Изучаем Haskell на примерах



Филипп Хагенлохер

MANNING

ДМК издательство

Филипп Хагенлохер

Изучаем Haskell на примерах

Learn Haskell by Example

PHILIPP HAGENLOCHER



MANNING
Shelter Island

Изучаем Haskell на примерах

ФИЛИПП ХАГЕНЛОХЕР

ДМК Пресс
издательство
Москва, 2025

УДК 004.438Haskell
ББК 32.973.2
Х13

Хагенлохер Ф.
Х13 Изучаем Haskell на примерах / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2025. – 518 с.: ил.

ISBN 978-5-93700-193-1

Haskell – это мощный и красивый, но непростой функциональный язык программирования. Вы научитесь создавать Haskell-приложения на примерах разработки полезных и увлекательных проектов. Вас ждут язык для сочинения музыки, библиотека обработки изображений и многое другое! Особое внимание уделено таким вопросам, как проектирование функций, композиция и неизменяемость данных.

Книга предназначена начинающим и опытным программистам, желающим освоить функциональное программирование. Ожидается, что читатель знает какой-либо объектно ориентированный язык.

УДК 004.438Haskell
ББК 32.973.2

Copyright © DMK Press 2025. Authorized translation of the English edition.

© 2025 Manning Publications. This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

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.

ISBN 978-1-6334-3893-4 (англ.)
ISBN 978-5-93700-193-1 (рус.)

© Manning Publications, 2025
© Перевод, оформление, издание, ДМК Пресс, 2025

Посвящается моим родителям, Хильдегард и Мартину.
Люблю вас всегда!

Оглавление

https://t.me/it_boooks/2

1 ■ Вступление 22
2 ■ Античное хранение секретов на современных машинах 36
3 ■ Каждая строка считается 70
4 ■ Утилита нумерации строк 89
5 ■ Слова и графы 120
6 ■ Решение игры в цепочки слов 151
7 ■ Работа с CSV-файлами 169
8 ■ Программа для работы с CSV-файлами 194
9 ■ QuickCheck и случайные тесты 219
10 ■ Цифровая музыкальная шкатулка 261
11 ■ Программирование музыкальных композиций 295
12 ■ Разбор пиксельных данных 315
13 ■ Параллельная обработка изображений 364
14 ■ Файлы и исключения 401
15 ■ Применение трансформеров для синхронизации 419
16 ■ JSON и SQL 447
17 ■ Построение API с применением Servant 466

Содержание

Предисловие ... 14
Благодарности ... 15
Об этой книге ... 17
Об авторе ... 20
Об иллюстрации на обложке ... 21

1 ***Вступление*** ... 22
1.1 Что такое Haskell? ... 23
1.1.1 Абстракции и теория ... 23
1.1.2 Островок безопасности ... 24
1.2 Путь чистого функционального программирования ... 26
1.2.1 Декларативный рецепт ... 26
1.2.2 От торта к программам ... 28
1.2.3 Это все для простоты ... 29
1.3 Использование абстракции ... 31
1.3.1 Преимущества ... 31
1.3.2 Недостатки ... 33
1.4 Чему мы научились ... 33
Резюме ... 34

2 ***Античное хранение секретов на современных машинах*** ... 36
2.1 Введение в Haskell ... 37
2.1.1 Шифр Цезаря ... 38
2.1.2 Новый проект ... 39
2.1.3 Первый модуль ... 42
2.2 Типичные типы и фантастические функции ... 44
2.2.1 Типы на атомарном уровне ... 45
2.2.2 Списки и кортежи ... 46

2.2.3 Типы функций ... 48
2.2.4 Добавление типов к математическим операциям ... 49
2.3 На помощь приходят алфавиты ... 50
2.3.1 Синонимы типов делают код понятнее ... 51
2.3.2 Виды букв ... 54
2.3.3 Логические комбинации ... 55
2.4 Вращение колеса ... 57
2.4.1 Нахождение индекса элемента ... 57
2.4.2 Нахождение элемента по индексу ... 60
2.4.3 Охрана потока управления ... 62
2.5 Преобразование строки ... 64
2.5.1 Отображение высшего порядка ... 64
2.5.2 Параметризация типов ... 65
2.5.3 Завершенный шифр ... 67
Резюме ... 69

3 *Каждая строка считается* ... 70
3.1 Общение с внешним миром ... 72
3.1.1 Простые действия ввода-вывода ... 72
3.1.2 Моделирование цикла ... 74
3.1.3 Выход из рекурсивного действия ... 75
3.2 Чистые функции внутри действий ... 77
3.2.1 Чтение и модификация пользовательских данных ... 78
3.2.2 Поток данных между чистым и нечистым кодами ... 79
3.3 Чтение из окружения ... 80
3.3.1 Разбор аргументов командной строки ... 81
3.3.2 Кодирование ошибок с помощью Maybe ... 82
3.4 Пример: чтение и печать аргументов командной строки ... 84
3.4.1 Ключевое слово let ... 86
3.4.2 Запуск программ с помощью stack ... 87
Резюме ... 87

4 *Утилита нумерации строк* ... 89
4.1 Чтение файлов и преобразование их содержимого ... 90
4.1.1 Написание чистой библиотеки ... 91
4.1.2 Сокрытие вспомогательных аргументов ... 94
4.2 Параметризация поведения функций высшего порядка ... 96
4.2.1 Частичное применение функции ... 98
4.3 Алгебраические структуры данных как кодирование возможностей ... 101
4.3.1 Типы-суммы или помеченные объединения ... 102
4.3.2 Не повторяйся ... 104
4.3.3 Функция zip ... 105
4.3.4 Работа с отсутствующими значениями ... 108
4.3.5 Печать списка значений с помощью tarM ... 109
4.4 От библиотеки к исполняемому файлу ... 110
4.4.1 Задание и разбор параметров командной строки ... 111
4.4.2 Итоги проекта ... 114
Резюме ... 119

5 *Слова и графы* 120
5.1 Построение графа 121
5.1.1 Полиморфные типы 122
5.1.2 Знакомство с новым модулем 125
5.1.3 Класс типов Eq и ограничения типа 127
5.1.4 Функция flip 129
5.2 Инкапсуляция реализации 130
5.2.1 Добавление и удаление записей 132
5.2.2 Использование списков экспорта для сокрытия конструкторов 133
5.2.3 Класс типов Show 137
5.3 Использование и повторное использование кода 139
5.3.1 Квалифицированный импорт 139
5.3.2 Построение карт перестановок 141
5.3.3 Создание карты перестановок по словарю 144
5.4 Параметризация преобразований 145
5.4.1 Списковые включения 147
Резюме 149

6 *Решение игры в цепочки слов* 151
6.1 Поиск в ширину 152
6.1.1 Общее описание алгоритма 152
6.1.2 Отслеживание состояния поиска 154
6.1.3 Нахождение решения с помощью обратного прохода 156
6.2 Область видимости переменной типа 157
6.2.1 Навешивание квантора всеобщности 157
6.2.2 Расширения языка 159
6.2.3 Использование переменных типа с лексической областью видимости 160
6.3 Повышение производительности с помощью хеш-таблиц 162
6.3.1 Анализ производительности с помощью профилирования 163
6.3.2 Добавление зависимостей в проект 165
6.3.3 Ленивое вычисление 167
Резюме 168

7 *Работа с CSV-файлами* 169
7.1 Моделирование CSV-данных 170
7.1.1 Синтаксис записей 171
7.1.2 Кодирование ошибок типом Either 175
7.2 Умные конструкторы 176
7.2.1 Проверка свойства на этапе конструирования 176
7.2.2 Предоставление небезопасной альтернативы 178
7.2.3 Оператор доллар 180
7.3 Использование классов типов 181
7.3.1 Полугруппа и моноид 183
7.3.2 Класс типов IsString 186
7.4 Создание нового класса типов 188
7.4.1 Класс типов для вырезания структур данных 188
7.4.2 Реэкспорт модулей 192
Резюме 193

8 *Программа для работы с CSV-файлами* 194
8.1 Разбор данных 195
8.1.1 Разбор числовых значений 197
8.2 Свертка структур данных 198
8.2.1 Концепция свертки 198
8.2.2 Структура данных для разбора 202
8.2.3 Класс типов Functor 203
8.2.4 Использование свертки при разборе 205
8.3 Печать CSV-таблицы 207
8.3.1 Операции с CSV-таблицами 210
8.4 Простой парсер командной строки 212
8.4.1 Поддержка флагов и более сложных аргументов 213
Резюме 218

9 *QuickCheck и случайные тесты* 219
9.1 Как тестировать 220
9.1.1 Тестирование свойств 221
9.1.2 Генерирование случайных значений 222
9.1.3 Random и Uniform 224
9.1.4 Использование глобального генератора случайных значений 226
9.1.5 Простой тест свойств 228
9.1.6 Определение постусловий для случайных значений 230
9.2 Рандомизированное тестирование 235
9.2.1 Преимущество ссылочной прозрачности 238
9.3 Каркас тестирования QuickCheck 239
9.3.1 Тип Property в QuickCheck 240
9.4 Генерирование случайных значений для тестирования 242
9.4.1 Генератор случайных значений Gen 243
9.4.2 Пример: AssocMap 246
9.4.3 Сокращение тестов 248
9.5 Практическое использование тестирования свойств 250
9.5.1 Уровень детализации и отчеты о покрытии 251
9.5.2 Модификация параметров теста 254
9.5.3 Построение комплектов тестов 254
9.5.4 Эффективность тестирования 258
Резюме 260

10 *Цифровая музыкальная шкатулка* 261
10.1 Моделирование звука с помощью чисел 262
10.1.1 Зоопарк классов числовых типов 263
10.1.2 Создание периодических функций 267
10.2 Работа с бесконечными списками 273
10.2.1 Подъем, спад, поддержание и затухание 273
10.2.2 Построение бесконечных списков и работа с ними 275
10.3 Управление синтезом 278
10.3.1 Селекторы частичных полей 278
10.3.2 Функция как тип 281
10.4 Нотные модели 283
10.4.1 Класс типов для высоты звука 283

10.4.2 Tun Ratio 288
10.4.3 Различные способы возведения в степень 291
Резюме 293

11 *Программирование музыкальных композиций* 295
11.1 Полиморфные структуры данных с несколькими типами 296
11.1.1 Навешивание квантора существования 297
11.1.2 Использование типов с квантором существования 299
11.2 Интерпретация структур 300
11.2.1 Микширование сигналов 303
11.2.2 Полифонические группы 306
11.3 Реализация предметно-ориентированного языка 309
11.3.1 Упрощение синтаксиса 310
11.3.2 Пользовательские операторы для спископодобных структур данных 310
11.3.3 Объявления приоритета операторов 312
Резюме 314

12 *Разбор пиксельных данных* 315
12.1 Написание парсера 316
12.1.1 Переносимые изображения из прошлого 316
12.1.2 Как разбирать файл 319
12.1.3 Композиция эффектов 324
12.1.4 Выбор варианта 330
12.1.5 Введение в монады 334
12.1.6 Обсуждение монад 341
12.1.7 Как правильно ошибаться 343
12.2 Разбор по-крупному 346
12.2.1 Введение в библиотеку Attoparsec 348
12.2.2 Разбор изображений 350
12.2.3 Выбор формата 354
12.2.4 Собираем все парсеры вместе 359
Резюме 362

13 *Параллельная обработка изображений* 364
13.1 Предоставление информации о типе вызывающей стороне 365
13.1.1 Проблемы полиморфизма по возвращаемому значению 365
13.1.2 Обобщенные алгебраические типы данных 369
13.1.3 Tun Vector 371
13.1.4 Динамические типы с навешенными кванторами существования 374
13.2 Проверка разобранных данных 375
13.3 Обобщенный алгоритм преобразования изображения 382
13.3.1 Применение матриц преобразования к отображениям 384
13.3.2 Экспорт изображений в формате PNG 387
13.4 Использование распараллеливания для преобразования данных 389
13.4.1 Измерение времени 392

13.4.2 Как устроен параллелизм? 393
13.4.3 Искры и контексты выполнения Haskell 395
Резюме 400

14 ***Файлы и исключения*** 401
14.1 Открытие и чтение файлов 402
14.1.1 System.IO и Handle 402
14.1.2 Буферизация и поиск 404
14.2 Чтение байтов из файла 406
14.2.1 Захват ресурсов и функция bracket 408
14.3 Работа с файловой системой и исключения 409
14.3.1 Модули System.Directory и System.FilePath 409
14.3.2 Перечисление файлов и каталогов 410
14.3.3 Основы исключений 411
14.4 Возбуждение и перехват исключений 413
14.4.1 Обработка ошибок 415
Резюме 418

15 ***Применение трансформеров для синхронизации*** 419
15.1 Монадные трансформеры 420
15.1.1 Чтение окружения с помощью ReaderT 420
15.1.2 StateT и WriterT 423
15.1.3 Организация стека трансформеров с помощью RWST 429
15.2 Реализация приложения 430
15.3 Организация интерфейса с командной строкой 436
15.3.1 Разбор аргументов с помощью optparse-applicative 436
15.3.2 Перечисление пользовательских типов 440
15.3.3 Класс Enum 441
15.3.4 Интерфейс с командной строкой для App 443
Резюме 446

16 ***JSON и SQL*** 447
16.1 Кодирование значений в формате JSON 448
16.1.1 Aeson и разбор JSON 448
16.1.2 Сериализация в формат JSON 452
16.2 Вывод классов типов с помощью Generic 454
16.3 Использование базы данных SQLite 456
16.3.1 Основы sqlite-simple 456
16.3.2 Функции ToRow и FromRow 457
16.3.3 Определение действий для доступа к базе данных 461
Резюме 465

17 ***Построение API с применением Servant*** 466
17.1 Определение типобезопасного API 466
17.1.1 Servant и типизированные API 467
17.1.2 Фантомные типы 470
17.1.3 Реализация API 472
17.2 Запуск приложения 477

17.2.1 WAI-приложение 478
17.2.2 Средний уровень приложения 482
17.2.3 Порождение HTML-кода по данным 484
17.3 Вывод клиента 490
17.3.1 Использование servant-client 490
17.3.2 Определение команд 491
17.3.3 Реализация клиента 494
Резюме 497

Приложение A. Комплект инструментов Haskell 498
Приложение B. Ленивое вычисление 502
Предметный указатель 508

Предисловие

Haskell – язык, который при взгляде извне может показаться окутанным тайной, изобилующим научной терминологией и непонятными концепциями. Кое-что из этого правда, но я абсолютно уверен, что решить задачу на этом языке можно, не обладая степенью PhD по математике. Все дело в правильной подводке. На Haskell можно очень быстро начать писать реальные программы, и книга именно об этом.

Открою вам маленький секрет: моя первая попытка изучить Haskell окончилась весьма плачевно. Никакой другой язык не казался мне таким странным и сбивающим с толку (за исключением разве что Prolog и Uiua). Прошло много лет, прежде чем я снова вернулся к Haskell, и счастье мое не знало пределов!

А изменилась только подводка. В тот момент у меня за плечами уже было несколько лет опыт опыта работы с языками функционального программирования, и я был хорошо знаком с академическим жаргоном, неотъемлемым от рассказа о Haskell. Прежде всего я понимал, как его применять, а ведь язык обретает смысл, только если осмысленно применять его к решаемой задаче, – поэтому я считаю этот аспект критически важным для изучения языка. И в этой книге я хочу акцентировать на нем внимание.

Но зачем вообще изучать Haskell? Что мы от этого получим? Помимо права гордиться проделанной работой, мы получаем совершенно новый способ структурирования программ и обдумывания задач. Haskell – чистый функциональный язык, он учит нас минимизировать состояние, в результате чего программу становится принципиально проще понять. Кроме того, умение применять Haskell к реальным задачам – еще один инструмент в вашем багаже, благодаря которому вы совершенствуетесь как программист. Да и другие функциональные языки после Haskell обычно изучать гораздо проще. Неважно, хотите ли вы изучить Haskell, чтобы более корректно писать программы, из чистого любопытства или чтобы расширить горизонты, эта книга послужит вам путеводителем по глубокому и полному чудес миру Haskell.

Благодарности

Никакая книга не возникает из ничего и не пишется одиноким автором за один вечер (о, как бы я этого желал!) – и эта книга не исключение. Она никогда не появилась бы на свет без неустанной работы многих людей и без неоценимой помощи еще гораздо большего их числа.

Прежде всего я хочу поблагодарить тех, кто всего ближе ко мне в жизни: родителей за их непрестанную поддержку на протяжении всей моей жизни. Без вас я не стал бы тем, кто я есть, и за это я вам вечно благодарен. Кроме того, хочу сказать спасибо Али, всегда готового слушать мои тирады на темы программирования.

Далее я хочу выразить благодарность всему коллективу издательства Manning, а особенно моим редакторам, Патрику Барбу и Кэти Спосато, за слова ободрения и терпение, с которым они переносили меня и эту книгу. Отдельное спасибо Александру Вершилову, который настойчиво побуждал меня уделять внимание техническим аспектам Haskell. Александр 15 лет работал программистом, имеет степень PhD по физике, а сейчас возглавляет команду инженеров, работающих над образовательным проектом на Haskell. Большое спасибо Ивану Мартиновичу за помощь в технических вопросах процесса написания книги. Отдельная благодарность техническому редактору, Тимо фон Хольцу. Без твоих подсказок эта книга никогда не была бы доведена до конца.

Спасибо также бесчисленным рецензентам: Алексу Мэйсону, Алексею Выскубову, Энди Дженнингс, Анкиту Митталу, Антону Ричу, Кэмерону Пресли, Клиффорду Тэрберу, Давиду Паку, Дамиану Эстебану, Дэну Шейху, Эрнесто Босси, Эрнесто Гарсиа, Франсу Ойлинки, Гаутаме Садависаму, Грегорио Пиччоли, Гуннару Альбергу, Джеку Келли, Джоэлю Холмсу, Джуральдиньо Шикину, Каю Гэллиену, Кате Паткин, Кристофу Семьену, Киту Л. Манноку, Кэвину Чунгу, Магнусу Тэрнингу, Мансуру Мухитдинову, Марко Перроне, Марку Эльсону, Майклу Д. Роучу, Полу Силистеану, Тэду Мейеру и Винисиосу Венцу. Ваши предложения помогли сделать книгу лучше.

Наконец, было бы непростительным упущением не поблагодарить чудесное сообщество Haskell – и не только за то, что оно вдохнуло жизнь в этот язык, но и за постоянные нововведения, добавляемые с неустанным рвением. Пишущие на Haskell – одни из самых умных, талантливых и доброжелательных людей, с которыми мне приходилось общаться, и надеюсь, что эта книга привлечет новых адептов в наше прекрасное сообщество.

Об этой книге

Эта книга ставит задачей научить языку Haskell с нуля до создания практически полезных приложений, а не нудных упражнений. Слишком часто к Haskell относятся как к чисто академическому языку. Долой это предвзятое мнение! Haskell практичен настолько, насколько это нужно.

Сместив акцент на применение языка, мы поможем читателю лучше понять фундаментально присущие ему сильные стороны, а также то, как его красивые абстракции становятся не просто мысленным экспериментом, а реальным инструментом для реальной работы.

На кого рассчитана эта книга

Это книга для программистов, которые уже знакомы хотя бы с одним языком программирования и хотят распробовать на вкус функциональное программирование на Haskell. Предварительное знакомство с функциональным программированием не предполагается. Книга не является полным руководством по языку, но может послужить точкой входа, которая позволит читателю приступить к написанию собственного кода.

Структура книги

Эта книга состоит из 15 глав, в которых описываются девять программных проектов. Понятия и конструкции языка объясняются по мере необходимости.

- Глава 1 содержит введение в Haskell и книгу в целом. В ней приводятся исторические сведения и рассматриваются основы функционального программирования, не вдаваясь в специфику кода на Haskell.
- В главе 2 начинается вступительный проект: шифр Цезаря, древний алгоритм шифрования.

- В главах 3 и 4 мы познакомимся с клоном программы `nl` для подсчета строк текстового файла; это первая в книге исполняемая программа.
- Предметом глав 5 и 6 служит обработка данных и проектирование алгоритмов на примере игры в цепочку слов.
- В главах 7 и 8 мы познакомимся с практическим инструментом для разбора, сериализации, поиска и модификации CSV-файлов.
- Глава 9 посвящена проверке свойств и содержит практические советы по написанию комплектов тестов для вашей программы.
- В главах 10 и 11 рассматривается программный синтезатор со встроенным предметно-зависимым языком для сочинения музыки.
- Главы 12 и 13 содержат введение в библиотеку для разбора данных изображений и предоставляют распараллеленный конвейер преобразования изображений.
- Главы 14 и 15 посвящены работе с исключениями в приложении для синхронизации файлов.
- В главах 16 и 17 читатель научится писать сервер REST API с использованием JSON, базы данных и автоматически генерируемого клиентского приложения.

Вообще говоря, каждая глава опирается на знания, полученные в предыдущих главах, поэтому читать их лучше подряд, ничего не пропуская. Даже если сам проект вас не интересует, необходимо освоить излагаемые в главе ключевые концепции, прежде чем переходить к следующей.

