с образцом;
- `TemplateHaskell` — инструменты для метапрограммирования;
- `DuplicateRecordFields` — решение проблемы из урока 16, когда одинаковые имена полей в разных типах, объявляемых в синтаксисе записей, вызывают конфликт;
- `NoImplicitPrelude` — отключает стандартный `Prelude` (мы уже отмечали, что многие программисты его не любят).

23.2.2. Простейшие функции обработки значений типа Text

Проблема с использованием типа `Text` вместо `String` состоит в том, что большинство полезных функций для работы с текстом предназначено для типа `String`. Вы определённо не хотите переводить `Text` обратно в `String` для использования функций вроде `lines`. К счастью, почти каждая важная функция для `String` имеет в `Data.Text` свою версию для `Text`. Давайте поиграем с неким текстом и разберёмся с тем, как эти функции работают.

Листинг 23.5 Некий текст типа Text

```
sampleInput :: T.Text
sampleInput = "это\nнекий\nтекст"
```

Чтобы вызвать правильную функцию `lines`, мы должны предварить её имя префиксом `T.` (это необходимо из-за квалифицированного импорта). Вот пример в GHCi:

```
GHCi> T.lines sampleInput
["это","некий","текст"]
```

Далее следуют некоторые другие полезные функции, существующие в версиях и для `Text`, и для `String`.

Функция words

Функция `words` аналогична `lines`, но работает для любых пробельных символов, а не только для символов перевода строк.

Листинг 23.6 Какой-то другой текст как пример ввода для words

```
someText :: T.Text
someText = "Какой-то\nсовсем другой\t текст"
```

Вы легко можете увидеть, как это работает, в GHCi:

```
GHCi> T.words someText
["Какой-то","совсем","другой","текст"]
```

Функция splitOn

В уроке 22 вкратце упоминалась функция `splitOn`, её версия для строк определена в модуле `Data.List.Split`. К счастью, версия для `Text` уже имеется в `Data.Text` и дополнительных импортов не нужно. Функция `splitOn` позволяет разделять текст по любой его подстроке.

Листинг 23.7 Пример использования splitOn

```
breakText :: T.Text
breakText = "просто"

newText :: T.Text
newText = "Это просто сделать"
```

И в GHCi:

```
GHCi> T.splitOn breakText newText
["Это "," сделать"]
```

Функции unwords и unlines

Разбиение текста по пробельным символам — обычная практика при работе с I/O. Обратное преобразование тоже часто встречается, соответствующие функции называются `unwords` и `unlines`. Применять их несложно, но эти функции очень полезны, так что стоит иметь их в своём репертуаре:

```
GHCi> T.unlines (T.lines sampleInput)
"это\nнекий\nтекст"
GHCi> T.unwords (T.words someText)
"Какой-то совсем другой текст"
```

Функция intercalate

Вы использовали строковую версию функции `intercalate` в уроке 18. Это обратная к `splitOn` функция:

```
GHCi> T.intercalate breakText (T.splitOn breakText newText)
"Это просто сделать"
```

Почти любая полезная функция для работы со строками имеет свою версию для `Text`.

Проверка 23.3. Напишите собственные реализации двух функций `T.lines` и `T.unlines`, используя функции `splitOn` и `T.intercalate`.

Операции с моноидами

Исключение из правила, что большая часть полезных функций для строк работает также с `Text`, — операция `++`. Пока что вы использовали `++` для комбинирования строк:

```
combined :: String
combined = "some" ++ " " ++ "strings"
```

К сожалению, `++` определена только на списках, поэтому она не будет работать с `Text`. В уроке 17 мы обсуждали классы типов `Monoid` и `Semigroup`, которые позволяют комбинировать значения типов, имеющих экземпляры этих классов типов, и конкатенировать списки элементов таких типов. Эти два класса предоставляют общее решение к комбинированию и строк, и текста. Вы можете либо импортировать `Semigroup` и использовать `<>`, либо использовать `mconcat`:

Ответ 23.3

```
myLines :: T.Text -> [T.Text]
myLines text = T.splitOn "\n" text

myUnlines :: [T.Text] -> T.Text
myUnlines textLines = T.intercalate "\n" textLines
```

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text as T
import Data.Semigroup

combinedTextMonoid :: T.Text
combinedTextMonoid = mconcat ["некий"," ","текст"]

combinedTextSemigroup :: T.Text
combinedTextSemigroup = "некий" <> " " <> "текст"
```

Поскольку тип `String` тоже имеет экземпляр `Monoid` и `Semigroup`, строки можно комбинировать точно так же.

23.3. Тип Text и Юникод

Тип `Text` отлично поддерживает прозрачную работу с текстом в Юникоде. Было время, когда программисты могли игнорировать сложности в работе с не ASCII-текстом. Если текст содержал акценты или умляуты, их просто выкидывали. Считалось вполне допустимым заменять, к примеру, *Charlotte Brontë* на *Charlotte Bronte*. Игнорирование Юникода сегодня — это путь к катастрофе. Нет никаких причин отказывать пользователю в возможности записать своё имя с диакритическими знаками или провалить поддержку японских кандзи.

23.3.1. Поиск в тексте на санскрите

Чтобы убедиться в лёгкости применения типа `Text` для работы с символами Юникода, вы создадите простую программу, которая подсвечивает слова в тексте. Штука в том, что вы будете подсвечивать текст на санскрите в системе записи деванагари! Текст примера в Юникоде можно скопировать по ссылке, скопируйте его отсюда и вставьте в свой редактор: `https://gist.github.com/willkurt/4bced09adc2ff9e7ee366b7ad681cac6`.

Весь ваш код будет находиться в файле с названием `bg_highlight.hs`. Программа будет принимать запрос на поиск и собственно текст, а в выводе использовать фигурные скобки {} для подсвечивания вхождений слова, которое вы ищете. Например, если ваш запрос — *собака*, а основной текст — это *собака гуляет с собаками*, то стоит ожидать следующего вывода:

```
{собака} гуляет с {собака}ми
```

В этом задании мы будем подсвечивать слово *дхарма* из санскрита во фрагменте текста из *Бхагавад Гита*. Слово *дхарма* имеет в санскрите множество значений, от *долга* до отсылок на вселенский порядок и высшую справедливость. Санскрит — это язык, в котором нет одной системы записи. Наиболее популярная сегодня — деванагари, алфавит, используемый более чем в 120 языках, включая хинди. Вот как выглядит слово *дхарма* на санскрите, записанное в записи деванагари.

Листинг 23.8 Слово «дхарма» как значение типа Text (деванагари)

```
dharma :: T.Text
dharma = "धर्म"
```

Теперь возьмём выдержку из *Бхагавад Гита*, части индийского эпоса *Махабхарата*. Вот наш фрагмент.

Листинг 23.9 Текст для поиска из Бхагавад Гита

```
bgText :: T.Text
bgText =
  "श्रेयान्स्वधर्मओ विगुणः परधर्मआत्स्वनुष्ठितात् स्वधर्मए् निधनं श्रेयः परधर्मओ भयावहः"
```

Ваша цель — выделить все вхождения слова *дхарма* в `bgText`. Во многих языках вы бы первым делом разбили предложение при помощи `T.words` на слова, а затем принялись бы искать среди них требуемое. Однако санскрит гораздо сложнее. Поскольку на санскрите говорили задолго до того, как на нём начали писать, всякий раз, когда произносимые слова сливаются в устной речи, они сливаются и при написании. Чтобы решить эту задачу, нужно разбить данный фрагмент текста по запросу, обернуть запрос в скобки и затем собрать обратно. Для разбиения текста можно использовать `T.splitOn`, для добавления скобок — `mconcat` и для сборки результата — `T.intercalate`. В результате получаем функцию `highlight`.

Листинг 23.10 Подсвечивание сегментов текста функцией highlight

```
highlight :: T.Text -> T.Text -> T.Text
highlight query fullText = T.intercalate highlighted pieces
  where pieces = T.splitOn query fullText
        highlighted = mconcat ["{",query,"}"]
```

Функция `splitOn` позволяет найти все вхождения текста запроса и разбить по ним исходный текст

Функцию `mconcat` можно применить для обрамления запроса фигурными скобками

Функция `intercalate` позволяет собрать всё воедино

Остаётся собрать это всё в функции `main`. Но прежде вам нужно узнать, как вводить и выводить значения типа `Text`.

23.4. Ввод-вывод для Text

Теперь, когда у вас есть функция highlight, вы хотите выводить результаты подсвечивания пользователям. Проблема в том, что до сих пор для вывода текста вы использовали тип String. Одно из решений состояло бы в том, чтобы распаковывать текст в строку и воспользоваться putStrLn. Однако хочется иметь putStrLn для Text, чтобы нам никогда не приходилось преобразовывать текст в строку (в надежде навсегда забыть о строках). Модуль Data.Text включает в себя только функции для манипуляций с текстом. Для организации ввода-вывода значений типа Text вам нужно импортировать модуль Data.Text.IO. Сделаем ещё один квалифицированный импорт:

```
import qualified Data.Text.IO as TIO
```

Теперь, пользуясь функцией TIO.putStrLn, вы можете печатать значения типа Text так же, как и String. Любое действие, связанное с IO в связи с String, имеет аналог в Data.Text.IO. Теперь всё готово для написания функции main, которая будет вызывать функцию highlight на наших данных. Приведём полный текст программы, включающий необходимые импорты и директиву LANGUAGE.

Листинг 23.11 Программа для подсвечивания результатов поиска

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text as T
import qualified Data.Text.IO as TIO

dharma :: T.Text
dharma = "धर्म"

bgText :: T.Text
bgText =
  "श्रेयान्स्वधर्मओ विगुणः परधर्मआत्स्वनुष्ठितात् स्वधर्मए निधनं श्रेयः परधर्मओ भयावहः"

highlight :: T.Text -> T.Text -> T.Text
highlight query fullText = T.intercalate highlighted pieces
  where pieces = T.splitOn query fullText
        highlighted = mconcat ["{",query,"}"]

main = do
  TIO.putStrLn (highlight dharma bgText)
```

Компилируем программу, запускаем и видим подсвеченный текст:

```
$./bg_highlight
श्रेयान्स्व{धर्म}ओ विगुणः पर{धर्म}आत्स्वनुष्ठितात् स्व{धर्म}ए निधनं श्रेयः पर{धर्म}ओ भयावहः
```

Теперь у вас есть программа, которая легко обрабатывает Юникод и делает это гораздо эффективнее, чем `String`.

Итоги

В этом уроке нашей целью было научить вас эффективно обрабатывать текст (включая Юникод), используя `Data.Text`. Хотя строки в виде списка символов — это полезный для изучения Haskell инструмент, на практике их использование может привести к проблемам с производительностью. Предпочтительная альтернатива для работы с текстовыми данными — это применение типа `Text`. Одна из проблем, на которую вы можете наткнуться, — это неумение Haskell по умолчанию распознавать строковые литералы как значения типа `Text`. Это можно вылечить включением языкового расширения `OverloadedStrings`. Давайте посмотрим, как вы это поняли.

Задача 23.1. Перепишите программу `hello_world.hs` из урока 21 (воспроизведена здесь), заменив `String` на `Text`.

```
helloPerson :: String -> String
helloPerson name = "Привет, " ++ name ++ "!"

main :: IO ()
main = do
   putStrLn "Привет! Как тебя зовут?"
   name <- getLine
   let statement = helloPerson name
   putStrLn statement
```

Задача 23.2. Пользуясь типом `Text` и импортируя модули `Data.Text.Lazy` и `Data.Text.Lazy.IO`, перепишите программу суммирования вводимых пользователем чисел из урока 22 (раздел по ленивому вводу-выводу).

```
toInts :: String -> [Int]
toInts = map read . lines

main :: IO ()
main = do
   userInput <- getContents
   let numbers = toInts userInput
   print (sum numbers)
```

24

Работа с файлами

После прочтения урока 24 вы:

- научитесь работать с файловыми дескрипторами в Haskell;
- сможете записывать и читать файлы;
- разберётесь с ограничениями ленивых вычислений с точки зрения ввода-вывода.

Одно из важнейших применений ввода-вывода — считывание и запись файлов. К данному моменту в этом модуле вы частично изучили синтаксис для работы с типами `IO` в Haskell, посмотрели, как создавать приложения, работающие с командной строкой с помощью ленивых вычислений, и узнали об эффективной обработке текстовых данных с помощью типа `Text`. Теперь вы рассмотрите работу с файлами, в том числе некоторые затруднения, возникающие в связи с ленивостью вычислений. Вы начнёте с основных операций для открытия, закрытия, чтения и записи в обычные файлы. Затем напишете программу, вычисляющую по содержимому входного файла некоторые статистические данные (такие как количество слов и символов) и записывающую их в другой файл. Вы выясните, что даже при решении такой простой задачи ленивые вычисления могут доставить ряд неудобств. Решение проблемы — в использовании строгих типов данных, они заставляют программу работать в соответствии с ожиданиями.

Обратите внимание. В уроке 22 вы решали задачу сложения чисел, вводимых пользователем. Как бы вы её решили для чисел в файле (не подавая их на вход программе вручную)?

24.1. Открытие и закрытие файлов

Перед изучением приёмов работы с файлами в Haskell нужно иметь хотя бы один. Сейчас вы рассмотрите основы открытия и закрытия файлов. Вашим первым заданием будет открыть и закрыть текстовый файл. Начнём с файла hello.txt со следующим содержимым:

Листинг 24.1 Файл hello.txt

```
Привет, мир!
Пока, мир!
```

Вам также понадобится файл, в который вы поместите свой код. Создайте файл под названием hello_file.hs. Для начала вам понадобится импортировать модуль `System.IO`, позволяющий производить считывание и запись в файлы:

```
import System.IO
```

Для того чтобы начать работать с файлом, вам понадобится его открыть. Для этого можно воспользоваться функцией `openFile`, имеющей следующую типовую аннотацию (напоминание: вы можете использовать команду `:t` для просмотра типа функции в GHCi):

```
openFile :: FilePath -> IOMode -> IO Handle
```

Как обычно бывает, чем лучше вы понимаете тип функции, тем лучше вы понимаете принцип её работы. Если вы откроете GHCi и воспользуетесь командой `:info`, то узнаете, что `FilePath` — просто синоним типа для `String`:

```
type FilePath = String
```

Применив `:info` к `IOMode`, вы узнаете, что этот тип похож на `Bool` и состоит только из конструкторов:

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

По именам конструкторов должно быть ясно, что `IOMode` определяет, собираетесь ли вы читать, писать, дополнять или совершать иные действия со своим файлом. Это похоже на работу с файлами практически во всех остальных языках программирования, где от пользователя требуется указать, что он собирается делать с файлом, к которому получает доступ.

Проверка 24.1. Как будет выглядеть вызов функции для открытия файла под названием `"stuff.txt"` на чтение?

При открытии файла вы получаете значение `IO Handle`. Тип `Handle` представляет собой дескриптор файла, позволяющий работать со ссылкой на файл. Как мы уже обсуждали в этом модуле, тип `IO` означает, что вы имеете дескриптор в контексте `IO`. Чтобы получить этот файловый дескриптор, вам придётся работать внутри действия ввода-вывода (например, `main`).

Теперь давайте соберём всё вместе и откроем файл hello.txt. Единственное, чего нам не хватает: как и в большинстве других языков программирования, после завершения работы с открытым файлом его надо закрывать. Это можно сделать с помощью функции `hClose` (от *handle close*, то есть *закрыть дескриптор*).

Листинг 24.2 Функция main, открывающая и закрывающая файл

```
main :: IO ()
main = do
   myFile <- openFile "hello.txt" ReadMode
   hClose myFile
   putStrLn "готово!"
```

Довольно скучно открывать и закрывать файл, если вы ничего с ним не делаете! Вам определённо нужно считывать и записывать данные в файлы. Это делается с помощью функций `hPutStrLn` и `hGetLine`. Единственное их отличие от `putStrLn` и `getLine` состоит в том, что вам нужно передавать дескриптор в качестве аргумента. Оказывается, `putStrLn` является частным случаем `hPutStrLn`: предполагается, что в качестве дескриптора ей передаётся `stdout` (стандартный поток вывода). Аналогичным образом `getLine` является частным случаем `hGetLine`, где в качестве дескриптора задан `stdin`. Ниже приведена модифицированная версия вашего кода, считывающая первую строку из файла hello.txt и выводящая её на консоль, затем считывающая вторую строку из того же файла и записывающая её в новый файл под названием goodbye.txt.

Ответ 24.1

```
openFile "stuff.txt" ReadMode
```

Листинг 24.3 Чтение из файла и запись в stdout и другой файл

```
main :: IO ()
main = do
   helloFile <- openFile "hello.txt" ReadMode
   firstLine <- hGetLine helloFile
   putStrLn firstLine
   secondLine <- hGetLine helloFile
   goodbyeFile <- openFile "goodbye.txt" WriteMode
   hPutStrLn goodbyeFile secondLine
   hClose goodbyeFile
   putStrLn "готово!"
```

Эта программа работает благодаря тому, что вы знаете, что hello.txt содержит две строки. Что, если вам нужно изменить программу для чтения и вывода всех строк в порядке их следования? Вам понадобится проверять достижение конца файла. Это делается с помощью `hIsEOF`. Ниже приведена версия программы, проверяющая содержимое файла hello.txt перед выводом первой строки.

Листинг 24.4 Проверка файла на непустоту перед чтением из него

```
main :: IO ()
main = do
   helloFile <- openFile "hello.txt" ReadMode
   hasLine <- hIsEOF helloFile
   firstLine <- if not hasLine
                then hGetLine helloFile
                else return "пустой файл"
   putStrLn "готово!"
```

Проверка 24.2. Проверьте наличие в файле второй строки.

Ответ 24.2. Если второй строки нет, возвращаем пустую строку:

```
hasSecondLine <- hIsEOF helloFile
secondLine <- if not hasSecondLine
              then hGetLine helloFile
              else return ""
```

24.2. Простые средства ввода-вывода

Хотя понимать, как работают файловые дескрипторы, полезно, во многих случаях вы можете обойтись без взаимодействия с ними напрямую. Есть несколько полезных функций под названиями `readFile`, `writeFile` и `appendFile`, которые скрывают многие детали чтения, записи и дополнения файлов. Ниже приведены их типовые аннотации:

```
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

Чтобы увидеть, как используются эти функции, создайте программу под названием fileCounts.hs. Ваша программа будет принимать файл в качестве аргумента и подсчитывать количество символов, слов и строк в файле. Она будет показывать эти данные пользователю, а также дописывать их в файл stats.dat. Ниже приведён пример возможного содержимого файла stats.dat после применения вашей программы к двум файлам: hello.txt и what.txt.

Листинг 24.5 Пример содержимого файла stats.dat

```
hello.txt символов: 29 слов: 5 строк: 2
what.txt символов: 30000 слов: 2404 строк: 1
```

Теперь вы можете написать код, выполняющий соответствующий анализ файлов. Первым шагом будет написание функции, вычисляющей все статистические данные. Её можно писать в предположении, что входные данные представлены как `String`. Вычисленные данные будут представлены как кортеж из трёх элементов.

Листинг 24.6 Функция getCounts собирает статистику в кортеж

```
getCounts :: String -> (Int, Int, Int)
getCounts input = (charCount, wordCount, lineCount)
  where charCount = length input
        wordCount = (length . words) input
        lineCount = (length . lines) input
```

Теперь вам нужно создать функцию `countsText`, преобразующую трёхэлементный кортеж в читаемое представление. Для склейки текста можно воспользоваться функцией `unwords`.

Листинг 24.7 Функция countsText и данные в читаемом виде

```
countsText :: (Int, Int, Int) -> String
countsText (cc, wc, lc) = unwords ["символов: "
                                   , show cc
                                   , " слов: "
                                   , show wc
                                   , " строк: "
                                   , show lc]
```

Работу этой функции вы можете проверить в GHCi:

```
GHCi> (countsText . getCounts) "это какой-то\n текст"
"символов: 20 слов: 3 строк: 2"
```

Проверка 24.3. Почему использование `unwords` предпочтительнее комбинирования строк с помощью `++`?

Теперь можно с лёгкостью совместить эти функции и реализовать требуемое поведение программы, воспользовавшись помощью `readFile` и `writeFile`.

Листинг 24.8 Совмещаем код в функции main

```
main :: IO ()
main = do
  args <- getArgs
  let fileName = head args
  input <- readFile fileName
  let summary = (countsText . getCounts) input
  appendFile "stats.dat"
             (mconcat [fileName, " ", summary, "\n"])
  putStrLn summary
```

Ответ 24.3. Операция `++` работает только со списками. В уроке 23 мы подробно разбирали другие текстовые типы, помимо `String`. Функция `unwords` имеет версию и для `Text`, и для `String`, в то время как `++` работает только для типа `String`. Использование `unwords` значительно облегчает модификацию кода в случае принятия решения о замене `String` на `Text`.

Если вы скомпилируете программу, то увидите, что она работает именно так, как требовалось.

```
$ ./fileCounts hello.txt
символов: 29 слов: 5 строк: 2

$ cat stats.dat
hello.txt символов: 29 слов: 5 строк: 2
```

Использование `readFile` и `appendFile` значительно облегчило решение этой задачи, по сравнению с использованием дескрипторов и функции `openFile`.

24.3. Проблемы ленивого ввода-вывода

В программе fileCounts.hs, очевидно, не хватает множества важных проверок: вы не проверяете наличие аргументов или существование файла. Эти проверки специально были опущены, чтобы сделать код проще для изучения. Есть один интересный способ вызвать наверняка неожиданную ошибку. Что произойдёт, если попытаться использовать `fileCounts` на собственном файле stats.dat?

```
$ ./fileCounts stats.dat
fileCounts: stats.dat: openFile: resource busy (file is locked)
```

Вы получили ошибку! Проблема состоит в том, что `readFile` не закрывает файловый дескриптор. Функция `readFile` неявно использует функцию `hGetContents`, работающую так же, как `getContents`, за исключением того, что `hGetContents` требует передачи дескриптора файла в качестве аргумента. Реализация `readFile` в Haskell выглядит следующим образом:

```
readFile :: FilePath -> IO String
readFile name = do
  inputFile <- openFile name ReadMode
  hGetContents inputFile
```

Как видите, этот код не закрывает дескриптор, а просто возвращает результат вызова `hGetContents`. Если вы хотите посмотреть исходный код какой-нибудь функции Haskell, то можете выполнить её поиск на Hackage, полученное в результате поиска описание будет содержать ссылку на исходный файл, где она определена.

Вы можете увидеть, почему readFile не закрывает дескриптор, если попытаетесь исправить main, явно выписав необходимые вам операции. Изменённый код будет выглядеть следующим образом:

Листинг 24.9 Раскрытие тела функции readFile в функции main

```
main :: IO ()
main = do
  args <- getArgs
  let fileName = head args
  file <- openFile fileName ReadMode
  input <- hGetContents file
  hClose file
  let summary = (countsText . getCounts) input
  appendFile "stats.dat"
             (mconcat [fileName. " ", summary, "\n"])
  putStrLn summary
```

Этот код выглядит слишком подробным, но, по идее, должен предотвратить ошибку, возникающую при вызове appendFile из-за записи в ещё открытый файл. Давайте перекомпилируем программу и попробуем снова:

```
$ ./fileCounts stats.dat
fileCounts: stats.dat: hGetContents: illegal operation
                              ↳ (delayed read on closed handle)
```

Ошибка в этот раз выглядит даже более странно, чем в предыдущий! Ваша программа теперь окончательно сломана и не будет работать даже со старым файлом hello.txt:

```
$ ./fileCounts hello.txt
fileCounts: stats.dat: hGetContents: illegal operation
                              ↳ (delayed read on closed handle)
```

Проблема возникает из-за ленивости вычислений. Ключевая особенность ленивых вычислений: код не выполняется до тех пор, пока этого не потребуется. Значение input не используется до тех пор, пока не определено summary. Но проблемы на этом не заканчиваются: summary не используется до тех пор, пока вы не вызовете appendFile. Так как appendFile выполняет действие ввода-вывода, для его выполнения приходится вычислить summary, что, в свою очередь, приводит к вычислению input. Настоящая проблема заключается в том, что hClose закрывает файл немедленно, потому что является действием ввода-вывода и выполняется сразу. На рис. 24.1 представлена визуализация описанного выше процесса.

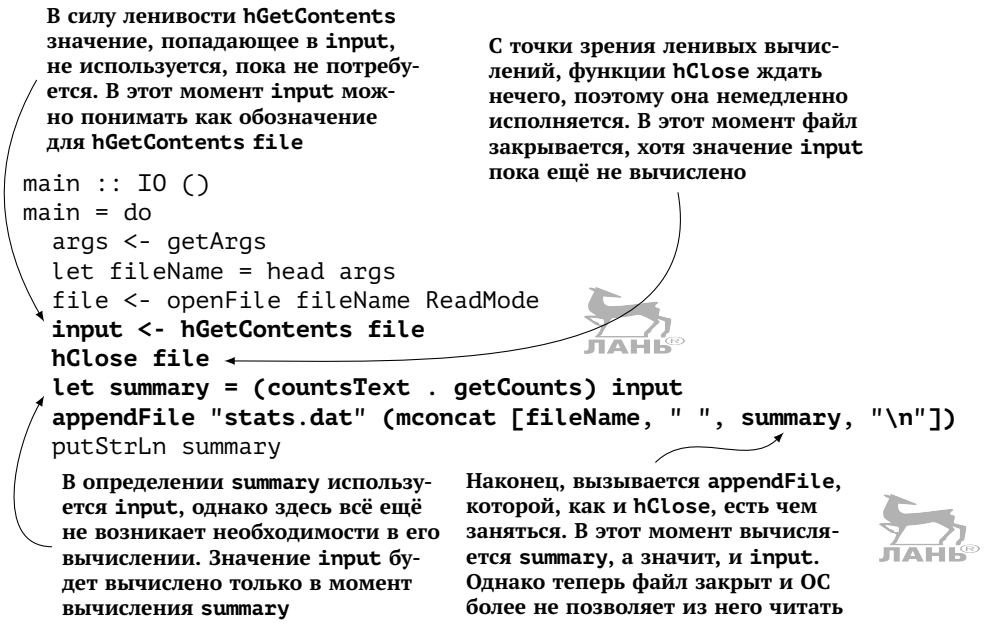


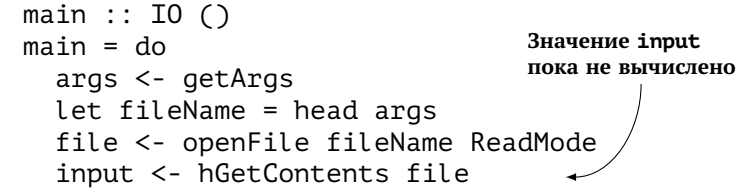
Рис. 24.1: Проблема ленивых вычислений: файл закрыт в момент чтения

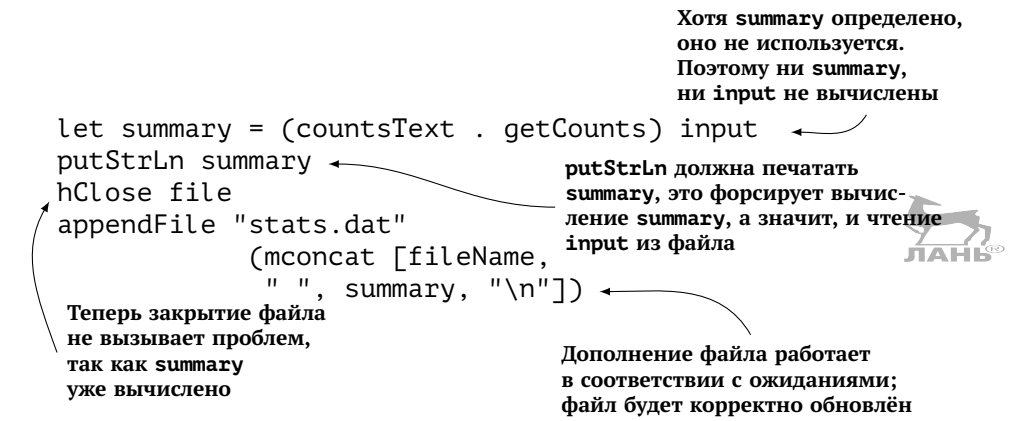
То есть можно поместить `hClose` после `appendFile`, потому что в этом месте `summary` наконец-то вычисляется, так?

```
appendFile "stats.dat"
           (mconcat [fileName, " ", summary, "\n"])
hClose file
```

Теперь вы вернулись туда, откуда начали; файл закрывается после того, как вам начали в него писать! Вам нужно решение, обеспечивающее вычисление `summary` перед записью в файл. Одним из способов достижения этого может быть выполнение `putStrLn summary` перед записью в файл. Таким образом вы обеспечите предварительное вычисление `summary`. Затем можно будет закрыть дескриптор, дополнив, наконец, файл.

Листинг 24.10 Функция main с исправлениями





Это должно быть достаточным доводом в пользу того, что ленивый ввод-вывод может быть мощным средством языка, но также может приводить к неприятным проблемам.

Проверка 24.4. Почему `readFile` не закрывает дескриптор?

24.4. Строгий ввод-вывод

Лучшим решением описанной в предыдущем разделе проблемы является использование строгого (неленивого) типа. В уроке 23 мы упоминали, что тип `Data.Text` предпочтительнее `String` при работе с текстовыми данными. Мы также отметили, что `Data.Text` является строгим типом данных (то есть не использует ленивых вычислений). Вы можете переписать исходную программу с использованием типа `Text`, и проблема будет решена!

```
{-# LANGUAGE OverloadedStrings #-}

import System.IO
import System.Environment
import qualified Data.Text as T
```

Ответ 24.4. Если бы `readFile` закрывала файл, то из-за ленивости вычислений вы бы никогда не смогли воспользоваться его содержимым. Причина в том, что функция, обрабатывающая содержимое файла, вызывается только после того, как файл уже закрыт.

```
import qualified Data.Text.IO as TI

getCounts :: T.Text -> (Int, Int, Int)
getCounts input = (charCount, wordCount, lineCount)
  where charCount = T.length input
        wordCount = (length . T.words) input
        lineCount = (length . T.lines) input

countsText :: (Int, Int, Int) -> T.Text
countsText (cc, wc, lc) = T.pack (unwords ["chars: "
                                           , show cc
                                           , " words: "
                                           , show wc
                                           , " lines: "
                                           , show lc])

main :: IO ()
main = do
  args <- getArgs
  let fileName = head args
  input <- TI.readFile fileName
  let summary = (countsText . getCounts) input
  TI.appendFile "stats.dat"
                (mconcat [(T.pack fileName),
                 " ", summary, "\n"])
  TI.putStrLn summary
```

Строгость вычисления означает, что ваш код для ввода-вывода работает так, как вы бы ожидали в любом другом языке программирования. Хотя ленивые вычисления имеют множество преимуществ, для любого нетривиального ввода-вывода поведение программы может быть довольно запутанным. Ваша fileCounts.hs была простой демопрограммой, однако даже в ней у вас возникла неприятная ошибка, связанная с ленивостью вычислений.

24.4.1. Выбор между ленивыми и строгими вычислениями

В этом модуле вы увидели случаи, в которых ленивые вычисления, связанные с вводом-выводом, могут значительно упростить или значительно усложнить жизнь программиста. Ключевым фактором, определяющим выбор между строгими и ленивыми вычислениями, является сложность ввода-вывода в вашей программе. Если программа читает данные из единственного файла и выполняет относительно мало работы с вводом-выводом, использование ленивых вычислений, скорее всего, обеспечит вам

множество преимуществ и малое количество проблем. Как только ввод-вывод становится даже умеренно сложным, включая чтение и запись в файлы или операции, в которых важен порядок, используйте строгие вычисления.

Итоги

В этом уроке нашей целью было научить вас основам чтения и записи файлов в Haskell. Файловый ввод-вывод в основном похож на другие формы ввода-вывода, уже вами рассмотренные. Используя ленивый ввод-вывод без понимания того, как это отразится на поведении вашей программы, вы можете столкнуться с определёнными проблемами. Хотя ленивый ввод-вывод может значительно упростить код, с возрастанием сложности программы становится очень трудно рассуждать о логике её работы. Давайте удостоверимся, что вы всё поняли.

Задача 24.1. Напишите свой аналог программы `cp` в Unix, копирующей файл и позволяющей его при этом переименовать (достаточно воспроизвести только базовый функционал; не волнуйтесь о специфических возможностях утилиты `cp`).

Задача 24.2. Напишите программу под названием capitalize.hs, принимающую имя файла в качестве аргумента, считывающую этот файл и перезаписывающую его теми же символами, но прописными буквами.

25

Работа с двоичными данными

После изучения урока 25 вы:

- сможете использовать `ByteString` для эффективной работы с двоичными данными;
- научитесь обрабатывать `ByteString` как обычные ASCII-строки с помощью `ByteString.Char8`;
- сможете добавлять помехи в JPEG-изображения;
- освоите работу с двоичными данными Юникода.

В данном уроке вы научитесь работать с двоичными файлами, используя тип `ByteString`. Этот тип позволяет обрабатывать сырые двоичные данные как обычные строки. В демонстрации использования `ByteString` мы сфокусируемся на небольшом проекте по манипуляции двоичными файлами. В этом проекте мы реализуем консольное приложение, которое позволит создавать *глитч-арты*, напоминающие то, что изображено на рис. 25.1.

Рис. 25.1: Сцена из глитч-видео Микаэля Бетанкура «Kodak Moment» (2013)

Глитч-арт — сознательная порча данных с целью создания интересных визуальных артефактов на изображениях и видеозаписях. Вы будете реализовывать относительно простую задачу — создание помех на картинках в формате JPEG. Также мы рассмотрим несколько аспектов работы с двоичными данными Юникода.

Обратите внимание. Допустим, у вас есть строка, содержащая имя японского писателя Такимото Тацухико, записанное с помощью кандзи в переменную типа `T.Text`:

```
tatsuhikoTakimoto :: T.Text
tatsuhikoTakimoto = "滝本 竜彦"
```

Вам нужно узнать количество байт в этом тексте. Если бы это был ASCII-текст, то оно бы соответствовало длине строки, но в нашем случае использование `T.length` возвращает количество символов. Сможете ли вы узнать число байт текста?

25.1. Обработка двоичных данных с помощью ByteString

До сих пор в этой главе мы рассматривали только взаимодействие с текстом в файлах. Вы начали работать с базовым типом `String` и затем узнали, что `Text` лучше подходит для работы с текстом. Другой важный тип, напоминающий `String` и `Text`, называется `ByteString`. Интересная особенность этого типа заключается в том, что он не предназначен именно для хранение текста, как можно подумать из-за присутствия в названии `String`. `ByteString` — эффективный способ оперирования потоками двоичных данных. Как и в случае с `Data.Text`, вы можете использовать квалифицированный импорт модуля `Data.ByteString`, связав его с какой-нибудь буквой:

```
import qualified Data.ByteString as B
```

Даже несмотря на то, что `ByteString` является не текстовым типом, а всего лишь обычным массивом байтов, вы всё равно можете использовать ASCII для представления байтов в виде строк. В таблице ASCII находится 256 (2^8, так как один байт равен 8 битам) символов, так что каждый байт может быть представлен отдельным символом. При условии, что вы включили расширение `OverloadedStrings`, вы можете использовать строковые литералы из ASCII-символов как значения типа `ByteString`.

Листинг 25.1 Значение ByteString, определённое с OverloadedStrings

```
sampleBytes :: B.ByteString
sampleBytes = "Привет!"
```

Но вы столкнётесь с проблемой, если попытаетесь преобразовать значения типа `ByteString` к обычной строке, используя `B.unpack`. Следующий код вызовет ошибку компиляции.

Листинг 25.2 Распаковываем ByteString в String — ошибка!

```
sampleString :: String
sampleString = B.unpack sampleBytes
```

Как вы можете видеть по типовой аннотации, `B.unpack` пытается преобразовать `ByteString` в список байтов, которые представлены с помощью типа `Word8`:

```
B.unpack :: B.ByteString -> [GHC.Word.Word8]
```

По умолчанию `Data.ByteString` не позволяет обрабатывать байты как `Char`, поэтому для работы с отдельными байтами стоит использовать тип `Data.ByteString.Char8`, где `Char8` расшифровывается как «8-битный `Char`» (ASCII-символ). Модуль, в котором этот тип определён, нужно подключать отдельно, обычно это делается с префиксом `BC`:

```
import qualified Data.ByteString.Char8 as BC
```

Разницу между `ByteString` и `ByteString.Char8` можно легко заметить, если посмотреть на типы относящихся к ним функций `unpack`:

```
B.unpack :: BC.ByteString -> [GHC.Word.Word8]
BC.unpack :: BC.ByteString -> [Char]
```

Тут вы можете увидеть, что `unpack ByteString.Char8` работает аналогично одноимённой функции для `Data.Text`. Тип `ByteString.Char8` позволяет вам использовать те же базовые функции, предназначенные для работы с текстом, что и `Data.Text`. Внимательный читатель также заметит, что тип функции `B.unpack` тоже изменился! `B.unpack` теперь использует представление `ByteString` из `ByteString.Char8`. Это означает, что вы вполне можете рассматривать `ByteString` как ASCII-текст.

Как и `Text`, `ByteString` имеет такой же API, что и `String`. Как мы увидим в следующем разделе, при работе с двоичными данными вы можете использовать те же функции, которыми обрабатывали `Text` и `String`. Это позволяет рассуждать об эффективности работы с двоичными данными, проводя параллели с обычными списками.

Проверка 25.1. Напишите функцию, которая принимает число в виде строки из ASCII-символов и переводит его в `Int`. Для примера преобразуйте следующее значение в `Int`:

```
bcInt :: BC.ByteString
bcInt = "6"
```

25.2. Добавление помех на изображение JPEG

Мы рассмотрели основы работы с `ByteString`, теперь же давайте погрузимся в глитч-арт! Весь код для этой программы нужно поместить в файл glitcher.hs. Воспользуемся изображениями, загруженными из Википедии (например, `https://en.wikipedia.org/wiki/H._P._Lovecraft#/media/File:H._P._Lovecraft,_June_1934.jpg`). Сохраните эту фотографию, показанную также на рис. 25.2, в файл lovecraft.jpg.

Для начала давайте посмотрим, какой основной функционал вам потребуется для чтения изображения и его изменения. Ниже представлена базовая структура программы:

(1) на вход подаётся имя файла;

(2) из этого файла читаются двоичные данные;

(3) случайным образом изменяются байты изображения;

(4) получившаяся картинка сохраняется в новом файле.

Для работы с данными, представляющими изображение, вы будете использовать и `Data.ByteString`, и `Data.ByteString.Char8`. Так как вы работаете с двоичными данными, чтение файла удобнее будет осуществлять

Ответ 25.1

```
bcInt :: BC.ByteString
bcInt = "6"

bcToInt :: BC.ByteString -> Int
bcToInt = read . BC.unpack
```

Рис. 25.2: Наша цель для создания глитчей

с помощью `BC.readFile`. Вот быстрый набросок вашей будущей программы; код, отвечающий за создание помех, пока что отсутствует.

Листинг 25.3 Наброски glitcher.hs

```
import System.Environment
import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as BC

main :: IO ()
main = do
    args <- getArgs
    let fileName = head args
    imageFile <- BC.readFile fileName
    glitched <- return imageFile
    let glitchedFileName = mconcat ["glitched_",fileName]
    BC.writeFile glitchedFileName glitched
    putStrLn "Готово!"
```

Использование `getArgs` для чтения имени файла

Единственным аргументом должен быть файл

Затем читаем файл `BC`-версией `readFile`

Здесь используется `return`, которое мы позже заменим на действие ввода-вывода, изменяющее содержимое файла

Записываем результат в файл `BC`-версией функции `writeFile`

Создание помех может испортить файл, поэтому сохраним результат под новым именем

Этот кусочек кода позволит прочитать файл, название которого передаётся в качестве аргумента, и получить новый JPEG-файл, в который добавятся помехи. Единственная отсутствующая часть — код для добавления этих самых помех.

Проверка 25.2. На данном этапе переменная, связанная с искажёнными данными, не обязательно должна быть обёрнута в `IO`. Измените эту строку так, чтобы `glitched` стала обычной переменной.

25.2.1. Вставка случайных байтов

Своеобразный шарм глитч-арту придаёт то, что для его создания можно применять разные подходы, которые могут принести самые неожиданные результаты. Начнём мы с того, что заменим какой-нибудь байт случайным образом. Создание случайного числа требует работы с `IO`, но всегда лучше вынести как можно больше кода за пределы таких действий, поскольку обычные функции являются чистыми и предсказуемыми. К тому же вы можете легко протестировать то, что у вас получилось, загрузив код в GHCi и запустив его на нескольких вариантах входных данных.

Перед тем как приступить к созданию `IO`-действия, мы напишем функцию, которая переводит `Int` в `Char`. Так как `Char` реализует экземпляр `Enum`, можно воспользоваться функцией `toEnum`. Вы можете использовать только `toEnum`, но в таком случае нельзя будет обеспечить ограничение значений функции, которые должны быть в отрезке от 0 до 255. Для наложения этого ограничения вы можете взять остаток от деления аргумента на 255, а потом передать его `toEnum`. Соедините всё это в функции `intToChar`.

Листинг 25.4 Функция intToChar преобразует байт в Int

```
intToChar :: Int -> Char
intToChar int = toEnum safeInt
    where safeInt = int `mod` 255
```

Теперь вам потребуется функция, преобразующая `Char` в `ByteString`. Вы можете сделать это с помощью `BC.pack`, эта функция принимает `Char` и возвращает `BC.ByteString`. Так как `BC.pack` принимает строки, нужно завернуть `Char` в список.

Ответ 25.2

```
let glitched = imageFile
```

Листинг 25.5 intToBC преобразует Int в односимвольную ByteString

```
intToBC :: Int -> BC.ByteString
intToBC int = BC.pack [intToChar int]
```

Теперь, когда у вас есть способ преобразовать `Int` в один байт, представленный `BC.ByteString`, вы можете написать функцию для замены байта полученным значением. На данном этапе по-прежнему не требуется работа с `IO`.

Функция `replaceByte` — детерминированная часть нашего алгоритма, содержащего элемент случайности. Эта функция принимает позицию изменяемого байта, новое значение `Char`/`Byte`, представленное `Int`, и исходную последовательность байтов. Чтобы разбить эту последовательность в нужной позиции, вы можете воспользоваться функцией `BC.splitAt`, которая возвращает пару, состоящую из первой части списка и оставшегося куска (как если бы вы использовали `take` и `drop`). После нужно будет удалить первый байт второго элемента пары, чтобы освободить место для нового байта. В завершение остаётся соединить элементы пары, поместив между ними новый байт.

Листинг 25.6 Функция replaceByte заменяет байт на новый

Результатом работы функции является исходный список с одним изменённым значением

`BC.splitAt` возвращает пару значений, соответствующих `take` и `drop`. Сопоставление с образцом связывает их сразу с двумя переменными

```
replaceByte :: Int -> Int -> BC.ByteString -> BC.ByteString
replaceByte loc chV bytes = mconcat [before,newChar,after]
    where (before,rest) = BC.splitAt loc bytes
          after = BC.drop 1 rest
          newChar = intToBC chV
```

Здесь можно воспользоваться `BC.drop 1` для удаления заменяемого байта

Новый байт представлен ASCII-символом

Теперь мы готовы к работе с `IO`. Для достижения цели вам нужно будет воспользоваться `randomRIO` из модуля `System.Random`. Функция `randomRIO` принимает пару значений и возвращает случайный элемент между этими значениями. Назовём это `IO`-действие `randomReplaceByte`. Вся работа `randomReplaceByte` заключается в том, чтобы сгенерировать два случайных числа: одно для нового байта, а второе — для выбора места замены.

Листинг 25.7 Получение случайных чисел в randomReplaceByte

```
randomReplaceByte :: BC.ByteString -> IO BC.ByteString
randomReplaceByte bytes = do
    let bytesLength = BC.length bytes
    location <- randomRIO (1,bytesLength)
    chV <- randomRIO (0,255)
    return (replaceByte location chV bytes)
```

Теперь вы можете воспользоваться этой функцией в `main` для модификации изображения:

```
main :: IO ()
main = do
    args <- getArgs
    let fileName = head args
    imageFile <- BC.readFile fileName
    glitched <- randomReplaceByte imageFile
    let glitchedFileName = mconcat ["glitched_",fileName]
    BC.writeFile glitchedFileName glitched
    putStrLn "Готово!"
```

Скомпилируем программу и запустим её из командной строки:

```
$ ghc glitcher.hs
$ ./glitcher lovecraft.jpg
Готово!
```

Всё работает нормально, но результаты не такие впечатляющие, как вы, наверное, надеялись. Демонстрация работы программы представлена на рис. 25.3. Давайте попробуем сделать что-нибудь посложнее, вдруг результаты окажутся более интересными.

Рис. 25.3: Разочаровывающие результаты изменения одного байта

Проверка 25.3. Напишите IO-действие, возвращающее случайное значение типа Char.

25.2.2. Сортировка случайных байтов

Другой распространённой техникой для создания помех на изображениях является сортировка подпоследовательности байтов. Для этого можно разбить ByteString в какой-нибудь точке с помощью BC.splitAt, а потом разбить вторую половину ещё раз, отделив от неё фрагмент фиксированного размера; затем этот фрагмент сортируется и всё соединяется обратно с помощью mconcat. Ниже приведена функция sortSection, которая принимает начальную точку фрагмента для сортировки, его размер и последовательность байтов.

Листинг 25.8 Сортировка фрагмента данных

```
sortSection :: Int -> Int -> BC.ByteString -> BC.ByteString
sortSection start size bytes =
       mconcat [before,changed,after]
    where (before,rest) = BC.splitAt start bytes
          (target,after) = BC.splitAt size rest
          changed = BC.reverse (BC.sort target)
```

Всё, что теперь требуется, — добавить элемент случайности, реализовав соответствующее IO-действие, вызывающее функцию sortSection, а потом использовать это действие в main.

Ответ 25.3

```
randomChar :: IO Char
randomChar = do
    randomInt <- randomRIO (0,255)
    return (toEnum randomInt)
```

Вместо констант 0 и 255 можно было бы воспользоваться парой значений minBound и maxBound.

Листинг 25.9 Элемент случайности в sortSection

```
randomSortSection :: BC.ByteString -> IO BC.ByteString
randomSortSection bytes = do
    let sectionSize = 25
    let bytesLength = BC.length bytes
    start <- randomRIO (0,bytesLength - sectionSize)
    return (sortSection start sectionSize bytes)
```

Здесь вы произвольно выбираете размер сортируемого участка

Использование `randomRIO` помогает определить начало участка сортировки

Теперь можно заменить в функции `main` вызов `randomReplaceByte` на `randomSortSection` и попробовать запустить новую версию.

Листинг 25.10 Новая версия main с randomSortSection

```
main :: IO ()
main = do
    args <- getArgs
    let fileName = head args
    imageFile <- BC.readFile fileName
    glitched <- randomSortSection imageFile
    let glitchedFileName = mconcat ["glitched_",fileName]
    BC.writeFile glitchedFileName glitched
    print "all done"
```

Как вы видите на рис. 25.4, используя этот подход, вы можете получить более интересные результаты. Но результат был бы ещё лучше, если бы вы попробовали соединить эти приёмы!

Рис. 25.4: С `randomSortSection` можно получить кое-что поинтереснее

25.2.3. Комбинирование IO-действий с помощью foldM

Предположим, вы собираетесь применить на исходном изображении два раза `randomSortSection` и три раза `randomReplaceByte`. Вы могли бы сделать это таким образом.

Листинг 25.11 Выполнение набора действий: громоздкий подход

```
main :: IO ()
main = do
    args <- getArgs
    let fileName = head args
    imageFile <- BC.readFile fileName
    glitched1 <- randomReplaceByte imageFile
    glitched2 <- randomSortSection glitched1
    glitched3 <- randomReplaceByte glitched2
    glitched4 <- randomSortSection glitched3
    glitched5 <- randomReplaceByte glitched4
    let glitchedFileName = mconcat ["glitched_",fileName]
    BC.writeFile glitchedFileName glitched5
    putStrLn "Готово!"
```

Это будет нормально работать, но код определённо получился слишком громоздким, в нём легко совершить какую-нибудь простую ошибку, учитывая, что приходится следить за большим количеством переменных. Вместо этого вы можете воспользоваться функцией `foldM` из модуля `Control.Monad`. Прямо как `mapM` обобщает `map` для монад (в данный момент пока что считайте, что речь о do-нотации), так `foldM` делает то же самое для свёрток. С `foldM` вы можете взять `imageFile` в качестве начального значения и список IO-действий, которые преобразуют ваш файл. Единственная вещь, которой пока что не хватает, — функция, которая применит все эти функции. В нашем случае вы можете использовать простую лямбда-функцию. Вот как будет выглядеть `main`, если переписать эту функцию с использованием `foldM`.

Листинг 25.12 Улучшенная версия с использованием foldM

```
main :: IO ()
main = do
    args <- getArgs
    let fileName = head args
    imageFile <- BC.readFile fileName
    glitched <- foldM (\bytes func -> func bytes) imageFile
                      [ randomReplaceByte , randomSortSection
                      , randomReplaceByte , randomSortSection
```

```
                    , randomReplaceByte]
let glitchedFileName = mconcat ["glitched_",fileName]
BC.writeFile glitchedFileName glitched
putStrLn "Готово!"
```

Теперь вы можете наконец-то скомпилировать вашу программу последний раз и посмотреть, какие глитчи могут появиться на изображении! На рис. 25.5 представлен возможный результат.

Рис. 25.5: Теперь наш любимый автор больше похож на жителя Иннсмута

Возможно, есть приёмы, которые позволят сделать это изображение ещё круче, но теперь вы можете заняться этим самостоятельно, так как у вас есть способ соединять вместе настолько необычные преобразования, насколько только хватит фантазии.

Проверка 25.4. Создайте за пределами функции `main` переменную `glitchActions`, содержащую список всех требуемых действий. Не забудьте указать для неё корректный тип.

Ответ 25.4

```
glitchActions :: [BC.ByteString -> IO BC.ByteString]
glitchActions = [randomReplaceByte ,randomSortSection
                ,randomReplaceByte ,randomSortSection
                ,randomReplaceByte]
```

25.3. ByteString, Char8 и Юникод

Как вы видели в примере с созданием глитч-арта, `ByteString.Char8` — очень удобный инструмент для работы с байтами так, будто они являются обычным текстом. Но очень важно быть предельно внимательным при работе с `ByteString`, `ByteString.Char8` и данными Юникода. Вот пример подстановки Юникод-строки в `BC.ByteString` (в качестве строки воспользуемся именем известного философа Нагарджуны, записанным с помощью деванагари).

Листинг 25.13 Создание BC.ByteString с символами Юникода

```
nagarjunaBC :: BC.ByteString
nagarjunaBC = "नागार्जुन"
```

Если вы загрузите это в GHCi, то увидите, что Юникод-символы не сохранились:

```
GHCi> nagarjunaBC
"(>\ETB0M\FSA("
```

И это неудивительно, учитывая, что `ByteString` из `Char8` предназначены только для ASCII. Но вам может потребоваться преобразовать текст в байты, для этого могут быть разные причины, основная — записать Юникод-текст в файл в виде нескольких `ByteString`. Предположим, у вас есть состоящая из символов Юникода строка, представленная в виде `Text`.

Листинг 25.14 Юникод-символы, представленные как Text

```
nagarjunaText :: T.Text
nagarjunaText = "नागार्जुन"
```

Перевод `nagarjunaText` в массив байтов не получится осуществить с помощью `BC.pack`, так как тип этой функции: `String -> ByteString`, поэтому сначала воспользуемся функцией `T.unpack`, а затем `BC.pack`.

Листинг 25.15 Попытка преобразовать Text в ByteString

```
nagarjunaB :: B.ByteString
nagarjunaB = (BC.pack . T.unpack) nagarjunaText
```

Судя по типу, вы должны были получить Юникод-строку в виде байтов без всяких изменений, но если вы выполните обратное преобразование,

то поймёте, что что-то пошло не так. Обратите внимание, что для корректного вывода текста вам потребуется выполнить уточнённый импорт модуля `Data.Text.IO`:

```
GHCi> TIO.putStrLn ((T.pack . BC.unpack) nagarjunaB)
"(>\ETB0M\FSA("
```

И опять та же проблема! Если бы вы записали `nagarjunaB` в файл, вы бы потеряли все данные Unicode. Нам требуется способ трансформации `Text` напрямую в `B.ByteString` без промежуточного преобразования к `BC.ByteString`. Для этого нужно будет воспользоваться модулем `Data.Text.Encoding`, поэтому придётся выполнить ещё один квалифицированный импорт:

```
import qualified Data.Text.Encoding as E
```

Этот модуль содержит две важные функции, которые позволят вам выполнить прямое преобразование:

```
E.encodeUtf8 :: T.Text -> BC.ByteString
E.decodeUtf8 :: BC.ByteString -> T.Text
```

Теперь вы можете спокойно трансформировать Юникод в последовательность байтов и обратно.

Листинг 25.16 Преобразования между Text и ByteString

```
nagarjunaSafe :: B.ByteString
nagarjunaSafe = E.encodeUtf8 nagarjunaText

GHCi> TIO.putStrLn (E.decodeUtf8 nagarjunaSafe)
नागार्जुन
```

Во избежание подобных проблем никогда не пытайтесь использовать преимущества `Data.ByteString.Char8`, если работаете с данными, которые могут содержать Юникод. Если же вы работаете с двоичными данными, как в случае с примером из этого урока, комбинации `ByteString` и `Char8` вполне хватит. Для всего остального используйте `ByteString`, `Text` и `Text.Encoding`; в итоговом проекте этого модуля вы увидите хороший пример того, как работать с такими задачами.

Итоги

В этом уроке нашей основной целью было рассказать вам, как работать с двоичными данными при помощи Haskell. `ByteString` позволяет обрабатывать обычные двоичные данные как обычные строки. Это может здорово облегчить задачу по написанию программ, которые взаимодействуют с двоичными данными. Но очень важно помнить, что не стоит использовать вместе однобайтовое представление двоичных данных (`Char8`) и текст с символами Юникода. Давайте освежим в памяти пройденный материал.

Задача 25.1. Напишите программу, которая сравнивает количество символов в текстовом файле с его размером в байтах.

Задача 25.2. Реализуйте ещё один способ добавления помех под названием `randomReverseBytes`, который переворачивает случайную подпоследовательность байтов в файле.

26

Итоговый проект: обработка двоичных файлов и книжных данных

Этот итоговый проект включает в себя:

- изучение уникального используемого в библиотеках формата двоичных файлов;
- написание утилит для массовой обработки двоичных данных с помощью `ByteString`;
- работу с данными в кодировке Юникод с помощью типа `Text`;
- структурирование большой программы, выполняющей сложную задачу по вводу-выводу.

В этом итоговом проекте вы будете использовать двоичные описания книг в специальном формате, разработанном библиотеками, для создания простого документа в формате HTML. Библиотеки тратят большое количество денег и времени на создание и поддержание в актуальном состоянии каталогов с информацией о всех существующих в мире книгах. К счастью, довольно большое количество этих данных доступно бесплатно любому желающему. Одна только библиотека Гарвардского университета выложила в публичный доступ 12 миллионов записей о книгах (`http://library.harvard.edu/open-metadata`). Проект «The Open Library» также содержит миллионы записей для бесплатного использования (`https://archive.org/details/ol_data`).

Во времена огромной популярности науки о данных было бы здорово создать несколько интересных проектов для работы с этими данными. Но есть большое затруднение, связанное с их использованием. Библиотеки

хранят свои данные о книгах в довольно странном формате под названием *MARC record* (от Machine-Readable Cataloging record, машиночитаемая каталоговая запись). Это делает использование библиотечных данных гораздо более трудоёмким, по сравнению с более распространёнными форматами, такими как JSON или XML. MARC-записи хранятся в двоичном формате в кодировке Юникод, позволяющей должным образом хранить коды любых символов. При использовании MARC-записей нужно внимательно разделять работу с байтами и работу с текстовыми данными. Это превосходная задача для применения полученных вами в этом модуле знаний!

Цель этого проекта — преобразовать коллекцию MARC-записей в HTML-документ, содержащий перечисление авторов и названий всех книг в коллекции. План работы над проектом выглядит следующим образом:

- вы начнёте своё путешествие с создания типа для книжных данных, которые вы будете хранить и преобразовывать в HTML;
- затем вы изучите особенности формата MARC-записей;
- потом вы научитесь разбивать группы записей, объединённых в один файл, на списки записей;
- разделив записи, вы сможете обрабатывать файлы по отдельности для поиска нужной информации;
- наконец, вы совместите всё это в одну программу, которая будет преобразовывать MARC-записи в HTML-файлы.

Весь код вы будете писать в файле под названием marc_to_html.hs. Для начала вам потребуется импортировать несколько модулей (и включить расширение `OverloadedStrings`).

Листинг 26.1 Необходимые модули для программы marc_to_html.hs

Директива `LANGUAGE OverloadedStrings` нужна для использования строковых литералов со всеми строковыми типами

Так как вы работаете с двоичными данными, вам нужен способ обработки байтов

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString as B
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import qualified Data.Text.Encoding as E
import Data.Maybe
```

Для работы с текстом, а особенно с Юникодом, вам нужен тип `Text`

Функции ввода-вывода для типа `Text` подключаются отдельно

Вы будете использовать типы `Maybe`, поэтому понадобится функция `isJust`

Часть работы с Юникодом заключается в безопасном его кодировании и декодировании из двоичного вида

Возможно, вы заметили, что среди импортируемых модулей отсутствует `Data.ByteString.Char8`. Это связано с тем, что *при работе с данными в кодировке Юникод никогда не стоит смешивать Юникод и ASCII-текст*. Лучший способ обеспечить это разделение — использовать обычные `ByteString` для работы с байтами и `Text` для всего остального.

26.1. Работа с книжными данными

«Распаковка» MARC-записей обещает быть трудной задачей, поэтому было бы неплохо разобраться с тем, чего мы хотим достичь до того, как вы успеете запутаться. Ваша основная цель — преобразовать список книг в HTML-документ. Одним из препятствий на пути к нашей цели является неудобный формат, в котором хранятся книги. Для выполнения этого итогового проекта вам достаточно научиться обрабатывать только названия и авторов книг. Для этих свойств вы можете использовать синонимы типов. Вы могли бы воспользоваться типом `String`, но, как было отмечено в уроке 23, существует общее правило: использовать `Text` при выполнении любых серьёзных задач, связанных с обработкой текстовых данных.

Теперь вы можете создать синонимы типов `Author` и `Title`.

Листинг 26.2 Синонимы типов Author и Title

```
type Author = T.Text
type Title = T.Text
```

Тип `Book` будет типом-произведением `Author` и `Title`.

Листинг 26.3 Тип Book

```
data Book = Book {
    author :: Author
  , title :: Title} deriving Show
```

Ваша финальная функция будет называться `booksToHtml` и иметь тип `[Books] -> Html`. Перед тем как реализовать её, вам нужно определить тип для `Html` и, в идеале, создание HTML по данным одной книги. Для моделирования HTML вы можете снова воспользоваться типом `Text`.

Листинг 26.4 Синоним типа Html

```
type Html = T.Text
```

Для облегчения преобразования списка книг в HTML вы начнёте с создания фрагмента HTML для одной книги. HTML для книги будет представлять собой информацию, заключённую в тег <p>, с названием, выделенным с помощью , и автором — с помощью .

Листинг 26.5 Создание фрагмента HTML-кода для описания книги

```
bookToHtml :: Book -> Html
bookToHtml book = mconcat ["<p>\n"
                      , titleInTags
                      , authorInTags
                      , "</p>\n"]
  where titleInTags = mconcat [ "<strong>", title book
                              , "</strong>\n"]
        authorInTags = mconcat [ "<em>", author book
                               , "</em>\n"]
```

Вам также понадобятся примеры данных о книгах.

Листинг 26.6 Несколько примеров книг

```
book1 :: Book
book1 = Book {
    title = "Заговор против человека"
  , author = "Лиготти, Томас"
  }

book2 :: Book
book2 = Book {
    title = "Трактат о разложении"
  , author = "Чоран, Эмиль"
  }

book3 :: Book
book3 = Book {
    title = "Слёзы Эроса"
  , author = "Батай, Жорж"
  }
```

Эту часть кода вы можете протестировать в GHCi:

```
GHCi> bookToHtml book1
"<p>\n<strong>Заговор против человека</strong>\n<em>Лиготти,
                                                ↳ Томас</em>\n</p>\n"
```

Для преобразования списка книг можно применить `map` с функцией `bookToHtml` в качестве аргумента. Вам также стоит удостовериться,

что вы не забыли добавить теги `html`, `head`, `body`.

Листинг 26.7 Преобразование списка книг в HTML-документ

```
booksToHtml :: [Book] -> Html
booksToHtml books =
  mconcat [ "<html>\n"
          , "<head><title>книги</title>"
          , "<meta charset='utf-8'/>"
          , "</head>\n"]
          , "<body>\n"
          , booksHtml
          , "\n</body>\n"
          , "</html>"]
  where booksHtml = (mconcat . (map bookToHtml)) books
```

Важно указать кодировку, так как вы работаете с Юникодом

Для тестирования этой функции вы можете поместить свои книги в список:

```
myBooks :: [Book]
myBooks = [book1, book2, book3]
```

Наконец, можно создать `main` и протестировать написанный к данному моменту код. Предположим, что вы будете записывать данные в файл books.html. Как вы помните, тип `Html` — синоним типа `Text`. Чтобы записать текст в файл, нужно подключить модуль `Text.IO`.

Листинг 26.8 Версия main с записью списка книг в HTML

```
main :: IO ()
main = TIO.writeFile "books.html" (booksToHtml myBooks)
```

Запуск этой программы запишет данные в файл books.html. Открыв его (например, в браузере), вы сможете увидеть, что содержимое выглядит так, как и ожидалось (см. рис. 26.1).

Заговор против человека *Лиготти, Томас*
Трактат о разложении *Чоран, Эмиль*
Слёзы Эроса *Батай, Жорж*

Рис. 26.1: Данные о книгах в виде, отображаемом HTML-браузером

Получив возможность записывать книги в файл, можно разобраться с более сложной задачей обработки MARC-записей.

26.2. Работа с MARC-записями

MARC-запись — используемый библиотеками стандарт для хранения и передачи информации о книгах (такая информация называется *библиографическими данными*). Если вас интересуют данные о книгах, важно научиться понимать формат MARC-записей. В сети можно найти множество доступных бесплатно MARC-записей. Для итогового проекта вы будете использовать библиотечные записи библиотеки Орегонского университета здоровья и науки. Как уже было сказано, MARC является сокращением от *Machine-Readable Cataloging record* (машиночитаемая каталоговая запись). Как можно заключить по названию, MARC-записи представляют формат, удобный для обработки компьютером. В отличие от форматов JSON и XML, они *не предназначены* для чтения человеком. Если вы откроете файл, содержащий данные в формате MARC, то содержимое будет выглядеть примерно как на рис. 26.2.

```
01292cam  2200337
4500001000300000003000600003005001700009008004100026010001600067019001
2000830290021000950350029001160400030001450430012001750490009001870500
0170019608200100021310000460022324502590026926000590052830000320058744
0007500619504003200694650005100726650003600777650003300813700003800846
700004200884994001200926945001600938 2 OCoLC20060313170419.0690410s1963
laua     b    000 0 eng    a   63022268  a97725971 aNLGGCb861755170
a(OCoLC)2z(OCoLC)9772597  aDLCcDLCdOCLCQdTSEdOCL  an-us-la
aOCLC00aGB475.L6bM6 4a589.31 aMorgan, James P.q(James
Plummer),d1919-10aMudlumps at the mouth of South Pass, Mississippi
River;bsedimentology, paleontology, structure, origin, and relation to
deltaic processes,cby James P. Morgan, James M. Coleman [and] Sherwood
M. Gagliano. Including appendices by R.D. Adams ... [et al.].  aBaton
Rouge,bLouisiana State University Press,c1963.  axvi, 190 p.billus.c28
cm. 0aLouisiana State University studies.pCoastal studies series ;vno.
10.  aBibliography: p. [183]-190. 0aMud lumpszLouisianazMississippi
River Delta. 0aSediments (Geology)zLouisiana. 7aSciencesxPhilosophie.
2ram.1 aColeman, James M.,ejoint author.1 aGagliano, Sherwood
M.,ejoint author.  a02bOCL  aGB475.L6 M6
```

Рис. 26.2: Непосредственный вид содержимого MARC-записи

Если вам доводилось работать с форматом тегов ID3 для хранения информации о MP3-файлах, MARC-записи могут показаться знакомыми.

26.2.1. Структура MARC-записи

Стандарт MARC-записей был разработан в 1960-х годах с целью максимально эффективного хранения и передачи информации. В связи с этим

MARC-записи являются гораздо менее гибкими и расширяемыми, чем такие форматы, как XML или JSON. MARC-запись состоит из трёх основных частей:

- начальная часть;
- каталог;
- основная запись.

На рис. 26.3 представлена аннотированная версия MARC-записи, чтобы вам было легче представить её структуру.

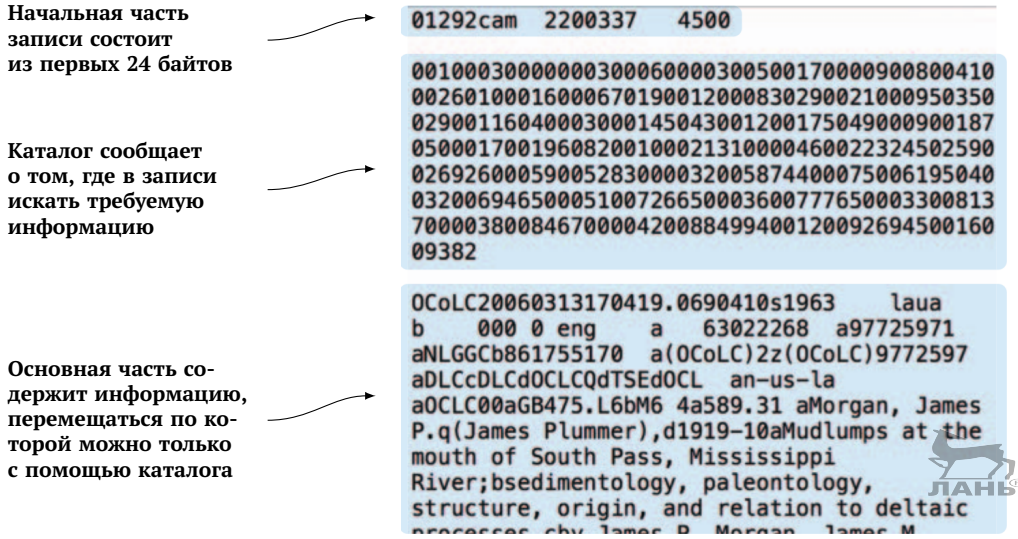


Рис. 26.3: Аннотированная версия MARC-записи

Начальная часть содержит информацию о самой записи, например её длину и место расположения основной записи. *Каталог* записи сообщает, какая информация содержится в записи и как получить доступ к ней. Например, вам нужны только автор и название книги. Из каталога вы узнаете о том, что запись содержит эту информацию, и о том, где её искать. Наконец, *основная запись* — это место, где находится вся информация, хранимая в записи. Без информации, содержащейся в начальной части и каталоге, вы не смогли бы разобраться со строением этой части файла.

26.2.2. Получение данных

Первое, что вам нужно сделать, — получить несколько MARC-записей для работы с ними. К счастью, archive.org содержит отличную коллекцию

доступных бесплатно MARC-записей. В этом проекте вы будете использовать коллекцию записей библиотеки Орегонского университета здоровья и науки. Перейдите на следующую страницу сайта archive.org: `https://archive.org/download/marc_oregon_summit_records/catalog_files/`. Скачайте файл ohsu_ncnm_wscc_bibs.mrc и переименуйте его в sample.mrc. Этот файл имеет размер 156 Мбайт и является самым маленьким из коллекции, но если вам интересно, вы можете поэкспериментировать с другими файлами оттуда же — они должны быть устроены аналогично.

26.2.3. Проверка начальной части и проход по записям

Файл с расширением .mrc не является единственной MARC-записью, а представляет собой коллекцию таких записей. Перед тем как начать думать о деталях отдельных записей, вам нужно разобраться с отделением записей друг от друга. В отличие от многих других форматов хранения сериализованных данных, в .mrc-файлах отсутствуют специальные разделители. Поэтому вы не можете просто разбить ваш поток `ByteString` по определённому символу, чтобы разделить записи. Вместо этого нужно просмотреть начальную часть каждой записи и узнать длину соответствующей записи. Зная длину, можно пройти по списку, попутно разбирая отдельные записи.

Для начала объявим синонимы типов `MarcRecordRaw` и `MarcLeaderRaw`:

Листинг 26.9 Синонимы типов MarcRecordRaw и MarcLeaderRaw

```
type MarcRecordRaw = B.ByteString
type MarcLeaderRaw = B.ByteString
```

Так как в основном вы работаете с байтами, все ваши типы для работы с MARC-записями будут синонимами типа `ByteString`. Но использование синонимов типов значительно повысит читаемость кода и облегчит понимание типовых аннотаций. Первое, что вам нужно сделать, — получить начальную часть записи:

```
getLeader :: MarcRecordRaw -> MarcLeaderRaw
```

Начальная часть записи состоит из первых 24-х байтов (рис. 26.4).

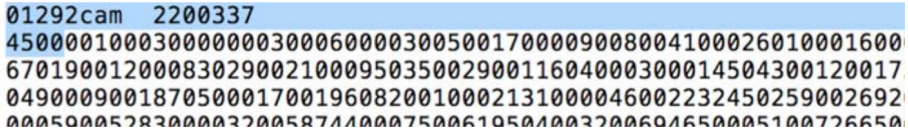


Рис. 26.4: Начальная часть записи выделена

Объявим переменную для хранения длины начальной части:

Листинг 26.10 Длина начальной части

```
leaderLength :: Int
leaderLength = 24
```

Получение начальной части записи выполняется довольно очевидным способом — нужно взять первые 24 байта.