О примерах кода

В этой книге много примеров кода как в пронумерованных листингах, так и в виде небольших фрагментов прямо в тексте. В обоих случаях исходный код набран `моноширинным шрифтом`, чтобы отделить его от обычного текста. Иногда код набран **`полужирным шрифтом`**, чтобы показать, что он отличается от предыдущих шагов в этой же главе, например когда в старую строку кода добавляется новая возможность.

Во многих случаях оригинальный исходный код переформатирован; мы разбивали строки и изменяли отступы, сообразуясь с размером печатной страницы. Иногда этого оказывалось недостаточно, и тогда мы добавляли маркеры продолжения строк (➥). Кроме того, комментарии, присутствующие в исходном коде, часто удалялись из листингов, если код описан в тексте. Многие листинги сопровождаются аннотациями, подчеркивающими важные концепции.

Исполняемые фрагменты кода можно найти в онлайновой версии книги на сайте https://livebook.manning.com/book/learn-haskell-by-example. Полный код примеров можно скачать бесплатно с сайта издательства Manning по адресу https://www.manning.com/books/learn-haskell-by-example, также он доступен на Github по адресу https://github.com/phagenlocher/learn-haskell-by-example.

Отзывы и пожелания

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

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

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе



Филипп Хагенлохер (Philipp Hagenlocher) получил степень магистра информатики в Мюнхенском техническом университете, где приобрел солидные знания в области формальных методов и функционального программирования. Он работает программистом на Haskell на полной ставке и занимается распределенными системами, для которых абсолютная корректность программы – насущная необходимость. Филипп уже давно питает страсть к обучению других концепциям функционального программирования и на своем опыте знает, какие трудности могут возникнуть у читателя. Его серия роликов «Haskell for Imperative Programmers» на YouTube собрала миллион просмотров.

Об иллюстрации на обложке

На обложке книги изображена «Femme de Saint-Pierre», или «женщина из Сен-Пьера». Рисунок взят из книги Луи Курмера, изданной в 1841 году. Все иллюстрации изящно нарисованы и раскрашены вручную.

В те времена легко было по одежде определить, где человек живет, чем занимается и каков его статус. Издательство Manning откликается на новации и инициативы в компьютерной отрасли обложками своих книг, на которых представлено широкое разнообразие местных укладов быта в прошлых веках. Мы возвращаем его в том виде, в каком оно запечатлено на рисунках из таких собраний, как это.

1 Вступление

https://t.me/it_boooks/2

Краткое содержание этой главы:

- что такое Haskell;
- что такое функциональное программирование и почему оно важно;
- в чем польза от использования абстракций в программировании;
- чему вы научитесь, прочитав эту книгу.

Сегодня нас окружает крайне сложное программное обеспечение, и нам, программистам, нужны инструменты, для того чтобы писать его. В первую очередь это языки программирования, которые упрощают процесс разработки. И Haskell – современный язык, сочетающий в себе передовые технологии и элегантность, которой удается достичь не многим языкам, – именно таков. Со своим впечатляющим набором возможностей Haskell окутан легендами и мифами.., но мы попробуем приоткрыть завесу тайны.

В этой книге мы рассмотрим реализацию нескольких небольших (кто-то даже назовет их *миниатюрными*) проектов. Одни созданы просто ради интереса, другие окажутся полезными утилитами, а часть специально подобрана, чтобы показать вам, как эффективно использовать Haskell для создания быстрых, безопасных и надежных решений. В процессе выполнения этих проектов вы узнаете обо

всех нюансах написания программного обеспечения с использованием некоторых наиболее элегантных и развитых моделей программирования.

Если вы уже сталкивались с Haskell ранее и думали: «Я даже не знаю, с чего начать», или сомневались в его практичности, не переживайте. Эта книга не требует никаких знаний о возможностях Haskell, да и о функциональном программировании в целом. Мы начнем с простых алгоритмов и постепенно дойдем до написания собственного веб-сервера!

Так что не теряем времени. Вперед – этот путь определенно стоит того!

1.1 Что такое Haskell?

Если вы открыли эту книгу, то, скорее всего, у вас уже есть общее представление о том, что такое Haskell. Вы знаете, что это язык программирования, возможно, даже что это функциональный язык. Вероятно, вы также слышали о его уникальных возможностях. Но зачем вам использовать Haskell? Что делает его таким особенным?

1.1.1 Абстракции и теория

Во-первых, Haskell предлагает множество высокоуровневых абстракций, таких как алгебраические типы данных, классы типов и монады, которые подробно разобраны в этой книге. Эти абстракции позволяют писать аккуратный, модульный и чистый код, упрощая сложные задачи и превращая их в повторно используемые функциональные блоки. Но чем же это полезно? Такие подходы экономят время и избавляют от головной боли при отладке программ. Например, в Haskell алгоритмы часто пишутся один раз, а затем многократно используются в разных контекстах. Зачем каждый раз разрабатывать новый алгоритм сортировки для разных типов данных, если можно сделать это один раз?

Еще одна важная особенность Haskell – его тесная связь с академической средой и современными исследованиями в области теории типов и проектирования языков программирования. Многие популярные языки создавались для решения задач индустрии. А это очень сильно влияет на дизайн языка. Один из лучших примеров – Java, который изначально рекламировался под девизом «*пишешь один раз – запускаешь везде*», что повлияло на многие низкоуровневые решения. Позднее появилась версия Enterprise Edition, нацеленная на разработку веб-сервисов и распределенные вычисления. Haskell, напротив, изначально разрабатывался как экспериментальный язык, основанный на результатах теории типов и исследований

в области языков программирования. Такой подход позволил сфокусироваться на простых, но мощных концепциях, лежащих в основе языка. Это сделало Haskell в большей степени ориентированным на корректность программ, чем другие языки, что привело к решениям, направленным на повышение безопасности.

Haskell – не просто функциональный язык программирования, а *чистый функциональный язык*. Говоря «чистый», мы имеем в виду, что функции в программах на Haskell очень похожи на функции в математическом смысле. У них есть входные данные и результат, но отсутствуют побочные *эффекты*. Это значит, что функции работают исключительно с переданными им данными и не могут, например, считывать информацию с диска или из сети. Полезно ли это? Безусловно, такой подход упрощает анализ программ на Haskell: их можно понять, просто читая код. Это обычно приводит к меньшему числу ошибок и к коду, который легче поддерживать. Мы подробно рассмотрим этот принцип в книге, так как он заставляет взглянуть на программирование с совершенно иной точки зрения.

1.1.2 Островок безопасности

Итак, как Haskell соотносится с другими языками программирования, учитывая вышеупомянутые особенности дизайна? Один из ключевых атрибутов языков программирования – это *сила утверждений*, то есть количество правил, налагаемых на семантику программирования компилятором или интерпретатором языка. Этот аспект приводит нас к двум важным свойствам языков программирования: *свободе* и *безопасности*. *Свобода* отражает, насколько программист ограничен в своих возможностях изменять состояние программы или работать с такими ресурсами, как память, файлы, сокеты и потоки. *Безопасность*, в свою очередь, описывает вероятность того, что программа будет изначально свободна от неопределенного поведения и ошибок. Эти два свойства находятся в обратной зависимости: чем больше возможностей для выполнения опасных операций предоставляет язык, тем ниже его безопасность. И наоборот, запрет таких операций повышает безопасность, но снижает свободу, так как либо эти операции становятся недоступны, либо их реализация сопряжена с большими накладными расходами. Этот баланс демонстрируется на рис. 1.1.

Когда речь заходит об этих свойствах, языкам программирования приходится искать компромисс. Многие динамически типизированные скриптовые языки, такие как Python, позволяют программисту делать практически все, что он захочет. Однако результатом такого подхода становится низкий уровень безопасности. Это упрощает написание скриптов, но одновременно повышает вероятность того, что программа завершится ошибкой во время выполнения.

Статически типизированные компилируемые языки стремятся повысить безопасность за счет того, что компилятор выполняет проверки корректности и вынуждает программиста соблюдать правила системы типов, чтобы успешно скомпилировать программу. Чем больше ограничений накладывает компилятор, тем надежнее программа в плане защиты от ошибок, но тем сложнее написать код, который будет работать. Например, Python позволяет передавать значения произвольных типов в любую функцию. Это упрощает создание полиморфных функций, но также делает код уязвимым для ошибок, потому что вы не можете быть уверены в типах аргументов функций. В таких языках, как C, компилятор проверяет хотя бы то, что типы переменных совпадают, но это дает вам полную свободу действий с памятью, что позволяет легко управлять данными на низком уровне, но может запросто привести к неопределенному поведению. Java использует сборщик мусора как абстракцию, позволяющую решить эту проблему, но при этом позволяет разделять ссылки на объекты между процессами, что может приводить к гонкам за данные. Такие языки, как Rust, запрещают подобное поведение, исключая многие типы неопределенного поведения, но все еще разрешают изменяемое состояние. Haskell – язык, который находится на *безопасной* стороне, в нем много ограничений, например обязательное использование неизменяемых данных и запрет побочных эффектов без использования специальных программных средств.



Рис. 1.1 Соотношение свободы и безопасности

На рис. 1.2 показано, как процесс компиляции программы можно представить в виде конвейера. Программист пишет (синтаксически корректный) исходный код, который затем обрабатывается компилятором (или интерпретатором) для создания двоичного исполняемого файла. Чем более строгие правила накладывает компилятор, тем сложнее работа программиста. С другой стороны, это исключает множество программ, которые могли бы завершиться ошибкой во время выполнения. Таким образом, если компилятор предъявляет высокие требования, это существенно увеличивает вероятность того,

что скомпилированная программа будет работать корректно, а не завершится ошибкой и не вызовет неопределенное поведение. По сути, компилятор проверяет, что намерения программиста правильно выражены в коде. Haskell относится к этой категории безопасных языков. Чтобы вы лучше поняли уникальность и преимущества подхода Haskell к обеспечению безопасности, начнем с небольшого экскурса в основы.

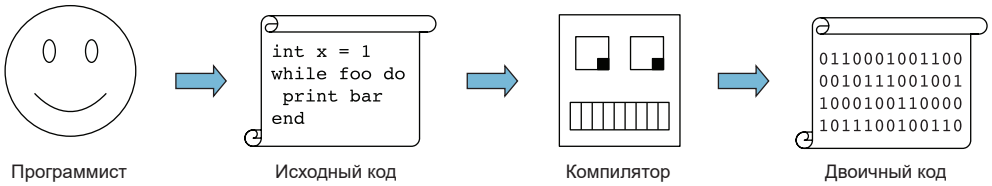


Рис. 1.2 Конвейер компиляции

1.2 Путь чистого функционального программирования

С самого начала развития программирования мы задаемся вопросом, который легко сформулировать, но крайне сложно решить: как убедиться, что наши программы корректны? Под этим мы понимаем не только отсутствие ошибок в работе программ, но и то, что они ведут себя точно так, как задумал и описал программист. Сегодня значительная часть усилий разработчика сосредоточена на обеспечении качества и тестировании программного обеспечения. Иногда кажется, что мы просто не можем полностью доверять собственным программам. Но почему это так? Разве мы не знаем, что делает написанная нами функция? В большинстве случаев знаем! Однако существует одна вещь, которую мы не можем полностью контролировать, – *состояние*. Во время выполнения у программы есть состояние, которое включает данные, с которыми она работает (наши переменные), и контекст выполнения (системное окружение, в котором запускается программа). Одно из решений – разбить состояние на части и упаковать его в объекты, предоставляя интерфейсы для контролируемого взаимодействия с ними. По сути, это и есть объектно ориентированное программирование. Однако существуют и другие подходы к решению этой проблемы.

1.2.1 Декларативный рецепт

А что, если вместо того чтобы разделять состояние, мы попробуем просто *уменьшить* его? Ведь чем меньше вокруг нас отвлекающих

деталей, тем лучше, не так ли? Эта идея лежит в основе *чистого функционального программирования*, на котором построен Haskell!

Для тех, кто еще незнаком с этим понятием, ниже приведен небольшой экскурс в мир функционального программирования. Что это такое? Для начала давайте рассмотрим, как обычно работают нефункциональные программы. Такую программу можно представлять себе как последовательность действий, что-то вроде рецепта. Она указывает, какие шаги нужно выполнить, чтобы достичь результата. Это может выглядеть примерно так:

1. Растопите *масло* на сковороде на слабом огне.
2. Добавьте *шоколад*.
3. Разбейте *8 яиц* в миску.
4. Добавьте *муку*, *сахар* и *разрыхлитель*.
5. Тщательно перемешайте с помощью ручного миксера.
6. Добавьте *масло-шоколадную смесь*.
7. Перелейте *тесто для шоколадного торта* на противень.
8. Выпекайте 25 минут при температуре 200 °C.

Наконец, выполнив все шаги, вы получите шоколадный торт. Но что, если вам вдруг захочется испечь лимонный торт? Скорее всего, рецепт будет выглядеть почти так же, как рецепт шоколадного торта, за исключением шага 2, где вместо добавления шоколада потребуется что-то лимонное. Итак, если бы мы захотели написать рецепт лимонного торта, нам пришлось бы скопировать почти все шаги из предыдущего рецепта. Проблема, не правда ли? Какие шаги копировать? Мы понимаем, что шаги 1 и 2 не имеют отношения к приготовлению теста для любого торта – это дополнительные действия. Шаг 6 тоже очень специфичен и не может быть применен к другим рецептам, потому что требует масло-шоколадной смеси. Это и есть проблема состояния! Каждый шаг изменяет текущее состояние (в данном случае – приготовленные ингредиенты), и некоторые шаги зависят от этого состояния. Инструкция вроде «тщательно перемешайте» имеет смысл только при условии, что есть что перемешивать!

А как мог бы выглядеть функциональный рецепт? Вот пример:

- *Масло-шоколадная смесь* состоит из растопленных на слабом огне *масла* и *шоколада*.
- *Тесто для торта* состоит из *8 взбитых яиц*, *муки*, *сахара* и *разрыхлителя*.
- *Тесто для шоколадного торта* – это *тесто для торта*, перемешанное с *масло-шоколадной смесью*.
- *Шоколадный торт* – это *тесто для шоколадного торта*, выпеченное в духовке при 200 °C в течение 25 минут.

Мы сразу видим огромную разницу. Такой рецепт не указывает, какие шаги выполнять, а определяет промежуточные результаты! Из этих определений мы легко можем вывести, какие действия нужно

предпринять, чтобы получить готовый шоколадный торт. Во-первых, сначала нужно приготовить тесто для шоколадного торта, для чего требуется тесто для торта, смешанное с масло-шоколадной смесью. В свою очередь, то и другое определяются своими рецептами. Таким образом, следуя определениям рекурсивно, мы дойдем до базовых шагов (добавления ингредиентов), из которых получим конечный результат. А что, если мы хотим лимонный торт? Легко! В третьем определении мы просто заменим масло-шоколадную смесь на лимонную, и все готово! При этом определение теста для торта остается неизменным. В функциональном рецепте нет состояния – есть только определения промежуточных результатов. Кроме того, в нем нет последовательности шагов. Неважно, что вы сделаете вначале: подготовите тесто или масло-шоколадную смесь. Вы как пекарь можете выбрать порядок действий, который вам удобен!

1.2.2 От торта к программам

Но как же все это связано с программированием? Если представить, что ингредиенты, смеси и торты – это переменные, становится наглядно видно, что в «императивном» рецепте их значения изменяются со временем в результате выполнения инструкций. В функциональном рецепте значения переменных не меняются – они единожды создаются, и значение сохраняется навсегда. Шаги выполнения не зависят от текущего состояния переменных, а лишь от их наличия. Как только переменная потребуется, ее значение может быть вычислено. В функциональном рецепте некоторые шаги включают в себя информацию о том, как выполнять, например: «выпеченное в духовке при 200 °C в течение 25 минут». В программе такая информация будет представлена функциями, которые, в свою очередь, могут быть параметризованы. Это одно из фундаментальных свойств функционального программирования: функции являются полноценными объектами. Это значит, что их можно передавать как аргументы другим функциям!

Такой стиль программирования также называют *декларативным*, поскольку программа может быть интерпретирована как одно большое определение, собранное из других маленьких или даже рекурсивных определений. Этот подход облегчает создание повторно используемого кода, поскольку одно и то же определение можно использовать в разных контекстах. При таком подходе очевидно, что мы хотим избежать изменяемых переменных, поскольку не контролируем, как их значение будет изменяться во времени. Однако некоторые функциональные языки программирования допускают изменяемое состояние. Эти языки называются функциональными, но не *чистыми*. Для того чтобы функциональный язык можно было назвать чистым, нужно, чтобы функции не имели побочных эффектов.

Побочным эффектом называют любое взаимодействие с состоянием программы, определенным вне функции, во время нахождения внутри этой функции. Это означает, что в чистых языках программирования функции не могут:

- читать или записывать данные:
  - в файлы,
  - в сетевые сокеты,
  - в другие потоки и процессы,
  - в базы данных;
- иметь значения, которые могут изменяться между доступом к ним:
  - случайные числа,
  - информация о текущем времени;
- обращаться к памяти:
  - читать данные из памяти непосредственно,
  - записывать в память.

Конечно, даже в чистых языках есть способы выполнять такие операции, но они строго контролируются с помощью специальных абстракций и паттернов проектирования. В Haskell была выбрана парадигма *монадического программирования*, с которым мы познакомимся далее. Этот подход сильно отличается от того программирования, к которому вы, возможно, привыкли, и заставляет переосмыслить многие концепции, которые мы принимали как само собой разумеющееся. Интересно, что многие современные языки все активнее заимствуют элементы функционального программирования, например Java, TypeScript, Python и даже C++. Кроме того, многие языки, разрабатываемые крупными корпорациями, отходят от таких традиционных парадигм, как императивное или объектно ориентированное программирование, и вместо этого концентрируются на функциональном дизайне. F# (Microsoft) или Reason (Facebook) – два примера наследников функционального языка OCaml. Понимание концепций функционального программирования – очень важный шаг к тому, чтобы стать компетентным разработчиком.

1.2.3 Это все для простоты

Кроме того, в Haskell используется множество экзотических и непривычных концепций в качестве своих основных свойств. Вот некоторые из них:

- монадическое программирование;
- классы типов;
- ленивые вычисления;
- программная транзакционная память;
- обобщенные алгебраические типы.

Но зачем? Для чего? В программировании существует множество принципов, которые помогают создавать удобную для сопровождения архитектуру решений. Например: **KISS** (Keep It Simple, Stupid – «Не усложняй, глупышка») и **DRY** (Don't Repeat Yourself – «Не повторяйся»). Тем не менее даже тем разработчикам, кто старается соблюдать эти правила, зачастую приходится бороться с языком программирования, чтобы добиться нужного результата. Иногда приходится копировать код из одного класса в другой, чтобы избежать рефакторинга иерархии классов. Иногда не удается сделать процедуры короткими и простыми, поскольку язык не предоставляет удобных средств для более детальной декомпозиции процесса, который мы пытаемся описать. Декларативный подход Haskell в сочетании с его мощными возможностями абстрагирования не просто помогает соблюдать принципы чистого кода, но и активно поддерживает разработчика в создании удобного для сопровождения кода. Именно благодаря этому Haskell становится все более популярным в индустрии. Академические корни Haskell проявляются практически во всех его особенностях. Однако они были созданы не ради теоретических экспериментов: у каждой из них есть практическое применение, если понимать, как ими пользоваться.

Чистота Haskell делает тестирование функциональности чрезвычайно простым. Если спецификация функциональности ясна, тесты могут буквально повторять ее шаг за шагом. Нам не нужно писать адаптеры классов, чтобы интегрировать код с каркасом тестирования. Почти никогда не требуется подменять окружение, чтобы имитировать реальную производственную среду, и понять, насколько код покрыт тестами, обычно гораздо проще, чем в других языках. Что мы тестируем? Функции без побочных эффектов. Для наблюдения за поведением функции нужно лишь проверить, что она выводит.

Но оставим пока в стороне производственные приложения и тестопригодность. Вокруг Haskell сформировалось активное сообщество, которое неустанно разрабатывает инструменты и библиотеки для этого языка. Такой энтузиазм подпитывается не только интеллектуальным любопытством или стремлением создавать программы быстрее и эффективнее. Думаю, есть причина поважнее: Haskell – это весело! Стоит понять его основы, как Haskell становится волшебной палочкой, которая ждет, чтобы вы произнесли правильное заклинание и применили магию к цифровой реальности. Разработчики, сделавшие карьеру на написании кода на Haskell, часто упоминают стремление к корректности программ, простоте и применению теории к практике как основные источники мотивации. Но мне кажется, причина еще проще: это радость от процесса программирования. Возможность создавать сложные, изящные компоненты программного обеспечения, которые никогда не кажутся слишком

сложными и всегда могут быть упрощены или улучшены, – это особенное удовольствие. Проще говоря, это *радость от программирования*. Во всяком случае я и мои коллеги используем Haskell, по крайней мере отчасти, именно из-за этого.

1.3 Использование абстракции

Очевидно, что нет языков программирования, одинаково полезных во всех ситуациях. Haskell отличается высокой степенью абстракции. Он *переопределяет* само понятие высокоуровневого языка. Это означает, что исходный код, написанный разработчиком, находится очень далеко от машинных команд, исполняемых процессором. У этого подхода есть очевидные преимущества и недостатки.

1.3.1 Преимущества

Абстракции раскрываются во всем блеске, когда сложность становится чрезмерной, а в современном ПО очень много сложных мест. Haskell позволяет справиться с ними, что делает его основным кандидатом для применения в криптографических и распределенных системах. Сокращая количество внешних факторов и изменяемых данных, разработчик снижает вероятность возникновения ошибок и неожиданного поведения. Поэтому реализация протоколов безопасности и разработка логики веб-серверов не требует особых усилий. Декларативное программирование изобилует определениями, это замечательно, когда сама логика программы основана на определениях. Это относится к компиляторам, транспиляторам и программам преобразования файлов.

Однако Haskell – язык общего назначения. Это значит, что теоретически на нем можно написать практически любую программу. Для языка существует множество библиотек, активно применяемых в производственных приложениях. Haskell по-настоящему раскрывает свой потенциал при разработке программ, работающих с разнообразными источниками данных, так как его ключевая идея – *композиция* различных компонентов программ. В конце концов, целое всегда больше суммы своих частей.

Как новички в языке, мы можем задаться вопросом: какие программы можно написать на Haskell? Хорошим решением будет создание утилиты с понятными входными и выходными данными. Вспомните утилиты UNIX. Их философия проста: «Делай что-то одно, но делай *хорошо*». Такие программы обычно считывают входные данные, обрабатывают их и возвращают результат. Чистота Haskell делает его идеальным инструментом для таких задач. Даже если реальная задача оказывается сложной, например вычисление

статистики по пакету данных, зачастую ее можно легко смоделировать. Отличный пример подобного инструмента – *Pandoc*, программа с открытым исходным кодом, предназначенная для преобразования документов из одного формата в другой. Она может принимать документы, например, в форматах HTML, OpenDocument или LaTeX и преобразовывать их в другие форматы, скажем DocBook, EPUB, PDF.

Одной из отличительных черт Haskell является его рождение в академической, а не производственной среде. Неудивительно, что многие проекты на Haskell, написанные как любителями, так и профессионалами, посвящены созданию компиляторов, интерпретаторов и предметно-ориентированных языков. Что может быть увлекательнее, чем писать программы на языке, который создал кто-то другой? Конечно, создавать программы на языке, который создал сам! Haskell предоставляет возможность определять собственные операторы, что позволяет разработчику создавать предметно-ориентированные языки и использовать их внутри самого Haskell. Академическое наследие языка также заметно в большом количестве написанных на нем инструментов для автоматизированного доказательства теорем и автоматических рассуждений. Эти приложения можно использовать для построения математических доказательств и тестирования спецификаций программ.

Возможности Haskell по упрощению сложной архитектуры программного обеспечения делают его популярным выбором для серверных частей приложений, связанных с анализом данных или сложным управлением данными. Его используют такие крупные компании, как Facebook, Target, Barclays, Standard Chartered и NASA, в приложениях, интенсивно работающих с данными. И на то есть веские причины: код на Haskell часто проще читать, подвергать рефакторингу, а также тестировать и верифицировать. Если вам нужен «беспроигрышный вариант» при разработке ПО, то Haskell – отличный выбор.

Мне кажется, что Haskell прекрасно подходит для быстрого построения прототипов, поскольку на нем очень легко написать приложение всего лишь в несколько строк кода, а затем расширить его, зачастую даже без рефакторинга. Вы можете превратить однопоточную модель в многопоточную, изменив всего несколько строк кода. Часто есть возможность перейти на новую структуру конфигурационного файла, просто сменив синтаксический анализатор. Можно построить прототип приложения с вводом и выводом в файл, а затем, значительно позже, переключиться на получение данных из сети, не изменяя основных частей системы. Приложения на Haskell легко адаптируются вместе с изменением ваших потребностей.

1.3.2 Недостатки

Абстракции – это замечательно. Но Haskell настолько же далек от низкоуровневого кода, как космический челнок от Марианской впадины. И это существенный недостаток: у программиста никогда не будет полного контроля над тем, как исполняется его программа. Между вашими намерениями и «железом» стоит среда исполнения, которой практически невозможно управлять. Она управляет потоками, выделяет память, когда ей нужно, и запускает сборщик мусора по своему усмотрению. Конечно, с одной стороны, это упрощает программирование, но с другой – делает язык практически неприменимым в таких областях, как:

- системы реального времени;
- программы с большим объемом состояния, например видеоигры или мультимедийные приложения;
- драйверы устройств и операционные системы.

Не то чтобы такие приложения было невозможно написать на Haskell, но это, скорее всего, будет настоящим вызовом и потребует глубокого изучения технических аспектов Haskell, которые не описаны в этой книге.

1.4 Чему мы научились

Некоторые утверждают, что для изучения Haskell нужно знать математику. Мы постараемся разрушить этот стереотип, объяснив все основные концепции языка простым и понятным языком. Вам вообще не понадобится предварительное знание функционального программирования! Конечно, нужно иметь представление о процедурном или императивном программировании (например, на C, Java, Python) и понимать основы алгоритмов, структур данных, а также операционных и файловых систем. Идеальный читатель книги выглядит так:

- имеет хотя бы двухлетний опыт программирования;
- работал над различными (небольшими) проектами и знает, с какими проблемами можно столкнуться в реальных программах;
- обладает базовыми знаниями об операционных системах (особенно UNIX).

В отличие от многих других, эта книга не является кратким курсом по наиболее продвинутым идеям и методам Haskell, а знакомит с интересными, творческими и полезными проектами, показывая, как решать реальные задачи, возникающие при разработке реаль-

ных приложений. Haskell может показаться океаном незнакомых концепций, но эта книга не ставит целью погружение в темные глубины. После прочтения последней главы вы сможете, если захотите, взойти на борт подводной лодки и исследовать их самостоятельно. Это будет замечательное путешествие.

Эта книга устроена так, чтобы охватить широкое разнообразие проектов. Сначала мы будем писать простенькие программы для начинающих, такие как несложный (но умный) искусственный интеллект для игры в цепочку слов и утилита для красивого форматирования CSV-файлов с возможностью печати их в табличном виде и с дополнительными функциями, например поиском. Затем мы рассмотрим работу с данными, например с аудиофайлами и изображениями, будем изменять их в нашем приложении, создадим собственный синтезатор музыки и многопоточную библиотеку для обработки изображений. Конечно, ни одна книга не обходится без попытки эмулировать большие системы, поэтому мы создадим собственный микросервис – веб-сервер, способный в ответ на запросы выполнять действия и безопасно сохранять данные потокобезопасным способом! И при этом у нас будет возможность познакомиться с передовыми библиотеками Haskell в действии!

В основном мы сосредоточимся на написании приложений для UNIX-подобных систем, таких как Linux, BSD и macOS. Если вы хотите использовать Haskell в Windows, это также возможно – с помощью подсистемы Linux для Windows (WSL). Все проекты являются приложениями, запускаемыми из командной строки или терминала. Это осознанный выбор, поскольку мы тем самым упрощаем процесс разработки и можем сосредоточиться на минималистском интерфейсе взаимодействия с пользователем, не заботясь о программировании графического интерфейса.

После прочтения этой книги читатель сможет уверенно использовать Haskell для разработки реальных проектов, избегая распространенных ошибок, которые допускают начинающие. Кроме того, читатель будет знать, как использовать полученные знания и в других языках программирования.

Резюме

- Haskell – чистый функциональный язык, делающий акцент на безопасности и создании допускающего композицию кода.
- Безопасный код ограничивает использование опасных операций, что снижает количество ошибок, в т. ч. вызывающих неопределенное поведение, и таким образом более точно отражает намерения программиста.

- Haskell создан в академической среде, это видно из включения в него многочисленных результатов современного теоретического программирования.
- Haskell – компилируемый язык с автоматической сборкой мусора, поддерживающий параллелизм и конкурентность.
- Чистые функции и функции без побочных эффектов могут только принимать входные данные и выдавать результат.
- Функциональное и декларативное программирование сосредоточено на определениях и промежуточных результатах, а не на последовательности инструкций.
- Побочный эффект – это взаимодействие с состоянием программы вне функции во время ее исполнения.
- Декларативный подход Haskell в сочетании с абстракциями помогает следовать принципам чистого кодирования.
- Абстракции Haskell упрощают написание архитектуры сложного ПО и делают его популярным выбором при разработке серверного программного обеспечения для анализа данных и сложного управления данными.

2 Античное хранение секретов на современных машинах

https://t.me/it_boooks/2

Краткое содержание этой главы:

- как подготовить простой проект на Haskell и выполнить код;
- некоторые типы данных в Haskell и принципы работы функций;
- использование стражей и сопоставления с образцами для обработки различных случаев;
- работа с неизменяемыми данными и представление преобразований.

В предыдущей главе мы узнали, что такое Haskell, и обсудили специфические для него проектные решения. Мы также узнали об особенностях чистых функций и о структурировании чистых функциональных программ, таких, в частности, какие пишутся на Haskell. Понимать эти концепции необходимо, но просто распространяться о них толку мало – нужно попрактиковаться. Нельзя изучить теорию, не применив ее к чему-то. Так давайте займемся своим первым проектом!

Изучение любого языка программирования нужно с чего-то начать. Я мог бы наскучить вам математикой или предложить какое-нибудь сугубо академическое упражнение, но для этой книги выбрал темы повеселее. Кто-то назовет это забавой, но так далеко мы заходить не будем. Эта глава является в основном введением в синтаксис

Haskell и в порядок написания программ на этом языке. Поскольку принципы Haskell необычны (по сравнению с другими языками), я попросил бы вас постараться забыть некоторые знакомые паттерны. Изменяемые переменные? Статические данные? Императивное управление потоком? Выкиньте все это из головы, потому что мир, в который мы направляемся, не знает этой черной магии.

Это описание, пожалуй, чересчур образное, поэтому позвольте объясниться: при виде *чистого функционального* дизайна Haskell поначалу не мудрено и оробеть. Если у вас нет опыта работы с функциональными языками, излагаемые в этой главе концепции могут показаться странными. Это связано с тем, что дизайн такого рода языков сильно отличается от языков, которые вы использовали раньше и с которыми хорошо знакомы. Например, в Haskell нет циклов! Конечно, есть много способов обойти это ограничение. По чести говоря, нам и не нужны никакие циклы, но это значит, что многие вещи, касающиеся программирования, придется переосмыслить.

И хотя начало немного пугает, все же путешествие того стоит. Концепции Haskell универсальны, их можно реализовать почти в любом языке. Изучение чистого функционального программированиия – это изучение совершенно нового способа рассуждений о написании программ и контроля правильности написанного!

Мы начнем эту главу с демонстрации подготовки нового проекта с использованием компилятора Haskell в интерактивном режиме для создания прототипа кода. Затем мы познакомимся со многими фундаментальными типами данных Haskell и с тем, как они применяются для задания типов функций и выражений. Также приводится введение в написание функций на Haskell. После этого мы обсудим работу со списками и строками. Обсуждение функций мы дополним рассказом об использовании стражей и сопоставлении с образцами. И закончим главу описанием встроенной функции `map`.

2.1 Введение в Haskell

Современные компьютерные системы почти всегда зависят от эффективного использования внешней сети. Возможности безграничны: синхронизация по протоколам NTP или PTP, сохранение данных в распределенных базах данных NoSQL или просто просмотр видео котиков в разрешении 4K по вечерам. Но за это приходится дорого платить: необходимостью защищать наши персональные данные при передаче по открытым сетям. Не станем же мы громко разглашать данные своего банковского счета в лесу, надеясь, что нас никто не услышит!

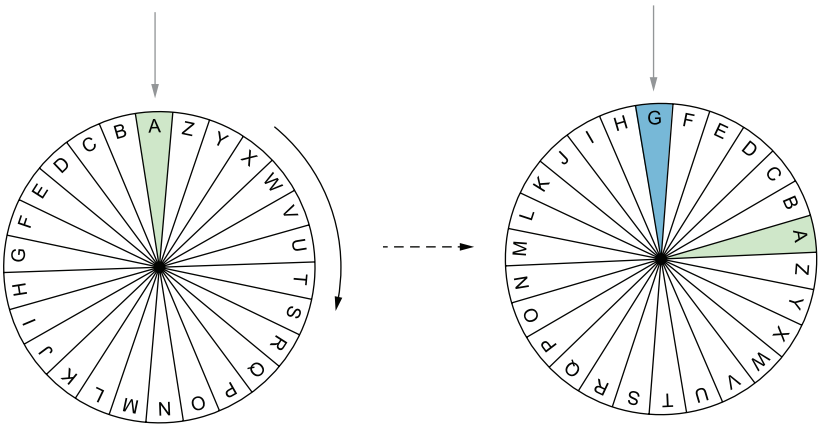
Как вы, наверное, уже догадались, решение проблемы – *шифрование*. Обычно для этого используется процедура, которая кодирует

данные секретным ключом, известным обеим сторонам. Криптография прочно вошла в нашу жизнь, и исследователи придумывают все новые и новые постоянно усложняющиеся алгоритмы и процедуры, чтобы сохранить секреты в секрете.

Схема шифрования, которую мы хотим рассмотреть, называется *шифром Цезаря*, это довольно примитивный способ хранения секретов. На практике он не применяется, но зато поможет нам узнать об основах функционального программирования.

2.1.1 Шифр Цезаря

Как работает этот античный алгоритм? Каждая буква алфавита заменяется другой буквой, отстоящей от исходной на некоторую величину смещения в алфавите. Мы можем представить себе алфавит в виде колеса, расположив все буквы по его окружности. Взяв какую-то букву и повернув колесо на определенное число позиций, мы получим новую букву. Эта идея иллюстрируется на рис. 2.1.



1. Начинаем с буквы и поворачиваем колесо на заданное количество позиций. В данном примере смещение равно 6

2. После поворота стрелка указывает на новый – «зашифрованный» – символ. Заметим, что колесо можно повернуть и на больший угол!

Рис. 2.1 Колесо подстановки букв

Преобразуя каждую букву сообщения, мы можем зашифровать его! Величина смещения в этом преобразовании является *ключом шифрования*, она должна быть известна обеим сторонам. Для расшифровки необходимо обратить процесс, восстановив исходное сообщение. Это показано на рис. 2.2, где короткое сообщение зашифровано со смещением 5.

Если шифруются только буквы латинского алфавита (причем для строчных и заглавных букв используются разные алфавиты), то наблюдается интересная симметрия. Когда смещение равно 13, шиф-

рование и дешифрирование аннулируют друг друга, т. е. для того и другого используется в точности один и тот же процесс. Этот частный случай шифра Цезаря называется *ROT13* и обыкновенно применяется для искажения слов, потому что его очень легко реализовать. В некоторые редакторы, например Vim и Emacs, ROT13 даже встроен! Vfa'g gung arng?

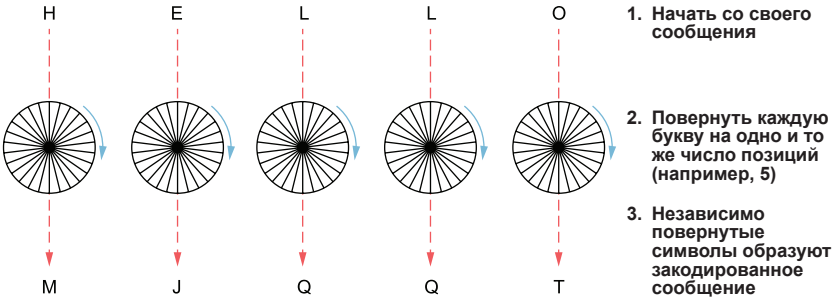


Рис. 2.2 Шифрование с помощью шифра Цезаря

> **ПРЕДУПРЕЖДЕНИЕ** Во имя любви к своим и чужим данным *никогда* не используйте этот алгоритм шифрования в приложениях!

2.1.2 Новый проект

Итак, создадим новый проект командой `stack`. Если вы еще не установили `stack`, обратитесь к приложению А. После того как комплект инструментов окажется на вашем компьютере, можете открыть командную строку, как принято в вашей операционной системе. В macOS для этого служит команда `iTerm`, в Windows – `cmd`, а в Linux возможностей гораздо больше. Затем перейдите в каталог, где будет создано рабочее пространство, и выполните команду

```
stack new caesar
```

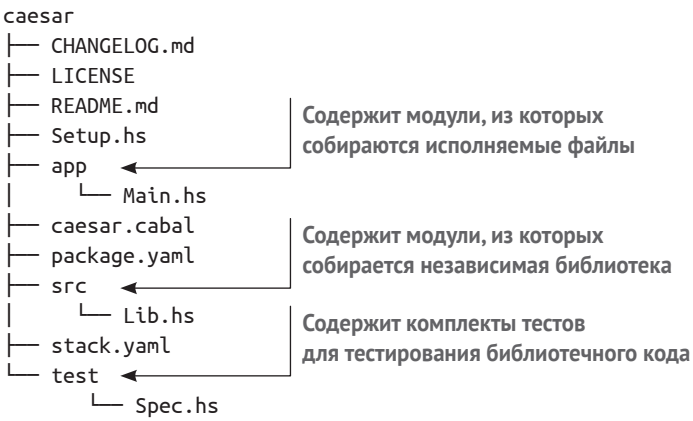
Будет создан новый каталог `caesar`, уже содержащий готовую структуру проекта Haskell.

Структура каталогов, созданная `stack`, показана в листинге 2.1. Мы видим файлы, содержащие метаданные, в частности журнал изменений, текст лицензии и файл README, в котором будут находиться пояснения, если проекту суждено быть выпущенным. Файлы `caesar.cabal`, `package.yaml` и `stack.yaml` нужны для конфигурирования зависимостей и параметров компиляции, но в этом проекте они нам не понадобятся.

И соль проекта: каталоги `app` и `src`. В каталоге `app` находятся модули, содержащие точку входа в программу, `main`. Именно здесь окажутся исполняемые файлы – как в языках C или Java. В каталоге `src`

хранится библиотечный код, т. е. определения всех модулей, типов и функций, которые могут быть организованы как автономная библиотека. Это позволяет использовать одну и ту же функциональность в разных исполняемых файлах или откомпилировать проект не как исполняемый файл, а как библиотеку. Последний каталог, `test`, можно использовать для поставки комплектов тестов вместе с библиотекой и исполняемым кодом. С ним мы разберемся в главе 9.

Листинг 2.1 Структура нового проекта, созданного stack



Для нас сейчас важен только файл `Lib.hs`. Он понадобится при написании функций и их использовании в интерпретируемом режиме компилятора Haskell. И раз уж зашла об этом речь, воспользуемся циклом чтения–выполнения–печати (read–eval–print loop – REPL), чтобы свести первое знакомство с синтаксисом Haskell. Выполните команду

```
stack repl
```

Она запускает `GHCi`, интерпретируемый режим компилятора Glasgow Haskell Compiler (GHC). Приглашение выглядит так:

```
ghci>
```

Если текущим является каталог проекта, созданный `stack`, то модули и определения автоматически загружаются в GHCi, и мы можем работать с ними. Давайте введем какой-нибудь код на Haskell! Начнем с простейшей арифметики:

```
ghci> a = 1 :: Int
ghci> b = 2 :: Int
ghci> c = a + b
ghci> c
3
```

Как видно, мы можем определить в своем коде *переменные*, связывающие значения с именами. Любая строка, введенная в REPL, вычисляется. Вычисление присваивания не порождает никакого результата, а вычисление переменной дает ее значение. Кроме того, мы видим, что у значений есть типы, которые в Haskell указываются после `::`.

> **ПРИМЕЧАНИЕ** При вводе в GHCi выражения вида `1+1` будет вычислено его значение `2`, но при этом выдается предупреждение о принятых по умолчанию ограничениях типа. Почти всегда такие предупреждения выдаются, когда в выражении не заданы типы. Чтобы подавить их, можно выполнить команду `:set -Wno-type-defaults`.

Пришло время поговорить о *функциях* в *функциональном программировании*, для чего нужно определить какие-нибудь функции. Их семантика сопоставима с семантикой функций в других языках программирования. Базовый синтаксис функций очень прост. За именем следует список параметров, за ним знак равенства (=) и определение. Рассмотрим пример:

```
square x = x * x
```

Здесь определена функция `square` с одним аргументом, который умножается сам на себя. Вызывается функция так:

```
square 2
```

Ни в определении, ни при вызове параметры не заключаются в скобки и не разделяются запятыми, как во многих других языках. Набрав код вызова функции `square 2` в REPL, мы заставляем интерпретатор вычислить функцию и напечатать результат. Введем в REPL определение:

```
ghci> :{
ghci| square :: Int -> Int
ghci| square x = x * x
ghci| :}
ghci> square 2
4
```

Начало ввода многострочной конструкции в GHCi

Конец ввода многострочной конструкции в GHCi

Тип в `square`, строго говоря, не обязателен, но если его опустить, будет выдано предупреждение. Этого достаточно, чтобы начать эксперименты с Haskell. Потратьте несколько минут в GHCi, чтобы освоиться.

REPL хорош для создания прототипов и рудиментарного тестирования кода, но может очень быстро стать неповоротливым, особенно при написании длинных функций. Потому-то мы в своем проекте

обратим внимание на Lib.hs, где и будет находиться большая часть реализации.