ЛАНЬ

Листинг 26.11 Функция getLeader получает первые 24 байта записи

```
getLeader :: MarcRecordRaw -> MarcLeaderRaw
getLeader record = B.take leaderLength record
```

В то время как первые 24 байта MARC-записи представляют собой начальную часть MARC-записи, первые 5 байт начальной части представляют собой число, обозначающее длину записи. Например, на рис. 26.4 вы можете увидеть, что запись начинается с 01292, что означает, что длина записи составляет 1292 байта. Чтобы получить длину записи, вам нужно взять первые 5 символов и конвертировать их к типу `Int`. Для этого можно создать вспомогательную функцию под названием `rawToInt`, которая будет безопасно преобразовывать `ByteString` в `Text`, затем преобразовывать `Text` в `String`, и, наконец, использовать `read` для получения значения типа `Int`.

Листинг 26.12 Функции rawToInt и getRecordLength

```
rawToInt :: B.ByteString -> Int
rawtoInt = (read . T.unpack . E.decodeUtf8)

getRecordLength :: MarcLeaderRaw -> Int
getRecordLength leader = rawToInt (B.take 5 leader)
```

Теперь, когда у вас есть средство для вычисления длины одной записи, вы можете подумать о разделении записей на список элементов типа `MarcRecordRaw`. Вы будете рассматривать содержимое своего файла как `ByteString`. Вам понадобится функция, принимающая `ByteString` и разделяющая её на две части: первая запись и остаток `ByteString`. Эта функция будет называться `nextAndRest` и иметь следующий тип:

```
nextAndRest :: B.ByteString -> (MarcRecordRaw, B.ByteString)
```

Вы можете думать о получении этой пары значений как о разделении списка на первый элемент и «хвост». Чтобы получить эту пару, вам понадобится вычислить длину первой записи и затем отделить её.

Листинг 26.13 Функция nextAndRest отделяет первую запись потока

```
nextAndRest :: B.ByteString -> (MarcRecordRaw, B.ByteString)
nextAndRest marcStream = B.splitAt recordLength marcStream
  where recordLength = getRecordLength marcStream
```

Чтобы пройти через весь файл, вам нужно рекурсивно вызывать эту функцию для получения записи и оставшейся части файла. Затем вы будете помещать извлечённую запись в список и повторять эти действия до достижения конца файла.

Листинг 26.14 Преобразование потока в список записей

```
allRecords :: B.ByteString -> [MarcRecordRaw]
allRecords marcStream = if marcStream == B.empty
                        then []
                        else next : allRecords rest
  where (next, rest) = nextAndRest marcStream
```

Давайте проверим работу функции `allRecords`, переписав `main` на чтение файла sample.mrc и вывод на экран количества записей в нём:

```
main :: IO ()
main = do
  marcData <- B.readFile "sample.mrc"
  let marcRecords = allRecords marcData
  print (length marcRecords)
```

Вы можете запустить `main`, скомпилировав свою программу или загрузив её в GHCi и вызвав `main`:

```
GHCi> main
140328
```

В вашей коллекции содержится 140 328 записей! Теперь, когда вы разделили все записи, можно перейти к изучению того, как извлекать из них данные `Title` и `Author`.

26.2.4. Считывание каталога

MARC-записи хранят всю информацию о книге в *полях*. Каждое поле имеет тег и подполя, сообщающие вам информацию о книге (автор, название, предмет, дата публикации). Перед тем как начать думать об обработке полей, вам нужно найти всю информацию об этих полях в ка-

талоге. Как и всё остальное в MARC-записях, каталог представляет собой `ByteString`, но для повышения читаемости вы можете объявить ещё один синоним типа:

Листинг 26.15 Синоним типа MarcDirectoryRaw

```
type MarcDirectoryRaw = B.ByteString
```

В отличие от начальной части, длина которой всегда равняется 24 байтам, каталог может иметь переменный размер. Это связано с тем, что записи содержат разное количество полей. Как вы уже знаете, каталог начинается прямо за начальной частью, но вам ещё нужно выяснить, где он заканчивается. К сожалению, начальная часть не содержит непосредственно эту информацию. Вместо этого она содержит базовый адрес, то есть адрес начала основной записи. Каталог находится между концом начальной части и началом основной записи.

Информация о базовом адресе располагается в начальной части начиная с 12-го символа и заканчивается 16-м (включительно, и таким образом составляет 5 байтов) при нумерации байтов с 0. Чтобы получить доступ к ней, вам нужно взять начальную часть, отбросить первые 12 символов и взять 5 первых символов из оставшихся 12. Затем вам нужно преобразовать это значение из `ByteString` в `Int`, как вы делали с `recordLength`.

Листинг 26.16 Получение базового адреса

```
getBaseAddress :: MarcLeaderRaw -> Int
getBaseAddress leader = rawToInt (B.take 5 remainder)
  where remainder = B.drop 12 leader
```

Потом, чтобы вычислить длину каталога, вам нужно вычесть значение `(leaderLength + 1)` из базового адреса, таким образом получив разность этих значений.

Листинг 26.17 Вычисление длины каталога

```
getDirectoryLength :: MarcLeaderRaw -> Int
getDirectoryLength leader =
    getBaseAddress leader - (leaderLength + 1)
```

Теперь вы можете собрать эти части вместе для получения каталога. Для начала вам нужно найти длину каталога в записи, а затем отбросить начальную часть и взять нужное количество байтов из оставшихся.

Листинг 26.18 Совмещаем всё в функции getDirectory

```
getDirectory :: MarcRecordRaw -> MarcDirectoryRaw
getDirectory record = B.take directoryLength afterLeader
  where directoryLength = getDirectoryLength record
        afterLeader = B.drop leaderLength record
```

К данному моменту вы уже проделали длинный путь в понимании этого таинственного формата хранения данных. Теперь вам нужно разобраться с содержимым каталога.

26.2.5. Использование каталога для просмотра полей

На данный момент каталог представляет собой большую `ByteString`, с которой вам нужно разобраться. Как было отмечено ранее, каталог позволяет вам просматривать поля в основной записи. Он также содержит информацию о расположении полей. К счастью, размер каждой каталоговой записи с данными о полях составляет ровно 12 байт.

Листинг 26.19 Тип для каталоговой записи и её длина

```
type MarcDirectoryEntryRaw = B.ByteString

dirEntryLength :: Int
dirEntryLength = 12
```

Затем вам нужно разбить информацию, содержащуюся в каталоге, на список элементов типа `MarcDirectoryEntryRaw`. Типовая аннотация соответствующей функции выглядит следующим образом:

```
splitDirectory :: MarcDirectoryRaw -> [MarcDirectoryEntryRaw]
```

Реализация функции довольно прямолинейна: вы берёте фрагмент из 12 байт и добавляете его к списку, пока есть данные.

Листинг 26.20 splitDirectory разбивает каталог на список записей

```
splitDirectory directory =
    if directory == B.empty
    then []
    else nextEntry : splitDirectory restEntries
  where (nextEntry, restEntries) = B.splitAt dirEntryLength
                                             directory
```

Теперь, когда вы смогли получить список необработанных значений типа `MarcDirectoryEntryRaw`, вы близки к получению данных об авторах и названиях книг.

26.2.6. Обработка данных из каталога и поиск полей MARC-записей

Каждая запись в каталоге похожа на миниатюрную версию начальной части MARC-записи. Метаданные для каждой записи содержат следующую информацию:

- тег поля (первые три символа);
- длина поля (следующие четыре символа);
- местоположение начала поля относительно базового адреса (оставшиеся символы).

Так как вам потребуется использовать всю эту информацию, то понадобится тип данных `FieldMetadata`.

Листинг 26.21 Тип FieldMetadata

```
data FieldMetadata =
     FieldMetadata { tag :: T.Text
                   , fieldLength :: Int
                   , fieldStart :: Int } deriving Show
```

Затем нужно преобразовать список `MarcDirectoryEntryRaw` в список `FieldMetadata`. Как часто бывает при работе со списками, проще начать с преобразования одного элемента типа `MarcDirectoryEntryRaw` в значение типа `FieldMetadata`.

Листинг 26.22 Получение каталоговой записи типа FieldMetadata

```
makeFieldMetadata :: MarcDirectoryEntryRaw -> FieldMetadata
makeFieldMetadata entry =
    FieldMetadata textTag theLength theStart
  where (theTag, rest) = B.splitAt 3 entry
        textTag = E.decodeUtf8 theTag
        (rawLength, rawStart) = B.splitAt 4 rest
        theLength = rawToInt rawLength
        theStart = rawToInt rawStart
```

Теперь можно преобразовать имеющийся у вас список с помощью `map`.

Листинг 26.23 Построение списка каталоговых записей

```
getFieldMetadata :: [MarcDirectoryEntryRaw] -> [FieldMetadata]
getFieldMetadata rawEntries = map makeFieldMetadata rawEntries
```

С помощью `getFieldMetadata` вы можете написать функцию, позволяющую вам находить само поле. Теперь, когда вы занимаетесь полями,

вам нужно перестать думать о байтах и начать думать о тексте. Ваши поля будут содержать информацию об авторе и названии, а также другие текстовые данные. Вам понадобится ещё один синоним типа под названием `FieldText`.

Листинг 26.24 Синоним типа FieldText

```
type FieldText = T.Text
```

Теперь нужно реализовать возможность по заданным `MarcRecordRaw` и `FieldMetadata` получать `FieldText`, тогда можно будет начать искать нужные значения.

Чтобы это сделать, вам сначала нужно отбросить из `MarcRecordRaw` начальную часть записи и каталог, таким образом оставшись с основной записью. Затем вам нужно отбросить `fieldStart` из записи и, наконец, взять `fieldLength` от оставшейся части.

Листинг 26.25 Получение FieldText

```
getTextField :: MarcRecordRaw -> FieldMetadata -> FieldText
getTextField record fieldMetadata =
    E.decodeUtf8 byteStringValue
  where recordLength = getRecordLength record
        baseAddress = getBaseAddress record
        baseRecord = B.drop baseAddress record
        baseAtEntry = B.drop (fieldStart fieldMetadata)
                             baseRecord
        byteStringValue = B.take (fieldLength fieldMetadata)
                                 baseAtEntry
```

Вы значительно продвинулись в понимании формата MARC-записей. Остался единственный шаг — обработка `FieldText` с целью получения нужных данных.

26.2.7. Получение автора и названия книги из MARC-поля

В MARC-записях с каждым тегом ассоциировано специальное значение. Например, тегу `Title` соответствует число `245`. К сожалению, трудности на этом не заканчиваются. Каждое поле состоит из подполей, которые отделяются друг от друга с помощью разделителя — ASCII-символа под номером 31. Для получения этого символа вы можете воспользоваться функцией `toEnum`:

Листинг 26.26 Получение разделителя подполей

```
fieldDelimeter :: Char
fieldDelimeter = toEnum 31
```

Для разбиения `FieldText` на подполя можно использовать `T.split`. Каждое подполе содержит значение, например название или автора. Перед каждым значением находится код подполя, представленный единственным символом, как показано на рис. 26.5.

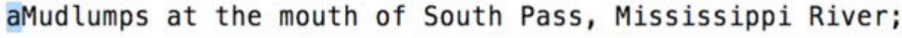


Рис. 26.5: Пример подполя с названием книги (символ `'a'` — код подполя)

Чтобы получить название книги, вам потребуется подполе `'a'` поля `245`. Автор находится в подполе `'a'` поля `100`.

Листинг 26.27 Теги и коды подполей для названия и автора

```
titleTag :: T.Text
titleTag = "245"

titleSubfield :: Char
titleSubfield = 'a'

authorTag :: T.Text
authorTag = "100"

authorSubfield :: Char
authorSubfield = 'a'
```

Чтобы получить значение поля, вам нужно найти его местоположение в записи с помощью `FieldMetadata`. Затем нужно разбить поле на подполя. Наконец, вы сможете посмотреть на первый символ каждого подполя и определить, является ли оно тем, что вам нужно.

Возникает новая проблема: вы не знаете наверняка, будет ли в записи нужное вам поле, а также вы не знаете, будет ли в поле (в случае его наличия) нужное вам подполе. Для выполнения этих двух проверок вам понадобится тип `Maybe`. Для начала реализуйте функцию `lookupFieldMetadata`, проверяющую каталог на наличие нужной вам `FieldMetadata`. Если поля нет в записи, функция будет возвращать `Nothing`; в противном случае она вернёт нужное поле.

Листинг 26.28 Безопасный поиск FieldMetadata в каталоге

```
lookupFieldMetadata aTag record = if length results < 1
                                  then Nothing
                                  else Just (head results)
  where metadata =
          (getFieldMetadata . splitDirectory . getDirectory)
          record
        results = filter ((== aTag) . tag) metadata
```

Так как вам необходимо будет находить и поле, и подполе, в функцию, выполняющую поиск подполя, вы будете передавать значение типа `Maybe FieldMetadata`. Функция под названием `lookupSubfield` будет принимать аргумент типа `Maybe FieldMetadata`, символ (`Char`), соответствующий подполю, и `MarcRecordRaw` и возвращать `Maybe BC.ByteString` с данными, хранящимися в подполе.

Листинг 26.29 Поиск потенциально отсутствующего подполя

Если метаданные не найдены, вы не сможете найти подполе

Если подполе не найдено в результате поиска, нужного подполя нет

```
lookupSubfield :: Maybe FieldMetadata -> Char ->
                  MarcRecordRaw -> Maybe T.Text
lookupSubfield Nothing subfield record = Nothing
lookupSubfield (Just fieldMetadata) subfield record =
    if results == []
    then Nothing
    else Just ((T.drop 1 . head) results)
  where rawField = getTextField record fieldMetadata
        subFields = T.split (== fieldDelimeter) rawField
        results = filter ((== subfield) . T.head) subFields
```

Пустой результат поиска означает, что возвращать нечего

Иначе вам нужно преобразовать значение подполя в `Text` и отбросить первый символ, соответствующий коду подполя

Всё, что вам нужно, — значение, соответствующее определённой комбинации поля и подполя. Теперь вам понадобится функция `lookupValue`, принимающая тег, код подполя и запись.

Листинг 26.30 Поиск значений по тегу и коду подполя

```
lookupValue :: T.Text -> Char -> MarcRecordRaw -> Maybe T.Text
lookupValue aTag subfield record =
    lookupSubfield entryMetadata subfield record
  where entryMetadata = lookupFieldMetadata aTag record
```

Вы можете создать две вспомогательные функции для получения нужных значений под названиями `lookupAuthor` и `lookupTitle` с помощью частичного применения функции `lookupValue`:

Листинг 26.31 Специальные случаи для Title и Author

```
lookupTitle :: MarcRecordRaw -> Maybe Title
lookupTitle = lookupValue titleTag titleSubfield

lookupAuthor :: MarcRecordRaw -> Maybe Author
lookupAuthor = lookupValue authorTag authorSubfield
```

К данному моменту вы полностью абстрагировались от деталей работы с MARC-записями и теперь можете написать итоговую версию `main`, использующую весь написанный ранее код.

26.3. Собираем всё вместе

Вы справились с испытанием, состоящим в написании парсера MARC-записей, и теперь у вас есть доступ к большому количеству информации о книгах. Помните о том, что вам нужно минимизировать количество действий ввода-вывода, а также максимально сократить количество действий для преобразования `ByteString` (представляющей MARC-файл) в HTML (представляющий ваш выходной файл). Первым шагом будет преобразование `ByteString` в список пар (`Maybe Title, Maybe Author`).

Листинг 26.32 Получение названий и авторов из MARC-записей

```
marcToPairs :: B.ByteString -> [(Maybe Title, Maybe Author)]
marcToPairs marcStream = zip titles authors
  where records = allRecords marcStream
        titles = map lookupTitle records
        authors = map lookupAuthor records
```

Затем вам нужно преобразовать эти `Maybe`-пары в список книг. Вы будете создавать книгу только для тех пар, у которых в наличии оба значения `Author` и `Title` (то есть оба значения с конструкторами `Just`). В качестве вспомогательной функции вы будете использовать функцию `fromJust` из модуля `Data.Maybe`.

Листинг 26.33 Преобразование значений Maybe в Book

```
pairsToBooks :: [(Maybe Title, Maybe Author)] -> [Book]
pairsToBooks pairs =
   map (\(title, author) -> Book {
                              title = fromJust title
                            , author = fromJust author
                            })
       justPairs
  where
    justPairs =
      filter (\(title, author) -> isJust title
                                & isJust author)
             pairs
```

У вас уже есть функция `booksToHtml`, так что вы можете скомбинировать все эти функции для получения финальной функции `processRecords`. В связи с тем, что записей в ваших файлах довольно много, вы также будете передавать в качестве параметра количество записей, которые хотите просмотреть.

Листинг 26.34 Совмещаем всё в функции processRecords

```
processRecords :: Int -> B.ByteString -> Html
processRecords n = booksToHtml
                 . pairsToBooks
                 . take n
                 . marcToPairs
```

Несмотря на то что это был урок про ввод-вывод, да и сама задача связана с довольно-таки интенсивным вводом-выводом, вас, возможно, удивит относительно небольшой объём кода в финальной версии IO-действия `main`:

```
main :: IO ()
main = do
   marcData <- B.readFile "sample.mrc"
   let processed = processRecords 500 marcData
   TIO.writeFile "books.html" processed
```

Вы успешно преобразовали MARC-записи в гораздо более читаемый формат. Обратите внимание: символы Юникода работают нормально!

Благодаря `lookupValue` у вас также появилось удобное обобщённое средство для поиска заданных тегов и подполей в MARC-записях.

Итоги

В этом итоговом проекте вы:

- смоделировали текстовые данные о книге с помощью типа `Text`;
- реализовали набор средств для преобразования двоичных данных, используя для работы с байтами тип `ByteString`;
- безопасно разобрались с текстом в кодировке Юникод из двоичного документа с помощью функций `decodeUtf8` и `encodeUtf8`;
- успешно преобразовали запутанный бинарный формат данных в читаемый HTML.

Расширение проекта

Теперь, когда вы умеете обрабатывать MARC-записи, перед вами открывается целый мир интересных для изучения данных. Если вы хотите расширить свой итоговый проект, попробуйте научиться узнавать больше информации о книге посредством обработки MARC-записи. Например, вы могли заметить, что после названия книги иногда встречаются некоторые знаки пунктуации. Это связано с тем, что подполе `b` содержит оставшуюся часть расширенного названия. Комбинирование подполей `a` и `b` позволит вам получить полное название. Библиотека конгресса США предоставляет большой объём информации о MARC-записях, которую вы можете изучить на сайте `www.loc.gov/marc/bibliographic/`.

Ещё одной задачей, с которой мы не разобрались, является работа с крайне неприятными символами не в кодировке Юникод, содержащимися в MARC-записях формата MARC-8. В MARC-8 небольшое подмножество символов Юникода по историческим причинам представлено нестандартным образом. Библиотека конгресса США располагает материалами, которые помогут вам с этим разобраться: `www.loc.gov/marc/specifications/speccharconversion.html`. Относится ли содержимое записи к MARC-8 или обычному Юникоду, определяется по начальной части записи. Почитать об этом можно в разделе «Character Coding Scheme» официальной документации Библиотеки конгресса: `www.loc.gov/marc/bibliographic/bdleader.html`.

Модуль 5

Работа с типами в контексте

В этом модуле вы рассмотрите три самых мощных и в то же время вызывающих наибольшее количество затруднений у изучающих Haskell класса типов — `Functor`, `Applicative` и `Monad`. Эти классы типов имеют забавные имена, но относительно простое предназначение. Каждый из них строится на основании предыдущего и позволяет вам работать в контекстах, таких как `IO`. В этом модуле вы получите гораздо более глубокое понимание того, как устроены контексты. Чтобы лучше понять, как работают абстрактные классы типов, вы будете изучать типы по аналогии с геометрическими фигурами.

Рис. 1: Круг и квадрат визуально представляют два типа

Один из способов интерпретации функции — преобразование одного типа в другой. Давайте визуально представим два типа с помощью двух геометрических фигур — квадрата и круга (см. рис. 1).

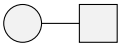


Рис. 2: Функция преобразует круг к квадрату

Эти фигуры могут представлять любые два типа данных: `Int` и `Double`, `String` и `Text`, `Name` и `FirstName` и т. д. Для преобразования круга в квадрат вы пользуетесь функцией. Вы можете визуально представить функцию как приспособление, соединяющее две фигуры, как показано на рис. 2.

Соединительная линия может представлять собой любую функцию из одного типа в другой. Фигура в целом может представлять функцию типов `(Int -> Double)`, `(String -> Text)`, `(Name -> FirstName)` и т. д. Когда вам нужно применить преобразование, вы можете представить себе вставку «коннектора» (соединительной части) между исходной фигурой

(в данном случае кругом) и желаемой фигурой (в данном случае квадратом); см. рис. 3.

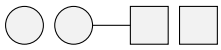


Рис. 3: Визуализация функции как способа соединения двух фигур

Если фигуры действительно имеют подходящую форму, вы можете выполнить желаемое преобразование.

В этом модуле вы рассмотрите работу с типами в контексте. Лучшими виденными вами к этому моменту примерами типов в контексте являются типы `Maybe` и `IO`. Типы `Maybe` представляют собой контекст, в котором значение может отсутствовать, а типы `IO` — контекст, в котором значение тем или иным образом связано с операциями ввода-вывода. Продолжая наши визуальные аналогии, вы можете представлять себе типы в контексте так, как показано на рис. 4.

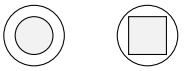


Рис. 4: Внешняя фигура обозначает контекст, например `Maybe` или `IO`

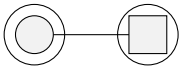


Рис. 5: Преобразование типов в контекстах

Эти фигуры в фигурах представляют собой такие типы, как `IO Int` и `IO Double`, `Maybe String` и `Maybe Text`, `Maybe Name` и `Maybe FirstName`. Так как эти типы находятся в контексте, вы не можете использовать свой старый коннектор для выполнения преобразования. Вам уже доводилось пользоваться функциями, входные и выходные значения которых находятся в контексте. Чтобы выполнить преобразование типов в контексте, вам нужен коннектор, показанный на рис. 5.

Этот коннектор представляет функции с такими типовыми аннотациями, как `(Maybe Int -> Maybe Double)`, `(IO String -> IO Text)` или, возможно, `(IO Name -> IO FirstName)`. С помощью данного коннектора вы можете преобразовывать типы в контексте так, как показано на рис. 6.

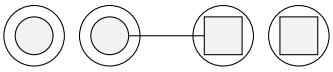


Рис. 6: Для выполнения преобразования нужен подходящий коннектор

Это решение может казаться идеальным, но у него есть проблема. Давайте рассмотрим функцию `halve`, имеющую тип `Int -> Double` и делящую на два свой целочисленный аргумент.

Листинг 1 Функция halve типа Int -> Double

```
halve :: Int -> Double
halve n = fromIntegral n / 2.0
```

Реализация функции довольно проста. Но представьте, что функция должна принимать значение типа `Maybe Int`. Пользуясь средствами, которые у вас уже есть, вы должны написать «обёртку», позволяющую вам работать с типами `Maybe`.

ЛАНЬ

Листинг 2 Функция halveMaybe — обёртка halve для работы с Maybe

```
halveMaybe :: Maybe Int -> Maybe Double
halveMaybe (Just n) = Just (halve n)
halveMaybe Nothing = Nothing
```

ЛАНЬ

В этом случае написание обёртки не является трудной задачей. Но если представить себе множество существующих функций с типами `a -> b`, то чтобы начать использовать их с типами `Maybe`, придётся написать множество практически идентичных обёрток! Хуже того: в случае с типом `IO` вы вообще не сможете реализовать такую обёртку!

В этот момент на сцену выходят `Functor`, `Applicative` и `Monad`. Вы можете думать об этих классах типов как об адаптерах, позволяющих вам работать с различными коннекторами при условии совпадения соединяемых ими типов. В случае с функцией `halve` вам нужно было изменить свой `(Int -> Double)`-коннектор для работы с типами в контексте. Именно это и делает класс типов `Functor`, проиллюстрированный на рис. 7.

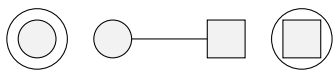


Рис. 7: `Functor` исправляет несоответствие типов в контексте и коннектора

Но вы также можете столкнуться с тремя другими случаями несоответствия типов в контексте и коннекторов. Класс типов `Applicative` помогает справиться с двумя из них. Первый случай возникает, когда одна из частей, соединяемых коннектором, находится в контексте, а вторая — нет, как показано на рис. 8.

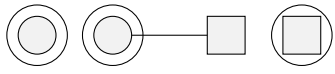


Рис. 8: Одно из несовпадений, разрешаемых с помощью `Applicative`

Второй случай несоответствия возникает, когда сама функция находится в контексте. Например, функция имеет тип `Maybe (Int -> Double)`, то есть сама функция может отсутствовать. Сейчас это может казаться странным, но такое вполне может произойти при частичном применении типов `Maybe` или `IO`. Интересный случай проиллюстрирован на рис. 9.

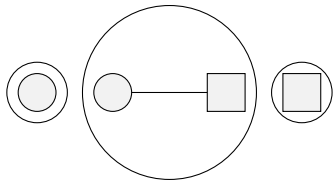


Рис. 9: Коннектор оказывается в контексте — работа для `Applicative`

Осталось всего одно возможное несоответствие между функцией и типами в контексте. Оно возникает, когда аргумент функции (в отличие от возвращаемого значения) не находится в контексте. Такая ситуация встречается чаще, чем может показаться на первый взгляд. Например, типы подобного вида имеют функции `Map.lookup` и `putStrLn`. Эту проблему решает класс типов `Monad`, показанный на рис. 10.

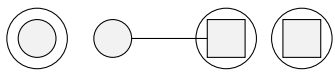


Рис. 10: `Monad` как адаптер для последнего из возможных несоответствий

Когда вы начинаете использовать эти три класса типов, больше не остаётся функций, которые не могут быть использованы в контексте, таком как `Maybe` или `IO`, если типы, находящиеся в контекстах, соответствуют типам, преобразование между которыми выполняет функция. Это важно, потому что вы получаете возможность выполнять любые нужные вам вычисления в контексте и средства для переиспользования больших фрагментов существующего кода в различных контекстах.

27

Класс типов Functor

После прочтения урока 27 вы:

- научитесь использовать класс типов `Functor`;
- сможете решать задачи при помощи `fmap` и `<$>`;
- разберётесь с видами типов для `Functor`.

К этому моменту вы уже видели несколько параметризованных типов (типов, которые принимают другой тип в качестве аргумента). Вы взглянули на типы, которые представляют контейнеры, вроде `List` и `Map`. Вы также видели параметризованные типы, которые представляют контекст, вроде `Maybe` для отсутствующих значений и `IO` для значений, которые приходят из сложного мира ввода-вывода. В этом уроке вы исследуете мощный класс типов `Functor`. Класс типов `Functor` предоставляет обобщённый интерфейс для применения функций к значениям в контейнере или контексте. Чтобы это прочувствовать, допустим, что у вас есть следующие типы:

- `[Int]`;
- `Map String Int`;
- `Maybe Int`;
- `IO Int`.

Это четыре разных типа, но все они параметризованы один и тем же типом — `Int` (`Map` — это особый случай, но значения в нём принадлежат к типу `Int`). Теперь предположим, что у вас есть функция следующего типа:

```
Int -> String
```

Эта функция принимает Int и возвращает String. В большинстве языков программирования вам было бы нужно написать отдельную версию такой функции Int -> String для каждого из этих параметризованных типов. Благодаря классу типов Functor у вас есть единый способ применять функцию ко всем этим случаям.

Обратите внимание. У вас есть потенциально отсутствующее значение Int (то есть Maybe Int). Вы хотите возвести его в квадрат, перевести в строку, а затем добавить в конец '!'. Функция printInt, в которую вы хотите передавать это значение, подразумевает, что там уже могут быть отсутствующие значения:

```
printInt :: Maybe String -> IO ()
printInt Nothing = putStrLn "Значение отсутствует"
printInt (Just val) = putStrLn val
```

Как вы могли бы перевести ваш тип Maybe Int в Maybe String, чтобы использовать printInt?

27.1. Пример: вычисление с Maybe

Тип Maybe уже доказал, что является полезным решением проблемы с потенциально отсутствующими значениями. Но когда вы познакомились с Maybe в уроке 19, вам всё равно пришлось решать проблему обработки возможно отсутствующего значения, как только вы сталкивались с ним в своей программе. Оказывается, вы всё-таки можете выполнять вычисления с потенциально отсутствующим значением без необходимости переживать, действительно ли оно отсутствует.

Допустим, у вас есть число из базы данных. Существует множество причин, почему запрос к базе данных может вернуть пустое значение. Вот два простых значения типа Maybe Int для примера: failedRequest и successfulRequest.

Листинг 27.1 Примеры потенциально пустых значений

```
successfulRequest :: Maybe Int
successfulRequest = Just 6

failedRequest :: Maybe Int
failedRequest = Nothing
```

Далее представьте, что вам нужно инкрементировать число, которое вы получили из базы данных, и перезаписать его обратно. Давайте предположим, что логика общения с базой данных обрабатывает случай с null-значениями, просто не записывая их в базу. В идеале вы хотели бы хранить ваши значения в `Maybe`. Пользуясь имеющимися знаниями, вы могли бы написать специальную функцию `incMaybe`, обработав эту ситуацию.

Листинг 27.2 Функция, инкрементирующая значение Maybe Int

```
incMaybe :: Maybe Int -> Maybe Int
incMaybe (Just n) = Just (n + 1)
incMaybe Nothing = Nothing
```

В GHCi это работает отлично:

```
GHCi> incMaybe successfulRequest
Just 7
GHCi> incMaybe failedRequest
Nothing
```

Проблема в том, что это решение ужасно плохо масштабируется. Функция инкремента — это просто `(+ 1)`, но в нашем примере вам нужно было переписать её для `Maybe`. Это решение означает, что вам нужно переписать специальную версию каждой существующей функции, которую вы хотите использовать с `Maybe`! Это сильно ограничивает полезность инструментов вроде `Maybe`. Оказывается, в Haskell есть класс типов под названием `Functor`, который решает данную проблему.

Проверка 27.1. Напишите функцию `reverseMaybe`, которая инвертирует строку в заданном значении `Maybe String` и возвращает результат как `Maybe String`, вот её типовая аннотация:

```
reverseMaybe :: Maybe String -> Maybe String
```

Ответ 27.1

```
reverseMaybe :: Maybe String -> Maybe String
reverseMaybe Nothing = Nothing
reverseMaybe (Just string) = Just (reverse string)
```

27.2. Класс типов Functor и вызов функций в контексте

Haskell предлагает замечательное решение этой проблемы. `Maybe` реализует экземпляр класса типов `Functor`. Класс типов `Functor` требует только одного определения метода `fmap`, как показано на рис. 27.1.

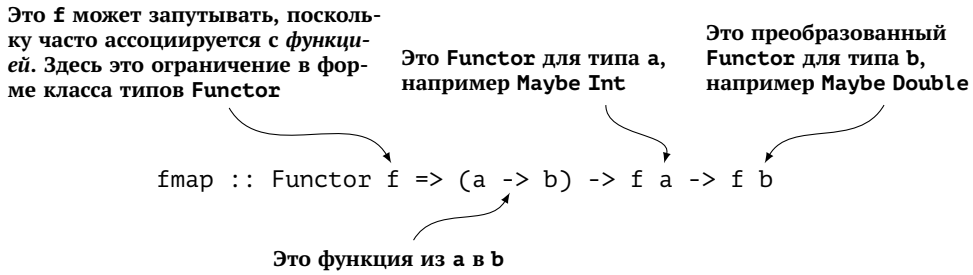


Рис. 27.1: Типовая аннотация функции `fmap`

Возвращаясь к визуальному языку из введения к этому модулю, представим `fmap` как адаптер, проиллюстрированный на рис. 27.2. Обратите внимание, что мы используем <$>, являющийся синонимом `fmap` (правда, это скорее бинарная операция, чем функция).

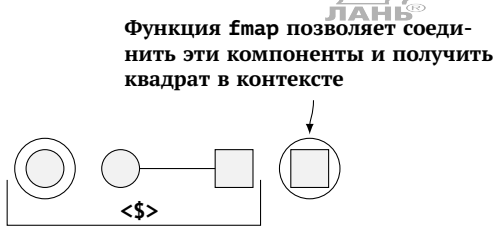


Рис. 27.2: Функция `fmap` (или <$>) как адаптер и типы в контексте

Вы можете определить `fmap` как обобщение функции `incMaybe`.

Листинг 27.3 Делаем Maybe экземпляром Functor

```
instance Functor Maybe where
  fmap func (Just n) = Just (func n)
  fmap func Nothing = Nothing
```

С `fmap` вам больше не нужна специальная функция для инкрементирования значения в `Maybe`:

```
GHCi> fmap (+ 1) successfulRequest
Just 7
GHCi> fmap (+ 1) failedRequest
Nothing
```

Хотя `fmap` — это официальное название функции, на практике бинарная операция `<$>` используется гораздо чаще:

```
GHCi> (+ 1) <$> successfulRequest
Just 7
GHCi> (+ 1) <$> failedRequest
Nothing
```

В этом примере `(+ 1)` добавляет 1 к значению внутри `Maybe Int` и возвращает `Maybe Int`. Но важно помнить, что тип функции `fmap` — `(a -> b)`, а значит, возвращаемое `Maybe` не обязательно должно быть параметризовано тем же типом. Вот примеры перехода от `Maybe Int` к `Maybe String`.

Листинг 27.4 Пример использования fmap с изменением типа

```
successStr :: Maybe String
successStr = show <$> successfulRequest

failStr :: Maybe String
failStr = show <$> failedRequest
```

В возможности преобразовывать типы значений внутри `Maybe` проявляется истинная мощь класса типов `Functor`.

Проверка 27.2. Используйте `fmap` или `<$>`, чтобы инвертировать значение `Maybe String`.

Ответ 27.2

```
GHCi> reverse <$> Just "кот"
Just "ток"
```

Странные названия классов типов?

`Semigroup`, `Monoid`, а теперь `Functor`! Откуда взялись эти странные названиями классов типов? Все они пришли из областей математики под названиями общая алгебра и теория категорий. Вам абсолютно не нужно знать никакой продвинутой математики, для того чтобы ими пользоваться. Эти классы типов просто отражают паттерны проектирования, применяемые в функциональном программировании. Если вы уже использовали Java, C# или другой промышленный язык программирования, то наверняка знакомы с объектно-ориентированными паттернами проектирования вроде Наблюдателя, Одиночки или Фабрики. Эти названия звучат более приемлемо, потому что стали частью каждодневного лексикона ОО-программистов. И паттерны проектирования в ООП, и теоретико-категорные классы типов обобщают основные паттерны программирования. Единственное отличие в том, что в Haskell они базируются на математических основаниях, а не на тут и там повторяющихся, когда-то обнаруженных в коде фрагментах. В Haskell как функции черпают свою силу из своей математической базы, так же поступают и паттерны проектирования.

27.3. Функторы повсюду!

Чтобы разобраться с экземплярами `Functor`, вы рассмотрите некоторые примеры. Из урока 18 мы помним, что виды — это типы типов. Типы вида `* -> *` — это параметризованные типы, которые принимают один типовый параметр. Все экземпляры класса `Functor` должны иметь вид `* -> *`. Также получается, что многие параметризованные типы вида `* -> *` принадлежат к `Functor`.

Типы, имеющие экземпляр класса типов `Functor`, которые вы уже видели в этой книге, включают в себя `List`, `Map`, `Maybe` и `IO`. Чтобы продемонстрировать, как `Functor` позволяет вам обобщать свой код, решая одну и ту же задачу одним образом для нескольких параметризованных типов, вы исследуете, как работа с одними и теми же типами данных в разных контекстах может доставлять разные проблемы. Затем вы увидите, как `<$>` из `Functor` упрощает решение каждой из этих проблем единым образом. Вместо работы с простыми типами вроде `Int` и `String` вы будете работать кое с чем более сложным, а именно с типом данных `RobotPart`.

27.3.1. Один интерфейс для четырёх проблем

В этом примере вы будете подразумевать, что работаете в компании по производству компонентов роботов. Вот базовый тип данных, описывающий компонент робота.

Листинг 27.5 Тип данных RobotPart

```
data RobotPart = RobotPart
    { name :: String
    , description :: String
    , cost :: Double
    , count :: Int
    } deriving Show
```

Вот примеры компонентов, которые вы будете использовать в этом разделе.

Листинг 27.6 Примеры компонентов роботов

```
leftArm :: RobotPart
leftArm  = RobotPart
   { name = "left arm"
   , description = "левая рука для того, чтобы бить в лицо!"
   , cost = 1000.00
   , count = 3
   }

rightArm :: RobotPart
rightArm  = RobotPart
   { name = "right arm"
   , description = "правая рука для добрых жестов"
   , cost = 1025.00
   , count = 5
   }

robotHead :: RobotPart
robotHead  = RobotPart
   { name = "robot head"
   , description = "эта голова выглядит безумной"
   , cost = 5092.25
   , count = 2
   }
```

Одна из самых частых вещей, которую вам нужно будет делать, — это представление информации, содержащейся в `RobotPart`, в виде HTML.

Вот код для перевода одного компонента RobotPart во фрагмент HTML-документа.

Листинг 27.7 Перевод RobotPart в HTML

```
type Html = String

renderHtml :: RobotPart -> Html
renderHtml part = mconcat [ "<h2>",partName, "</h2>"
                          , "<p><h3>desc</h3>", partDesc
                          , "</p><p><h3>cost</h3>"
                          , partCost
                          , "</p><p><h3>count</h3>"
                          , partCount, "</p>"]
  where partName = name part
        partDesc = description part
        partCost = show (cost part)
        partCount = show (count part)
```

В дальнейшем мы будем преобразовывать компоненты робота, возникающие в разных контекстах, к виду в HTML. Начнём с использования Map для создания partsDB, которая будет нашей внутренней базой данных компонентов роботов.

Листинг 27.8 База данных компонентов роботов

```
import qualified Data.Map as Map

partsDB :: Map.Map Int RobotPart
partsDB = Map.fromList keyVals
  where keys = [1,2,3]
        vals = [leftArm,rightArm,robotHead]
        keyVals = zip keys vals
```

Не забывайте включать эту строку вверху файла, если собираетесь использовать Map

Map — полезный для этого примера тип, потому что он позволяет использовать сразу три экземпляра Functor: он создаётся из списка, возвращает значения Maybe и сам по себе реализует Functor.

27.3.2. Перевод Maybe RobotPart в Maybe Html

Теперь предположим, что у вас есть сайт, работающий с partsDB. Будет полезно иметь запрос, содержащий идентификатор компонента, который вы хотите вставить на веб-страницу. Будем подразумевать, что IO-действие insertSnippet будет принимать HTML и вставлять его в шаблон

страницы. Также разумно подразумевать, что многие модели данных могут генерировать HTML-фрагменты. Поскольку любая из этих моделей может сгенерировать ошибку, вы будете подразумевать, что `insertSnippet` будет принимать на вход `Maybe Html`, позволяя движку шаблонизатора обрабатывать недостающие фрагменты так, как он считает нужным. Вот типовая аннотация работающей подобным образом воображаемой функции:

```
insertSnippet :: Maybe Html -> IO ()
```

Задача, которую вам предстоит решить, — это поиск и передача компонента в виде `Maybe Html` в `insertSnippet`. Вот пример получения значения `RobotPart` из `partsDB`.

Листинг 27.9 partVal: значение Maybe RobotPart

```
partVal :: Maybe RobotPart
partVal = Map.lookup 1 partsDB
```

Так как `Maybe` принадлежит к `Functor`, вы можете воспользоваться `<$>` для преобразования `RobotPart` в HTML, оставаясь в `Maybe`.

Листинг 27.10 Преобразование RobotPart в HTML в рамках Maybe

```
partHtml :: Maybe Html
partHtml = renderHtml <$> partVal
```

Благодаря наличию `Functor` вы теперь можете передавать `partHtml` в функцию `insertSnippet`.

27.3.3. Перевод списка RobotPart в список HTML

Далее предположим, что вы хотите создать индексированную страницу со всеми компонентами. Вы можете получить список компонентов из `partsDB` следующим образом.

Листинг 27.11 Список компонентов роботов

```
allParts :: [RobotPart]
allParts = map snd (Map.toList partsDB)
```

Список также является экземпляром `Functor`. По сути, `fmap` для списка — это обычная функция `map`, которой вы пользуетесь с модуля 1. Вот как вы можете применить `renderHtml` к списку значений, используя `<$>`.

Листинг 27.12 Преобразование списка компонентов в HTML

```
allPartsHtml :: [Html]
allPartsHtml = renderHtml <$> allParts
```

Так как <$> — это просто fmap, а на списках fmap — это просто map, этот код идентичен следующему.

Листинг 27.13 Привычный способ преобразования с вызовом map

```
allPartsHtml :: [Html]
allPartsHtml = map renderHtml allParts
```

На списках чаще используется map, а не <$>, но важно понимать, что они идентичны. Один из вариантов думать о классе типов Functor — как о вещах, к которым можно применить что-то вроде map.

Проверка 27.3. Перепишите определение функции allParts, используя <$> вместо map.

27.3.4. Перевод Map с компонентами роботов в HTML

Ваш Map под названием partsDB был полезен, но выходит так, что он был вам нужен только для перевода компонентов роботов в HTML. Если это так, не имеет ли смысл иметь htmlPartsDB, чтобы вам не нужно было постоянно их конвертировать? Так как Map реализует экземпляр Functor, вы легко можете это сделать.

Листинг 27.14 Преобразование partsDB в Map, содержащий HTML

```
htmlPartsDB :: Map.Map Int Html
htmlPartsDB = renderHtml <$> partsDB
```

Теперь вы можете увидеть, что преобразовали Map с компонентами роботов в Map с HTML-фрагментами!

Ответ 27.3

```
allParts :: [RobotPart]
allParts = snd <$> Map.toList partsDB
```

```
GHCi> Map.lookup 1 htmlPartsDB
Just "<h2>left arm</h2><p><h3>desc</h3>left ...
```

Этот пример показывает, насколько мощным может быть простой интерфейс, который предоставляет Functor. Вы можете тривиально выполнить любое преобразование, которое можно применить к отдельному компоненту, для целого Map с компонентами роботов.

Внимательный читатель мог заметить кое-что странное в том, что Map принадлежит к Functor. Вид Map — это * -> * -> *, так как Map принимает два аргумента: первый — для ключей, а второй — для значений. Ранее мы сказали, что экземпляры Functor должны иметь вид * -> *, тогда как это возможно? Если вы взглянете на поведение <$> с вашей partsDB, это станет понятно. Functor в случае с Map заботится только о значениях в Map, но не о ключах. Когда Map является экземпляром Functor, вас волнует только одна типовая переменная, которая используется для его значений. Так что, когда вам нужен Map в качестве экземпляра Functor, вы относитесь к нему как к типу вида * -> *. Когда мы представляли виды в уроке 18, они могли показаться чересчур абстрактными. Но они могут быть полезны для выявления проблем, которые возникают с более продвинутыми классами типов.

27.3.5. Преобразование IO RobotPart в IO HTML

Наконец, вы можете получать компоненты, приходящие из IO. Вы легко можете смоделировать такую ситуацию, воспользовавшись для создания RobotPart в контексте IO методом return.

Листинг 27.15 Моделирование RobotPart в контексте IO

```
leftArmIO :: IO RobotPart
leftArmIO = return leftArm
```

Предположим, вы хотите перевести этот компонент в HTML, чтобы потом записать HTML-фрагмент в файл. К этому моменту приём уже должен быть знаком.

Листинг 27.16 Преобразование IO RobotPart к IO HTML

```
htmlSnippet :: IO Html
htmlSnippet = renderHtml <$> leftArmIO
```

Давайте посмотрим на все эти преобразования разом:

```
partHtml :: Maybe Html
partHtml = renderHtml <$> partVal

allPartsHtml :: [Html]
allPartsHtml = renderHtml <$> allParts

htmlPartsDB :: Map.Map Int Html
htmlPartsDB = renderHtml <$> partsDB

htmlSnippet :: IO Html
htmlSnippet = renderHtml <$> leftArmIO
```

Как вы можете увидеть, <$> из Functor предоставляет общий интерфейс для применения функции к значению в контексте. Для таких типов, как список и Map, это разумный способ обновления значений в этих контейнерах. Для IO существенно иметь возможность изменять значения в контексте IO, так как вы не можете извлекать из него значения.

Итоги

В этом уроке нашей целью было представить вам класс типов Functor. Класс типов Functor позволяет применять обычную функцию к значениям внутри контейнера (вроде списка) или контекста (вроде IO или Maybe). Если у вас есть функция Int -> Double и значение типа Maybe Int, вы можете использовать fmap (или <$>) из Functor для применения одного к другому, получая значение Maybe Double. Экземпляры Functor невероятно мощны, так как они позволяют вам заново использовать функцию с любым типом из класса типов Functor. Значения [Int], Maybe Int, и IO Int могут обрабатываться одними и теми же функциями. Давайте проверим, как вы это поняли.

Задача 27.1. Когда мы представили параметризованные типы в уроке 15, то использовали минимальный тип Box в качестве примера:

```
data Box a = Box a deriving Show
```

Реализуйте класс типов Functor для типа Box. Затем реализуйте функцию morePresents, которая меняет тип с Box a на Box [a], который содержит *n* копий исходного значения в списке. Убедитесь, что использовали fmap для реализации.

Задача 27.2. Теперь предположим, что у вас есть простая коробка вроде этой:

```
myBox :: Box Int
myBox = Box 1
```

Используйте `fmap`, чтобы переложить ваше значение в коробке в другую коробку. Затем определите функцию `unwrap`, которая вынимает значение из коробки, и используйте `fmap` на этой функции, чтобы получить вашу первоначальную коробку. Вот как ваш код должен работать в GHCi:

```
GHCi> wrapped = fmap ? myBox
GHCi> wrapped
Box (Box 1)
GHCi> fmap unwrap wrapped
Box 1
```

Задача 27.3. Напишите интерфейс командной строки для базы данных компонентов роботов `partsDB`, который позволит пользователю искать стоимость предмета по его идентификатору. Используйте тип `Maybe` для обработки случая, когда пользователь запрашивает отсутствующий предмет.

28

Приступаем к аппликативным функторам: функции в контексте

После прочтения урока 28 вы:

- научитесь писать программы, обрабатывающие отсутствующие данные;
- сможете расширить возможности класса типов `Functor` с помощью `Applicative`;
- будете использовать `Applicative` для написания кода, способного работать в разных контекстах.

В предыдущих уроках вы разобрались в том, как класс типов `Functor` позволяет выполнять вычисления в таких контейнерах, как список, или контекстах вроде `Maybe` и `IO`. Ключевой определяющий метод `Functor` — `fmap` (или, как его чаще обозначают, операция `<$>`), который работает аналогично `map` для списков. В этом уроке вы будете работать с более мощным классом типов, называемым `Applicative`. Класс типов `Applicative` расширяет возможности `Functor` за счёт использования функций, которые сами по себе находятся в неком контексте. Хотя это может показаться не таким уж и полезным, это позволяет соединять вместе длинные последовательности вычислений в контекстах, таких как, например, `IO` или `Maybe`.

В первом же примере вы увидите ограничения класса `Functor`, когда будете разрабатывать консольное приложение для расчёта расстояния между двумя городами. Загвоздка в том, что вам потребуется передавать функции два значения типа `Maybe`, что, оказывается, не может быть осуществлено с помощью `Functor`. Вы увидите, как это можно решить при помощи `Applicative`. После того как вы научитесь использовать класс ти-

пов Applicative, вы увидите, как он помогает при создании данных в контекстах IO или Maybe, расширив ваши возможности по повторному использованию кода.

Обратите внимание. Вам требуется соединить имя и фамилию человека: "Алан" ++ " " ++ "Тьюринг". Проблема в том, что и фамилия, и имя являются Maybe String, так как они пришли из ненадёжного источника и поэтому могут отсутствовать. Подумайте, как можно использовать эти строки для получения значения типа Maybe String, содержащего полное имя.

28.1. Расчёт расстояния между городами

В этом разделе вы напишите небольшое консольное приложение, которое будет возвращать пользователю расстояния между городами, названия которых он укажет. Самым большим препятствием будет обработка случаев, когда пользователь вводит город, которого нет в базе данных. Эту задачу вы сможете решить с помощью Maybe и Functor, но потребуется нечто большее для работы с двумя значениями в контексте Maybe.

Примечание. Код в этом разделе следует записывать в файл dist.hs.

Давайте предположим, что у вас есть значение типа Map (не забудьте написать import qualified Data.Map as Map) для хранения названий городов и соответствующих им пар координат (широты и долготы).

Листинг 28.1 Map в качестве базы данных координат городов

```
type LatLong = (Double,Double)

locationDB :: Map.Map String LatLong
locationDB = Map.fromList [("Аркхем",(42.6054,-70.7829))
                          ,("Иннсмут",(42.8250,-70.8150))
                          ,("Каркоза",(29.9714,-90.7694))
                          ,("Нью-Йорк",(40.7776,-73.9691))]
```

Ваша цель — вычислить расстояние между двумя точками на поверхности земного шара, хранящимися в locationDB. Для этого потребуется формула расчёта расстояния между двумя точками на сфере. Так как сфера —

кривая поверхность, вы не сможете просто вычислить длину отрезка между этими точками. Вместо этого мы воспользуемся формулой гаверсинусов. Обратите внимание, что для начала вам потребуется перевести долготу и широту в радианы. Ниже приведена реализация `haversine` (необязательно вникать в детали реализации этой функции).

Листинг 28.2 Вычисление расстояния между двумя точками

```
toRadians :: Double -> Double
toRadians degrees = degrees * pi / 180

latLongToRads :: LatLong -> (Double,Double)
latLongToRads (lat,long) = (rlat,rlong)
    where rlat = toRadians lat
          rlong = toRadians long

haversine :: LatLong -> LatLong -> Double
haversine coords1 coords2 = earthRadius * c
    where (rlat1,rlong1) = latLongToRads coords1
          (rlat2,rlong2) = latLongToRads coords2
          dlat = rlat2 - rlat1
          dlong = rlong2 - rlong1
          a = (sin (dlat/2))^2 + cos rlat1 * cos rlat2
                                           * (sin (dlong/2))^2
          c = 2 * atan2 (sqrt a) (sqrt (1-a))
          earthRadius = 6378.1
```

Пример вызова `haversine` для расчёта расстояния на поверхности шара:

```
GHCi> haversine (40.7776,-73.9691) (42.6054,-70.7829)
333.134208
```

Далее вам нужно будет реализовать небольшой консольный интерфейс, позволяющий пользователю получить расстояние между городами. По нашей задумке пользователь вводит названия двух городов, а затем получает расстояние. Учитывая, что вы работаете с вводом от пользователя, определённо нужно обрабатывать случай, когда пользователь вводит город, которого нет в базе данных. Если один из городов отсутствует, пользователь будет уведомлён о том, что произошла ошибка, связанная с вводом.

Листинг 28.3 Вывод потенциально отсутствующего расстояния

```
printDistance :: Maybe Double -> IO ()
printDistance Nothing =
        putStrLn "Ошибка, введён отсутствующий в базе город"
printDistance (Just dist) = putStrLn (show dist ++ " км")
```

Теперь требуется связать всё вместе. Вам нужно получить два набора координат из `locationDB`, рассчитать дистанцию, а затем передать расстояние функции `printDistance`. Трудность в том, что `locationDB` возвращает значения типа `Maybe`. Давайте посмотрим на типы функций, тогда проблему легче будет заметить. Вот тип функции `haversine`:

```
haversine :: LatLong -> LatLong -> Double
```

А требуется нечто похожее на функцию, показанную на рис. 28.1.

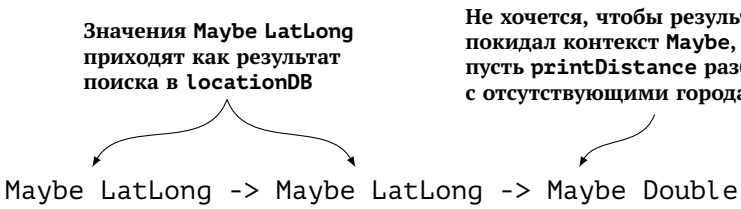


Рис. 28.1: Тип функции, соединяющей `locationDB` с `printDistance`

Это очень похоже на тип `haversine`, но с обёрткой `Maybe`. Эта проблема должна напоминать трудности, с которыми вы справлялись с помощью `Functor`. Для решения задачи вам требуется использовать обычные функции в каком-то контексте. Самое банальное и простое решение — написать функцию-обёртку `haversine` специально для работы с `Maybe`.

Листинг 28.4 Одно из решений — написать функцию-обёртку

```
haversineMaybe :: Maybe LatLong -> Maybe LatLong
                                -> Maybe Double
haversineMaybe Nothing _ = Nothing
haversineMaybe _ Nothing = Nothing
haversineMaybe (Just val1) (Just val2) =
                Just (haversine val1 val2)
```

Функция `haversineMaybe` является скверным решением по двум причинам. Во-первых, вам потребуется каждый раз писать подобные обёртки, что очень утомляет. Во-вторых, для разных контекстов придётся писать разные версии `haversineMaybe`. Так как основное предназначение `Functor` — обобщённый способ работы в разных контекстах, давайте посмотрим, получится ли решить эту проблему с его помощью.

Проверка 28.1. Напишите функцию `addMaybe`, вычисляющую сумму двух значений типа `Maybe Int`.

28.1.1. Ограничения класса Functor

Прежде чем уйти в дебри, давайте вспомним единственную функцию `fmap` из класса типов `Functor` и разберём её тип, показанный на рис. 28.2.

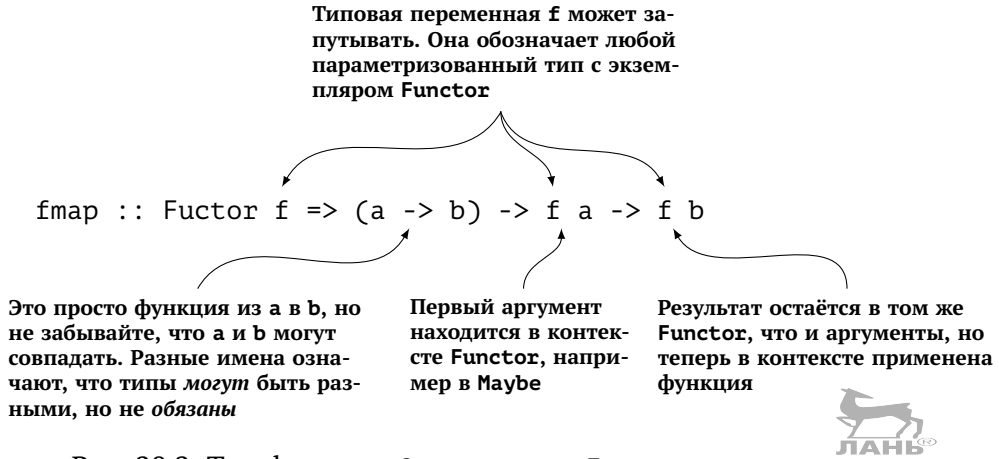


Рис. 28.2: Тип функции `fmap` из класс `Functor` с пояснениями

Функция `fmap` принимает любую функцию из типа `a` в тип `b` и значение типа `a` в контексте функтора (например, `Maybe`), возвращая значение типа `b` в том же контексте. Если рассуждать о проблеме в терминах типов, то это очень близко к тому, что вам нужно. Главное отличие в том, что у вас есть ещё один аргумент. Опишем то, что вам требуется:

(1) принять функцию `haversine` типа `LatLong -> LatLong -> Double`;

Ответ 28.1

```
addMaybe :: Maybe Int -> Maybe Int -> Maybe Int
addMaybe (Just x) (Just y) = Just (x + y)
addMaybe _ _ = Nothing
```

(2) затем принять два аргумента в контексте `Maybe`, то есть с типами `Maybe LatLong` и `Maybe LatLong`;

(3) вернуть результат в том же контексте: `Maybe Double`.

Это подводит нас к следующему типу:

```
(LatLong -> LatLong -> Double) ->
             (Maybe LatLong -> Maybe LatLong -> Maybe Double)
```

Если же вы перейдёте к более общему типу, то получите следующее:

```
Functor f => (a -> b -> c) -> f a -> f b -> f c
```

Это практически идентично `fmap`, за исключением того, что добавился ещё один аргумент. Это и есть одно из ограничений `fmap` — эта функция работает исключительно с функциями одного аргумента. Так как основная проблема состоит в наличии лишнего аргумента, использование частичного применения поможет вам приблизиться к решению.

Проверка 28.2. Предположим, что вы не хотите связываться с `Maybe` и собираетесь использовать для хранения координат обычные пары. Если у вас есть пара координат `newYork`, то как бы вы написали функцию `distanceFromNY`, которая принимает координаты другого города?

28.2. Операция <*> и частичное применение в контексте

Задача, которую вам теперь нужно решить, заключается в обобщении `fmap` для работы с несколькими аргументами. В уроке 5 вы разобрались в том, что частичное применение означает то, что функция, вызванная с меньшим количеством аргументов, чем ей требуется для работы, возвращает функцию, ожидающую оставшиеся аргументы. Затем в разделе 10.2.2 вы увидели, что все функции являются одноаргументными, а многоаргументные функции — всего лишь цепочка функций с одним аргументом.

Ответ 28.2

```
distanceFromNY = haversine newYork
```

Поэтому ключом к решению текущей проблемы оказывается возможность использовать частичное применение в каком-нибудь контексте, например `Maybe` или `IO`.

Самое серьёзное ограничение `<$>`: если результатом является функция в контексте, полученная с помощью частичного применения, то нет никакого способа её использовать. Например, можно использовать `<$>`, `(+)` и число 1 в контексте `Maybe` для получения функции `maybeInc`.

Листинг 28.5 Операция <$> и частичное применение в контексте

```
maybeInc = (+) <$> Just 1
```

Если вы посмотрите на тип этой функции, то увидите следующее:

```
maybeInc :: Maybe (Integer -> Integer)
```

Операция `(+)` — это функция, которая принимает два аргумента; а при использовании с `<$>` и значением в `Maybe` вы получили функцию, ожидающую ещё одно значение, которая находится внутри `Maybe`. Теперь у вас есть функция в `Maybe`, но нет способа взаимодействовать с ней! Вспомним на минуту о нашем языке кругов и квадратов для значений в контексте, который поможет визуализировать проблему. На рис. 28.3 проиллюстрирована суть проблемы.

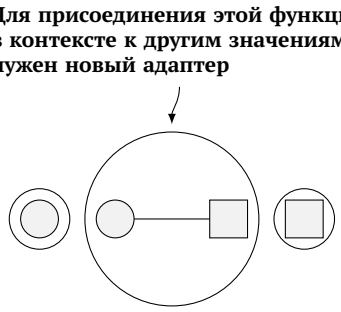


Рис. 28.3: Требуется адаптер, применяющий функцию в контексте к значению в контексте

К счастью, уже есть класс типов, идеально подходящий для решения этой проблемы!

28.2.1. Знакомьтесь, операция <*>

Класс типов `Applicative` увеличивает выразительную мощь `Functor` с помощью операции `<*>`. Если вы внимательно посмотрите на её тип,

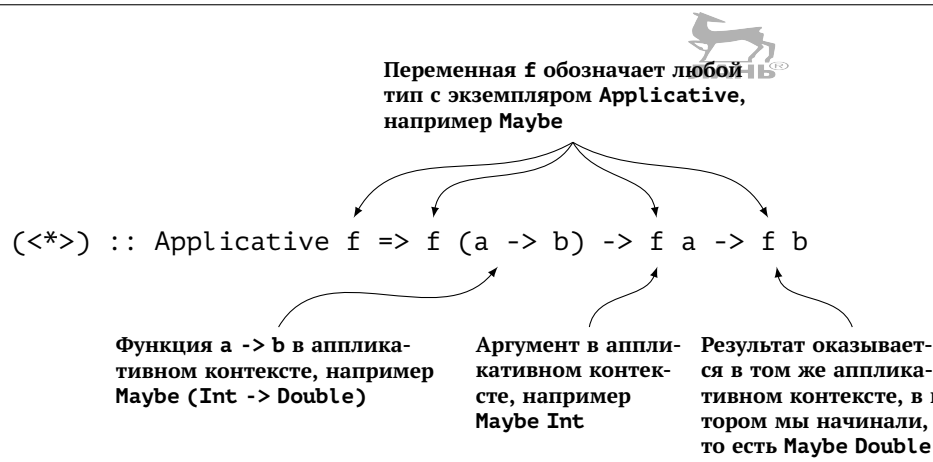


Рис. 28.4: Тип операции <*> с пояснениями

то вполне сможете разобраться в её работе самостоятельно (рис. 28.4).

Операция <*> из `Applicative` позволяет применять функции в контексте. Теперь вы можете использовать функцию `maybeInc` для увеличения значений в `Maybe`. Вот несколько примеров из GHCi:

```
GHCi> maybeInc <*> Just 5
Just 6
GHCi> maybeInc <*> Nothing
Nothing
GHCi> maybeInc <*> Just 100
Just 101
```

У вас не только получилось скомбинировать два значения в `Maybe`, но и найти способ для использования функций с двумя аргументами в контексте `Maybe`. Точно так же это можно использовать и с `Maybe String`:

```
GHCi> (++) <$> Just "кошки" <*> Just " и собаки"
Just "кошки и собаки"
GHCi> (++) <$> Nothing <*> Just " и собаки"
Nothing
GHCi> (++) <$> Just "кошки" <*> Nothing
Nothing
```

Благодаря частичному применению вы можете использовать <$> и <*> для создания функций с произвольным количеством аргументов.

Проверка 28.3. Используя шаблон из примера для двухаргументных функций, передайте функциям (*), div и mod эти два значения:

```
val1 = Just 10
val2 = Just 5
```

28.2.2. Использование <*> для завершения программы для вычисления расстояния между городами

С Applicative и <*> вы наконец-то сможете решить задачу, применив функцию haversine к двум значениям в Maybe:

```
GHCi> startingCity = Map.lookup "Каркоза" locationDB
GHCi> destCity = Map.lookup "Иннсмут" locationDB
GHCi> haversine <$> startingCity <*> destCity
Just 2277.2217600000004
```

Этот код трудновато читать из-за обильного использования операций. На рис. 28.5 иллюстрируются ключевые моменты, которые помогут разобраться в том, что здесь происходит.

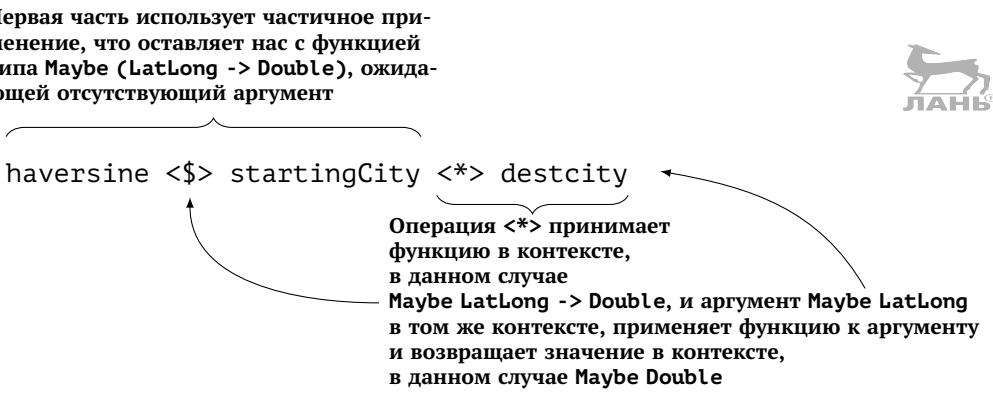


Рис. 28.5: Использование <$> и <*> для вычислений в контексте Maybe

Ответ 28.3

```
val1 = Just 10
val2 = Just 5
result1 = (+) <$> val1 <*> val2
result2 = div <$> val1 <*> val2
result3 = mod <$> val1 <*> val2
```

Теперь вы сможете расширить выразительность fmap с помощью <*>, например можете соединить всё вместе, написав программу, которая принимает названия двух городов в качестве входных данных, а возвращает расстояние между ними. Вот функция main из получившейся программы.

Листинг 28.6 Функция main из dist.hs

```
main :: IO ()
main = do
    putStrLn "Введите название первого города:"
    startingInput <- getLine
    let startingCity = Map.lookup startingInput locationDB
    putStrLn "Введите название второго города:"
    destInput <- getLine
    let destCity = Map.lookup destInput locationDB
    let distance = haversine <$> startingCity <*> destCity
    printDistance distance
```

Можете скомпилировать эту программу и убедиться, что теперь она корректно обрабатывает ошибки ввода:

```
$ ./dist
Введите название первого города:
Каркоза
Введите название второго города:
Иннсмут
2277.2217600000003 км
$ ./dist
Введите название первого города:
Каркоза
Введите название второго города:
Чикаго
Ошибка, введён отсутствующий в базе город
```

Этот пример демонстрирует, насколько удобно работать с Functor и Applicative. Только что вы написали программу, которая корректно обрабатывает недостающие данные, но вам ни разу не пришлось проверять наличие этих данных напрямую с помощью условных операций или волноваться по поводу обработки исключений. Даже лучше — вы смогли реализовать ядро вашей программы, функцию haversine, не переживая о том, что что-то может пойти не так.

Система типов Haskell не позволит случайно передать значение типа Maybe LatLong функции haversine. Практически во всех языках программирования, даже в языках со статической типизацией, например в Java

или C#, нет возможности убедиться в том, что функция точно не пропустит пустые значения. `Functor` и `Applicative` служат полезным дополнением к безопасности, предоставленной типизацией, помогая использовать обычные функции, например `haversine`, с `Maybe` и `IO`-типами без урона безопасности.

28.2.3. Функции нескольких переменных в контексте IO

Тип `IO` также принадлежит классу `Applicative`. Чтобы продемонстрировать это, давайте посмотрим, как, используя <$> и <*>, можно создать небольшое консольное приложение, возвращающее наименьшее из трёх введённых чисел, назовём его min3.hs. Начнём с функции `minOfThree`, принимающей три аргумента и возвращающей наименьшее из них.

Листинг 28.7 Вычисление наименьшего из трёх чисел

```
minOfThree :: (Ord a) => a -> a -> a -> a
minOfThree val1 val2 val3 = min val1 (min val2 val3)
```

Затем создадим простое `IO`-действие, `readInt`, которое читает значение типа `Int`, введённое пользователем.

Листинг 28.8 Простое IO-действие, написанное с помощью <$>

```
readInt :: IO Int
readInt = read <$> getLine
```

Теперь вы можете использовать <$> с <*>, чтобы реализовать `IO`-действие, которое читает три целых числа и возвращает наименьшее из них.

Листинг 28.9 Обработка нескольких аргументов с помощью <*>

```
minOfInts :: IO Int
minOfInts = minOfThree <$> readInt <*> readInt <*> readInt
```

Наконец-то вы можете соединить всё в `main`.

Листинг 28.10 Функция main для min3.hs

```
main :: IO ()
main = do
    putStrLn "Введите три числа"
    minInt <- minOfInts
    putStrLn (show minInt ++ " является наименьшим")
```

А сейчас вы можете скомпилировать и запустить min3.hs:

```
$ ghc min3.hs
$ ./min3
Введите три числа
1
2
3
1 является наименьшим
```

Благодаря силе частичного применения и <*> вы можете использовать столько аргументов, сколько захотите!

Проверка 28.4. Используя `minOfThree`, получите значение `Maybe Int` из этих трёх значений в контексте `Maybe`:

```
Just 10
Just 3
Just 6
```

28.3. Использование <*> для данных в контексте

Одним из наиболее частых применений `Applicative` на практике является создание данных с помощью информации в контексте. Например, предположим, что у вас есть тип с данными пользователя компьютерной игры.

Листинг 28.11 Тип для хранения информации об игроке

```
data User = User
    { name :: String
    , gamerId :: Int
    , score :: Int
    } deriving Show
```

Ответ 28.4

```
GHCi> minOfThree <$> Just 10 <*> Just 3 <*> Just 6
Just 3
```

Обратите внимание, несмотря на использование синтаксиса, характерного для записей, вы можете создавать значения этого типа как обычно. К примеру:

```
GHCi> User {name = "Сью", gamerId = 1337, score = 9001}
User {name = "Сью", gamerId = 1337, score = 9001}
GHCi> User "Сью" 1337 9001
User {name = "Сью", gamerId = 1337, score = 9001}
```

Давайте рассмотрим два случая, когда вам может потребоваться создавать значения типа `User` в контексте.

28.3.1. Создание данных пользователя в контексте Maybe

Первый пример — `Maybe`. Резонно предположить, что информация, которую вы собрали, могла быть получена из ненадёжных источников, поэтому часть данных может отсутствовать. Вот несколько `Maybe`-типов, их можно использовать для хранения данных с сервера, который мог не отослать часть информации.

Листинг 28.12 Информация для создания значений типа User

```
serverUsername :: Maybe String
serverUsername = Just "Сью"

serverGamerId :: Maybe Int
serverGamerId = Just 1337

serverScore :: Maybe Int
serverScore = Just 9001
```

Для создания данных пользователя вы можете использовать <$> и <*>, так как конструктор данных `User` работает прямо как обычная функция, принимающая три аргумента. Вот код для демонстрации этого в GHCi:

```
GHCi> User <$> serverUsername <*> serverGamerId <*> serverScore
Just (User {name = "Сью", gamerId = 1337, score = 9001})
```

Другим контекстом, в котором, возможно, потребуется создавать значения типа `User`, является `IO`. Вам может потребоваться реализовать консольное приложение, которое считывает три строки с данными пользователя из консоли, а возвращает информацию об игроке. Тут можно снова воспользоваться функцией `readInt` из предыдущего урока для преобразования пользовательского ввода в значение типа `Int`.

Листинг 28.13 Создание значения типа User из IO-типов

```
readInt :: IO Int
readInt = read <$> getLine
main :: IO ()
main = do
    putStrLn "Введите ник, ID игрока и его очки"
    user <- User <$> getLine <*> readInt <*> readInt
    print user
```

Самое приятное здесь то, что вам потребовалось определить только один тип, `User`, который состоит из обычных строк и целых чисел. Но благодаря классу типов `Applicative` вы можете использовать один и тот же код при работе в разных контекстах.

Проверка 28.5. Проверьте, что получится, если создать пользователя с пропущенным ником (`Nothing`).

Итоги

В этом уроке нашей целью было познакомить вас с `Applicative`. Операция `<*>` из этого класса типов позволит вам использовать функции, находящиеся в некотором контексте. Например, если у вас есть функция, которая, возможно, отсутствует, `Maybe (Int -> Double)`, вы можете передать ей значение в том же контексте, `Maybe Int`, и получить результат работы этой функции в том же контексте, `Maybe Double`. Может показаться, что эта операция нужна редко, но очень важно усилить выразительную мощь `Functor` с помощью инструментов для работы с функциями нескольких переменных. В связи с широкой распространённостью в программах на Haskell частичного применения совершенно обычным делом является работа с функциями в контексте. Без `Applicative` было бы невозможно

Ответ 28.5

```
GHCi> User <$> Nothing <*> serverGamerId <*> serverScore
Nothing
```

использовать эти функции. Давайте посмотрим, как вы поняли материал данного урока.

Задача 28.1. Реализация `haversineMaybe` была достаточно простой. Теперь попробуйте написать функцию `haversineIO` без использования <*>. Вот тип этой функции:

```
haversineIO :: IO LatLong -> IO LatLong -> IO Double
```

Задача 28.2. Перепишите `haversineIO`, на этот раз воспользуйтесь <*>.

Задача 28.3. Вспомните тип `RobotPart` из предыдущего урока:

```
data RobotPart = RobotPart
    { name :: String
    , description :: String
    , cost :: Double
    , count :: Int
    } deriving Show
```

Реализуйте консольное приложение, которое хранит базу из нескольких значений типа `RobotPart` (минимум 5), принимает от пользователя ID двух частей, а возвращает ту деталь, цена которой ниже. Обработайте случай, когда пользователь вводит ID, отсутствующий в базе.

29

Списки как контекст: углубляемся в аппликативные вычисления

После прочтения урока 29 вы:

- разберётесь с формальным определением класса типов `Applicative`;
- будете трактовать параметризованные типы как контейнеры либо контексты;
- научитесь использовать список как контекст для выполнения недетерминированных вычислений.

В предыдущем уроке вы научились использовать операцию <*> (произносится как *app*) для расширения функциональности операции <$> (читается как *fmap*) для функторов. В этом уроке вы познакомитесь с классом типов `Applicative` поближе. Вы изучите различия между типами-контейнерами и типами-контекстами. В конце урока вы увидите впечатляющие результаты использования списков в качестве контекста.

Обратите внимание. В кафе на завтрак вам предлагают выбрать один вариант по каждой позиции:

- кофе или чай;
- яйца, блинчики или вафли;
- тост или бисквит;
- сосиска, ветчина или бекон.

Каковы возможные комбинации выбранных блюд и как списки могут вам помочь их перечислить?

29.1. Представляем класс типов Applicative

Класс типов `Applicative` позволяет вам использовать функции, находящиеся в контексте, например `Maybe` или `IO`. Как вы увидели в предыдущем уроке, это расширяет возможности класса типов `Functor`. Функтор является «родительским» классом по отношению к `Applicative` (*суперклассом*). Схема наследования приведена на рис. 29.1, а полное определение — на рис. 29.2.

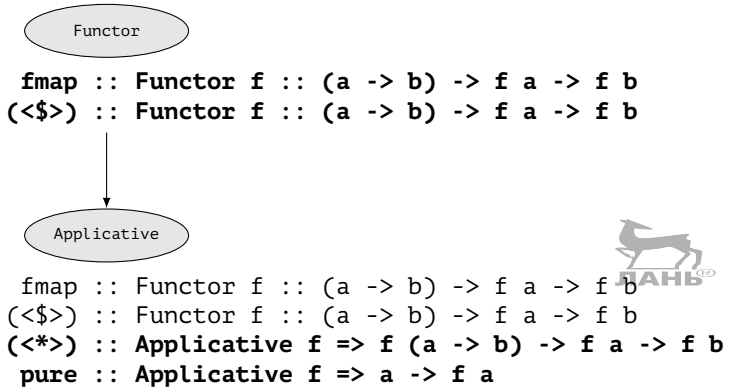


Рис. 29.1: Класс типов `Functor` является родительским для `Applicative`

Затруднение в этом определении может представлять то, что на типовую переменную `f` наложено сразу два ограничения. Первое говорит, что `f` — `Functor`, таким образом обеспечивая необходимость `Applicative` реализовывать `Functor`. Второе ограничение утверждает, что `f` реализует класс `Applicative`. В типовых аннотациях методов переменная `f` соответствует любому `Applicative` (см. рис. 29.2). Обратите внимание: тип операции `<*>` совпадает с типом `fmap`, за исключением того, что сама функция тоже находится в контексте. Это небольшое различие позволяет вам объединять в цепочки длинные последовательности функций, находящихся в контекстах `Functor`. Ниже приведены примеры использования `<$>` и `<*>` для выполнения математических операций на типах `Maybe`:

```
GHCi> (*) <$> Just 6 <*> Just 7
Just 42
GHCi> div <$> Just 6 <*> Just 7
Just 0
```

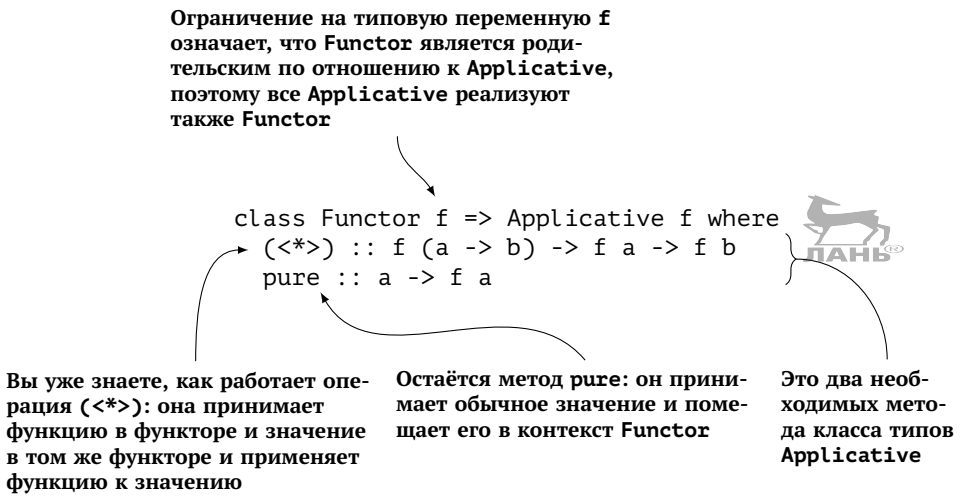


Рис. 29.2: Определение класса типов `Applicative`

```
GHCi> mod <$> Just 6 <*> Just 7
Just 6
```

Эти примеры могут выглядеть изящными или, наоборот, странными, в зависимости от того, насколько вы привыкли работать с инфиксными операциями в Haskell. Без сомнений, использование операций <$> и <*> поначалу может выглядеть несколько запутанно. Ещё больше усложняет ситуацию то, что, в отличие от <$> и `fmap`, операция <*> не имеет эквивалентной ей функции. Если вы испытываете трудности с применением этих операций, практикуйтесь. Попробуйте вернуться к некоторым примерам из модуля 1 и изменить значения, использованные в качестве аргументов, на значения в `Maybe`. Запомните: благодаря `fmap` и <*> вам не нужно переписывать функции для работы со значениями в контексте `Maybe`.

Проверка 29.1. Используя <$> и <*>, скомбинируйте два значения типа `Maybe String` с помощью операции ++.

Ответ 29.1

```
GHCi> (++) <$> Just "Программируй" <*> Just " на Haskell"
Just "Программируй на Haskell"
```

29.1.1. Функция pure

Функция pure является вторым методом класса Applicative. Эта полезная вспомогательная функция позволяет взять обычное значение или функцию и поместить их в контекст. Лучший способ понять pure — эксперименты в GHCi. В случае с Maybe pure будет возвращать Just:

```
GHCi> pure 6 :: Maybe Int
Just 6
```

Вы также можете использовать pure, для того чтобы поместить функцию в контекст Applicative. Например, чтобы прибавить 6 к (Just 5), можно использовать fmap или pure:

```
GHCi> (6+) <$> Just 5
Just 11
GHCi> pure (6+) <*> Just 5
Just 11
```

Это простые примеры, но на практике вам часто будет нужен быстрый способ преобразовать значение к требуемому типу Applicative. На нашем визуальном языке pure также играет важную роль (см. рис. 29.3).

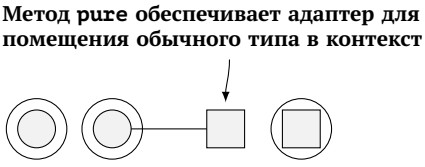


Рис. 29.3: Метод pure помещает значение произвольного типа в контекст

С помощью pure вы можете взять значение любого типа и поместить его в контекст. Это позволяет выполнять в контексте любые вычисления.

Проверка 29.2. Приведите строку "Привет, Мир!" к типу IO String.

Ответ 29.2

```
hello :: IO String
hello = pure "Привет, мир!
```

29.2. Контейнеры и контексты

До сих пор мы несколько пространно говорили о различиях между параметризованными типами, представляющими контейнеры и контексты. К данному моменту вам должно стать немного более ясно, что означают эти термины. Причина состоит в том, что, в отличие от `Functor`, `Applicative` и класс типов `Monad`, который будет представлен в следующем уроке, имеют смысл, только если вы говорите о типах как о контексте. Вся суть отличия состоит в следующем:

- параметризованные типы, представляющие контейнеры, отражают структуру, в соответствии с которой хранятся значения;
- когда тип представляет собой контекст, дополнительная информация связана с самим типом, а не со структурой данных, в которой хранятся его значения.

Давайте изучим этот вопрос глубже. Лучший способ проверить, является ли тип *контейнером*, — ответить на вопрос: можно ли сказать, на что способен этот тип, не глядя на его имя? Например, рассмотрим двухэлементный кортеж `(a, b)`. Вы могли бы реализовать такой же тип данных, назвав его `Blah`.

Листинг 29.1 Двухэлементный кортеж с неудачным именем

```
data Blah a b = Blah a b
```

Даже с таким неудачным именем, как `Blah`, едва ли можно утверждать, что этот тип чем-то отличается от обычной пары `(a, b)`.

Тип `Data.Map` также представляет собой структуру данных. Вы можете назвать его `Dictionary`, `BinarySearchTree`, `MagicLookupBox` и т. д. Но значение типа определяется самой структурой данных. Даже если бы все функции для работы с `Map` были бы написаны на инопланетном языке, вы бы смогли понять, для чего используется `Map` (рано или поздно).

Стоит отметить, что двухэлементный кортеж и `Data.Map` являются экземплярами класса `Functor`, но *не являются* экземплярами класса типов `Applicative`. Вспомните: ключевой особенностью `Applicative` является то, что он позволяет вам применять функцию к параметризованному типу. Другим хорошим способом отличить контейнеры от контекстов является вопрос: «Имеет ли смысл значение `<тип> <функция>`?». Значение типа `Maybe (a -> b)` представляет собой функцию из `a` в `b`, которая сама по себе может не существовать (то есть быть `Nothing`). Значе-

ние типа `IO (a -> b)` — любая функция, действующая в мире ввода-вывода. А что такое функция в `Data.Map`? Аналогично, что означает функция в двухэлементном кортеже? Если вы вдруг можете ответить на эти вопросы, то у вас есть основа для реализации экземпляра класса типов `Applicative` для этих типов.

Когда тип является контекстом, информация, которую он несёт, связана именно с типом данных (с какой-либо его особенностью), но не с его структурой. Самым очевидным примером типа-контекста является тип `IO`. Когда вы впервые узнаёте о параметризованных типах, вы знакомитесь с идеей, которую можно представить типом `Box`.

Листинг 29.2 Простой тип Box не сильно отличается от IO

```
data Box a = Box a
```

Тип `Box`, очевидно, является довольно простым и не очень полезным типом. Но на уровне конструкторов данных нет разницы между типами `Box` и `IO`. Тип `IO` с нашей точки зрения позволяет всего лишь сохранить внутри себя значения, появляющиеся в результате выполнения операций ввода-вывода (на самом деле тип `IO` гораздо сложнее, но вы не можете увидеть этого прямо сейчас).

Тип `Maybe` является ещё одним типом-контекстом. Допустим, вам нужно создать параметризованный тип для вычислений с ограниченными ресурсами. Этот тип мог бы выглядеть следующим образом:

Листинг 29.3 Тип для представления ресурсных ограничений

```
data ResourceConstrained a = NoResources | Okay a
```

За кадром, возможно, происходила бы какая-то магия, необходимая для определения наличия используемых ресурсов, но на уровне конструкторов типов `ResourceConstrained` не отличается от `Maybe`. Большая часть информации об этом типе содержится в предполагаемом контексте самого типа (то есть в его значении).

Лучший способ осознать различие контекстов и контейнеров — рассмотреть пример. Для этого идеально подходит список. Нетрудно догадаться, что список является контейнером, так как он представляет собой распространённую структуру данных. Но список также описывает контекст. Если вы сможете понять, каким образом список может быть контейнером и контекстом одновременно, то окажетесь на пути к истинному пониманию класса типов `Applicative`.

Проверка 29.3. Допустим, вы хотите, чтобы выполнялось равенство `pure (+) <*> (1,2) <*> (3,4) = (1 + 2, 1 + 4, 2 + 3, 2 + 4) = (3, 5, 5, 6)`. Почему это не работает?

29.3. Список как контекст

ЛАНЬ

Фундаментальный для Haskell тип списка является одновременно контейнером и контекстом. Понять, что список является контейнером, нетрудно: список представляет собой последовательность ячеек с данными любого (для всех ячеек одного и того же) типа. Однако список обладает экземпляром класса типов `Applicative`, так что должен быть способ трактовать его как контекст.

ЛАНЬ

Чтобы разобраться со списком как контекстом, вы должны иметь возможность ответить на вопрос: «Что означает применение функции к двум или более значениям в контексте списка?» Например, каким может быть смысл выражения `[1000, 2000, 3000] + [500, 20000]`? Что насчёт:

```
[1000, 2000, 3000]
+ [500, 20000]
= [1000, 2000, 3000, 500, 20000]
```

Это просто сложение списков (то есть конкатенация, операция `++`). Здесь важно разобраться, что означает комбинирование двух значений *в контексте* списка с помощью сложения. В терминах `Applicative` вы бы прочли это выражение следующим образом:

```
pure (+) <*> [1000, 2000, 3000] <*> [500, 20000]
```

Сам по себе список как структура данных не даёт вам достаточного объёма информации для ответа на этот вопрос. Чтобы понять, как сложить два значения в контексте списка, вам потребуется некая общая идея применения бинарной функции к значениям в списке.

Ответ 29.3. Это не работает, потому что тип `(3, 5, 5, 6)` отличается от типов `(1, 2)` и `(3, 4)`. Тип первого выражения: `(a, b, c, d)`, тогда как последних двух — `(a, b)`.

Лучший способ взглянуть на список как на контекст следующий: список описывает *недетерминированное* вычисление. Обычно о программировании думают как о совершенно детерминированном вычислении. Очередной шаг вычисления следует за предыдущим в строго определённом порядке, приводящем к единственному финальному результату. В недетерминированных вычислениях вы производите несколько возможных вычислений сразу. Говоря в терминах недетерминированного вычисления, когда вы складываете значения в контексте списка, вы складываете все возможные комбинации значений из двух списков. Вы можете увидеть неожиданный результат использования <*> с двумя списками в GHCi:

```
GHCi> pure (+) <*> [1000, 2000, 3000] <*> [500, 20000]
[1500,21000,2500,22000,3500,23000]
```

Сложение двух целых в контексте списка означает сложение всех возможных комбинаций значений из двух списков.

29.3.1. Различия контейнера и контекста в случае списка

Сейчас важно остановиться и выделить основные отличия между списком как контейнером и списком как контекстом:

- список как *контейнер* — это последовательность значений;
- список как *контекст* — это набор возможностей, он соответствует переменной, принимающей множество различных значений.

Не позволяйте ввести себя в заблуждение знакомой интерпретации списка в качестве контейнера. О типах `Maybe` и `IO` гораздо проще думать как о контекстах. `Maybe Int` — это целое число в контексте возможного отсутствия. `IO Int` — это целое число в контексте того, что оно было получено в результате действия ввода-вывода, которое могло привести к появлению побочных эффектов или других проблем. `[Int]` — это `Int` в контексте того, что на его месте может быть множество возможных значений. Из-за того, что у `[Int]` может быть несколько возможных значений, применение функции `(Int -> Int -> Int)` в контексте списка должно заставить вас думать о недетерминированном вычислении и привести к вычислению всех возможных результатов применения этой операции.

29.3.2. Пример с игровым шоу

В качестве примера предположим, что вы находитесь на игровом шоу, в котором нужно выбирать одну из трёх дверей, а затем — одну из двух ко-

робочек. За дверями находятся призы в 1000 р., 2000 р. и 3000 р. Вы не знаете, какой из призов получите, поэтому можете представить их как список.

Листинг 29.4 Недетерминированные результаты выбора двери

```
doorPrize :: [Int]
doorPrize = [1000, 2000, 3000]
```

Затем вы должны выбрать одну из двух коробочек, содержащих 500 р. или 20 000 р. Эти исходы также можно представить с помощью списка.

Листинг 29.5 Недетерминированные результаты выбора коробочки

```
boxPrize :: [Int]
boxPrize = [500, 20000]
```

Детерминированный контекст позволяет открыть только одну дверь и выбрать только одну коробочку, получив единственный возможный приз. Но если думать о задаче недетерминированно, то можно вычислить все возможные комбинации дверей и коробочек. С точки зрения детерминированных вычислений, говоря о призах, вы говорите об одном призе, который можете выиграть. На рис. 29.4 представлены детерминированный и недетерминированный подходы к вычислению призовых.

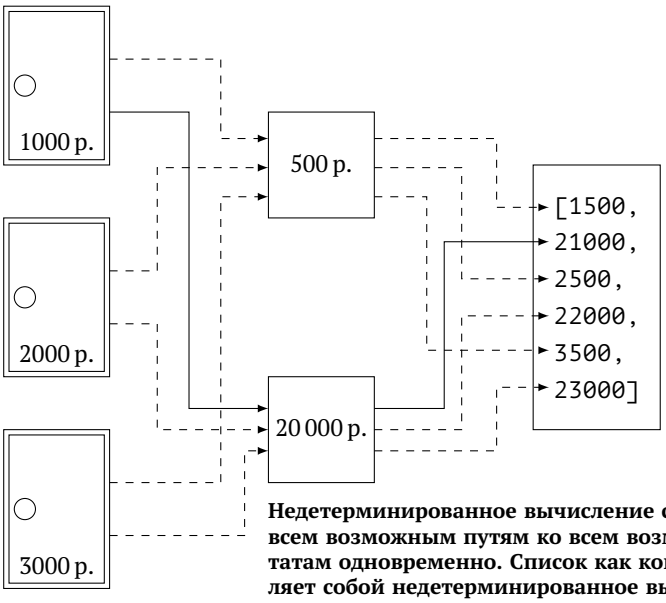


Рис. 29.4: Список как недетерминированное вычисление

Определение функции `totalPrize` в детерминированном мире выглядело бы следующим образом (для облегчения сравнения этой версии с версией для `Applicative` мы используем префиксную операцию `(+)`):

Листинг 29.6 Единственность детерминированного приза

```
totalPrize :: Int
totalPrize = (+) doorPrize boxPrize
```

В недетерминированном контексте мы говорим о всех возможных призах, которые только можно выиграть. Вы можете описать недетерминированную реализацию определения призовых сумм с помощью функции, представленной на рис. 29.5.

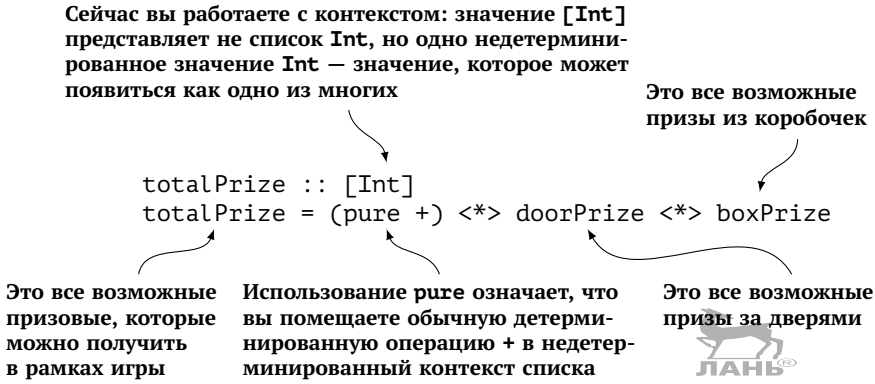


Рис. 29.5: Недетерминированное вычисление всех путей развития событий

В GHCi вы можете убедиться, что `totalPrize` теперь действительно представляет все возможные призы:

```
GHCi> totalPrize
[1500, 21000, 2500, 22000, 3500, 23000]
```

Когда вы складываете два списка в контексте, то получаете результаты всех возможных комбинаций. Для каждого приза за дверью вы можете выбрать один из двух возможных призов из коробочек. Результатом сложения двух списков в контексте списка будут все возможные решения недетерминированной задачи.

Далее мы рассмотрим ещё два примера недетерминированных вычислений. Мы применим их для вычисления простых чисел путём недетерминированного выявления всех составных. Затем с помощью недетерминированных вычислений мы будем генерировать множество всех возможных тестовых данных.

Проверка 29.4. Решить эту задачу, считая, что коробочки представляют собой мультипликатор призовых: при открытии первой коробочки призовые увеличиваются в 10 раз, а при открытии второй — в 50.

29.3.3. Генерация N первых простых чисел

Простым называется число, делящееся только на единицу и на само себя. Предположим, вам нужно сгенерировать список простых чисел. Благодаря тому что список является представителем `Applicative`, есть очень простой способ решения данной задачи.

Процесс решения можно описать следующим образом:

- начинаем со списка чисел от 2 до `n`;
- определяем все числа, не являющиеся простыми (то есть составные);
- отбираем все числа, не являющиеся составными.

Остаётся вопрос, как определять составные числа. *Составным* называется число, которое можно представить в качестве произведения двух или более различных других чисел. Вы можете легко получить этот список, умножая каждое число из списка `[2..n]` на числа из того же списка. Как это сделать? С помощью `Applicative`! Например, если у вас есть список `[2..4]`, вы можете использовать умножение `*`, `pure` и `<*>` для построения списка всех возможных чисел, которые могут быть получены из чисел, составляющих исходный список:

```
GHCi> pure (*) <*> [2..4] <*> [2..4]
[4, 6, 8, 6, 9, 12, 8, 12, 16]
```

Этот список не является оптимальным, так как включает числа, находящиеся за пределами исходного диапазона, к тому же он содержит повто-

Ответ 29.4

```
boxMultiplier = [10, 50]
newOutcomes = pure (*) <*> doorPrize <*> boxMultiplier

GHCi> newOutcomes
[10000, 50000, 20000, 100000, 30000, 150000]
```

ряющиеся значения. Но он также включает все возможные составные числа из вашего списка. Имея перед глазами этот код, вы легко можете написать функцию, перечисляющую все простые числа от 2 до n.

Листинг 29.7 Простое неэффективное перечисление простых чисел

```
primesToN :: Integer -> [Integer]
primesToN n = filter isNotComposite twoThroughN
  where twoThroughN = [2..n]
        composite = pure (*) <*> twoThroughN <*> twoThroughN
        isNotComposite = not . (`elem` composite)
```

Хотя этот алгоритм генерации простых чисел и не является эффективным, он очень прост в реализации и хорошо работает на достаточно маленьких диапазонах:

```
GHCi> primesToN 32
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```

Если вам когда-нибудь понадобится быстро написать генератор простых чисел, приведённый выше алгоритм может оказаться полезным.

29.3.4. Быстрая генерация большого количества тестовых данных

В предыдущем уроке мы показали, как можно создавать значение типа `User` в разных контекстах с помощью `Applicative`. Для моделирования пользователя компьютерной игры вы использовали следующий тип `User`:

```
data User = User {
     name :: String
   , gamerId :: Int
   , score :: Int
   } deriving Show
```

Класс `Applicative` использовался для создания значений `User` в контекстах `IO` и `Maybe`. Чтобы продемонстрировать мощь списка как контекста, сделаем то же самое для создания большого количества тестовых данных.

Предположим, у вас есть список `testNames` имён пользователей, среди которых встречаются как обычные, так и немного странные.

Листинг 29.8 Список имён testNames

```
testNames :: [String]
testNames = ["Джон Смит"
```

```
          , "Роберт'); DROP TABLE Students;--"
          , "Кристина NULL"
          , "Рэндалл Монро"]
```

Возможные ID игроков хранятся в `testIds`.

Листинг 29.9 Список ID testIds

```
testIds :: [Int]
testIds = [1337
          , 0123
          , 999999]
```

Заработанные игроками очки тоже хранятся в списке (в том числе имеются проблематичные случаи).

Листинг 29.10 Заработанные игроками очки testScores

```
testScores :: [Int]
testScores = [0
             , 100000
             , -99999]
```

Вы хотите сгенерировать тестовые данные, включающие все возможные комбинации этих значений. Это означает недетерминированное вычисление списка возможных пользователей. Вы могли бы сделать это вручную, но это означало бы выписывание `4 * 3 * 3 = 36` записей! Помимо этого, если позже вы бы решили добавить в эти списки некоторые значения, вам пришлось бы проделать много дополнительной работы.

Вместо этого можно воспользоваться свойствами класса `Applicative` для списка, позволяющими недетерминированно сгенерировать все тестовые данные. Вы сделаете это аналогично помещению `User` в контексты `IO` или `Maybe` из предыдущего урока.

Листинг 29.11 Генерация тестовых пользователей

```
testData :: [User]
testData = pure User <*> testNames
                     <*> testIds
                     <*> testScores
```

В GHCi вы можете увидеть, что успешно создали список из 36 возможных комбинаций этих значений. Более того, для обновления списка достаточно добавить любые необходимые значения в `testNames`, `testIds` или `testScores`:

```
GHCi> length testData
36
GHCi> take 3 testData
[User {name = "Джон Смит", gamerId = 1337, score = 0}
↳,User {name = "Джон Смит", gamerId = 1337, score = 100000}
↳,User {name = "Джон Смит", gamerId = 1337, score = -99999}]
```

Применение списков для выполнения недетерминированных вычислений демонстрирует мощь типа `Applicative` при работе с контекстами.

Проверка 29.5. Добавьте собственное имя к `testNames` и снова сгенерируйте данные. Сколько примеров будет содержать итоговый список пользователей?

Итоги

Целью этого урока было дать вам более глубокое понимание класса типов `Applicative`. Вы увидели его формальное определение, включающее операцию <*>, изученную в предыдущем уроке, и метод `pure`. Роль метода `pure` заключается в том, чтобы взять обычное значение любого типа и поместить его в нужный вам контекст; например, превратить `Int` в `Maybe Int`. Вы также фокусировались на различиях между контейнерами и контекстами, изучая список в качестве контекста. Контексты отличаются от контейнеров тем, что они требуют от вас понимания особенностей происходящих вычислений со значениями соответствующего типа. Список яв-

Ответ 29.5

```
testNames = ["Уилл Курт", "Джон Смит"
             , "Роберт'); DROP TABLE Students;--"
             , "Кристина NULL", "Рэндалл Монро"]

testData :: [User]
testData = pure User <*> testNames <*> testIds <*> testScores

GHCi> length testData
45
```

ляется не просто контейнером для данных, а представляет собой контекст недетерминированных вычислений. Давайте проверим, что вы всё поняли.

Задача 29.1. Чтобы доказать, что `Applicative` предоставляет больше возможностей, чем `Functor`, напишите универсальную версию `fmap` под названием `allFmap`, представляющую собой `fmap` для всех членов класса типов `Applicative`. Так как она должна работать со всеми экземплярами `Applicative`, можно пользоваться только его методами. Вот типовая аннотация вашей функции:

```
allFmap :: Applicative f => (a -> b) -> f a -> f b
```

Закончив реализацию функции, протестируйте её на типах списка и `Maybe`, являющихся представителями `Applicative`:

```
GHCi> allFmap (+1) [1, 2, 3]
[2, 3, 4]
GHCi> allFmap (+1) (Just 5)
Just 6
GHCi> allFmap (+1) Nothing
Nothing
```

Задача 29.2. Измените следующее выражение так, чтобы его результатом было значение типа `Maybe Int`. Сложность состоит в том, что вы не можете добавлять в код ничего, кроме `pure` и `<*>`. Вы не можете использовать конструктор `Just` или дополнительные скобки.

```
example :: Int
example = (*) ((+) 2 4) 6
```

Тип получившейся у вас функции должен выглядеть следующим образом:

```
exampleMaybe :: Maybe Int
```

Задача 29.3. Пользуясь приведёнными ниже данными и выполняя недетерминированные вычисления со списками, определите, сколько пива вам нужно купить, чтобы его наверняка хватило:

- вчера вы купили пиво, но не помните, была ли это упаковка из 6 или 12 бутылок;
- прошлой ночью вы с соседом выпили по две бутылки пива;
- сегодня к вам могут прийти двое или трое друзей, смотря как у них будет получаться;
- ожидается, что за ночь один человек выпьет три или четыре бутылки.

30

Введение в класс типов Monad

После прочтения урока 30 вы:

- разберётесь с ограничениями `Functor` и `Applicative`;
- научитесь применять операцию (>>=) из класса `Monad` для реализации последовательного вычисления функций в контексте;
- сможете писать код, работающий с вводом-выводом без do-нотации.

Вы только что закончили изучать два важных класса типов, `Functor` и `Applicative`. Каждый из них позволял вам выполнять очень мощные вычисления в контекстах вроде `Maybe` или `IO`. `Functor` позволяет вам изменять одиночные значения в контексте:

```
GHCi> (+ 2) <$> Just 3
Just 5
```

Класс `Applicative` увеличивает ваши возможности, позволяя использовать в контексте частичное применение. Это, в свою очередь, позволяет вам использовать функции нескольких аргументов в контексте:

```
GHCi> pure (+) <*> Just 3 <*> Just 2
Just 5
```

В этом уроке мы начнём разбираться с результатом эволюции данного процесса, классом типов `Monad`. Добавляя ещё одну операцию, класс типов `Monad` позволит вам выполнять любые произвольные вычисления в нужном вам контексте. Вы уже видели эту мощь в модуле 4, поскольку пользовались do-нотацией, которая представляет собой синтаксический сахар для методов класса типов `Monad`.

Листинг 30.1 Напоминание о do-нотации

```
main :: IO ()
main = do
  putStrLn "Помните о do-нотации!"
  putStrLn "Она многое упрощает!"
```

Чтобы разобраться с вопросом, зачем вам нужен класс типов `Monad`, в этом уроке мы будем игнорировать do-нотацию и увидим, почему вообще нужны монады, особенно с учётом выразительной силы классов `Functor` и `Applicative`. Поскольку do-нотация делает жизнь проще, мы вернёмся к ней уже в следующем уроке.

Вы начнёте этот урок с рассмотрения двух относительно простых задач, которые не просто решить с помощью `Functor` или `Applicative`. Затем вы узнаете о мощной операции `>>=` (часто произносится как *баинд*, от англ. *bind*) из `Monad` и как она может упростить решение этих задач. Этот урок завершится применением методов `Monad` для написания `IO`-действий, похожих на те, что использовали do-нотацию в уроке 4.

Обратите внимание. Как вы могли бы написать `IO`-действие, считывающее введённое пользователем число и печатающее его обратно, без do-нотации?

30.1. Ограничения Applicative и Functor

Давайте вернёмся к визуальному представлению наших классов типов `Functor`, `Applicative` и `Monad`. Вы решили три из четырёх возможных случаев несоответствий, попробуем их все сейчас вспомнить. Функция `fmap` из `Functor` предоставляет адаптер для случая, когда у вас есть значение в контексте и обычная функция, а нужен результат в контексте. Соответствующий случай проиллюстрирован на рис. 30.1.

Далее, операция `<*>`, определённая в классе типов `Applicative`, позволяет соединить функцию в контексте со значениями в контексте, данная

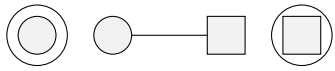


Рис. 30.1: Несоответствие между функцией и контекстом, разрешаемое классом `Functor`

ситуация демонстрируется на рис. 30.2.

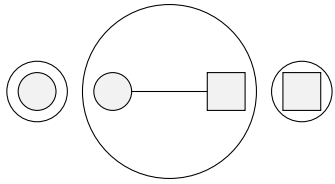


Рис. 30.2: Операция <*> из Applicative решает проблему нахождения функций в контексте

ЛАНЬ

И наконец, метод pure из Applicative позволяет обрабатывать случай, когда итоговый результат не находится в контексте (см. рис. 30.3).

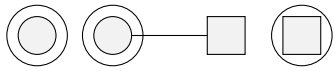


Рис. 30.3: Метод pure из Applicative помещает значение в контекст

Осталось решить одну задачу, когда первоначальный аргумент не находится в контексте, а результат находится (см. рис. 30.4).

ЛАНЬ

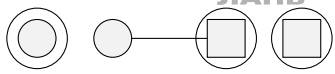


Рис. 30.4: Не решённое пока несоответствие: класс Monad даст ответ

Когда у вас будет решение для этого случая, то будет решение вообще для любой возможной функции в контексте. Этот последний случай может казаться странным, но он встречается довольно часто. Далее разберёмся с двумя примерами, когда он возникает, и выясним, почему Functor и Applicative не могут здесь ничем помочь.

30.1.1. Комбинирование поиска по двум Map

В этом разделе вы рассмотрите частую ситуацию, когда необходимо искать значение в одном Map, чтобы получить доступ к значению во втором. Это может произойти в любой момент, когда вам нужно одно значение, чтобы найти другое, например:

- поиск города по почтовому индексу, поиск области по городу;
- использование имени сотрудника для поиска его ID и использование ID для поиска личного дела;

- поиск компании по биржевому имени, а затем поиск её адреса.

Вы будете писать код для управления пользовательскими данными для мобильной игровой платформы. На данный момент каждый пользователь определён как уникальный `GamerId` — просто `Int`. Предположим, что прошлые версии вашей программы использовали уникальный `UserName` типа `String`, чтобы ассоциировать пользователя с данными в его профиле. Из-за этой устаревшей зависимости от имени пользователя как идентификатора для поиска пользовательских данных новых пользователей вам всё равно придётся искать имя пользователя с помощью `GamerId`, а затем применять `UserName` для поиска данных в пользовательской учётной записи. На следующем листинге представлены необходимые типы и данные.

Листинг 30.2 Типы и данные для задачи поиска по двум Map

```
import qualified Data.Map as Map

type UserName = String
type GamerId = Int
type PlayerCredits = Int

userNameDB :: Map.Map GamerId UserName
userNameDB = Map.fromList [(1,"nYarlathoTep")
                          ,(2,"KINGinYELLOW")
                          ,(3,"dagon1997")
                          ,(4,"rcarter1919")
                          ,(5,"xCTHULHUx")
                          ,(6,"yogSOThoth")]

creditsDB :: Map.Map UserName PlayerCredits
creditsDB = Map.fromList [("nYarlathoTep",2000)
                         ,("KINGinYELLOW",15000)
                         ,("dagon1997",300)
                         ,("rcarter1919",12)
                         ,("xCTHULHUx",50000)
                         ,("yogSOThoth",150000)]
```

Это база данных для поиска UserName по GamerId

Это база данных для поиска PlayerCredits по UserName

Теперь всё готово для начала работы над главной задачей — написания функции для поиска пользовательских данных по `GamerId`. Вы хотите получить функцию, которая будет искать `PlayerCredits` по `GamerId`. Вы всё ещё хотите, чтобы значение `PlayerCredits` имело тип `Maybe PlayerCredits`, потому что вполне возможно, что у вас отсутствует `GamerId` или для вашего `GamerId` отсутствует запись в `creditsDB`. Функция, которую вы хотите реализовать, должна иметь следующий тип.

Листинг 30.3 Тип целевой функции creditsFromId

```
creditsFromId :: GamerId -> Maybe PlayerCredits
```

Для создания этой функции вам необходимо скомбинировать два вызова функции `Map.lookup`. Определим вспомогательные функции, которые позволят абстрагироваться от наших баз данных. Функция `lookupUserName` будет принимать `GamerId` и возвращать `Maybe UserName`, а `lookupCredits` будет принимать `UserName` и возвращать `Maybe PlayerCredits`.

Листинг 30.4 Функции lookupUserName и lookupCredits

```
lookupUserName :: GamerId -> Maybe UserName
lookupUserName id = Map.lookup id userNameDB

lookupCredits :: UserName -> Maybe PlayerCredits
lookupCredits username = Map.lookup username creditsDB
```

Прежде чем продолжать, следует подумать о типе требуемой функции. Нужно соединить результат вызова `lookupUserName` типа `Maybe UserName` с функцией `lookupCredits` типа `UserName -> Maybe PlayerCredits`, поэтому типовая аннотация вашей функции выглядит следующим образом:

```
Maybe UserName -> (UserName -> Maybe PlayerCredits)
               -> Maybe PlayerCredits
```

Классы `Applicative` и `Functor` научили вас думать о решении задач более абстрактно, нежели в типах вроде `Maybe`. Основная форма комбинирующей функции, которую вы хотите, выглядит следующим образом:

```
Applicative f => f a -> (a -> f b) -> f b
```

Мы указали ограничение `Applicative` вместо `Functor`, потому что `Applicative` мощнее. Если вы не сможете решить задачу с его помощью, то не сможете и при помощи `Functor`. Теперь давайте посмотрим на инструменты, которые вы получаете от `Applicative` и `Functor`:

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
pure :: Applicative f => a -> f a
```

К сожалению, при всей той силе, которую вы получили от `Applicative`, не похоже, чтобы какая бы то ни было комбинация этих инструментов решит эту довольно простую задачу соединения двух функций. Впрочем, вы

можете решить эту задачу, написав обёртку для `lookupCredits`, чтобы она была функцией вида `Maybe UserName -> Maybe PlayerCredits`.

Листинг 30.5 Решение задачи без Functor или Applicative

```
altLookupCredits :: Maybe UserName -> Maybe PlayerCredits
altLookupCredits Nothing = Nothing
altLookupCredits (Just username) = lookupCredits username
```

Теперь можно собрать окончательную версию функции `creditsFromId`.

Листинг 30.6 Функция типа GamerId -> Maybe PlayerCredits

```
creditsFromId :: GamerId -> Maybe PlayerCredits
creditsFromId id = altLookupCredits (lookupUserName id)
```

В GHCi вы можете убедиться, что это отлично работает:

```
GHCi> creditsFromId 1
Just 2000
GHCi> creditsFromId 100
Nothing
```

Однако необходимость написания обёртки для работы с `Maybe` должна настораживать. В уроке 28 вы видели похожую ситуацию, когда необходимость написания функции-обёртки для работы в контексте служила мотивацией для введения более мощного класса типов. Но в данный момент вы не можете быть уверены в необходимости того или иного, даже более мощного класса типов.

Проверка 30.1. Интересно, что следующая функция, похоже, делает то, что вам нужно, и компилируется как надо. Что с ней не так? (Подсказка: посмотрите в GHCi её тип.)

```
creditsFromIdStrange id = pure lookupCredits <*>
                          lookupUserName id
```

Ответ 30.1. Проблема этой функции — в возвращении значения типа `Maybe (Maybe PlayerCredits)`, то есть с вложенными `Maybe`!

30.1.2. Написание не столь тривиального IO-действия echo

Причина лёгкости решения этой задачи для Maybe состоит в том, что Maybe — контекст, с которым просто работать. Вы всегда можете решить любую задачу с Maybe вручную при помощи аккуратного сопоставления с образцами Just и Nothing. Тип IO, напротив, не столь дружелюбен. Чтобы это показать, давайте решим несложную, казалось бы, задачу. Вы хотите написать простое IO-действие echo, которое считывает пользовательский ввод и тут же печатает его обратно. Чтобы сделать это, вам нужно скомбинировать два IO-действия, которые вы уже хорошо знаете:

```
getLine :: IO String
putStrLn :: String -> IO ()
```

ЛАНЬ

Понятно, что тип функции echo выглядит как `echo :: IO ()`. Вам нужно скомбинировать getLine с putStrLn. Если вы ещё раз подумаете об этой задаче в типах, то увидите знакомую ситуацию. Вам нужна функция, которая комбинирует getLine и putStrLn и возвращает `IO String`:

```
IO String -> (String -> IO ()) -> IO ()
```

Если абстрагироваться от IO, то получится следующее:

```
Applicative f => f a -> (a -> f b) -> f b
```

Это именно тот тип, к которому вы уже приходили. Для решения этой задачи вам нужно что-то более мощное, чем Functor и Applicative. Эта проблема приводит нас к классу типов Monad!

Проверка 30.2. Почему для решения этой задачи нельзя написать функции вроде altLookupCredits и creditsFromId из предыдущего пункта?

Ответ 30.2. У вас нет способа получать значение без контекста IO, как вы делали с контекстом Maybe. Вам нужны более мощные, чем Applicative и Functor, инструменты для работы с типами IO.

30.2. Операция (>>=)

Операция, которой вам не хватает, — это >>=, вот её тип:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Как видите, (>>=) имеет именно тот тип, который нам нужен! Из ограничений на классы типов вы можете увидеть, что >>= — метод класса типов Monad. И Maybe, и IO реализуют экземпляры Monad, а значит, вы можете использовать >>= для решения своей задачи. Эта операция позволяет записать более элегантное решение для функции creditsFromId.

Листинг 30.7 Новая версия функции creditsFromId (на основе >>=)

```
creditsFromId :: GamerId -> Maybe PlayerCredits
creditsFromId id = lookupUserName id >>= lookupCredits
```

Видно, что >>= позволяет соединять функции вида (a -> m b). В случае с Maybe это означает, что вы можете соединить сколько угодно операций поиска. Предположим, что у вас есть ещё один уровень перенаправления. Представьте, что ваша игровая компания была куплена ООО «УиллКо», и теперь каждый GamerId будет ассоциироваться с WillCoId.

Листинг 30.8 Ещё один Map в поиске пользовательских данных

```
type WillCoId = Int

gamerIdDB :: Map.Map WillCoId GamerId
gamerIdDB = Map.fromList [(1001,1), (1002,2), (1003,3),
                          (1004,4), (1005,5), (1006,6)]

lookupGamerId :: WillCoId -> Maybe GamerId
lookupGamerId id = Map.lookup id gamerIdDB
```

Теперь вам требуется новая функция creditsFromWCId, тип которой — WillCoId -> Maybe PlayerCredits. Вы легко можете определить её, связывая все три свои функции поиска при помощи >>=.

Листинг 30.9 Связывание трёх функций поиска с помощью >>=

```
creditsFromWCId :: WillCoId -> Maybe PlayerCredits
creditsFromWCId id = lookupGamerId id >>=
                     lookupUserName >>=
                     lookupCredits
```

В GHCi можно убедиться, что всё работает в соответствии с ожиданиями:

```
GHCi> creditsFromWCId 1001
Just 2000
GHCi> creditsFromWCId 100
Nothing
```

Хотя использование >>= упростило связывание функций для Maybe, важно решить исходную задачу с IO-действием. Мы остановились, когда хотели соединить getLine и putStrLn. Тогда не было никакого способа «вскрыть» тип IO, чтобы написать обёртку, аналогичную решению для Maybe. С >>= определение функции echo тривиально. Давайте поместим его в файл echo.hs и посмотрим, как это работает.

Листинг 30.10 Использование >>= в определении функции echo

```
echo :: IO ()
echo = getLine >>= putStrLn

main :: IO ()
main = echo
```

```
$ ghc echo.hs
$ ./echo
Hello World!
Hello World!
```

Проверка 30.3. Соедините эти две функции в одно IO-действие:

```
readInt :: IO Int
readInt = read <$> getLine
printDouble :: Int -> IO ()
printDouble n = print (n*2)
```

Операция >>= — это сердце класса типов Monad. При своей относительной простоте >>= — это последняя часть вашего пазла. Теперь, когда у вас есть <$>, <*>, pure и >>=, вы можете соединить в контексте любые вычисления, которые вам могут понадобиться.

Ответ 30.3. Результат соединения: readInt >>= printDouble.

30.3. Класс типов Monad

Как класс типов `Applicative` расширяет мощность `Functor`, так и класс `Monad` расширяет `Applicative` (см. рис. 30.5).

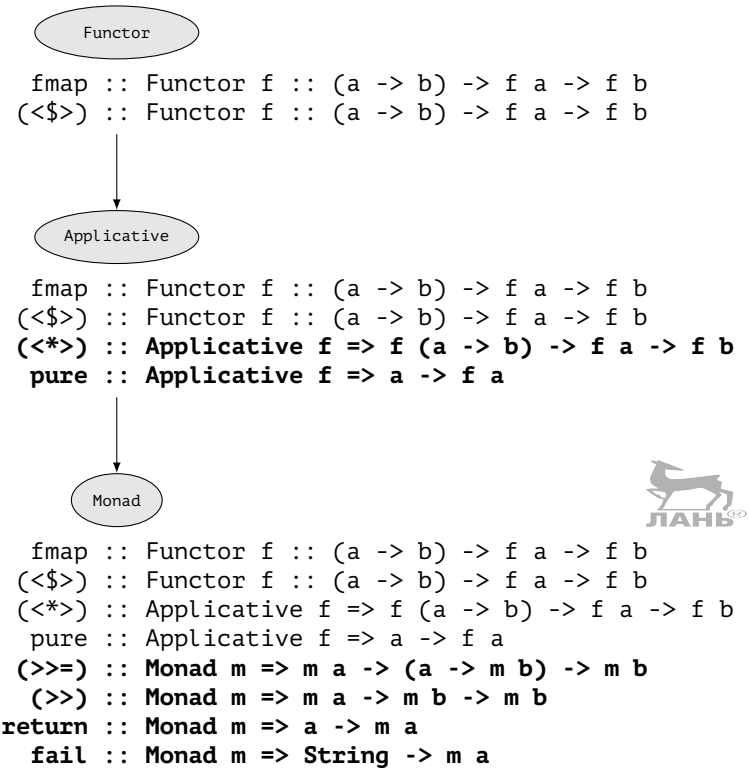


Рис. 30.5: Соотношение между классами `Functor`, `Applicative` и `Monad`

Вот определение класса типов `Monad`:

```
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a
```

В этом определении имеется четыре важных метода. Единственный метод, необходимый для минимально полного определения, — это `>>=`.

Вы уже видели, что >>= позволяет связывать последовательность функций, которые отображают обычное значение на значение в контексте. Метод `fail` обрабатывает ошибки, возникающие при работе с `Monad`. Например, для `Maybe` `fail` вернёт `Nothing`, а для `IO` выбросит ошибку ввода-вывода. Вы рассмотрите `fail` более подробно в модуле 7, когда мы будем обсуждать обработку ошибок в Haskell. Осталось рассмотреть только >> и `return`.

Метод `return` выглядит до боли знакомо. Если вы сравните его с `pure`, то увидите, что они практически идентичны:

```
pure :: Applicative f => a -> f a
return :: Monad m => a -> m a
```

Единственное отличие в том, что `pure` имеет ограничение класса типов `Applicative`, в то время как `return` ограничен классом типов `Monad`. Оказывается, `pure` и `return` действительно идентичны и имеют разные имена только по историческим причинам. Класс `Monad` появился в Haskell раньше `Applicative`, поэтому метод `return` существует только по причинам обратной совместимости. Поскольку каждый экземпляр `Monad` должен быть экземпляром `Applicative`, то может показаться уместным всюду использовать `pure` вместо `return`, ведь `pure` сработает везде, где сработает `return`. Но это не совсем так. При работе в контексте `Monad` всё-таки лучше использовать `return`[1].

Остаётся разобраться с операцией >>. Если посмотреть на неё внимательно, то выяснится, что у >> довольно странный тип (см. рис. 30.6).

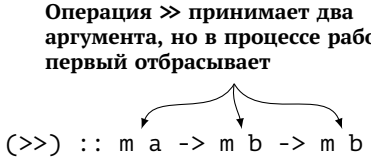


Рис. 30.6: Операция >> странного типа для монад с побочными эффектами

Похоже, эта операция отбрасывает первое значение типа `m a`. Действительно, именно это она и делает. Почему вам это может быть нужно? Отчасти это полезно при работе с контекстами, имеющими побочные эффекты, вроде `IO` (есть и другие, мы обсудим их в модуле 7). На данный момент единственный контекст такого рода, который вы видели, — это `IO`.

[1] В сообществе Haskell-разработчиков есть разные мнения на этот счёт: в частности, имеются планы по объявлению метода `return` устаревшим и даже его удалению из класса типов `Monad`. По этой причине применять всюду `pure` может быть безопаснее с точки зрения долговременной поддержки кодовой базы. — *Прим. ред.*

Когда вы используете `putStrLn`, вы ничего не получаете обратно. Зачастую вы хотите напечатать что-то пользователю и просто вернуть результат `IO ()`. Например, вы можете модифицировать программу echo.hs так, чтобы пользователь знал, что она делает.

Листинг 30.11 Преимущества >> в разговорчивой версии echo

```
echoVerbose :: IO ()
echoVerbose = putStrLn "Введите строку, а мы её повторим!" >>
              getLine >>= putStrLn

main :: IO ()
main = echoVerbose
```

При работе с `IO` операция `>>` полезна, если вам нужно выполнить `IO`-действие, результата которого не представляет для вас интереса.

30.3.1. Класс типов Monad и программа-приветствие

Чтобы продемонстрировать, как связать всё это вместе, давайте напишем простое `IO`-действие, которое будет спрашивать имя пользователя, а затем выводить строку `"Привет, <имя>!"`. Для этого вам понадобится соединить несколько простых функций. Первая — `IO`-действие, спрашивающее имя, — это просто `putStrLn` с запросом.

Листинг 30.12 IO-действие askForName

```
askForName :: IO ()
askForName = putStrLn "Как вас зовут?"
```

Следующее `IO`-действие, которое вам нужно использовать, — это функция `getLine`. После этого нужно взять результат `getLine` и сформировать строку `"Привет, <имя>!"`. Это обычная функция типа `String -> String`.

Листинг 30.13 Функция на строках nameStatement

```
nameStatement :: String -> String
nameStatement name = "Привет, " ++ name ++ "!"
```

Теперь нужно отправить результат в `putStrLn`, и ваше действие закончено. Начнём со связывания `askForName` и `getLine` посредством `>>`, так как результат вас не интересует:

```
(askForName >> getLine)
```

Следующая часть хитрее. Теперь у вас есть значение `IO String`, но вам нужно связать его с `nameStatement`, которая имеет тип `String -> String`. Вы можете использовать для этого `>>=`, но тогда `nameStatement` будет возвращать `IO String`. Вы можете переписать `nameStatement`, но более частое решение — это обернуть `nameStatement` в лямбда-функцию и использовать в конце `return`. Благодаря выводу типов Haskell знает, в какой контекст нужно поместить типы (см. рис. 30.7).

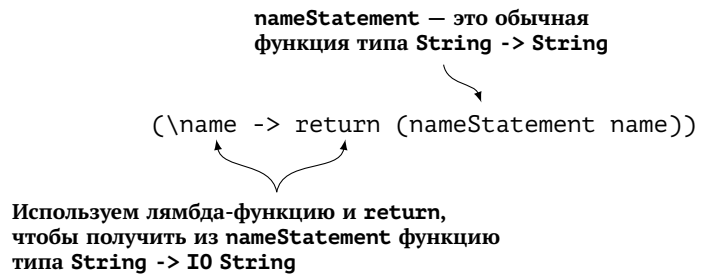


Рис. 30.7: Лямбда-функция для преобразования типа `a -> a` в `a -> m a`

Проверка 30.4. Преобразуйте `(+ 2)` из типа `Num a => a -> a` в тип `Num a => a -> IO a`, используя лямбда-функцию и `return`. Воспользуйтесь командой GHCi `:t` и убедитесь, что получили верный тип.

Вот ваша программа на данный момент:

```
(askForName >> getLine) >>=
(\name -> return (nameStatement name))
```

Для её завершения воспользуемся `>>=` и передадим результат в функцию `putStrLn`. Вот окончательная версия требуемой программы.

Листинг 30.14 Программа-приветствие с методами класса Monad

```
helloName :: IO ()
helloName = askForName >>
            getLine >>=
            (\name -> return (nameStatement name)) >>=
            putStrLn
```

Ответ 30.4. Лямбда-выражение: `(\n -> return ((+ 2) n))`.

Теперь можно либо добавить функцию `main`, либо использовать для тестов GHCi. Вот результаты в GHCi:

```
GHCi> helloName
Как вас зовут?
Уилл
Привет, Уилл!
```

В классе `Monad` приятно то, что для решения этой задачи вы смогли связать все свои функции и действия относительно просто. Плохо то, что вам пришлось добавить функции, работающие с `IO`, вроде обёртки над `nameStatement`, да и вообще все эти лямбды могут немного раздражать. Кроме того, возня со всеми этими операциями может сбивать с толку. В следующем уроке вы увидите, что do-нотация является синтаксическим сахаром над методами класса типов `Monad`.

Итоги

В этом уроке нашей целью было знакомство с классом типов `Monad`. Класс типов `Monad` представляет собой уточнение понятия вычислений в контексте, которое вы начали осваивать с класса `Functor`. Самый важный метод `Monad` — это операция `>>=`. Вы можете использовать `>>=` для связывания функций типа `(a -> m b)`. Это особенно важно для работы с `IO`, где, в отличие от `Maybe`, вы не можете для получения доступа к значениям в контексте использовать сопоставление с образцом. Класс типов `Monad` делает программирование ввода-вывода возможным. Давайте проверим, как вы это поняли.

Задача 30.1. Чтобы доказать, что `Monad` строго мощнее `Functor`, напишите универсальную версию `<$>` под названием `allFmapM`, которая определит `<$>` для всех членов класса типов `Monad`. Так как она должна работать для всех экземпляров `Monad`, вы можете использовать только методы из определения класса `Monad` (и лямбда-функции). Вот тип этой функции:

```
allFmapM :: Monad m => (a -> b) -> m a -> m b
```

Задача 30.2. Чтобы доказать, что `Monad` строго мощнее `Applicative`, напишите универсальную версию `<*>` под названием `allApp`, которая определит `<*>` для всех членов класса типов `Monad`. Так как она должна работать для всех экземпляров `Monad`, вы можете использовать только методы из определения класса `Monad` (и лямбда-функции). Вот тип этой функции:

```
allApp :: Monad m => m (a -> b) -> m a -> m b
```

Это задание гораздо хитрее предыдущего. Дадим две подсказки:

- попробуйте рассуждать исключительно в терминах типов;
- если хочется, примените <$>, заменив её своим ответом к задаче 29.1.

Задача 30.3. Реализуйте аналогичную >>= функцию для `Maybe`:

```
bind :: Maybe a -> (a -> Maybe b) -> Maybe b
```

ЛАНЬ

31 Облегчение работы с монадами с помощью do-нотации

После прочтения урока 31 вы:

- будете использовать do-нотацию для упрощения работы с монадами;
- сможете преобразовывать функции и лямбда-функции, работающие с монадами, в do-нотацию;
- научитесь обобщать код для одного экземпляра `Monad` на все монады.

Класс типов `Monad` является мощной абстракцией для работы с типами в контексте. Но использование методов `>>=`, `>>` и `return` класса `Monad` быстро приводит к написанию громоздкого, трудночитаемого кода. В этом уроке вы рассмотрите два средства, существенно облегчающих работу с монадами. Первым из них является работающая за кадром do-нотация, которую вы уже использовали в 4-м модуле. В этом уроке вы поймёте принципы её работы. Затем вы узнаете, как в качестве монады работает список. Это позволит вам начать работать с ещё одной абстракцией над монадами, сильно упрощающей жизнь, — генераторами списков. Хотя понимать методы класса `Monad` важно, на практике работа с монадами будет происходить с использованием этих приёмов для упрощения кода.

В предыдущем уроке вы написали реализацию действия ввода-вывода `helloName`, которое спрашивало у пользователя его имя и приветствовало его. Вот полученный тогда код:

Листинг 31.1 Программа-приветствие

```
askForName :: IO ()
askForName = putStrLn "Как тебя зовут?"
```

```
nameStatement :: String -> String
nameStatement name = "Привет, " ++ name ++ " !"

helloName :: IO ()
helloName = askForName >>
            getLine >>=
            (\name ->
              return (nameStatement name)) >>=
            putStrLn
```

В реализации этого действия вы использовали методы класса `Monad`. Напомним их:

- `>>` позволяет выполнить действие ввода-вывода и связать его со следующим действием, проигнорировав результат первого действия;
- `>>=` позволяет выполнить действие ввода-вывода и передать его результат в следующую функцию, ожидающую значения на вход;
- выражение `(\x -> return (func x))` позволяет вам взять обычную функцию и заставить её работать в контексте `IO`.

Хорошей новостью является то, что у вас теперь есть возможность делать всё, что захочется, в таких контекстах, как `IO` или `Maybe`. К сожалению, этот код выглядит запутанным, его трудно читать и расширять. К счастью, в Haskell есть отличное решение этой проблемы!

Обратите внимание. Напишите программу, которая с помощью средств класса типов `Monad` принимает пару значений в контексте и возвращает наибольшее из них. Тип функции должен выглядеть следующим образом:

```
maxPairM :: (Monad m, Ord a) => m (a, a) -> m a
```

Ваша функция должна работать с `IO (a,a)`, `Maybe (a,a)` и `[(a,a)]`.

31.1. Возвращаемся к do-нотации

Оказывается, вы уже сталкивались со средством, позволяющим сделать ваш код чище, — с do-нотацией. Она представляет собой синтаксический сахар для замены `>>`, `>>=` и `(\x -> return (func x))`. Ниже приведён предыдущий пример с использованием do-нотации.

Листинг 31.2 Переписывание helloName с помощью do-нотации

```
helloNameDo :: IO ()
helloNameDo = do
   askForName
   name <- getLine
   putStrLn (nameStatement name)
```

На рис. 31.1 представлена аннотированная версия этого преобразования.

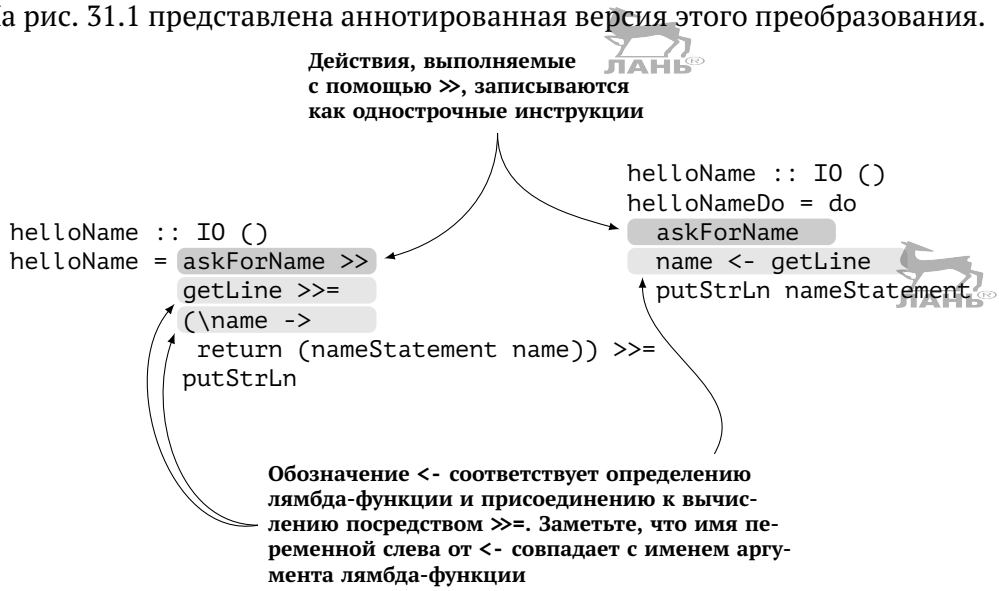


Рис. 31.1: Соответствие между методами класса Monad и do-нотацией

Полезным упражнением является перевод из do-нотации в монадические операции и наоборот. В модуле 4 вы писали следующую простую программу для вывода приветствия.

Листинг 31.3 Программа, использующая do-нотацию

```
helloPerson :: String -> String
helloPerson name = "Привет," ++ " " ++ name ++ "!"

main :: IO ()
main = do
   name <- getLine
   let statement = helloPerson name
   putStrLn statement
```

Процесс «рассахаривания» действия ввода-вывода main из этой про-

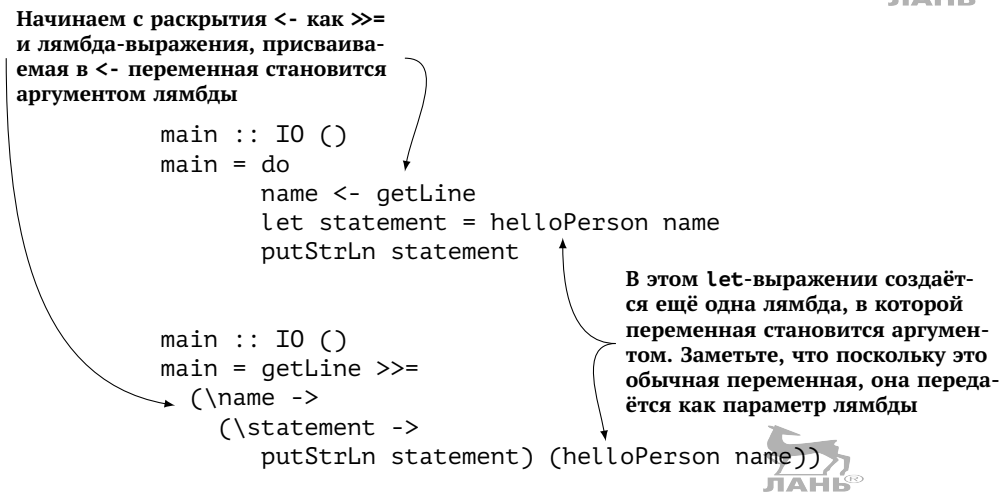


Рис. 31.2: Рассахаривание do-нотации

граммы представлен на рис. 31.2. Если вы испытываете проблемы с переходом от `let` и `<-` к лямбда-выражениям, стоит пересмотреть урок 3 модуля 1. По причинам, которые должны быть очевидны, использование do-нотации строго предпочтительно по отношению к использованию монадических операций. Но в простых случаях, таких как функция `echo`, использование `>>=` часто оказывается проще, чем применение do-нотации.

Листинг 31.4 IO-действие, для которого >>= предпочтительнее

```
echo :: IO ()
echo = getLine >>= putStrLn
```

Проверка 31.1. Перепишите `echo` с использованием do-нотации.

Ничего страшного, если во время обучения вы будете испытывать затруднения с переводом из do-нотации в монадические операции и наоборот. Однако важно понять, что с do-нотацией не связано никакой магии.

Ответ 31.1

```
echo :: IO ()
echo = do
  val <- getLine
  putStrLn val
```

31.2. Использование кода в разных контекстах и do-нотация

В модуле 4, где вы впервые увидели do-нотацию, мы слегка коснулись идеи, что мощь класса типов `Monad` позволяет вам создавать различные программы, используя одинаковый код в разных контекстах. Пример, который был приведён в уроке 21: создание программы ввода-вывода, запрашивающей у пользователя информацию для сравнения стоимости двух пицц. Так как do-нотация работает со всеми монадами, вы смогли тривиальным образом преобразовать эту программу для работы с типом `Maybe` в случае, когда значения поступали из `Data.Map` вместо `IO`.

Чтобы более полно продемонстрировать идею того, что `Monad` позволяет вам с лёгкостью переиспользовать один и тот же код в разных контекстах, рассмотрим серию примеров использования одного кода в разных контекстах. Основной задачей будет обработка данных о кандидатах для приёма на работу в вашу компанию. Нужно будет определить, прошли или провалили они собеседование. Вы увидите, как один и тот же код может обрабатывать данные о кандидатах в контекстах `IO`, `Maybe` и даже в списках. В конце вы сможете вынести код, использованный в каждой из секций, в общую функцию, работающую со всеми экземплярами класса `Monad`.

31.2.1. Постановка задачи

Для начала смоделируем данные о кандидатах. Каждому из них будет соответствовать уникальный ID. Во время собеседования кандидаты проходят код-ревью и проверку на соответствие корпоративной культуре. За каждый из этапов собеседования кандидат получает оценку.

Листинг 31.5 Тип данных Grade для оценки этапов собеседования

```
data Grade = F | D | C | B | A
  deriving (Eq, Ord, Enum, Show, Read)
```

В связи с тем, что есть несколько исследовательских вакансий, вы также будете учитывать уровень образования кандидата, а для некоторых должностей будет предусмотрен минимально допустимый уровень.

Листинг 31.6 Тип данных Degree для уровня образования

```
data Degree = HS | BA | MS | PhD
  deriving (Eq, Ord, Enum, Show, Read)
```

В итоге модель данных Candidate выглядит следующим образом.

Листинг 31.7 Представление результатов собеседования

```
data Candidate = Candidate
   { candidateId :: Int
   , codeReview :: Grade
   , cultureFit :: Grade
   , education :: Degree } deriving Show
```

Важная задача — проверить, соответствует ли кандидат минимальным требованиям. Если да, вы отправляете данные о нём в комитет, рассматривающий заявки. Ниже приведена реализация функции viable, проверяющей соответствие кандидата установленным минимальным требованиям.

Листинг 31.8 Проверка на соответствие минимальным требованиям

```
viable :: Candidate -> Bool
viable candidate = all (== True) tests
  where passedCoding = codeReview candidate > B
        passedCultureFit = cultureFit candidate > C
        educationMin = education candidate >= MS
        tests = [passedCoding, passedCultureFit, educationMin]
```

Далее вы рассмотрите три контекста, в которых вам может понадобиться проверять, соответствует ли кандидат минимальным требованиям.

Проверка 31.2. Создайте значение типа Candidate и проверьте его с помощью функции viable.

Ответ 31.2

```
testCandidate :: Candidate
testCandidate = Candidate
   { candidateId = 1
   , codeReview = A
   , cultureFit = A
   , education = PhD}

GHCi> viable testCandidate
True
```

31.2.2. Контекст IO: создание утилиты командной строки

Давайте начнём с разработки утилиты командной строки, чтобы пользователь мог вручную вводить данные о кандидатах. Эта задача похожа на те, что вам доводилось решать в модуле 4. Единственное различие состоит в том, что там вы работали с do-нотацией как с чем-то магическим. Для начала вам потребуются простые действия ввода-вывода для считывания `Int`, `Grade` и `Degree`. Для их реализации вы можете воспользоваться do-нотацией, но этот случай является отличным примером ситуации, когда операция >>= оказывается удобнее. Каждое из этих действий должно совместить результат `getLine` с преобразованием строки к нужному типу данных и возвратом соответствующего результата в контексте `IO`.

Листинг 31.9 Считывание данных о кандидате

```
readInt :: IO Int
readInt = getLine >>= (return . read)

readGrade :: IO Grade
readGrade = getLine >>= (return . read)

readDegree :: IO Degree
readDegree = getLine >>= (return . read)
```

Имея эти вспомогательные функции, вы можете создать IO-действие для считывания данных о кандидате. Вам лишь нужно добавить вывод пользователю информации о порядке считываемых данных. Использование здесь do-нотации очень похоже на решение задачи из модуля 4, так что следующий код должен выглядеть знакомо.

Листинг 31.10 Функция readCandidate

```
readCandidate :: IO Candidate
readCandidate = do
 putStrLn "введите id:"
 cId <- readInt
 putStrLn "введите оценку код-ревью:"
 codeGrade <- readGrade
 putStrLn "введите оценку соответствия корпоративной культуре"
 cultureGrade <- readGrade
 putStrLn "введите уровень образования:"
 degree <- readDegree
 return (Candidate { candidateId = cId, codeReview = codeGrade
                   , cultureFit = cultureGrade
                   , education = degree })
```

Логика программы зашита в IO-действие `assessCandidateIO`, считывающее данные о кандидате, проверяющее их соответствие минимальным требованиям и возвращающее строку `"прошёл"`, если кандидат соответствует требованиям, и `"провалился"`, если нет. Написать код этого действия, используя do-нотацию, несложно.

Листинг 31.11 Действие ввода-вывода для проверки кандидата

```
assessCandidateIO :: IO String
assessCandidateIO = do
  candidate <- readCandidate
  let passed = viable candidate
  let statement = if passed
                  then "прошёл"
                  else "провалился"
  return statement
```

Вы можете поместить этот код в `main`, скомпилировать свою программу и запустить её, но в данном случае проще воспользоваться GHCi:

```
GHCi> assessCandidateIO
введите id:
1
введите оценку код-ревью:
A
введите оценку соответствия корпоративной культуре:
B
введите уровень образования:
PhD
"прошёл"
```

Благодаря классу типов `Monad` вы смогли взять тип `Candidate`, не проектировавшийся специально для `IO`, и поместить его в нужный контекст.

Проверка 31.3. Перепишите `readGrade` с применением do-нотации.

Ответ 31.3

```
readGradeDo :: IO Grade
readGradeDo = do
  input <- getLine
  return (read input)
```

31.2.3. Контекст Maybe: работаем с Map кандидатов

Вводить данные о кандидатах вручную с помощью командной строки утомительно. В следующем примере для хранения и поиска информации о кандидатах вы будете использовать `Data.Map`. Для начала вам потребуется несколько кандидатов и база данных, составленная из данных о них.