2.1.3 Первый модуль

Файл будет иметь очень простую структуру, позволяющую высветить наиболее важные аспекты *модуля* Haskell. Модули содержат наш код, состоящий из определений и реализаций. Они дают возможность структурировать проекты путем группировки тесно связанного кода. Вообще говоря, имя модуля должно совпадать с путем к файлу (за исключением модуля `Main`, зависящего от конфигурации проекта). В этом простом случае модуль находится на самом верхнем уровне каталога `src`, поэтому его имя совпадает с именем файла без суффикса. Кроме того, модуль может определять *список экспорта*, показывающий, какие определения должны быть видны другим модулям, поскольку одни модули могут импортировать другие. Файл выглядит так, как показано в следующем листинге.

Листинг 2.2 Пример простого модуля, созданного stack

```
module Lib                          <-- Определяет имя модуля
    ( someFunc                      <-- Определяет список экспорта
    ) where

                                    <-- Задает сигнатуру типа
someFunc :: IO ()
someFunc = putStrLn "someFunc       <-- Включает определение
```

У этого модуля только одна цель – служить иллюстрацией структуры. Мы видим единственное определение *действия ввода-вывода* с именем `someFunc`. Оно экспортируется с помощью списка экспорта, находящегося в скобках после имени модуля. Что такое действие ввода-вывода, мы узнаем в главе 3, а пока нам это не важно.

Основной способ импорта другого модуля – предложение `import`. Оно импортирует из модуля все определения, перечисленные в его списке экспорта (или вообще *все*, если список экспорта не задан). Это хорошо, если импортируется некоторый служебный модуль, предоставляющий уникальные функции, но в иных случаях может привести к конфликту имен. Чтобы избежать этого, мы можем предпослать имени функции имя модуля и точку, т. е. вместо `x` написать `ModuleName.x`.

Теперь обратимся к файлу `Main.hs`, чтобы понять, как импортируется модуль `Lib` и как выглядит «главный модуль». Этот модуль является точкой входа в программу, поскольку содержит функцию `main`. Сейчас нас не интересует, что она делает; можно считать, что это просто какая-то вызываемая функция. Эта функция *main* – первый код, вызываемый при выполнении нашей программы. Модуль `Main` содержит предложение `import` для импорта нашего модуля `Lib`.

Таким образом, `someFunc` будет частью модуля `Main`, потому что она встречается в списке экспорта модуля `Lib`. Код выглядит следующим образом:

Листинг 2.3 Простой главный модуль, созданный stack

```
module Main where

import Lib ◄——— Импортирует модуль

main :: IO () ◄——— Определяет действие main
main = someFunc
```

Этот проект будет целиком реализован в модуле `Lib` и протестирован с помощью GHCi, поэтому мы пока можем не обращать внимания на модуль `Main`. Чтобы исполняемый файл компилировался и IDE не выдавала странных ошибок, изменим модуль `Main`, отказавшись от импорта `Lib`, а в главное действие поместим просто печать `Hello World`. Тогда реализация пример такой вид:

```
module Main where

main :: IO ()
main = putStrLn "Hello World"
```

Вернемся к `Lib.hs` и немного ближе познакомимся с Haskell. Пока что можем удалить список экспорта и функцию `someFunc`. В модуле останется вот что:

```
module Lib where
```

Пусто и красиво – можно начинать с чистого листа! Пока мы определяем функции и записываем их в этот модуль, GHCi о них ничего не знает. Помимо определения функции, ее нужно экспортировать, включив в *список экспорта*, который указывается после имени модуля:

```
module Lib
  ( square,
  )
where

square :: Int -> Int
square x = x * x
```

После сохранения этого файла его можно загрузить. Для этого перейдите в каталог проекта и выполните команду `stack repl`.

> **ПРИМЕЧАНИЕ** Не забывайте добавлять функции и типы, которые хотите сделать видимыми другим модулям, в *список экспорта* модуля, содержащего определение. В противном

случае определение не будет найдено. Можно вместо этого экспортировать вообще все, не определяя список экспорта, но это считается дурным тоном и впоследствии приведет к неуправляемому коду.

Модуль будет загружен автоматически. После этого функции можно будет использовать в новом сеансе GHCi. Не закрывайте сеанс GHCi, мы будем создавать в нем прототип!

2.2 Типичные типы и фантастические функции

Чтобы правильно структурировать проект, мы должны подумать о том, с какими данными собираемся работать. Точнее, это означает, что нужно подумать о *типах* данных. В языках типа Haskell типы играют важнейшую роль. Как и некоторые другие языки, Haskell является *статически типизированным*, т. е. типы известны уже на этапе компиляции и не могут изменяться динамически. Кроме того, в Haskell применяется *вывод типов*. Если типы не указаны явно, то Haskell попытается вывести их из контекста, в котором используются функции и переменные. В большинстве случаев это работает безупречно, но иногда Haskell'у нужно указать типы явно. Механизм вывода типов позволяет нам записывать функции, как в сеансе GHCi, – *без* указания типов.

Однако считается хорошим стилем явно указывать типы в определениях, созданных на верхнем уровне модуля. Для этого используются *выражения типа*. Простое выражение типа выглядит так:

```
name :: type
```

В нем указывается тип имени (которое иногда называется *идентификатором*). Сами же имена отображаются на некоторые значения. И таким образом, мы говорим, что у значений должны быть определенные типы. Обычно сразу после выражения типа идет соответствующее определение, хотя это не строго обязательно. Дополнением к выражениям типа служат *семантические выражения*, из которых состоит программа. Мы уже встречались с семантическими выражениями: например, таковым является `x * x`. При вычислении такое выражение должно порождать значение. Если у семантического выражения имеется выражение типа, то порожденное значение будет иметь тип, определяемый выражением типа, а не самим семантическим выражением. В чем разница, мы разберемся позже, при изучении функций.

2.2.1 Типы на атомарном уровне

Мы познакомились с выражениями типа, но как выглядят сами типы? Простейшими являются *атомарные* типы. Именно они чаще всего встречаются в программах. Некоторые перечислены в табл. 2.1.

Таблица 2.1 Употребительные атомарные типы в Haskell

Имя	Описание	Примеры
`Bool`	Булевы значения истинности	`True` и `False`
`Char`	Литеры Юникода	`'a'`, `'A'`, `'1'`
`Integer`	Целые произвольного размера	`-1`, `9999999`
`Int`	Целые фиксированного размера	`-1`, `10`
`Float`	Вещественные с плавающей точкой одинарной точности	`3.1415`, `-1`, `1e2`
`Double`	Вещественные с плавающей точкой двойной точности	`1e-2`, `9.99999`

Как видно из примеров, некоторые значения могут принадлежать нескольким числовым типам. Встречая неоднозначность, Haskell привлекает свой механизм вывода типа и пытается понять, какой тип имелся в виду. Теперь, зная о типах, мы можем подумать о том, как нам реализовать шифр.

В этом нам поможет рис. 2.3. Поскольку каждый шаг поворота колеса – дискретный переход от предыдущего элемента к следующему, мы понимаем, что ключ шифрования должен иметь тип `Int`, поскольку нам совершенно ни к чему целые типа `Integer`, которые могут иметь произвольный размер. Шифруемое сообщение будет состоять только из букв, т. е. является коллекцией значений типа `Char`. Функция, которую мы создадим в этом проекте, должна уметь работать с этими типами.



Рис. 2.3 Колесо подстановки букв

Сделаем важное замечание об атомарных типах: сами такие типы являются выражениями типов! Поэтому мы можем присвоить переменной значение (или семантическое выражение) такого типа, например:

```
kilo :: Int
kilo = 1000
mega :: Int
mega = kilo ^ (2 :: Int)
nano :: Double
nano = 1e-9
```

Здесь мы определяем значения для трех префиксов единиц измерения. Присвоив идентификатору `kilo` тип `Int`, мы явно определили, что он целый, тогда как `nano` – число с плавающей точкой типа `Double`. Мы видим, что `mega` определена с помощью операции над `kilo`, а именно возведения в степень (`^`). Таким образом, она должна иметь тот же тип, что `kilo`, поскольку числовые операторы обычно принимают и порождают значения одного и того же типа.

2.2.2 Списки и кортежи

Простых атомарных типов недостаточно для конструирования полезных структур данных. В нашем случае необходимо что-то для представления коллекции букв. По счастью, Haskell предлагает и более сложные типы, которые мы рассмотрим ниже.

Первый из них – списки. Это необходимый строительный блок не только в Haskell, но и во многих других языках функционального программирования. Список – простая структура. Это *однородная* последовательность значений, т. е. все принадлежащие ей значения должны иметь один и тот же тип. Список записывается в виде последовательности разделенных запятыми значений, заключенной в квадратные скобки. На размер списка не налагается никаких ограничений, он может расти динамически. Приведем несколько примеров:

```
intList :: [Int]
intList = [1, 2, 3]

boolList :: [Bool]
boolList = [True]

floatList :: [Float]
floatList = []

doubleList :: [Double]
doubleList = [1, 1 + 1, 1 + 2]
```

Выражение типа для списка задается другим выражением типа, заключенным в квадратные скобки, похожим на то, как записываются значения списка. Поскольку все элементы списка должны иметь один и тот же тип, тип, определяемый выражением типа в скобках, является внутренним типом списка. Значения (или выражения, порождающие значения в списке) обязаны иметь одинаковые типы. В примерах выше мы видим, что даже пустой список (`floatList`) может иметь определенный тип. Тип списка говорит нам, что у любого значения из списка имеется определенный тип, но при этом не предполагается, что в списке вообще есть значения. Кроме того, `doubleList` дает пример списка, содержащего выражения, которые при последующем вычислении должны порождать значения типа `Double`. Здесь мы также видим вывод типа в действии. Список `doubleList` можно использовать в программах на Haskell, и компилятор правильно выведет, что все его элементы имеют тип `Double`. Но почему? `1` – выражение, которое вполне могло бы иметь тип `Int`. Откуда Haskell знает, что это значение должно быть типа `Double`? Из-за типа `doubleList`! Haskell знает, что его тип `[Double]`, а значит, все значения в списке должны иметь тип `Double`!

Списки – основной тип данных в функциональном программировании. В языках ML, Lisp (сокращение от *list interpreter* – интерпретатор списков) и производных от них без списков вообще никуда. Поэтому умение работать со списками в Haskell – навык, который можно перенести на многие другие языки, даже нефункциональные.

Еще один важный тип данных – *кортеж*. Как и список, кортеж представляет собой последовательность значений, но есть два отличия. Во-первых, длина кортежа фиксирована, а во-вторых, кортеж может содержать данные разных типов! Кортеж записывается в виде последовательности разделенных запятыми значений, заключенной в круглые скобки. Выражения типа также похожи на семантические выражения:

```
intPair :: (Int, Int)
intPair = (0, 1)

intFloatPair :: (Int, Float)
intFloatPair = (0, 1)

intCharDoubleTriple :: (Int, Char, Double)
intCharDoubleTriple = (1, 'a', 2)

septuple :: (Int, Float, Int, Float, Int, Float, Char)
septuple = (1, 1, 1, 1, 1, 1, 'a')
```

Как видим, составные типы можно строить из многих разных типов. На количество разных значений не накладывается ограниче-

ний, но их должно быть не меньше двух. Этот тип очень полезен для построения ассоциации между различными значениями.

2.2.3 Типы функций

Рассмотрим последний на данный момент (и пожалуй, самый важный) тип данных: *функции*. Да, именно так: функции – это тип данных! Более того, *функции являются значениями*! До сих пор мы записывали выражения и значения простых, преимущественно числовых типов. А вот о чем мы не говорили, так это почему приведенное выше определение `square` вообще дает функцию, которую можно вычислить. Мы создали значение, которое затем связали с именем `square`. Для начала посмотрим, как можно записать функцию, не давая ей имя. Такие функции называются *анонимными*, по очевидным причинам. Они записываются с помощью так называемой *лямбда-абстракции*:

```
\x -> x * x
```

Это наша прежняя функция `square`. Она принимает один аргумент `x` и порождает выражение `x * x`. С этой функцией можно связать имя:

```
square = \x -> x * x
```

И вот мы вернулись к нашей предыдущей функции! Прежняя функция `square` делала то же самое, только была записана чуть по-другому. Посмотрим, как выглядит тип этого значения:

```
square :: Int -> Int
square = \x -> x * x
```

Выражение типа очень похоже на то, как записывается само функциональное значение. Функция отображает переменную (например, `x`) в значение или выражение (например, `x * x`), и тип делает нечто аналогичное. Функция отображает `Int` в другое `Int`.

Это выражение типа для *унарной функции*, т. е. функции, принимающей один аргумент. Теперь посмотрим, как строится *бинарная функция*:

```
add :: Int -> Int -> Int
add = \x -> \y -> x + y
```

Если функция принимает переменную и порождает выражение, то это выражение само может быть функцией! Мы можем подчеркнуть это, добавив скобки вокруг «внутренней» функции:

```
add :: Int -> (Int -> Int)
add = \x -> (\y -> x + y)
```

Однако мало того, что писать функции таким образом было бы громоздко, их еще и читать было бы трудно, поэтому в нашем распоряжении имеется синтаксический сахар:

```
add :: Int -> Int -> Int
add = \x y -> x + y
```

Здесь мы видим лямбда-абстракцию с двумя аргументами, но можем использовать еще больше синтаксического сахара:

```
add :: Int -> Int -> Int
add x y = x + y
```

Вот так обычно и записываются функции в Haskell. Аргументы лямбда-абстракции можно просто расположить сбоку от имени функции. Во многих других языках говорят о *сигнатуре функции*, состоящей из типов аргументов и типа возвращаемого значения. Выражения типа функции в Haskell можно интерпретировать таким же образом. Первые типы относятся к аргументам, а последний – к «возвращаемому» значению функции. Это показано на рис. 2.4. Слово *возвращаемое* заключено в кавычки не зря. В функциональных программах функции не *возвращают*, а *вычисляют* значение.

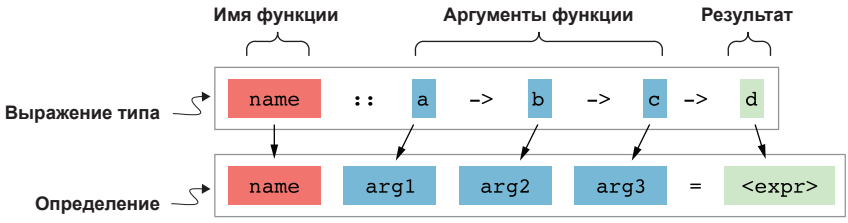


Рис. 2.4 Пример выражения типа для функции и соответствующее ему определение

2.2.4 Добавление типов к математическим операциям

Получив некоторое представление о том, как записываются типы, мы видим, что `Int -> Int` – допустимый тип для функции `square`. Но почему мы используем `Int`, а не какой-то другой числовой тип? В общем-то, без всякой причины, однако мы не можем свободно менять типы. Оператор `(*)` работает с любыми числовыми значениями, однако его наличие заставляет нас использовать один и тот же тип для обоих аргументов. Если бы мы передали функции значения разных типов, то столкнулись бы с ошибкой типизации:

```
ghci> a = 1 :: Int
ghci> b = 2 :: Double
ghci> a * b
```

Определяет переменную типа Int

Определяет переменную типа Double

```
<interactive>:4:5: error:
    • Couldn't match expected type 'Int' with actual type 'Double'
    • In the second argument of '(*)', namely 'b'
      In the expression: a * b
      In an equation for 'it': it = a * b
```

Описание ошибки вполне ожидаемо. Выражение `a * b` приводит к ошибке типизации, потому что вследствие использования `a` переменная `b` должна иметь тип `Int`, а это не так.

Возвращаясь к нашему алгоритму шифрования, мы уже можем предположить, что понадобится функция, которая будет поворачивать одну букву на указанное смещение, т. е. функция типа `Int -> Char -> Char`. Но поскольку строчные и заглавные буквы нужно обрабатывать по-разному, надлежит подумать о том, как рассуждать об используемых нами наборе символов и алфавите.

2.3 На помощь приходят алфавиты

Для построения алгоритма нужно лучше осмыслить работу шифра Цезаря. На рис. 2.5 видно, что мы *подставляем* одну букву вместо другой, из-за чего шифр называется *подстановочным*. Так как мы хотим этот шифр реализовать, то должны задать важный вопрос: как работать с буквами и алфавитами?

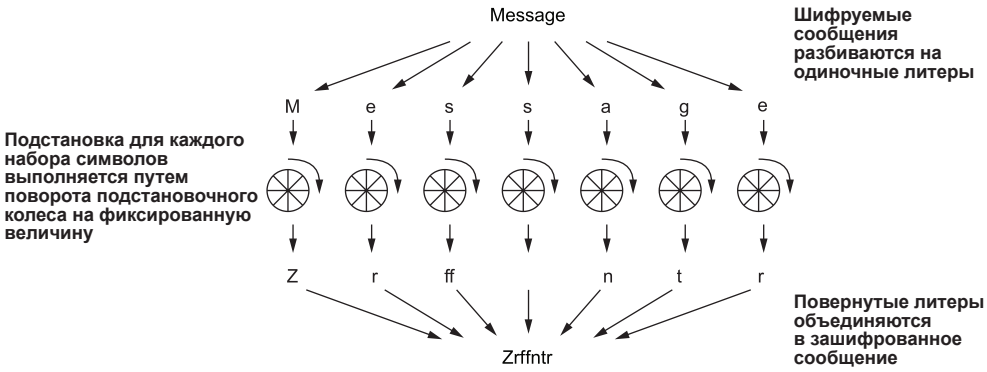


Рис. 2.5 Общий обзор шифра Цезаря

Мы уже начали размышлять о том, с какими *типами* работаем. Нам нужен способ задать смещение, которое может быть положительным или отрицательным, поэтому тип `Int`, вероятно, подойдет. Сообщение состоит из букв, поэтому для его представления мог бы подойти тип `String`.

2.3.1 Синонимы типов делают код понятнее

Позвольте, `String`? Мы с таким типом еще не сталкивались. Хотя нет – сталкивались, только неявно. Вот полное определение типа `String` в Haskell:

```
type String = [Char]
```

Ключевое слово `type` определяет простой синоним типа. Так что `String` – это просто синоним списка `Char`! Но как определить значение типа `String`?

```
string1 :: String
string1 = "Hello World"

string2 :: String
string2 = ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
```

В Haskell, как и во многих других языках, имеется явный синтаксис для строк, но это просто более удобный способ записать экземпляр списка литер. Поэтому обе строки в примере выше в точности одинаковы.

Зная, с какими типами мы работаем, можно начать думать о том, что необходимо для записи алгоритма. Каким бы ни было решение, написанная нами функция должна будет порождать подстановки одиночных литер. Чтобы упростить себе жизнь, сначала напишем несколько вспомогательных функций. Для шифрования сообщений очень полезно знать, с чем мы работаем. Мы хотим, чтобы функции сообщали нам, является ли элемент `String` заглавной буквой, строчной буквой, цифрой или ничем из вышеперечисленного. Можно предположить, что типы этих функций будут такими:

```
isUpper :: Char -> Bool

isLower :: Char -> Bool

isDigit :: Char -> Bool

isMisc :: Char -> Bool
```

Для реализации этих функций мы должны решить, принадлежит ли `Char` определенному множеству литер. Такие множества называются *алфавитами*. Стало быть, сначала мы хотим определить новый синоним типа для алфавита, который есть не что иное, как список литер:

```
type Alphabet = [Char]
```

Может показаться забавным, что типы строк и алфавиты в точности одинаковы. Однако каждый тип призван обозначать, для чего должны использоваться значения этого типа. `String` – это нечто, что можно напечатать на экране, и (как правило) он представляет понятный человеку текст. Наш новый тип для алфавитов говорит, что значение является коллекцией литер, которые будут использоваться в подстановочном шифре и могут быть повернуты, как показано на рис. 2.1.

Располагая типом для алфавитов, мы можем приступить к их определению. Вот пример алфавита строчных латинских букв:

```
lowerAlphabet :: Alphabet
lowerAlphabet = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
➥ 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
➥ 'x', 'y', 'z']
```

Но это длинно и неуклюже. По счастью, в Haskell есть сокращенный синтаксис для *диапазонов*:

```
ghci> [1 .. 10 :: Int]
[1,2,3,4,5,6,7,8,9,10]
ghci> [1, 5 .. 100 :: Int]
[1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61,65,69,73,77,81,85,89,93,97]
ghci> [0, -2 .. -10 :: Int]
[0,-2,-4,-6,-8,-10]
```

Как видим, поддерживается даже шаг изменения. Для нас важно, что диапазоны применимы также к типу `Char`. И это позволяет нам определить алфавит, как показано в листинге 2.4.

Листинг 2.4 Определения различных алфавитов латинских букв и цифр

```
lowerAlphabet :: Alphabet
lowerAlphabet = ['a' .. 'z']

upperAlphabet :: Alphabet
upperAlphabet = ['A' .. 'Z']

digits :: Alphabet
digits = ['0' .. '9']
```

Надеюсь, что сеанс GHCi у вас еще открыт, потому что теперь мы можем проинспектировать свои алфавиты! Сохранив эти определения в файле `Lib.hs`, вы можете перезагрузить сеанс GHCi. При этом все модули, которые могли измениться, будут заново импортированы, а значит, внесенные в файл изменения станут видны:

```
ghci> :reload
[3 of 3] Compiling Lib ( .../src/Lib.hs, interpreted )
Ok, three modules loaded.
ghci> lowerAlphabet
"abcdefghijklmnopqrstuvwxyz"
ghci> upperAlphabet
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
ghci> digits
"0123456789"
```

Перезагружает модули, после чего становятся доступными их новые версии

Если вы закрыли предыдущий сеанс, тоже не страшно. Просто сохраните алфавиты в файле `Lib.hs`, а затем снова выполните в каталоге проекта команду `stack repl`.

> **ПРЕДУПРЕЖДЕНИЕ** При использовании команды `:reload` в GHCi все локальные определения и значения будут потеряны! Эта команда работает как закрытие и повторное открытие GHCi.

Как видите, алфавиты содержат все те литеры, которые мы и хотели. Но почему они печатаются как значения типа `String`? Разве мы не задали специальный тип для алфавитов? Еще раз проверим это в GHCi, воспользовавшись командой `:type`:

```
ghci> :type lowerAlphabet
lowerAlphabet :: Alphabet
ghci> :type upperAlphabet
upperAlphabet :: Alphabet
ghci> :type digits
digits :: Alphabet
```

Да, это определенный нами тип. Однако поскольку алфавит – синоним списка символов, который почти всегда используется как значение типа `String`, Haskell по умолчанию так его и печатает. Синонимы типов взаимозаменяемы. Но тогда может возникнуть вопрос: зачем вообще создавать синонимы типов, если обрабатываются они одинаково? Короткий ответ – ради *удобочитаемости*. Если функция принимает в качестве аргумента тип `Alphabet`, то мы можем заключить, что она обращается с таким списком символов особым образом. В данном случае мы можем сделать вывод, что написали функцию самостоятельно и что эти значения рассматриваются в контексте шифра.

Тип FilePath

Такое использование типов ради удобочитаемости наблюдается и в стандартной библиотеке. Существует специальный тип для путей к файлам, он называется `FilePath`; но если посмотреть на него в GHCi, мы увидим кое-что интересное:

```
ghci> :info FilePath
type FilePath :: *
type FilePath = String
    -- Defined in 'GHC.IO'
```

Этот специальный тип – не что иное, как обычный тип `String` (а тот, в свою очередь, просто `[Char]`)! Но многие функции при этом становятся понятнее. Когда функции передается параметр типа `FilePath`, гораздо понятнее, что она собирается с ним делать. Такое использование синонимов типов может сделать ваш код намного яснее!

Для того чтобы узнать, как и где определены типы, мы можем воспользоваться командой `:info`.

ПРИМЕЧАНИЕ Вместо команд `:reload`, `:type` и `:info` можно использовать их сокращенные имена `:r`, `:t` и `:i` соответственно.

2.3.2 Виды букв

Имея алфавиты, мы теперь можем заняться написанием функций, которые решают, принадлежит ли литера некоторому алфавиту. Для этого служит *бинарная* функция `elem`, которая принимает два аргумента: сам элемент и список, принадлежность к которому мы хотим проверить. Она возвращает `True`, если элемент присутствует в списке, и `False` в противном случае. Разрешается применять эту функцию к нашим алфавитам, поскольку тип `Alphabet` – синоним `[Char]`. Вызывается она без затей:

```
ghci> elem 'a' lowerAlphabet
True
ghci> elem 'a' upperAlphabet
False
ghci> elem '1' lowerAlphabet
False
ghci> elem '1' digits
True
```

В рукаве у Haskell припрятан синтаксический трюк для работы с бинарными функциями, который делает их чуть более понятными:

```
ghci> 'a' `elem` lowerAlphabet
True
```

Любую бинарную функцию можно использовать в *инфиксной нотации*, добавив обратные апострофы! Теперь мы можем прочитать наш код вслух. Это выражение эквивалентно вопросу: «Является ли *a* элементом алфавита строчных букв?» Собрав все вместе, мы получим код, приведенный в листинге ниже.

Листинг 2.5 Вспомогательные функции для проверки принадлежности алфавиту

```
isLower :: Char -> Bool
isLower char = char `elem` lowerAlphabet

isUpper :: Char -> Bool
isUpper char = char `elem` upperAlphabet

isDigit :: Char -> Bool
isDigit char = char `elem` digits
```

Возвращает True, если литера является строчной буквой

Возвращает True, если литера является заглавной буквой

Возвращает True, если литера является цифрой

Протестируем эти функции в GHCi:

```
ghci> :r
[3 of 3] Compiling Lib (.../src/Lib.hs, interpreted )
Ok, three modules loaded.
ghci> isLower 'a'
True
ghci> isLower 'A'
False
ghci> isUpper 'A'
True
ghci> isDigit '5'
True
```

2.3.3 Логические комбинации

Но как определить, что литера относится к категории «прочие», т. е. не принадлежит ни одному алфавиту? Для этого имеются булевы операторы:

```
isMisc :: Char -> Bool
isMisc char = not (isUpper char || isLower char || isDigit char)
```

К числу булевых операторов относятся логические ИЛИ (`||`), И (`&&`) и НЕ (`not`). Их типы совпадают с ожидаемыми:

```
ghci> :type not
not :: Bool -> Bool
```

```
ghci> :type (||)
(||) :: Bool -> Bool -> Bool
ghci> :type (&&)
(&&) :: Bool -> Bool -> Bool
```

Скобки вокруг операторов нужны, чтобы они интерпретировались как функции, потому что именно таковыми они в Haskell и являются: функциями, записанными в инфиксной нотации. Можно было бы записать их и в префиксной нотации:

```
ghci> (||) False True
True
ghci> (+) 1 2
3
```

Возвращаясь к нашей функции `isMisc`, мы видим, что хотя она и правильна, обобщить ее будет нелегко. Предположим, что мы захотели добавить еще один алфавит. Тогда придется завести еще одну вспомогательную функцию, а ее результат добавить в дизъюнкцию. Этого мы не хотим. Так, может быть, полезнее проверять, что литера не является элементом всех имеющихся алфавитов? Тогда возникает вопрос, как алфавиты объединить. Для этого понадобится оператор дописывания в конец списка (`(++)`):

```
ghci> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
ghci> "Hello" ++ " " ++ "World!"
"Hello World!"
```

Заметим, что этот оператор работает и для значений типа `String`, потому что это просто списки! Мы можем использовать этот оператор для объединения алфавитов. Кроме того, Haskell предоставляет функцию, являющуюся отрицанием `elem`, – `notElem`! В сочетании эти функции дают код, приведенный в листинге ниже.

Листинг 2.6 Вспомогательная функция, которая устанавливает, что литера не является ни буквой, ни цифрой

```
isMisc :: Char -> Bool
isMisc char = char `notElem` lowerAlphabet ++ upperAlphabet ++ digits
```

Возвращает True, если литера не является элементом алфавитов строчных букв, заглавных букв и цифр

Ну вот, так-то лучше! Обобщить эту функцию на большее число алфавитов или литеры, не считающиеся прочими, теперь можно, просто добавив еще один список в выражение, создающее проверяемый список литер.

2.4 Вращение колеса

Разобравшись со вспомогательными функциями, можно подумать о реализации шифра. Мы хотим понять, как лаконично оформить различение случаев в нашем алгоритме.

Необходимо реализовать следующие шаги:

1. Получить индекс буквы в алфавите.
2. Получить букву алфавита с данным индексом.
3. Преобразовать `String` символ за символом.

2.4.1 Нахождение индекса элемента

Начнем с первой подзадачи: найти индекс элемента в списке. Для этого потребуется сделать две вещи: пробежаться по списку и следить за текущим индексом. Но есть подвох: данные в Haskell *неизменяемы*. Один раз присвоив значение, вы уже не сможете его изменить. Может показаться, что в GHCi это возможно, но на самом деле это нет так. «Изменяя» значение, вы его перезаписываете. Кроме того, в языке нет циклов. Тогда как же обойти список? В чистом функциональном программировании ответом служит *рекурсия*.

Разрешается определять функции в терминах их самих. В таком случае функция вызывает себя же. В этом и заключается идея рекурсии. Поскольку использовать циклы мы не можем, нужно придумать, как применять одно и то же определение снова и снова к разным частям данных.

Сначала посмотрим, как рекурсивно реализовать функцию, определяющую длину списка. Предположим, что существует способ разбить список на две части: первый элемент и остаток, тогда мы смогли бы вычислить длину списка как длину остатка плюс единица. После этого длина остатка вычисляется рекурсивно. Но как разбить структуру данных?

Для этого можно применить *сопоставление с образцом*, мощный языковой механизм, позволяющий писать определения функций для различных «образцов» данных. Сначала посмотрим, как реализовать функцию нахождения длины списка. Мы точно знаем длину пустого списка:

```
listLength [] = 0
```

Здесь мы говорим, что если на вход подается пустой список, то результат будет равен `0`. В данный момент это *частичная функция*, т. е. существуют такие входные данные, для которых она не определена. Это легко проверить:

```
ghci> listLength []
0
ghci> listLength [1,2,3]
*** Exception: .../src/Lib.hs:...: Non-exhaustive patterns in function
➥ listLength
```

Haskell сообщает нам, что не так. Существует вход, для которого образец не задан, т. е. множество образцов не исчерпывающее (non-exhaustive). Как сделать его исчерпывающим? Чтобы ответить на этот вопрос, нужно понимать, что у списков есть два *конструктора*. Конструкторы служат для построения списков и в данном случае называются `[]` и `(:)`. С `[]` мы уже знакомы, он строит пустой список. `(:)` – операция добавления в начало, которая принимает элемент и существующий список и добавляет элемент в начало списка.

Ниже приведены примеры эти конструкторов:

```
ghci> []
[]
ghci> 1 : []
[1]
ghci> 1 : [2, 3]
[1,2,3]
ghci> 1 : 2 : 3 : []
[1,2,3]
```

Эти конструкторы мы и используем для сопоставления с образцом. Конструктор `[]` уже встречается в нашем определении. Чтобы сопоставление с образцом было исчерпывающим, нужно добавить второй конструктор. И в итоге мы приходим к сопоставлению с образцом, которое часто можно видеть при работе со списками.

Листинг 2.7 Функция, демонстрирующая обычные приемы сопоставления с образцом для списков

```
listLength [] = 0      <-- Сопоставляется, если список пуст
listLength (x:xs) = ...  <-- Сопоставляется, если список построен
                             применением операции дописывания в начало
```

Может показаться странным, но в Haskell определения частичных функций для различных образцов могут иметь совершенно разное поведение.

> **ПРИМЕЧАНИЕ** Для списков два определенных выше образца исчерпывают все возможности!

В определении для случая `(x:xs)` мы используем `x` как переменную для первого элемента (головы), а `xs` – для остатка (хвоста)

списка, заданного в аргументе. На рис. 2.6 видно, как список можно разбить на две части. Заметим, что для одноэлементного списка `xs` была бы равна [].

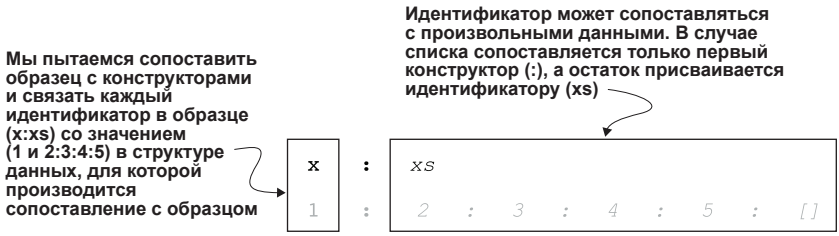


Рис. 2.6 Сопоставление с образцом для непустого списка

Так как же вычислить длину списка? Ясно, что длина пустого списка равна 0, но как быть со случаем, когда список состоит из элемента `x` и остатка `xs`? Как мы уже обсудили, результат должен быть равен длине `xs` + 1, т. к. `x` – элемент списка (поэтому его нужно учесть при подсчете), а оставшаяся часть длины определяется рекурсивным применением к остатку списка (т. е. к `xs`). Вот как это делается:

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

Здесь мы видим нашу первую *рекурсивную* функцию! Эта функция завершается, если переданный ей в качестве аргумента список конечен, поскольку это означает, что его определение кончается `[]`.

Искать индекс элемента также можно рекурсивно. Если список содержит хотя бы один элемент, то это довольно просто. Если первый же элемент списка является искомым, то мы нашли индекс, но что, если это не тот элемент, который мы ищем? Начнем так:

```
indexOf :: Char -> [Char] -> Int
indexOf ch (x : xs) = if x == ch then 0 else 1 + indexOf ch xs
```

Эта функция начинает работу с начала списка, используя сопоставление с образцом. Если первый элемент искомый, то индекс должен быть равен 0 (нумерация всегда начинается с нуля). Если поиск нужно продолжить, то мы производим его в хвосте текущего списка. Результат этого рекурсивного поиска увеличивается на 1, чтобы учесть пропущенный первый элемент. Разумеется, мы предполагаем, что рекурсивно полученный результат правилен. Но вот важный вопрос: что делать, когда список пуст? Тогда правильного результата нет! Следовательно, мы явно объявляем такой случай *неопределенным*. Эта реализация показана в листинге ниже.

Листинг 2.8 Функция вычисления индекса литеры в списке литер

```
indexOf :: Char -> [Char] -> Int
indexOf ch [] = undefined
indexOf ch (x : xs) = if x == ch then 0 else 1 + indexOf ch xs
```

Возбуждает исключение, когда входной список пуст или достигнут конец рекурсии

Рекурсивно вычисляет индекс элемента в списке

Однако при таком подходе мы должны быть уверены, что искомый элемент присутствует в списке. В противном случае программа завершится аварийно:

```
ghci> indexOf 'a' lowerAlphabet
0
ghci> indexOf 'f' lowerAlphabet
5
ghci> indexOf 'f' upperAlphabet
*** Exception: Prelude.undefined
CallStack (from HasCallStack):
  error, called at libraries/base/GHC/Err.hs:75:14 in base:GHC.Err
  undefined, called at .../src/Lib.hs:... in main:Lib
```

> **ПРИМЕЧАНИЕ** Явное использование `undefined` – очень дурной стиль. В главе 3 мы узнаем о других способах решения таких проблем. Для возврата индекса элемента в списке следует пользоваться функцией `elemIndex` из модуля `Data.List`.

Поскольку мы контролируем использование этой функции, пока с этим можно смириться. Тем не менее нужно сделать более очевидным, что функция должна использоваться только для алфавита, а не для произвольного списка литер, поэтому изменим ее тип:

```
indexOf :: Char -> Alphabet -> Int
```

2.4.2 Нахождение элемента по индексу

Находить индекс элемента в списке мы теперь умеем, но еще нужна функция, которая возвращает элемент с данным индексом. По счастью, такая функция существует и называется `(!!)`. В инфиксной нотации ее можно использовать так:

```
ghci> [1,2,3] !! 0
1
ghci> [1,2,3] !! 2
3
ghci> [1,2,3] !! 3
*** Exception: Prelude.!!: index too large
```

Упражнение. Написать функцию (!!), используя сопоставление с образцом

Попробуйте написать функцию `(!!)` самостоятельно, используя сопоставление с образцом и рекурсию. Если индекс слишком велик или отрицателен, то можете использовать `undefined`, чтобы заставить функцию возбудить исключение.

Как и в функции поиска индекса элемента, в этой функции есть проблема: не все входные данные порождают корректный результат. Поэтому использовать обе функции нужно с осторожностью! Однако теперь мы можем написать функции, которые выполняют показанный на рис. 2.1 поворот для алфавитов строчных и заглавных букв! Если задана литера и смещение в виде целого числа, то мы можем получить соответствующую литеру из алфавита:

```
upperRot :: Int -> Char -> Char
upperRot n ch = upperAlphabet !! ((indexOf ch upperAlphabet + n) `mod` 26)

lowerRot :: Int -> Char -> Char
lowerRot n ch = lowerAlphabet !! ((indexOf ch lowerAlphabet + n) `mod` 26)
```

Мы вычисляем новый индекс путем прибавления смещения к индексу старой литеры. Чтобы получить эффект колеса, новый индекс вычисляется по модулю 26 (длина алфавита) с помощью функции `mod`. Как легко убедиться, это работает даже для отрицательных индексов:

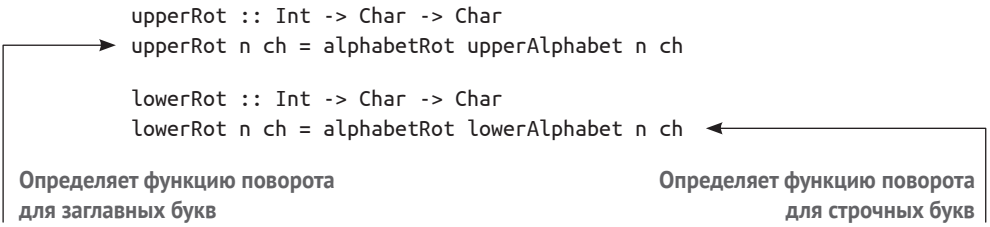
```
ghci> upperRot 3 'Z'
'C'
ghci> upperRot (-3) 'C'
'Z'
```

Однако функции `upperRot` и `lowerRot`, по сути дела, совпадают. Определение можно сделать более общим, добавив еще одну вспомогательную функцию, которая получает алфавит в качестве аргумента. Окончательный код показан ниже.

Листинг 2.9 Функции поворота для общего алфавита, а также для алфавитов строчных и заглавных букв

```
alphabetRot :: Alphabet -> Int -> Char -> Char
alphabetRot alphabet n ch =
  alphabet !! ((indexOf ch alphabet + n) `mod` length alphabet)
```

Определяет общий поворот для произвольного алфавита



Мы пришли к весьма общему определению функции поворота для произвольного алфавита.

2.4.3 Охрана потока управления

Теперь мы можем написать общую функцию, которая преобразует произвольную литеру с помощью наших вспомогательных функций. Мы хотели бы либо вообще не преобразовывать литеру (например, цифры или пробелы), либо использовать одну из ранее определенных функций поворота – в зависимости от аргумента. Поэтому необходимо как-то различать случаи. Для этого служит предложение *if-then-else*. Оно есть почти в любом языке программирования и применяется для описания *потока управления* на основе предикатов:

```
ghci> x = 10 :: Int
ghci> if x > 5 then "x is larger than 5" else "x is not larger than 5"
"x is larger than 5"
```

Однако в Haskell эта конструкция работает иначе, чем во многих других языках: невозможно иметь `if … then` *без* `else`. Дело в том, что if-then-else не отличается от любого другого выражения – оно должно вычислять значение. Но для `if` без `else` может оказаться, что никакое значение не вычислено, поэтому такая конструкция запрещена. Напишем функцию, которая вызывает подходящую функцию поворота в зависимости от входа:

```
rotChar :: Int -> Char -> Char
rotChar n ch =
  if isLower ch
    then lowerRot n ch
    else
      if isUpper ch
        then upperRot n ch
        else ch
```

Эта функция корректна, но читать ее трудно. У вложенных предложений if-then-else много недостатков. Во-первых, их становится трудно читать, когда выражения внутри if-then-else длинные и сложные. Во-вторых, попросту невозможно добавлять все новые и новые

случаи, если их много. И наконец, нужно еще исхитриться, чтобы придумать удобочитаемый стиль для записи такого кода. К счастью, конструкция if-then-else в Haskell встречается редко, потому что есть подход гораздо лучше: *стражи*!

Прежде чем вдаваться в детали, посмотрим, как будет выглядеть наша функция после перехода на стражи.

Листинг 2.10 Функция поворота для произвольных литер

```
rotChar :: Int -> Char -> Char
rotChar n ch
  | isLower ch = lowerRot n ch
  | isUpper ch = upperRot n ch
  | otherwise = ch
```

Проверяет, является ли аргумент строчной буквой, и если да, то применяет к нему функцию поворота строчных букв

То же, что и выше, но для заглавных букв

Оставляет литеру без изменения, если это не буква

Стражи дают элегантный способ задания предикатов и соответствующих им определений, очень похожий на сопоставление с образцом! Однако сопоставление производится не с образцами, а с булевыми выражениями. В общем случае в роли стража может выступать *произвольное* булево выражение.

> **ПРИМЕЧАНИЕ** Применение стражей и сопоставление с образцом производятся в определенном порядке. Если одна операция расположена в коде раньше другой, то и выполняться она будет раньше, поэтому варианты, которые сопоставляются всегда, должны находиться в конце.

Но что такое `otherwise`? Забавно, но это синоним `True`! Поскольку (в большинстве случаев) мы хотим, чтобы функция была определена для всех входных значений, последним стражем всегда должен быть `otherwise`, потому что это *случай по умолчанию*, улавливающий все, что не сопоставилось перед ним.

Протестируем нашу реализацию:

```
ghci> rotChar 13 'A'
'N'
ghci> rotChar 13 'a'
'n'
ghci> rotChar (-1) 'A'
'Z'
ghci> rotChar 10 '1'
'1'
ghci> rotChar 10 ' '
' '
```

2.5 Преобразование строки

Теперь, когда мы умеем поворачивать одиночные литеры, осталось только понять, как проделать это для всех литер в заданной строке `String`. Поскольку `String` – это просто список литер, мы должны применить нашу функцию `rotChar` к каждому элементу этого списка. Но как изменить что-то в списке, если все данные *неизменяемы*? Легко: мы не можем изменять значения, но можем создавать новые! С этим паттерном мы будем часто встречаться при написании чистого функционального кода. Вместо того чтобы модифицировать старый список, мы должны создать новый. Для создания функции, реализующей шифр Цезаря, можно снова воспользоваться сопоставлением с образцом:

```
caesar :: Int -> String -> String
caesar n [] = []
caesar n (x : xs) = rotChar n x : caesar n xs
```

Пустой список остается пустым, а для списка с первым элементом `x` и остатком `xs` создается новый список, который начинается с преобразованного `x`, а хвостом является результат преобразования `xs`. Эта функция и есть наша реализация шифра:

```
ghci> caesar 13 "Caesars Cipher!"
"Pnrfnef Pvcure!"
ghci> caesar 13 (caesar 13 "Caesars Cipher!")
"Caesars Cipher!"
```

Посмотрим внимательнее, чего мы пытались достичь функцией `caesar`. Мы хотели применить некоторое преобразование (в данном случае `rotChar`) к списку, но, быть может, эту идею можно обобщить?