Листинг 31.12 Несколько кандидатов

```
candidate1 :: Candidate
candidate1 = Candidate { candidateId = 1, codeReview = A
                       , cultureFit = A, education = BA}

candidate2 :: Candidate
candidate2 = Candidate { candidateId = 2, codeReview = C
                       , cultureFit = A, education = PhD}

candidate3 :: Candidate
candidate3 = Candidate { candidateId = 3, codeReview = A
                       , cultureFit = B, education = MS}
```

Листинг 31.13 База данных о кандидатах

```
candidateDB :: Map.Map Int Candidate
candidateDB = Map.fromList [ (1, candidate1), (2, candidate2)
                           , (3, candidate3)]
```

Вам снова нужно оценивать кандидатов и возвращать соответствующие строки. На этот раз вы работаете с `candidateDB`. Так как каждый поиск возвращает тип `Maybe`, вы столкнётесь с проблемой — контекст отличается от `IO`, с которым вы работали в прошлом случае. В предыдущем примере вам нужно было взаимодействовать с пользователем, а в этот раз нужно работать с потенциально отсутствующими значениями, поэтому вам понадобится функция, похожая на `assessCandidateIO`, но работающая с `Maybe`.

Листинг 31.14 Оценка кандидатов в контексте Maybe

```
assessCandidateMaybe :: Int -> Maybe String
assessCandidateMaybe cId = do
  candidate <- Map.lookup cId candidateDB
  let passed = viable candidate
  let statement = if passed
                  then "прошёл"
                  else "провалился"
  return statement
```

Остаётся передать ID потенциального кандидата в функцию и получить результат в контексте Maybe:

```
GHCi> assessCandidateMaybe 1
Just "провалился"
GHCi> assessCandidateMaybe 3
Just "прошёл"
GHCi> assessCandidateMaybe  4
Nothing
```

Обратите внимание: код практически идентичен коду из предыдущего раздела. Это связано с тем, что после присваивания значения переменной с помощью <- в do-нотации вы делаете вид, что работаете с обычным типом, не находящимся в контексте. Класс типов Monad и do-нотация позволяют абстрагироваться от контекста, в котором вы работаете. Непосредственным преимуществом при решении этой задачи является то, что вам не приходится думать о проблемах, которые могут быть связаны с отсутствующими значениями. Более существенный плюс в терминах абстракции состоит в том, что вы получаете возможность единообразно думать обо всех задачах, связанных с вычислениями в контексте. Это не только облегчает вам работу с отсутствующими значениями, но и позволяет начать разрабатывать программы, работающие *в любом* контексте.

Проверка 31.4. Напишите функцию типа `Maybe String -> String`, возвращающую `"прошёл"` или `"провалился"`, если значение присутствует, и `"ошибка: id не найден"` — в противном случае.

31.3. Контекст списка — обработка списка кандидатов

Тот факт, что список является монадой, не должен вас удивлять, поскольку список является примером буквально всех реализованных в Haskell

Ответ 31.4

```
failPassOrElse :: Maybe String -> String
failPassOrElse Nothing = "ошибка: id не найден"
failPassOrElse (Just val) = val
```

интересных особенностей. В следующем уроке вы разберётесь, что это может означать, а сейчас давайте посмотрим, что произойдёт, если вы будете работать со списком кандидатов.

Листинг 31.15 Список возможных кандидатов

```
candidates :: [Candidate]
candidates = [candidate1, candidate2, candidate3]
```

Благодаря тому что список является представителем класса `Monad`, вы должны суметь преобразовать свои старые функции для проверки кандидатов в функцию `assessCandidateList`. Если вы сделаете это и передадите в функцию список кандидатов, то получите полезный результат.

Листинг 31.16 Проверка кандидатов с помощью списка как монады

```
assessCandidatesList :: [Candidate] -> [String]
assessCandidateList candidates = do
  candidate <- candidates
  let passed = viable candidate
  let statement = if passed
                  then "прошёл"
                  else "провалился"
  return statement
```

Данная функция проверяет на соответствие минимальным требованиям всех кандидатов в списке и возвращает список результатов проверки:

```
GHCi> assessCandidateList candidates
["провалился", "провалился", "прошёл"]
```

И снова не пришлось изменять логику функции `assessCandidateX`. При работе со списками с использованием средств класса типов `Monad` вы можете воспринимать целые списки как одиночные значения. Не зная Haskell, вы смогли бы прочесть код функции `assessCandidateList`, но, вероятно, предположили бы, что она работает с единственным значением, а не со списком. Тот же код можно написать, применив функцию `map`.

Листинг 31.17 Специфичный способ оценки кандидатов в списке

```
assessCandidates :: [Candidate] -> [String]
assessCandidates candidates =
     map (\x -> if x
                then "прошёл"
                else "провалился") passed
  where passed = map viable candidates
```

В этом коде как минимум две проблемы с точки зрения абстракции. Первая состоит в том, что вы вынуждены думать о задаче в терминах списков. Человек, незнакомый с Haskell, был бы озадачен этим кодом из-за использования `map`. Вторая и более важная проблема — в том, что у вас нет способа обобщить этот код для работы с другими контекстами. Такой код `assessCandidates` существенно отличается от написанных вами ранее `assessCandidateIO` и `assessCandidateMaybe`, хотя делает он то же самое.

В следующем разделе вы начнёте думать о задачах в терминах монад и осознаете, что у вас есть обобщённое решение, которое можно с лёгкостью использовать для работы во всех трёх рассмотренных контекстах.

Проверка 31.5. Работает ли assessCandidateList с пустым списком?

31.3.1. Собираем всё вместе и пишем монадическую функцию

До сих пор вы фокусировались в основном на том, как do-нотация и класс типов `Monad` позволяют вам решать задачи, абстрагируясь от контекста:

- вы можете писать код для работы в контексте `IO`, не заботясь о различиях между `IO String` и обычными `String`;
- вы можете писать код для работы в контексте `Maybe` и не думать о том, как обрабатывать отсутствующие значения;
- вы даже можете писать код для списков так, как будто работаете с единственным значением.

Но класс типов `Monad` даёт вам ещё одно преимущество, которое возникает благодаря возможности абстрагироваться от конкретного контекста при написании программ. Три функции `assessCandidate*` имеют практически идентичный код. Класс типов `Monad` позволяет не только упростить решение задач в определённом контексте, но и создать решение, работающее в любом возможном контексте.

Единственное ограничение на использование ранее написанного кода сразу во всех трёх контекстах состоит в ограниченности соответствующих

Ответ 31.5. Работает! Если вызвать функцию `assessCandidateList` с пустым списком в качестве аргумента, то мы получим пустой список.

типовых аннотаций. Так как IO, Maybe и список являются представителями класса типов Monad, вы можете использовать его в своём определении универсальной функции assessCandidate. Удивительно, но нам достаточно взять функцию assessCandidateList, поменяв ей типовую аннотацию.

Листинг 31.18 Монадическая функция для IO, Maybe и списка

```
assessCandidate :: Monad m => m Candidate -> m String
assessCandidate candidates = do
  candidate <- candidates
  let passed = viable candidate
  let statement = if passed
                  then "прошёл"
                  else "провалился"
  return statement
```

В GHCi вы можете убедиться в возможности использовать эту функцию с тремя разными контекстами:

```
GHCi> assessCandidate readCandidate
введите id:
1
введите оценку код-ревью:
A
введите оценку соответствия корпоративной культуре:
B
введите образование:
PhD
"прошёл"

GHCi> assessCandidate (Map.lookup 1 candidateDB)
Just "провалился"
GHCi> assessCandidate (Map.lookup 2 candidateDB)
Just "провалился"
GHCi> assessCandidate (Map.lookup 3 candidateDB)
Just "прошёл"

GHCi> assessCandidate candidates
["провалился", "провалился", "прошёл"]
```

Многие из виденных вами к данному моменту примеров показывают, как применение класса типов Monad позволяет вам писать код для обычных типов и использовать его для работы в разнообразных контекстах, таких как Maybe, IO или список. В этом уроке вы увидели, как взять код, работающий в одном из этих контекстов, и обобщить его для работы во всех контекстах с помощью класса типов Monad.

Итоги

В этом уроке нашей целью было научить вас использовать do-нотацию для работы с монадами. Хорошие новости состоят в том, что у вас уже есть достаточно много опыта использования do-нотации, так как она активно использовалась в модуле 4. Ещё важно понимать «рассахаренный» монадический код, так как это может существенно помочь при отладке и понимании проблем, возникающих при работе с монадами. Вы также увидели, как код, написанный для контекста `IO` с использованием do-нотации, может быть легко переписан для работы с типами `Maybe`. Хотя эта возможность полезна сама по себе, она также означает, что вы можете писать более обобщённый код, работающий со всеми монадами. Давайте удостоверимся, что вы всё поняли.

Задача 31.1. В конце урока 21 (модуль 4) вы видели следующую программу, используемую для сравнения стоимости пиццы разных размеров:

```
main :: IO ()
main = do
   putStrLn "Введите размер первой пиццы"
   size1 <- getLine
   putStrLn "Введите стоимость первой пиццы"
   cost1 <- getLine
   putStrLn "Введите размер второй пиццы"
   size2 <- getLine
   putStrLn "Введите стоимость второй пиццы"
   cost2 <- getLine
   let pizza1 = (read size1, read cost1)
   let pizza2 = (read size2, read cost2)
   let betterPizza = comparePizzas pizza1 pizza2
   putStrLn (describePizza betterPizza)
```

Избавьтесь от синтаксического сахара, используя операции `>>=` и `>>`, метод `return` и лямбда-функции вместо do-нотации.

Задача 31.2. В конце урока 21 (модуль 4) мы также впервые представили вам идею того, что do-нотация может быть использована не только для `IO`. В итоге вы написали функцию, работающую с типом `Maybe`:

```
maybeMain :: Maybe String
maybeMain = do
  size1 <- Map.lookup 1 sizeData
  cost1 <- Map.lookup 1 costData
```

```
size2 <- Map.lookup 2 sizeData
cost2 <- Map.lookup 2 costData
let pizza1 = (size1, cost1)
let pizza2 = (size2, cost2)
let betterPizza = comparePizzas pizza1 pizza2
return (describePizza betterPizza)
```

Перепишите эту функцию для работы со списком вместо `Map` (не переживайте, если результат будет выглядеть странно).

Задача 31.3. Перепишите код функции `maybeMain` из предыдущего упражнения так, чтобы он работал с любой монадой. Вам понадобится изменить типовую аннотацию и убрать из тела функции специфические для конкретного типа фрагменты кода.

32

Монада списка и генераторы списков

После прочтения урока 32 вы:

- научитесь использовать do-нотацию для генерации списков;
- узнаете, как фильтровать результаты в do-нотации, используя `guard`;
- сможете писать генераторы списков.

В конце предыдущего урока вы увидели, что список является экземпляром `Monad`, его монадические возможности использовались там для обработки списка кандидатов.

Листинг 32.1 Функция assessCandidateList из предыдущего урока

```
assessCandidateList :: [Candidate] -> [String]
assessCandidateList candidates = do
   candidate <- candidates
   let passed = viable candidate
   let statement = if passed
                   then "passed"
                   else "failed"
    return statement
```

Благодаря <- вы можете обращаться со списком кандидатов как с одним кандидатом

Вы снова обращаетесь с результатами вычисления на кандидатах как с операциями на одном кандидате

Функция принимает одного кандидата в качестве аргумента

При возвращении результата вы получаете свой список обратно

Список как монаду интересным делает то, что когда вы присваиваете свой список переменной, используя <-, вы обращаетесь с ним, будто это одиночное значение. Остаток этого кода выглядит как операции на одном кандидате, а итоговый результат соответствует применению вашей логики к каждому из кандидатов в списке.

Когда вы рассматривали список как `Applicative`, вы видели несколько сперва смущающих примеров недетерминированных вычислений. Например, если у вас есть два списка, используя `pure (*)` для их перемножения с `<*>`, вы получите все возможные комбинации пар значений этих списков:

```
GHCi> pure (*) <*> [1 .. 4] <*> [5,6,7]
[5,6,7,10,12,14,15,18,21,20,24,28]
```

Возможно, вы ожидаете, что список как `Monad` окажется ещё более запутанным, но на самом деле всё удивительно знакомо. Монада списков позволяет вам тривиально строить сложные списки в лёгком для написания стиле. Это похоже на LINQ в C# или генераторы списков в Python и других языках. Оказывается, есть даже способ ещё сильнее упростить donотацию для генерации списков.

Обратите внимание. Каков простейший способ создать список квадратов каждого нечётного числа, меньшего 20?

32.1. Построение списков при помощи монады

Основное применение монады списков — это быстрое их построение. На рис. 32.1 показан пример использования монады списков для создания списка степеней двойки.

```
powersOfTwo :: Int -> [Int]
powersOfTwo n = do
  value <- [1 .. n]
  return (2^value)
```

Когда вы присваиваете значения посредством `<-`, вы как бы притворяетесь, что эти значения не находятся в контексте. В данном случае контекст представлен списком `[Int]`, поэтому это значение трактуется как `Int`

Несмотря на то что мы работаем со списком значений, вы можете возвести 2 в степень `value`, как будто это одно значение. Класс `Monad` так и работает: мы считаем, что типы в контексте — это просто обычные типы

Рис. 32.1: Генерация списков при рассмотрении списка как монады

Давайте построим десять первых степеней двойки в GHCi:

```
GHCi> powersOfTwo 10
[2,4,8,16,32,64,128,256,512,1024]
```

Обратите внимание, что в этом определении вы можете обращаться со списком как с одиночным значением, и результаты соответствуют ожиданиям. Нетрудно проделать то же самое с помощью `map`, как в модуле 1:

```
powersOfTwoMap :: Int -> [Int]
powersOfTwoMap n = map (\x -> 2^x) [1 .. n]
```

Однако в этом случае вы думаете о списке как о структуре данных, не пытаясь обобщить контекст. Для таких случаев версия с использованием `map`, вероятно, намного проще для написания и чтения. Но когда вы генерируете более сложные списки, возможность сфокусироваться на том, как вы преобразуете одно значение, может быть полезной. Давайте рассмотрим ещё несколько примеров генерации списков в do-нотации. Предположим, вы хотите иметь степени двойки и тройки в виде списка пар.

Листинг 32.2 Создание списка пар с помощью do-нотации

```
powersOfTwoAndThree :: Int -> [(Int,Int)]
powersOfTwoAndThree n = do
    value <- [1 .. n]
    let powersOfTwo = 2^value
    let powersOfThree = 3^value
    return (powersOfTwo,powersOfThree)
```

Снова работаем со списком как одним значением

`powersOfTwo` — это одно значение, представляющее список степеней двойки

`powersOfThree` — это одно значение, представляющее список степеней тройки

Возвращая одну пару, вы на самом деле формируете список пар

Теперь у вас есть список пар степеней двойки и степеней тройки:

```
GHCi> powersOfTwoAndThree 5
[(2,3),(4,9),(8,27),(16,81),(32,243)]
```

В предыдущем примере вы использовали один список `value` для генерации степеней двойки. Если вы создадите два списка и скомбинируете их в пары тем же образом, то получите другие результаты. Вот функция, которая генерирует все возможные комбинации чётных и нечётных чисел:

```
allEvenOdds :: Int -> [(Int,Int)]
allEvenOdds n = do
    evenValue <- [2,4 .. n]
    oddValue <- [1,3 .. n]
    return (evenValue,oddValue)
```

`evenValue` — это одиночное значение, представляющее список

`oddValue` — это ещё одно одиночное значение, представляющее список

Так как `evenValue` и `oddValue` были созданы при помощи `<-`, их пара представляет все возможные пары из двух значений

Как вы можете увидеть в GHCi, вы получаете не список размера *n*, а скорее все возможные комбинации чётных и нечётных значений:

```
GHCi> allEvenOdds 5
[(2,1),(2,3),(2,5),(4,1),(4,3),(4,5)]
GHCi> allEvenOdds 6
[(2,1),(2,3),(2,5),(4,1),(4,3),(4,5),(6,1),(6,3),(6,5)]
```

Проверка 32.1. Используйте do-нотацию, чтобы сгенерировать список пар чисел от 1 до 10 и их квадратов.

32.1.1. Функция guard

Ещё один полезный трюк — это фильтрация списков. Опять же, вы могли бы использовать `filter`, но при работе с монадами хочется иметь возможность рассуждать о значении вне контекста. В модуле `Control.Monad` есть функция под названием `guard`, позволяющая вам фильтровать значения в списке. Вам нужно импортировать `Control.Monad`, чтобы её использовать. Вот метод генерации чётных чисел с использованием `guard`:

```
evensGuard :: Int -> [Int]
evensGuard n = do
  value <- [1 .. n]
  guard (even value)
  return value
```

Функция guard отфильтрует значения, которые не пройдут проверку

Хотя do-нотация и упрощает генерацию сколь угодно сложных списков с использованием методов класса `Monad`, для этой цели существует более знакомый интерфейс.

Ответ 32.1

```
valAndSquare :: [(Int,Int)]
valAndSquare = do
   val <- [1 .. 10]
   return (val, val^2)
```

Проверка 32.2. Реализуйте `filter`, используя `guard` и do-нотацию.

Функция guard и класс типов Alternative

Если вы посмотрите на типовую аннотацию функции `guard`, то обнаружите, что это странная функция. Прежде всего она содержит ограничение на класс типов, который вы ранее не встречали:

```
guard :: Alternative f => Bool -> f()
```

Класс типов `Alternative` — это *подкласс* `Applicative` (то есть все экземпляры `Alternative` должны быть экземплярами `Applicative`), но, в отличие от `Applicative`, `Alternative` не является подклассом класса `Monad`. Не все экземпляры `Monad` являются также экземплярами `Alternative`. Ключевым для реализации функции `guard` методом класса `Alternative` является метод `empty`, работающий в точности как `mempty` из `Monoid`. И список, и `Maybe` являются экземплярами `Alternative`. Пустое значение для списка — это `[]`, а для `Maybe` — `Nothing`. `IO`, однако, не имеет экземпляра `Alternative`, поэтому нельзя использовать `guard` со значениями в контексте `IO`.

Первая встреча с `guard` может показаться фокусом. Наверняка там за кулисами творится что-то страшное! На удивление, `guard` — чистая функция. Это выходит далеко за рамки данной книги, но если вы комфортно чувствуете себя с монадами, вернитесь к `guard` и посмотрите, можете ли вы реализовать её самостоятельно. Понять `guard` поможет переход от do-нотации обратно к `>>=`, `>>` и лямбдам. Изучение `guard` также многое вам расскажет о тонкостях `>>`. Ещё раз, это не особо полезное для новичков упражнение, но им стоит заняться после того, как вы научитесь комфортно работать с монадами.

Ответ 32.2

```
guardFilter :: (a -> Bool) -> [a] -> [a]
guardFilter test vals = do
    val <- vals
    guard (test val)
    return val
```

32.2. Генераторы списков

Если вы разработчик на Python, наверное, вы найдёте этот метод генерации списков немного многословным. Python использует для таких задач специальный синтаксис *генераторов списков*. Вот генератор списка в Python, формирующий список степеней двойки:

```
Python> [n**2 for n in range(10)]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

Генераторы списков в Python также позволяют вам фильтровать список по какому-либо условию. Вот список чётных квадратов чисел:

```
Python> [n**2 for n in range(10) if n**2 \% 2 == 0]
[0, 4, 16, 36, 64]
```

В принципе, эта запись похожа на do-нотацию, но выглядит чуть более компактно. Посмотрите на версию последнего генератора списка в Python, записанную с помощью do-нотации.

Листинг 32.3 Эмуляция генераторов списков из Python

```
evenSquares :: [Int]
evenSquares = do
    n <- [0 .. 9]
    let nSquared = n^2
    guard (even nSquared)
    return nSquared
```

А теперь сюрприз для любого разработчика на Python! Генераторы списков — это просто специализированное приложение монад!

Конечно, наш пример на Haskell значительно более многословен, чем на Python. К этому моменту на пути изучения Haskell фраза «Haskell более многословен, чем…» должна вас удивлять. Чтобы не дать превзойти себя в краткости, Haskell предлагает дополнительное улучшение специально для списков — генераторы списков в Haskell.

Генераторы списков предоставляют собой ещё более простой способ формирования списков, чем do-нотация, причём переписать код из do-нотации на генераторы списков можно практически механически.

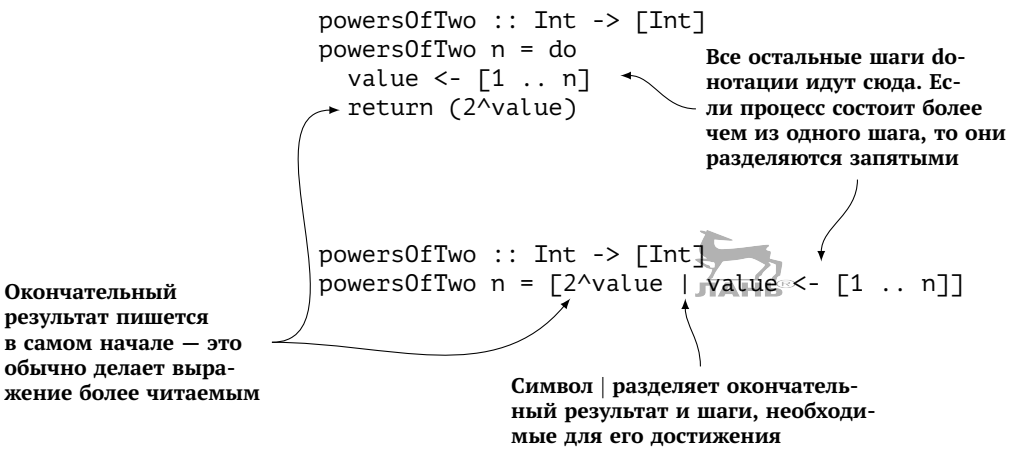


Рис. 32.2: Генераторы списков упрощают do-нотацию

В качестве примера давайте перепишем функцию powersOfTwo из do-нотации на генераторы списков (см. рис. 32.2). А вот как переписать функцию powersOfTwoAndThree:

```
powersOfTwoAndThree :: Int -> [(Int,Int)]
powersOfTwoAndThree n =
   [(powersOfTwo,powersOfThree)
   | value <- [1 .. n]
   , let powersOfTwo = 2^value
   , let powersOfThree = 3^value]
```

Как вы можете увидеть, эта запись почти идентична do-нотации, кроме того что строки разделены запятыми и переход на новую строку необязателен

Для ясности здесь используются полные имена переменных, хотя обычно пишут более короткие имена, что позволяет записывать выражение в одну строку

То, что вы начинаете с результата, а затем показываете, как он был получен, упрощает написание генераторов списков. Зачастую проще понять, что делает генератор, если просто посмотреть на его начало:

```
allEvenOdds :: Int -> [(Int,Int)]
allEvenOdds n = [ (evenValue, oddValue)
                | evenValue <- [2,4 .. n]
                , oddValue <- [1,3 .. n]]
```

Функция guard в генераторах списков аккуратно спрятана:

```
evensGuard :: Int -> [Int]
evensGuard n = [ value | value <- [1 .. n], even value]
```

Генераторы списков — это хороший способ сделать работу с монадами ещё проще. Другая замечательная вещь здесь заключается в том, что если вы использовали их на другом языке, то у вас уже есть опыт написания монадического кода! Несмотря на их выдающееся положение в Haskell, ничто не мешает монадам существовать в других языках, поддерживающих хотя бы самые основы функционального программирования, которые мы обсуждали в модуле 1: функции как значения первого класса, лямбда-выражения и замыкания. Вы можете реализовать систему генераторов списков в любом языке, который это поддерживает: достаточно определить `>>=`, `>>` и `return`.

Проверка 32.3. Напишите генератор списка, который берёт слова из следующего списка, делает первую букву заглавной и добавляет в начало строки `"Г-н "`:

```
["коричневый","синий","розовый","оранжевый"]
```

(Подсказка: используйте функцию `toUpper` из `Data.Char`.)

32.3. Монады: больше, чем просто списки

В итоговом проекте этого модуля вы сделаете ещё один шаг в обобщении операций со списками, разработав SQL-подобный интерфейс для работы с ними. После всего этого времени, проведённого с монадой для списков, вы могли подумать, что монады только для них и нужны. Но не забывайте о модуле 4! Хоть мы и не особо об этом говорили, почти каждая строка кода, написанная в тех уроках, использовала класс типов `Monad`.

К концу этого модуля вы довольно глубоко погрузились в два способа использования монад — для `IO` и для списков. Цель в том, чтобы пока-

Ответ 32.3

```
import Data.Char
answer :: [String]
answer = ["Г-н " ++ capVal |
          val <- ["коричневый","синий","розовый","оранжевый"]
         ,let capVal = (\(x:xs) -> toUpper x:xs) val]
```

зать вам, насколько мощной абстракцией может быть идея работы в контексте. Вы использовали работу в контексте для отделения «грязного» кода с изменяемым состоянием, необходимого для выполнения ввода-вывода, от остальной предсказуемой и безопасной программной логики. Вы также видели, как упрощать генерацию сложных данных с использованием класса типов `Monad`. Во многих примерах мы использовали монады для работы с типами `Maybe`, что позволяло вам писать сложные программы, работающие с отсутствующими значениями, без необходимости думать о том, как вы будете обрабатывать факт их отсутствия. Все эти три контекста абсолютно разные, но класс типов `Monad` позволяет вам работать с ними одним и тем же образом.

Итоги

В этом уроке нашей целью было объяснить класс типов `Monad`, продемонстрировав, как ведёт себя список как монада. Для многих изучающих Haskell может быть сюрпризом, что генераторы списков, популярные в Python, эквивалентны монадам. Любой генератор списка может быть тривиально записан в do-нотации, а из любого кода в do-нотации можно убрать сахар, вернувшись к `>>=` и лямбдам. В классе типов `Monad` поразительно то, что он позволяет вам обобщить логику, используемую в генераторах списков, и легко применить её к типам `Maybe` и `IO`! Давайте посмотрим, как вы это поняли.

Задача 32.1. Напишите генератор списка правильных календарных дат, учитывая количество дней в каждом месяце. Например, список должен начинаться с `1 .. 31` для января и продолжаться числами `1 .. 28` для февраля.

Задача 32.2. Перепишите решение предыдущего упражнения с использованием do-нотации, а затем выполните рассахаривание в методы класса типов `Monad` и лямбды.

Итоговый проект: SQL-подобные запросы в Haskell

Этот итоговый проект включает в себя:

- использование класса типов `Monad` для создания SQL-подобных запросов на списках;
- обобщение функций, написанных для одной монады (например, для списка), на множество монад;
- использование типов для написания более читаемых функций.

В предыдущих уроках вы увидели, что такая монада, как список, может трактоваться как генератор списка, инструмент, часто используемый в Python. В данном проекте вы продвинетесь ещё дальше в использовании списков и других монад для создания SQL-подобного интерфейса для списков (или любых других монад). Язык SQL обычно используется для запросов, связанных с реляционными базами данных. Его синтаксис хорошо подходит для описания отношений между данными. Например, если у вас есть информация о преподавателях, ведущих какие-то предметы, вы можете запросить данные о преподавателях английского языка:

```
select teacherName from
teacher inner join course
on teacher.id = course.teacherId
where course.title = "Английский язык";
```

Это позволяет легко комбинировать два набора данных, например информацию о преподавателях и о курсах, чтобы узнать что-то, с ними связанное. Вы создадите набор инструментов, который будет называться HINQ (позаимствуем это название у одного из инструментов .NET, LINQ).

HINQ позволит вам запрашивать данные, работая с ними как с отношениями. Для реализации этого вам потребуется много работать с монадами. Когда же вы завершите этот проект, то получите инструмент для работы с данными, который:

- обеспечивает знакомый интерфейс для запросов реляционных данных в Haskell;
- является сильно типизированным;
- благодаря ленивости вычислений позволяет свободно передавать запросы, не вычисляя их;
- может легко быть использован с другими функциями.

Мы начнём с создания select-запросов на списках, затем научимся фильтровать результаты запросов с помощью функции `where`, а в завершение напишем функцию `join`, которая позволит с лёгкостью комбинировать сложные структуры данных в монадах.

33.1. Начало работы

Давайте начнём с данных. Вы можете реализовывать проект в файле hinq.hs. Чтобы убедиться, насколько хорошо списки могут служить моделью для таблиц реляционных баз данных, напишем пример с данными о студентах, преподавателях, курсах и регистрациях на курсы. На рис. 33.1 показана схема отношений между этими данными.

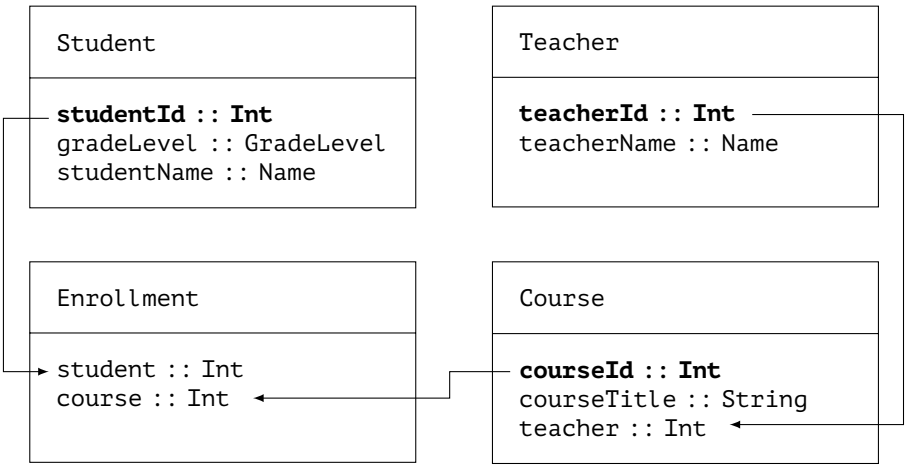


Рис. 33.1: Отношения между данными, с которыми мы будем работать

Начнём с создания модели для студента. У каждого студента есть полное имя, которое состоит из `firstName` и `lastName`.

Листинг 33.1 Тип Name с экземпляром Show

```
data Name = Name { firstName ::String
                 , lastName :: String }

instance Show Name where
    show (Name first last) = mconcat [first," ",last]
```

Также для каждого студента можно сказать, на каком курсе он учится.

Листинг 33.2 Тип GradeLevel представляет ступень обучения

```
data GradeLevel = Freshman
                | Sophmore
                | Junior
                | Senior deriving (Eq,Ord,Enum,Show)
```

В дополнение к вышеперечисленному добавим студентам уникальный идентификатор и получим готовую версию типа `Student`.

Листинг 33.3 Тип Student

```
data Student = Student
    { studentId :: Int
    , gradeLevel :: GradeLevel
    , studentName :: Name } deriving Show
```

Создадим также список студентов для тестов.

Листинг 33.4 Тестовый список студентов

```
students :: [Student]
students = [(Student 1 Senior (Name "Одри" "Лорд"))
           ,(Student 2 Junior (Name "Лесли" "Силко"))
           ,(Student 3 Freshman (Name "Джудит" "Батлер"))
           ,(Student 4 Senior (Name "Ги" "Дебор"))
           ,(Student 5 Sophmore (Name "Жан" "Бодрийяр"))
           ,(Student 6 Junior (Name "Юлия" "Кристева"))]
```

Уже только с одним этим списком мы можем приступить к определению базовых операций: `select` и `where`. В дополнение к ним нам также потребуется операция `join`, которая позволит совмещать списки, содержащие общее поле.

Ориентируясь на типы, можно утверждать следующее: `select` должна принимать функцию, показывающую, какие данные следует отобрать, и список, из которого эти данные нужно взять. Результатом же будет список, где находятся только некоторые значения из исходного списка. Тип функции `select` приведён на рис. 33.2.

Рис. 33.2: Тип функции `select`

Функция `where` принимает предикат (функцию `a -> Bool`) и список, а возвращает только те значения, которые удовлетворяют предикату. Тип этой функции приведён на рис. 33.3.

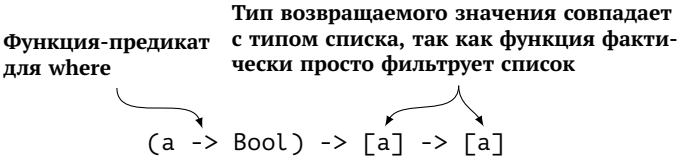


Рис. 33.3: Тип функции `where`

А теперь приступим к функции `join`, она должна принимать два списка со значениями, типы которых могут быть разными, и две функции, каждая из которых позволяет определить, по каким свойствам отбирались элементы списка. Обратите внимание, что тип, характеризующий признак, по которому значения будут сравниваться, должен быть одинаков для обеих функций и для него должен быть экземпляр `Eq`. Результатом работы функции будет список пар элементов из исходных списков, совпадающих по определённому признаку. Вот тип функции `join` (углубимся в детали этой функции чуть позже, когда будем её реализовывать):

```
Eq c => [a] -> [b] -> (a -> c) -> (b -> c) -> [(a,b)]
```

Далее мы займёмся реализацией функций `select` и `where`, которые позволят выполнить некоторые простейшие запросы.

33.2. Простые запросы на списках: select и where

Функции, с которых мы начнём: `select` и `where`. SQL-запрос `select` позволяет выбрать из таблицы определённые поля:

```
select studentName from students;
```

Этот запрос вернёт вам имена всех студентов из таблицы, в которой хранятся данные этих самых студентов. В случае наших HINQ-запросов мы хотим, чтобы `select` возвращал все имена студентов из списка. Условие `where` даёт возможность отфильтровать то, что вернёт `select`, с помощью каких-нибудь ограничений на значения:

```
select * from students where gradeLevel = 'Senior';
```

Этим запросом мы выбрали все записи из таблицы студентов, у которых ступень обучения — `Senior`. Обратите внимание, что в большинстве баз данных вам бы пришлось использовать строки для хранения этих этапов обучения, но в Haskell вы можете позволить себе использовать для этого отдельный тип.

Давайте теперь перейдём к реализации этих функций. Все функции HINQ мы будем записывать с префиксом в виде нижнего подчёркивания (_). Мы делаем это не только потому, что хотим исключить коллизии, так как вы ещё не работали с модулями, но и потому, что `where` является в Haskell зарезервированным словом.

33.2.1. Реализация _select

Наиболее простой операцией является `_select`. Эта функция работает аналогично `fmap`, но мы воспользуемся для её реализации do-нотацией.

Листинг 33.5 Функция _select — это та же функция fmap

```
_select :: (a -> b) -> [a] -> [b]
_select prop vals = do
    val <- vals
    return (prop val)
```

Несколько примеров выборки определённых полей из списка студентов:

```
GHCi> _select (firstName . studentName) students
["Одри","Лесли","Джудит","Ги","Жан","Юлия"]
GHCi> _select gradeLevel students
[Senior,Junior,Freshman,Senior,Sophmore,Junior]
```

Из-за этого примера может показаться, что _select способно отбирать только одно поле, но вы с лёгкостью можете написать лямбда-функцию, которая будет осуществлять выборку сразу двух полей:

```
GHCi> _select (\x -> (studentName x, gradeLevel x)) students
[(Одри Лорд,Senior),(Лесли Силко,Junior),
↳ (Джудит Батлер,Freshman),(Ги Дебор,Senior),
↳ (Жан Бодрийяр,Sophmore),(Юлия Кристева,Junior)]
```

Даже учитывая, как много приёмов функционального программирования мы изучили, очень легко забыть, насколько выразительным может быть использование функций как значений первого класса совместно с лямбда-функциями.

Также обратите внимание, что наша функция _select определённо менее мощная, чем fmap, исключительно из-за её типа. Если бы вы определили _select как _select = fmap, то она бы работала со всеми членами класса типов Functor. Чуть позже мы немного реорганизуем код для этого проекта (для работы с монадами), ну а сейчас просто оцените, насколько выразительным инструментом могут быть типовые аннотации.

33.2.2. Реализация _where

Наша функция _where тоже будет неожиданно простой. Мы просто сделаем небольшую обёртку вокруг guard, которая будет принимать функцию-предикат и список. Напомним, что для использования guard вам нужно будет импортировать Control.Monad. Функция _where сложнее, чем _select, так как она представляет собой нечто большее, чем просто guard (тогда как _select может быть определена как fmap). Воспользуемся присваиванием с помощью <- для обработки списка как одиночного значения, а затем воспользуемся функцией test вместе с guard для фильтрации тех элементов, которые не удовлетворяют предикату.

Листинг 33.6 Функция _where позволяет фильтровать запросы

```
_where :: (a -> Bool) -> [a] -> [a]
_where test vals = do
    val <- vals
    guard (test val)
    return val
```

Чтобы продемонстрировать, как работать с функцией _where, напишем небольшую вспомогательную функцию, которая проверяет, начинается ли строка с определённой буквы.

Листинг 33.7 Пример предиката для _where

```
startsWith :: Char -> String -> Bool
startsWith char string = char == (head string)
```

Теперь можно попробовать воспользоваться `where` и `select` для выборки только тех студентов, фамилии которых начинаются с буквы Б:

```
GHCi> _where (startsWith 'Б' . lastName) (_select studentName
                                                ↳ students)
[Джудит Батлер,Жан Бодрийяр]
```

Функции `_select` и `_where` готовы, поэтому можно приступать к сердцу нашей системы реляционных запросов — функции `_join`.

33.3. Соединение типов данных Course и Teacher

Прежде чем приступить к соединению двух наборов данных, нужно определить несколько типов для этих данных. Начнём с определения типов, хранящих данные о преподавателях и курсах. У каждого преподавателя есть идентификатор (`teacherId`) и имя (`teacherName`).

Листинг 33.8 Тип Teacher

```
data Teacher = Teacher
    { teacherId :: Int
    , teacherName :: Name } deriving Show
```

Вот несколько примеров значений этого типа.

Листинг 33.9 Список преподавателей

```
teachers :: [Teacher]
teachers = [Teacher 100 (Name "Симона" "де Бовуар")
           ,Teacher 200 (Name "Сьюзен" "Зонтаг")]
```

У каждого курса должны быть идентификатор (`courseId`), название (`courseTitle`) и преподаватель (`teacher`), где `teacher` представляет собой значение типа `Int` с идентификатором преподавателя, ведущего этот курс.

Листинг 33.10 Тип Course

```
data Course = Course { courseId :: Int
                     , courseTitle :: String
                     , teacher :: Int } deriving Show
```

Приведём несколько примеров.

Листинг 33.11 Список курсов

```
courses :: [Course]
courses = [Course 101 "Французский язык" 100
          ,Course 201 "Английский язык" 200]
```

Теперь займёмся соединением этих двух наборов данных. В терминах SQL это соединение понимается как *внутреннее соединение*, что означает, что значение имеют только совпадающие пары. В SQL следующий запрос вернёт пары учителей и курсы, которые они ведут:

```
select * from
teachers inner join courses
on (teachers.teacherId = courses.teacher);
```

Для `_join` вам нужно будет проверять, совпадают ли поля данных из одного списка с полями данных из другого списка. Для этого потребуется описать достаточно сложный тип. Нам требуется передать нашей функции `_join` два списка, а затем функцию, которая будет отображать элементы списка в определённое поле, по которому будет выполняться соединение. Возвращать получившаяся функция будет соединённые списки. На рис. 33.4 продемонстрирован тип `_join` с комментариями, чтобы вам было легче вникнуть в смысл этой функции.

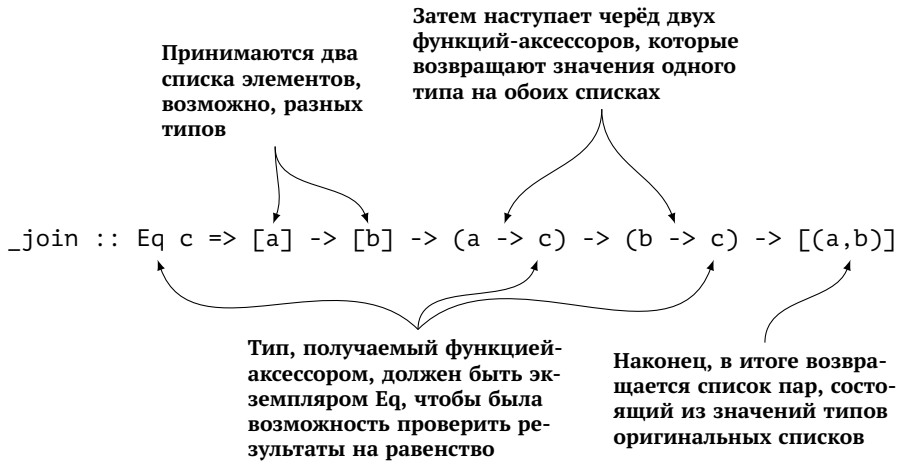


Рис. 33.4: Тип `_join` с пояснениями

Мы будем реализовывать `_join` по тому же принципу, по которому аналогичная операция действует в реляционной алгебре, теоретической основе баз данных. Начнём с построения декартова произведения списков (само по себе оно является операцией перекрёстного соединения в SQL). Декартово произведение является комбинацией всех возможных пар. Прежде чем вернуть результат работы функции, отберём только те пары, поля элементов которых совпадают (сделаем это с помощью функций, которые тоже были переданы `_join`, как показано на рис. 33.5).

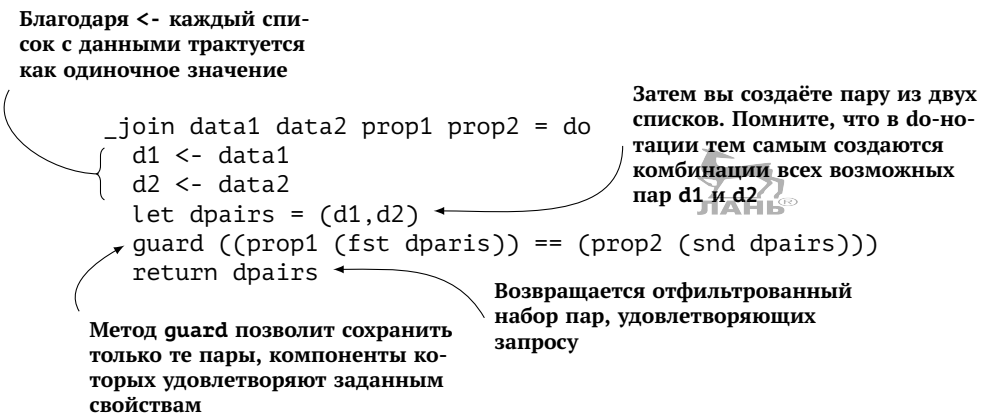


Рис. 33.5: `_join` соединяет два набора данных с совпадающими полями

Можете воспользоваться `_join` для соединения данных о преподавателях и о студентах:

```
GHCi> _join teachers courses teacherId teacher
[(Teacher {teacherId = 100, teacherName = Симона де Бовуар},
↳ Course {courseId = 101, courseTitle = "Французский язык",
↳ teacher = 100}), (Teacher {teacherId = 200,
↳ teacherName = Сьюзен Зонтаг},Course {courseId = 201,
↳ courseTitle = "Английский язык", teacher = 200})]
```

Реализовав три основные части нашего языка запросов, мы можем скомпоновать `_select`, `_where` и `_join` в более удобные функции.

33.4. Построение интерфейса HINQ и тестовые запросы

Нам бы хотелось, чтобы части запроса было легче комбинировать. Вот пример использования всех трёх функций для нахождения всех преподавателей английского языка (в нашей выборке такой только один).

Листинг 33.12 Пример совмещения _join, _select и _where

```
joinData = (_join teachers courses teacherId teacher)
whereResult = _where ((== "Английский язык") . courseTitle
                                                . snd) joinData
selectResult = _select (teacherName . fst) whereResult
```

Это неплохое решение, но нам нужно, чтобы запросы были похожи на их SQL-версию. Обычно SQL-запросы выглядят примерно так:

```
select <поля> from <данные> where <условия>
```

Исходные данные могут быть списком или двумя соединёнными списками. Нам бы хотелось, чтобы запрос можно было перестроить так:

```
(_select (teacherName . fst))
(_join teachers courses teacherId teacher)
(_where ((== "Английский язык") .courseTitle . snd))
```

Этого можно добиться, использовав для реструктуризации кода лямбда-функции. Создадим функцию `_hinq`, которая будет принимать `_select`, `_join` и `_where`-запросы в таком порядке, в каком они вам требуются, а затем с помощью лямбда-функции всё перестраивать.

Листинг 33.13 Функция _hinq позволяет перестроить текст запроса

```
_hinq selectQuery joinQuery whereQuery =
   (\joinData ->
    (\whereResult ->
       selectQuery whereResult)
    (whereQuery joinData)
   ) joinQuery
```

Функцию `_hinq` можно использовать для отправки запроса. Эта реализация, разумеется, не будет идеальной копией SQL или LINQ, но по смыслу она достаточно близка и, что более важно, даёт возможность думать о соединении двух списков так, как пришлось бы при работе с обычными реляционными запросами. Вот наш предыдущий запрос, переписанный с использованием функции `_hinq`.

Листинг 33.14 Функция _hinq как приближение Haskell к SQL

```
finalResult :: [Name]
finalResult =
  _hinq (_select (teacherName . fst))
        (_join teachers courses teacherId teacher)
        (_where ((== "Английский язык") .courseTitle . snd))
```

Осталась одна досадная мелочь. Предположим, что вы хотите извлечь из `finalResult` фамилии всех перечисленных там преподавателей. Для этого вам не обязательно использовать `_where`. Можно просто воспользоваться функцией `(\_ -> True)`, которая всегда будет возвращать `True`.

Листинг 33.15 Возможно решение проблемы отсутствующего _where

```
teacherLastName :: [String]
teacherLastName = _hinq (_select lastName)
                        finalResult
                        (_where (\_ -> True))
```

Это будет работать, но передавать универсальную функцию-заглушку каждый раз не особенно удобно. К сожалению, в Haskell нет аргументов по умолчанию. Так как же тогда работать с запросами без `_where`? Воспользуемся типом `HINQ`, у которого будет два конструктора.

33.5. Определение типа HINQ для запросов

В этом разделе вы создадите тип `HINQ`, который будет представлять запрос. Вы уже знаете, что любой запрос может состоять из операций `select`, `join` и `where` или только из двух первых операций. Это позволяет составлять запросы с и без инструкции `_where`. Однако прежде чем двинуться дальше, вам потребуется немного улучшить определения функций `_select`, `_join` и `_where`. В данный момент эти операции работают только со списками, но мы бы хотели обобщить их для всех монад. Для этого не придётся сильно менять код, достаточно лишь сделать типовую аннотацию чуть менее строгой. Правда, придётся добавить ограничение на класс типов, так как функция `guard` работает только на типах из `Alternative`. `Alternative` является подклассом `Applicative` и включает в себя определение пустого элемента для типа (почти как `Monoid`). И списки, и `Maybe` являются членами `Alternative`, а вот про `IO` такого сказать нельзя. Для использования класса типов `Alternative` вам нужно будет импортировать модуль `Control.Applicative`. Вот скорректированные типовые аннотации, которые позволят расширить область работы наших HINQ-запросов.

Листинг 33.16 Типы _select, _where и _join для монад

```
_select :: Monad m => (a -> b) -> m a -> m b
_where :: (Monad m, Alternative m) => (a -> Bool) -> m a -> m a
_join :: (Monad m, Alternative m, Eq c) =>
         m a -> m b -> (a -> c) -> (b -> c) -> m (a,b)
```

Это великолепный пример, почему стоит использовать do-нотацию, когда есть такая возможность. Вы начали решать проблему, работая только со списками. Но оказалось, что благодаря небольшому изменению типов можно сделать код гораздо более обобщённым! Если бы мы использовали функции `map` и `filter`, то переписывание кода потребовало гораздо больше времени. Теперь, когда все типы исправлены, можно заняться написанием обобщённого типа `HINQ`, который будет представлять собой тип запросов (рис. 33.6).

Рис. 33.6: Определение типа `HINQ` с комментариями

Этот тип использует типы функций `_select`, `_join` и, при необходимости, `_where`. Используя его, мы можем написать функцию `runHINQ`, которая будет принимать значение типа `HINQ` и запускать запрос на выполнение.

Листинг 33.17 Функция runHINQ позволяет выполнить HINQ-запрос

```
runHINQ :: (Monad m, Alternative m) => HINQ m a b -> m b
runHINQ (HINQ sClause jClause wClause) =
   _hinq sClause jClause wClause
runHINQ (HINQ_ sClause jClause) =
   _hinq sClause jClause (_where (\_ -> True))
```

Ещё одним бонусом типа `HINQ` является то, что он немного облегчает понимание исходного большого типа, с которым вам приходилось работать. Давайте составим пару запросов и посмотрим, как они сработают!

33.6. Выполнение HINQ-запросов

Используя `HINQ`, можно начать рассматривать различные виды запросов, которые могут вам потребоваться. Давайте начнём с возвращения к нашим преподавателям. Вот полный HINQ-запрос с указанием типа:

```
query1 :: HINQ [] (Teacher, Course) Name
query1 =
  HINQ (_select (teacherName . fst))
       (_join teachers courses teacherId teacher)
       (_where ((== "Английский язык") .courseTitle . snd))
```

Благодаря ленивости обычное определение запроса не приводит к его выполнению. Это очень хорошо, так как эмулирует поведение .NET LINQ (да, там тоже используются ленивые вычисления), что позволяет составлять и передавать крупные запросы без риска запустить их выполнение до того, как потребуется результат. Другим важным плюсом HINQ является то, что все его запросы являются сильно типизированными. Это окажет неоценимую помощью при поиске ошибок в запросах, так как компилятор Haskell не потерпит такого. А благодаря механизму вывода типов вы можете не писать типы небольших запросов. Попробуйте запустить `query1`:

```
GHCi> runHINQ query1
[Сьюзен Зонтаг]
```

Если вы хотите получить имена преподавателей из того же набора данных, то можете пропустить инструкцию `where` и воспользоваться `HINQ_`:

```
query2 :: HINQ [] Teacher Name
query2 = HINQ_ (_select teacherName) teachers
```

Этот запрос аналогичен простому использованию `_select`, но он хорошо демонстрирует, что наш тип для запросов прекрасно работает и для самых простых случаев. Вы можете проверить, что результаты соответствуют тому, что вы ожидаете увидеть, в GHCi:

```
GHCi> runHINQ query2
[Симона де Бовуар,Сьюзен Зонтаг]
```

Списки — это привычный полигон для чего-то наподобие HINQ, но мы-то обобщили запросы на всех членов `Monad` и `Alternative`. Чуть дальше мы посмотрим на примеры запросов в контексте `Maybe`.

33.6.1. Использование HINQ с Maybe-типами

Не так трудно представить ситуацию, в которой можно столкнуться с `Maybe Teacher` и `Maybe Course`. То, что у вас нет списка значений, не означает, что не возникнет ситуации, в которой вам нужно будет связать преподавателя с курсом. Вот пример того, как могли быть получены эти `Maybe Teacher` и `Maybe Course`.

Листинг 33.18 Пример данных в Maybe

```
possibleTeacher :: Maybe Teacher
possibleTeacher = Just (head teachers)

possibleCourse :: Maybe Course
possibleCourse = Just (head courses)
```

Запуск запроса с Maybe означает, что результат будет получен только в том случае, если запрос не провалится. Он может провалиться из-за недостающих данных *или* из-за того, что не было найдено подходящих значений. Вот пример поиска преподавателей французского для Maybe.

Листинг 33.19 Пример запроса для Maybe

```
maybeQuery1 :: HINQ Maybe (Teacher,Course) Name
maybeQuery1 =
 HINQ (_select (teacherName . fst))
      (_join possibleTeacher possibleCourse teacherId teacher)
      (_where ((== "Французский язык") . courseTitle . snd))
```

Даже в контексте Maybe вы можете использовать знакомые реляционные понятия, осуществлять запросы и получать результаты:

```
GHCi> runHINQ maybeQuery1
Just Симона де Бовуар
```

Если какого-то курса нет, это не мешает запросить данные.

Листинг 33.20 Соединение и обработка отсутствия в Maybe

```
missingCourse :: Maybe Course
missingCourse = Nothing

maybeQuery2 :: HINQ Maybe (Teacher,Course) Name
maybeQuery2 =
  HINQ (_select (teacherName . fst))
       (_join possibleTeacher missingCourse teacherId teacher)
       (_where ((== "Французский язык") .courseTitle . snd))
```

Можно убедиться, что отсутствие данных корректно обрабатывается:

```
GHCi> runHINQ maybeQuery2
Nothing
```

Завершим этот проект, решив с помощью HINQ более сложную задачу, требующую нескольких операций объединения.

33.6.2. Сведения о записи на курсы и соединение множества списков

Далее мы рассмотрим запрос, который вернёт данные обо всех записях на курсы. Для этого потребуется ещё один тип, который будет представлять запись студента на какой-либо курс. Тип `Enrollment` состоит из идентификатора студента и номера курса. Ниже приведён требуемый тип.

Листинг 33.21 Тип Enrollment связывает студента с курсом

```
data Enrollment = Enrollment { student :: Int
                             , course :: Int } deriving Show
```

Все записи можно представить с помощью списка, каждый элемент которого связывает студента с курсом.

Листинг 33.22 Пример списка записей на курсы

```
enrollments :: [Enrollment]
enrollments = [ (Enrollment 1 101), (Enrollment 2 101)
              , (Enrollment 2 201), (Enrollment 3 101)
              , (Enrollment 4 201), (Enrollment 4 101)
              , (Enrollment 5 101), (Enrollment 6 201) ]
```

Предположим, что вы хотите получить список имён всех студентов вместе с курсами, на которые они записаны. Для этого вам потребуется соединить студентов с записями, а затем соединить результат с курсами. Данные о всех записях можно получить с помощью `HINQ_`-запроса. Это прекрасный пример того, как время от времени можно в полной мере насладиться преимуществами вывода типов. Учитывая, какими сложными могут стать типы запросов, их ручное указание может стать нетривиальной задачей. К счастью, механизм вывода типов берёт самую неприятную часть на себя! В следующем запросе делается всё необходимое.

Листинг 33.23 Запрос по студентам и записи их на курсы

```
studentEnrollmentsQ =
  HINQ_ (_select (\(st,en) -> (studentName st, course en))
        (_join students enrollments studentId student)
```

Даже несмотря на то, что мы не хотели заниматься написанием типа запроса, понятно, что результатом должен быть список пар имён и идентификаторов. Когда вы напишите этот запрос, вы можете убедиться, что тип результата соответствует ожидаемому.

Листинг 33.24 Исполнение запроса studentEnrollmentsQ

```
studentEnrollments :: [(Name, Int)]
studentEnrollments = runHINQ studentEnrollmentsQ
```

В GHCi можно на всякий случай всё ещё раз проверить:

```
GHCi> studentEnrollments
[(Одри Лорд,101),(Лесли Силко,101),(Лесли Силко,201),
↳ (Джудит Батлер,101),(Ги Дебор,201),(Ги Дебор,101),
↳ (Жан Бодрийяр,101),(Юлия Кристева,201)]
```

Теперь представим, что вам нужно получить список всех студентов, которые посещают курсы английского языка. Для этого придётся соединить список `studentEnrollments` с курсами. Вот пример запроса имён студентов, которые записались на курс английского языка.

Листинг 33.25 Соединение списка studentEnrollments с курсами

```
englishStudentsQ =
  HINQ (_select (fst . fst))
       (_join studentEnrollments courses snd courseId)
       (_where ((== "Английский язык") . courseTitle . snd))
```

Обратите внимание, что функция `_where` обращалась к данным курсов, а `_select` требовалась только информация о том, какие студенты записаны на курс. Теперь можно сделать запрос и получить список `englishStudents`.

Листинг 33.26 Получение списка студентов, изучающих английский

```
englishStudents :: [Name]
englishStudents = runHINQ englishStudentsQ
```

HINQ позволил соединить эти три списка так, будто они являются таблицами в реляционной базе данных. Обратите внимание, что можно использовать `HINQ` внутри функций, чтобы писать обобщённые инструменты для запрашивания данных. Предположим, что вам требуется функция `getEnrollments`, которая возвращает список студентов, посещающих определённый курс. Это можно устроить, если передавать название курса в запрос, который вы написали до этого.

Листинг 33.27 Запрос данных о записях

```
getEnrollments :: String -> [Name]
getEnrollments courseName = runHINQ courseQuery
  where courseQuery =
```

```
HINQ (_select (fst . fst))
     (_join studentEnrollments courses snd courseId)
     (_where ((== courseName) . courseTitle . snd))
```

Можно проверить работу функции в GHCi:

```
GHCi> getEnrollments "Английский язык"
[Лесли Силко,Ги Дебор,Юлия Кристева]
GHCi> getEnrollments "Французский язык"
[Одри Лорд,Лесли Силко,Джудит Батлер,Ги Дебор,Жан Бодрийяр]
```

Вот и всё! Выразительная сила монад позволила успешно реализовать небольшой движок реляционных запросов, который очень сильно напоминает SQL и LINQ. У получившегося проекта масса преимуществ: запросы легко читать, используются ленивые вычисления, а сильная система типов делает всю системой более надёжной. А ещё, если для какого-нибудь типа вы реализуете экземпляры `Monad` и `Alternative`, то сможете спокойно использовать `HINQ` и на этом новом типе! Практически всё, что было написано для этого проекта, было сделано с помощью функций из класса типов `Monad`. Благодаря комбинации из монад, типов-сумм (тип `HINQ`), ленивых вычислений и функций как значений первого класса у вас получилось создать с нуля достаточно мощный движок запросов!

Итоги

В этом проектном задании вы:

- узнали, как можно реализовать `_select` и `_where` для списков;
- применили декартово произведение для соединения данных;
- без проблем обобщили код, написанный для списков, на монады;
- реструктурировали вызов функций, применив лямбда-функции;
- упростили работу с HINQ, определив тип `HINQ`.

Расширение проекта

Теперь, когда у вас есть фундамент для HINQ-запросов, попробуйте расширить функционал, используя инструменты Haskell! Посмотрите, можно ли реализовать для HINQ `Semigroup` и `Monoid`. Для `Monoid` вам, возможно, потребуется преобразовать тип `HINQ` так, чтобы он включал в себя пустой запрос. Если вы сможете определить экземпляр `Monoid` для `HINQ`, то сможете с лёгкостью компоновать список HINQ-запросов в один запрос!

Модуль 6

Организация кода и сборка проектов

Поздравляем! Вы преодолели наиболее сложные темы из рассматриваемых в этой книге. Начиная с этого модуля, вся оставшаяся часть книги будет сфокусирована на практическом применении пройденного материала. После того как вы изучите эту информацию, вы будете свободнее себя чувствовать при разработке различных проектов на Haskell.

В этом модуле мы приступим к рассмотрению темы, которая хорошо вам знакома, если вы прежде занимались программированием, — организация кода и сборка проектов. Здесь Haskell тоже припас несколько тузов в рукаве, впрочем, ничего более непривычного, чем то, с чем вы уже успели столкнуться на пути сюда.

Модуль начнётся с изучения системы модулей Haskell. Удивительно, но в этой системе нет абсолютно ничего сложного или незнакомого. Как и в любом другом языке программирования, модули служат для группировки связанных функций в одном пространстве имён, что помогает организовывать код для повторного использования. А после всего этого вы погрузитесь в изучение одной из систем сборки проектов для Haskell под названием stack. Эта утилита представляет собой надёжное и понятное средство по автоматизации сборки проектов. Ещё одним интересным, но не особенно сложным пунктом в нашей программе будет библиотека для тестирования кода QuickCheck. QuickCheck в автоматическом режиме генерирует тесты, основанные на заданном наборе свойств, которые нужно проверить в коде.

После того как вы пройдёте материал этой части книги, программирование на Haskell будет больше напоминать обычную разработку программного обеспечения. А в завершение этого модуля вы напишете библиотеку для работы с простыми числами и поймёте на практике, что организация кода в Haskell мало чем отличается от таковой в других языках.

34

Организация кода на Haskell с помощью модулей

После изучения урока 34 вы:

- разберётесь с неявно создаваемым для ваших функций модулем `Main`;
- сможете с помощью модулей выносить функции в разные пространства имён;
- научитесь разделять программы на разные файлы;
- сможете импортировать из модулей только нужные функции.

В предыдущих уроках вы разобрались во множестве интересных тем, связанных с Haskell. Но мы до сих пор не обсуждали одну из важнейших деталей: создание пространств имён для написанных функций. Haskell использует систему модулей, чтобы разделять пространства имён и чтобы помочь вам в более удобной организации кода. Эта система очень похожа на модули, которые используются в Ruby и Python, а также на пакеты и пространства имён Java и C#.

Вы уже успели поработать с системой модулей Haskell. Каждый раз, когда вы использовали `import`, вы добавляли в проект новый модуль. Да и все встроенные функции и типы, такие как `[]`, `length` и `(:)`, включены в стандартный модуль `Prelude`, который импортируется автоматически. Всю документацию к `Prelude` можно найти на Hackage (`https://hackage.haskell.org/package/base/docs/Prelude.html`).

До сих пор мы избегали модулей, помещая весь код для отдельных проектов в один файл и награждая функции уникальными именами, чтобы избежать коллизии имён. В этом уроке вы начнёте организовывать код с помощью модулей. Для этого сосредоточимся на сравнительно простой задаче — написании консольной программы, которая будет запрашивать

у пользователя слово и проверять, является ли это слово палиндромом. В идеале нам бы хотелось поместить основное IO-действие и функции для проверки слов в разные файлы. Это поможет лучше организовать код и сделает расширение функционала программы более лёгкой задачей. Начнём с того, что реализуем всё в одном файле, а затем посмотрим, как можно корректно разделить код на две части, поместив их в разные файлы.

Обратите внимание. У вас есть два типа: один — для книг, а второй — для журналов. У них присутствуют поля с одинаковыми названиями, но представляют они совершенно разные вещи:

```
data Book = Book
     { title :: String
     , price :: Double }

data Magazine = Magazine
     { title :: String
     , price :: Double }
```

Оба типа реализованы как записи, что усложняет задачу, так как для этих типов автоматически генерируются функции для доступа к полям `title` и `price`. К сожалению, это вызовет ошибку из-за того, что вы пытаетесь определить две функции с одинаковыми именами. Вы бы хотели избежать переименования этих полей во что-то вроде `bookTitle` и `bookPrice`. Как можно разрешить эту ситуацию?

34.1. Что случится, если использовать имя из Prelude?

Начнём с написания улучшенной версии функции `head`, которую вам уже приходилось не раз использовать. В `Prelude` функция `head` реализована следующим образом.

Листинг 34.1 Определение head из Prelude

```
head         :: [a] -> a
head (x:_)   =  x
head []      =  errorEmptyList "head"
```

`errorEmptyList` — специализированная функция генерации ошибки для списков

Проблема функции `head` в том, что она завершается с ошибкой, если в качестве аргумента подать ей пустой список. Это не особенно хорошее

поведение для функций в Haskell, как мы выясним в уроке 38, когда будем обсуждать обработку ошибок. Основной причиной того, что `head` выбрасывает ошибку, является то, что очень часто не существует подходящего значения, которое функция может вернуть. Языки вроде Lisp и Scheme вернули бы в аналогичной ситуации пустой список, но система типов Haskell не позволит это сделать (так как тип пустого списка обычно отличается от типа элементов списка). Вы могли бы избежать проблемы, если бы ограничили `head` для работы только со списками элементов, для которых есть экземпляр класса типов `Monoid`. Напомним определение класса `Monoid`.

Листинг 34.2 Определение класса типов Monoid

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
```

У каждого моноида должен быть элемент `mempty`. Этот элемент представляет нейтральное значение. Списки тоже являются моноидами, а нейтральный элемент для них — пустой список, `[]`. Для всех членов класса типов `Monoid` вы можете возвращать элемент `mempty`, если функции был передан пустой список. Вот новая и более безопасная версия `head`.

Листинг 34.3 Ой, мы случайно создали функцию с занятым именем!

```
head :: Monoid a => [a] -> a
head (x:xs) = x
head [] = mempty
```

Если вы запишете код этой функции в файл, а затем его скомпилируете, то всё будет пройдёт нормально, даже учитывая, что вы «случайно» использовали занятое имя, поскольку функция `head`, которую вы использовали до сих пор, является частью модуля `Prelude`. Чтобы протестировать нашу новую функцию, нужно придумать пример списка, тип значений которого является членом `Monoid`. В данном случае мы воспользуемся пустым списком списков (помните, что на тип элементов списка наложено ограничение).

Листинг 34.4 Пример списка со значениями из класса Monoid

```
example :: [[Int]]
example = []
```

Этот код спокойно скомпилируется, но если вы попробуете использовать `head` в GHCi, то увидите сообщение об ошибке, поскольку функция

с таким названием уже существует:

```
Ambiguous occurrence 'head'
It could refer to either 'Main.head'
defined at ...
or 'Prelude.head'
```

Ошибка заключается в том, что у вас есть две функции `head`, и Haskell не знает, какая именно вам требуется

А это уже знакомая вам функция из `Prelude`

Это ваша собственная функция, Haskell автоматически создал для неё модуль

Мы столкнулись с проблемой, так как Haskell не может определить, какая из двух версий `head` вам требуется — та, которая была определена в `Prelude`, или та, которую вы написали сами. Что интересно, новая функция называется `Main.head`. Когда вы явно не обозначаете, что пишете код в неком модуле, Haskell предполагает, что этот код относится к модулю `Main`. Вы можете сделать это явно, добавив одну строку в начале файла.

Листинг 34.5 Явное указание модуля

```
module Main where

head :: Monoid a => [a] -> a
head (x:xs) = x
head [] = mempty

example :: [[Int]]
example = []
```

Эта строка — единственное, что отличается от первоначального варианта

Для выбора нужной версии `head` вы можете сослаться на полное название функции, включающее имя модуля. Использование `Main.head` сигнализирует о том, что используется ваша функция, а `Prelude.head` — функция из `Prelude`. Вот пример в GHCi:

```
GHCi> Main.head example
[]
GHCi> Prelude.head example
*** Exception: Prelude.head: empty list
```

Далее мы углубимся в изучение модулей, создав простую программу, разбитую по двум файлам.

Проверка 34.1. Предположим, что вам нужно сохранить длину некоторого объекта в виде переменной. Например:

```
length :: Int
length = 8
```

Как бы вы обратились к этому значению, не вызывая конфликтов с функцией `length` из `Prelude`?

34.2. Сборка многофайловой программы с помощью модулей

В этом разделе мы займёмся сборкой простой программы, которая будет читать строку из консоли, а затем возвращать информацию о том, является ли эта строка палиндромом. Начнём с наброска, сделанного в одном файле, который сможет определять такие палиндромы как *доход*, но ошибётся со словом *Доход*. Затем разобьём код на две части в разных файлах: один будет отвечать за основную логику работы программы, а второй будет библиотекой с функциями для определения палиндромов.

Перемещение набора связанных функций в отдельный модуль является очень полезной практикой. Основной же модуль должен быть сосредоточен только на логике выполнения программы. Именно поэтому все функции, отвечающие за работу с палиндромами, следует вынести в отдельный файл, так как тогда будет легче работать с функциями из этой библиотеки. Также вы можете спрятать некоторые функции внутри модуля, как это делают приватные методы в Java и C#. Это позволит вам инкапсулировать функции, чтобы только некоторые из них были доступны извне.

Ответ 34.1. Явно укажите модуль, имя из которого хотите применить:

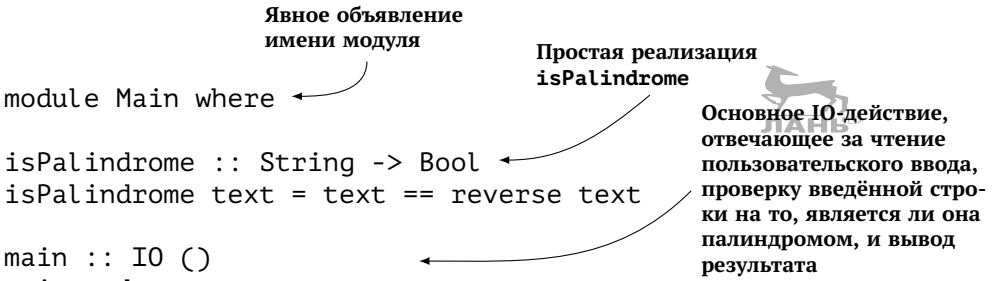
```
length :: Int
length = 8

doubleLength :: Int
doubleLength = Main.length * 2
```

34.2.1. Создание модуля Main

До сих пор мы относились к именам функций спустя рукава. Теперь же, когда мы начинаем рассуждать о правильной организации кода, мы должны быть более осторожны. Как вы увидели в модуле 4, каждая программа на Haskell имеет функцию `main`, как и в программах на Java обязательно присутствует метод `main`. Обычно функция `main` находится в модуле `Main`. Есть правило, по которому модули в Haskell должны размещаться в файлах с такими же названиями, как и у модуля. Создавая этот проект, вы должны начать с файла Main.hs. Вот первая версия программы.

Листинг 34.6 Первая версия модуля Main




```
module Main where

isPalindrome :: String -> Bool
isPalindrome text = text == reverse text

main :: IO ()
main = do
    putStrLn "Введите слово, чтобы узнать, является ли оно
                                                ↳ палиндромом."
    text <- getLine
    let response = if isPalindrome text
                   then "Да, это палиндром."
                   else "Нет, это не палиндром."
    putStrLn response
```

Вы можете скомпилировать эту программу и протестировать код, загрузив его в GHCi. Там вы можете убедиться, что получившаяся программа не так хороша, как нам бы хотелось:

```
GHCi> main
Введите слово, чтобы узнать, является ли оно палиндромом.
доход
Да, это палиндром.
GHCi> main
Введите слово, чтобы узнать, является ли оно палиндромом.
Меч - а зачем?
Нет, это не палиндром.
```

Программа корректно определила *доход*, но с другим палиндромом возникла проблема. Нам требуется выполнить небольшую предваритель-

ную обработку, в ходе которой строка очистится от пробелов, знаков пунктуации, а также все буквы переведутся в один регистр. В недалёком прошлом вы бы просто добавили все требуемые функции прямо в этот файл. Но по двум причинам было бы благоразумнее перенести весь код, работающий с палиндромами, в отдельный файл. Во-первых, это поможет сделать модуль `Main` аккуратнее, а во-вторых, вам будет легче использовать код для работы с палиндромами в других программах.