2.5.1 Отображение высшего порядка

На практике такого рода преобразования очень часто встречаются в функциональных программах, поэтому было бы неплохо продолжить исследование этой операции. Поэлементное преобразование списка можно рассматривать как *отображение* одного значения в другое, без изменения порядка элементов. Графически это изображено на рис. 2.7. Конечно, эта логика применима к любым значениям и любому преобразованию.

Если мы хотим обобщить функцию `caesar` на общую функцию преобразования, то должны будем передать функцию в качестве аргумента. Напомним, что функции – это просто значения, поэтому мы можем использовать их и в качестве аргументов тоже!

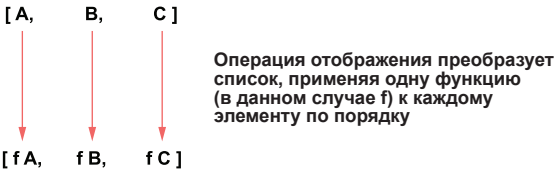


Рис. 2.7 Отображение значений

Глядя на функцию, мы видим, что аргумент `n` в общем случае не нужен, потому что он относится к конкретной функции `rotChar`. И кстати, `rotChar` нужно заменить аргументом, который мы назовем просто `fun`. Этот аргумент обозначает функцию, потребляющую один элемент из списка и возвращающую новый элемент, который займет его место. В итоге получаем следующую функцию:

```
transform fun [] = []
transform fun (x : xs) = fun x : transform fun xs
```

О, это *могущественная* функция! Следующие примеры демонстрируют, на что она способна:

```
ghci> transform (\x -> x + 1) [1..10] :: [Int]
[2,3,4,5,6,7,8,9,10,11]
ghci> transform (\x -> x > 5) [1..10] :: [Bool]
[False,False,False,False,False,True,True,True,True,True]
ghci> transform (\x -> if x == 'e' then 'u' else x) "Hello World!"
"Hullo World!"
```

Если данные представлены в виде последовательности значений в списке, то эту функцию можно использовать для изменения данных, не занимаясь разбивкой списка на части и обратной сборкой. Это прекрасный пример семейства функций, которые определяют только, *как* работать со структурой данных, и параметризуются другой функцией. В общем случае функции, принимающие другие функции в качестве аргументов, называются *функциями высшего порядка*. Они составляют неотъемлемую часть функционального программирования, поскольку дают общие описания алгоритмов, которые впоследствии будут использованы в конкретных определениях.

2.5.2 Параметризация типов

А вот что действительно интересно: как будет выглядеть тип столь общей функции? Он должен быть таким же общим, как определение. Инспекция ее типа в GHCi показывает:

```
ghci> :type transform
transform :: (t -> a) -> [t] -> [a]
```

Эти типы выглядят странно. Что за `t`? Что за `a`? Здесь мы видим пример *параметрического полиморфизма*. Типы в этой функции не фиксированы, а являются параметрами, т. е. мы можем подставить произвольные типы вместо `t` и `a`. Но при этом мы должны подставлять одинаковые типы вместо *всех* вхождений соответствующих параметров. Так, если мы собираемся заменить `t` конкретным типом, то должны будем заменить каждое вхождение `t` в наше выражение типа. Впрочем, `t` и `a` могут быть разными типами (хотя и не обязаны).

> **ПРИМЕЧАНИЕ** Имена параметров типов несущественны. `t` можно было бы назвать `b`, а `a` – `foo`.

Что именно говорит нам это выражение типа? Что `transform` имеет два аргумента, первый из которых – функция, принимающая `t` в качестве аргумента и возвращающая значение типа `a`. Второй аргумент `transform` – список значений типа `t`. А возвращает эта функция список значений типа `a`. Допустим, у нас есть такая функция:

```
f :: Int -> String
```

Если мы передадим ее в качестве первого аргумента `transform`, то вместо `t` будет подставлен `Int`, а вместо `a` – `String`, т. е. второй аргумент будет иметь тип `[Int]`, а возвращаемое значение – тип `[String]`:

```
ghci> :{
ghci| f :: Int -> String
ghci| f n = show n
ghci| :}
ghci> g xs = transform f xs
ghci> :t g
g :: [Int] -> [String]
```

Здесь мы также можем наблюдать работы механизма вывода типов в Haskell. Подстановку, выполненную нами в голове списка, мог бы произвести и компилятор! Такой вид полиморфизма присутствует почти в каждой функции и типе данных, которые встретятся нам на страницах этой книги, так что познакомиться с ним необходимо. Это одна из самых базовых идей, применяемых при написании абстрактного, допускающего композицию и общего кода.

> **ПРИМЕЧАНИЕ** Параметрический полиморфизм можно сравнить с концепцией обобщенных типов во многих других языках, например в Java и Go. В нашем случае переменные типа были бы обобщенными типами с абсолютно произвольным интерфейсом. В главе 5 мы узнаем о том, как ограничить природу переменных типов, когда будем обсуждать классы типов.

Полезность функций высшего порядка зависит от такого полиморфизма. Мы не хотим снова и снова писать одну и ту же функцию для разных типов. Напротив, мы хотим найти самый общий из возможных типов, чтобы повторно использовать функции многократно. Написанная нами функция `transform` – очень общий и важный строительный блок для создания алгоритмов на списках.

2.5.3 Завершенный шифр

На самом деле эта функция `transform` настолько важна, что уже встроена в Haskell и является одной из самых важных функций, о которых мы узнаем в этой книге; она называется `map`. Эта функция получает в качестве первого аргумента функцию для преобразования одного элемента, а в качестве второго – список. Поскольку наш шифр должен выполнять поворот каждой литеры в строке, то для его описания мы можем воспользоваться функцией `map` в своей функции `caesar`:

Листинг 2.11 Функции, которые выполняют шифры Цезаря и ROT13

```
caesar :: Int -> String -> String
caesar n message = map (\ch -> rotChar n ch) message

rot13 :: String -> String
rot13 message = caesar 13 message
```

Определяет шифр Цезаря, применяя функцию поворота к каждой литере сообщения

Выполняет шифрование методом ROT13, применяя шифр Цезаря с фиксированным смещением 13

Вот мы и добрались до конца! Разумеется, шифр параметризован смещением, определяющим поворот литер. Таким образом, частный случай ROT13 легко можно получить из самого шифра Цезаря. И теперь мы можем шифровать свою переписку, как делали римляне!

Упражнение. Шифр ROT135

Шифр ROT13 на загляденье симметричен, однако нам пришлось опустить поворот цифр, потому что иначе симметрия пропала бы, т. к. размеры алфавитов различны. Однако поворот на 5 позиций восстанавливает симметрию для цифр. Ваша задача – объединить алгоритмы ROT13 и ROT5 (для цифр) в комбинированный шифр ROT13.5, чтобы можно было симметрично кодировать буквы латиницы и цифры!

Напишите функцию `rot135`, выполняющую такое кодирование!

Убедитесь, что двукратное применение этой функции приводит сообщение к исходному виду:

```
ghci> rot135 "Hello 1 2 3 ... 7 8 9"
"Uryyb 6 7 8 ... 2 3 4"
ghci> rot135 (rot135 "Hello 1 2 3 ... 7 8 9")
"Hello 1 2 3 ... 7 8 9"
```

И вот мы имеем «абсолютно безопасную» схему шифрования. Оглянемся назад и вспомним, что мы сделали. Сначала мы создали синоним типа `Alphabet` и определили константы для строчных букв, заглавных букв и цифр, чтобы было понятно, о каких данных идет речь. Затем мы написали функции, которые определяют, принадлежит ли литера (`Char`) такому алфавиту. Пользуясь этими определениями, мы написали функцию, которая выполняет поворот литер алфавита, и реализовали шифр Цезаря.

Упражнение. Частотный анализ строк

ROT13 и шифр Цезаря – небезопасный способ «зашифровать» сообщение. Это легко показать с помощью частотного анализа. В английском языке некоторые буквы встречаются чаще других (например, e встречается гораздо чаще всех остальных букв). Напишите функцию, которая подсчитывает количество вхождений литеры в строку:

```
count :: Char -> String -> Int
```

Затем воспользуйтесь этой функцией для выполнения частотного анализа открытого и «зашифрованного» текста. Для этого примените функцию подсчета к каждой букве определенных ранее алфавитов. Можете ли вы придумать схему дешифрирования шифра Цезаря на основе наилучшей догадки?

Мы определили синоним типа для алфавитов, чтобы сделать код немного понятнее. Для определения алфавитов строчных букв, заглавных букв и цифр мы воспользовались диапазонами литер. На основе этих определений мы написали вспомогательные функции, которые определяют, принадлежит ли символ одному из этих алфавитов. Кроме того, мы написали функцию, которая находит индекс литеры в алфавите, и функцию, которая поворачивает литеры алфавита на заданное смещение. Объединив эти функции, мы написали функцию, которая умеет поворачивать произвольную литеру, определив предварительно, какому алфавиту она принадлежит и как ее следует повернуть. Воспользовавшись функцией `map`, мы затем реализовали шифр Цезаря на основе этой функции, а затем и шифр ROT13.

Резюме

- Со всяким выражением Haskell, результатом вычисления которого является значение, ассоциировано выражение типа. Общепринято и считается хорошей практикой явно обозначать их.
- Ключевое слово `type` можно использовать для создания синонимов типов, что позволяет сделать типы более выразительными и самодокументированными.
- Команды `:type`, `:info` и `:reload` в GHCi (сокращенно `:t`, `:i` и `:r`) позволяют получить информацию о функциях и значениях, а также перезагрузить все модули, загруженные в GHCi.
- Любую именованную бинарную функцию можно записать в инфиксной нотации, заключив имя функции в обратные апострофы `` ` ``.
- Сопоставление с образцом – способ деконструировать структуру данных и предоставить подходящие определения функций для разных образцов.
- При определении рекурсивных функций мы предполагаем, что частичные результаты правильны, потому что определение функции в целом правильно, только если правилен каждый шаг рекурсии.
- Стражи – это лаконичный способ различить случаи в зависимости от булевых предикатов. То есть это простая замена конструкции if-then-else.
- Функция `map` используется для того, чтобы применить некоторую функцию к каждому элементу списка по порядку.
- Параметрический полиморфизм позволяет создавать общие функции, применимые к различным типам.

3 Каждая строка считается

Краткое содержание этой главы:

- реализация побочных эффектов и ввода-вывода в Haskell;
- использование свободного от побочных эффектов, чистого кода внутри нечистого кода;
- взаимодействие с операционными системами;
- включение анализатора аргументов командной строки в программу.

В предыдущей главе мы узнали, как строить алгоритмы преобразования строк, и изучили основы программирования на Haskell; однако протестировать функциональность мы смогли только в GHCi. Теперь мы хотим перенести внимание на написание первой настоящей программы, которую можно будет вызывать из командной строки.

В Unix-подобных системах при использовании командной строки или написании автоматизированных скриптов для решения конкретных задач, например поиска или преобразования данных, часто используются конвейеры, состоящие из нескольких программ. Для реализации межпроцессных взаимодействий между процессами передаются сообщения в виде текстовых потоков. Обычно для этого применяется символ канала |. Эта идея иллюстрируется на рис. 3.1.

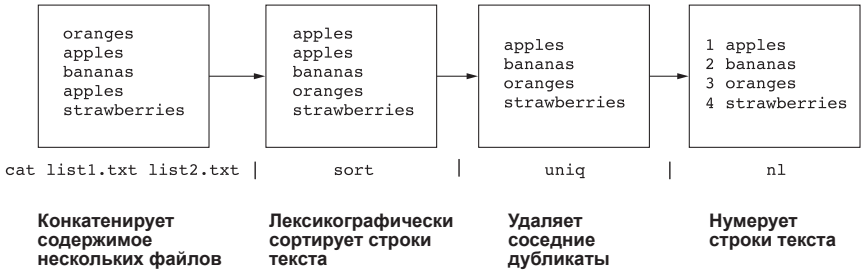


Рис. 3.1 Пример преобразования данных с помощью конвейера, состоящего из команд оболочки

Чтобы все это работало, в UNIX существует много инструментов, выполняющих элементарные задачи, которые, будучи соединены в конвейер, решают более сложную задачу. Один из таких инструментов – программа `nl`, имя которой образовано первыми буквами *number lines* (пронумеровать строки). Ее задача проста – напечатать файл, снабдив каждую строку порядковым номером. По сути дела, она создает листинги из произвольного текста. Вот пример:

```
shell $ nl testFile.txt
    1 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas a
    2 mattis nisi, eleifend auctor erat. Vivamus nisl sapien, suscipit non
    3 gravida sed, mattis a velit. Fusce non iaculis urna, a volutpat leo.
    4 Ut non mauris vel massa tincidunt mattis. Sed eu viverra lectus.
    5 Donec pulvinar justo quis condimentum suscipit. Donec vel odio eu
    6 odio maximus consequat at at tellus. Ut placerat suscipit vulputate.
    7 Donec eu eleifend massa, et aliquet ipsum.
    8 Mauris eget massa a tellus tristique consequat. Nunc tempus sit amet
    9 est sit amet malesuada. Curabitur ultrices in leo vitae tristique.
   10 Suspendisse potenti. Nam dui risus, gravida eu scelerisque sit amet,
   11 tincidunt id neque. In sit amet purus gravida, venenatis lectus sit
   12 amet, gravida massa. In iaculis commodo massa, in viverra est mollis
   13 et. Nunc vehicula felis a vestibulum egestas. Phasellus eu libero sed
   14 odio facilisis feugiat id quis velit. Proin a ex dapibus, lacinia dui
   15 at, vehicula ipsum.
```

Эту программу можно использовать разными способами. Например, пронумеровать строки файла, поискать в результате определенные строки, а затем отрезать сами строки, оставив только номера. В итоге получатся номера строк, содержащих искомый текст. Это можно использовать для поиска ссылок в исходном коде. Другой пример: создать список записей данных, отсортировать его по некоторому предикату, а затем пронумеровать строки. Получится листинг, показывающий элементы по порядку вместе с их индексами.

Почему бы нам не поупражняться, написав этот скромный инструмент самостоятельно на Haskell? Конечно, было бы хорошо остаться в уютном мирке чистых функций и простых определений, но нам

как программистам гораздо полезнее взяться за интересную задачу выполнения *ввода-вывода* на языке, знаменитом своей парадигмой чистого программирования. Как совместить свободные от побочных эффектов функции с побочными эффектами, неотъемлемыми от ввода и вывода? Этот вопрос нужно задать и ответить на него на ранних этапах нашего путешествия!

Конечно, мы рассматриваем эту задачу только как тренировочное упражнение. Утилита `nl` включает много режимов форматирования, которые мы повторять не будем. Однако ей не хватает некоторых полезных возможностей, например обратной нумерации, а мы ее добавим. Итак, сделаем первые шаги по пути создания реальных программ, написав свой инструмент нумерации строк файла!

Мы начнем эту главу знакомством с *do-нотацией* и обсуждением ввода и вывода в Haskell. Затем научимся интегрировать синтаксис и функции, о которых узнали во второй главе, с этой новой нотацией. Узнав о взаимодействии с окружением, созданным операционной системой, мы начнем строить фундамент для своего инструмента и завершим его создание в главе 4.

3.1 Общение с внешним миром

Первый очевидный вызов, брошенный нам, связан с чистым функциональным дизайном Haskell. Нашим функциям запрещено иметь побочные эффекты или как-то взаимодействовать с внешним окружением. Результат функции может зависеть *только* от ее аргументов и больше ни от чего. Следовательно, нам не разрешается читать файлы и опрашивать окружение. Но как тогда наделить программы на Haskell этой (довольно тривиальной) функциональностью? Ответ дает специальный тип: `IO`.

В предыдущей главе мы видели, что «функция» `main` в нашей программе – не настоящая функция, а имеет тип `IO()`. Это называется *действием ввода-вывода*. Такие действия можно вызывать, и они могут возвращать значения. Действие `main` не возвращает ничего осмысленного, кроме значения `()`, которое является единственным значением типа `()`. Этот тип называется *единичным* и приходит на помощь, когда вы не хотите возвращать ничего.

3.1.1 Простые действия ввода-вывода

Рассмотрим еще несколько примеров действий ввода-вывода:

```
getLine :: IO String

putStrLn :: String -> IO ()
```

Эти действия выполняют ввод и вывод в окружении, внешнем по отношению к нашей программе. `getLine` читает одну строку, завершающуюся знаком новой строки из описателя `stdin`; обычно это текст, введенный в командной строке. Поэтому она имеет тип `IO String`, т. к. это действие ввода-вывода, возвращающее значение типа `String`. `putStrLn` печатает строку в `stdout`, выходной поток программы, и добавляет в конце знак новой строки. Ее тип показывает, что это не действие ввода-вывода, а функция, значением которой является действие ввода-вывода. Действия ввода-вывода можно рассматривать как описание взаимодействия с окружением. Однако это взаимодействие необязательно имеет место просто потому, что действия существуют. Чтобы это произошло, действия необходимо вычислить.

Поговорим о том, как используются действия. В листинге ниже показана полная программа, которая читает строку, введенную пользователем, и печатает ее.

Листинг 3.1 Простая программа, которая читает строку, введенную пользователем, и печатает ее

```
module Main (main) where

main :: IO ()
main = do
  line <- getLine
  putStrLn line
```

Экспортирует действие main из модуля Main

Начинает блок do

Вызывает getLine и сохраняет возвращенное значение как line

Вызывает putStrLn для line и возвращает ()

Этот синтаксис значительно отличается от всего, что мы видели до сих пор, и называется он *do-нотацией*. Конструкция начинается ключевым словом `do`, которое означает начало действия ввода-вывода. Именно в таком *блоке* `do` мы можем вызывать действия ввода-вывода. Это разрешается делать только там, потому что вычислить действие ввода-вывода в чистых функциях (обычно) невозможно. С помощью стрелки влево (`<-`) мы говорим, что действие должно быть вычислено при достижении этой точки программы, а возвращенное действием значение должно быть присвоено указанной переменной. Вычисление преобразует описание взаимодействия в его реализацию. Вычисление `getLine` в листинге 3.1 служит примером. Последнее предложение в нашем блоке `do` определяет тип всего действия ввода-вывода. Поскольку `putStrLn line` имеет тип `IO ()`, типы согласованы!

> **ПРИМЕЧАНИЕ** do-нотация используется не только для действий ввода-вывода. Это синтаксический сахар для *монады* любого вида. Но детали мы сейчас опустим и вернемся к ним в следующих главах.

По поводу этой нотации стоит сказать, что она выглядит очень похоже на *императивное программирование*, поскольку действия вызываются сверху вниз. Но не дайте нотации ввести себя в заблуждение: мы по-прежнему пишем функциональные программы. Эта нотация применяется для *композиции* действий, поскольку их можно использовать для представления сколь угодно сложного поведения, например:

- печать текста на экран;
- отправка данных через сетевой интерфейс;
- конструирование и представление графического интерфейса (GUI) пользователю.

На самом деле *любая программа является действием ввода-вывода*, потому что ее точка входа (`main`) имеет тип `IO ()`. Таким образом, от действий ввода-вывода (обычно) невозможно «убежать», т. е. нельзя определить вычисление действия внутри чистого кода. Если значение возвращается действием ввода-вывода, то обработать его необходимо в другом действии ввода-вывода.

3.1.2 Моделирование цикла

Попрактикуемся в применении этого нового синтаксиса, написав простое действие ввода-вывода, которое интерактивно нумерует строки, вводимые пользователем. Если хотите, можете называть это интерактивной версией `nl`. Для начала просто прочитаем строку и напечатаем перед ней `` `1. ` ``:

```
interactiveLines :: IO ()
interactiveLines = do
  line <- getLine
  putStrLn ("1. " ++ line)
```

Однако этот код напечатает номер только один раз. Нам необходимо повторять процесс и одновременно увеличивать номер строки. Но как этого достичь? Во-первых, в Haskell нет циклов, а данные неизменяемы, поэтому нельзя создать циклическое действие и инкрементировать счетчик, как мы сделали бы в других языках. Чтобы реализовать «цикл», мы можем вызвать действие еще раз:

```
interactiveLines :: IO ()
interactiveLines = do
  line <- getLine
  putStrLn ("1. " ++ line)
  interactiveLines
```

А как инкрементировать счетчик? На данный момент все строки будут первыми. Чтобы исправить положение, мы можем параметризовать действие, передав аргумент. Таким образом, мы создаем

функцию, которая принимает счетчик в качестве аргумента и *вычисляет действие ввода-вывода*:

```
interactiveLines :: Int -> IO ()
interactiveLines counter = do
  line <- getLine
  putStrLn (show counter ++ ". " ++ line)
  interactiveLines (counter + 1)
```

Мы можем преобразовать произвольное число в строку с помощью функции `show`, которую здесь используем для преобразования счетчика. Прежде чем выполнять рекурсивный вызов, мы инкрементируем счетчик, готовясь к отображению следующего номера.

> **ПРИМЕЧАНИЕ** Функцию `show` можно использовать для преобразования значений самых разных типов в строки. В главе 5 мы узнаем, как идентифицировать эти типы, когда будем говорить о *классах типов*.

Важно понимать, что это возможно, потому что написанная нами функция вычисляет действие ввода-вывода. Разумеется, эта функция изменяет результат в зависимости от аргументов. То есть для любого значения счетчика мы, по сути дела, вызываем *новое* действие. Просто вызвать `interactiveLines` в блоке `do` недостаточно, потому что это *не* действие ввода-вывода из-за отсутствия параметра. Короче говоря, тип `Int -> IO ()` не является действием ввода-вывода!

3.1.3 Выход из рекурсивного действия

Способ организовать цикл мы нашли, но нужно еще придумать, как из этого цикла выйти. Можно просто прекратить подсчет и выйти из цикла, как только пользователь введет пустую строку. Для этого можно воспользоваться функцией `null`, которая проверяет, что список пуст, и предложением if-then-else. Пока что можно предполагать, что функция `null` имеет тип `String -> Bool`. Однако мы должны определить, что делать, если список пуст. Когда мы находимся в блоке `do` действия ввода-вывода, последнее выражение само должно быть действием ввода-вывода. Это значит, что в случае if-then-else мы должны возвращать действие ввода-вывода в обоих случаях. Конечно, если ввод не пуст, то можно просто вызвать наше действие рекурсивно, но что делать, если ввод пуст? Нужно придумать, как породить действие ввода-вывода типа `IO ()`, которое не делает ничего. Для этой цели можно использовать очень полезную функцию `return`:

```
return :: a -> IO a
```

С ее помощью можно обернуть значение действием ввода-вывода (a просто означает, что здесь можно использовать любой тип). Однако не следует думать, что `return` действительно производит возврат из действия – происходит нечто иное. По определению, значением, возвращаемым действием, является последнее вычисляемое в нем действие!

```
myAction :: IO Int
myAction = do
  return 1
  return 2
  return 3
```

Это действие всегда возвращает `3`, поскольку его можно прочитать следующим образом:

```
myAction :: IO Int
myAction = do
  _ <- return 1
  _ <- return 2
  return 3
```

Конструкция `_ <-` просто отбрасывает возвращенное значение. В общем случае именно это мы делаем с действиями типа `IO ()`, потому что нас не интересуют значения, не несущие информации.

В примере `interactiveLines` мы можем использовать функцию `return` для конструирования действия ввода-вывода в случае else. Но если ввод не пуст, то мы выполняем два действия ввода-вывода. Однако два действия ввода-вывода, записанных друг под другом, не образуют действия ввода-вывода. Поэтому важно поместить их в отдельный блок `do`, т. к. именно он определяет действие. Это показано в листинге ниже.

Листинг 3.2 Действие ввода-вывода, которое повторяет введенную пользователем строку, печатая перед ней возрастающий счетчик

```
interactiveLines :: Int -> IO ()
interactiveLines counter = do
  line <- getLine
  if null line
    then return ()
    else do
      putStrLn (show counter ++ ". " ++ line)
      interactiveLines (counter + 1)
```

Читает строку, введенную пользователем

Проверяет, пуста ли строка

Возвращает действие ввода-вывода, не делающее ничего

Начинает новый блок do, определяя тем самым новое действие

Рекурсивно выполняет текущее действие с увеличенным счетчиком

Печатает введенную пользователем строку, предваряя ее значением счетчика

Мы написали первый прототип желаемой программы – простую демонстрацию действий ввода-вывода. С помощью многострочного ввода в GHCi (с использованием `:{` и `:}`) мы можем протестировать свое действие:

```
ghci> :{
ghci| ...
ghci| :}
ghci> interactiveLines 1
Hello
1. Hello
IO
2. IO
Action
3. Action

ghci>
```

Мы видим, что каждая введенная строка печатается, за исключением пустой строки. Действия ввода-вывода в GHCi ведут себя не совсем так, как в нормальной программе. Напомним, что GHCi вычисляет введенные предложения. Поэтому после ввода предложения какого-то типа ввода-вывода оно вычисляется:

```
ghci> getLine
Hello
"Hello"
```

Если действие ввода-вывода возвращает значение (не единичного типа), то это значение печатается на экране.

3.2 Чистые функции внутри действий

Располагая первым прототипом программы, мы должны подумать о том, как ее структурировать. Какие части программы могут находиться в коде чистой функции, а какие производят побочные эффекты?

> **ПРИМЕЧАНИЕ** Еще раз повторим, что побочные эффекты – это взаимодействия с окружением, находящимся вне функции, изнутри этой функции. В этой главе нас в основном интересует интерфейс с операционной системой. Но речь могла бы идти о некотором изменяемом состоянии или разделяемых ресурсах в многопоточной программе.

Наша программа должна получить от операционной системы две вещи. Во-первых, т. к. мы пишем командную утилиту, необходим

способ настроить ее поведение с помощью аргументов командной строки. Во-вторых, программа должна читать содержимое файла, заданного таким аргументом. Эта функциональность, равно как и вывод в самом конце, понуждает нас производить побочные эффекты, для достижения которых нужны действия ввода-вывода. Однако реализовать нумерацию строк, прочитанных из файла, вполне можно в чистом коде.

3.2.1 Чтение и модификация пользовательских данных

Тут должен возникнуть вопрос: как использовать чистый функциональный код внутри действий ввода-вывода? Само собой, есть несколько способов. Как мы только что видели, можно передавать аргументы функции, которая вычисляет действие ввода-вывода. Ничто не мешает преобразовывать эти аргументы с помощью чистых функций! Допустим, мы хотим написать программу, которая читает строку, преобразует ее в верхний регистр и печатает. Это похоже на программу в листинге 3.1, но перед печатью требуется простая модификация переменной `line`. Для преобразования одной литеры в верхний регистр мы можем использовать функцию `toUpper` из модуля `Data.Char`. Но преобразования одной литеры недостаточно. Нам нужно преобразовать каждую литеру строки, а, как вы помните, для этой цели есть подходящий кандидат: функция `map`! Используя `toUpper` в сочетании с `map`, мы можем написать функцию, которая преобразует строку в верхний регистр. И включить ее в простую программу, написанную выше. Полученная в результате программа будет использовать это преобразование перед печатью строки на экран. Она показана в листинге ниже.

Листинг 3.3 Программа, которая читает строку, полученную от пользователя, преобразует ее в верхний регистр и печатает

```
module Main (main) where

import Data.Char

main :: IO ()
main = do
  line <- getLine
  putStrLn (map toUpper line)
```

Импортирует все определения из Data.Char

Читает строку, введенную пользователем

Переводит строку в верхний регистр и печатает результат

Этот пример показывает, как можно использовать *чистый* код, свободный от побочных эффектов, в реальных программах. За обработку побочных эффектов отвечают действия, которые вызывают написанный нами чистый код. Функции типа `map` и `toUpper` свободны от любых побочных эффектов, а действия ввода-вывода *обработали*

побочные эффекты, не раскрывая их остальному коду. Эта идея иллюстрируется на рис. 3.2. Конечно, обработка исключений в Haskell несколько сложнее, но начать можно с представления о действиях ввода-вывода как о среде, которая обрабатывает за нас побочные эффекты.

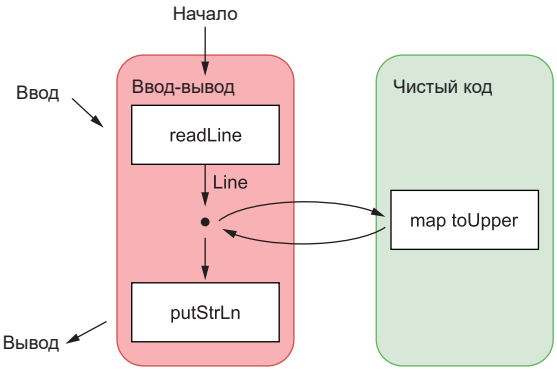


Рис. 3.2 Взаимодействие побочных эффектов, действий и чистого кода

В общем случае мы всегда должны думать о том, как уменьшить объем нечистого кода (например, действий) и увеличить объем чистого. Почему? Потому что так проще рассуждать о программе и тестировать код. Функцию, которой запрещено производить побочные эффекты, можно без труда протестировать с помощью автономного тестирования или тестирования свойств, поскольку подготавливать тестовую среду необязательно. Поэтому лучше использовать действия ввода-вывода только в случае острой необходимости, а в остальное время ограничиться чистым кодом.

3.2.2 Поток данных между чистым и нечистым кодами

Теперь, когда мы знаем, как использовать чистый код внутри действий, можно продумать устройство утилиты `nl` более детально. Какие компоненты и с какой функциональностью нам понадобятся, какие из этих частей будут чистыми, а какие нечистыми? Код работы со строками файла и даже с аргументами чистый, потому что нам нужно только преобразовать строки и различать регистр литер. Так что нумерация, фильтрация и форматирование строк – чистый код. Но все относящееся к ресурсам вне нашей программы, в частности к аргументам командной строки и файлам, нуждается в побочных эффектах, поэтому придется использовать действия.

- Чистый код (функции):
  - разбор аргументов;
  - чтение, фильтрация и форматирование строк.

- Нечистый код (действия):
  - чтение аргументов;
  - чтение файла;
  - печать на экран.

Уже можно представить себе, как скомпоновать эти задачи. На рисунке ниже показаны различные компоненты нашей утилиты и поток данных между ними.

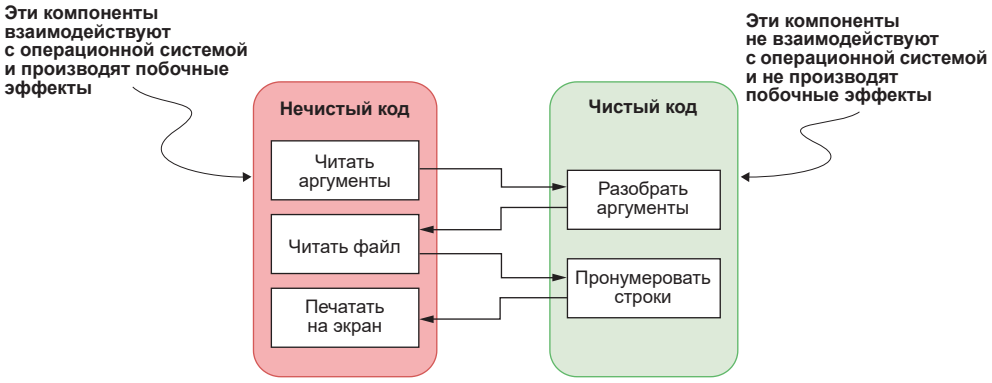


Рис. 3.3 Компоненты и действия утилиты нумерации строк; поток данных показан стрелками

Причина такого строгого разделения компонентов не техническая. На самом деле после компиляции все компоненты будут слиты в единое целое, но пока мы их пишем, разумно держать в уме, есть ли какие-то побочные эффекты и как они влияют на семантику программы. И раз уж мы заговорили о побочных эффектах, начнем с первого – чтения аргументов командной строки.

3.3 Чтение из окружения

Важная часть любой программы – взаимодействие с окружением. Программы, работающие в современной операционной системе, обычно имеют доступ к аргументам командной строки и системным переменным окружения. Haskell предоставляет нам модуль, умеющий делать именно это: `System.Environment`. В нем есть несколько функций, которые можно использовать для получения информации из окружения программы, а именно аргументов (`getArgs`), имени программы (`getProgName`) и значения переменной окружения (`lookupEnv`). В нашей утилите нас интересуют только аргументы.

3.3.1 Разбор аргументов командной строки

`getArgs` – действие ввода-вывода, возвращающее список строк (оно имеет тип `IO [String]`), которое получает полный список аргументов командной строки. Так что прочитать аргументы не просто, а очень просто. Трудная часть – разобрать аргументы, потому что обычно они устроены по определенной схеме. Чтобы не усложнять эту схему, мы начнем с единственного аргумента, содержащего имя файла. То есть программа будет вызываться так: `nl <filename>`. Если аргумент не задан, мы будем печатать текст справки. Если задан один или более аргументов, то первый интерпретируется как имя файла, а остальные игнорируются.

Если вы еще не создали новый проект, то самое время сделать это сейчас! Как и раньше, перейдите в нужный каталог и начните проект `lnums`, выполнив команду `stack`:

```
stack new lnums
```

На этот раз мы не станем игнорировать `Main.hs`, а начнем с реализации разбора аргументов в этом файле. Наша упрощенная программа должна проверять, заданы ли вообще аргументы, а точнее есть ли хотя бы один элемент в списке, возвращенном `getArgs`. Это можно сделать в чистой функции с помощью сопоставления с образцом. Образец, соответствующий списку, содержащему ровно один элемент, имеет вид `[x]`. Этот единственный элемент идентифицируется по имени, указанному в образце (в данном случае `x`). Это эквивалентно использованию образца вида `x : []`. Набросок разбора аргумента, содержащего имя файла, показан в листинге ниже:

Листинг 3.4 Простая функция разбора аргументов

```
parseArguments :: [String] -> FilePath
parseArguments [filePath] = filePath
parseArguments _ = undefined
```

Вычисляет имя файла, если входной список содержит ровно один элемент

Возбуждает исключение, если структура входного списка иная

Здесь мы использовали тип `FilePath` (синоним типа `String`), чтобы было понятно, что эта функция вычисляет путь к файлу. Если один тип является синонимом другого, то их можно использовать взаимозаменяемо, так что `String` становится `FilePath`. В этом примере мы также видим применение универсального образца `_`. Этот образец всегда сопоставляется, поскольку эквивалентен использованию простого имени. Но в случае универсального образца ввод не связан ни с каким идентификатором, поэтому отбрасывается.

> **ПРИМЕЧАНИЕ** Ставя универсальный образец на последнее место, мы гарантируем, что сопоставление будет исчерпывающим, т. к. он сопоставляется с чем угодно.

Но этот код, конечно, не годится, потому что программа будет «падать» всякий раз, как пользователь неправильно введет данные. Нужно проявить больше вежливости и вместо аварийного завершения сообщить пользователю, в чем он ошибся. И это подводит нас к вопросу о том, как обрабатывать ошибки. Как вычислить чистое значение, которое также представляет ошибку? Использовать новый тип, который называется `Maybe`!

3.3.2 Кодирование ошибок с помощью Maybe

`Maybe` – тип, представляющий возможность отсутствия значения, т. е. значение этого типа может быть чем-то или ничем. Как и списки, он параметризуется каким-то типом, т. е. тип `Maybe Int` существует так же, как и `[Int]` (список `Int`). Определение `Maybe` выглядит следующим образом:

```
data Maybe a = Just a | Nothing
```

Это новый для нас синтаксис, который будет встречаться в нашей книге часто. Ключевое слово `data` служит для определения *алгебраических типов данных* (algebraic data types – ADT) с помощью *конструкторов*. Здесь мы определяем тип `Maybe` с одной свободной переменной типа `a`. Значение типа `Maybe` строится либо конструктором `Just` в сочетании со значением полиморфного типа, либо простым конструктором `Nothing`, который не привносит никакой новой информации. Встретив значение типа `Maybe`, мы можем быть уверены, что оно создано одним из этих конструкторов. Это позволяет писать чистые функции, которые могут завершаться ошибкой, но *не* критичной. Результатом будет просто `Nothing`.

> **ПРИМЕЧАНИЕ** `Maybe` используется не только для функций, которые могут завершаться ошибкой, но и для обозначения факультативных аргументов функции. Однако чаще всего он обозначает результирующее значение, которое может отсутствовать.

Этот тип уже предоставляется Haskell, поэтому мы можем модифицировать функцию в листинге 3.4 так, как показано в листинге 3.5. Вместо ошибки вследствие использования `undefined` мы просто вычисляем значение `Nothing`! Разумеется, тип функции теперь нужно изменить на `Maybe FilePath`, потому что вычисляем мы не путь к файлу, а значение типа `Maybe`, которое может содержать

`FilePath`. Если это действительно путь к файлу, то он будет обернут конструктором `Just`.

Листинг 3.5 Простая функция разбора аргументов с Maybe

```
parseArguments :: [String] -> Maybe FilePath
parseArguments [filePath] = Just filePath
parseArguments _ = Nothing
```

Кодирует результат с помощью конструктора Just

Кодирует ошибку разбора с помощью конструктора Nothing

С такой функцией программа больше не падает, поскольку ошибка кодируется с помощью типа. Но как работать с таким значением? Да мы же уже знаем ответ: *сопоставление с образцом*! Как и в случае списков, мы можем сопоставлять с разными конструкторами типа `Maybe`. Это позволит извлечь значение конструктора `Just`. Поскольку извлечь значение из `Nothing` не получится, мы должны предоставить значение по умолчанию. Это подводит нас к определению функции в листинге 3.6. Обратите внимание на сигнатуру этой функции. Тип `a` – свободная переменная типа, т. е. `a` могла бы быть любым типом, лишь бы во всех ее вхождениях тип был один и тот же. Так что если вы хотите использовать эту функцию со значением типа `Maybe Int`, то должны будете задать `Int` в качестве первого аргумента.

Листинг 3.6 Функция извлечения значения типа Maybe с использованием умолчания

```
fromMaybe :: a -> Maybe a -> a
fromMaybe _ (Just v) = v
fromMaybe v Nothing = v
```

Извлекает значение, обернутое конструктором Just

Используется значение по умолчанию, если значением Maybe является Nothing

Эту функцию можно использовать для «потрошения» `Maybe`. Это полезно, когда мы хотим сцепить вычисления, которые могут давать `Maybe`, потому что в них возможна ошибка. В таком случае мы распознаем ошибку (посредством сопоставления с образцом) и без труда изменим поток управления или возьмем значение по умолчанию. Ниже показана функция `fromMaybe` в действии:

```
ghci> fromMaybe 0 (Just 100 :: Maybe Int)
100
ghci> fromMaybe 0 (Nothing :: Maybe Int)
0
```

К счастью, нам нет нужды реализовывать эти функции самостоятельно! Модуль `Data.Maybe` предоставляет большинство нужных функций. Некоторые из них перечислены в табл. 3.1.

Таблица 3.1 Наиболее употребительные функции, используемые совместно с типом `Maybe`

Имя	Тип	Описание
`maybe`	`b -> (a -> b) -> Maybe a -> b`	Эта функция принимает значение по умолчанию, функцию и `Maybe`. Если `Maybe` содержит `Nothing`, то возвращается значение по умолчанию, в противном случае функция применяется к значению, обернутому конструктором `Just`, и возвращается результат этого вызова
`fromMaybe`	`a -> Maybe a -> a`	Эта функция принимает значение по умолчанию и `Maybe` в качестве аргументов. Если `Maybe` содержит `Nothing`, то возвращается значение по умолчанию, в противном случае возвращается значение, обернутое конструктором `Just`
`catMaybes`	`[Maybe a] -> [a]`	Эта функция принимает список значений `Maybe` в качестве аргумента и возвращает список всех значений, обернутых конструктором `Just`. Значения `Nothing` игнорируются
`mapMaybe`	`(a -> Maybe b) -> [a] -> [b]`	Эта функция принимает в качестве первого аргумента функцию, которая получает значение и возвращает `Maybe` потенциально другого типа, а в качестве второго – список значений. Возвращается список, являющийся результатом отображения функции на второй аргумент, при этом результаты, равные `Nothing`, отбрасываются. Определение эквивалентно `mapMaybe f xs = catMaybes (map f xs)`

Тип `Maybe` постоянно прокрадывается в стандартную и во многие внешние библиотеки, потому что является основным средством представления отсутствующих значений или ошибок. Примером может служить функция `elemIndex` из модуля `Data.List`, которая возвращает либо индекс элемента, обернутый `Just`, либо `Nothing`, если элемента нет в списке.

Упражнение. Безопасная функция индексирования

В главе 2 мы написали упрощенную версию функции, которая получает индекс элемента в списке. Она называлась `indexOf`, имела тип `Char -> [Char] -> Int` и аварийно завершала программу, если элемент отсутствовал в списке. Перепишите ее с использованием `Maybe`. Тип функции нужно изменить на `Char -> [Char] -> Maybe Int`.

3.4 Пример: чтение и печать аргументов командной строки

Воспользуемся функцией `fromMaybe` для печати полезного сообщения, в случае если пользователь забыл указать путь к файлу. Мы можем с помощью функции `maybe` выбрать нужное действие в зависимости от заданного пути. Если путь отсутствует, то выбираем действие, печатающее сообщение. В противном случае конструируем действие из пути, полученного в результате разбора аргументов.

Идея кода представлена в листинге ниже. Эта программа печатает аргумент, если он задан, и текст справки в противном случае.