34.2.2. Перенос кода для палиндромов в отдельный модуль

Перенесём код для работы с палиндромами в отдельный модуль. Было бы логично назвать его `Palindrome`, так что файл для него будет называться Palindrome.hs. В этом модуле будет функция `isPalindrome`, которая предназначена для использования в модуле `Main`. Вам нужно написать более качественную версию `isPalindrome`, поэтому в модуле будет несколько вспомогательных функций: `stripWhiteSpace`, `stripPunctuation`, `toLowerCase` и `preprocess`, которая применит остальные. Вот содержимое этого файла.

Листинг 34.7 Файл Palindrome.hs

Вы объявили, что модуль называется `Palindrome` и он экспортирует только одну функцию `isPalindrome`

Вы могли бы импортировать модуль `Data.Char` целиком, но потребуются только эти три функции

Всё остальное похоже на то, что вы писали до этого

```
module Palindrome (isPalindrome) where

import Data.Char (toLower,isSpace,isPunctuation)

stripWhiteSpace :: String -> String
stripWhiteSpace text = filter (not . isSpace) text

stripPunctuation :: String -> String
stripPunctuation text = filter (not . isPunctuation) text

toLowerCase :: String -> String
toLowerCase text = map toLower text

preprocess :: String -> String
preprocess = stripWhiteSpace . stripPunctuation . toLowerCase

isPalindrome :: String -> Bool
isPalindrome text = cleanText == reverse cleanText
    where cleanText = preprocess text
```

Давайте разберём этот файл строка за строкой, чтобы прочувствовать, что здесь происходит. Вы могли начать файл таким образом:

```
module Palindrome where
```

По умолчанию это объявление экспортирует все функции, определённые в Palindrome.hs. Но единственная функция, которую нужно экспортировать, — `isPalindrome`, вспомогательные функции экспортировать необязательно. Этого можно добиться, перечислив все экспортируемые функции после названия модуля:

```
module Palindrome (isPalindrome) where
```

Вот ещё один способ записи этого объявления, позволяющий с лёгкостью добавлять дополнительные функции:

```
module Palindrome
    ( isPalindrome
    ) where
```

Теперь единственное доступное извне имя из модуля `Palindrome` — это функция `isPalindrome`.

Для реализации вспомогательных функций вам потребуется несколько функций из модуля `Data.Char`. В прошлом вы бы просто и грубо импортировали весь модуль, даже если бы вам требовалась всего одна функция. Но как вы можете выборочно экспортировать функции, так вы можете и выборочно их импортировать. Данная инструкция `import` импортирует только те три функции, которые вам нужны.

Листинг 34.8 Импорт определённого множества функций Data.Char

```
import Data.Char (toLower,isSpace,isPunctuation)
```

Основными преимуществами такого метода импорта является то, что улучшается читаемость кода и уменьшается вероятность непредвиденной коллизии имён при неквалифицированном импорте.

Оставшаяся часть файла ничем не отличается от того, что вы писали, читая эту книгу. Все вспомогательные функции просты для понимания.

Листинг 34.9 Код, позволяющий корректно определять палиндромы

```
stripWhiteSpace :: String -> String
stripWhiteSpace text = filter (not . isSpace) text
```

Эта функция убирает из текста пробелы

```
stripPunctuation :: String -> String
stripPunctuation text = filter (not . isPunctuation) text

toLowerCase :: String -> String
toLowerCase text = map toLower text

preprocess :: String -> String
preprocess = stripWhiteSpace . stripPunctuation . toLowerCase

isPalindrome :: String -> Bool
isPalindrome text = cleanText == reverse cleanText
    where cleanText = preprocess text
```

Далее нужно удалить все знаки пунктуации

Последний шаг: убедимся, что все символы находятся в нижнем регистре

Для соединения вспомогательных функций воспользуемся композицией функций

А вот и улучшенная версия `isPalindrome`

ЛАНЬ

В этом модуле отсутствует `main`, так как это просто библиотека функций. Даже без `main` вы можете загрузить файл в GHCi и протестировать его:

```
GHCi> isPalindrome "доход"
True
GHCi> isPalindrome "Меч - а зачем?"
True
```

ЛАНЬ

Теперь, когда мы разобрались с `Palindrome`, можно вернуться назад и заняться реструктуризацией модуля `Main`.

Проверка 34.2. Измените объявление модуля так, чтобы функция `preprocess` тоже экспортировалась.

34.2.3. Использование модуля Palindrome в Main

Для использования `Palindrome` вам потребуется импортировать этот модуль в `Main`, как вы уже делали с другими модулями. Как вы вскоре увидите, если модуль находится в одном каталоге с `Main`, то компиляция `Main` влечёт за собой компиляцию другого модуля.

Ответ 34.2

```
module Palindrome(
      isPalindrome
    , preprocess
    ) where
```

Предположим, что вам захотелось сохранить имеющееся в `Main` определение `isPalindrome`. В прошлом вы уже использовали объявление квалифицированного импорта `import qualified Module as X` для создания псевдонима модулям, которые вы собираетесь использовать (например, `import qualified Data.Text as T`). Если вы пропустите концовку `as X`, то для доступа к функциям модуля нужно будет ссылаться на название самого модуля. Вот улучшенная версия `main`.

Листинг 34.10 Квалифицированный импорт модуля Palindrome

```
module Main where
import qualified Palindrome
```

Осталось вызвать `isPalindrome` с помощью `Palindrome.isPalindrome`.

Листинг 34.11 Вызов функции Palindrome.isPalindrome

```
let response = if Palindrome.isPalindrome text
```

Вот полностью исправленный модуль Main.hs.

Листинг 34.12 В Main.hs используется файл Palindrome.hs

```
module Main where
import qualified Palindrome

isPalindrome :: String -> Bool
isPalindrome text = text == (reverse text)

main :: IO ()
main = do
    putStrLn "Введите слово, чтобы узнать, является ли оно
                                                ↳ палиндромом."
    text <- getLine
    let response = if Palindrome.isPalindrome text
                   then "Да, это палиндром."
                   else "Нет, это не палиндром."
    putStrLn response
```

При компиляции Main.hs компилятор автоматически найдёт ваш модуль:

```
$ ghc Main.hs
[1 of 2] Compiling Palindrome  ( Palindrome.hs, Palindrome.o )
[2 of 2] Compiling Main        ( Main.hs, Main.o )
Linking Main ...
```

А теперь вы можете запустить скомпилированный исполняемый файл:

```
$ ./Main
Введите слово, чтобы узнать, является ли оно палиндромом.
Меч - а зачем?
Да, это палиндром.
```

Мы рассмотрели простой случай, когда модуль расположен в том же каталоге. В следующем уроке вы узнаете про stack, популярное средство для сборки проектов на Haskell. Если вы собираетесь работать над чем-то сложным, обязательно им воспользуйтесь. Тем не менее в любом случае полезно понимать, как происходит ручная компиляция многофайловых программ.

Проверка 34.3. Функция `Main.isPalindrome` нигде не используется, поэтому оставлять её в файле необязательно. Сможете ли вы переделать файл таким образом, чтобы после её удаления не приходилось указывать модуль при обращении к `Palindrome.isPalindrome`?

Итоги

Целью этого урока было научить вас использованию модулей. Вы узнали, что большинство ваших программ автоматически помещалось в модуль `Main`. Затем вы увидели, что код можно дробить на несколько файлов, а потом компилировать их в единую программу. Также вы научились выборочно экспортировать определённые функции из модулей, пряча остальные. Давайте проверим, как вы поняли пройденный материал.

Задача 34.1. В модуле 4 упоминалось, что при работе с текстом `Data.Text` предпочтительнее `String`. Переделайте проект из этого урока, заменив `String` на `Data.Text` (в модуле `Main` и в модуле `Palindrome`).

Задача 34.2. В уроке 25 вы написали программу для искажения изображений. Вернитесь к этой программе и перенесите весь код, связанный с созданием искажений, в отдельный модуль `Glitch`.

Ответ 34.3 Измените строку `import qualified Palindrome` на `import Palindrome`, после чего избавьтесь от префикса `Palindrome.` в `Palindrome.isPalindrome`.

35

Сборка проектов при помощи stack

После прочтения урока 35 вы:

- научитесь работать с инструментом для сборки проектов stack;
- будете собирать проекты с помощью stack;
- сможете настраивать конфигурационные файлы, генерируемые stack.

При переходе от первоначального изучения языка программирования к использованию его для решения более-менее серьёзных задач одной из наиболее важных вещей становится автоматическая сборка. Универсальным решением является использование инструментов наподобие GNU Make. Но у множества языков есть собственные средства для сборки: для Java есть инструменты промышленного уровня Ant и Maven, для Scala — sbt, а для Ruby — rake. Учитывая академическое происхождение Haskell, может оказаться неожиданным, что у этого языка тоже есть мощная система сборки stack, которая хотя и является относительно новым дополнением в экосистеме языка, но уже успела произвести колоссальный эффект. Система stack автоматизирует и контролирует несколько этапов разработки проектов на Haskell, в частности она:

- предоставляет отдельное окружение с корректной версией компилятора GHC для каждого проекта;
- управляет установкой пакетов и их зависимостей;
- автоматизирует сборку проектов;
- помогает в подготовке и запуске тестов для проектов.

В этом уроке будет рассказано об основах создания и сборки проектов с использованием stack. В предыдущем уроке вы увидели, как с помощью модулей можно разделить код на несколько файлов. А в этом уроке

вы будете работать с тем же проектом, но с двумя важными изменениями. Во-первых, вместо `String` будет использоваться `Data.Text`, так как, как было уже подмечено в модуле 4, `Data.Text` предпочтительнее для работы с обычным текстом, чем `String`. Во-вторых, в этот раз вместо компиляции модулей мы воспользуемся stack. Утилита stack является одним из компонентов Haskell Platform (этот пакет программ рекомендовался в первом уроке), вы также можете установить её отдельно, загрузив с сайта `https://docs.haskellstack.org/`. Так как stack управляет установкой различных версий GHC и GHCi, то для начала работы с Haskell достаточно установить только его.

Обратите внимание. Как и все остальные языки, со временем Haskell претерпевает изменения. Как вы можете быть уверены в том, что код, написанный сегодня, будет нормально компилироваться через 5 лет?

35.1. Создание нового проекта stack

Первым делом нужно проверить, используете ли вы актуальную версию stack. Это можно сделать с помощью команды `stack update`:

```
$ stack update
```

Вероятно, выполнение этой операции в первый раз (или после длительного периода простоя) займёт много времени. Так как stack устанавливает новое окружение для сборки проектов, выполняется множество операций, что может занять несколько минут. Но будьте уверены, при последующем использовании stack эти действия будут выполняться намного быстрее, так как stack прекрасно умеет распоряжаться ресурсами.

После обновления можно создать первый проект. Для этого потребуется сначала воспользоваться командой `new` и указать название проекта:

```
$ stack new palindrome-checker
```

Запуск этой команды приведёт к созданию нового проекта. После её выполнения вы увидите папку palindrome-checker. Если вы в неё заглянете, то увидите следующий набор каталогов и файлов:

```
LICENSE                 src             Setup.hs
stack.yaml              test            app
palindrome-checker.cabal
```

Чуть дальше мы прольём свет на предназначение всего этого.

35.2. Разбор структуры проекта

Когда вы запустили `stack new`, stack создал новый проект по шаблону. В качестве аргумента для этой команды не было ничего передано, поэтому был использован стандартный шаблон. Для начала поработаем с этим шаблоном, но существует и множество других вариантов (некоторые можно найти по адресу `https://github.com/commercialhaskell/stack-templates`).

35.2.1. Cabal-файл проекта и автоматически создаваемые файлы

В корневом каталоге проекта присутствуют следующие файлы, которые сгенерировал stack:

- LICENSE;
- Setup.hs;
- palindrome-checker.cabal;
- stack.yaml.

Пока займёмся конфигурационным файлом palindrome-checker.cabal, содержащим все относящиеся к проекту метаданные. В его шапке находится основная информация об имени проекта, версии, его описании и т. п.:

```
name:                palindrome-checker
version:             0.1.0.0
synopsis:            Initial project template from stack
description:         Please see README.md
homepage:            https://github.com/githubuser/palindrome-
                                                ↳ checker#readme
license:             BSD3
license-file:        LICENSE
author:              Author name here
maintainer:          example@example.com
copyright:           2016 Author name here
```

Один из разделов этого файла включает информацию о том, где находятся файлы библиотек проекта, какие библиотеки потребуются для работы и какая версия Haskell используется:

```
library
    hs-source-dirs:   src
    exposed-modules:  Lib
    build-depends:    base >= 4.7 && < 5
    default-language: Haskell2010
```

Посмотрите на строки `hs-source-dirs` и `exposed-modules`. В строке `hs-source-dirs` определяет подкаталог, в котором расположены файлы библиотек. Значение этого параметра по умолчанию — `src`. Вы можете обратить внимание на то, что stack уже успел сгенерировать эту папку. А в параметре `exposed-modules` перечислены используемые библиотеки. По умолчанию stack создаёт модуль `Lib`, который можно найти по адресу src/Lib.hs. Вы можете добавить новые значения к `exposed-modules`, поместив их на разные строки и разделив значения запятыми, следующим образом:

```
exposed-modules: Lib,
                 Palindrome,
                 Utils
```

Когда вы только начнёте работать со stack, особенно с небольшими проектами, можно хранить весь библиотечный код в src/Lib.hs. Параметр `build-depends` будет рассмотрен в следующем разделе, а о значении параметра `default-language` в большинстве случаев можно не волноваться.

Также в конфигурации указано, где лежат файлы, которые будут использованы при сборке исполняемых файлов, имя модуля `Main` и параметры командной строки по умолчанию, которые будут использованы при запуске программы:

```
executable palindrome-checker-exe
    hs-source-dirs:     app
    main-is:            Main.hs
    ghc-options:        -threaded -rtsopts -with-rtsopts=-N
    build-depends:      base
                      , palindrome-checker
    default-language:   Haskell2010
```

Утилита stack разделяет код для библиотек и для запуска программ, помещая их в разные каталоги. Значение `hs-source-dirs` указывает на папку, в которой находится модуль `Main`. По аналогии с секцией `library` в разделе `executable` есть параметр `main-is`, который показывает, где находится функция `main`. Ещё раз, stack автоматически создаёт папку для проекта (определённую параметром `hs-source-dirs`) и помещает туда Main.hs.

В этом файле ещё много различных настроек, но тех основ, которые мы уже обсудили, должно хватить для создания новых проектов. Далее мы рассмотрим несколько папок и файлов с исходным кодом, созданных stack. Мы будем указывать на другие важные и интересные части cabal-файла по мере работы с проектами из оставшейся части книги, использующими возможности stack.

Проверка 35.1. Укажите своё имя в качестве автора проекта.

35.2.2. Каталоги app, src и test

Также stack автоматически создал три каталога: app, src и test. В предыдущем разделе вы узнали, что app и src созданы для модулей библиотек и исполняемых файлов. Проигнорируем пока test, поскольку тестирование будет подробно обсуждаться в уроке 36. Мы также упомянули, что stack самостоятельно включает два файла в эти папки: app/Main.hs и src/Lib.hs. Эти файлы и каталоги служат частью стандартного шаблона для минимального проекта. Автоматически созданный файл Main.hs выглядит так.

Листинг 35.1 Генерируемый по умолчанию модуль Main

```
module Main where

import Lib

main :: IO ()
main = someFunc
```

Это простой файл, но он даёт информацию о том, как устроены проекты stack. Единственным модулем, который импортирует `Main`, является `Lib`, а затем в `Main` определяется `IO`-действие `main`, содержащее вызов функции `someFunc`. Откуда появилась эта функция? Она определена в модуле `Lib`, который расположен в src/Lib.hs и выглядит следующим образом.

Листинг 35.2 Стандартный модуль Lib, генерируемый stack

```
module Lib
( someFunc
) where

someFunc :: IO ()
someFunc = putStrLn "someFunc"
```

Ответ 35.1. Измените строку cabal-файла, задающую имя автора:

```
author: Will Kurt
```

Функция `someFunc` тривиальна, она печатает `"someFunc"`. Хотя эти файлы и не особенно сложны, они позволяют понять, как начать разработку проекта, используя stack. Модуль `Main` должен быть очень лаконичным и в основном полагаться на библиотечные функции, определённые в `Lib`. Теперь давайте перенесём проект из предыдущего урока на stack!

Проверка 35.2. Мы ещё не рассмотрели сборку проектов на stack, но как вы думаете, что должно произойти, если запустить проект, созданный по умолчанию?

35.3. Написание кода

Начнём с модуля `Palindrome`. В отличие от прошлого раза, сейчас нужно писать библиотеку, которая вместо `String` использует `Data.Text`. Так как вы применяете stack, файл модуля должен быть помещён в папку src. На этот раз вместо создания файла Palindrome.hs вы будете работать с файлом Lib.hs. Для таких простых программ, как эта, вы спокойно можете помещать все вспомогательные функции в один модуль.

Листинг 35.3 Изменение модуля Palindrome для работы с Text

```
{-# LANGUAGE OverloadedStrings #-}

module Lib
( isPalindrome
) where

import qualified Data.Text as T
import Data.Char (toLower,isSpace,isPunctuation)

stripWhiteSpace :: T.Text -> T.Text
stripWhiteSpace text = T.filter (not . isSpace) text

stripPunctuation :: T.Text -> T.Text
stripPunctuation text = T.filter (not . isPunctuation) text
```

Ответ 35.2. Программа должна напечатать `someFunc`.

```
preprocess :: T.Text -> T.Text
preprocess = stripWhiteSpace . stripPunctuation . T.toLower

isPalindrome :: T.Text -> Bool
isPalindrome text = cleanText == T.reverse cleanText
    where cleanText = preprocess text
```

Далее вам нужно будет написать код для модуля `Main`. На этот раз мы не будем пользоваться квалифицированным импортом (но придётся внести небольшие правки, чтобы код работал с `Data.Text`). Так как `Main` — ключевой файл для исполняемой программы, он находится в папке app (это указано в cabal-файле проекта). Перепишем сгенерированный файл Main.hs следующим образом.

Листинг 35.4 Main.hs для программы, определяющей палиндромы

```
{-# LANGUAGE OverloadedStrings #-}

module Main where

import Lib
import Data.Text as T
import Data.Text.IO as TIO

main :: IO ()
main = do
    TIO.putStrLn "Введите слово, чтобы узнать, является ли оно
                                                  ↳ палиндромом."
    text <- TIO.getLine
    let response = if isPalindrome text
                   then "Да, это палиндром."
                   else "Нет, это не палиндром."
    TIO.putStrLn response
```

Теперь всё почти готово, осталось только немного исправить cabal-файл. Нужно указать stack, от каких модулей теперь зависит проект. И для Main.hs, и для Lib.hs требуется `Data.Text`. Так что в оба раздела нужно добавить пакет text в качестве зависимости:

```
library
    hs-source-dirs:      src
    exposed-modules:     Lib
    build-depends:       base >= 4.7 && < 5
                       , text
    default-language:    Haskell2010
```

```
executable palindrome-checker-exe
    hs-source-dirs:      app
    main-is:             Main.hs
    ghc-options:         -threaded -rtsopts -with-rtsopts=-N
    build-depends:       base
                       , palindrome-checker
                       , text
    default-language:    Haskell2010
```

Теперь всё готово для сборки проекта!

Проверка 35.3. Если бы вы хотели сохранить имя модуля Palindrome и название соответствующего файла, то какие изменения пришлось бы внести в cabal-файл?

35.4. Сборка и запуск вашего проекта

Наконец-то у вас всё готово для сборки проекта. Первая команда, которую нужно будет запустить, — это `stack setup`. В корневой папке проекта просто введите:

```
$ stack setup
```

Эта команда проверит, установлена ли корректная версия GHC. Для таких небольших программ, как наша, это не особенно важно, но из-за изменчивой природы Haskell очень важно, чтобы проект собирался с помощью той версии компилятора, для которой создавался. Определение предпочтительной версии GHC происходит не напрямую, а с помощью указания специального значения поля `resolver` в файле stack.yaml:

```
resolver: lts-7.9
```

Ответ 35.3. Смените значение `exposed-modules` на `Palindrome`:

```
library
    hs-source-dirs:      src
    exposed-modules:     Palindrome
```

Эта книга была подготовлена с помощью версии lts-7.9, где используется GHC версии 8.0.1. По умолчанию stack использует наиболее свежую и стабильную версию `resolver`. В большинстве случаев это как раз то, что и требуется, но если вы столкнётесь с проблемами при сборке проектов из этой книги, то смена версии на lts-7.9, скорее всего, исправит ситуацию. Вы можете найти список текущих версий на `www.stackage.org`, а информация о конкретной версии может быть найдена с помощью добавления версии к ссылке (например, `www.stackage.org/lts-7.9` содержит информацию об lts-7.9).

Далее нужно собрать проект. Это делается с помощью команды:

```
$ stack build
```

Не волнуйтесь, если эта команда займёт много времени при первом использовании stack для сборки проекта. Последующие сборки будут проходить гораздо быстрее.

После запуска этой команды у вас будет всё готово для запуска проекта. В прошлом вам приходилось вручную использовать GHC для компиляции кода в исполняемый файл, а затем запускать этот файл. У stack же другой подход — для запуска нужно воспользоваться командой `exec`. Этой команде нужно передать название исполняемого файла, которое определено в palindrome-checker.cabal параметром `executable` в секции исполняемых файлов:

```
executable palindrome-checker-exe
```

Поэтому запускать проект следует так:

```
$ stack exec palindrome-checker-exe
Введите слово, чтобы узнать, является ли оно палиндромом.
Меч - а зачем?
Да, это палиндром.
```

Великолепно, программа работает!

35.4.1. Бонус: избавление от языковых расширений

В нашей программе, да и в любой крупной программе, в которой используется `Data.Text`, есть одна раздражающая деталь: приходится постоянно включать расширение `OverloadedStrings` почти в каждом файле. К счастью, в stack есть возможность избежать этого. Вы можете добавить всего одну строку в palindrome-checker.cabal, чтобы включить расширение

`OverloadedStrings` для всего проекта. Добавьте следующую строку после `default-language: Haskell2010` в секцию для библиотек и в секцию для исполняемых файлов:

```
extensions: OverloadedStrings
```

Использование этого параметра избавит вас от необходимости добавлять директиву `LANGUAGE` или помнить нужные флаги компиляции. Можете теперь снова собрать проект, чтобы убедиться, что всё работает.

Итоги

В этом уроке нашей целью было научить вас работать со stack для управления пакетами Haskell и для их сборки. Вы научились этому, создав новый проект stack и изучив структуру автоматически созданных для вас файлов. Затем вы перенесли код из предыдущего урока в stack-проект. Наконец, вы собрали проект, используя более надёжный и удобный метод. Давайте проверим, как вы усвоили информацию из этого урока.

Задача 35.1. Внесите в проект следующие изменения:

- поставьте корректные значения параметров, задающих имя автора проекта, его почту и описание проекта;
- измените определение Lib.hs на первоначальное, созданное stack;
- добавьте функцию `isPalindrome` в модуль `Palindrome`, находящийся в src/Palindrome.hs;
- убедитесь, что расширение `OverloadedStrings` включено для всего проекта.

Задача 35.2. Переработайте код из урока 21 (модуль 4), сравнивающий цену двух пицц, в stack-проект. Весь вспомогательный код должен быть либо в Lib.hs, либо в дополнительных модулях.

36

Тестирование свойств с помощью QuickCheck

После прочтения урока 36 вы:

- начнёте применять `stack ghci` для работы со stack-проектами;
- будете запускать тестирование с помощью `stack test`;
- научитесь использовать QuickCheck для тестирования свойств;
- сможете устанавливать пакеты посредством `stack install`.

В предыдущем уроке вы познакомились с мощным средством сборки проектов stack и успели собрать проект, но немного схитрили. В проекте использовался код, который был написан при обучении работе с модулями. Теперь мы снова будем работать с палиндромами, но на этот раз разработка будет вестись с нуля, и сфокусируемся мы на тестировании кода. До сих пор вы тестировали функции вручную, но stack позволяет создавать автоматические тесты. Начнётся этот урок с использования GHCi, встроенного в stack, для ручного тестирования модулей. Потом вы научитесь запускать простые написанные вами модульные тесты с помощью `stack test`. А затем поработаете с тестированием свойств и познакомитесь с потрясающим инструментом QuickCheck, который позволит вам быстро генерировать наборы тестов.

Вы могли обратить внимание, что весь материал этого модуля отдалённо знаком, чего не скажешь о том, с чем вы успели столкнуться до этого. Даже QuickCheck является всего лишь мощным инструментом тестирования, трудно назвать эту библиотеку особенно сложной, скорее, интересной и полезной. Этот материал может показаться вам знакомым, так как stack был разработан людьми, занимающимися промышленной разработкой, которым была важна удобная система поддержки и распростране-

ния кода. Очень вероятно, что любой распространённый способ разработки, который вы предпочитаете (например, разработка через тестирование или BDD-методология), можно использовать с помощью stack. Благодаря таким инструментам, как stack, Haskell можно сейчас с удобством использовать для промышленной разработки так, как никогда ранее. Этот модуль предоставит вам только небольшой вводный курс по разработке программ с помощью Haskell и stack. Для более детальной информации посетите `haskellstack.org` и `stackage.org`.

Обратите внимание. Как при разработке проектов с помощью stack можно взаимодействовать с кодом, чтобы проверить, работает ли всё так, как было задумано?

36.1. Создание нового проекта

Давайте начнём новый проект, притворившись, что это абсолютно новая задача. Проблема остаётся той же, поэтому можно сосредоточиться на разработке программ с использованием stack, а не на функциональности кода. Так как в этом уроке мы интересуемся тестированием, назовём проект palindrome-testing. Вот команда, которая создаст новый проект:

```
$ stack new palindrome-testing
```

Прежде чем приступить к модулю `Main`, давайте посмотрим, как начать реализовывать требуемый функционал в src/Lib.hs. Утилита stack создала модуль `Main` по умолчанию, поэтому вам может потребоваться немного его подчистить, поскольку придётся переписать Lib. Замените вывод функции `someFunc` на традиционное `"Привет, мир!"`.

Листинг 36.1 Корректировка Main.hs для начала работы с Lib.hs

```
module Main where

import Lib

main :: IO ()
main = putStrLn "Привет, мир!"
```

В этом проекте мы по большей части будем работать в src/Lib.hs, реализуя библиотечные функции, которые в итоге будут использованы в `Main`.

Начнём с написания простейшей реализации `isPalindrome`, как пришлось бы поступить при разработке проекта с нуля.

Листинг 36.2 Простейшее определение isPalindrome

```
module Lib
    ( isPalindrome
    ) where

isPalindrome :: String -> Bool
isPalindrome text = text == reverse text
```

Теперь, когда часть кода уже написана, вы можете начать его тестирование с помощью stack.

Проверка 36.1. Вам нужно будет добавить всего несколько функций в модуль Lib, а потом экспортировать их все. Как в этом случае лучше определить модуль?

36.2. Разные виды тестирования

Когда мы говорим о тестировании кода, мы часто имеем в виду модульное тестирование отдельных потенциально опасных случаев. Но тестирование не всегда подразумевает модульное тестирование. Каждый раз, когда вы загружаете код в GHCi, вы тоже тестируете код, просто делаете это вручную. Этот раздел начнётся с обзора возможностей GHCi, встроенного в stack, для проверки корректности кода. Затем мы рассмотрим использование `stack test` для автоматического запуска простых модульных тестов, которые вы напишете. А в заключение вы увидите, что Haskell предлагает мощную альтернативу модульным тестам, которая называется *тестирование свойств*. Если модульное тестирование существенно автомати-

Ответ 36.1

```
module Lib where

isPalindrome :: String -> Bool
isPalindrome text = text == reverse text
```

зирует обычные тесты, то тестирование свойств позволяет автоматизировать модульные тесты.

В этом уроке будет рассматриваться традиционный подход к тестированию. Начинается такой подход с написания кода, затем следует ручная проверка, и только потом пишутся более формальные тесты.

В Haskell нет никаких ограничений, принуждающих следовать при разработке именно такой последовательности действий. Если вы придерживаетесь разработки через тестирование (test-driven development, TDD), то запросто можете начать разбирать этот урок с конца, написав сначала тесты. Если же вы предпочитаете методологию BDD, популяризованную RSpec для Ruby, то в Haskell есть аналогичная библиотека для тестирования Hspec (она здесь не рассматривается, но работа с этой библиотекой после завершения модуля должна быть вполне понятна.)

36.2.1. Ручное тестирование и использование GHCi из stack

Так как вы используете stack, то можете запускать GHCi немного по-другому. Во-первых, запустите и соберите ваш проект, чтобы проверить, всё ли работает нормально:

```
$ cd palindrome-testing
$ stack setup
...
$ stack build
...
```

Из-за того, что stack создаёт безопасное, изолированное и переносимое окружение для проекта, вам вряд ли захочется вызывать `ghci` в командной строке, так как у каждого проекта есть свой набор библиотек и, возможно, своя версия GHC, установленная только для этого проекта. Для безопасного взаимодействия с проектом вам потребуется запустить `stack ghci`. В этом разделе вместо обычного диалога с `GHCi` будет подразумеваться диалог, вызываемый с помощью `stack ghci`:

```
$ stack ghci
*Main Lib>
```

Так как запуск GHCi был осуществлён с помощью stack, вы увидите в консоли, что загружены модули `Main` и `Lib`. Теперь вы можете протестировать имеющийся код:

```
*Main Lib> isPalindrome "доход"
True
```

```
*Main Lib> isPalindrome "кот"
False
*Main Lib> isPalindrome "доход!"
False
```

Ага, первая ошибка! Строка `"доход!"` является палиндромом, но программа заявляет, что это не так. Теперь вы можете исправить этот недочёт.

Листинг 36.3 Исправление isPalindrome по итогам тестирования

```
isPalindrome :: String -> Bool
isPalindrome text = cleanText == reverse cleanText
    where cleanText = filter (not . (== '!')) text
```

Для проверки вам даже не придётся снова запускать сборку проекта; достаточно выйти из GHCi и запустить его заново. Если коррективы были внесены только в файлы с кодом, а конфигурационные файлы были оставлены без изменений, то можно просто набрать `:r` в GHCi для перезагрузки кода в интерпретаторе без выхода из него:

```
*Main Lib> :r
*Main Lib> isPalindrome "доход!"
True
```

И теперь программа работает!

Проверка 36.2. Является ли палиндромом `"мат и тут и там"`?

36.2.2. Написания собственных модульных тестов и использование

Ручное тестирование прекрасно подходит для придания очертаний новым идеям. Но с ростом сложности проекта вам захочется как-то это автоматизировать. К счастью, в stack есть встроенная команда для запуска тестов. В каталоге test вы можете найти другой сгенерированный утилитой stack файл Spec.hs, в котором содержится заготовка с кодом.

Ответ 36.2

```
*Main Lib> isPalindrome "мат и тут и там"
True
```

Листинг 36.4 Заготовка для файла Spec.hs

```
main :: IO ()
main = putStrLn "Test suite not yet implemented"
```

Для Haskell уже есть пакеты модульного тестирования (Hspec — пакет, похожий на RSpec из Ruby, и HUnit — копия JUnit из Java), но пока что начнём с простого самописного фреймворка для модульного тестирования. Всё, что нужно определить, — это `IO`-действие `assert`, которое принимает `Bool` (в нашем случае тест для функции) и выводит либо сообщение об успехе, либо уведомление об ошибке.

Листинг 36.5 Простейшая функция для модульного тестирования

```
assert :: Bool -> String -> String -> IO ()
assert test passStatement failStatement =
     if test
     then putStrLn passStatement
     else putStrLn failStatement
```

Теперь вы можете внести в `main` несколько тестов. Также потребуется импортировать модуль `Lib`. Вот первый тестовый набор.

Листинг 36.6 Spec.hs с несколькими простыми модульными тестами

```
import Lib

assert :: Bool -> String -> String -> IO ()
assert test passStatement failStatement =
     if test
     then putStrLn passStatement
     else putStrLn failStatement

main :: IO ()
main = do
    putStrLn "Запуск тестов..."
    assert (isPalindrome "доход") "пройдено: 'доход'"
                                  "провал: 'доход'"
    assert (isPalindrome "доход!") "пройдено: 'доход!'"
                                   "провал: 'доход!'"
    assert ((not . isPalindrome) "кот") "пройдено: 'кот'"
                                        "провал: 'кот'"
    putStrLn "Готово!"
```

Чтобы запустить тесты, воспользуйтесь командой `stack test`:

```
$ stack test

Запуск тестов...
пройдено: 'доход'
пройдено: 'доход!'
пройдено: 'кот'
Готово!
```

Великолепно! Далее добавьте другой тест со словом `"доход."` (с точкой).

Листинг 36.7 Добавление другого теста в main

```
assert (isPalindrome "доход.") "пройдено: 'доход.'"
                               "провал: 'доход.'"
```

Если вы снова запустите тесты, то увидите, чего не хватает вашей функции `isPalindrome`:

```
Запуск тестов...
пройдено: 'доход'
пройдено: 'доход!'
пройдено: 'кот'
провал: 'доход.'
Готово!
```

Теперь можете исправить эту ошибку с помощью элементарной заплатки, переопределив `isPalindrome` ещё раз.

Листинг 36.8 Ещё одно исправление проблемы с isPalindrome

```
isPalindrome :: String -> Bool
isPalindrome text = cleanText == reverse cleanText
    where cleanText = filter (not . ( ` elem ` ['!',','])) text
```

Вы уже знаете, что правильным решением является использование функции `isPunctuation` из модуля `Data.Char`. Но подобный итерационный процесс исправления ошибок очень распространён, хотя он и не такой тривиальный. Если вы снова запустите тесты, вы увидите, что эта ошибка исправлена:

```
Запуск тестов...
пройдено: 'доход'
пройдено: 'доход!'
пройдено: 'кот'
пройдено: 'доход.'
Готово!
```

Впрочем, эта заплатка выглядит очень неубедительно, ведь ещё множество знаков пунктуации не обрабатывается корректно. Даже учитывая, что мы точно знаем, что `isPunctuation` является лучшим решением, для проверки придётся придумать и написать множество тестов: `"дох-од"`, `":доход"`, `"доход?"` и т. д. В следующем разделе вы увидите всю выразительную мощь другого типа тестирования, доступного при разработке на Haskell, называемого *тестированием свойств*. Это подход автоматизирует работу по созданию отдельных модульных тестов.

Проверка 36.3. Добавьте проверку `":доход:"` к списку тестов и перезапустите набор тестов.

36.3. Тестирование свойств с помощью QuickCheck

Прежде чем окунуться в тестирование свойств, давайте немного приведём библиотеку в порядок. Очевидно, что часть `isPalindrome`, отвечающая за создание `cleanText`, разрастётся до приличных размеров, так что неплохо было бы вынести этот кусок в функцию `preprocess`.

Ответ 36.3

```
main :: IO ()
main = do
    putStrLn "Запуск тестов..."
    assert (isPalindrome "доход") "пройдено: 'доход'"
                                  "провал: 'доход'"
    assert (isPalindrome "доход!") "пройдено: 'доход!'"
                                   "провал: 'доход!'"
    assert ((not . isPalindrome) "кот") "пройдено: 'кот'"
                                        "провал: 'кот'"
    assert (isPalindrome "доход.") "пройдено: 'доход.'"
                                   "провал: 'доход.'"
    assert (isPalindrome ":доход:") "пройдено: ':доход:'"
                                    "провал: ':доход:'"
    putStrLn "Готово!"
```

Это сработает, так как строка `":доход:"` является палиндромом даже с символами пунктуации.

Листинг 36.9 Код становится более аккуратным

```
module Lib
    ( isPalindrome
    , preprocess
    ) where

preprocess :: String -> String
preprocess text = filter (not . ( ‘ elem ‘ [’!’,’.’])) text

isPalindrome :: String -> Bool
isPalindrome text = cleanText == reverse cleanText
    where cleanText = preprocess text
```

Теперь можно сконцентрироваться на тестировании `preprocess`. Вам требуется организовать проверку `preprocess`, но, как мы уже заметили, написание модульных тестов может быть очень утомительным, да и можно упустить какие-то случаи.

36.3.1. Тестирование свойств

По сути, нам нужно проверить, обладает ли `preprocess` требуемыми свойствами. В основном требуется проверять сохранение некоторого инварианта, в данном случае *пунктуационного* (это такой прикольный способ сказать, что вывод не зависит от символов пунктуации во вводе).

Это свойство можно выразить с помощью функции. Вам потребуется импортировать `Data.Char (isPunctuation)` и поместить следующую функцию в Spec.hs.

Листинг 36.10 Представление проверяемого свойства функцией

```
prop_punctuationInvariant text = preprocess text ==
                                 preprocess noPuncText
    where noPuncText = filter (not . isPunctuation) text
```

Если вы переведёте текст этой функции на русский язык, то это будет именно то, чего вы пытаетесь достичь:

```
"Вызов preprocess на строке должен возвращать тот же ответ,
что и вызов на том же тексте, но без символов пунктуации"
```

Конечно, приятно иметь это свойство в виде кода, но пока что у нас нет способа протестировать его. Вам требуется автоматический способ получения множества возможных значений. Вот тут-то и пригодится QuickCheck.

Проверка 36.4. Реализуйте свойство `prop_reverseInvariant`, которое показывает очевидный факт, что результат `isPalindrome` должен быть одинаковым на строке и на её перевёрнутой версии.

36.3.2. Представляем вам QuickCheck

Библиотека QuickCheck построена вокруг идеи тестирования свойств. Функция `prop_punctuationInvariant` представляет собой первый пример такого свойства. QuickCheck работает следующим образом: вы предоставляете библиотеке определённые свойства, которыми должен обладать код, а она автоматически генерирует значения и вызывает для них функции, проверяя, выполнены ли наложенные условия. Теперь давайте заменим наши простые модульные тесты проверкой свойств.

Первым делом нужно добавить QuickCheck в список зависимостей (поле build-depends) cabal-файла.

Листинг 36.11 Изменение palindrome-testing.cabal

```
test-suite palindrome-testing-test
    type:             exitcode-stdio-1.0
    hs-source-dirs:   test
    main-is:          Spec.hs
    build-depends:    base
                    , palindrome-testing
                    , QuickCheck
    ghc-options:      -threaded -rtsopts -with-rtsopts=-N
    default-language: Haskell2010
```

Теперь вы можете добавить модуль `Test.QuickCheck` в шапку Spec.hs. Для использования QuickCheck вам нужно вызвать функцию `quickCheck` в `main`, передав ей нужное свойство.

Ответ 36.4

```
prop_reverseInvariant text = isPalindrome text
                          == (isPalindrome (reverse text))
```

Листинг 36.12 Использование quickCheck в Spec.hs

```
main :: IO ()
main = do
    quickCheck prop_punctuationInvariant
    putStrLn "Готово!"
```

Когда вы запустите тесты, будет выведена информация об ошибке (как и предполагалось):

```
Progress: 1/2***
Failed! Falsifiable (after 4 tests and 2 shrinks):
"\187"
```

При передаче значений функции `prop_punctuationInvariant` библиотека QuickCheck попробовала символ `'\187'`, который является знаком пунктуации Unicode. В некотором смысле QuickCheck автоматически создал для вас набор модульных тестов. Для демонстрации лёгкости, с которой QuickCheck отлавливает ошибки, давайте пойдём простым путём, изменив `preprocess` так, чтобы эта функция обрабатывала символ `'\187'`.

Листинг 36.13 Удовлетворяем замечание QuickCheck

```
preprocess :: String -> String
preprocess text = filter (not . ( ` elem ` ['!','.','\187']))
                         text
```

Запустив тесты, обнаруживаем, что QuickCheck снова недоволен:

```
Failed! Falsifiable (after 11 tests and 2 shrinks):
";"
```

Теперь проблемы из-за точки с запятой.

> **Примечание.** QuickCheck использует подобранные по определённым правилам, но случайные значения, поэтому у вас могут быть другие ошибки, когда вы запустите тесты у себя.

Давайте исправим ошибки с помощью функции `isPunctuation`.

Листинг 36.14 Корректировка обработки символов пунктуации

```
import Data.Char(isPunctuation)

preprocess :: String -> String
preprocess text = filter (not . isPunctuation) text
```

Теперь при запуске `stack test` выводится более приятная надпись:

```
OK, passed 100 tests.
```

Как вы могли понять из этого сообщения, QuickCheck, тестируя заданное свойство, попробовал 100 строк, и все они прошли проверку! 100 тестов кажутся вам слишком маленьким тестовым набором? Попробуем 1000 с помощью `quickCheckWith`. Для этого воспользуйтесь синтаксисом для записей, чтобы передать аргумент в функцию (см. урок 12).

Листинг 36.15 Задание количества тестов QuickCheck

```
main :: IO ()
main = do
    quickCheckWith stdArgs {maxSuccess = 1000}
                   prop_punctuationInvariant
    putStrLn "Готово!"
```

Если вы снова запустите тесты, то увидите, что QuickCheck по-прежнему сообщает об успехе, а количество пройденных тестов внушает доверие:

```
OK, passed 1000 tests.
```

Даже учитывая, что вы определили всего одно свойство `isPalindrome`, исчезла необходимость писать бесчисленные модульные тесты!

Проверка 36.5. Добавьте в `main` вызов `quickCheck` для свойства `prop_reverseInvariant`, определённого в предыдущем упражнении.

Ответ 36.5

```
prop_reverseInvariant text = isPalindrome text
                           == isPalindrome (reverse text)
main :: IO ()
main = do
    quickCheckWith stdArgs {maxSuccess = 1000}
                   prop_punctuationInvariant
    quickCheck prop_reverseInvariant
    putStrLn "Готово!"
```

36.3.3. Применение QuickCheck с разными типами и установка библиотек

Одной из самых сложных частей QuickCheck является генерация входных значений, которые используются для проверок. Все типы, для которых QuickCheck может автоматически создавать значения, должны быть представителями класса `Arbitrary`. Детальное определение этого класса типов выходит за рамки данной книги. И у нас для вас есть плохие новости: только для некоторых стандартных типов есть экземпляры `Arbitrary`. Но есть и хорошие новости: можно установить пакет, которые сильно расширит множество типов, с которыми может работать QuickCheck.

К примеру, по умолчанию для `Data.Text` нет экземпляра `Arbitrary`, поэтому QuickCheck не будет работать с этим типом. Проблему можно исправить с помощью установки пакета quickcheck-instances. Это можно сделать, применив команду `stack install`:

```
$ stack install quickcheck-instances
```

Эта команда установит в проект palindrome-testing новый пакет, после чего вы сможете пользоваться его возможностями.

Давайте теперь посмотрим, как следует изменить файл Lib.hs для работы с типом `Data.Text`.

Листинг 36.16 Рефакторинг модуля Lib по замене String на Data.Text

```
module Lib
    ( isPalindrome
    , preprocess
    ) where

import Data.Text as T
import Data.Char(isPunctuation)

preprocess :: T.Text -> T.Text
preprocess text = T.filter (not . isPunctuation) text

isPalindrome :: T.Text -> Bool
isPalindrome text = cleanText == T.reverse cleanText
    where cleanText = preprocess text
```

Не забудьте упомянуть пакет `text` в относящемся к библиотекам разделе `build-depends` cabal-файла проекта. В файл Spec.hs также нужно внести соответствующие изменения.

Листинг 36.17 Исправление Spec.hs для работы с Data.Text

```
import Lib
import Test.QuickCheck
import Test.QuickCheck.Instances
import Data.Char(isPunctuation)
import Data.Text as T

prop_punctuationInvariant text = preprocess text ==
                                 preprocess noPuncText
    where noPuncText = T.filter (not . isPunctuation) text

main :: IO ()
main = do
    quickCheckWith stdArgs { maxSuccess = 1000}
                   prop_punctuationInvariant
    putStrLn "Готово!"
```

Добавляем `text` и `quickcheck-instances` в раздел `build-depends` секции тестов, и, наконец, можно протестировать улучшенный код:

```
$ stack test
...
OK, passed 1000 tests.
```

Тут можно увидеть ещё одно преимущество тестирования свойств. Внесённые изменения были достаточно очевидны. Представьте, как много работы бы потребовалось, если бы пришлось вручную написать полный набор модульных тестов, а потом поменять везде их типы!

Итоги

В этом уроке нашей основной целью было показать вам, как тестировать код. Вы начали с ручной проверки с помощью `stack ghci`. Использование GHCi, предоставленного stack, гарантирует, что код будет корректно собираться. После ручного тестирования вы использовали `stack test` для сборки и запуска серии простых модульных тестов. Наконец, вы обобщили привычные модульные тесты, реализовав тестирование свойств с помощью QuickCheck. Давайте посмотрим, как вы освоили этот материал.

Задача 36.1. Допишите проект palindrome-testing так, чтобы он стал похож на проект из предыдущего урока. Затем добавьте больше проверок свойств функции `preprocess`, например проверку того, что от пробелов и заглавных букв результат не зависит.

37

Итоговый проект: библиотека для простых чисел

Этот итоговый проект включает в себя:

- сборку нового проекта с помощью stack;
- написание простой библиотеки для работы с простыми числами;
- проверку кода с помощью stack test и QuickCheck;
- изменение кода для исправления ошибок;
- добавление новых функций и тестов.

До сих пор в этом модуле вы были сосредоточены на одной проблеме — создании программ для работы с палиндромами. В этом итоговом проекте вы снова проделаете всю работу по созданию модулей и ещё лучше освоите stack в процессе работы над новым проектом. На этот раз вам предстоит создать библиотеку для работы с простыми числами. Вы сфокусируетесь на нескольких важных задачах:

- перечисление всех простых чисел меньше заданного числа;
- проверка числа на простоту;
- факторизация числа на произведение простых сомножителей.

Первым делом нужно разобраться, как создавать список простых чисел. Вот тип такого списка:

```
primes :: [Int]
```

Этого можно добиться, использовав решето для простых чисел, которое будет отбирать только простые числа. Тип этой функции можно представить следующим образом: берётся список `[Int]` потенциально простых

чисел, а возвращается список `[Int]` простых чисел. Эту работу будет выполнять функция `sieve`:

```
sieve :: [Int] -> [Int]
```

Теперь, обладая списком простых чисел, вы можете с лёгкостью проверять, является ли число простым, посмотрев, есть ли оно в списке. Обычно это можно было бы выразить с помощью типа `Int -> Bool`, но, так как некоторые числа не подходят для проверки на простоту (например, отрицательные), лучше возвращать `Maybe Bool`:

ЛАНЬ

```
isPrime :: Int -> Maybe Bool
```

Наконец, приступим к разложению чисел на произведение простых сомножителей. По тем же причинам, по которым `isPrime` возвращает `Maybe Bool`, `primeFactors` будет возвращать `Maybe [Int]`, список, представляющий возможные значения сомножителей:

```
primeFactors :: Int -> Maybe [Int]
```

ЛАНЬ

Весь этот проект будет собран при помощи stack, в процессе разработки вы будете писать тесты. Все идеи, которые будут использоваться в этом проекте, довольно простые, так как нашей основной целью является изучение разработки с использованием утилиты stack.

37.1. Создание нового проекта

Как обычно, первым шагом будет создание нового проекта с помощью команды `stack new`. Проект мы назовём `primes`:

```
$ stack new primes
```

Когда stack завершит выполнение этой команды, вас будет ждать новый каталог под проект. Зайдите в эту папку:

```
$ cd primes
```

Для освежения в памяти структуры проекта посмотрите на файлы и каталоги, созданные stack. Вот список каталогов:

- *app* — это каталог для модуля `Main`, по умолчанию содержит файл Main.hs;
- *src* — здесь все библиотечные файлы, по умолчанию содержит Lib.hs;
- *test* — каталог с кодом для тестов, по умолчанию содержит Spec.hs.

На рис. 37.1 приведён список файлов корневого каталога проекта:

- *primes.cabal* — файл, в котором указываются настройки по сборке проекта;
- *LICENSE* — текст лицензии, под которой ваша библиотека будет распространяться;
- *stack.yaml* — содержит дополнительные конфигурационные данные;
- *Setup.hs* — файл, используемый системой Cabal, его можно игнорировать.

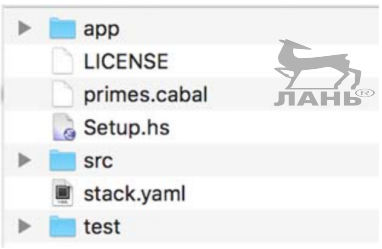


Рис. 37.1: Файлы, созданные утилитой stack

Этого богатства вполне хватит, чтобы начать работать над кодом!

37.2. Изменение файлов, созданных по умолчанию

Утилита stack по умолчанию создаёт небольшой набор файлов: файлы app/Main.hs и src/Lib.hs содержат несколько функций, предоставленных в модельных целях. Так как нас заботят по большей части библиотечные функции, начнём с изменения app/Main.hs, чтобы в этом файле был необходимый минимум — импорт модуля `Primes`, над которым мы будем в дальнейшем работать.

Листинг 37.1 Новая версия модуля Main в app/Main.hs

```
module Main where

import Primes

main :: IO ()
main = return ()
```

Файл Lib.hs следует заменить на Primes.hs

Функция someFunc больше не потребуется, заменим её на пустой кортеж

Далее переключимся на src/Lib.hs. Так как вы будете трудиться над библиотекой для работы с простыми числами, название файла лучше изменить на Primes.hs, а название модуля — на `Primes`. Работу начнём с создания списка-болванки с простыми числами, который можно будет использовать с другими функциями. Сначала это будет просто список всех натуральных чисел, а затем мы его переделаем.

Листинг 37.2 Изменение src/Lib.hs на src/Primes.hs

```
module Primes where

primes :: [Int]
primes = [1 .. ]
```

Не забудьте, что вам также нужно сообщить stack, что название файла библиотеки было изменено. Для этого откройте файл primes.cabal, найдите раздел library и внесите изменение в значение поля `exposed-modules`.

Листинг 37.3 Отражение переименования модуля в primes.cabal

```
library
    hs-source-dirs:     src
    exposed-modules:    Primes
    build-depends:      base >= 4.7 && < 5
    default-language:   Haskell2010
```

Ранее здесь был указан модуль `Lib`

Для подстраховки давайте установим окружение и соберём проект:

```
$ stack setup
...
$ stack build
...
```

На данном этапе вы не сделали ничего сложного, так что если вы получите ошибку, то она, скорее всего, вызвана опиской или тем, что вы забыли сохранить какой-то изменённый файл.

37.3. Реализация основных библиотечных функций

Теперь, когда построен фундамент проекта, можно приступить к написанию кода. Первым делом начнём работу с генерации списка простых чисел, который будет использоваться другими функциями. В модуле 5 вы уже увидели, как использовать операцию <*> из класса типов `Applicative` для создания списка простых чисел, но этот способ неэффективен. На этот раз вам предстоит познакомиться с более эффективным алгоритмом, который называется *решето Эратосфена*. Основная идея этого алгоритма заключается в обходе списка и фильтрации всех чисел, за исключением простых. Сначала возьмём 2, это число является первым простым, а затем удалим все числа, которые без остатка делятся на 2. Продолжим, взяв следующее

простое число 3. Теперь удалим все числа, кратные 3. Вот пошаговое описание действий алгоритма для всех чисел, которые меньше 10:

(1) начнём со списка чисел от 2 до 10: `[1,2,3,4,5,6,7,8,9,10]`;

(2) возьмём следующее число 2 и поместим его в список простых чисел, удалив все чётные числа, теперь наши списки выглядят так: `[2]` и `[3,5,7,9]`;

(3) возьмём число 3, добавим в список простых и удалим все числа, которые делятся на 3, из второго списка: `[2,3]` и `[5,7]`;

(4) повторим процесс с числом 5: `[2,3,5]` и `[7]`;

(5) и наконец, с числом 7: `[2,3,5,7]` и `[]`.

Эту функцию можно реализовать рекурсивно. Как и в случае с большинством рекурсивных функций, сигналом к завершению служит достижение конца списка. В противном случае нужно повторить описанные ранее шаги. Вот реализация решета на Haskell (эту функцию нужно поместить в файл src/Primes.hs).

Листинг 37.4 Рекурсивная реализация решета Эратосфена

```
sieve :: [Int] -> [Int]
sieve [] = []
sieve (nextPrime:rest) = nextPrime : sieve noFactors
    where noFactors = filter (not . (== 0)
                                   . (`mod` nextPrime)) rest
```

Теперь можете воспользоваться `stack ghci` для простой проверки вашей новой функции:

```
GHCi> sieve [2 .. 20]
[2,3,5,7,11,13,17,19]
GHCi> sieve [2 .. 200]
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,
↳ 79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,
↳ 157,163,167,173,179,181,191,193,197,199]
```

Но решето не было нашей конечной задачей. Теперь вам предстоит использовать эту функцию для генерации списка простых чисел, который можно будет использовать с другими функциями, к примеру с `isPrime`.

37.3.1. Построение списка простых чисел

Вы можете спокойно заменить список-заглушку на список, созданный с помощью решета Эратосфена. Это значение можно записать очень легко.

Листинг 37.5 Создание списка всех простых чисел

```
primes :: [Int]
primes = sieve [2 .. ]
```

В функции `primes` генерируется список всех простых чисел, которые меньше, чем максимальное значение `Int`. Вы можете увидеть, насколько оно велико, вызвав `maxBound` для `Int` в GHCi:

```
GHCi> maxBound :: Int
9223372036854775807
```

Даже с более эффективным способом вычисления простых чисел нежелательно предоставлять пользователям библиотеки основанный на этом алгоритме способ проверки на простоту больших чисел, так как это займёт много времени. Кроме того, существует около 2×10^{17} простых чисел, которые меньше `maxBound`! Если вы наивно предполагали, что значения типа `Int` занимают 4 байта (что не соответствует истине), то вам бы потребовалось 800 петабайт для хранения этого списка! Ленивые вычисления очень помогают в подобных ситуациях, но было бы нежелательно дать пользователям возможность случайно запросить такой большой объём памяти.

Вместо этого вы начнёте работу с более рационально выбранной верхней гранью для поиска простых чисел. В этом простом примере будут использоваться простые числа меньше 10 000.

Листинг 37.6 Список простых чисел разумного размера

```
primes :: [Int]
primes = sieve [2 .. 10000]
```

Можете немного поиграть с этим списком в GHCi:

```
GHCi> length primes
1229
GHCi> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

Если вы решите поднять верхнюю грань (например, до 100 000), то может произойти кое-что досадное. При первом использовании этого списка вычисление ответа займёт очень много времени. Частично это связано с тонкостями ленивых вычислений при работе со списками. В большинстве других языков программирования вы бы реализовали решето с помощью массива и просто бы обновляли в нём значения. В модуле 7 вы узнаете, как добиться этого в Haskell.

37.3.2. Определение функции isPrime

Теперь, когда у вас есть список простых чисел, реализация функции `isPrime` выглядит достаточно тривиальной: нужно просто проверить, находится ли значение в списке. Начнём с типа функции:

```
isPrime :: Int -> Bool
```

Но с деталями сложнее. Вот несколько трудностей:

- как поступать с отрицательными числами?
- и что делать с числами, которые больше верхней грани списка?

Как преодолеть эти проблемы? Конечно, можно просто возвращать `False`. Но это не совсем корректно. Наибольшая трудность будет возникать, если пользователь передаст функции простое, но очень большое число, и получит неправильный ответ. К тому же хоть -13 и не простое, но нельзя сказать, что оно не является простым в том же смысле, что и число 4. Число 4 не простое, потому что оно является составным. Составным называется число, которое делится на два простых числа — в данном случае на 2 и на 2. А число -13 нельзя назвать простым, так как это понятие обычно не используется для отрицательных чисел.

Эта ситуация — идеальный полигон для применения `Maybe`. Вы будете возвращать `Just True` для простых чисел в нашем диапазоне, `Just False` для составных чисел и `Nothing` для всех упомянутых исключительных случаев. Вот немного более усложнённая реализация функции `isPrime`.

Листинг 37.7 Более корректная версия isPrime

```
isPrime :: Int -> Maybe Bool
isPrime n | n < 0 = Nothing
          | n >= length primes = Nothing
          | otherwise = Just (n `elem` primes)
```

Вы уже использовали `Maybe` для обработки нулевых значений, а здесь можете убедиться, что `Maybe` используют и для обработки отсутствующих в ином смысле значений. В данном случае может отсутствовать разумный результат, так как в некоторых случаях ответ не несёт никакого смысла.

После добавления `isPrime` в Primes.hs её можно проверить в GHCi:

```
GHCi> isPrime 8
Just False
GHCi> isPrime 17
Just True
```

```
GHCi> map isPrime [2 .. 20]
[Just True,Just True,Just False,Just True,Just False,Just True,
↳ Just False,Just False,Just False,Just True,Just False,
↳ Just True,Just False,Just False,Just False,Just True,
↳ Just False,Just True,Just False]
GHCi> isPrime (-13)
Nothing
```

Так как вы написали более сложное определение `isPrime`, чем предполагалось первоначально, теперь самое время приступить к созданию максимально строгого набора тестов.

37.4. Написание тестов для кода

Следующим шагом будет организация несколько более удобного тестирования, чем обычная проверка в GHCi, для этого очень пригодится инструмент для организации тестов из предыдущего урока QuickCheck. Давайте начнём с первоначальной установки. Во-первых, вам нужно будет отредактировать файл primes.cabal так, чтобы в разделе `test-suite` была указана библиотека QuickCheck. На этот раз пакет quickcheck-instances не потребуется, так как тесты будут затрагивать только значения типа `Int`, с которыми QuickCheck работает по умолчанию.

Листинг 37.8 Внесение QuickCheck в список зависимостей test-suite

```
test-suite primes-test
    type:             exitcode-stdio-1.0
    hs-source-dirs:   test
    main-is:          Spec.hs
    build-depends:    base
                    , primes
                    , QuickCheck
    ghc-options:      -threaded -rtsopts -with-rtsopts=-N
    default-language: Haskell2010
```

Единственное изменение, которое нужно внести для работы с QuickCheck

Следующим на очереди будет изменение test/Spec.hs. Добавим в него импорт необходимых модулей.

Листинг 37.9 Импорт необходимых модулей в test/Spec.hs

```
import Test.QuickCheck
import Primes
```

```
main :: IO ()
main = putStrLn "Test suite not yet implemented"
```

Даже учитывая, что вы ещё не написали никаких тестов, лучше запустить `stack test`, чтобы убедиться, что всё работает, как положено:

```
$ stack test
Test suite not yet implemented
```

Если вы увидели похожее сообщение, то всё нормально работает. Пришло время приступить к описанию простых свойств функции `isPrime`.

37.4.1. Определение свойств isPrime

В первую очередь нужно проверить, возвращает ли функция корректные `Maybe`-значения. Помните, что для чисел больше, чем верхняя грань списка простых чисел, и меньше, чем 0, должно возвращаться значение `Nothing`; для любых других входных данных должно возвращаться `Just`-значение. Вам нужно импортировать `Data.Maybe`, чтобы можно было использовать функцию `isJust` для проверки того, является ли значение типа `Maybe` значением в обёртке `Just`. Вот определение `prop_validPrimesOnly`.

Листинг 37.10 Проверка получения Nothing или Just-значения

```
import Data.Maybe

prop_validPrimesOnly val =
    if val < 0 || val >= length primes
    then result == Nothing
    else isJust result
  where result = isPrime val
```

Всё, что вам требуется сделать, — это переписать IO-действие `main`, чтобы там запускалась проверка данного свойства.

Листинг 37.11 Добавление prop_validPrimesOnly в main

```
main :: IO ()
main = do
    quickCheck prop_validPrimesOnly
```

Прежде чем запустить тестирование, добавим ещё несколько тестов.

Проверка простоты наших «простых» чисел

Наиболее очевидным свойством простых чисел является их простота. Это можно сделать, создав список всех целых чисел, которые меньше данного числа, начиная с числа 2. Затем нужно проверить, делит ли какой-нибудь из элементов списка число без остатка. В данном тесте важен только случай, когда функция проверки на простоту возвращает `Just True`, сигнализируя о том, что число простое. Вот определение `prop_primesArePrime`.

Листинг 37.12 Тест на корректность выявления простых чисел

```
prop_primesArePrime val = if result == Just True
                          then length divisors == 0
                          else True
  where result = isPrime val
        divisors = filter ((== 0) . (val `mod` ))
                          [2 .. (val - 1)]
```

Этот способ проверки простых чисел не такой эффективный, как способ их генерации, но это нормально, учитывая, что тестовый набор не придётся часто запускать. Это великолепная демонстрация того, насколько полезным может быть тестирование свойств. Для тестирования крупного диапазона возможных входных значений можно использовать вот такие неэффективные, но простые для понимания методы.

Проверка того, что «непростые» числа являются составными

Завершающим тестом для `isPrime` будет проверка свойства, обратного предыдущему. Вам нужно проверить, что если функция возвращает `Just False`, то на вход ему подавалось число, у которого есть хотя бы один делитель меньше этого самого числа.

Листинг 37.13 Тест на корректность выявления составных чисел

```
prop_nonPrimesAreComposite val = if result == Just False
                                 then length divisors > 0
                                 else True
  where result = isPrime val
        divisors = filter ((== 0) . (val `mod` ))
                   [2 .. (val - 1)]
```

Это другой хороший пример свойства, которому должна удовлетворять ваша функция. Вместо тестирования на множестве подобранных вручную составных чисел вы описали, что же это означает — *быть непростым*.

Запуск тестов

Теперь остаётся добавить ещё несколько вызовов `quickCheck` в `main`. На этот раз воспользуемся функцией `quickCheckWith` и запустим 1000 прогонов, а не обычные 100. В конце концов, в качестве ввода может приниматься всё множество значений типа `Int`, так что нужно удостоверится, что проверяется достаточное количество чисел.

Листинг 37.14 Проверка дополнительных свойств в main

```
main :: IO ()
main = do
    quickCheck prop_validPrimesOnly
    quickCheckWith stdArgs {maxSuccess = 1000}
                   prop_primesArePrime
    quickCheckWith stdArgs {maxSuccess = 1000}
                   prop_nonPrimesAreComposite
```

Если запустить тесты, то вы увидите, что кое-что было упущено!

```
+++ OK, passed 100 tests.
+++ OK, passed 1000 tests.
*** Failed! Falsifiable (after 1 test):
0
```

Всё из-за того, что 0 не является составным! Когда вы решали, в каких случаях должно возвращаться значение `Nothing`, было решено, что это должно происходить с числами меньше нуля. Но тестирование привело к интересной проблеме: 0 и 1 не являются ни составными, ни простыми. Так как с этим разобраться? Возможно, самым большим преимуществом тестирования свойств является то, что у вас уже записан ответ. Вы определили, что функция `isPrime` должна возвращать `Just True` для простых чисел, а `Just False` — для составных. Это означает, что пользователь функции `isPrime` может спокойно рассчитывать на то, что `Just False` возвращается, только если число составное. Написанное свойство помогло обнаружить просчёт в рассуждениях и прийти к решению.

37.4.2. Исправление ошибки

Исправление этой ошибки будет достаточно банальным и простым. Нужно переделать `isPrime`, чтобы эта функция возвращала `Nothing` для всех значений меньше 2, а не 0.

Листинг 37.15 Исправление ошибки в isPrime

```
isPrime :: Int -> Maybe Bool
isPrime n | n < 2 = Nothing
          | n >= length primes = Nothing
          | otherwise = Just (n `elem` primes)
```

Вам также следует изменить свойство `prop_validPrimesOnly`, чтобы оно отражало внесённые изменения.

Листинг 37.16 Обновлённое свойства prop_validPrimesOnly

```
prop_validPrimesOnly val =
    if val < 2 || val >= length primes
    then result == Nothing
    else isJust result
  where result = isPrime val
```

Перезапустив набор тестов, можно удостовериться, что теперь всё работает нормально:

```
+++ OK, passed 100 tests.
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
```

С инструментарием для получения списка простых чисел и проверки на простоту вы можете переключиться на более интересную задачу — разложение числа на произведение простых сомножителей.

37.5. Написание кода факторизации чисел

Завершающим штрихом будет реализация функции для получения канонического разложения числа. Каноническое разложение числа — набор простых чисел, произведение которых равно этому числу. Пример:

```
4 = [2,2]
6 = [2,3]
18 = [2,3,3]
```

По тем же причинам, по которым `isPrime` возвращает `Maybe Bool`, каноническое разложение числа следует возвращать в виде `Maybe` списка целых чисел. Это поможет при обработке случаев отрицательных и слишком больших входных значений.

Начнём с создания небезопасной версии этой функции, которая будет возвращать не Maybe-список, а самый обычный список. Алгоритм достаточно прост. Начинаем с числа и списка простых чисел. Затем нужно проверить, делится ли число на простые числа из списка без остатка. Простые, на которые число не делится без остатка, нужно удалять из списка. Эту функцию можно рекурсивно определить следующим образом.

Листинг 37.17 Небезопасная версия функции факторизации

```
unsafePrimeFactors :: Int -> [Int] -> [Int]
unsafePrimeFactors 0 [] = []
unsafePrimeFactors n [] = []
unsafePrimeFactors n (next:primes) =
   if n `mod` next == 0
   then next:unsafePrimeFactors (n `div` next) (next:primes)
   else unsafePrimeFactors n primes
```

Теперь осталось всего лишь обернуть небезопасную функцию в код, который будет обрабатывать исключительные случаи.

Листинг 37.18 Безопасная обёртка для unsafePrimeFactors

```
primeFactors :: Int -> Maybe [Int]
primeFactors n
  | n < 2 = Nothing
  | n >= length primes = Nothing
  | otherwise = Just (unsafePrimeFactors n primesLessThanN)
  where primesLessThanN = filter (<= n) primes
```

Финальным шагом будет тестирование этих функций.

Наиболее очевидным свойством канонического разложения является то, что произведение чисел разложения должно быть равно данному числу. Это будет тестироваться с помощью свойства prop_factorsMakeOriginal.

Листинг 37.19 Проверка произведения чисел из разложения

```
prop_factorsMakeOriginal val =
    if result == Nothing
    then True
    else product (fromJust result) == val
  where result = primeFactors val
```

Ещё одной возможной ошибкой может быть присутствие составных чисел в разложении. Свойство prop_allFactorsPrime будет проверять, что

все полученные делители являются простыми. Так как функция `isPrime` уже была протестирована, ею можно воспользоваться для тестирования.

Листинг 37.20 Проверка элементов разложения на простоту

```
prop_allFactorsPrime val =
    if result == Nothing
    then True
    else all (== Just True) resultsPrime
  where result = primeFactors val
        resultsPrime = map isPrime (fromJust result)
```

Последний шаг — это обновление функции `main`.

Листинг 37.21 Запуск всех тестов в main из src/Spec.hs

```
main :: IO ()
main = do
    quickCheck prop_validPrimesOnly
    quickCheckWith stdArgs {maxSuccess = 1000}
                   prop_primesArePrime
    quickCheckWith stdArgs {maxSuccess = 1000}
                   prop_nonPrimesAreComposite
    quickCheck prop_factorsSumToOriginal
    quickCheck prop_allFactorsPrime
```

При запуске этих пяти функций для тестирования свойств, эквивалентных набору из 2300 модульных тестов, вы можете убедиться, что всё работает так, как предполагалось:

```
+++ OK, passed 100 tests.
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

Только что вы закончили ваш второй проект на stack, и на этот раз вы всё сделали за один заход. Как вы видите, stack — очень полезный инструмент, который помогает с лёгкостью писать код, который к тому же легко поддерживать. В следующем модуле вам предстоит пользоваться утилитой stack при написании всех практических примеров.

Итоги

В этом итоговом проекте вы:

- создали новый проект с помощью stack;
- изменили созданные по умолчанию файлы и конфигурацию проекта;
- написали собственную простую библиотеку и вручную её протестировали в GHCi;
- реализовали проверку свойств для поиска ошибок;
- воспользовались QuickCheck и `stack test` для локализации ошибок;
- переписали часть кода для исправления этих самых ошибок;
- расширили библиотеку с помощью новых функций и тестов.

Расширение проекта

Простейшим способом расширить проект является доработка файла app/Main.hs, чтобы пользователю предлагалось протестировать число на простоту или разложить его на множители, или сделать и то, и другое. Интересной проблемой может быть обработка значений `Nothing`, полученных от этих двух функций. Вот пример того, как может выглядеть такая программа:

```
$ stack exec primes-exe
Введите число, чтобы узнать, является ли оно простым:
4
Оно составное!
Введите число, чтобы получить его каноническое разложение:
100000000000
К сожалению, это число слишком велико.
```

Непростой задачей может стать реализация более совершенных алгоритмов проверки на простоту. Например, есть интересный вероятностный тест простоты, называемый тестом Миллера—Рабина. Реализация этого алгоритма позволит использовать вашу функцию `isPrime` с числами большего размера. В Википедии имеется подробное описание этого алгоритма: `https://ru.wikipedia.org/wiki/Тест_Миллера_—_Рабина`.

Модуль 7

Применение Haskell на практике

Добро пожаловать в последний модуль *Программируй на Haskell*! К этому моменту вы уже успели преодолеть долгий путь, начинавшийся с основ ссылочной прозрачности и преимуществ функционального программирования. Последний модуль отличается от большинства остальных. В этом модуле нашей целью будет облегчить переход от изучения языка программирования к написанию реальных программ. Я часто вижу, что изучающие Haskell (включая меня) считают переход от чтения материалов по Haskell к реализации повседневных вещей более сложным, чем они рассчитывали.

Чтобы облегчить такой переход, целью этого модуля будет дать вам представление о множестве разнообразных задач, которое предоставит крепкий фундамент для написания более крупных и сложных проектов. Начнёте вы этот модуль с изучения того, как Haskell обрабатывает ошибки, и затем познакомитесь с полезным типом под названием `Either`.

Потом вы реализуете три прикладных проекта. В первом вы займётесь написанием простых HTTP-запросов на Haskell, используя REST API. Взаимодействие с HTTP — невероятно большая часть того, с чем многие программисты сталкиваются каждый день. Даже если вы не заинтересованы в веб-разработке, в определённый момент вы обнаружите, что вам требуется получать какие-то данные из сети. Затем, используя наработки этого проекта, вы посмотрите, как можно читать данные в JSON в Haskell с помощью библиотеки Aeson. JSON является, возможно, самым распространённым форматом данных на сегодняшний день; необходимость его читать возникает в самых разнообразных проектах. После этого вы узнаете, как в Haskell можно работать с базами данных SQL, реализовав при этом утилиту командной строки для взаимодействия со свободной библиотекой для распространения различных инструментов. В рамках этого проекта мы обсудим все основные задачи по взаимодействию с базами данных.

В последнем уроке этого модуля вы попрощаетесь с данной книгой, изучив напоследок, как в Haskell выглядит нечто, для этого языка крайне непривычное, — классические алгоритмы на массивах. В этом уроке вам предстоит воспользоваться типом `STUArray` для реализации алгоритма

сортировки, который будет сохранять промежуточные результаты, в котором не будет ленивых вычислений, и который должен работать так же, как в не-функциональных языках программирования. Если вы сможете написать обычную сортировку пузырьком на чистом функциональном языке программирования, то сможете реализовать на Haskell абсолютно всё.

38

Ошибки в Haskell и тип Either

После прочтения урока 38 вы:

- сможете выбрасывать ошибки с помощью функции `error`;
- осознаете опасности выбрасывания ошибок;
- научитесь использовать `Maybe` в качестве метода обработки ошибок;
- разберётесь с применением типа `Either` для работы с ошибками.

Большая часть того, что делает Haskell таким мощным, основывается на его типобезопасности, поведение программ на нём легко предсказать, и на него можно положиться. Хотя Haskell упрощает, а бывает, полностью освобождает от множества проблем, ошибок при решении реальных проблем всё равно избежать не получится. В этом уроке вы узнаете, как рассуждать об обработке ошибок в Haskell. Традиционный подход выбрасывания исключений при программировании на Haskell воспринимается неодобрительно, так как он приводит к ошибкам во время выполнения, которые компилятор не может обнаружить. Хотя Haskell и позволяет выбрасывать ошибки, существуют лучшие способы решения проблем, с которыми вы можете столкнуться в своих программах. Вы уже достаточно много поработали с одним из таких методов, применяя тип `Maybe`. Но у этого типа есть проблема — у вас почти нет возможности сообщить о том, что именно пошло не так. В Haskell имеется более мощный тип `Either`, позволяющий сопроводить информацию об ошибке любым нужным значением.

В этом уроке вы будете использовать функцию `error` и типы `Maybe` и `Either` для разрешения исключительных ситуаций в программах. Начнём с изучения функции `head`. Хотя `head` — одна из первых изученных вами функций, её определение — достаточно сложный вопрос: при использовании `head` можно элементарно получить ошибку во время выполнения,

которая никак не сможет быть отслежена системой типов Haskell. В уроке вы рассмотрите несколько альтернативных путей обработки исключительных ситуаций при работе `head`. Эта проблема может быть решена более качественно с использованием типа `Maybe`, а больше информации об ошибке можно получить с помощью нового типа, который будет рассмотрен в уроке, `Either`. И заключительный этап — вы создадите простое приложение командной строки, которое будет проверять, является ли число простым. Вы воспользуетесь типом `Either` и собственным типом ошибок для их представления и демонстрации пользователю.

Обратите внимание. У вас есть список идентификаторов работников компании. Эти ID находятся в диапазоне от 0 и до 10 000. Также у вас есть функция `idInUse`, которая проверяет, занят ли определённый ID. Как вы можете составить функцию, которая с помощью `idInUse` отличает пользователя, отсутствующего в базе данных, от значения, находящегося вне диапазона корректных идентификаторов?

38.1. Функция head, частичные функции и ошибки

Одной из первых функций, с которой вы познакомились, была функция `head`. Эта функция возвращает вам первый элемент списка, если такой существует. Проблемы с этой функцией начинаются тогда, когда у списка нет первого элемента (в случае пустого списка). Посмотрите на рис. 38.1.

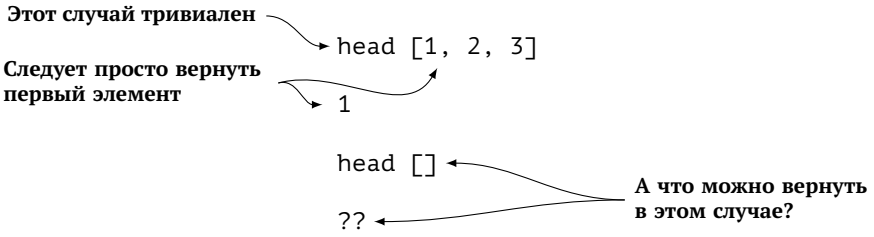


Рис. 38.1: Проблема с поведением `head` на пустом списке

Изначально функция `head` казалась невероятно полезной. Очень многие рекурсивные функции в Haskell реализованы с помощью списков, и запрос первого элемента списка является обычным делом. К сожалению, у `head` есть одна большая проблема. Когда вы передаёте этой функции пустой список, то получаете ошибку:

```
GHCi> head [1]
1
GHCi> head []
*** Exception: Prelude.head: empty list
```

В большинстве языков программирования выбрасывание исключений вроде этого является обычной практикой. Однако в Haskell это большая проблема, так как выбрасывание исключения делает код непредсказуемым. Одним из ключевых преимуществ Haskell является то, что программы на нём безопасны и предсказуемы. Но ничего в типовой аннотации `head` не сигнализирует о том, что что-то внезапно может пойти не так:

```
head :: [a] -> a
```

К этому моменту вы уже успели узнать из книги, что если программа на Haskell компилируется, то, скорее всего, она будет работать так, как предполагалось. Но `head` нарушает это правило, поощряя написание кода, который компилируется, но вызывает ошибку во время выполнения.

Предположим, что вы написали собственную функцию `myTake`, воспользовавшись в реализации функциями `head` и `tail`.

Листинг 38.1 Корректная, но подверженная ошибкам функция

```
myTake :: Int -> [a] -> [a]
myTake 0 _ = []
myTake n xs = head xs : myTake (n-1) (tail xs)
```

Давайте скомпилируем этот код, указав флаг компиляции `-Wall`, который требует от компилятора выдавать предупреждение обо всех потенциальных проблемах в коде. Это может быть сделано с помощью добавления `-Wall` к параметру `ghc-options` в разделе для исполняемых файлов cabal-файла проекта. В качестве напоминания материала из урока 35 откройте файл headaches.cabal в корневом каталоге проекта `headaches`, найдите раздел для исполняемых файлов и добавьте `-Wall` в список `ghc-options`:

```
executable headaches-exe
    hs-source-dirs:     app
    main-is:            Main.hs
    ghc-options:        -threaded -rtsopts -with-rtsopts=-N -Wall
    build-depends:      base
                      , headaches
    default-language:   Haskell2010
```

Флаг `-Wall` заставляет компилятор выводить все предупреждения во время компиляции программы

После внесённых изменений нужно перезапустить GHCi (что автоматически пересоберёт проект), но даже теперь вы не увидите жалоб компилятора. Тем не менее абсолютно очевидно, что этот код приводит к ошибке:

```
GHCi> myTake 2 [1,2,3] :: [Int]
[1,2]
GHCi> myTake 4 [1,2,3] :: [Int]
[1,2,3,*** Exception: Prelude.head: empty list
```

Представьте, что этот код активно участвует в обработке пользовательских запросов. Тогда эта ошибка была бы очень печальной, особенно для Haskell. Чтобы увидеть опасность функции `head`, давайте сравним нашу функцию с аналогичной, использующей сопоставление с образцом.

Листинг 38.2 Функция, вызывающая предупреждение компилятора

```
myTakePM :: Int -> [a] -> [a]
myTakePM 0 _ = []
myTakePM n (x:xs) = x : myTakePM (n-1) xs
```

Этот код идентичен `myTake`, но если вы скомпилируете его с `-Wall`, то получите довольно информативное предупреждение:

```
Pattern match(es) are non-exhaustive
    In an equation for 'myTakePM':
        Patterns not matched: p [] where p is not one of {0}
```

Это сообщение уведомляет вас, что функция не обрабатывает случай пустого списка! Хотя код очень похож на версию с `head`, GHC может предупредить вас об этом.

> **Примечание.** Если вы не хотите пропустить ни одного предупреждения в больших проектах, вы можете компилировать с ключом `-Werror`, что вызывает ошибку каждый раз, когда компилятор предупреждает о чём-то.

Проверка 38.1. Какой из образцов спасёт определение `myTakePM`?

```
myTakePM _0 [] = []
myTakePM _ [] = []
myTakePM 0 (x:xs) = []
```

Ответ 38.1. Вам нужно добавить следующий образец:

```
myTakePM _ [] = []
```

38.1.1. Head и частичные функции

Функция `head` даёт пример *частичной функции*. В уроке 2 вы узнали, что каждая функция должна принимать аргумент и возвращать результат. Частичные функции не нарушают этого требования, но у них есть один значительный недостаток. Они определены не для всех входных данных. Функция `head` не определена для пустого списка.

Практически все ошибки в программах являются результатом вызова частичных функций. Ваши программы получают на вход данные, обработка которых не была предусмотрена, и у программы нет способа справиться с подобной ситуацией. Выбрасывание ошибки является наиболее очевидным решением этой проблемы. В Haskell выбрасывание ошибки легко осуществляется: вам нужно воспользоваться функцией `error`. Вот функция `myHead` с выбрасыванием ошибки.

Листинг 38.3 Пример выбрасывания ошибки, myHead

```
myHead :: [a] -> a
myHead [] = error "empty list"
myHead (x:_) = x
```

Вызывает ошибку, если функция встречает пустой список

Выбрасывание ошибок считается в Haskell плохой практикой. Это связано с тем, что, как вы увидели, рассматривая `myTake`, облегчает появление в коде ошибок, которые компилятор не может проконтролировать. На практике вам лучше *никогда* не использовать `head`, а пользоваться сопоставлением с образцом. Если вы замените все вхождения `head` и `tail` на сопоставление с образцом, то компилятор сможет предупредить вас о наличии ошибок.

Главный вопрос: как поступать с частичными функциями в общем? В идеале вам хотелось бы иметь способ преобразовывать все программы так, чтобы они работали со всеми значениями. Другой известной частичной функцией в Haskell является функция `(/)`, которая не определена для 0. Но в данном случае Haskell избегает выбрасывания ошибки, предоставляя другое решение:

```
GHCi> 2 / 0
Infinity
```

Это прекрасное решение для проблемы деления на ноль, но подобные решения существуют только для некоторых случаев. Вы бы хотели использовать типы для вылавливания встречаемых ошибок. Компилятор может помочь вам в написании более устойчивого к ошибкам кода.

Проверка 38.2. Вот несколько частичных функций из `Prelude`:

- `maximum`
- `succ`
- `sum`

Как вы думаете, для каких входных данных они не определены?

38.2. Обработка частичных функций с помощью Maybe

Оказывается, вы уже успели столкнуться с одним из самых удобных способов обработки частичных функций — типом `Maybe`. В большинстве примеров, в которых вы встречались с `Maybe`, в других языках вы бы воспользовались специальным `null`-значением. Но `Maybe` является достаточно разумным способом превращения любой частичной функции во всюду определённую. Вот как вы бы могли реализовать `maybeHead`.

Листинг 38.4 Преобразование head с помощью Maybe

```
maybeHead :: [a] -> Maybe a
maybeHead [] = Nothing
maybeHead (x:_) = Just x
```

Функция `maybeHead` безопасно извлекает первый элемент списка:

```
GHCi> maybeHead [1]
Just 1
GHCi> maybeHead []
Nothing
```

В модуле 5 вы узнали, что `Maybe` является представителем класса типов `Monad` (как следствие `Functor` и `Applicative`), который позволяет совершать вычисления на значениях в контексте `Maybe`. Вспомните, что `Functor`

Ответ 38.2

- `maximum` — отказывает на пустых списках;
- `succ` — не работает на значении `maxBound` типа аргумента;
- `sum` — зацикливается на бесконечных списках.

позволяет с помощью <$> применять функцию к значению типа Maybe. Вот пример использования функции maybeHead и операции <$> для манипуляции значениями, полученными с помощью этой функции:

```
GHCi> (+2) <$> maybeHead [1]
Just 3
GHCi> (+2) <$> maybeHead []
Nothing
```

Класс типов Applicative предоставляет операцию <*>, которая позволяет связывать вместе функции в контексте и которая чаще всего используется для функций нескольких аргументов. Вот пример конкатенации результата функции maybeHead со значением Just [] с помощью <$> и <*>:

```
GHCi> (:) <$> maybeHead [1,2,3] <*> Just []
Just [1]
GHCi> (:) <$> maybeHead [] <*> Just []
Nothing
```

Вы можете скомбинировать maybeHead, <*> и <$> для создания новой безопасной версии myTake.

Листинг 38.5 Более безопасная версия myTake

```
myTakeSafer :: Int -> Maybe [a] -> Maybe [a]
myTakeSafer 0 _ = Just []
myTakeSafer n (Just xs) =
    (:) <$> maybeHead xs
        <*> myTakeSafer (n-1) (Just (tail xs))
```

В GHCi вы можете убедиться, что функция myTakeSafer прекрасно работает с входными данными, которые ранее вызывали ошибку:

```
GHCi> myTakeSafer 3 (Just [1,2,3])
Just [1,2,3]
GHCi> myTakeSafer 6 (Just [1,2,3])
Nothing
```

Как вы видите, myTakeSafer работает так, как вы рассчитывали (хотя и определена не так, как функция take, которая бы вернула весь список). Обратите внимание на причину, по которой ваша функция была названа safer, а не safe. На самом деле *tail также является частичной функцией*.

38.3. Первая встреча с Either

Мы успели потратить достаточно много времени на обсуждение выразительности типа `Maybe`, но у него есть одно существенное ограничение. Когда вы разрабатываете более сложные программы, `Nothing` становится труднее интерпретировать. Вспомните, как в итоговом проекте модуля 6 вы работали с функцией `isPrime`. Вот её упрощённая версия:

```
primes :: [Int]
primes = [2,3,5,7]

maxN :: Int
maxN = 10

isPrime :: Int -> Maybe Bool
isPrime n
    | n < 2 = Nothing
    | n > maxN = Nothing
    | otherwise = Just (n ` elem ` primes)
```

Список простых, используемый для выявления простоты числа

Наибольшее число, которое можно проверить на простоту

Если число меньше 2, то оно не проверяется

Если число больше выбранного предела, вы не можете определить, является ли оно простым

Если число подходит для проверки, то выполняется тест на простоту

Вы определили тип этой функции как `Int -> Maybe Bool`, так как вам требовалось обработать граничные случаи. Ключевым моментом было то, что вы хотели возвращать `False` для составных чисел, но столкнулись с двумя проблемами. Такие числа, как 0 и 1, не являются ни составными, ни простыми. Кроме того, `isPrime` ограничивает входные числа сверху, а вы бы не хотели возвращать в таком случае `False` только потому, что для полученного числа трудно вычислить результат.

Теперь представьте, что вы используете `isPrime` в своём собственном проекте. Когда вы вызовете `isPrime 9997`, то получите в ответ `Nothing`. Но что это значит? Вам бы пришлось заглянуть в документацию (в надежде, что она есть) для поиска ответа. Самой полезной вещью в ошибках являются сообщения, содержащиеся в них. Хотя `Maybe` и делает код более безопасным, если у `Nothing` нет очевидной интерпретации, как в случае с `null`, это значение не особенно полезно. К счастью, в Haskell есть другой тип, напоминающий `Maybe`, который позволит вам создавать более информативные ошибки, не жертвуя безопасностью.

Проверка 38.3. Предположим, у вас есть список:

```
oddList :: [Maybe Int]
oddList = [Nothing]
```

Если бы вы писали функцию, вычисляющую наибольшее среди имеющихся в такого типа списке чисел, обёрнутых в `Just`, какой тип результата вы бы для неё выбрали?

Тип, который вы подыскиваете, называется `Either`. Он лишь незначительно сложнее `Maybe`, но его определение способно вас запутать:

```
data Either a b = Left a | Right b
```

У этого типа есть два конструктора данных с непонятными названиями `Left` и `Right`. При использовании этого типа для обработки ошибок вы можете рассматривать конструктор `Left` как конструктор для ошибок, а `Right` — как конструктор для успешно вычисленного результата. Более понятным, хотя и менее общим является следующее определение:

```
data Either a b = Fail a | Correct b
```

На практике `Right` работает для `Either` абсолютно как `Just` для `Maybe`. Ключевым отличием между этими типами является то, что `Left` позволяет вам хранить больше информации, чем `Nothing`. Также обратите внимание, что `Either` принимает два типовых параметра. Это позволит вам иметь тип для отправки сообщений об ошибке и отдельный тип для обычных данных. Для демонстрации приведён пример более безопасной версии функции `head`, написанный с помощью `Either`.

Листинг 38.6 Более безопасная версия head, использующая Either

```
eitherHead :: [a] -> Either String a
eitherHead [] = Left "У списка нет первого элемента, он пуст"
eitherHead (x:xs) = Right x
```

Заметьте, что конструктор `Left` принимает `String`, а `Right` — значение первого элемента списка. Объявим несколько списков для тестов:

```
intExample :: [Int]
intExample = [1,2,3]
```

Ответ 38.3. В качестве типа результата можно взять `Maybe Int`.

```
intExampleEmpty :: [Int]
intExampleEmpty = []

charExample :: [Char]
charExample = "кот"

charExampleEmpty :: [Char]
charExampleEmpty = ""
```

В GHCi вы можете посмотреть, как `Either` работает, а также разобраться с типами возвращаемых значений:

```
GHCi> eitherHead intExample
Right 1
GHCi> eitherHead intExampleEmpty
Left "У списка нет первого элемента, он пуст"
GHCi> eitherHead charExample
Right 'к'
GHCi> eitherHead charExampleEmpty
Left "У списка нет первого элемента, он пуст"
```

Тип `Either` также реализует экземпляр класса типов `Monad` (следовательно, `Functor` и `Applicative`). Вот простенький пример использования `<$>` для увеличения первого элемента `intExample`:

```
GHCi> (+1) <$> (eitherHead intExample)
Right 2
GHCi> (+1) <$> (eitherHead intExampleEmpty)
Left "У списка нет первого элемента, он пуст"
```

Тип `Either` совмещает безопасность `Maybe` с дополнительной понятностью, которую дают сообщения об ошибках.

Проверка 38.4. Воспользуйтесь операциями `<*>` и `<$>` для суммирования первого и второго элементов списка `intExample`, полученных с помощью функций `eitherHead` и `tail`.

Ответ 38.4

```
(+) <$> eitherHead intExample <*> eitherHead (tail intExample)
```

38.3.1. Проверка на простоту с использованием Either

Для демонстрации работы `Either` давайте посмотрим, как создать простое консольное приложение для проверки чисел на простоту. Вы составите максимально короткое определение `isPrime`, фокусируясь на работе с типом `Either`. Начнём с использования `String` для сообщений об ошибках. Затем вы сможете воспользоваться преимуществом того, что `Either` позволяет вам использовать любой желаемый тип, предоставляя возможность создавать собственные типы для ошибок.

Очень удобным свойством `Either` является то, что вы не привязаны к одному-единственному сообщению об ошибке. У вас их может быть насколько много, насколько пожелаете. Улучшенная версия функции `isPrime` позволит вам определять, является ли число неподходящим для проверки на простоту или же оно слишком велико.

Листинг 38.7 Функция isPrime с сообщениями об ошибках

```
isPrime :: Int -> Either String Bool
isPrime n
    | n < 2 = Left "Числа меньше 2 не проверяются на простоту"
    | n > maxN = Left "Число слишком велико для проверки"
    | otherwise = Right (n `elem` primes)
```

Вот несколько тестовых прогонов этой функции в GHCi:

```
GHCi> isPrime 5
Right True
GHCi> isPrime 6
Right False
GHCi> isPrime 100
Left "Число слишком велико для проверки"
GHCi> isPrime (-29)
Left "Числа меньше 2 не проверяются на простоту"
```

До сих пор вы не использовали возможность `Either` принимать два типа; вы пока что использовали исключительно `String` для конструктора `Left`. В большинстве языков программирования вы можете представлять ошибки с помощью классов. Это позволяет с лёгкостью моделировать различные типы ошибок. `Either` позволит вам сделать то же самое. Давайте определим тип для ошибок.

Листинг 38.8 Тип PrimeError для представления ошибок

```
data PrimeError = TooLarge | InvalidValue
```

Теперь можно сделать этот тип экземпляром Show, чтобы было легко выводить текст ошибки.

Листинг 38.9 Экземпляр Show для PrimeError

```
instance Show PrimeError where
  show TooLarge = "Число слишком велико"
  show InvalidValue = "Число не подходит для проверки"
```

С этим новым типом PrimeError вы можете переработать функцию isPrime для вывода информации об ошибках.

Листинг 38.10 Функция isPrime с использованием PrimeError

```
isPrime :: Int -> Either PrimeError Bool
isPrime n
    | n < 2 = Left InvalidValue
    | n > maxN = Left TooLarge
    | otherwise = Right (n ‘ elem ‘ primes)
```

Это делает код гораздо более читаемым. Кроме того, у вас теперь есть тип, который с лёгкостью будет работать с ошибками в самых различных ситуациях. Вот примеры использования вашей новой функции в GHCi:

```
GHCi> isPrime 5
Right True
GHCi> isPrime 6
Right False
GHCi> isPrime 99
Left Число слишком велико
GHCi> isPrime 0
Left Число не подходит для проверки
```

Далее вам нужно создать функцию displayResult, которая преобразует ответ типа Either в String.

Листинг 38.11 Перевод результата isPrime в читаемый формат

```
displayResult :: Either PrimeError Bool -> String
displayResult (Right True) = "Это простое число"
displayResult (Right False) = "Это составное число"
displayResult (Left primeError) = show primeError
```

Наконец, вы можете собрать всё это вместе в простое IO-действие, которое выглядит следующим образом.

Листинг 38.12 Проверка на простоту вводимых пользователем чисел

```
main :: IO ()
main = do
    putStrLn "Введите число для проверки на простоту:"
    n <- read <$> getLine
    let result = isPrime n
    putStrLn (displayResult result)
```

Теперь вы можете собрать и запустить свою программу:

```
$ stack build
$ stack exec primechecker-exe
Введите число для проверки на простоту:
6
Это составное число

$ stack exec headaches-exe
Введите число для проверки на простоту:
5
Это простое число

$ stack exec headaches-exe
Введите число для проверки на простоту:
213
Число слишком велико

$ stack exec headaches-exe
Введите число для проверки на простоту:
0
Число не подходит для проверки
```

Тип `PrimeError` позволяет эмулировать более сложные способы представления ошибок из ООП-языков. Великолепным свойством `Either` является то, что конструктор `Left` может быть любого типа, ничто не ограничивает вашу фантазию. Если захотите, вы даже можете вернуть функцию!

Итоги

В этом уроке нашей целью было научить вас безопасно обрабатывать ошибки в Haskell. Вы начали с изучения того, как `head` использует `error` для сигнала о том, что функции был передан пустой список, у которого

нет первого элемента. Ни контроль типов, ни предупреждения GHC не сигнализировали о проблеме. На самом деле это происходит из-за того, что `head` — частичная функция, которая возвращает результат не для всех возможных входных данных. Это может быть решено с помощью типа `Maybe`. Хотя `Maybe`-типы делают код более безопасным, они могут усложнить определение причины ошибки. Наконец, вы увидели, что тип `Either` предоставляет возможности предыдущих двух способов представления ошибок, позволяя вам безопасно обрабатывать ошибки и передавать детальную информацию о них.

Задача 38.1. Напишите функцию `addStrInts`, которая принимает два целых числа, представленных в виде значений типа `String`, и складывает их. Эта функция вернёт `Either String Int`. Конструктор `Right` должен возвращать результат, если оба аргумента могут быть успешно преобразованы в значения типа `Int` (используйте Data.Char.isDigit для проверки). Функция должна возвращать три разных значения с помощью конструктора `Left` для следующих случаев:

- первое значение не удалось считать;
- второе значение не удалось считать;
- ни одно из значений не удалось считать.

Задача 38.2. Далее перечислены частичные функции. Используйте указанные типы для определения более безопасных версий этих функций:

- `succ` — `Maybe`;
- `tail` — `[a]` (сохраните первоначальный тип);
- `last` — `Either` (функция `last` терпит неудачу на пустых и бесконечных списках, используйте верхний предел для обработки бесконечных списков).

39

Создание HTTP-запросов в Haskell

После прочтения урока 39 вы:

- научитесь загружать веб-страницы с помощью Haskell;
- сможете генерировать более сложные запросы путём установки заголовков и использования HTTPS;
- поймёте, как начинать изучение новых типов или библиотек Haskell.

В этом уроке вы изучите, как составлять HTTP-запросы в Haskell и сохранять результаты в файл. Вы будете получать данные от климатического API Национального управления океанических и атмосферных явлений (National Oceanic and Atmospheric Administration, NOAA). Этот API требует от вас отправки специальных HTTP-запросов, которые используют SSL и включают в себя особый заголовок для аутентификации. Вы будете пользоваться библиотекой `Network.HTTP.Simple`, которая позволит вам как делать простые запросы, так и формировать более сложные. Вы начнёте с изучения того, как можно использовать `Network.HTTP.Simple` для получения веб-страницы по её адресу. Затем вы создадите запрос специального вида для API NOAA. В итоге с помощью этого API вы получите данные в формате JSON, с которыми мы будем работать в следующем уроке.

Обратите внимание. Как бы вы написали на Haskell программу, которая при запуске получает домашнюю страницу `reddit.com` и сохраняет её в виде html-файла?

39.1. Первоначальная настройка проекта

В этом уроке мы рассмотрим одну из самых распространённых в современном программировании задач — создание HTTP-запросов. Целью этого проекта является написание сценария, который будет запрашивать климатические данные от NOAA API, предоставляющего доступ к очень широкому кругу климатических данных. На сайте с описанием этого API (`https://www.ncdc.noaa.gov/cdo-web/webservices/v2#gettingStarted`) вы можете найти список всех оконечных точек, предлагаемых API. Вот несколько из них:

- /datasets — сообщает о том, какие наборы данных доступны;
- /locations — показывает, какие местности доступны для просмотра;
- /stations — предоставляет информацию о доступных метеостанциях;
- /data — предоставляет доступ к сырым данным.

Написание полноценной обёртки поверх NOAA API было бы для одного урока слишком серьёзной задачей. Пока вы сфокусируетесь на первом шаге этой задачи — на получении результатов от оконечной точки /datasets, предоставляющей важные метаданные, необходимые для запроса данных от /data. Вот пример записи, которую можно от неё получить:

```
"uid":"gov.noaa.ncdc:C00822",
"mindate":"2010-01-01",
"maxdate":"2010-12-01",
"name":"Normals Monthly",
"datacoverage":1,
"id":"NORMAL_MLY"
```

Даже несмотря на то, что получение данных составляет лишь небольшую часть API, после того как вы разберётесь с основами работы с HTTP в Haskell, расширение проекта будет достаточно тривиальной задачей. После отправки запроса вы запишете тело ответа в JSON-файл. Хотя, честно говоря, это очень простая задача, так вы сможете узнать много нового о реальной разработке на Haskell.

Вам нужно будет создать проект `http-lesson`. В качестве напоминания тут приведена пошаговая инструкция по созданию и сборке проекта:

```
$ stack update
$ stack new http-lesson
$ cd http-lesson
$ stack setup
$ stack build
```

Всё необходимое для этого проекта нужно будет записывать в модуль `Main`, хранящийся в app/Main.hs.

> **Примечание.** В этом проекте используется NOAA Climate Data API для получения JSON-данных и сохранения их в виде файла. В следующем уроке вы займётесь чтением этих данных. Этот API открыт для свободного пользования, но для обращения к нему необходимо запросить специальный API-токен. Чтобы получить этот токен, перейдите на `www.ncdc.noaa.gov/cdo-web/token` и внесите в форму свой адрес электронной почты. Ваш токен должен оперативно прийти. Вы будете осуществлять запросы, чтобы узнать, какие наборы данных предоставляет API.

После получения API-токена вы можете начать разработку проекта.

39.1.1. Первые штрихи

Начнём с добавления требуемых инструкций импорта в модуль `Main`. Заметьте, что здесь нужно будет сразу импортировать и `Data.ByteString`, и `Data.ByteString.Lazy`. Вообще, импорт нескольких текстовых типов или нескольких типов `ByteString` встречается при разработке на Haskell достаточно часто. В данном случае это потребовалось, так как некоторые части библиотеки, которую вы собираетесь использовать, требуют применения ленивой версии `ByteString`, тогда как другие — строгой. Также вы произведёте импорт модуля `Char8` для обеих этих библиотек, поскольку это облегчит работу с ними, как уже обсуждалось в уроке 25. Наконец, вам нужно будет добавить библиотеку `Network.HTTP.Simple`, которую вам предстоит использовать для HTTP-запросов.

Листинг 39.1 Объявления импорта в файле app/Main.hs

```
module Main where

import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as BC
import qualified Data.ByteString.Lazy as L
import qualified Data.ByteString.Lazy.Char8 as LC
import Network.HTTP.Simple
```

Прежде чем продолжить, вам нужно будет обновить файл с настройками проекта http-lesson.cabal и зафиксировать соответствующие библиотеки. Добавьте `bytestring` и `http-conduit` к разделу `build-depends`. Так как вы будете работать с `ByteString` и `Char8`, полезно также будет включить расширение `OverloadedStrings`.

Листинг 39.2 Изменение cabal-файла проекта

```
executable http-lesson-exe
    hs-source-dirs:      app
    main-is:             Main.hs
    ghc-options:         -threaded -rtsopts -with-rtsopts=-N
    build-depends:       base
                       , http-lesson
                       , bytestring
                       , http-conduit
    default-language:    Haskell2010
    extensions:          OverloadedStrings
```

Это необходимо для импорта разнообразных модулей `Data.ByteString`

`http-conduit` — библиотека, включающая модуль `Network.HTTP.Simple`

`OverloadedStrings` облегчает работу с типами `ByteString`

По счастью, stack самостоятельно разберётся с загрузкой всех зависимостей `http-conduit`, так что вам не нужно явно устанавливать пакеты с помощью `stack install`.

Далее приступим к внесению в `Main` необходимых для работы значений. Вы будете полностью сосредоточены только на одном API-запросе, который позволит вам получить список всех наборов данных, предоставляемых NOAA Climate Data API.

Листинг 39.3 Полезные при создании HTTP-запросов значения

```
myToken :: BC.ByteString
myToken = "<ваш API-токен>"

noaaHost :: BC.ByteString
noaaHost = "www.ncdc.noaa.gov"

apiPath :: BC.ByteString
apiPath = "/cdo-web/api/v2/datasets"
```

Также нужно будет добавить заглушку в `main IO`, чтобы программа могла скомпилироваться.

Листинг 39.4 Заглушка для main

```
main :: IO ()
main = putStrLn "привет"
```

Проверка 39.1. Если бы вы не стали указывать в cabal-файле расширение `OverloadedStrings`, то какие бы изменения пришлось внести в Main.hs для поддержки этого расширения?

39.2. Использование модуля HTTP.Simple

Теперь, когда заложен фундамент, можно начать разбираться с HTTP-запросами. Вы будете использовать модуль `Network.HTTP.Simple`, который включён в пакет `http-conduit`. Как показывает имя, `HTTP.Simple` облегчает отправку простых HTTP-запросов. Для отправки запроса вы будете использовать функцию `httpLBS` (LBS расшифровывается как *ленивая ByteString*). Обычно приходится создавать значение типа данных `Request` для передачи соответствующих данных в функцию. Но функция `httpLBS` способна воспользоваться преимуществами `OverloadedStrings`, для того чтобы удостовериться, что передано корректное значение. Вот небольшой пример запроса данных с популярного новостного сайта о технологиях Hacker News (`https://news.ycombinator.com`):

```
GHCi> import Network.HTTP.Simple
GHCi> response = httpLBS "http://news.ycombinator.com"
```

Если вы наберёте это в GHCi, то увидите, что значение `response` становится доступным мгновенно, хотя вы вроде бы делаете HTTP-запрос. Обычно при отправке HTTP-запроса появляется ощутимая задержка, связанная с природой самого запроса. Значение же становится доступным сразу благодаря ленивости вычислений. Вы определили запрос, но ещё не отправили его. Если вы снова введёте `response`, то заметите задержку:

```
GHCi> response
<длинный ответ>
```

Вам бы хотелось иметь возможность получать доступ к разным частям ответа. Первым делом стоит проверить код состояния ответа. Это специальный HTTP-код, который показывает, был ли запрос успешен.

Ответ 39.1. Вы могли бы воспользоваться директивой `LANGUAGE`:

```
{-# LANGUAGE OverloadedStrings #-}
```

Часто встречающиеся HTTP-коды

Если вы незнакомы с кодами состояния HTTP, вот самые часто встречающиеся:

- 200 OK — запрос был успешен;
- 301 Moved Permanently — запрашиваемый ресурс перемещён;
- 404 Not Found — ресурс не был найден.

В модуле `Network.HTTP.Simple` есть функция `getResponseStatusCode`, которая возвращает код состояния ответа. Если вы запустите эту функцию в GHCi, то сразу столкнётесь с проблемой:

```
GHCi> getResponseStatusCode response
<interactive>:6:23: error:
    No instance for (Control.Monad.IO.Class.MonadIO Response)
        arising from a use of 'response'
```

Проблема в том, что функция `getResponseStatusCode` ожидает аргумента вне контекста, как можно увидеть, посмотрев на её тип:

```
getResponseStatusCode :: Response a -> Int
```

Но, совершая HTTP-запрос, вы воспользовались возможностями `IO`, а значит, у значения ответа тип `IO (Response a)`.

Популярная альтернатива HTTP.Simple

Хотя библиотека `Network.HTTP.Simple` достаточно легка для понимания, она не особенно выразительная. В Haskell есть множество других пакетов, способных совершать HTTP-запросы. Одним из наиболее популярных является `wreq` (`https://hackage.haskell.org/package/wreq`). Хотя `wreq` — очень неплохая библиотека, для работы с ней вам бы потребовалось освоить другую абстракцию, линзы (lens). Стоит упомянуть, что очень часто случается так, что в пакетах, написанных на Haskell, используются новые и интересные абстракции. Если вам понравился модуль 5, в котором рассказывалось о монадах, то вам определённо понравится этот аспект работы с Haskell. Но использование большого количества абстракций также может стать разочарованием для начинающих, которые, возможно, не хотят погружаться в изучение чего-то нового в процессе простого получения данных с помощью API.

Эту проблему можно решить двумя способами. Во-первых, вы можете воспользоваться операцией <$> из класса типов `Functor`:

```
GHCi> getResponseStatusCode <$> response
200
```

Напомним, что <$> позволяет вам помещать чистые функции в контекст. Если вы посмотрите на тип результата, то увидите, что он тоже находится в контексте:

```
GHCi> :t getResponseStatusCode <$> response
getResponseStatusCode <$> response
:: Control.Monad.IO.Class.MonadIO f => f Int
```

Альтернативным решением может стать присваивание `response` с помощью <- вместо =. Точно так же, как и при использовании do-нотации, это позволяет вам обращаться со значениями в контексте так, словно они являются обычными чистыми значениями:

```
GHCi> response <- httpLBS "http://news.ycombinator.com"
GHCi> getResponseStatusCode response
200
```

Теперь, когда вы понимаете основы отправки HTTP-запросов, вы готовы двигаться к более сложным примерам.

Проверка 39.2. В библиотеке есть функция `getResponseHeader`. Используя её вместе с <$> и <-, получите заголовок ответа.

Ответ 39.2 Первый метод:

```
GHCi> import Network.HTTP.Simple
GHCi> response = httpLBS "http://news.ycombinator.com"
GHCi> getResponseHeader <$> response
```

Второй метод:

```
GHCi> response <- httpLBS "http://news.ycombinator.com"
GHCi> getResponseHeader response
```

39.3. Создание HTTP-запроса

Хотя такой простой способ использования функции `httpLBS` очень удобен, вам нужно будет изменить несколько вещей. Ваш запрос к API должен использовать HTTPS, а не HTTP, он также должен содержать в заголовке токен аутентификации. Поэтому вы не можете просто указать адрес запроса, придётся дополнительно сделать следующее:

- добавить токен в заголовок;
- указать в запросе имя сервера и путь к ресурсу;
- удостовериться, что используется метод запроса `GET`;
- убедиться, что запрос работает с использованием SSL-соединения.

Вы можете сделать это с помощью вызова последовательности функций, которые установят все необходимые параметры запроса. Код для создания запроса приведён чуть ниже. Вообще-то, создание запроса — достаточно простой процесс, но вам придётся столкнуться с операцией, с которой вы ещё не успели познакомиться в этой книге, а именно с операцией `$`. Операция `$` автоматически оборачивает код после себя в скобки (для ознакомления с деталями изучите пояснение на следующей странице).

Листинг 39.5 Код для создания HTTPS-запроса к API

```
buildRequest :: BC.ByteString -> BC.ByteString
             -> BC.ByteString -> Request
buildRequest token host method path =
     setRequestMethod method
   $ setRequestHost host
   $ setRequestHeader "token" [token]
   $ setRequestPath path
   $ setRequestSecure True
   $ setRequestPort 443
   $ defaultRequest

request :: Request
request = buildRequest myToken noaaHost "GET" apiPath
```

Операция $

Операция $ чаще всего используется для автоматической расстановки скобок. Вы можете представить себе это следующим образом: открывающая скобка располагается вместо $, а закрывающая — в конце определения функции (в том числе если функция заканчивается через несколько строк). Например, предположим, что вы хотите удвоить выражение 2 + 2. Вам требуется добавить скобки, чтобы выражение было корректно:

```
GHCi> (*2) 2 + 2
6
GHCi> (*2) (2 + 2)
8
```

Вместо этого вы могли бы написать:

```
GHCi> (*2) $ 2 + 2
8
```

Вот другой пример:

```
GHCi> head (map (++"!") ["dog","cat"])
"dog!"
GHCi> head $ map (++"!") ["dog","cat"]
"dog!"
```

Начинающим эта операция часто усложняет чтение кода. На практике операция $ используется достаточно часто, и, скорее всего, вы будете предпочитать использовать её, а не множество скобок. В работе $ нет ничего магического; если вы посмотрите на тип этой операции, то увидите, как она работает:

```
($) :: (a -> b) -> a -> b
```

Первым аргументом операции является функция, а вторым — значение. Штука в том, что $ — бинарная операция, поэтому у неё более низкий приоритет, чем у других функций. Именно поэтому аргумент функции вычисляется так, будто он заключён в скобки.

Интересным моментом в этом коде является то, как обрабатывается состояние запроса. У вас есть набор функций `set`, но как они устанавлива-

ют значения? Вы лучше разберётесь в том, что происходит, если потратите некоторое время на изучение типов функций из этого набора:

```
GHCi> :t setRequestMethod
setRequestMethod :: BC.ByteString -> Request -> Request

GHCi> :t setRequestHeader
setRequestHeader:: HeaderName -> [BC.ByteString] -> Request
                                                  ↳  -> Request
```

Здесь вы можете увидеть один из функциональных подходов к обработке состояния. Каждая функция из набора `set` принимает в качестве аргумента параметр, который нужно поменять, и существующие данные запроса. Вы начинаете с первоначального запроса, `defaultRequest`, который предоставлен модулем `Network.HTTP.Simple`. Затем вы создаёте новую копию данных запроса с изменённым параметром, возвращая модифицированный запрос в качестве результата. Вы уже видели похожий подход в первом модуле, только обсуждался там он подробнее. Вы могли бы переписать эту функцию, явно контролируя состояние с помощью `let`-выражения. Заметьте, что вызовы функций расположены в обратном порядке.

Листинг 39.6 Функция buildRequest с явным состоянием

```
buildRequest token host method path =
  let state1 = setRequestPort 443 defaultRequest
  in let state2 = setRequestSecure True state1
     in let state3 = setRequestPath path state2
        in let state4 = setRequestHeader "token" [token] state3
           in setRequestHost host state4
```

Использование операции `$` заставляет каждую функцию из набора `set*` служить в качестве аргумента для следующей функции, сильно сокращая код. Программисты на Haskell очень любят писать краткий код, когда это возможно, хотя временами это усложняет чтение кода при первом знакомстве с ним.

39.4. Собираем всё вместе

Теперь всё готово для соединения в IO-действии `main`. Вы можете передать запрос в `httpLBS`. Затем, используя полученный статус, вам нужно проверить, получен ли код 200. Если это так, то записываете данные в файл, используя функцию `getResponseBody`. В противном случае вам

нужно сообщить пользователю, что в запросе была ошибка. При записи данных в файл помните, что очень важно использовать сырые ленивые ByteString-строки с L.writeFile, а не с LC.writeFile из модуля Char8. В уроке 25 мы уже упоминали, что если используемые двоичные данные могут содержать символы Unicode, то ни в коем случае нельзя использовать интерфейс Char8, так как это может испортить ваши данные.

Листинг 39.7 Финальная версия main с записью запроса в JSON-файл

```
main :: IO ()
main = do
    response <- httpLBS request
    let status = getResponseStatusCode response
    if status == 200
        then do
           putStrLn "Результаты запроса были сохранены в файл"
           let jsonBody = getResponseBody response
           L.writeFile "data.json" jsonBody
        else
           putStrLn "Запрос не удалось совершить из-за ошибки"
```

Теперь у вас есть простое приложение для получения данных с помощью REST API и записи их в файл. Это была всего лишь небольшая демонстрация отправки HTTP-запросов с помощью Haskell. Полную документацию к данной библиотеке можно найти по следующему адресу: https://haskell-lang.org/library/http-client.

Итоги

В этом уроке нашей целью было предоставить вам краткий обзор того, как можно совершать HTTP-запросы с помощью Haskell. В дополнение к навыкам составления HTTP-запросов вы посмотрели, с чего нужно начинать изучение новых библиотек Haskell. Давайте проверим, как вы усвоили материалы урока.

Задача 39.1. Реализуйте функцию buildRequestNOSSL, которая работает аналогично buildRequest, но не поддерживает SSL.

Задача 39.2. Улучшите вывод программы при возникновении ошибок. Функция getResponseStatus возвращает значение, содержащее два компонента: statusCode и statusMessage. Исправьте main так, чтобы при получении кода, отличного от 200, выводилось подходящее сообщение об ошибке.

40

Работа с данными JSON с использованием Aeson

После прочтения урока 40 вы:

- научитесь преобразовывать типы Haskell в JSON;
- сможете преобразовывать JSON в типы Haskell;
- разберётесь с использованием расширения `DeriveGeneric` для порождения реализации требуемых классов;
- сможете писать собственные экземпляры `ToJSON` и `FromJSON`.

В этом уроке вы будете работать с данными, записанными с помощью объектной нотации JavaScript — JavaScript Object Notation (JSON), одним из наиболее популярных способов хранения и передачи данных. Формат JSON произошёл от простых объектов JavaScript и очень широко используется для передачи данных с помощью HTTP API. Так как этот формат очень простой, он распространился за пределы сети и часто используется в качестве способа хранения данных и для таких задач, как, например, создание конфигурационных файлов. На рис. 40.1 показан пример объекта JSON, используемого Google Analytics API.

Предыдущий урок вы завершили загрузкой JSON-файла, содержащего информацию о наборах данных, доступных благодаря NOAA Climate API. В этом уроке вы будете разрабатывать простое приложение командной строки, которое сможет открыть этот JSON-файл и извлечь из него источники данных. Но прежде вы изучите, как работать с JSON. Также вы создадите типы, которые можно превращать в JSON, и типы, с которыми можно провести обратное преобразование.

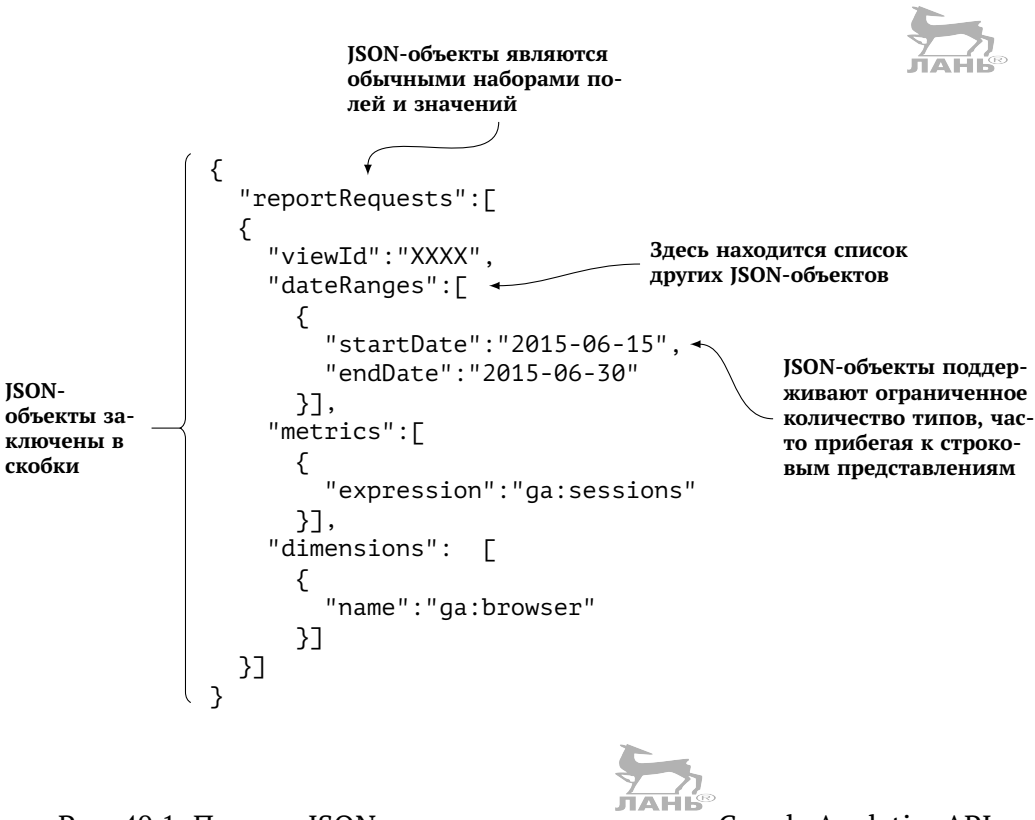


Рис. 40.1: Пример JSON-данных, полученных от Google Analytics API

Обратите внимание. У вас есть тип данных, представляющий пользователя:

```
data User =
     User
     { userId :: Int
     , userName :: T.Text
     , email :: T.Text
     }
```

Процесс трансформации данных в JSON и обратно известен как сериализация и десериализация соответственно. Вы могли уже сталкиваться с этими понятиями в других языках программирования. Если у вас есть тип, представляющий пользователя, то как бы вы провели его сериализацию и десериализацию?

40.1. Первоначальная настройка

Основной трудностью при работе с JSON в Haskell является то, что этот формат поддерживает только несколько базовых типов: объекты, строки, числа (технически обычные числа с плавающей запятой), списки и логический тип. В большинстве языков программирования JSON поддерживается с помощью структуры данных, напоминающей словарь. Вы будете использовать библиотеку Aeson, которая предоставляет более подходящее для Haskell решение. Aeson позволяет вам совершать преобразования между мощной системой типов Haskell и JSON.

Aeson для выполнения этих преобразований полагается на две ключевые функции: `encode` и `decode`. Для использования этих двух функций вам нужно будет сделать ваши типы данных экземплярами двух классов типов: `ToJSON` (`encode`) и `FromJSON` (`decode`). Мы продемонстрируем два способа это сделать. Первый заключается в автоматическом выводе классов типов с помощью одного из расширений языка, а другой — в реализации этих экземпляров вручную.

После того как вы разберётесь с основами Aeson, вы сможете создать тип данных, представляющий JSON-данные, полученные от NOAA. JSON-ответ от NOAA Climate Data API включает в себя вложенные объекты, так что вам придётся реализовать нетривиальный тип для взаимодействия с этими данными. Наконец, вы соедините всё вместе, что позволит вам вывести всё содержимое файла.

40.1.1. Подготовка stack-проекта

Для этого урока вы будете использовать stack-проект `json-lesson`. Как и предыдущий раз, для удобства вы будете хранить весь код в модуле `Main`. Первым делом вам нужно будет подготовить файл Main.hs. Начнёте вы с импорта основных библиотек. Вы будете работать с популярной библиотекой для работы с JSON, Aeson (названной так в честь Эсона, который был отцом героя древнегреческих мифов, Ясона). В этом уроке все текстовые данные, с которыми вам придётся работать, будут в виде `Data.Text`, так как это предпочтительный для Haskell метод хранения текстовых данных. Для этого вам также нужно будет импортировать ленивую версию `ByteString` и вспомогательный модуль `Char8`. Ваши JSON-данные по умолчанию будут представлены в виде `ByteString`, пока вы не преобразуете их в более выразительные типы. Вот начальная версия файла Main.hs, которая включает все требуемые для этого урока объявления импорта.

Листинг 40.1 Файл Main.hs

```
module Main where

import Data.Aeson
import Data.Text as T
import Data.ByteString.Lazy as B
import Data.ByteString.Lazy.Char8 as BC
import GHC.Generics

main :: IO ()
main = print "привет"
```

Вам потребуется добавить эти библиотеки в файл json-lesson.cabal. Стоит также указать там расширение `OverloadedStrings`. В этом уроке вы столкнётесь с ещё одним расширением.

Листинг 40.2 Добавление зависимостей и включение расширений

```
build-depends:       base
                   , json-lesson
                   , aeson
                   , bytestring
                   , text
default-language:    Haskell2010
extensions:          OverloadedStrings
                   , DeriveGeneric
```

Теперь можно начать изучение моделирования JSON-данных в Haskell.

40.2. Использование библиотеки Aeson

Для работы с JSON вы будете использовать самую популярную в Haskell библиотеку для JSON — Aeson. Основной трудностью, с который вы столкнётесь при работе с JSON в Haskell, будет то, что JSON не особенно активно использует типы, представляя большую часть данных в виде строк. Великолепным качеством Aeson является то, что эта библиотека позволяет применять сильную систему типов Haskell в отношение JSON. Благодаря этому вы получаете лёгкую работу с распространённым и гибким форматом данных без потери мощи Haskell и типовой безопасности.

Aeson для осуществления основной массы работы полагается на две простые функции. Функция `decode` принимает данные JSON и преобразует их в целевой тип. Вот типовая аннотация этой функции:

```
decode :: FromJSON a => ByteString -> Maybe a
```

Здесь стоит заострить внимание на двух вещах. Во-первых, возвращается тип Maybe. Как было замечено в уроке 38, Maybe является хорошим способом обрабатывать ошибки в Haskell. В данном случае вас будут заботить ошибки, связанные с корректным разбором JSON. Существует множество ситуаций, в которых разбор может пойти не по плану; например, JSON может быть искажён или не соответствовать ожидаемому типу. Если что-то во время разбора пойдёт не так, вы получите значение Nothing. Вы успели узнать, что Either часто оказывается более удачным типом для подобных целей, так как он может указать, что именно пошло не так. Aeson также предоставляет функцию eitherDecode, которая возвращает более информативные сообщения об ошибках, используя конструктор Left (вспомните, что этот конструктор обычно используется для ошибок):

```
eitherDecode :: FromJSON a => ByteString -> Either String a
```

Другой важной вещью, которую следует упомянуть, является то, что типовый параметр Maybe (или Either) ограничен классом типов FromJSON. То, что тип сделан экземпляром FromJSON, позволяет вам переводить сырые JSON-данные в значение Maybe этого типа. Способы, которые позволят сделать тип экземпляром FromJSON, мы рассмотрим в следующем разделе.

В Aeson есть другая важная функция, encode, которая выполняет обратную decode функцию. Вот тип encode:

```
encode :: ToJSON a => a -> ByteString
```

Функция encode принимает значения типа, являющегося экземпляром ToJSON, и возвращает JSON-объект, представленный ByteString. Класс ToJSON — двойник FromJSON. Если тип является экземпляром и FromJSON, и ToJSON, то он может быть с лёгкостью преобразован в JSON и обратно. Далее вы увидите, как делать ваши типы экземплярами этих классов.

Проверка 40.1. Почему encode возвращает ByteString, а не, к примеру, Maybe ByteString?

Ответ 40.1 Поскольку в JSON можно преобразовать любое значение. Проблемы возникают, когда не удаётся преобразовать JSON в нужный тип.

40.3. Экземпляры FromJSON и ToJSON для своих типов

Основной целью Aeson является организация простого способа совершать преобразования между типами Haskell и сырыми данными JSON. Это особенно интересная задача, учитывая, что у JSON в наличии только ограниченное количество типов: числа (технически вещественные числа с плавающей запятой), строки, массивы значений и логический тип. Для этого в Aeson используется два класса типов: `FromJSON` и `ToJSON`. Класс типов `FromJSON` позволяет вам разбирать JSON и преобразовывать данные в тип Haskell, а `ToJSON` помогает превращать типы Haskell в JSON. Aeson прекрасно справляется со своими целями, в большинстве случаев облегчая решение этих задач.

40.3.1. Простой способ

Для большинства типов данных Haskell реализация `FromJSON` и `ToJSON` достаточно тривиальна. Давайте начнём с типа `Book`, который вы сделаете экземпляром обоих классов типов. Тип `Book` будет невероятно простым, он будет содержать одно текстовое поле для названия и другое — для автора, а также поле типа `Int` для года публикации. Чуть дальше в этом уроке вы рассмотрите более сложные типы данных. Вот определение типа `Book`.

Листинг 40.3 Простой тип Book

```
data Book = Book
    { title :: T.Text
    , author :: T.Text
    , year :: Int
    } deriving Show
```

Есть простой способ сделать `Book` экземпляром обоих рассматриваемых классов типов. Для этого вам потребуется использовать ещё одно языковое расширение `DeriveGeneric`. Это расширение добавляет поддержку для более продвинутого обобщённого программирования в Haskell. Это даёт возможность писать обобщённые экземпляры определений классов типов, позволяя новым типам с лёгкостью становиться экземплярами классов типов без написания лишнего кода. Расширение `DeriveGeneric` позволит без всяких трудностей выводить экземпляры `FromJSON` и `ToJSON`. Единственное, что требуется исправить, — добавить `Generic` в инструкцию `deriving`.

Листинг 40.4 Добавление deriving Generic к типу Book

```
data Book = Book
    { title :: T.Text
    , author :: T.Text
    , year :: Int
    } deriving (Show,Generic)
```

Наконец, вам нужно объявить Book экземпляром FromJSON и ToJSON. Кроме этих двух строчек, вам не придётся ничего делать (никаких дополнительных where-выражений и определений не требуется).

Листинг 40.5 Объявление Book экземпляром ToJSON и FromJSON

```
instance FromJSON Book
instance ToJSON Book
```

Для демонстрации мощи этих классов типов давайте возьмём значение вашего типа и закодируем его.

Листинг 40.6 Преобразование Book в JSON

```
myBook :: Book
myBook = Book { author="Уилл Курт"
              , title="Программируй на Haskell"
              , year=2019}

myBookJSON :: BC.ByteString
myBookJSON = encode myBook
```

В GHCi вы можете увидеть, как это выглядит:

```
GHCi> myBook
Book {title = "Программируй на Haskell", author = "Уилл Курт",
                                                  ↳ year = 2019}
GHCi> myBookJSON
"{\"author\":\"Уилл Курт\",\"title\":
                  ↳ \"Программируй на Haskell\",\"year\":2019}"
```

С той же лёгкостью можно произвести обратные действия. Вот сырые данные JSON в виде ByteString, которые вы преобразуете в нужный тип.

Листинг 40.7 Преобразование JSON книги в значение типа Book

```
rawJSON :: BC.ByteString
rawJSON = "{\"author\":\"Эмиль Чоран\",\"title\":
```

```
                ↳ \"Трактат о разложении основ\",\"year=1949}"

bookFromJSON :: Maybe Book
bookFromJSON = decode rawJSON
```

В GHCi вы можете убедиться, что у вас получилось успешно создать значение типа Book из JSON:

```
GHCi> bookFromJSON
Just (Book { title = "Трактат о разложении основ"
           , author = "Эмиль Чоран"
           , year = 1949})
```

Это очень мощная особенность Aeson. Из JSON-строки, которая обычно не содержит практически никакой информации о типе, вы смогли успешно создать значение типа Haskell. Во многих языках разбор JSON означает, что для хранения результата придётся использовать хеш-таблицу или словарь ключей и значений. Благодаря Aeson вы можете получить из JSON нечто, гораздо более выразительное.

Заметьте, что результат обёрнут в конструктор данных Just. Это случилось из-за того, что ошибка в разборе может с лёгкостью сделать невозможным получение значения желаемого типа. Если вы работаете с испорченным объектом JSON, то ничего из него не получите.

Листинг 40.8 Объект JSON, который не соответствует нужному типу

```
wrongJSON :: BC.ByteString
wrongJSON = "{\"writer\":\"Эмиль Чоран\",\"title\":
              ↳ \"Трактат о разложении основ\",\"year\"=1949}"

bookFromWrongJSON :: Maybe Book
bookFromWrongJSON = decode wrongJSON
```

При загрузке этого кода в GHCi вы получите Nothing:

```
GHCi> bookFromWrongJSON
Nothing
```

Такой результат является великолепным примером ограниченности типа Maybe. Вы точно знаете, что пошло не так при разборе JSON, поскольку вы целенаправленно внесли в код ошибку. Но в реальном проекте это была бы невероятно неприятная ошибка, особенно если бы у вас не было прямого доступа к сырым данным JSON для проверки. В качестве альтернативы вы можете воспользоваться eitherDecode, функцией, которая предоставит гораздо больше информации:

```
GHCi> eitherDecode wrongJSON :: Either String Book
Left "Error in $: The key \"author\" was not found"
```

Теперь вы сможете узнать, почему же разбор не удался: не найдено требуемое поле `"author"`.

Хотя использование `DeriveGeneric` делает использование Aeson невероятно простым, у вас не всегда будет возможность воспользоваться таким преимуществом. Время от времени вы должны будете помогать Aeson определять, как именно разбирать ваши данные.

Проверка 40.2. Воспользуйтесь `Generic` для реализации `ToJSON` и `FromJSON` для этого типа:

```
data Name = Name
    { firstName :: T.Text
    , lastName :: T.Text
    } deriving (Show)
```

40.3.2. Написание собственных экземпляров FromJSON и ToJSON

В предыдущем примере вы начали с типа, определённого вами, и заставили его работать с JSON. На практике вам часто придётся работать с чужими JSON-данными. Вот пример сообщения об ошибке, которую вы можете получить в ответ на JSON-запрос из-за ошибки на стороне сервера.

Листинг 40.9 Пример неподконтрольного вам объекта JSON

```
sampleError :: BC.ByteString
sampleError = "{\"message\":\"упс!\",\"error\": 123}"
```

Ответ 40.2

```
data Name = Name
    { firstName :: T.Text
    , lastName :: T.Text
    } deriving (Show,Generic)

instance FromJSON Name
instance ToJSON Name
```

Для использования Aeson вам нужно сделать модель для этого запроса с помощью собственного типа. Когда вы сделаете это, то сразу увидите, что есть проблема. Вот первая попытка создания модели для этого сообщения об ошибке.

Листинг 40.10 Проблема с созданием JSON-модели на Haskell

```
data ErrorMessage = ErrorMessage
    { message :: T.Text
    , error :: Int
    } deriving Show
```

Ошибка из-за этого свойства

Ошибка произошла из-за того, что у вас есть свойство с названием `error`, но вы не можете его использовать, так как это имя в Haskell уже определено. Вы могли бы переписать данный тип и избежать коллизии.

Листинг 40.11 Тип, который не соответствует оригинальному JSON

```
data ErrorMessage = ErrorMessage
    { message :: T.Text
    , errorCode :: Int
    } deriving Show
```

К сожалению, если вы попытаетесь автоматически вывести `ToJSON` и `FromJSON`, ваша программа будет ожидать поле `errorCode` вместо `error`. Если у вас есть доступ к управлению этим JSON, вы могли бы просто переименовать поле, но такой возможности у вас нет. Вам требуется другое решение этой проблемы.

Чтобы сделать `ErrorMessage` экземпляром `FromJSON`, вам нужно определить ещё одну функцию `parseJSON`. Вы можете сделать это следующим образом.

Листинг 40.12 Реализация экземпляра FromJSON для ErrorMessage

```
instance FromJSON ErrorMessage where
    parseJSON (Object v) =
        ErrorMessage <$> v .: "message"
                     <*> v .: "error"
```

Этот код может сбить вас с толку, так что стоит его разобрать. Первая часть содержит название функции и аргумент, который она принимает:

```
parseJSON (Object v)
```

Здесь `(Object v)` — JSON-объект, разбор которого мы выполняем. Когда вы принимаете только внутреннюю часть `v`, вы получаете доступ к значению этого JSON-объекта. Далее следует несколько инфиксных операций, с которыми вам предстоит разобраться. Вы уже видели подобную конструкцию ранее, в модуле 5, когда изучали примеры работы с `Applicative`:

```
ErrorMessage <$> значение <*> значение
```

Предположим, к примеру, что значения для `ErrorMessage` находятся в контексте `Maybe`.

Листинг 40.13 Создаём ErrorMessage в контексте Maybe

```
exampleMessage :: Maybe T.Text
exampleMessage = Just "упс!"

exampleError :: Maybe Int
exampleError = Just 123
```

Если вы хотите создать значение типа `ErrorMessage`, можно скомбинировать `<$>` и `<*>` для безопасного создания `ErrorMessage` в контексте `Maybe`:

```
GHCi> ErrorMessage <$> exampleMessage <*> exampleError
Just (ErrorMessage {message = "упс!", errorCode = 123})
```

Эта конструкция работает со всеми экземплярами `Monad`. В данном случае вы работаете со значениями в контексте `Parser`, а не `Maybe`. Это приводит вас к заключительной тайне: что же такое операция `(.:)`? Вы можете раскрыть эту загадку, рассмотрев её тип повнимательнее.

```
(.:) :: FromJSON a => Object -> Text -> Parser a
```

Эта операция принимает `Object` (ваш JSON-объект) и некий текст, а возвращает значение, считанное в контексте `Parser`. Например, следующая строка попытается считать поле `message` из JSON-объекта:

```
v .: "message"
```

В результате будет получено значение в контексте `Parser`. Причина, по которой требуется контекст, заключается в том, что разбор может провалиться, если произошла ошибка считывания.

Проверка 40.3. Сделайте `Name` экземпляром `FromJSON` без `Generic`:

```
data Name = Name
    { firstName :: T.Text
    , lastName :: T.Text
    } deriving (Show)
```

Теперь `ErrorMessage` является экземпляром `FromJSON`, и вы, наконец, можете разобрать входящие сообщения об ошибках в виде JSON.

Листинг 40.14 Разбор JSON в тип ErrorMessage

```
sampleErrorMessage :: Maybe ErrorMessage
sampleErrorMessage = decode sampleError
```

В GHCi вы можете убедиться, что всё работает так, как предполагалось:

```
GHCi> sampleErrorMessage
Just (ErrorMessage {message = "упс!", errorCode = 123})
```

Теперь нужно проделать обратное преобразование. Синтаксис для создания сообщений немного другой:

```
instance ToJSON ErrorMessage where
    toJSON (ErrorMessage message errorCode) =
        object [ "message" .= message
               , "error" .= errorCode
               ]
```

И снова перед вами запутанный кусок кода. На этот раз вы определяете функцию `toJSON`. Вы можете увидеть, что эта функция принимает конструктор данных и сопоставляет с образцом два его компонента, `message` и `errorCode`:

```
toJSON (ErrorMessage message errorCode)
```

Ответ 40.3

```
instance FromJSON Name where
    parseJSON (Object v) =
        Name <$> v .: "firstName"
             <*> v .: "lastName"
```

Затем для создания JSON-объекта используется функция `object`, которая передаёт значения полей типа в соответствующие поля JSON-объекта:

```
object [ "message" .= message
       , "error" .= errorCode
       ]
```

Тут вас встречает очередная операция, на этот раз (`.=`). Она используется для создания пары ключ-значение, которая соответствует данным в конкретном поле JSON-объекта.

Проверка 40.4. Наконец, сделайте `Name` экземпляром `ToJSON` без использования `Generic`:

```
data Name = Name
    { firstName :: T.Text
    , lastName :: T.Text
    } deriving (Show)
```

Теперь вы можете создать собственный JSON-объект, точно такой же, как те, что вы получали.

Листинг 40.15 Сообщение об ошибке для преобразования в JSON

```
anErrorMessage :: ErrorMessage
anErrorMessage = ErrorMessage "Всё прекрасно" 0
```

И снова вы можете убедиться, что всё работает, как планировалось:

```
GHCi> encode anErrorMessage
"{\"error\":0,\"message\":\"Всё прекрасно\"}"
```

Теперь вы освоили основы работы с данными JSON с помощью Haskell, так что давайте переключимся на более сложную задачу.

Ответ 40.4

```
instance ToJSON Name where
    toJSON (Name firstName lastName) =
        object [ "firstName" .= firstName
               , "lastName" .= lastName
               ]
```

40.4. Чтение данных, полученных от NOAA

В предыдущем уроке вы изучили, как сохранить JSON-файл на жёсткий диск. Вы сохранили список наборов данных NOAA в файл data.json. Если вы не запускали код из урока 39, можно взять эти данные здесь: `https://gist.github.com/willkurt/9dc14babbffea1a30c2a1e121a81bc0a`. Теперь давайте приступим к чтению этого файла и выводу названий наборов данных. Интересной особенностью этого файла является то, что JSON, хранящийся в нём, не тривиален. В JSON-объекте есть вложенные поля, и он выглядит примерно так.

Листинг 40.16 JSON, полученный от NOAA

```
{
    "metadata":{
        "resultset":{
            "offset":1,
            "count":11,
            "limit":25
        }
    },
    "results":[
        {
            "uid":"gov.noaa.ncdc:C00861",
            "mindate":"1763-01-01",
            "maxdate":"2017-02-01",
            "name":"Daily Summaries",
            "datacoverage":1,
            "id":"GHCND"
        },
    .....
```

Далее вам предстоит создать модель всего ответа с помощью типа `NOAAResponse`. Значения этого типа будут содержать компоненты двух других типов: `Metadata` и `Results`. `Metadata`, в свою очередь, содержит ещё один тип `Resultset`. Затем у вас будет тип `NOAAResults`, который содержит значения.

Начнём с собственно результата, так как это именно то, в чём вы заинтересованы, и он не содержит никаких сложных типов. Так как внутри результата находится значение `id`, вам нужно будет определить собственную реализацию экземпляров. Назовём этот тип `NOAAResult`, чтобы отличать его от типа `Result` из Aeson.

Листинг 40.17 Тип NOAAResult с названиями наборов данных

```
data NOAAResult = NOAAResult { uid :: T.Text
                             , mindate :: T.Text
                             , maxdate :: T.Text
                             , name :: T.Text
                             , datacoverage :: Int
                             , resultId :: T.Text
                             } deriving Show
```

Так как в JSON вместо `resultId` используется `id`, вам придётся написать свою реализацию экземпляра `FromJSON`. Экземпляр `ToJSON` нас не заботит, поскольку нужны преобразования только в одном направлении.

Листинг 40.18 Определение экземпляра FromJSON для NOAAResult

```
instance FromJSON NOAAResult where
    parseJSON (Object v) =
        NOAAResult <$> v .: "uid"
                   <*> v .: "mindate"
                   <*> v .: "maxdate"
                   <*> v .: "name"
                   <*> v .: "datacoverage"
                   <*> v .: "id"
```

Далее нужно переключиться на тип `Metadata`. Первая часть этого типа — `Resultset`. К счастью, тут вам не потребуется собственная реализация `FromJSON`. Вам просто нужно будет определить требуемый тип, добавить `deriving (Generic)` и сделать его экземпляром нужного класса типов.

Листинг 40.19 Тип Resultset с порождением экземпляра FromJSON

```
data Resultset = Resultset { offset :: Int
                           , count :: Int
                           , limit :: Int
                           } deriving (Show,Generic)

instance FromJSON Resultset
```

Тип `Metadata` содержит только `Resultset`, поэтому его записать легко.

Листинг 40.20 Тип Metadata с порождением экземпляра FromJSON

```
data Metadata = Metadata { resultset :: Resultset
                         } deriving (Show,Generic)

instance FromJSON Metadata
```

Наконец, всё готово для соединения этих типов в `NOAAResponse`. Как и с другими типами, тут нет никаких конфликтов имён, поэтому можно спокойно породить требуемый класс.

Листинг 40.21 Соединение описанных типов в NOAAResponse

```
data NOAAResponse = NOAAResponse
                  { metadata :: Metadata
                  , results :: [NOAAResult]
                  } deriving (Show,Generic)

instance FromJSON NOAAResponse
```

Вашей целью является вывод всех значений из файла. Для этого вам нужно будет создать IO-действие `printResults`. Так как данные будут значением типа `Maybe`, вам нужно предусмотреть случай неудачной попытки разбора. В этом случае нужно вывести сообщение, что произошла ошибка. В противном случае вы воспользуетесь функцией `forM_` из модуля `Control.Monad` (не забудьте его импортировать) для обхода результатов и их печати. Функция `forM_` работает аналогично `mapM_`, только порядок аргументов — данных и функции, обрабатывающей эти данные, — обращён.

Листинг 40.22 Вывод результатов

```
printResults :: Maybe [NOAAResult] -> IO ()
printResults Nothing = putStrLn "ошибка загрузки данных"
printResults (Just results) = forM_ results (print . name)
```

Теперь вы можете написать функцию `main`, в которой будет происходить чтение файла, считывание JSON и обход результатов.

Листинг 40.23 Соединение всего написанного в функциии main

```
main :: IO ()
main = do
     jsonData <- B.readFile "data.json"
     let noaaResponse = decode jsonData :: Maybe NOAAResponse
     let noaaResults = results <$> noaaResponse
     printResults noaaResults
```

Теперь вы можете загрузить свой проект в GHCi (или же воспользуйтесь `stack build` для запуска, если предпочитаете такой способ) и посмотреть, как всё работает:

```
GHCi> main
"Daily Summaries"
"Global Summary of the Month"
"Global Summary of the Year"
"Weather Radar (Level II)"
"Weather Radar (Level III)"
"Normals Annual/Seasonal"
"Normals Daily"
"Normals Hourly"
"Normals Monthly"
"Precipitation 15 Minute"
"Precipitation Hourly"
```

Готово! Вы успешно применили Haskell для разбора сложного JSON-файла.

Итоги

В этом уроке нашей целью было научить вас, как читать и создавать JSON-файлы в программах на Haskell. Вы использовали популярную библиотеку Aeson, которая позволяет осуществлять переходы между JSON и типами в Haskell. Эти преобразования осуществляются с помощью двух классов типов `ToJSON` и `FromJSON`. В лучшем случае вы можете воспользоваться языковым расширением `DeriveGeneric` для порождения экземпляров этих классов автоматически. Даже в худшем случае, когда вам приходится помогать Aeson переводить данные, делать это сравнительно просто. Давайте посмотрим, как вы усвоили материал этого урока.

Задача 40.1. Сделайте `NOAAResponse` экземпляром `ToJSON`. Это потребует написания экземпляров `ToJSON` для всех вложенных типов.

Задача 40.2. Напишите тип-сумму `IntList` и сделайте его экземпляром `ToJSON`, используя `DeriveGeneric`. Не используйте существующий тип для списка, а напишите свой собственный с нуля. Вот пример значения типа `IntList`:

```
intListExample :: IntList
intListExample = Cons 1 $
                 Cons 2 EmptyList
```

41

Использование баз данных в Haskell

После прочтения урока 41 вы:

- научитесь присоединяться к базам данных SQLite с помощью Haskell;
- сможете переводить строки таблиц SQL в тип данных Haskell;
- разберётесь с созданием, чтением, обновлением и удалением данных из баз данных с помощью Haskell.

В этом уроке вы изучите, как работать с базами данных, используя Haskell. Если конкретнее, вы будете работать с системой управления реляционными базами данных SQLite3 (*relational database management system*, RDBMS) и библиотекой `sqlite-simple`. Изучение будет сопровождаться созданием текстового интерфейса для склада по сдаче инструментов в аренду. Это потребует использования всех задач CRUD, которые обычно ассоциируются с работой с БД и включают в себя:

- *create* — добавление новых данных в базу;
- *read* — запрос данных из базы;
- *update* — модификация существующей в базе данных информации;
- *delete* — удаление данных из базы.

Для взаимодействия со своей базой данных вы будете использовать библиотеку `sqlite-simple`, представляющую собой абстракцию *среднего уровня* над базой данных. Это означает, что хотя большинство низкоуровневых задач и не придётся решать вручную, вам всё равно придётся самостоятельно писать множество «сырых» SQL-запросов. Наиболее сильной абстракцией будет то, что `sqlite-simple` поможет вам преобразовывать результаты SQL-запросов в списки Haskell.

Вот краткое изложение того, что вам предстоит реализовать в проекте.

Предположим, что у вас есть друг, который хочет организовать небольшой склад по сдаче инструментов в аренду местным жителям. Ему требуется система для помощи в отслеживании инвентаря и сдачи инструментов обратно на склад. Если посмотреть на проблему, рассуждая в терминах БД, то становится понятно, что проект должен включать три таблицы: для инструментов, для пользователей и для данных по аренде. Если же переместиться на уровень Haskell, то вас будет заботить только создание моделей данных для пользователей и инструментов. К концу этого урока у вас будет приложение командной строки, которое сможет выполнять следующие операции:

- вывод информации обо всех пользователях и всех инструментах;
- печать всех арендованных и доступных инструментов;
- добавление новых пользователей в базу данных;
- перевод инструментов в категорию доступных после их возвращения;
- сохранение информации о количестве раз, которые инструмент был одолжен, и времени последней сдачи его в аренду.

К моменту завершения урока вы успеете поработать с большинством операций, связанных с базами данных, с помощью Haskell.

41.1. Первоначальная настройка проекта

В этом уроке вы будете работать с проектом, который называется `db-lesson`. Как и во всех других уроках этого модуля, весь код будет для простоты находиться в модуле `Main` (хотя этот проект можно с лёгкостью раздробить на несколько файлов). Вы начнёте с файла app/Main.hs. Вот первоначальное содержимое этого файла, включающее импорт модулей, с которыми вы будете работать.

Листинг 41.1 Первоначальная версия app/Main.hs

```
module Main where

import Control.Applicative
import Database.SQLite.Simple
import Database.SQLite.Simple.FromRow
import Data.Time

main :: IO ()
main = print "db-lesson"
```

Этот модуль пригодится вам для работы с SQLite

`FromRow` — важный класс типов, который вам потребуется в работе

В этом уроке вы также изучите типы, связанные с датами

В качестве зависимостей в cabal-файл проекта нужно будет добавить библиотеку `sqlite-simple`, которую вы будете использовать для взаимодействия со SQLite, и `time`, которая поможет вам работать с датами.

Листинг 41.2 Изменение списка зависимостей в db-lesson.cabal

```
build-depends:    base
                , db-notes
                , time
                , sqlite-simple
```

Также вы будете использовать расширение `OverloadedStrings`, поскольку библиотека SQLite Simple будет интерпретировать многие из ваших строк как SQL-запросы.

Листинг 41.3 Расширение OverloadedStrings в db-lesson.cabal

```
extensions:     OverloadedStrings
```

После добавления в проект всего вышеперечисленного, вы сможете выполнить команды `setup` и `build` и убедиться, что проект готов к разработке.

41.2. Использование SQLite и настройка базы данных

В этом уроке мы воспользуемся системой для управления базами данных SQLite3, так как её легко установить и начать работать. SQLite можно загрузить по адресу `www.sqlite.org`. Так как эта система была разработана для лёгкого развёртывания, установка в вашей операционной системе должна быть простой.

Складу потребуется база данных, содержащая таблицы пользователей, инструментов и сведений о сдаче инвентаря в аренду (какие инструменты были отданы и кому). На рис. 41.1 приведена простая схема связей между таблицами.

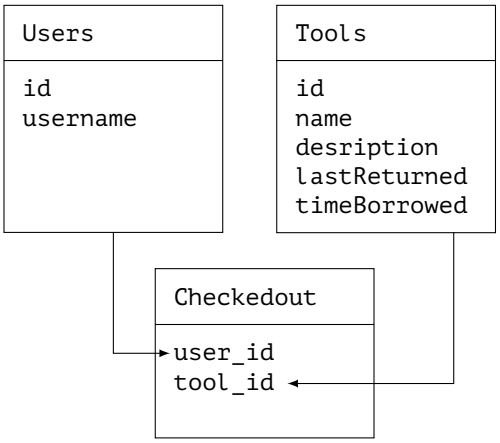


Рис. 41.1: Схема базы данных

Работу с проектом вы начнёте с тестовыми данными, занесёнными в базу. Вот код для создания таблиц базы данных и добавления тестовой информации; этот код следует записать в файл build_db.sql, который должен лежать в корневом каталоге проекта.

Листинг 41.4 Код для создания базы данных

```
DROP TABLE IF EXISTS checkedout;
DROP TABLE IF EXISTS tools;
DROP TABLE IF EXISTS users;

CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    username TEXT
    );

CREATE TABLE tools (
    id INTEGER PRIMARY KEY,
    name TEXT,
    description TEXT,
    lastReturned TEXT,
    timesBorrowed INTEGER
    );

CREATE TABLE checkedout (
    user_id INTEGER,
    tool_id INTEGER
    );

INSERT INTO users (username) VALUES ('уиллкурт');

INSERT INTO tools (name,description,lastReturned,timesBorrowed)
VALUES ('молоток','забивает всякое','2017-01-01',0);

INSERT INTO tools (name,description,lastReturned,timesBorrowed)
VALUES ('пила','пилит разное','2017-01-01',0);
```

SQLite не поддерживает типы для дат

Для запуска SQLite вам нужно ввести в командную строку `sqlite3`. Также вам нужно будет передать название базы данных, tools.db. Наконец, отправьте программе содержимое сохранённого ранее файла build.sql. Вот полная команда:

```
$ sqlite3 tools.db < build_db.sql
```

Вы можете проверить базу данных с помощью команды `sqlite3`, вызванной вместе с путём к файлу базы. После этого вы можете отправить несколько SQL-запросов, чтобы убедиться, что всё было корректно установлено. Префикс строки `sqlite>` сигнализирует, что вы находитесь в режиме интерактивной работы с БД:

```
$ sqlite3 tools.db
sqlite> select * from tools;
1|молоток|забивает всякое|2017-01-01|0
2|пила|пилит разное|2017-01-01|0
```

Так как вашей целью является работа с SQLite с помощью Haskell, вам больше не придётся напрямую использовать команду `sqlite`. Впрочем, если захочется, вы всегда можете перейти в режим интерактивной работы с базой данных и перепроверить результаты своей работы.

41.2.1. Данные на Haskell

Одной из основных трудностей при работе с Haskell и БД, такими как, например, SQLite, является то, что типы в Haskell обычно гораздо более мощные, чем в БД. SQLite, к примеру, не содержит никаких типов, представляющих даты. Эта ситуация похожа на ту, с которой вы уже сталкивались во время работы с JSON-данными. Прежде чем погрузиться в создание данных в базе, давайте посмотрим на данные с точки зрения Haskell.

Листинг 41.5 Определение типа Tool из app/Main.hs

```
data Tool = Tool { toolId :: Int
                 , name :: String
                 , description :: String
                 , lastReturned :: Day
                 , timesBorrowed :: Int
                 }
```

В определении типа `Tool` вы можете заметить тип `Day`, с которым ещё не сталкивались. Тип `Day` определён в модуле `Data.Time`, который также содержит множество функций, связанных с обработкой времени. Вот пример получения текущего времени с помощью функции `getCurrentTime` и преобразования его в тип `Day` с помощью функции `utctDay`:

```
GHCi> getCurrentTime
2017-02-26 07:05:12.218684 UTC
GHCi> utctDay <$> getCurrentTime
2017-02-26
```

Другим типом нашей модели данных является `User`. Этот тип гораздо проще `Tool`, он включает в себя только поля для идентификатора и имени.

Листинг 41.6 Тип User

```
data User = User { userId :: Int
                 , userName :: String
                 }
```

Типы `User` и `Tool` позволяют вам совершать вычисления с любыми данными, полученными из базы данных. Одной из наиболее частых операций будет вывод запрошенной информации о пользователях и инструментах. Вы можете убедиться, что результаты выглядят так, как вы хотите, сделав для этих данных экземпляры `Show`.

Листинг 41.7 Написание экземпляров User и Tool для Show

```
instance Show User where
    show user = mconcat [ show $ userId user
                        , ".) "
                        , userName user]

instance Show Tool where
    show tool = mconcat [ show $ toolId tool
                        , ".) "
                        , name tool
                        , "\n описание: "
                        , description tool
                        , "\n последний раз возвращено: "
                        , show $ lastReturned tool
                        , "\n количество раз сдано в аренду: "
                        , show $ timesBorrowed tool
                        , "\n"]
```

Когда вы попытаетесь вызвать печать этих данных, вывод должен выглядеть примерно так:

```
1.) уиллкурт

1.) молоток
    описание: забивает всякое
    последний раз возвращено: 2017-01-01
    количество раз сдано в аренду: 0
```

Теперь всё готово для начала взаимодействия с вашей базой данных!

Проверка 41.1. Почему для конкатенации строк лучше использовать `mconcat`, а не ++ (выберите верный ответ)?

(1) меньше набор символов для набора;

(2) ++ работает только на списках, но не на тексте;

(3) `mconcat` облегчает поддержку кода для текстовых типов.

41.3. Вставка данных: пользователи и данные об аренде

Первой из операций, зашифрованных в аббревиатуре CRUD, мы рассмотрим *create*. Вы только что использовали голый SQL для создания таблиц и данных; теперь сделаем то же самое в Haskell. Для добавления данных в базу вам нужно к ней подключиться, создать строку с SQL-запросом и затем его выполнить.

41.3.1. Добавление новых пользователей в базу данных

К этому моменту у вас в базе данных есть только один пользователь. Вам нужна команда для добавления пользователей. Начнём с написания функции, которая будет принимать имя пользователя `userName` и добавлять его в базу данных. Для этого вы будете использовать команду `execute`, которая позволит вам вставлять записи о пользователях в базу. Строка, содержащая запрос, будет содержать подстроку `(?)` на месте добавляемых значений, что позволит безопасно передавать эти значения для формирования SQL-запроса в виде строки. Прежде чем добавить пользователя, вам нужно установить соединение с базой данных. Затем это соединение будет передаваться вместе с запросами и их параметрами действию `execute`. Вот код действия `addUser`.

Ответ 41.1. (3) — хотя в этом уроке мы используем `String`, обычно отдаётся предпочтение типу `Text`. Функция `mconcat` работает со всеми основными типами строк — `String`, `Text` и `ByteString`, что позволяет без особых проблем переходить на новые типы, просто изменяя типовые аннотации.

Листинг 41.8 Соединение с БД и вставка данных пользователя

```
addUser :: String -> IO ()
addUser userName = do
    conn <- open "tools.db"
    execute conn "INSERT INTO users (username) VALUES (?)"
            (Only userName)
    putStrLn "пользователь добавлен"
    close conn
```

Сначала нужно установить соединение с базой данных

Конструктор `Only` используется для передачи одноэлементного кортежа с параметром запроса

Выполняем команду, передавая соединение и запрос с параметрами

Важно закрыть соединение после завершения операции

Неплохое начало, но большая часть кода этого проекта должна иметь доступ к базе данных, поэтому фрагменты этого IO-действия будут повторяться слово в слово. Вы можете немного абстрагироваться от конкретных действий с БД, определив функцию `withConn`, которая автоматически будет обрабатывать открытие и закрытие базы данных.

Листинг 41.9 Абстрагирование соединения с базой данных

```
withConn :: String -> (Connection -> IO ()) -> IO ()
withConn dbName action = do
    conn <- open dbName
    action conn
    close conn
```

Эта функция принимает строку, которая является именем базы данных, и действие, которое использует соединение в качестве аргумента. Конечным результатом будет действие типа `IO ()`.

Проверка 41.2. Перепишите `addUser`, воспользовавшись `withConn`.

Ответ 41.2

```
addUser :: String -> IO ()
addUser userName =
  withConn "tools.db" $
    \conn -> do
      execute conn "INSERT INTO users (username) VALUES (?)"
              (Only userName)
      putStrLn "пользователь добавлен"
```

41.3.2. Создание записей об аренде

Другим полезным дополнением будет функция `checkout` для добавления записей об аренде. Функции `checkout` требуется и `userId`, и `toolId`. Определение `checkout` напоминает код для добавления пользователей, только этой функции требуется передать два значения.

Листинг 41.10 Вставка записи об аренде

```
checkout :: Int -> Int -> IO ()
checkout userId toolId =
  withConn "tools.db" $
   \conn ->
      execute conn "INSERT INTO checkedout (user_id, tool_id)
                  ↳ VALUES (?,?)" (userId, toolId)
```

Заметьте, что `(userId, toolId)` представляет собой обычный кортеж, для которого не требуется конструктор `Only`.

Функции `checkout` и `addUser` формируют фундамент для многих ключевых операций, которые следует реализовать в этом проекте. Вы можете протестировать эти функции, но пока у вас нет способа проверить корректность результатов, кроме как открыть SQLite для ручной проверки изменённой базы данных. В следующем разделе вы рассмотрите проблему чтения информации из базы данных и перевода этих данных в типы Haskell.

41.4. Чтение данных из БД и класс типов FromRow

Трудностью при работе с данными SQL в Haskell является то, что вам нужен простой способ создавать значения типов Haskell из сырых данных. Для облегчения этого процесса библиотека `sqlite-simple` определяет класс типов `FromRow`. Вот его определение, содержащее всего одну требующую реализации функцию `fromRow`.

Листинг 41.11 Определение класса типов FromRow

```
class FromRow a where
    fromRow :: RowParser a
```

Функция `fromRow` возвращает значение типа `RowParser a`, где `a` — тип, для которого вы создаёте экземпляр `FromRow`. Вам не придётся напрямую использовать функцию `fromRow`, но она будет использоваться другими функциями для запроса данных. Если вы реализуете `FromRow`, то сможете получать результаты запросов в виде списков значений Haskell.

41.4.1. Написание реализации класса FromRow

Написание реализаций класса типов FromRow похоже на создание экземпляров FromJSON, рассмотренное в предыдущем уроке. Вы должны сообщить RowParser, как создавать требуемые типы данных. Ключевой здесь является функция field из модуля SQLite.Simple. Функция field используется внутри SQLite.Simple для извлечения данных из SQL-строки и преобразования их в значения с помощью ваших конструкторов данных. Вот экземпляры FromRow для User и Tool.

Листинг 41.12 Написание реализаций FromRow для User и Tool

```
instance FromRow User where
    fromRow = User <$> field
                   <*> field

instance FromRow Tool where
    fromRow = Tool <$> field
                   <*> field
                   <*> field
                   <*> field
                   <*> field
```

Теперь и User, и Tool являются экземплярами FromRow, поэтому вы можете осуществлять запросы к вашей базе данных и записывать их результаты напрямую в списки пользователей или, соответственно, инструментов.

41.4.2. Получение списков пользователей и инструментов

Чтобы сделать запрос к данным, можно использовать две связанные функции: query и query_ (обратите внимание на нижнее подчёркивание). Посмотрев внимательно на типы этих функций, вы можете увидеть разницу между ними:

```
query :: (FromRow r, ToRow q) =>
                      ↳ Connection -> Query -> q -> IO [r]
query_ :: FromRow r => Connection -> Query -> IO [r]
```

Типовые аннотации почти идентичны, но версия с подчёркиванием принимает на один аргумент меньше. Функция query ожидает, что ей будет передана строка запроса и его параметр. Функция query_ существует для запросов, которым не требуется передавать параметры. Также обратите внимание, что Query представляет собой отдельный тип. До сих пор вы относились к запросам как к строкам, но благодаря расширению OverloadedStrings всё автоматически преобразуется к нужному типу.

Проверка 41.3. Зачем требуется две функции `query` и `query_`?

Вы будете использовать эти запросы для печати пользователей и инструментов. Вот функция `printUsers`; обратите внимание, что вам нужно указать тип того, что вы запрашиваете.

Листинг 41.13 Печать пользователей из базы данных

```
printUsers :: IO ()
printUsers =
  withConn "tools.db" $
    \conn -> do
      resp <- query_ conn "SELECT * FROM users;" :: IO [User]
      mapM_ print resp
```

Функция `printUsers`, пользуясь тем, что `User` — экземпляр `Show`, отображает данные о пользователе так, как вы того хотели. Теперь, когда вы можете выводить данные о пользователях, можно проверить их добавление:

```
GHCi> printUsers
1.) уиллкурт
GHCi> addUser "тестовый пользователь"
пользователь добавлен
GHCi> printUsers
1.) уиллкурт
2.) тестовый пользователь
```

Следующей вашей целью является печать списка инструментов. Единственная сложность при работе с инструментами возникает в связи с необходимостью выполнять несколько видов запросов:

- все инструменты;
- инструменты, находящиеся в аренде;
- доступные на данный момент инструменты.

Ответ 41.3. В основном из-за того, как Haskell работает с типами, в нём не поддерживается переменное количество аргументов. Альтернативой написанию нескольких функций может служить тип-сумма для представления обоих наборов аргументов и сопоставлением с образцом в определении функции.

Для облегчения задачи нужно написать функцию `printToolQuery`, которая будет принимать запрос и выводить список инструментов, возвращённых после выполнения запроса. Вот функция `printToolQuery` вместе с другими функциями, которые её используют.

Листинг 41.14 Функции запросов на вывод списков инструментов

```
printToolQuery :: Query -> IO ()
printToolQuery q = withConn "tools.db" $
                   \conn -> do
                       resp <- query_ conn q :: IO [Tool]
                       mapM_ print resp

printTools :: IO ()
printTools = printToolQuery "SELECT * FROM tools;"

printAvailable :: IO ()
printAvailable = printToolQuery $
      mconcat [ "select * from tools "
              , "where id not in "
              , "(select tool_id from checkedout);"]

printCheckedout :: IO ()
printCheckedout = printToolQuery $
      mconcat [ "select * from tools "
              , "where id in "
              , "(select tool_id from checkedout);"]
```

Проверим эти функции вместе с написанной ранее `checkout` и удостоверимся, что они работают так, как было задумано.

```
GHCi> printTools
1.) молоток
описание: забивает всякое
последний раз возвращено: 2017-01-01
количество раз сдано в аренду: 0

2.) пила
описание: пилит разное
последний раз возвращено: 2017-01-01
количество раз сдано в аренду: 0

GHCi> checkout 1 2
GHCi> printCheckedout
2.) пила
описание: пилит разное
```

```
последний раз возвращено: 2017-01-01
количество раз сдано в аренду: 0
```

Осталось два крупных шага, прежде чем проект будет готов. Вам нужно научиться отражать в БД возвращение инструментов, а после возвращения нужно обновлять информацию о них. Следующим пунктом в изучении CRUD является *update*, поэтому приступим к рассмотрению способов обновления данных в базе.

41.5. Модификация существующих данных

Когда пользователь возвращает инструмент на склад, вам следует произвести два обновления. Во-первых, нужно увеличить существующее значение `timesBorrowed` на единицу; во-вторых, следует обновить дату, хранящуюся в `lastReturned`, заменив её на текущую. Это требует модификации существующей строки в базе данных, что является самым сложным шагом, если вы хотите быть уверены, что избежали ошибок.

Первым делом вам нужно найти соответствующий инструмент в базе данных, используя его идентификатор. Функция `selectTool` принимает соединение и идентификатор инструмента и возвращает значение довольно сложного типа `IO (Maybe Tool)`. Наличие `IO` показывает, что операции с базами данных всегда происходят в контексте `IO`, а внутренний `Maybe` используется, так как может быть передан некорректный ID, что выльется в результат запроса. Вот реализация `selectTool` и вспомогательной функции `firstOrNothing`.

Листинг 41.15 Безопасная выборка значения Tool, найденного по ID

```
selectTool :: Connection -> Int -> IO (Maybe Tool)
selectTool conn toolId = do
    resp <- query conn
                "SELECT * FROM tools WHERE id = (?)"
                (Only toolId) :: IO [Tool]
    return $ firstOrNothing resp

firstOrNothing :: [a] -> Maybe a
firstOrNothing [] = Nothing
firstOrNothing (x:_) = Just x
```

Функция `firstOrNothing` просматривает список, полученный в результате запроса. Если список пустой, то возвращается `Nothing`; если же по-

лучен другой результат (скорее всего, список из одного элемента, так как ID уникален), то возвращается только первый элемент.

После получения записи об инструменте вам нужно её обновить. Получение текущей даты требует IO-действия, поэтому, чтобы сохранить чистоту функции `updateTool`, предположим, что дата передаётся в функцию в виде аргумента. Эта функция будет принимать существующий инструмент и возвращать новый, в котором будут обновлены поля `lastReturned` (дата возврата) и `timesBorrowed` (количество сдач в аренду) с помощью синтаксиса записей (см. урок 11).

Листинг 41.16 Обновление информации об инструменте

```
updateTool :: Tool -> Day -> Tool
updateTool tool date = tool
    { lastReturned = date
    , timesBorrowed = 1 + timesBorrowed tool
    }
```

Затем вам нужен способ добавления обновления `Maybe Tool`. Так как инструмент представлен в виде `Maybe Tool`, вам нужно убедиться, что таблица обновляется, только если значение `Maybe` не является `Nothing`. Функция `updateOrWarn` выведет сообщение, что инструмент не был найден, если получит `Nothing`; в противном случае эта функция обновит необходимые поля в базе данных.

Листинг 41.17 Безопасное обновление базы данных

```
updateOrWarn :: Maybe Tool -> IO ()
updateOrWarn Nothing = putStrLn "id не был найден"
updateOrWarn (Just tool) =
  withConn "tools.db" $
    \conn -> do
      let q = mconcat [ "UPDATE TOOLS SET "
                      , "lastReturned = ?,"
                      , " timesBorrowed = ? "
                      , "WHERE ID = ?;"]
      execute conn q (lastReturned tool,
                      timesBorrowed tool,
                      toolId tool)
      putStrLn "данные об инструменте обновлены"
```

Остаётся связать все эти шаги вместе. Функция `updateToolTable` принимает `toolId`, запрашивает текущую дату, а затем выполняет необходимые для обновления базы данных шаги.

Листинг 41.18 Функция updateToolTable обновляет БД

```
updateToolTable :: Int -> IO ()
updateToolTable toolId =
  withConn "tools.db" $
    \conn -> do
      tool <- selectTool conn toolId
      currentDay <- utctDay <$> getCurrentTime
      let updatedTool = updateTool <$> tool
                                   <*> pure currentDay
      updateOrWarn updatedTool
```

Функция `updateToolTable` позволяет вам безопасно обновлять таблицу инструментов, а также проинформирует вас, если случится какая-нибудь ошибка при обновлении данных. Финальным шагом будет фиксация факта возвращения инструмента, что в нашем случае означает удаление строки из таблицы `checkedout`.

Класс типов ToRow

При необходимости вы также можете воспользоваться классом типов `ToRow`. Впрочем, он гораздо менее полезен, поскольку просто переводит типы данных в кортежи значений. Как вы могли увидеть в примере с созданием и обновлением значений, у вас либо не хватает какой-то информации (в случае создания), либо требуется только определённое её подмножество. Для справки, вот пример того, как можно сделать `Tool` экземпляром `ToRow` (заметьте, что этот код требует импортировать `Data.Text` как `T`):

```
instance ToRow Tool where
  toRow tool =
    [ SQLInteger $ fromIntegral $ toolId tool
    , SQLText $ T.pack $ name tool
    , SQLText $ T.pack $ description tool
    , SQLText $ T.pack $ show $ lastReturned tool
    , SQLInteger $ fromIntegral $ timesBorrowed tool ]
```

Конструкторы `SQLText` и `SQLInteger` преобразуют здесь типы `Text` и `Integer` в SQL-данные. На практике вам гораздо чаще придётся использовать `FromRow`, нежели `ToRow`. Но всё равно неплохо бы знать, что этот класс типов существует.

41.6. Удаление данных из БД

Последним аккордом изучения операций CRUD будет удаление (*delete*). Удаление данных — достаточно простая задача — как и при вставке данных, здесь используется функция `execute`. Функция `checkin` принимает `toolId` и удаляет строку из таблицы записей об аренде. Так как каждый инструмент может быть передан в аренду только одному человеку, идентификатор инструмента — единственная информация, которая потребуется.

Листинг 41.19 Возвращение инструмента в функции checkin

```
checkin :: Int -> IO ()
checkin toolId =
  withConn "tools.db" $ \conn ->
      execute conn
              "DELETE FROM checkedout WHERE tool_id = (?);"
              (Only toolId)
```

Как упоминалось в предыдущем разделе, нашей целью не является простое возвращение инструмента на склад, нужно также убедиться, что информация о нём была обновлена. Итоговое действие `checkinAndUpdate` будет сначала вызывать `checkin`, а затем `updateToolTable`.

Листинг 41.20 Полное обновление информации об инструменте

```
checkinAndUpdate :: Int -> IO ()
checkinAndUpdate toolId = do
    checkin toolId
    updateToolTable toolId
```

К этому моменту вы увидели все операции CRUD в действии при работе с базами данных при помощи Haskell. Теперь мы можем доделать остающиеся части интерфейса и всё проверить.

41.7. Собираем всё вместе

Вы написали всё, что требовалось для взаимодействия с базой данных. Остаётся обернуть эти действия в удобный интерфейс. Большинство изменений базы данных требует от пользователя либо имя, либо ID. Вот несколько `IO`-действий, которые помогают реализовать эти требования.

Листинг 41.21 Организация взаимодействия с БД

```
promptAndAddUser :: IO ()
promptAndAddUser = do
    putStrLn "Введите имя нового пользователя:"
    userName <- getLine
    addUser userName

promptAndCheckout :: IO ()
promptAndCheckout = do
    putStrLn "Введите ID пользователя:"
    userId <- pure read <*> getLine
    putStrLn "Введите ID инструмента:"
    toolId <- pure read <*> getLine
    checkout userId toolId

promptAndCheckin :: IO ()
promptAndCheckin = do
    putStrLn "Введите ID инструмента:"
    toolId <- pure read <*> getLine
    checkinAndUpdate toolId
```

Вы можете связать все эти действия воедино, разработав единую функцию, которая ожидает ввода от пользователя, принимает команду, а затем выполняет её. Заметьте, что каждая из этих команд, исключая `quit`, использует операцию >> (которая выполняет действие, отбрасывает результат и выполняет следующее действие) для вызова `main`. Это позволяет командному интерфейсу повторно запрашивать команды от пользователя, пока не будет получена команда выхода из программы `quit`.

Листинг 41.22 Пользовательский интерфейс в командной строке

```
performCommand :: String -> IO ()
performCommand "users" = printUsers >> main
performCommand "tools" = printTools >> main
performCommand "adduser" = promptAndAddUser >> main
performCommand "checkout" = promptAndCheckout >> main
performCommand "checkin" = promptAndCheckin >> main
performCommand "in" = printAvailable >> main
performCommand "out" = printCheckedout >> main
performCommand "quit" = putStrLn "Чао!"
performCommand _ = putStrLn "Команда не найдена" >> main
```

Проверка 41.4. Почему нельзя воспользоваться >>= вместо >> (выберите правильный ответ)?

(1) на самом деле можно, это будет нормально работать;

(2) >>= подразумевает, что `main` принимает аргумент, но это не так;

(3) >>= не является корректной операцией Haskell.

Нам удалось вынести большую часть кода в отдельные функции, так что функция `main` получилась довольно лаконичной.

Листинг 41.23 Окончательная версия IO-действия main

```
main :: IO ()
main = do
    putStrLn "Введите команду:"
    command <- getLine
    performCommand command
```

Внимательный читатель может заметить, что `performCommand` вызывает `main`, а `main` выполняет `performCommand`, что приводит к рекурсии. В большинстве языков это был бы прямой путь к переполнению стека, но Haskell может это обработать. Компилятор заметит, что функции вызывают друг друга, и сможет безопасно это всё оптимизировать.

Теперь вы можете собрать и запустить свой проект (вводимые пользователем строки выделены полужирным):

```
$ stack exec db-lesson-exe
Введите команду:
users
1.) уиллкурт
Введите команду:
adduser
Введите имя нового пользователя:
test user
пользователь добавлен
```

Ответ 41.4. (2) — при использовании >>= вы передаёте аргумент в контексте; операция >> используется, когда вы хотите связать действия в цепочку и отбросить их результаты.

```
Введите команду:
tools
1.) молоток
 описание: забивает всякое
 последний раз возвращено: 2017-01-01
 количество раз сдано в аренду: 0

2.) пила
 описание: пилит разное
 последний раз возвращено: 2017-01-01
 количество раз сдано в аренду: 0

Введите команду:
checkout
Введите ID пользователя:
1
Введите ID инструмента:
2
Введите команду:
out
2.) пила
 описание: пилит разное
 последний раз возвращено: 2017-01-01
 количество раз сдано в аренду: 0

Введите команду:
checkin
Введите ID инструмента:
2
данные об инструменте обновлены
Введите команду:
in
1.) молоток
 описание: забивает всякое
 последний раз возвращено: 2017-01-01
 количество раз сдано в аренду: 0

2.) пила
 описание: пилит разное
 последний раз возвращено: 2017-02-26
 количество раз сдано в аренду: 1

Введите команду:
quit
Чао!
```

Вы успешно реализовали все операции CRUD в Haskell и создали удобный инструмент, который ваш друг мог бы использовать для управления своим складом для сдачи инструментов в аренду.

Итоги

В этом уроке нашей целью было научить вас, как создать простую программу, основанную на взаимодействии с базой данных с использованием модуля `SQLite.Simple`. Вы смогли воспользоваться экземпляром `FromRow`, чтобы облегчить пользователям преобразование данных SQLite3 в типы Haskell. Вы изучили, как создавать, читать, обновлять и удалять данные из базы данных, используя Haskell. В конце урока вы разработали простое приложение, которое позволяет вам выполнять набор задач, связанных с управлением складом инструментов. Давайте посмотрим, насколько хорошо вы разобрались в этом материале.

Задача 41.1. Определите `IO`-действие `addTool`, которое добавляет в базу данных новые инструменты аналогично `addUser`.

Задача 41.2. Добавьте команду `addtool`, которая запрашивает у пользователя необходимую информацию, а затем добавляет в базу данных новый инструмент, используя функцию `addTool` из предыдущего упражнения.

42

Эффективные массивы с изменением состояния в Haskell

После прочтения урока 42 вы:

- сможете применять тип `UArray` для эффективного хранения и поиска;
- научитесь вычислять с изменением значений на массивах с `STUArray`;
- разберётесь с интерпретацией надёжно инкапсулированных, изменяющих состояние функций как чистых.

После завершения этой книги вы получаете звонок от HR-менеджера GooMicroBook, который приглашает вас на собеседование. Вы говорите, что с радостью придёте, и менеджер упоминает, что на собеседовании будет задание по программированию на языке, который вы предпочитаете. «На любом языке?» — с нетерпением спрашиваете вы. HR-менеджер подтверждает, что да, действительно можно использовать любой язык. С восторгом в голосе вы сообщаете, что хотите пройти испытание на Haskell. Вы добираетесь до места проведения собеседования, в комнату заходит интервьюер и просит решить несколько записанных на доске алгоритмических задач. После того как вы много месяцев буквально жили Haskell, вы едва дожидаетесь момента, когда сможете продемонстрировать свою принадлежность к программистской элите. Собеседование начинается с традиционного задания, вас просят реализовать односвязный список. Вы подбегаете к доске и выписываете следующее:

```
data MyList a = EmptyList | Cons a (MyList a)
```

К вашей плохо скрываемой радости, интервьюер немного смущена, когда вы сообщаете ей: «Готово!» Вы начинаете напыщенную речь, расписывая

достоинства Haskell, рассказывая о мощи системы типов и преимуществах чистых функций. После того как она терпеливо вас выслушивает, скромно умалчивая о своей диссертации по теории типов, она говорит, что впечатлена, и добавляет: «Отлично, раз вы настолько хорошо справились с этим заданием, следующий вопрос будет лёгким». Вы готовы продемонстрировать, насколько хорошо знаете Haskell; возможно, вы даже сможете покрасоваться применением монад! Интервьюер продолжает: «Я хочу попросить вас реализовать простенькую сортировку массива методом пузырька, только, пожалуйста, без создания копии массива». Внезапно вы понимаете, что Haskell, возможно, был не лучшим выбором для собеседования.

Реализация сортировки пузырьком в Haskell сопряжена с несколькими большими сложностями. Во-первых, вы сильно полагались на списки в качестве основной структуры данных, но они даже близко не стоят с массивами по эффективности в решении подобных проблем. Гораздо более серьёзной сложностью является требование реализовать сортировку без создания копии массива. Практически всё, что вы писали в этой книге, основывалось на изменении структур данных в функциональном стиле — с созданием новой версии структуры данных и отбрасыванием старой. Для большинства задач это довольно эффективно и легко. Сортировка массива — одна из тех задач, где ради эффективности необходимо изменять состояние.

К счастью, если вы прочитаете этот урок, то будете готовы к такому развитию событий. Вы закончите эту книгу, решив проблему, которая казалась неподъёмной в Haskell, — эффективная сортировка массива без создания копии. Вы узнаете о строгом (неленивом) типе массивов `UArray`. Затем вы увидите, что для модификации содержимого массива существует контекст `STUArray`. Наконец, вы соедините всё это вместе для реализации алгоритма сортировки пузырьком. Даже учитывая, что пузырьковая сортировка — не самый эффективный алгоритм, ваш код будет гораздо быстрее, чем если бы вы использовали списки.

Обратите внимание. Во многих случаях изменение состояния может привести к трудно уловимым ошибкам в коде. Но изменение состояния необходимо почти во всех эффективных алгоритмах сортировки массивов. Причиной, по которой в Haskell избегают изменения состояния, является то, что такие манипуляции нарушают принципы ссылочной прозрачности: функции всегда должны возвращать одинаковые результаты на одинаковых значениях.

Но даже в объектно-ориентированном программировании бывает желательно и временами возможно реализовать идеальную инкапсуляцию. Даже с учётом изменения состояния внутри объекта можно сделать так, чтобы программист этого не заметил и ссылочная прозрачность была сохранена. Если вы получите возможность, используя типы, убедиться, что код, меняющий состояние, идеально инкапсулирован, сочтёте ли вы такое программирование безопасным?

42.1. Тип UArray и эффективные массивы

В своём сражении с сортировкой методом пузырька вы столкнётесь с тремя связанными с эффективностью проблемами:

- списки по своей природе медленнее массивов при выполнении операций доступа к значениям;
- на практике ленивые вычисления могут существенно ударить по производительности;
- сортировка элементов массива требует программирования с изменением состояния.

Первые две проблемы могут быть решены с помощью типа `UArray`. В следующем разделе вы в деталях рассмотрите, как `UArray` может здесь помочь, а также изучите основы создания массивов типа `UArray`.

42.1.1. Неэффективность ленивых списков

Первой трудностью является то, что вам нужен тип массивов. До сих пор при чтении этой книги вы в основном полагались на списки. Но для таких проблем, как, например, сортировка, списки обычно невероятно неэффективны. Одной из причин подобной неэффективности является невозможность напрямую получать доступ к элементам списка. В первом модуле вы узнали об операции доступа к элементу списка (`!!`). Однако если

вы создадите достаточно большой список, то сможете увидеть, насколько плохо эта операция работает на практике.

Листинг 42.1 Пример списка из 10 миллионов элементов

```
aLargeList :: [Int]
aLargeList = [1 .. 10000000]
```

Чуть ранее вы узнали, что GHCi можно использовать для выяснения времени работы функции с помощью команды `:set +s`. Воспользовавшись этой командой, вы можете увидеть, как много времени занимает получение доступа к последнему элементу списка:

```
GHCi> :set +s
GHCi> aLargeList !! 9999999
10000000
(0.05 secs, 460,064 bytes)
```

Вы можете увидеть, что это заняло 0.05 секунды (50 миллисекунд). Хотя это не так уж и медленно в сравнении с HTTP-запросами, это всё равно довольно медленно для доступа к элементу.

Для работы с массивами мы будем использовать тип `UArray`. Для сравнения, вот значение типа `UArray` того же размера (мы объясним создание массива в следующем разделе).

Листинг 42.2 Массив типа UArray с 10 миллионами элементов

```
aLargeArray :: UArray Int Int
aLargeArray = array (0,9999999) []
```

Когда вы проведёте аналогичный тест в GHCi, используя операцию доступа к элементам массива (`!`), то обнаружите, что теперь доступ осуществляется практически мгновенно:

```
GHCi> aLargeArray ! 9999999
0
(0.00 secs, 456,024 bytes)
```

Буква `U` в названии типа `UArray` означает *неупакованный (unboxed)*. Неупакованные массивы не используют ленивость при вычислениях. Что-то подобное мы уже видели в модуле 4 при рассмотрении типов `Text` и `ByteString`. Ленивые вычисления, хотя и являются мощным свойством языка, во многих случаях могут быть причиной неэффективности.

Для изучения производительности ленивых вычислений давайте посмотрим на модифицированную версию нашего большого списка.

Листинг 42.3 Удвоение значений в списке и производительность

```
aLargeListDoubled :: [Int]
aLargeListDoubled = map (*2) aLargeList
```

Теперь в GHCi (с `:set +s`) вы можете увидеть, что происходит, когда вы пытаетесь получить длину `aLargeListDoubled`:

```
GHCi> length aLargeListDoubled
10000000
(1.58 secs, 1,680,461,376 bytes)
```

Ого! Получение длины списка заняло 1.58 секунды. А ещё более поразительно то, что эта операция потребовала 1.68 гигабайта оперативной памяти! Природная неэффективность списков не может быть причиной такой расточительной траты ресурсов. Вы можете продолжить изучение, запустив этот код в GHCi ещё раз (в том же сеансе):

```
GHCi> length aLargeListDoubled
10000000
(0.07 secs, 459,840 bytes)
```

Для понимания причин таких результатов вам нужно вспомнить принципы работы ленивых вычислений. Никакие вычисления не производятся, пока их результат вам не потребуется. В первую очередь это включает в себя генерацию списков. Когда вы определяете `aLargeList`, Haskell сохраняет вычисления, требуемые для генерации такого списка. Когда вы умножаете элементы списка на 2 для создания `aLargeListDoubled`, Haskell ничего не вычисляет. Наконец, когда вы выводите длину списка, Haskell приступает к работе и начинает создавать список, вспоминая, что ему ещё нужно умножить каждое значение. Все подобные вычисления, которые Haskell планирует выполнить (они называются *задумками*, от англ. *thunks*), хранятся в памяти. В случае работы с маленькими списками это не особенно сильно бьёт по производительности, но вы можете видеть, насколько затрагивается производительность при работе с большими списками. Учитывая, что 10 миллионов символов — это не особо большой объём текста, `Data.Text` предпочтительнее использовать, чем `String`.

Особенностью работы с неупакованными массивами является то, что они работают только с примитивными типами: `Int`, `Char`, `Bool` и `Double`. Ещё Haskell предоставляет более обобщённый тип, `Array`, который будет работать с любыми данными так же, как списки, но `Array` — ленивая структура данных. Нашей же целью в этом уроке является эффективность классических алгоритмов. Для этих целей лучше всего подходит `UArray`. Чтобы

использовать тип UArray, вам нужно импортировать Data.Array.Unboxed в шапке модуля. Кроме того, если вы используете stack, вам нужно добавить array к списку зависимостей.

42.1.2. Создание массивов типа UArray

Как и в большинстве языков программирования, когда вы создаёте массив в Haskell, вам нужно указать его размер. Но, в отличие от многих других языков, у вас есть возможность выбирать, чем являются индексы массива! Тип UArray принимает два параметра: первый — тип индексов, а второй — тип значений. У вас есть определённая гибкость в вопросе выбора типа индексов. Вы можете использовать типы, которые являются членами классов Enum и Bounded. Это означает, что индексами могут быть значения типа Char или Int, но не Double (не экземпляр Enum) и не Integer (не экземпляр Bounded). Вы могли бы даже использовать индексы типа Bool, но это наложило бы ограничение на размер массива в 2 элемента. В большинстве случаев вам будет удобно использовать в качестве индексов значения типа Int от 0. Для создания значения типа UArray используется функция array. Она принимает два аргумента:

- первый — пара значений в кортеже, представляющих нижнюю и верхнюю грани для индексов;
- второй — список пар вида (индекс, значение).

Если какие-то индексы среди пар пропущены, то Haskell подставит значения по умолчанию. Для Int это будет 0, а для Bool — False. Вот пример создания массива значений типа Bool с индексацией с нуля. Только одно значение равно True, остальные по умолчанию будут False.

Листинг 42.4 Создание массива Bool, индексируемого с нуля

```
zeroIndexArray :: UArray Int Bool
zeroIndexArray = array (0,9) [(3,True)]
```

Вы можете получить доступ к значениям в UArray, используя операцию ! (аналогично операции !! для списков). В GHCi вы можете увидеть, что все значения, которые не были явно указаны в списке пар, равны False:

```
GHCi> zeroIndexArray ! 5
False
GHCi> zeroIndexArray ! 3
True
```

Если вы занимаетесь вычислительной математикой с использованием таких языков, как, например, R и Matlab, то вы, наверное, привыкли к массивам, которые индексируются с 1. Большинство языков программирования, нацеленных на математические вычисления, используют массивы с индексацией с 1, так это больше соответствует нотации, принятой в математике. В Haskell легко можно изменить систему индексов массивов, просто передав другую пару границ. Вот массив `Bool`, в котором индексы начинаются с 1, а все элементы равны `True`. Для генерации списка пар из 10 элементов, вторые элементы которых равны `True`, можно воспользоваться функциями `zip` и `cycle`.

Листинг 42.5 Массив, индексируемый с единицы

```
oneIndexArray :: UArray Int Bool
oneIndexArray = array (1,10) $ zip [1 .. 10] $ cycle [True]
```

Как вы можете видеть в GHCi, все элементы массива равны `True`, а индексы начинаются с 1 и заканчиваются на 10:

```
GHCi> oneIndexArray ! 1
True
GHCi> oneIndexArray ! 10
True
```

Как и в большинстве других языков программирования, если вы попытаетесь получить доступ к элементу за пределами диапазона индексов, то получите ошибку:

```
GHCi> oneIndexArray ! 0
*** Exception: Ix{Int}.index: Index (0) out of range ((1,10))
```

Проверка 42.1. Создайте массив `qcArray` типа `UArray Int Bool`, содержащий пять элементов, индексируемый с нуля и такой, чтобы его второй и третий элементы были равны `True`.

Ответ 42.1

```
qcArray :: UArray Int Bool
qcArray = array (0,4) [(1,True),(2,True)]
```

42.1.3. Модификация UArray

Получение доступа к элементам массива UArray — это, конечно, чудесно, но вам также требуется возможность их изменять. UArray можно модифицировать точно так же, как и другие функциональные структуры данных, — при помощи создания копии этой структуры с изменённым значением требуемого элемента. Давайте предположим, что у вас есть массив, представляющий количество бобов, разложенных по четырём корзинкам.

Листинг 42.6 Массив UArray, представляющий бобы в корзинках

```
beansInBuckets :: UArray Int Int
beansInBuckets = array (0,3) []
```

Так как передаётся пустой список начальных значений, массив UArray инициализирован нулями:

```
GHCi> beansInBuckets ! 0
0
GHCi> beansInBuckets ! 2
0
```

Проверка 42.2. Не полагаясь на умолчания, явно укажите, что массив следует инициализировать нулями.

Теперь предположим, что вы хотите положить 5 бобов в корзинку номер 1 и 6 — в корзинку номер 3 (корзинка номер 0 — первая в массиве). Вы можете сделать это с помощью операции (//). Первым аргументом этой операции является UArray, а вторым — новый список пар. В результате вернётся новый массив UArray с изменённым значением.

Листинг 42.7 Изменение UArray с помощью операции //

```
updatedBiB :: UArray Int Int
updatedBiB = beansInBuckets // [(1,5),(3,6)]
```

Ответ 42.2

```
beansInBuckets' :: UArray Int Int
beansInBuckets' = array (0,3) $ zip [0 .. 3] $ cycle [0]
```

В GHCi вы можете убедиться, что значения действительно изменились:

```
GHCi> updatedBiB ! 1
5
GHCi> updatedBiB ! 2
0
GHCi> updatedBiB ! 3
6
```

Теперь вам срочно потребовалось добавить по два боба в каждую корзинку. Понятно, что вам может потребоваться часто изменять существующее значение. Чтобы сделать это, вы можете воспользоваться функцией `accum`. Она принимает бинарную функцию, `UArray` и список значений, к которым функция будет применена. Вот пример добавления двух бобов в каждую корзинку:

```
GHCi> accum (+) updatedBiB $ zip [0 .. 3] $ cycle [2]
array (0,3) [(0,2),(1,7),(2,2),(3,8)]
```

Вы решили одну из самых неприятных проблем применения списков для хранения данных. Пользуясь `UArray`, вы можете более эффективно получать доступ к данным, а также гораздо эффективнее их хранить. Но одна вопиющая проблема всё же остаётся. Практически все наиболее эффективные алгоритмы на массивах изменяют их содержимое без создания копий. Даже сортировка пузырьком, которая является одним из наименее производительных алгоритмов сортировки, не требует использования новых массивов. Когда вы изменяете значения в массиве, у вас нет копии этого массива, в которой фиксируются изменения. Но это возможно только благодаря изменению состояния массива. Когда вы использовали `UArray`, вы могли создать иллюзию изменяемого состояния. Для многих примеров предпочтительнее вносить изменения в структуру данных. Но если вы используете состояние исключительно в целях повышения производительности, это ужасное решение.

Проверка 42.3. Увеличьте число бобов в каждой корзинке втрое.

Ответ 42.3

```
accum (*) updatedBiB $ zip [0 .. 3] $ cycle [3]
```

42.2. Изменение состояния с помощью STUArray

ЛАНЬ

Большую часть времени Haskell принуждает программиста избавляться от работы с состоянием, что приводит к более безопасному и предсказуемому коду без потери в производительности. Но это не так для алгоритмов на массивах. Если бы в Haskell не было никакого способа манипулировать состоянием в таких ситуациях, очень большая часть алгоритмов была бы недоступна для реализации в Haskell.

Haskell предлагает решение этой проблемы. Вы будете пользоваться специальной разновидностью неупакованных массивов под названием `STUArray`, которая, в свою очередь, основана на работе в контексте `ST`. Этот контекст допускает изменение состояния и поддерживает энергичные (то есть неленивые) вычисления. В этом уроке вы сфокусируетесь только на `STUArray`, но важно понимать, что решения, используемые здесь для работы с массивами, могут быть перенесены на более широкий круг задач, требующих работы с изменяемым состоянием.

ЛАНЬ

Для использования `STUArray` необходимо импортировать несколько модулей:

```
import Data.Array.ST
import Control.Monad
import Control.Monad.ST
```

Тип `STUArray` — экземпляр `Monad`. В модуле 5 вы потратили довольно много времени, изучая семейство классов типов, включающее в себя `Functor`, `Applicative` и `Monad`. Цель всех этих классов типов — позволить вам совершать произвольные вычисления в контексте. Вы уже наблюдали несколько примеров на страницах книги:

- контекст `Maybe` служит для моделирования отсутствующих значений;
- списки реализуют контекст для недетерминированных вычислений;
- `IO` позволяет разделять операции ввода-вывода и чистые функции;
- `Either` предоставляет более информативный способ обработки ошибок, нежели `Maybe`.

Как и `IO`, `STUArray` позволяет вам выполнять вычисления, которые обычно запрещены в безопасном контексте Haskell. При работе с `STUArray` вы будете использовать do-нотацию, что позволит обращаться с типами в контексте `STUArray` так, будто они являются обычными данными.

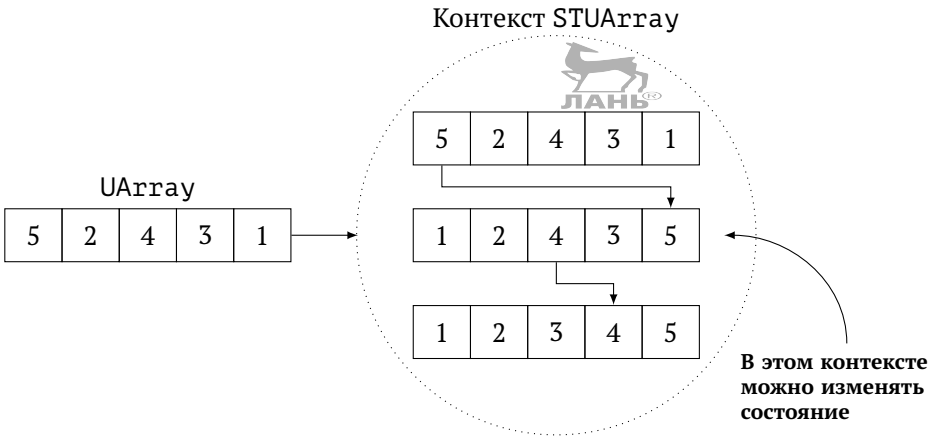


Рис. 42.1: Контекст `STUArray` с возможностью изменять состояние

Ключевым преимуществом, которое предлагает `STUArray`, является способность изменять значения в `UArray` (см. рис. 42.1 с визуальным объяснением). Это как раз то увеличение эффективности, которое поможет добиться производительности наравне с языками программирования, эксплуатирующими изменяемое состояние. Возможность менять значения по месту поможет сэкономить колоссальное количество памяти и времени, так как пропадает необходимость делать копию данных при каждом изменении. Это ключ для реализации настолько эффективной сортировки пузырьком, насколько можно найти в учебнике по алгоритмам.

Для понимания `STUArray`, да и `ST` в общем, важно осознавать, что это не лазейки, которые позволят вам избавиться от функциональной чистоты, к которой вы до этого стремились. `STUArray` существует, чтобы позволить вам работать с состоянием только тогда, когда использование состояния неотличимо от чистого кода для людей, использующих ваши функции. Подавляющее большинство изменений в структурах данных может быть сделано эффективно и безопасно с помощью функционального подхода.

Вы изучите, как использовать `STUArray`, на примере написания функции `listToSTUArray`, которая принимает список `Int` и трансформирует его в `STUArray`. В первой черновой версии `listToSTUArray` будет просто создаваться пустой массив `STUArray`, размер которого равен длине исходного списка. Это очень похоже на инициализацию пустого массива фиксированного размера, только происходить это будет в монаде. `STUArray` использует функцию `newArray`, которая принимает пару, представляющую границы массива, а также значение для инициализации его элементов.

Листинг 42.8 Первый набросок listToSTUArray

```
listToSTUArray :: [Int] -> ST s (STUArray s Int Int)
listToSTUArray vals = do
    let end = length vals - 1
    stArray <- newArray (0,end) 0
    return stArray
```

end — обычная переменная, поэтому пользуемся let

stArray является изменяемым массивом в контексте, полученным с помощью <-

Когда всё готово, нужно вернуть массив обратно в контекст

Далее нужно добавить цикл, который будет обходить список и обновлять значение `stArray`. Для этого воспользуемся функцией `forM_` из модуля `Control.Monad`. Эта функция принимает данные и функцию, использующую данные в качестве аргументов. Это позволяет моделировать поведение цикла `for/in` в таких языках, как, например, Python.

Для демонстрации того, каким боком это похоже на привычный цикл `for`, воспользуемся списком индексов и операцией `(!!)` для просмотра его содержимого. Было бы эффективнее связать эти индексы со значениями в списке с помощью функции `zip`, но сейчас нашей целью является эмуляция работы с языком, использующим состояния. Нам осталось только записать значения из списка в `stArray`. Для этого вы будете использовать функцию `writeArray`, которая принимает `STUArray`, индекс и значение. Эта функция меняет содержимое данного массива, не создавая его копии.

Листинг 42.9 Копирование содержимого списка в STUArray

```
listToSTUArray :: [Int] -> ST s (STUArray s Int Int)
listToSTUArray vals = do
    let end = length vals - 1
    myArray <- newArray (0,end) 0
    forM_ [0 .. end] $ \i -> do
        let val = vals !! i
        writeArray myArray i val
    return myArray
```

Функция forM_ копирует поведение цикла for в других языках

Доступ к элементу списка не влияет на состояние, поэтому присваивание выполняется с помощью let

Функция writeArray переписывает данные в массиве

Цикл `forM_` позволил вам написать код в стиле какого-нибудь языка с поддержкой изменяемого состояния, если бы там потребовалось сделать нечто подобное.

Если вы загрузите этот код в GHCi, то увидите, что он запускается нормально, но есть одна серьёзная проблема:

```
GHCi> listToSTUArray [1,2,3]
<<ST action>>
```

Точно так же, как и при работе с `IO` типами, при использовании `ST` вы получаете программу в контексте. Но, в отличие от `IO`, у вас нет способа вывести результаты обычным образом. К счастью, опять же, в отличие от `IO`, из этого контекста можно извлекать значения.

42.3. Извлечение значений из контекста ST

Когда вы только познакомились с типом `IO`, вы увидели, что он позволяет держать не совсем безопасный IO-код в рамках контекста. Хотя `STUArray` чем-то напоминает `IO`, сейчас вы работаете с более безопасным контекстом. Для начала, даже с учётом манипуляций с состоянием, ссылочная прозрачность не нарушается. Всякий раз при запуске `listToSTUArray` на одинаковых входных данных вы получаете одинаковый результат. Это поднимает важный вопрос о ссылочной прозрачности и инкапсуляции в ОО-языках программирования. Инкапсуляция в ООП означает, что объекты скрывают от пользователя детали реализации. Даже в ООП не особенно поощряется работа с состоянием, если она нарушает инкапсуляцию. Проблема в том, что в ООП нет механизма для контроля корректности инкапсуляции внутреннего состояния. В коде `listToSTUArray`, поскольку работа с состоянием происходит в контексте, вы вынуждены проверять, что все манипуляции с состоянием не нарушают инкапсуляцию. Так как `STUArray` обеспечивает инкапсуляцию, вы не связаны теми же ограничениями, что и `IO`. Вы можете извлечь из `STUArray` значения, используя функцию `runSTUArray` (проиллюстрировано на рис. 42.2).

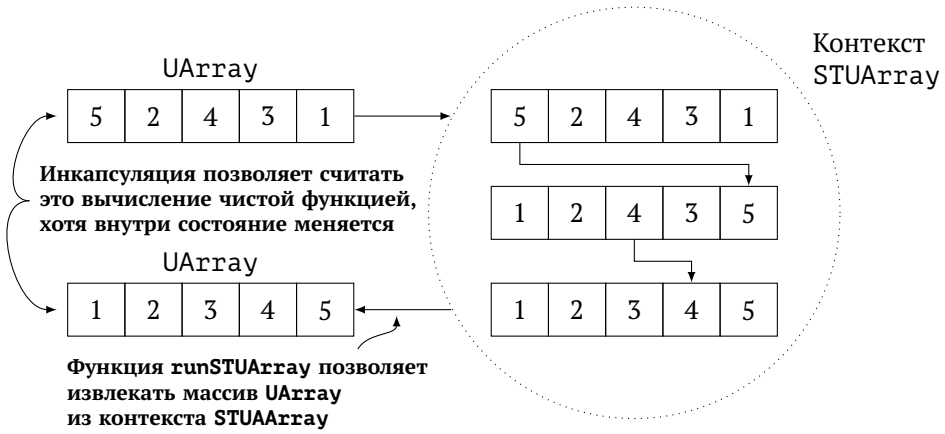


Рис. 42.2: Контекст `STUArray` поддерживает извлечение значений

Вот тип функции runSTUArray:

```
runSTUArray :: ST s (STUArray s i e) -> UArray i e
```

Эта функция позволяет взять лучшее от работы с состоянием и чистого кода. Вы можете держать все манипуляции с состоянием внутри безопасного контекста, но также обращаться с этим кодом как с чистым. Вот новая функция listToUArray, которая работает с состоянием, оставаясь при этом чистой.

Листинг 42.10 Чистая функция listToUArray

```
listToUArray :: [Int] -> UArray Int Int
listToUArray vals = runSTUArray $ listToSTUArray vals
```

Теперь вы можете запустить свою программу в GHCi и получить адекватный результат:

```
GHCi> listToUArray [1,2,3]
array (0,2) [(0,1),(1,2),(2,3)]
```

Важно понимать, что использование runSTUArray не является лазейкой, которая позволит вам протаскивать опасные манипуляции с состоянием в чистые программы. Так как STUArray вынуждает вас поддерживать нерушимую инкапсуляцию, вы можете покинуть контекст STUArray без нарушения каких-либо основных правил обращения с функциями, знакомым вам по уроку 2. Ваш код остаётся безопасным и предсказуемым.

Также стоит подчеркнуть, что Haskell-программисты предпочитают избегать написания промежуточных функций, подобных listToSTUArray. Лучше реализовать listToUArray примерно таким образом.

Листинг 42.11 Обычный метод работы с STUArray и runSTUArray

```
listToUArray :: [Int] -> UArray Int Int
listToUArray vals = runSTUArray $ do
    let end = length vals - 1
    myArray <- newArray (0,end) 0
    forM_ [0 .. end] $ \i -> do
        let val = vals !! i
        writeArray myArray i val
    return myArray
```

Эта реализация listToUArray комбинирует определения двух функций.

Тип ST

Контекст ST расширяет возможности STUArray. Тип STUArray в основном рассчитан на три действия newArray, readArray и writeArray. Для типа ST они меняются на более общие — newSTRef, readSTRef и writeSTRef, а вместо runSTUArray используется runST. Простой пример — функция swapST, меняющая местами значения элементов пары внутри самой пары:

```
swapST :: (Int,Int) -> (Int,Int)
swapST (x,y) = runST $ do
    x' <- newSTRef x
    y' <- newSTRef y
    writeSTRef x' y
    writeSTRef y' x
    xfinal <- readSTRef x'
    yfinal <- readSTRef y'
    return (xfinal,yfinal)
```

Подобно STUArray, основное предназначение ST-типов — позволить вам реализовывать идеально инкапсулированные вычисления с изменяемым состоянием.

42.4. Реализация сортировки методом пузырька

Теперь вы, наконец-то, готовы написать свою собственную версию сортировки пузырьком в Haskell! В случае если вы незнакомы с этим алгоритмом, приведём его основные шаги:

(1) работа алгоритма начинается с первого элемента массива, этот элемент сравнивается со следующим;

(2) если значение первого элемента больше, то сравнивавшиеся элементы меняются местами;

(3) процесс повторяется до тех пор, пока элемент с наибольшим значением не «всплывёт» к концу массива;

(4) действия (1)–(3) следует повторять для $n - 1$ элементов, остающихся в массиве.

Пример работы этого алгоритма продемонстрирован на рис. 42.3.

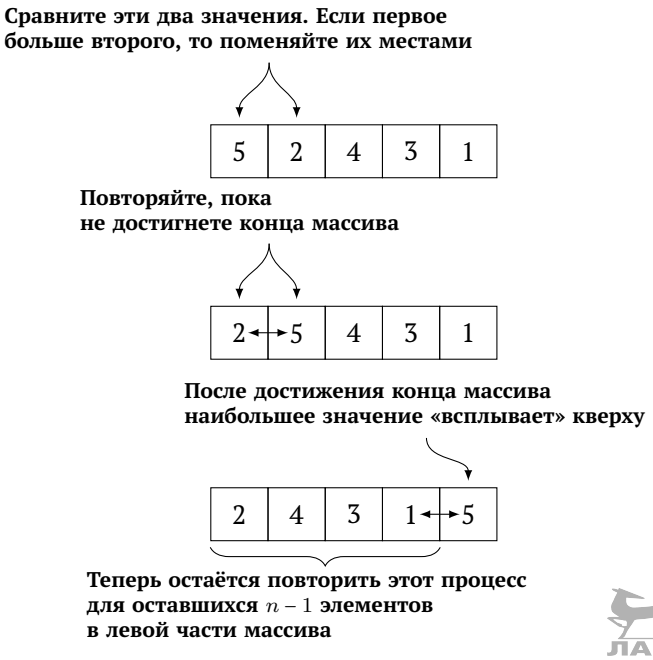


Рис. 42.3: Алгоритм сортировки пузырьком

Начнём с массива типа `UArray`, требующего сортировки. Нужно воспользоваться функцией `listArray`, похожей на написанную ранее функцию `listToUArray`, но дополнительно принимающей на вход пару границ.

Листинг 42.12 Тестовый массив, созданный с помощью listArray

```
myData :: UArray Int Int
myData = listArray (0,5) [7,6,4,8,10,2]
```

Проверка 42.4. Определите `myData`, пользуясь `listToUArray`.

Ответ 42.4

```
myData' :: UArray Int Int
myData' = listToUArray [7,6,4,8,10,2]
```

Для написания функции `bubbleSort`, реализующей метод пузырька, вам понадобится несколько функций. Есть одна вещь, которой вы ещё не занимались, — это использование существующего `UArray` в контексте `STUArray`. Тот факт, что вы работаете в контексте, не означает, что вы можете обращаться с `UArray` так, будто его разрешено изменять. Чтобы сделать это возможным, вам следует воспользоваться функцией `thaw`, которая «разморозит» `UArray`, позволив с ним работать в нужном режиме. Также вы воспользуетесь функцией `bounds`, которая возвращает границы массива, чтобы вы могли посмотреть, где он заканчивается. Контекст `STUArray` поддерживает также специальную функцию `readArray`, которая читает значение элемента массива. Наконец, вы примените полезную функцию `when`, которая работает аналогично инструкции `if` без ветки `else` в других языках программирования. Далее приведён код функции `bubbleSort`.

Листинг 42.13 Реализация сортировки методом пузырька

```
bubbleSort :: UArray Int Int -> UArray Int Int
bubbleSort myArray = runSTUArray $ do
    stArray <- thaw myArray
    let end = (snd . bounds) myArray
    forM_ [1 .. end] $ \i -> do
        forM_ [0 .. (end - i)] $ \j -> do
            val <- readArray stArray j
            nextVal <- readArray stArray (j + 1)
            let outOfOrder = val > nextVal
            when outOfOrder $ do
                writeArray stArray j nextVal
                writeArray stArray (j + 1) val
    return stArray
```

Нужно «разморозить» UArray в STUArray

Конец массива указан вторым элементом пары границ

Здесь readArray используется для получения значения из STUArray

Функция when позволяет выполнять код, только если выполнено условие

Давайте проведём испытания `bubbleSort` на тестовых данных:

```
GHCi> bubbleSort myData
array (0,5) [(0,2),(1,4),(2,6),(3,7),(4,8),(5,10)]
```

Теперь у вас есть императивный алгоритм, реализованный в Haskell. Потрясающей новостью является то, что вам не пришлось отказываться ни от одного из достоинств Haskell, которые вы уже успели полюбить. Вы написали эффективную (настолько, конечно, насколько сортировка пузырьком может быть эффективной) программу. В то же самое время вы получили код с предсказуемым поведением, который не нарушает требование ссылочной прозрачности.

Итоги

ЛАНЬ

В этом уроке нашей целью было научить вас реализации средствами Haskell эффективных алгоритмов, требующих изменения состояния. Сначала вы изучили тип `UArray`, который позволяет использовать в Haskell энергично вычисляемые массивы. Негативным аспектом `UArray` является то, что вам всё равно приходится работать с состоянием так, как и с другими функциональными структурами данных. Затем вы узнали, что `STUArray` позволяет вам выполнять вычисления с изменяемым состоянием, подобно тому, как `IO`-типы позволяют выполнять операции ввода-вывода. При работе с контекстом `STUArray` вам нужно поддерживать надлежащую инкапсуляцию. На практике идеальная инкапсуляция — то же самое, что и ссылочная прозрачность. Именно поэтому вы можете спокойно преобразовывать `STUArray` обратно в обычный массив `UArray`. В итоге это позволяет вам работать с кодом, меняющим состояние, как с чистой функцией, так как они ведут себя одинаковым образом. Давайте посмотрим, как вы это поняли.

ЛАНЬ

Задача 42.1. Одной из наиболее важных операций в реализации генетических алгоритмов является комбинирование двух массивов булевых значений с помощью операции, называемой *скрещивание*. Эта операция принимает на вход пару массивов одинаковых размеров. Затем выбирается точка рассечения, и части, находящиеся по разные стороны от этой точки, меняются местами. Результатом является новая пара массивов. Вот небольшая иллюстрация работы этой операции с использованием списков (1 используется вместо `True`, а 0 — вместо `False`):

```
([1,1,1,1,1],[0,0,0,0,0])
```

Если скрещивание будет осуществлено по индексу 3, то результат будет выглядеть так:

```
([1,1,1,0,0],[0,0,0,1,1])
```

Реализуйте скрещивание так, чтобы результатом операции было значение типа `UArray`, но само скрещивание осуществлялось с помощью `STUArray`.

Задача 42.2. Напишите функцию `replaceZeros`, принимающую на вход набор нулей и других чисел в виде массива типа `UArray Int Int`. Функция должна возвращать массив, в котором все нули заменены на значение –1.

Послесловие

Пожалуй, самым трудным испытанием при написании книги о Haskell является выбор набора тем для изучения. В Haskell впечатляет и в то же время устрашает кажущийся неограниченным круг тем для изучения. К несчастью, невозможно написать книгу о Haskell, не испытывая при этом ощущения, что пришлось пропустить массу интересных вопросов.

Эта книга была написана с целью обеспечить прочный фундамент для понимания Haskell и функционального программирования в целом. Хорошая новость состоит в том, что если вам удалось достичь этой части книги, перед вами открыто множество путей для продолжения вашего путешествия в мир функционального программирования. Даже если на этом вы остановитесь, я уверен, что ваш взгляд на программное обеспечение, программирование и вычисления в целом значительно расширился. Если вы заинтересованы в дальнейшем изучении тем, рассмотренных в этой книге, в этом послесловии я предложу несколько вариантов продолжения вашего пути в зависимости от того, что вас больше интересует.

Углублённое изучение Haskell

Если вам понравился модуль 5, в котором рассматривались `Functor`, `Applicative` и `Monad`, то у меня есть хорошая новость: эти классы — только верхушка айсберга Haskell. Многие другие классы типов и темы в Haskell предлагают похожие уровни интересных абстракций и новые способы рассуждений о программах. Лучшим местом для продолжения их изучения является обширная, написанная в энциклопедическом стиле статья под названием «Typeclassopedia», сейчас она представляет собой раздел Haskellwiki и доступна по адресу `https://wiki.haskell.org/Typeclassopedia`.

Одной из основных целей нашей книги является формирование устойчивого фундамента, необходимого для изучения некоторых более абстрактных классов типов, чем те, которые вы могли бы изучить самостоятельно. «Typeclassopedia» начинается с рассмотрения интересных классов типов, которые были представлены в нашей книге, и затем переходит ко всё более мощным и абстрактным классам типов.

Одной из тем, которые нам не удалось рассмотреть в этой книге, является параллельное и конкурентное программирование в Haskell. Если вам доводилось разрабатывать параллельные алгоритмы на языках вроде C++, то вы знаете, насколько сложно бывает обеспечить эффективную работу с глобальным состоянием при выполнении асинхронных вычислений. Серьёзным преимуществом ориентации Haskell на чистое функциональное программирование является лёгкость распараллеливания кода на Haskell. Великолепная книга одного из ведущих разработчиков компилятора GHC Саймона Марлоу «Parallel and Concurrent Programming in Haskell» (изданная O'Reilly Media в 2013 году и переведённая на русский язык под названием «Параллельное и конкурентное программирование на языке Haskell» издательством «ДМК Пресс» в 2014 году) содержит подробное описание соответствующих методик программирования на Haskell. После прочтения нашей книги материал, содержащийся в книге Марлоу, должен быть вам понятен. Ссылку на её бесплатную электронную версию можно найти на сайте автора: `https://simonmar.github.io/pages/pcph.html`.

Самое большое изменение, произошедшее в Haskell с тех пор, как я сам начал его изучать, — его приближение к «реальному» программированию, то есть к профессиональной разработке программного обеспечения. Число программистов, пишущих на Haskell промышленный код, возрастает. Одним из свидетельств этого явления может считаться относительно недавнее возникновение системы сборки stack. Хорошим местом для начала изучения библиотек для Haskell является страница `https://haskelllang.org/libraries`. Там вы сможете найти информацию об основных библиотеках и средствах, которыми можно воспользоваться при написании программ на Haskell (некоторые из них мы в этой книге затрагивали).

Более мощные, чем в Haskell, системы типов?

Если больше всего в Haskell вас заинтересовала система типов, вы, вероятно, обрадуетесь, узнав, что существуют основанные на Haskell языки, пытающиеся развивать идеи системы типов. Двумя интересными приме-

рами являются Idris и Liquid Haskell. Оба этих языка расширяют систему типов Haskell, предоставляя возможности для использования более детализированных и мощных ограничений на типы при написании программ. Представьте, что компилятор мог бы предупредить вас, что `head` является частично определённой функцией (то есть что она может не сработать на некоторых входных данных) или что у вас была бы возможность определить размер списка в его типе. Обе эти проверки выходят за пределы возможности системы типов Haskell, но возможны в Liquid Haskell и Idris.

Idris — программирование с зависимыми типами

Idris — это язык программирования, позволяющий работать с зависимыми типами. Типы в Idris, как функции в Haskell, являются значениями первого класса, то есть в Idris вы можете производить вычисления с типами. Какие возможности нам это даёт? Одна из проблем, относящихся к `foldl`, в Haskell состоит в том, что хотя `foldl` часто более интуитивно понятна, чем `foldr`, она не работает с бесконечными списками. Это делает `foldl` частично определённой функцией, так как вы не можете проверить, является ли список бесконечным, а бесконечные списки являются значениями, на которых эта функция не определена. Эту проблему можно решить, если гарантировать, что списки будут конечными. Система типов Haskell не предоставляет такой возможности, но зависимые типы в Idris позволяют указать, что список, передаваемый в качестве аргумента, должен быть конечным.

Вы можете узнать больше о языке Idris, посетив его домашнюю страницу (`http://www.idris-lang.org/documentation/`). Помимо этого, издательство Manning опубликовало книгу «Type-Driven Development with Idris», написанную создателем этого языка Эдвином Брэйди (2017).

Liquid Haskell — доказуемые типы

Liquid Haskell расширяет систему типов Haskell, позволяя вам дополнительно к типам указывать логические предикаты, которые специфицируют желаемое поведение программ. Типы с предикатами называются *уточнёнными*. Как и в Idris, система типов в Liquid Haskell позволяет устранить частично определённые функции с помощью системы типов. Предполагаемые в программе ограничения проверяются во время компиляции. Например, Liquid Haskell позволяет на уровне типов запретить деление на ноль. Здорово в этом то, что ошибки, связанные с делением на ноль, можно *выявить во время компиляции*. Лучшим местом для того, чтобы узнать больше о Liquid Haskell, является домашняя страница проекта (`https://ucsd-progsys.github.io/liquidhaskell-blog/`).

Другие функциональные языки программирования

Возможно, после прочтения этой книги у вас появилась любовь к функциональному программированию, но вы не уверены, что вам подходит именно Haskell. Существует много развитых и мощных функциональных языков программирования, которые вы можете изучить. Haskell определённо является самым чистым из них, но иногда это может быть минусом.

Языки функционального программирования можно разделить на два больших семейства — семейство Lisp и семейство ML. Haskell является хорошим примером функционального языка программирования из семейства ML. Большинство этих языков имеет похожую на Haskell систему типов (хотя, разумеется, каждая система типов имеет свои особенности). Языки семейства Lisp обычно являются динамически типизированными, а для их синтаксиса характерно активное использование скобок и префиксных операций. Если вы интересуетесь функциональным программированием, я настоятельно рекомендую изучить как семейство языков ML, так и семейство Lisp. Хотя у них и много общего, они предлагают различные подходы к программированию.

Рекомендуемые языки программирования из семейства Lisp

Самой большой неожиданностью для всех новичков в языках семейства Lisp является обилие скобок. В Lisp все программы представляются как деревья вычислений, а вложенные скобки являются хорошим способом представления таких деревьев. Структура дерева предоставляет возможности для сложных манипуляций с программами в качестве данных. Отличительным признаком многих Lisp-языков являются макросы, позволяющие генерировать код во время компиляции. Это позволяет программистам на Lisp при необходимости с лёгкостью определять собственный синтаксис. Создание своего предметно-ориентированного языка (domain-specific language, DSL) в Lisp зачастую может быть реализовано буквально несколькими строками кода. Ниже представлено несколько хороших вариантов для более глубокого изучения Lisp.

Racket

Язык программирования Racket является потомком длинной цепочки диалектов Lisp под общим названием Scheme. Он, вероятно, является самым чистым современным представителем семейства Lisp и имеет отличную поддержку сообщества. Как и Haskell, Racket имеет относительно небольшое коммерческое применение сравнительно с сообществом лю-

дей, использующих его в качестве средства исследования теории языков программирования. Несмотря на академические наклонности, сообщество Racket проделало отличную работу по облегчению изучения этого языка новичками. Вы можете узнать больше о языке Racket на его официальном сайте: `https://racket-lang.org/`.

Clojure

Язык программирования Clojure на данный момент является наиболее активно используемым в промышленности Lisp-языком. Clojure использует виртуальную машину Java (JVM) и, таким образом, имеет доступ ко всем Java-библиотекам. Большое сообщество прагматично мыслящих, профессиональных разработчиков программного обеспечения использует Clojure. Если вас интересует Lisp, но вашей основной целью является разработка реального кода, вам понравится сообщество Clojure. Вы сможете найти больше информации на сайте Clojure: `https://clojure.org/`.

Common Lisp

Хотя Common Lisp к настоящему времени несколько устарел, он является одним из самых мощных языков программирования из когда-либо созданных. Common Lisp является языком, ориентированным на абстракции, насколько это вообще возможно, и, по моему мнению, самым выразительным языком программирования из существующих сейчас. Серьёзным минусом Common Lisp является то, что сейчас его довольно трудно использовать для разработки практических приложений. Если вы изучите этот язык достаточно глубоко, то безнадёжно влюбитесь в него. Отличным введением в язык является книга Питера Сайбеля «Practical Common Lisp» (издательство Apress, 2005 г.), её бесплатно можно найти в сети: `www.gigamonkeys.com/book/`. Перевод этой книги издан «ДМК Пресс» в 2014 г. под названием «Практическое использование Common Lisp».

Рекомендуемые языки программирования семейства ML

Язык Haskell принадлежит к семейству функциональных языков программирования ML. Главной отличительной чертой ML-языков являются их мощные системы типов. Haskell по распространённому мнению считается самым сложным из семейства ML из-за совмещения ленивых вычислений, обеспечивающих чистое функциональное программирование, и абстрактных концепций, таких как монады. Если вам понравилась большая часть изученного в этой книге, но вы считаете Haskell слишком сложным, вы наверняка сможете найти свой любимый язык программирования среди других языков семейства ML. Семейство ML включает множество акаде-

мических (в той же мере, что и Haskell) языков программирования, таких как Miranda и Standard ML. Я решил опустить их рассмотрение, сосредоточившись на более практичных альтернативах.

F#

Язык программирования F# является реализацией другого варианта ML (OCaml) для среды разработки Microsoft .NET. F# является мультипарадигменным языком программирования с хорошей поддержкой функционального и объектно-ориентированного программирования. Если вы .NET-разработчик и используете языки вроде C#, то, вероятно, сочтёте F# отличным средством, предоставляющим множество из понравившихся вам в Haskell возможностей в экосистеме .NET. Благодаря поддержке Microsoft у F# есть отличная документация и широкий выбор существующих библиотек и приспособлений, позволяющих разработчикам выполнять множество практических задач. Вы можете узнать больше на домашней странице F#: `http://fsharp.org/`.

Scala

Как и F#, Scala совмещает развитую систему типов, функциональное и объектно-ориентированное программирование. Scala работает поверх JVM и, как и Clojure, может использовать множество Java-библиотек. Scala является довольно гибким языком, позволяющим вам написать что угодно, от традиционного, но менее многословного, чем в Java, кода до сложного кода с использованием монад и функторов. Scala имеет отличное сообщество разработчиков и, вероятно, является лучшим вариантом, если вы хотите заниматься функциональным программированием на работе. Средства и ресурсы, доступные в Scala, не уступают другим используемым в индустрии языкам программирования. Вы можете узнать больше о Scala на сайте `www.scala-lang.org`.

Elm и PureScript

Elm (`http://elm-lang.org`) и PureScript (`www.purescript.org`) — это языки программирования, служащие одной цели — создать язык, напоминающий Haskell и компилирующийся в JavaScript. Язык программирования Elm фокусируется на создании пользовательских интерфейсов на JavaScript с помощью функционального программирования. Сайт Elm содержит множество отличных примеров для начала изучения. PureScript (не путать с TypeScript) фокусируется на создании языка, похожего на Haskell, компилирующегося в JavaScript. Синтаксически PureScript похож на Haskell, и после прочтения этой книги переход на него не должен составить проблемы.

Примерные решения задач

В программировании прекрасно то, что пока вы получаете верный результат, неправильного решения не существует. Приводимые здесь решения задач должны рассматриваться как одни из возможных. Существует много путей к верному решению, и если ваше отличается, но даёт верный результат, то оно правильное.

Модуль 1

Урок 2

Задача 2.2

```
inc x = x + 1
double x = x*2
square x = x^2
```

Задача 2.3

```
-- Версия с |even|:
ifEven2 n = if even n
            then n - 2
            else 3 * n + 1

-- Версия с |mod|:
ifEven1 n = if n `mod` 2 == 0
            then n - 2
            else 3 * n + 1
```

Урок 3

Задача 3.1

```
simple = (\x -> x)
makeChange = (\owed given ->
                if given - owed > 0
                then given - owed
                else 0)
```

Задача 3.2

```
inc = (\x -> x+1)
double = (\x -> x*2)
square = (\x -> x^2)

counter x = (\x -> x + 1)
            ((\x -> x + 1)
            ((\x -> x) x))
```

Урок 4

Задача 4.1

Если фамилии совпадают, то следует проверить имена:

```
compareLastNames name1 name2 =
    if result == EQ
    then compare (fst name1) (fst name2)
    else result
  where result = compare (snd name1) (snd name2)
```

Задача 4.2

Добавляем новый офис в округе Колумбия:

```
dcOffice name = nameText ++
                " - А/я 1337, Вашингтон, округ Колумбия, 20001"
  where nameText = "Многоуважаемый(ая) " ++
                   (fst name) ++ " " ++ (snd name)

getLocationFunction location = case location of
  "ny" -> nyOffice
  "sf" -> sfOffice
  "reno" -> renoOffice
  "dc" -> dcOffice
  _ -> (\name -> (fst name) ++ " " ++ (snd name))
```

Урок 5

Задача 5.1

```
ifEven myFunction x = if even x
                      then myFunction x
                      else x

inc n = n + 1
double n = n*2
square n = n^2

ifEvenInc = ifEven inc
ifEvenDouble = ifEven double
ifEvenSquare = ifEven square
```

Задача 5.2

```
binaryPartialApplication binaryFunc arg =
                                (\x -> binaryFunc arg x)
```

Вот пример:

```
takeFromFour = binaryPartialApplication (-) 4
```

Урок 6

Задача 6.1

```
repeat n = cycle [n]
```

Задача 6.2

```
subseq start end myList = take difference (drop start myList)
  where difference = end - start
```

Задача 6.3

```
inFirstHalf val myList = val `elem` firstHalf
  where midpoint = (length myList) `div` 2
        firstHalf = take midpoint myList
```

Урок 7

Задача 7.1

```
myTail [] = []
myTail (_:xs) = xs
```

Задача 7.2

```
myGCD a 0 = a
myGCD a b = myGCD b (a `mod` b)
```

Урок 8

Задача 8.1

```
myReverse [] = []
myReverse (x:[]) = [x]
myReverse (x:xs) = (myReverse xs) ++ [x]
```

Задача 8.2

```
fastFib _ _ 0 = 0
fastFib _ _ 1 = 1
fastFib _ _ 2 = 1
fastFib x y 3 = x + y
fastFib x y c = fastFib (x + y) x (c - 1)
```

Обратите внимание, что вы можете использовать вспомогательную функцию, чтобы скрыть тот факт, что всегда начинаете с 1 1:

```
fib n = fastFib 1 1 n
```

Урок 9

Задача 9.1

```
myElem val myList = (length filteredList) /= 0
  where filteredList = filter (== val) myList
```

Задача 9.2

```
isPalindrome text = processedText == reverse processedText
   where noSpaces = filter (/= ' ') text
         processedText = map toLower noSpaces
```

Задача 9.3

```
harmonic n = sum (take n seriesValues)
  where seriesPairs = zip (cycle [1.0])  [1.0,2.0 .. ]
        seriesValues = map (\pair -> (fst pair)/(snd pair))
                           seriesPairs
```

Модуль 2

Урок 11

Задача 11.1

```
filter :: (a -> Bool) -> [a] -> [a]
```

Если вы взглянете на `map`, то увидите два отличия:

```
map :: (a -> b) -> [a] -> [b]
```

Первое заключается в том, что функция, переданная в `filter`, должна возвращать `Bool`. Второе — в том, что `map` может изменить тип списка, а `filter` — нет.

Задача 11.2

```
safeTail :: [a] -> [a]
safeTail [] = []
safeTail (x:xs) = xs
```

Вы не можете сделать то же самое для `head`, так как нет вменяемого значения по умолчанию для отсутствующего элемента. Вы не можете возвращать пустой список, потому что пустой список имеет тип списка, а вам нужен элемент. См. урок 37 для более подробного рассмотрения этой темы.

Задача 11.3

```
myFoldl :: (a -> b -> a) -> a -> [b] -> a
myFoldl f init [] = init
myFoldl f init (x:xs) = myFoldl f newInit xs
  where newInit = f init x
```

Урок 12

Задача 12.1

Вы можете упростить решение, воспользовавшись `canDonateTo`:

```
donorFor :: Patient -> Patient -> Bool
donorFor p1 p2 = canDonateTo (bloodType p1) (bloodType p2)
```

Задача 12.2

Добавьте вспомогательную функцию для отображения пола:

```
showSex Male = "мужской"
showSex Female = "женский"

patientSummary :: Patient -> String
patientSummary patient =
   "**************\n" ++
   "Пол: " ++ showSex (sex patient) ++ "\n" ++
   "Возраст: " ++ show (age patient) ++ "\n" ++
   "Рост: " ++ show (height patient) ++ "см\n" ++
   "Вес: " ++ show (weight patient) ++ "кг.\n" ++
   "Тип крови: " ++ showBloodType (bloodType patient)
   ++ "\n**************\n"
```

Урок 13

Задача 13.1

Если вы посмотрите на классы типов, к которым принадлежат эти типы, то получите о них хорошее представление.

Для `Word`:

```
instance Bounded Word
instance Enum Word
instance Eq Word
instance Integral Word
instance Num Word
instance Ord Word
instance Read Word
instance Real Word
instance Show Word
```

Для `Int`:

```
instance Bounded Int
instance Enum Int
instance Eq Int
instance Integral Int
instance Num Int
instance Ord Int
instance Read Int
instance Real Int
instance Show Int
```

Вы можете увидеть, что они разделяют одни и те же классы типов. Правильной догадкой будет то, что `Word` имеет отличные от `Int` границы. Если вы взглянете на `maxBound`, то увидите, что `Word` больше, чем `Int`:

```
GHCi> maxBound :: Word
18446744073709551615
GHCi> maxBound :: Int
9223372036854775807
```

Но в то же время `Word` имеет нижнюю границу 0, в то время как `Int` гораздо ниже:

```
GHCi> minBound :: Word
0
GHCi> minBound :: Int
-9223372036854775808
```

Как вы уже могли догадаться, `Word` — это `Int`, который принимает только положительные значения — беззнаковый `Int`.

Задача 13.2

Вы можете увидеть разницу, использовав `inc` и `succ` на значении `maxBound` типа `Int`:

```
GHCi> inc maxBound :: Int
-9223372036854775808
GHCi> succ maxBound :: Int
*** Exception: Prelude.Enum.succ{Int}: tried to take ‘succ’
                                                  ↳ of maxBound
```

Так как нет настоящей функции следования для типа `Bounded`, `succ` выдаёт ошибку. Функция `inc` возвращает вас к наименьшему значению типа.

Задача 13.3

```
cycleSucc :: (Bounded a, Enum a, Eq a) => a -> a
cycleSucc n = if n == maxBound
              then minBound
              else succ n
```

Урок 14

Задача 14.1

Допустим, у вас есть тип данных вроде этого:

```
data Number = One | Two | Three deriving Enum
```

Теперь вы можете использовать `fromEnum` для перевода значения в `Int`. Это упрощает реализацию и `Eq`, и `Ord`:

```
instance Eq Number where
  (==) num1 num2 = (fromEnum num1) == (fromEnum num2)

instance Ord Number where
  compare num1 num2 = compare (fromEnum num1) (fromEnum num2)
```

Задача 14.2

```
data FiveSidedDie = Side1 | Side2 | Side3 | Side4 | Side5
  deriving (Enum, Eq, Show)

class (Eq a, Enum a) => Die a where
  roll :: Int -> a

instance Die FiveSidedDie where
  roll n = toEnum (n `mod` 5)
```

Модуль 3

Урок 16

Задача 16.1

```
data Pamphlet = Pamphlet {
  pamphletTitle :: String,
  description :: String,
  contact :: String
}

data StoreItem = BookItem Book
               | RecordItem VinylRecord
               | ToyItem CollectibleToy
               | PamphletItem Pamphlet
```

Теперь вам нужно дополнить определение функции `price`:

```
price :: StoreItem -> Double
price (BookItem book) = bookPrice book
price (RecordItem record) = recordPrice record
price (ToyItem toy) = toyPrice toy
price (PamphletItem _) = 0.0
```

Задача 16.2

```
type Radius = Double
type Height = Double
type Width = Double

data Shape = Circle Radius
           | Square Height
           | Rectangle Height Width deriving Show

perimeter :: Shape -> Double
perimeter (Circle r) = 2*pi*r
perimeter (Square h) = 4*h
perimeter (Rectangle h w) = 2*h + 2*w

area :: Shape -> Double
area (Circle r) = pi*r^2
area (Square h) = h^2
area (Rectangle h w) = h*w
```

Урок 17

Задача 17.1

```
data Color = Red | Yellow | Blue | Green | Purple | Orange
           | Brown | Clear deriving (Show,Eq)

instance Semigroup Color where
   (<>) Clear any = any
   (<>) any Clear = any
   (<>) Red Blue = Purple
   (<>) Blue Red = Purple
   (<>) Yellow Blue = Green
   (<>) Blue Yellow = Green
   (<>) Yellow Red = Orange
   (<>) Red Yellow = Orange
   (<>) a b | a == b = a
            | all (`elem` [Red,Blue,Purple]) [a,b] = Purple
            | all (`elem` [Blue,Yellow,Green]) [a,b] = Green
            | all (`elem` [Red,Yellow,Orange]) [a,b] = Orange
            | otherwise = Brown

instance Monoid Color where
   mempty = Clear
   mappend col1 col2 = col1 <> col2
```

Задача 17.2

```
data  Events = Events [String]
data  Probs = Probs [Double]

combineEvents :: Events -> Events -> Events
combineEvents (Events e1) (Events e2) =
    Events (cartCombine combiner e1 e2)
  where combiner = (\x y -> mconcat [x,"-",y])

instance Semigroup Events where
  (<>) = combineEvents

instance Monoid Events where
  mappend = (<>)
  mempty = Events []

combineProbs :: Probs -> Probs -> Probs
combineProbs (Probs p1) (Probs p2) =
```

```
    Probs (cartCombine (*) p1 p2)

instance Semigroup Probs where
  (<>) = combineProbs

instance Monoid Probs where
  mappend = (<>)
  mempty = Probs []
```

Урок 18

Задача 18.1

```
boxMap :: (a -> b) -> Box a -> Box b
boxMap func (Box val) = Box (func val)

tripleMap :: (a -> b) -> Triple a -> Triple b
tripleMap func (Triple v1 v2 v3) =
    Triple (func v1) (func v2) (func v3)
```

Задача 18.2

Тип `Organ`, чтобы быть ключом для `Map`, должен принадлежать классу `Ord`. Добавьте `Enum`, чтобы легко строить список всех органов.

```
data Organ = Heart | Brain | Kidney | Spleen
  deriving (Show, Eq, Ord, Enum)

values :: [Organ]
values = map snd (Map.toList organCatalog)
```

Теперь у вас есть список всех органов:

```
allOrgans :: [Organ]
allOrgans = [Heart .. Spleen]
```

Посчитаем их:

```
organCounts :: [Int]
organCounts = map countOrgan allOrgans
  where countOrgan = (\organ ->
                        (length . filter (== organ)) values)
```

И сформируем `Map`:

```
organInventory :: Map.Map Organ Int
organInventory = Map.fromList (zip allOrgans organCounts)
```

Урок 19

Задача 19.1

```
data Organ = Heart | Brain | Kidney | Spleen
  deriving (Show, Eq)

sampleResults :: [Maybe Organ]
sampleResults = [(Just Brain),Nothing,Nothing,(Just Spleen)]

emptyDrawers :: [Maybe Organ] -> Int
emptyDrawers contents = (length . filter isNothing) contents
```

Задача 19.2

```
maybeMap :: (a -> b) -> Maybe a -> Maybe b
maybeMap func Nothing = Nothing
maybeMap func (Just val) = Just (func val)
```

Модуль 4

Урок 21

Задача 21.1

```
helloPerson :: String -> String
helloPerson name = "Привет" ++ " " ++ name ++ "!"

sampleMap :: Map.Map Int String
sampleMap = Map.fromList [(1,"Уилл")]

mainMaybe :: Maybe String
mainMaybe = do
   name <- Map.lookup 1 sampleMap
   let statement = helloPerson name
   return statement
```

Задача 21.2

```
fib 0 = 0
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n - 2)
```

```
main :: IO ()
main = do
  putStrLn "Введите число:"
  number <- getLine
  let value = fib (read number)
  putStrLn (show value)
```

Урок 22

Задача 22.1

Ленивый ввод-вывод позволяет вам обращаться с вводом как списком:

```
sampleInput :: [String]
sampleInput = ["21","+","123"]
```

Эта функция неидеальна, но цель в том, чтобы познакомиться с ленивым вводом-выводом:

```
calc :: [String] -> Int
calc (val1:"+":val2:rest) = read val1 + read val2
calc (val1:"*":val2:rest) = read val1 * read val2

main :: IO ()
main = do
  userInput <- getContents
  let values = lines userInput
  print (calc values)
```

Задача 22.2

```
quotes :: [String]
quotes = ["quote 1", "quote 2", "quote 3", "quote 4",
          "quote 5"]

lookupQuote :: [String] -> [String]
lookupQuote [] = []
lookupQuote ("n":xs) = []
lookupQuote (x:xs) = quote : (lookupQuote xs)
  where quote = quotes !! (read x - 1)

main :: IO ()
main = do
  userInput <- getContents
  mapM_ putStrLn (lookupQuote  (lines userInput))
```

Урок 23

Задача 23. 1

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text as T
import qualified Data.Text.IO as TIO

helloPerson :: T.Text -> T.Text
helloPerson name = mconcat [ "Привет, ", name, "!"]

main :: IO ()
main = do
  TIO.putStrLn "Привет! Как тебя зовут?"
  name <- TIO.getLine
  let statement = helloPerson name
  TIO.putStrLn statement
```

Задача 23.2

```
import qualified Data.Text.Lazy as T
import qualified Data.Text.Lazy.IO as TIO

toInts :: T.Text -> [Int]
toInts = map (read . T.unpack) . T.lines

main :: IO ()
main = do
  userInput <- TIO.getContents
  let numbers = toInts userInput
  TIO.putStrLn ((T.pack . show . sum) numbers)
```

Урок 24

Задача 24.1

```
import System.IO
import System.Environment
import qualified Data.Text as T
import qualified Data.Text.IO as TI

main :: IO ()
main = do
  args <- getArgs
```

```
  let source =  args !! 0
  let dest = args !! 1
  input <- TI.readFile source
  TI.writeFile dest input
```

Задача 24.2

```
import System.IO
import System.Environment
import qualified Data.Text as T
import qualified Data.Text.IO as TI

main :: IO ()
main = do
  args <- getArgs
  let fileName = head args
  input <- TI.readFile fileName
  TI.writeFile fileName (T.toUpper input)
```

Урок 25

Задача 25.1

```
import System.IO
import System.Environment
import qualified Data.Text as T
import qualified Data.ByteString as B
import qualified Data.Text.Encoding as E

main :: IO ()
main = do
  args <- getArgs
  let source =  args !! 0
  input <- B.readFile source
  putStrLn "Bytes:"
  print (B.length input)
  putStrLn "Characters:"
  print ((T.length . E.decodeUtf8) input)
```

Задача 25.2

```
reverseSection :: Int -> Int -> BC.ByteString -> BC.ByteString
reverseSection start size bytes = mconcat [ before
                                          , changed
```

```
                                                    , after]
  where (before,rest) = BC.splitAt start bytes
        (target,after) = BC.splitAt size rest
        changed =  BC.reverse target

randomReverseBytes :: BC.ByteString -> IO BC.ByteString
randomReverseBytes bytes = do
  let sectionSize = 25
  let bytesLength = BC.length bytes
  start <- randomRIO (0,(bytesLength - sectionSize))
  return (reverseSection start sectionSize bytes)
```

Модуль 5

Урок 27

Задача 27.1

```
data Box a = Box a deriving Show

instance Functor Box where
   fmap func (Box val)  = Box (func val)
```

Задача 27.2

```
myBox :: Box Int
myBox = Box 1

unwrap :: Box a -> a
unwrap (Box val) = val
```

Задача 27.3

```
printCost :: Maybe Double -> IO()
printCost Nothing = putStrLn "Компонент не найден"
printCost (Just cost)= print cost

main :: IO ()
main = do
  putStrLn "Введите идентификатор компонента:"
  partNo <- getLine
  let part = Map.lookup (read partNo) partsDB
  printCost (cost <$> part)
```

Урок 28

Задача 28.1

В отличие от `haversineMaybe`, вы не можете использовать сопоставление с образцом для значений, так что вам придётся использовать знакомую запись с `do`, раз уж применять `<*>` запретили.

```
haversineIO :: IO LatLong -> IO LatLong -> IO Double
haversineIO ioVal1 ioVal2 = do
   val1 <- ioVal1
   val2 <- ioVal2
   let dist = haversine val1 val2
   return dist
```

Задача 28.2

```
haversineIO :: IO LatLong -> IO LatLong -> IO Double
haversineIO ioVal1 ioVal2 =  haversine <$> ioVal1 <*> ioVal2
```

Задача 28.3

```
printCost :: Maybe Double -> IO()
printCost Nothing = putStrLn "Компонент не найден"
printCost (Just cost)= print cost

main :: IO ()
main = do
  putStrLn "Введите компонент №1:"
  partNo1 <- getLine
  putStrLn "Введите компонент №2:"
  partNo2 <- getLine
  let part1 = Map.lookup (read partNo1) partsDB
  let part2 = Map.lookup (read partNo2) partsDB
  let cheapest = min <$> (cost <$> part1) <*> (cost <$> part2)
  printCost cheapest
```

Урок 29

Задача 29.1

```
allFmap :: Applicative f => (a -> b) -> f a -> f b
allFmap func app = (pure func) <*> app
```

Задача 29.2

```
example :: Int
example = (*) ((+) 2 4) 6

exampleMaybe :: Maybe Int
exampleMaybe = pure (*) <*> (pure (+) <*> pure 2 <*> pure 4)
                        <*> pure 6
```

Задача 29.3

```
startingBeer :: [Int]
startingBeer = [6,12]

remainingBeer :: [Int]
remainingBeer = (\count -> count - 4) <$> startingBeer

guests :: [Int]
guests = [2,3]

totalPeople :: [Int]
totalPeople = (+ 2) <$> guests

beersPerGuest :: [Int]
beersPerGuest = [3,4]

totalBeersNeeded :: [Int]
totalBeersNeeded = (pure (*)) <*>  beersPerGuest
                              <*> totalPeople

beersToPurchase :: [Int]
beersToPurchase = (pure (-)) <*> totalBeersNeeded
                             <*> remainingBeer
```

Урок 30

Задача 30.1

```
allFmapM :: Monad m => (a -> b) -> m a -> m b
allFmapM func val = val >>= (\x -> return (func x))
```

Задача 30.2

```
allApp :: Monad m => m (a -> b) -> m a -> m b
allApp func val = func >>= (\f -> val >>= (\x -> return (f x)))
```

Задача 30.3

```
bind :: Maybe a -> (a -> Maybe b) -> Maybe b
bind Nothing _ = Nothing
bind (Just val) func = func val
```

Урок 31

Задача 31.1

Теперь, когда вы сделали это один раз, вы никогда не забудете, как полезна запись с do!

```
main :: IO ()
main = putStrLn "Каков размер пиццы 1" >>
       getLine >>=
       (\size1 ->
         putStrLn "Сколько стоит пицца 1" >>
         getLine >>=
         (\cost1 ->
            putStrLn "Каков размер пиццы 2" >>
            getLine >>=
            (\size2 ->
              putStrLn "Сколько стоит пицца 2" >>
                getLine >>=
                (\cost2 ->
                  (\pizza1 ->
                    (\pizza2 ->
                      (\betterPizza ->
                        putStrLn (describePizza betterPizza)
                      ) (comparePizzas pizza1 pizza2)
                    )(read size2,read cost2)
                )(read size1, read cost1)
             ))))
```

Задача 31.2

```
listMain :: [String]
listMain = do
  cost1 <- [12.0,15.0,20.0]
  size2 <- [10,11,18]
  cost2 <- [13.0,14.0,21.0]
  let pizza1 = (size1,cost1)
  let pizza2 = (size2,cost2)
  let betterPizza = comparePizzas pizza1 pizza2
  return (describePizza betterPizza)
```

Задача 31.3

```
monadMain :: Monad m => m Double -> m Double
          -> m Double -> m Double -> m String
monadMain s1 c1 s2 c2 = do
   size1 <- s1
   cost1 <- c1
   size2 <- s2
   cost2 <- c2
   let pizza1 = (size1,cost1)
   let pizza2 = (size2,cost2)
   let betterPizza = comparePizzas pizza1 pizza2
   return (describePizza betterPizza)
```

Урок 32

Задача 32.1

```
monthEnds :: [Int]
monthEnds = [31,28,31,30,31,30,31,31,30,31,30,31]

dates :: [Int] -> [Int]
dates ends = [date | end <- ends, date <- [1 .. end ] ]
```

Задача 32.2

```
datesDo :: [Int] -> [Int]
datesDo ends = do
   end <- ends
   date <- [1 .. end]
   return date

datesMonad :: [Int] -> [Int]
datesMonad ends  =  ends >>=
(\end ->
  [1 .. end] >>=
     (\date -> return date))
```

Модуль 6

Задачи в модуле 6 состоят из переработки кода на несколько файлов. Результат занимает слишком много места, а задачи требуют не столько правильности, сколько последовательного повторения шагов, описанных в каждом из уроков.

Модуль 7

Урок 38

Задача 38.1

Начнём с вспомогательной функции:

```
allDigits :: String -> Bool
allDigits val = all (== True) (map isDigit val)

addStrInts :: String -> String -> Either Int String
addStrInts val1 val2
  | allDigits val1 && allDigits val2 =
      Left (read val1 + read val2)
  | not (allDigits val1 || allDigits val2) =
      Right "оба аргумента неверны"
  | not (allDigits val1) =
      Right "первый аргумент неверен"
  | otherwise = Right "второй аргумент неверен"
```

Задача 38.2

```
safeSucc :: (Enum a, Bounded a, Eq a) => a -> Maybe a
safeSucc n = if n == maxBound
             then Nothing
             else Just (succ n)

safeTail :: [a] -> [a]
safeTail [] = []
safeTail (x:xs) = xs

safeLast :: [a] -> Either a String
safeLast [] = Right "пустой список"
safeLast xs =  safeLast' 10000 xs
```

Вы знаете, что пустой список невозможен, так как только `safeLast` вызывает эту функцию, а она уже проверяет пустоту списка:

```
safeLast' :: Int -> [a] -> Either a String
safeLast' 0 _ = Right "Список превышает безопасную границу"
safeLast' _ (x:[]) = Left x
safeLast' n (x:xs) = safeLast' (n - 1) xs
```

Урок 39

Задача 39.1

```
buildRequestNOSSL :: BC.ByteString -> BC.ByteString
                  -> BC.ByteString -> BC.ByteString
                  -> Request
buildRequestNOSSL token host method path =
   setRequestMethod method
   $ setRequestHost host
   $ setRequestHeader "token" [token]
   $ setRequestSecure False
   $ setRequestPort 80
   $ setRequestPath path
   $ defaultRequest
```

Задача 39.2

Обратите внимание, что вам нужно добавить `http-types` в ваш проект и импортировать `Network.HTTP.Types.Status`:

```
main :: IO ()
main = do
  response <- httpLBS request
  let status = getResponseStatusCode response
  if status == 200
     then do
          putStrLn "Сохраняем результат запроса в файл"
          let jsonBody = getResponseBody response
          L.writeFile "data.json" jsonBody
     else print $ statusMessage $ getResponseStatus response
```

Урок 40

Задача 40.1

```
instance ToJSON NOAAResult where
   toJSON (NOAAResult uid mindate maxdate name datacoverage
                      resultId) =
          object ["uid" .= uid, "mindate" .= mindate
                 ,"maxdate" .= maxdate, "name" .= name
                 , "datacoverage" .= datacoverage
                 ,"id" .= resultId]

instance ToJSON Resultset
```

```
instance ToJSON Metadata
instance ToJSON NOAAResponse
```

Задача 40.2

```
data IntList = EmptyList | Cons Int IntList
  deriving (Show,Generic)

instance ToJSON IntList
instance FromJSON IntList
```

Урок 41

Задача 41.1

```
addTool :: String -> String -> IO ()
addTool toolName toolDesc =
  withConn "tools.db" $
    \conn -> do
      execute conn
              (mconcat ["INSERT INTO tools,"(name,description "
                       ,",timesBorrowed)","VALUES (?,?,?)"])
              (toolName,toolDesc,(0 :: Int))
      putStrLn "инструмент добавлен"
```

Задача 41.2

```
promptAndAddTool :: IO ()
promptAndAddTool = do
   putStrLn "Введите название инструмента:"
   toolName <- getLine
   putStrLn "Введите описание инструмента:"
   toolDesc <- getLine
   addTool toolName toolDesc

performCommand :: String -> IO ()
performCommand "users" = printUsers >> main
performCommand "tools" = printTools >> main
performCommand "adduser" = promptAndAddUser >> main
performCommand "checkout" = promptAndCheckout >> main
performCommand "checkin" = promptAndCheckin >> main
performCommand "in" = printAvailable >> main
performCommand "out" = printCheckedout >> main
performCommand "quit" = print "Чао!"
performCommand "addtool" = promptAndAddTool >> main
performCommand _ = putStrLn "Команда не найдена" >> main
```

Урок 42

Задача 42.1

```
crossOver :: (UArray Int Int ,UArray Int Int)
             -> Int -> UArray Int Int
crossOver (a1,a2) crossOverPt = runSTUArray $ do
  st1 <- thaw a1
  let end = (snd . bounds) a1
  forM_ [crossOverPt .. end] $ \i -> do
    writeArray st1 i $ a2 ! i
  return st1
```

Задача 42.2

```
replaceZeros :: UArray Int Int -> UArray Int Int
replaceZeros array = runSTUArray $ do
  starray <- thaw array
  let end = (snd . bounds) array
  let count = 0
  forM_ [0 .. end] $ \i -> do
    val <- readArray starray i
    when (val == 0) $ do
      writeArray starray i (-1)
  return starray
```

Предметный указатель

' (одинарная кавычка), 81
(!!) операция, 84–85, 586
(!) операция, 586
(*) операция, 60, 76
(+) операция, 76, 388
(++) операция, 64, 403
(/) операция, 76
(//) операция, 590
(/=) операция, 166, 176
(:) операция, 80
(<*>) операция
  многоаргументная функция в IO с использованием <$> и, 392–393
  обзор, 388–390
  создание пользователя в контексте Maybe, 393–395
(<>) операция, 265
(==) операция, 166, 176
($) операция, 543
(>>=) операция, 419–420, 422
(>>) операция, 422
-> символ, 138
-Wall флаг, 523
-X флаг, 310
:info команда, 177, 320
:type команда, 162
 тег, 349
 тег, 349
_ (подчёркивание), 65
_where функция, 456–457
` (обратная кавычка), 86
accum функция, 591
add3ToAll функция, 110
addBang функция, 112
addQuestion функция, 111
addressLetter функция, 64, 75
addThenDouble функция, 164
addUser действие, 569
Aeson библиотека, 546–562
  stack, 548–549
  обзор, 549–550
  создание экземпляров классов типов FromJSON и ToJSON, 551–558
  установка, 548–549
aList параметр, 96
amazonExtractor функция, 67

app каталог, 484–485
append функция, 227
appendFile функция, 323–325
Applicative класс типов, 382–411
  pure метод, 400
  контейнеры и контексты, 401–403
  обзор, 398–400
  ограничения, 413–418
  операция, 387–396
    многоаргументная функция в IO с использованием <$> и, 392–393
    обзор, 388–390
    создание пользователя в контексте Maybe, 393–395
  списки как контекст, 403–411
    быстрая генерация большого набора тестовых данных, 408–411
    генерация первых n простых чисел, 407–408
    списки как контейнеры, 403–404
Arbitrary класс типов, 502
array функция, 588
assert, действие ввода-вывода, 495
assessCandidate функция, 437–438
assessCandidateIO функция, 434, 437
assessCandidateList функция, 437–438
assignToGroups функция, 88
Author тип, 209
BC.readFile, 335
BC.splitAt, 337
BinarySearchTree, 401
Bits синоним типов, 196–199
booksToHtml функция, 350
bookToHtml функция, 349
Bounded класс, 167
ByteString тип, 331–334, 343–344
cabal-файл, 482–484
calcChange функция, 42
cartCombine функция, 231
Char8 тип, 343–344
charToBits, 198
checkin действие, 578
Cipher класс, 201–203
cleanText функция, 497
Clojure, язык программирования, 605
CLOS (Common Lisp Object System), 121
Color тип, 223
  ассоциативность, 224–226
  обзор, 223–224
combineEvent функция, 231
combineTS функция, 267
Common Lisp, язык программирования, 605
compare метод, 178
compareLastNames функция, 62
compareTS функция, 271–273
concat функция, 227
concatAll функция, 114
Control.Monad модуль, 299, 445, 561
countsText функция, 323
createTS функция, 262–263, 267
cycle функция
  обзор, 87–88
  рекурсия на списках, 103
damage функция, 127
Data.Array.Unboxed, 588
Data.ByteString, 332, 537
Data.ByteString.Lazy, 537
Data.Char модуль, 476
Data.List модуль, 61–62, 117
Data.List функции, 88
Data.List.Split модуль, 303, 313
Data.Maybe модуль, 254
Data.Text модуль, 309–315
  intercalate функция, 314
  OverloadedStrings расширение, 310–311

splitOn функция, 313
unwords и unlines функции, 313
words функция, 313
вспомогательные функции, 312–315
операции класса типов Monoid, 314–315
Data.Text.Encoding, 344
Data.Text.Lazy, 308
Data.Time модуль, 567
db-lesson проект, 564–565
decode функция, 549
defaultRequest функция, 544
DeriveGeneric расширение, 546, 551
deriving ключевое слово, 180
Describable класс, 165
Dictionary тип, 244
diffPair функция, 274
diffTS функция, 275
DirectoryEntries, 358
displayResult функция, 532
div функция, 139
do-нотация, 288–290
для повторного использования кода в различных контекстах, 431–441
IO контекст, 433–434
Maybe контекст, 435–436
список как контекст, 436–438
для типа Maybe, 292
обзор, 428–430
double функция, 138
drop функция, 86
DuplicateRecordFields расширение, 312
echo функция, 430
Either тип
обзор, 519, 528–534
реализация проверки на простоту с, 531–534
eitherDecode функция, 550, 553
elem функция, 86
Elm, язык программирования, 606
empty функция, 227
Enum тип, 181
Eq класс типов, 166–167, 176, 253
error функция, 97, 521, 525
errorCode поле, 555
ErrorMessage тип, 555
exampleUrlBuilder функция, 71
execute функция, 569
exposed-modules значение, 483, 507
F#, язык программирования, 606
field функция, 572
FieldMetadata, 358, 361
FieldText, 359
fileCounts функция, 325
fileCounts.hs программа, 323
filter функция, 113, 114
firstOrNothing функция, 575
flipBinaryArgs функция, 75
fmap функция, 372, 378
foldl и foldr функции, свёртка списков, 114–118
foldM, 341–342
for циклы, 90
forM_ функция, 561
fromEnum метод, 181, 190
fromIntegral функция, 139
FromJSON класс типов, 551–558
fromList функция, 245
FromRow класс типов, 571–575
fst функция, 61
Functor класс типов, 185, 243, 367, 369–381
вычисление с типом Maybe, 370–371
использование функции в контексте, 372–374
ограничения, 386–387, 413–418
преобразование Map в HTML, 378–379
преобразование значений типа данных Maybe в Maybe

HTML, 376–377
преобразование списка значений типа данных в список HTML, 377–378
преобразование типа данных в IO в IO HTML, 379–380
genApiRequestBuilder функция, 72–74
genIfEven функция, 68
GET запрос, 69
getArgs функция, 296
getContents функция, 302, 325
getCurrentTime функция, 567
getFieldMetadata, 358
getLine функция, 283, 292, 321, 418
getLocation функция, 65
getMl message, 122
getPrice функция, 58, 67
getRequestURL функция, 70
getResponseBody функция, 544
getResponseHeader функция, 542
getResponseStatusCode функция, 540
GHC (Glasgow Haskell Compiler), 23–25
ghc-options значение, 523
GHCi интерактивный интерфейс
взаимодействие с, 25–27
вызов из stack, 493–494
командная история, 47
glitcher.hs файл, 335
guard функция, 445–446
Hackage, 178, 190
halve функция, 366
Handle тип, 321
Haskell, 22, 519–520
Idris, 603
Liquid Haskell, 603
более мощные системы типов чем, 602–603
hClose (закрыть дескриптор), 321
head функция
ошибки и, 522–526
частичные функции и, 525–526
helloName действие ввода-вывода, 427
helloPerson функция, 283, 288
hGetContents функция, 325
hGetLine функция, 321
highlight функция, 316
Hoogle, 178
hPutStrLn функция, 321
hs-source-dirs, 483
http-lesson проект, 536
http-lesson.cabal файл, 537
HTTP-запросы, 535–545
HTTP.Simple модуль, 539–541
начало проекта, 536–539
начальный код, 537–539
создание, 542–544
HTTP.Simple модуль, 539–541
httpLBS функция, 539
Idris, язык программирования
программирование с зависимыми типами, 603
ifEvenCube функция, 59
ifEvenDouble функция, 59, 68
ifEvenInc функция, 68
ifEvenNegate функция, 59
ifEvenSquare функция, 59, 68
ifEvenX функция, 68
IIFE (немедленно вызываемая функция), 54
inc функция, 161
incEven функция, 59
incEvenInc функция, 58
incMaybe функция, 371
init value, 115
insertMaybePair функция, 266
Int тип, 135
Integer тип, 136
intercalate функция, 255, 314
intToBits функция, 196–198
intToChar функция, 336
IO класс типов

do-нотация и повторное использование кода в различных контекстах, 433–434
многоаргументная функция в, с использованием <$> и <*>, 392–393
преобразование типов данных в IO в IO HTML, 379–380
реализация IO-действия echo, 418
IO типы, 282–292, 366–368
do-нотация, 288–290
действия ввода-вывода, 283–287
пример вычисления стоимости пиццы, 290–292
хранение значений в контексте IO, 288
isJust функция, 254, 512
isNothing функция, 254
isPalindrome функция, 118, 477, 496
isPrime функция, 510–515, 518, 528, 531–533
isPunctuation функция, 496
JPEG, помехи, 342
вставка случайных байтов, 336–339
соединение действий ввода-вывода с помощью foldM, 341–342
сортировка случайных байтов, 340
JPEG, помехи , 334
JPEGs, glitching
сортировка случайных байтов, 339
JSON данные, 546–562
Aeson библиотека, 549–550
stack, 548–549
определение экземпляров типов данных FromJSON и ToJSON, 551–558
установка, 548–549
json-lesson.cabal-файл, 549
L.writeFile, 544
LANGUAGE директива, 310, 317
избавление от директив, 488–489
lastReturned функция, 576
length функция
обзор, 85
рекурсия на списках, 100–101
let выражения, 51–53
let ключевое слово, 44
Lib модуль, 483–485, 492, 495, 502
Lib.hs файл, 485
libraryAdd функция, 53–54
LICENSE файл, 506
Liquid Haskell
уточнённые типы, 603
Lisp, семейство языков
Clojure, 605
Common Lisp, 605
Racket, 604, 605
рекомендуемые языки, 604–605
List тип, 240–241
listArray функция, 598
listToSTUArray, 593, 595
listToUArray функция, 596, 598
locationDB, 383
lookupFieldMetadata, 360
lookupSubfield, 361
lookupValue функция, 361, 362
main функция, 474
main-is значение, 483
Main.hs файл, 483–484, 486, 548
makeAddress функция, 141, 145
makeTriple функция, 145
makeTSCompare функция, 271
Map класс типов
обзор, 244–246
преобразование Map в HTML, 378–379
Map тип

комбинирование двух вызовов lookup для Map, 414–417
map функция, 110–113
mapM функция, 298, 300, 561
MARC (Machine-Readable Cataloging) записи, 351–362
использование каталога для поиска полей, 357
обработка каталоговых записей и поиск полей, 358–359
получение данных, 352–353
получение данных об авторе и названии из поля, 359–362
проверка заголовка и проход по записям, 353–355
структура, 351–352
чтение каталога, 355–357
marc_to_html.hs файл, 347
MarcDirectoryEntries, 357
MarcDirectoryEntryRaw, 358
MarcDirectoryRaw, 356
MarcLeaderRaw, 353
MarcRecordRaw, 353, 358, 361
maxBound value, 190
maxBound значение, 167, 508
Maybe Organ тип, 250
Maybe класс типов, 248–259
do-нотация для повторного использования кода в различных контекстах, 435–436
выполнение SQL-подобных запросов, 463–464
вычисление, 370–371
вычисление с, 255–259
обзор, 249–250
обработка частичных функций с, 526–527
преобразование значений типа данных Maybe в Maybe HTML, 376–377
создание пользователя в контексте, 393–395
Maybe тип
null значение, 251–253
Maybe типы, 366–368
maybeInc функция, 388
maybeMain функция, 293
mconcat метод, 228, 269
mean функция, 276
meanTS функция, 270–271
mempty элемент, 471
minBound значение, 167
minOfThree функция, 392
ML, семейство языков
рекомендуемые языки, 605–606
Elm, 606
F#, 606
PureScript, 606
Scala, 606
Monad класс типов, 292, 412–450
>>= операция, 419–420
do-нотация
в Maybe, 292
для повторного использования кода в различных контекстах, 431–441
обзор, 428–430
комбинирование двух вызовов lookup для Map, 414–417
ограничения Applicative и Functor, 413–418
программа-приветствие, 423–425
реализация IO-действия echo, 418
список как монада, 443–446
Monoid класс типов, 226–233
временные ряды, 265–270
законы, 228
и тип Text, 314–315
комбинирование нескольких элементов, 228

построение таблиц вероятностей, 229–233
movingAverageTS функция, 276
mul3ByAll функция, 110
myAdd функция, 73
myAny функция, 222
myGCD функция, 95
myList, 40
myProduct функция, 115
myReverse функция, 115
mystery функции, 73, 280
mystery1 метод, 280
mystery2 метод, 280
myTake функция, 523
myTakeSafer функция, 527
NA (не доступно), 262
Network.HTTP.Simple библиотека, 535, 539–541, 544
newArray функция, 593, 597
newList, 40
newSTRef функция, 597
newtype ключевое слово, 184
NOAAResponse тип данных, 559
NoImplicitPrelude расширение, 312
null pointer exception, 251
null значение, 251–253
Num class, 164
openFile функция, 320
Ord класс типов
обзор, 166–167
реализация, 178–179
Organ тип, 244
organCatalog функция, 246
OverloadedStrings расширение, 310–311, 332, 347, 537–539, 549, 565, 572
overwrite функция, 52
Palindrome модуль, 475–477
palindrome-checker.cabal, 487–489
patientInfo функция, 149
performCommand действие, 579
powersOfTwoAndThree функция, 448
Prelude модуль, 84, 469
preprocess функция, 477, 500
Primes модуль, 506
primes.cabal файл, 506–507
primes.cabal-файл, 511
print функция, 299
printDistance, 384
printDouble функция, 140
printResults IO действие, 561
printToolQuery функция, 574
printUsers функция, 573
process функция, 257
processRecords функция, 362
processRequest функция, 259
prop_allFactorsPrime свойство, 517
prop_factorsMakeOriginal свойство, 516
prop_primesArePrime свойство, 513
prop_punctuationInvariant свойство, 498
prop_reverseInvariant свойство, 499
prop_validPrimesOnly свойство, 512
PTable тип, 230
pure функция, 414
PureScript, язык программирования, 606
putStrLn функция, 286, 297, 321
putStrln функция, 418
query функция, 572
QuickCheck, 468, 497–503
использование с разными типами и установка библиотек, 502–503
обзор, 499–501
quickCheck функция, 499
quickcheck-instances, 502
quickCheckWith функция, 501
Racket, язык программирования, 604, 605
randomReplaceByte, 337, 340
randomRIO функция, 286, 287
randomSortSection, 340

rawToInt функция, 354
readArray действие, 597
readFile функция, 323–325
readInt функция, 392
readSTRef функция, 597
Real класс, 270
realToFrac функция, 270
recordLength, 356
REPL (цикл чтение-вычисление-печать), 38
replaceByte функция, 337
replicateM функция, 299–300
report функция, 257, 259
reset функция, 39
respond функция, 86
Resultset тип, 559
return функция, 422
reverse функция, 40, 85–86
ROT13 шифр, 186–194
  rotN алгоритм, 188–190
  кодирование строк, 190–191
  проблемы с, 191–194
  реализация, 187–190
rotDecoder, 192
rotEncoder, 192
rotN функция, 188–191
rotNdecoder, 192
RowParser тип, 571
runSTUArray функция, 595
sayAmount функция, 96
Scala, язык программирования, 606
select запрос, 455–456
  _select функция, 455
selectTool функция, 575
Semigroup класс типов, 222–226
  временные ряды, 265–270
  сделать цвета ассоциативными и охранные выражения, 224–226
  смешивание цветов, 223–224
Setup.hs файл, 506
setValue функция, 544
Show класс типов, 168–169
simple функция, 37, 144, 236
SixSidedDie тип, 173
snd функция, 61
someFunc функция, 484, 491
sortBy функция, 62, 183
sortSection функция, 339
splitOn функция, 303, 313
SQL-подобные запросы, 451–467
  select и where, 455–457
    реализация функции _select, 455–456
    реализация функции _where, 456–457
  выполнение запросов, 462–467
    использование с типами Maybe, 463–464
    соединение множества списков, 465–467
  интерфейс и примеры запросов, 459–461
  определение типа для запросов, 461–462
  соединение данных, 457–459
SQLite, 565–569
sqlite-simple библиотека, 563, 571
sqlite3 утилита, 566
squareAll функция, 110–112
src каталог, 484–485
ST тип, 597
stack ghci команда, 493
stack install команда, 538
stack new команда, 505
stack, утилита для сборки, 480–489
  вызов GHCi из stack, 493–494
  написание кода, 485–487
  начало проекта, 481
  сборка и запуск проектов, 487–489
  свойства, тестирование, 494–497
  структура проекта, 482–485

cabal-файл, 482–484
каталоги app, src и test, 484–485
сгенерированные файлы, 482–484
stack.yaml файл, 488, 506
StreamCipher тип, 203–204
String тип
сравнение с Text, 308
stripPunctuation функция, 477
stripWhiteSpace функция, 476
STUArray тип, изменение состояния с, 592–595
subtract2 функция, 76
sum.hs программа, 296
sumOfSquares функция, 115
sumSquareOrSquareSum функция, 49–50
swapST функция, 597
System.Environment, 296
take функция
обзор, 86
рекурсия на списках, 102–103
TDD (разработка через тестирование), 491
TemplateHaskell расширение, 312
test каталог, 484–485
Test.QuickCheck, 499
Text тип, 307–318
Data.Text модуль, 309–315
OverloadedStrings расширение, 310–311
вспомогательные функции, 312–315
IO тип, 317–318
и Юникод, 315–318
сравнение с String, 308
tick функция, 39
TIO.putStrLn функция, 317
toEnum method, 181
toInts функция, 304
ToJSON класс типов, 551–558
toList функция, 238
toLowerCase функция, 477
Tool тип, 567
ToRow class, 577
toString метод, 126, 176
Triple тип, 237–239
TwoSidedDie тип, 175
type ключевое слово, 150
UArray тип
неэффективность ленивых списков, 585–588
обновление значения, 590–591
создание, 588–589
Unicode
и тип Text, 315–318
unlines функция, 313
until циклы, 90
unwords функция, 313
update команда, 481
updateOrWarn действие, 576
updateTool функция, 576
URL, генерация для API, 69–74
User тип, 568
ViewPatterns расширение, 312
where блок, 43
написание с нуля, 48–51
where, в запросах, 455–457
реализация функции _where, 456–457
while циклы, 90
withConn действие, 570
words функция, 313
wreq пакет, 541
writeArray функция, 594, 597
writeFile функция, 323
writeSTRef функция, 597
XOR (исключающее ИЛИ) операция, криптография
Bits синоним типов, 196–199
одноразовый блокнот, 199–201
xor функция, 194–196, 199
xorBool функция, 196

xorPair функция, 196
xs переменная, 97
zip функция, 87, 589
zipWith функция, 230, 275
Аккермана функция, 104–105
Бхагавад Гита, 316
Коллатца гипотеза, 105–107
НОД (наибольший общий делитель), 94–95
ООП (объектно-ориентированное программирование), 119–131
  объекты с одним свойством, 120–123
    реализация аксессоров, 122–123
    создание конструкторов, 121
  программирование без состояния, 128–130
  сложные объекты, 124–128
  типы, 130–131
Тьюринга машина, 39
Фон Неймана архитектура, 34
Чёрча-Тьюринга тезис, 38
Юникод
  использование с типами ByteString и Char8, 343–344
автогенерация файла, 482–484
аксессоры, добавление к объектам, 122–123
алгебраические типы данных, 207
анонимная функция, 47
аргументы, тип функции с несколькими, 141–142
ассоциативность, Color тип, 224–226
базы данных, 563–582
  добавление новых пользователей, 569–570
  запись данных об аренде, 571
  настройка проекта, 564–565
  обновление существующих данных, 575–577
  удаление данных из, 578
  установка базы данных, 565–569
  чтение данных из, 571–575
    печать пользователей и инструментов, 572–575
    реализация экземпляра класса типов FromRow, 572
библиографические данные, 351
библиотека для простых чисел, 504–518
  запуск тестов, 514–515
  изменение файлов по умолчанию, 506–507
  исправление ошибок, 514–515
  написание базовых библиотечных функций, 507–511
  написание тестов для кода, 511–515
  начало проекта, 505–506
  определение свойств функции isPrime, 512–514
  определение функции isPrime, 510–511
  построение списка простых чисел, 508–509
  проверка чисел, не являющихся простыми, 513
  реализация разложения на множители, 515–518
  тест на корректность определения простоты, 512–513
ввод-вывод
  IO типы, 282–292
    do-нотация, 288–290
    IO действия, 283–287
    пример вычисления стоимости пиццы, 290–292
  ленивость, 295–306
    взаимодействие с командной строкой в энергичном

режиме, 296–301
взаимодействие с командной строкой, ленивое, 301–306
обзор, 279–281
текст и, 317–318
файлы
ленивый ввод-вывод и, 325–328
простые средства для, 323–325
строгость и, 328–330
виды, 243
временные ряды, анализ, 260–277
выполнение вычислений, 270–273
комбинирование, 266
комбинирование с помощью классов Semigroup и Monoid, 265–270
преобразование, 273–277
создание базового типа для, 262–265
вспомогательные функции, 312–315
intercalate функция, 314
Monoid класс типов, операции, 314–315
splitOn функция, 313
вычисление стоимости пиццы, IO типы, 290–292
гаверсинусов формула, 384
генераторы списков, 447–449
глобальная переменная, 40
данных конструктор, 151
двоичные данные, 331–364
ByteString тип, 332–334, 343–344
Char8 тип, 343–344
MARC записи, 351–362
использование каталога для поиска полей, 357
обработка каталоговых записей и поиск полей, 358–359
получение данных, 352–353
получение данных об авторе и названии из поля, 359–362
проверка заголовка и проход по записям, 353–355
структура, 351–352
чтение каталога, 355–357
Юникод, 343–344
помехи в JPEG, 334–342
вставка случайных байтов, 336–339
соединение действий ввода-вывода с помощью foldM, 341–342
сортировка случайных байтов, 339–340
двойные кавычки, 81
деванагари система письма, 315
десериализация, 547
динамическая типизация, 133
дроби, 208
зависимые типы, программирование на Idris, 603
задачи, их решения, 607
замыкания, 67–77
генерация URL для API, 69–74
обзор, 68–69
частичное применение, 72–77
записи, MARC, 351–362
использование каталога для поиска полей, 357
обработка каталоговых записей и поиск полей, 358–359
получение данных, 352–353
получение данных об авторе и названии из поля, 359–362
проверка заголовка и проход по записям, 353–355
структура, 351–352
чтение каталога, 355–357
запросы, SQL-подобные, 451–467
select и where, 455–457

реализация функции _select, 455–456
реализация функции _where, 456–457
выполнение запросов, 462–467
использование с типами Maybe, 463
соединение множества списков, 465–467
интерфейс и примеры запросов, 459–461
определение типа для запросов, 461–462
соединение данных, 457–459
зашифрованный текст, 201
и операция, комбинирование типов с, 208–213
или операция, комбинирование типов с, 213–216
классы типов, 161–185
Ord класс типов, реализация, 178–179
Show класс типов, 173
для более сложных типов, 182
минимальная полная реализация, 178
минимально полное определение, 176
обзор, 162–163
определение, 164–165
основные, 166
Bounded класс типов, 167
Eq класс типов, 166–167
Ord класс типов, 166–167
Show класс типов, 168–169
полиморфизм и, 174–176
порождение, 180–182
порождение экземпляров, 169–170
преимущества, 164
реализация по умолчанию, 176–178
схема, 185
книжные данные, 346–364
MARC записи, 351–362
использование каталога для поиска полей, 357
обработка каталоговых записей и поиск полей, 358–359
получение данных, 352–353
получение данных об авторе и названии из поля, 359–362
проверка заголовка и проход по записям, 353–355
структура, 351–352
чтение каталога, 355–357
обзор, 348–350
код
IO контекст, 433–434
Maybe контекст, 435–436
использование в разных контекстах, 431–441
написание, 28–32
работа с, 28–32
список как контекст, 436–438
команд история, GHCi, 47
командная строка
взаимодействие с, 296–301
взаимодействие с ленивым вводом-выводом, 301–306
композиция, 220–234
Monoid класс типов, 226–233
законы, 228
комбинирование нескольких элементов, 228
построение таблиц вероятностей, 229–233
Semigroup класс типов, 222–226
охранные выражения, 224–226
сделать цвета ассоциативными, 224–226
смешивание цветов, 223–224
комбинирование функций,

221–222
конструкторы данных, 151
конструкторы, создание, 121
кортеж как аргумент, 124
кортеж, тип, 235
кортежи, 241–242
криптография, 186–204
Cipher класс, 201–203
ROT13 шифр, 186–194
rotN алгоритм, 188–190
XOR операция, 194–196
Bits синоним типов, 196–199
одноразовый блокнот, 199–201
кодирование строк, 190–191
проблема с, 191–194
реализация, 187–190
лексическая область видимости, 46, 53–56
ленивые вычисления, 82–83
ленивые списки, их неэффективность, 585–588
ленивый ввод-вывод, 295–306
взаимодействие с, 301–306
взаимодействие с командной строкой в энергичном режиме, 296–301
против строгого, 329–330
файлы и, 325–328
линейный конгруэнтный генератор, 204
лямбда-функции, 46–56
let выражения, 51–53
как аргументы, 60
лексическая область видимости, 53–56
обзор, 47–48
реализация блока where с нуля, 48–51, 51
массивы, 583–600
извлечение значений из контекста, 595–597
изменение состояния, 592–595
неупакованные, 586
пузырьковая сортировка, 597–600
создание эффективное, 585–591
задание типа, 588–589
неэффективность ленивых списков, 585–588
обновление значения, 590–591
методы, 173
многофайловые программы, создание с помощью модулей, 473–479
использование собственного модуля в модуле Main, 477–479
размещение кода в собственном модуле, 475–477
создание модуля Main, 473–475
модули, создание многофайловых программ с, 473–479
использование созданного модуля в модуле Main, 477–479
размещение кода в собственном модуле, 475–477
создание модуля Main, 473–475
написание кода, 28–32
наследование, 211
невзламываемый текст, 201
недетерминированные вычисления, 403, 408
нейтральный элемент, 226–233
неупакованные массивы, 586
обратные кавычки, 86
одноразовые блокноты, 199–201
операций приоритет, 60
организация кода с помощью модулей, 469–479
многофайловые программы, 473–479

написание функции с именем из Prelude, 470–473
основная запись, 352
охранные выражения, 224–226
ошибки, 521–534
Either тип, 528–534
частичные функции, 522–527
head функция и, 525–526
обработка с помощью класса типов Maybe, 526–527
параметризованные типы, 235–247
IO типы, 282–292
do-нотация, 288–290
IO действия, 283–287
вычисление стоимости пиццы, 292
пример вычисления стоимости пиццы, 290
хранение значений в контексте IO, 288
List тип, 240–241
Map тип, 244–246
Maybe тип, 248–259
вычисление с, 253–255, 259
обзор, 249–250
отсутствующие значения, 251–253
Triple тип, 237–239
виды, 243
кортежи, 241–242
первого класса значения, функции, 57–66
возвращение функции, 63–66
лямбда-функции как аргументы, 60
пользовательская сортировка, 61
функциональный тип для, 142
переменные, 41–43
переприсваивание, 44–45
тип, 143–146
побочные эффекты, 39
подчёркивание, символ, 65
полиморфизм, классы типов и, 174–176
полиморфные константы, 139
помехи в JPEG, 334–342
вставка случайных байтов, 336–339
соединение действий ввода-вывода с помощью foldM, 341–342
сортировка случайных байтов, 339–340
преобразования, 205
присоединение, 80
программирование без состояния, 128–130
простой блокнот, 199
простые числа, библиотека, 504–518
изменение файлов по умолчанию, 506–507
написание базовых библиотечных функций, 507–511
определение функции isPrime, 510–511
построение списка простых чисел, 508–509
написание тестов для кода, 511–515
запуск тестов, 514–515
исправление ошибок, 514–515
определение свойств функции isPrime, 512–515
проверка чисел, не являющихся простыми, 513
тест на корректность определения простоты, 512–513
начало проекта, 505–506
реализация разложения на множители, 515–518
прототипы, основанное на них ООП, 126

пузырьковая сортировка, 597–600
расширения языковые, 310
реализация по умолчанию, 177
рекурсивные функции, 99–108
  Аккермана функция, 104–105
  Коллатца гипотеза, 105–107
  списки, 100–103
    cycle функция, 103
    length функция, 100–101
    take функция, 102–103
рекурсия, 90–98
  абстрагирование функцией map, 111–113
  наибольший общий делитель, 94
  обзор, 91
  правила, 92–94, 100
    выявление конечной цели, 92
    перечень всех возможностей, 93
    продуктивность на пути достижения цели, 94
    что повторяется, 93
    что происходит при достижении цели, 93
решето Эратосфена, 508
римские цифры, библиотека, 164
сводная статистика, 270
свойства, тестирование, 490–503
  QuickCheck, 497–503
    использование с разными типами и установка библиотек, 502–503
    обзор, 499–501
  виды тестирования, 492–497
  модульное тестирование и stack test, 494–497
  начало проекта, 491–492
сглаживание, 273
сериализация, 547
синтаксис записей, создание новых типов, 156–159
синтаксический сахар, 80
  обозначение, 80
скользящее среднее, 275–277
словари, 245
сообщения, передача между объектами, 126–128
сопоставление с образцом, 90, 95–98, 253
сортировка, с использованием функций как значений первого класса, 61–63
состояние, программирование без него, 128–130
списки, 78–89, 442–450
  в контексте, 436–438
  генераторы списков, 447–449
  их строение, 79–82
  как контекст, 403–411
    быстрая генерация большого набора тестовых данных, 408–411
    генерация первых n простых чисел, 407–408
    и списки как контейнеры, 403–404
  ленивые, 303–306
  ленивые вычисления, 82–83
  основные функции на, 84–89
    (!!) операция, 84–85
    cycle, 87–88
    drop, 86
    elem, 86
    length, 85
    reverse, 85–86
    take, 86
    zip, 87
  построение средствами монады для списка, 443–446
  преобразование списка значений типа данных в список HTML, 377–378
  рекурсия на, 100–103
    cycle функция, 103

length функция, 100–101
take функция, 102–103
свёртка, 114–118
фильтрация, 113, 114
ссылочная прозрачность, 38
статическая типизация, 133
строгий I/O
и ленивый, 329–330
и файлы, 328–330
строки, функциональные типы для преобразования в и из, 140–141
таблицы вероятностей, 229–233
текстовые редакторы, 23
тестирование свойств, 490–503
QuickCheck, 497–503
использование с разными типами и установка библиотек, 502–503
обзор, 499–501
виды тестирования, 492–497
модульное тестирование и stack test, 494–497
начало проекта, 491–492
типа синонимы, 150–151
типов вывод, 133, 135
типовые переменные, 143–146
типы, 132–147, 162, 207–219
комбинирование, 222–226
обзор, 135–138
объявление, 149–160
новых типов, 151–156
с использованием синонимов типов, 150–151
с использованием синтаксиса записей, 156–159
с помощью newtype, 184
параметризованные, 235–259
List тип, 240–241
Map тип, 244–246
null значения, 251–253
Triple тип, 237–239
виды, 243
вычисление с, 253–255, 259
кортежи, 241–242
обзор, 249–250
программирование в, 205–206
типовые переменные, 143–146
типы-произведения, 208–213
типы-суммы, 213–216
функциональные типы, 138–143
для преобразования в строки и из строк, 140–141
для функций как значений первого класса, 143
с несколькими переменными, 141–143
типы в контексте, 365–368
Applicative класс типов, 382–411
pure метод, 400
контейнеры и контексты, 401–403
обзор, 398–400
операция, 387–396
списки как контекст, 403–411
Functor класс типов, 369–381
вычисление с классом типов Maybe, 370–371
использование функции в контексте с, 372–374
преобразование Map в HTML, 378–379
преобразование значений типа данных Maybe в Maybe HTML, 376–377
преобразование списка значений типа данных в список HTML, 377–378
преобразование типа данных в IO в IO HTML, 379–380
Monad класс типов, 412–441
>>= операция, 419–420
do-нотация, 428–441
комбинирование двух

вызовов lookup для Map, 414–417
ограничения Applicative и Functor, 413–418
программа-приветствие, 423–425
реализация IO-действия echo, 418
списки, 442–450
генераторы списков, 447–449
построение средствами монады для списка, 443–446
типы-произведения
иерархическое проектирование и, 210–213
обзор, 208–213
типы-суммы, 213–216
уточнённые типы на Liquid Haskell, 603
файлы, 319–330
ленивый I/O и, 325–328
открытие и закрытие, 320–322
простые средства ввода-вывода для, 323–325
строгий ввод-вывод и, 328–330
функции
комбинирование, 221–222
обзор, 37–38
функции высшего порядка, 109–118
filter функция, фильтрация списков, 113, 114
foldl и foldr функции, свёртка списков, 114–118
map функция, 110–111
функции и, 37–38
значение их, 45
общее обсуждение, 34–35, 38–39
переменные, 43
функциональное объектно-ориентированное программирование, 119–131
объекты с единственным свойством, 120–123
добавление аксессоров, 122–123
создание конструкторов, 121
программирование с неизменяемым состоянием, 128–130
сложные объекты, 124–128
типы, 130–131
функциональное программирование, 36–45
функциональный тип, 138–142
для преобразования в строки и из строк, 140–141
с несколькими аргументами, 141–143
функции как значения первого класса, 143
хеш-таблицы, 245
частичное применение, 67, 72–77
частичные функции, 522–527
head функция и, 525–526
обработка с помощью класса типов Maybe, 526–527
числовые константы, 139
чистая функция, 280
чистый код, 280

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

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.

Оптовые закупки: тел. +7 (499) 782-38-89.

Электронный адрес: books@alians-kniga.ru.

Уилл Курт

Программируй на Haskell

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Касюлевич Я. О.,*
Романовский А. А.,
Степаненко С. Д.
Редактор *Брагилевский В. Н.*
Вёрстка *Брагилевский В. Н.*
Корректор *Синяева Г. И.*
Дизайн обложки *Мовчан А. Г.*

Формат $70\times100\,1/16$.
Вёрстка выполнена средствами TEXLive 2018.
Гарнитура «Paratype». Печать офсетная.
Усл. печ. л. 52.65. Тираж 200 экз.

Издательство ДМК Пресс
Электронный адрес издательства: www.dmkpress.com