Листинг 3.7 Программа, которая печатает свой единственный аргумент или текст справки, если никакой аргумент не задан

```
module Main (main) where

import System.Environment
printHelpText :: String -> IO ()
  printHelpText msg = do
  putStrLn (msg ++ "\n")
  progName <- getProgName
  putStrLn ("Usage: " ++ progName ++ " <filename>")

parseArguments :: [String] -> Maybe FilePath
parseArguments [filePath] = Just filePath
parseArguments _ = Nothing

main :: IO ()
main = do
  cliArgs <- getArgs
  let mFilePath = parseArguments cliArgs
  maybe
    (printHelpText "Missing filename")
    (\filePath -> putStrLn filePath)
    mFilePath
```

Печатает информацию о порядке вызова

Получает имя, под которым была вызвана программа

Разбирает аргумент, содержащий путь к файлу

Получает аргументы, с которыми была вызвана программа

Печатает текст справки, если разбор завершился неудачно

Вычисляет действие ввода-вывода, которое должно быть выполнено в зависимости от результата разбора

Печатает имя файла, если разбор был успешен

В этом коде объединены уже встречавшиеся нам элементы. Он читает аргументы из окружения, разбирает их и в зависимости от результата печатает либо текст справки, либо заданное имя файла.

> **ПРИМЕЧАНИЕ** При использовании локальных переменных типа `Maybe` принято начинать их имя со строчной буквы `m`. Например, `mFilePath` означает переменную типа `Maybe`, которая может содержать путь к файлу.

Этот пример – демонстрация того, как можно использовать тип `Maybe` в качестве чистого метода обработки ошибок и соответственного изменения потока управления. Удивление может вызвать тот факт, что выполняемые *действия* сами могут быть результатом вызова функции. Это отчетливо видно на примере использования функции `maybe`. В зависимости от результата разбора она вычисляет разные действия. Так что чистая функция `maybe` решает, что наша программа должна сделать с окружением. Это важно понимать уже

на ранних этапах изучения Haskell. Не только функции могут являться значениями, но и действия.

3.4.1 Ключевое слово let

В нашем примере встречается также новое ключевое слово `let`. Оно позволяет создавать определения, или `let`-*привязки*, внутри действий так же, как переменные определяются в других языках. В простейшем варианте оно используется так:

```
let <identifier> = <expr>
```

Внутри действия это может выглядеть следующим образом:

```
let x = 1 + 1
```

Но не впадайте в заблуждение, видя этот синтаксис. Мы не создаем изменяемых переменных. С идентификаторами, созданными в результате такой привязки, следует обращаться как с неизменяемыми данными или константами.

> **ПРИМЕЧАНИЕ** Использовать `let`-привязки можно также для создания функций, если после идентификатора перечислить имена аргументов: `let functionName arg1 arg2 arg3 = ...`. Это позволяет использовать аргументы внешней функции в определении внутренних.

Это полезно, когда мы пытаемся сделать вложенные выражения немного понятнее или использовать одно и то же выражение несколько раз. Продемонстрируем это на примере написанной ранее программы:

```
main = do
  line <- getLine
  putStrLn (map toUpper line)
```

Выражение `map toUpper line` можно привязать к новому идентификатору до использования.

```
main = do
  line <- getLine
  let upperCaseLine = map toUpper line
  putStrLn upperCaseLine
```

Ключевое слово `let` можно использовать и за пределами действий в определениях обычных функций, поскольку они могут быть частью выражения:

```
ghci> let x = 1 + 1 in x + 1 :: Int
3
```

Здесь мы определяем привязку выражений `1 + 1` к идентификатору `x`, который объявлен как имеющий тип `Int`. В выражениях `let` закрывается ключевым словом `in`, показывающим, для каких выражений должна иметь место привязка.

3.4.2 Запуск программ с помощью stack

Теперь мы хотим протестировать поведение кода в листинге 3.7, собрав и запустив исполняемый файл. Для этого служит команда `stack run`, однако нужно позаботиться о передаче программе аргументов:

```
stack run <аргументы stack> -- <аргументы программы>
```

Желая передать аргументы исполняемому файлу, мы должны предпослать им два дефиса: `--`. Протестируем реализацию – сначала без аргументов:

```
shell $ stack run
Missing file name

Usage: lnums-exe <file name>
```

Мы видим свою справку! Прочитанное из окружения имя программы – `lnums-exe`. В данном случае *lnums* – имя проекта. Программа `stack` автоматически добавляет суффикс *exe* к имени исполняемого файла. Теперь попробуем запустить ее с аргументом:

```
shell $ stack run -- Testpath
Testpath
```

Программа корректно разбирает и печатает аргумент. Итак, мы реализовали базовый анализатор аргументов, который сможем расширить, когда возникнет необходимость.

Основу программы мы заложили. Мы умеем читать и разбирать аргумент и действовать соответственно. Это значительная часть нечистого кода программы. Далее предстоит подумать о том, как читать из файловой системы и как нумеровать строки файла. Этим мы займемся в следующей главе.

Резюме

- Действия ввода-вывода используются для взаимодействия с окружением операционной системы, в частности ввода-вывода из файловой системы или сети.
- Среда исполнения Haskell несет ответственность за вычисление действий ввода-вывода, на долю программиста остается только задание их последовательности.

- `do`-нотация используется для задания последовательности действий и способа передачи их результатов другим действиям и функциям.
- Цикл можно смоделировать с помощью рекурсивных вызовов функций.
- `return` служит для обертывания значения действием ввода-вывода, но не означает «возврат» из действия.
- При использовании if-then-else в сочетании с действиями необходимо следить за тем, чтобы ветви *then* и *else* вычисляли действия одинакового типа.
- Выражения, передаваемые действиям, можно строить из чистых функций, что позволяет использовать чистый код внутри нечистого.
- `Maybe` – *алгебраический тип данных*, используемый для возврата значения, когда в функции может возникнуть ошибка; он позволяет избежать использования `undefined`.
- Ключевое слово `let` используется для привязки определений внутри действий к идентификаторам.

4 Утилита нумерации строк

Краткое содержание этой главы:

- чтение файлов из файловой системы;
- параметризация поведения функций высшего порядка с помощью аргументов;
- использование алгебраических структур данных для кодирования возможных параметров;
- упаковка кода в виде исполняемого файла.

В предыдущей главе мы заложили основу для программы, умеющей нумеровать строки указанного файла. Мы узнали, как прочитать аргументы, заданные пользователем. Теперь можно подумать о том, как читать файл и получать из него отдельные строки.

Мы начнем эту главу с обсуждения чтения файлов и преобразования прочитанного содержимого. Затем напишем общую функцию, для которой потоки управления и данных определяются функциями, переданными в качестве аргументов. После этого мы узнаем, как кодировать параметры с помощью алгебраических структур данных, и напоследок завершим реализацию утилиты, начатую в предыдущей главе.

4.1 Чтение файлов и преобразование их содержимого

Для решения этой задачи нам понадобятся кое-какие действия ввода-вывода. Haskell предлагает рудиментарную функциональность для работы с файлами в виде двух действий:

```
readFile :: FilePath -> IO String

writeFile :: FilePath -> String -> IO ()
```

`readFile`, как следует из названия, читает содержимое файла и возвращает его в виде одной строки. `writeFile` получает строку и записывает ее в файл; если такой файл не существует, то он создается. Здесь мы снова видим синонимы типов, конкретно `FilePath`. Из самого имени типа понятно, к чему относится аргумент в обеих функциях. Если бы аргумент имел тип `String`, то его назначение было бы не столь очевидно. Одно лишь имя типа позволяет многое сказать об этих функциях.

Мы будем использовать `readFile` для конструирования действия, которое возвращает строки файла. Для этого необходимо просканировать содержимое файла и разбить по знаку новой строки `\n`. К счастью, Haskell уже предлагает функцию, которая именно это и делает: `lines`!

```
ghci> :t lines
lines :: String -> [String]
ghci> lines "Hello\nWorld\n"
["Hello","World"]
```

Как видим, она разбивает строку на части и даже устраняет знаки новой строки. У этой функции есть обратная, `unlines`, которая делает в точности противоположное:

```
ghci> :t unlines
unlines :: [String] -> String
ghci> unlines ["Hello", "World"]
"Hello\nWorld\n"
ghci> unlines (lines "Hello\nWorld")
"Hello\nWorld\n"
```

`unlines` добавляет знак новой строки после каждой части, однако `lines` игнорирует отсутствие такого знака в конце своего аргумента.

Помимо этих функций, существует функция `words`, которая разбивает строку по пробелам, и `unwords`, которая соединяет слова, вставляя между ними пробелы. Все эти функции находятся в модуле `Data`.

`String`, но импортировать их необязательно, потому что они подгружаются посредством модуля `Prelude`, который импортируется всегда.

> **ПРИМЕЧАНИЕ** `Prelude` реэкспортирует многие важные функции из разных модулей. Например, функции `readFile` и `writeFile` изначально находятся в модуле `System.IO`. Нам его импортировать нет нужды, потому что это делается автоматически. Такое поведение можно подавить с помощью флага компилятора или прагмы в коде, но в большинстве случаев лучше этого не делать.

Итак, теперь мы можем написать функцию, которая использует `readFile` и `lines` для чтения строк из файла. Нам нужно прочитать содержимое и вернуть результат функции `lines`. Это показано в следующем листинге.

Листинг 4.1 Действия для чтения строк из файла

```
readLines :: FilePath -> IO [String]
readLines filePath = do
  contents <- readFile filePath
  return (lines contents)
```

Читает все содержимое файла с указанным путем

Возвращает список строк, разбивая содержимое по знакам новой строки

Важно отметить, что это действие завершается исключением, если файл не существует. Нам это подходит, потому что сообщение об ошибке скажет пользователю, что он сделал не так, поэтому обрабатывать ошибку мы не станем.

Создав файл `testFile.txt` в каталоге проекта, мы можем что-то записать в него и протестировать свое действие. Мой файл содержал такой текст:

```
Hello
dear
reader!
```

Теперь запустим цикл REPL (командой `stack repl`) и протестируем действие:

```
ghci> readLines "testFile.txt"
["Hello","dear","reader!"]
```

Отлично! Теперь можно снабдить строки номерами.

4.1.1 Написание чистой библиотеки

Теперь обратимся к модулю `Lib` в нашем проекте. Поскольку код нумерации строк довольно общий, его следует поместить в библио-

течный модуль. Так как о многих *нечистых* аспектах программы мы уже позаботились, можно сосредоточиться на *чистом* коде, который покажется нам более знакомым.

Сначала подумаем, какие возможности мы хотим поддержать. Мы собираемся нумеровать все строки или только строки, удовлетворяющие некоторому предикату, например непустые или содержащие заданную подстроку, поэтому понадобится функция, позволяющая отображать номера на строки. Кроме того, мы хотим поддержать разные стили форматирования номеров строк, а чтобы иметь возможность выравнивать номера на левую или правую границу, нам нужно будет знать, сколько строк всего, чтобы вычислить максимальную длину номера.

Подумаем, как представить такое отображение номера на строку типом данных. Мы знаем, что нужно будет поддержать отображение произвольного количества номеров. Также мы знаем, что порядок отображения важен, т. к. нужно знать, в каком порядке печатать строки. Списки дают почти все, что нам нужно, поскольку могут расти неограниченно, и порядок их элементов детерминирован, но как представить отображение в виде списка? Ответ дают *ассоциативные списки*. Они содержат элементы, представляющие отображение одного значения на другое. Представить такое отображение проще всего с помощью *кортежа*, потому что он объединяет два типа в один.

Для нумерации можно было бы взять такой тип:

```
type NumberedLines = [(Int, String)]
```

С его помощью можно представить ассоциацию между номерами строк и строками.

> **ПРИМЕЧАНИЕ** Ассоциативные списки – чрезвычайно гибкий инструмент, их можно использовать для построения самых разных отображений. Однако многие алгоритмы для работы с ними обходятся довольно дорого с точки зрения временной сложности, поскольку чаще всего для нахождения ключа приходится просмотреть весь список. Но если производительность не особенно важна, то ассоциативные списки – прекрасная структура данных!

Выглядело бы это примерно так:

```
[(1, "First Line"), (2, "Second Line"), (3, "Third Line")]
```

Однако для нас это не подходит, поскольку мы хотим оставить возможность *нумеровать только некоторые строки*. Это значит, что у каких-то строк не будет номера! Поэтому тип должен иметь возможность представлять строку без номера. По счастью, мы уже

встречали тип, способный представить *отсутствующее* значение: `Maybe`! Представим номера строк следующим образом:

```
type NumberedLine = (Maybe Int, String)
type NumberedLines = [NumberedLine]
```

Одиночная пронумерованная строка состоит из необязательного номера и текстовой строки. Все пронумерованные строки помещаются в список таких отображений.

> **ПРИМЕЧАНИЕ** Здесь мы видим, что `Maybe` используется не только для обертывания ошибок, но и для кодирования факультативных значений. Во многих языках имеются типы, эквивалентные `Maybe`, которые часто называются `Option` или как-то похоже.

Имея тип для номеров, мы можем подумать о том, какие функции понадобятся в программе. Нужна функция, порождающая номера строк. Однако эту функцию следует параметризовать, так чтобы она могла пропускать некоторые строки при нумерации. Кроме того, нужен параметр, управляющий тем, как увеличивать номера строк, потому что существует фундаментальная разница между подсчетом и нумерацией строк. Мы уже знаем, что нужен параметр, позволяющий пропускать пустые строки, но нужно ли их при этом учитывать при подсчете? Как показано на рис. 4.1, есть разные подходы к этому вопросу. И лучше бы, чтобы решение принимал пользователь!

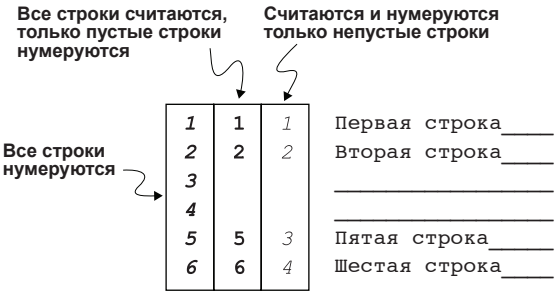


Рис. 4.1 Разные способы нумеровать пустые строки

В предыдущей главе мы узнали, что *функции высшего порядка* принимают в качестве аргументов другие функции, которые могут параметризовать их поведение. Поскольку нам нужны сложные правила игнорирования и нумерации строк, почему бы не воспользоваться этой возможностью для построения функции высшего порядка, которая принимает предикаты, показывающие, когда увеличивать номер и какие строки нумеровать?

4.1.2 Сокрытие вспомогательных аргументов

Начнем с функции, которая нумерует все строки без каких-либо предикатов, а затем будем ее постепенно усложнять. Прежде всего нужно решить вопрос о типе. Мы получаем список строк (отдельных строк файла) и должны породить список `NumberedLines`, как раньше, но как управлять счетчиком строк? Раскрыть его в качестве аргумента? В идеале не хотелось бы, потому что мы хотим начинать нумерацию с 1. Однако же если мы собираемся делать функцию рекурсивной, то нужно как-то передать увеличенный счетчик при рекурсивном вызове. Мы решим эту проблему с помощью *скрытой* функции внутри доступной извне. Для этого создадим определение локальной функции *внутри* нашей. Мы можем воспользоваться `let`-привязкой, набросок этой идеи показан в листинге ниже.

Листинг 4.2 Структура функции с локально определенной функцией

```
numberAllLines :: [String] -> NumberedLines
numberAllLines lines =
  let go :: Int -> [String] -> NumberedLines
      go counter lines = ...
  in go 1 lines
```

Создает определение локальной функции с помощью let-привязки

Вызывает локально определенную функцию

Это немного отличается от виденного нами ранее употребления `let`! Дело в том, что `let` ведет себя иначе, когда используется в сочетании с `do`-нотацией. В чистых функциях ключевое слово `let` создает локальные определения (которые могут ссылаться друг на друга) и делает эти определения доступными с помощью ключевого слова `in`. Это показано на рис. 4.2. Порядок следования определения неважен.

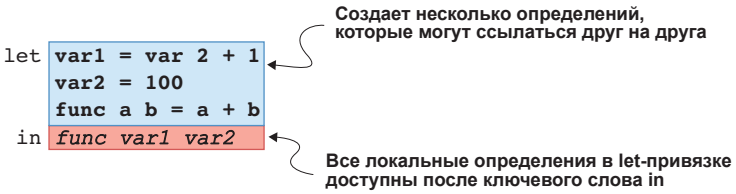


Рис. 4.2 Синтаксис let-привязок

Таким образом, мы можем использовать эту `let`-привязку для создания еще одной функции с другой сигнатурой типа, которую затем будем вызывать из нашей функции! Эта функция может иметь

больше аргументов (или даже совсем другие аргументы), чем исходная. При определении внешней функции в терминах определения внутренней мы можем использовать любые (фиксированные или переменные) аргументы. Таким образом, мы можем скрыть параметры по умолчанию рекурсивных функций, как, например, переменную `counter` в следующем примере.

> **ПРИМЕЧАНИЕ** При создании локально определенной функции с целью расширения сигнатуры типа путем добавления специальных аргументов бывает трудно подобрать для нее подходящее имя. Часто они называются просто `go` или `aux` (от слова *auxiliary* – вспомогательный) в знак того, что эта функция является частью объемлющего определения.

Теперь мы можем рекурсивно инкрементировать переменную счетчика `counter`. На каждом шаге рекурсии предоставляется новое *ассоциативное отображение* для нумерации строк, инкрементируется `counter`, и если еще не все строки пронумерованы, то производится следующий рекурсивный вызов. Новый список строится рекурсивно путем определения результата как конкатенации вновь созданного отображения, которое состоит из кортежа, содержащего значение типа `Maybe` и строку, с результатом рекурсивного вызова функции с увеличенным счетчиком. Это показано в листинге ниже.

Листинг 4.3 Функция, нумерующая все строки по порядку

```
numberAllLines :: [String] -> NumberedLines
numberAllLines lines =
  let go :: Int -> [String] -> NumberedLines
      go _ [] = []
      go counter (x : xs) = (Just counter, x) : go (counter + 1) xs
  in go 1 lines
```

Сигнатура типа локально определенной функции

Возвращает пустой список, когда входной список пуст, и работает как условие остановки рекурсии

Создает новое отображение с текущей строкой и рекурсивно строит остальные пронумерованные строки

Вызывает вспомогательную функцию с фиксированным аргументом в качестве счетчика

В этом примере мы видим, что в первом вхождении функции `go` используется универсальный образец `_` в качестве первого аргумента. Почему? Поскольку в данной точке этот аргумент не нужен (нечего нумеровать), мы можем опустить соответствующий идентификатор. И вообще, рекомендуется так поступать для аргументов, которые не используются.

Упражнение. Строки и слова

Функции `lines`, `unlines`, `words` и `unwords` очень полезны при работе со строками. Реализуйте их самостоятельно, воспользовавшись рекурсивными определениями! Помните, что `String` – это просто список литер, поэтому их можно использовать в сопоставлении с образцом и создавать определения с дополнительными параметрами внутри функций с помощью `let`-привязки. Для написания таких функций используйте сопоставление с образцом со стражами.

Итак, мы имеем функцию, которая умеет нумеровать все переданные ей строки. Протестируем ее в GHCi:

```
ghci> numberAllLines ["Hello", "", "World"]
[(Just 1,"Hello"),(Just 2,""),(Just 3,"World")]
```

4.2 Параметризация поведения функций высшего порядка

Далее рассмотрим проблему обнаружения пустых строк. Мы хотим предоставить пользователю возможность подавить для них нумерацию, но для этого их сначала нужно найти. Конечно, если строка не содержит ни одной литеры, то ее можно рассматривать как пустую. Но что, если она содержит только управляющие символы (которые не печатаются) или пробелы? В таком случае ее тоже следует считать пустой. Нам необходим предикат, который будет решать, что строка пустая. Иными словами, это должна быть функция, возвращающая значение типа Boolean. Чтобы проверить, содержит ли строка только управляющие символы и пробелы, мы можем воспользоваться полезной функцией `isPrint` из модуля `Data.Char`, которая идентифицирует печатаемые литеры, и функцией `isSeparator`, которая идентифицирует пробелы и другие символы-разделители, определенные в Юникоде:

```
ghci> import Data.Char
ghci> isPrint 'a'
True
ghci> isPrint '\n'
False
ghci> isSeparator ' '
True
ghci> isSeparator 'a'
False
```

Мы можем использовать эти функции для построения предиката, обнаруживающего пустые строки. Для этого нам понадобится функ-

ция `all`, которая проверяет истинность предиката для каждого элемента в списке:

```
ghci> all (\x -> x == 1) [1,1,1 :: Int]
True
ghci> all (\x -> x == 1) [1,1,2 :: Int]
False
```

Мы можем использовать ее для написания функции, которая возвращает `True`, если строка пуста или содержит только непечатаемые литеры и разделители:

Листинг 4.4 Предикат, проверяющий, что строка содержит хотя бы одну печатаемую литеру

```
isEmpty :: String -> Bool
isEmpty str =
  null str
    || all (\s -> not (isPrint s) || isSeparator s) str
```

Возвращает True, если строка пуста

Возвращает True, если строка содержит только непечатаемые литеры и разделители

Эта функция позволяет проверять строки на пустоту:

```
ghci> isEmpty "Test"
False
ghci> isEmpty " "
True
ghci> isEmpty "\n "
True
ghci> isEmpty "A "
False
```

Мы можем также определить дополнительную функцию, которая проверяет, что строка не пуста:

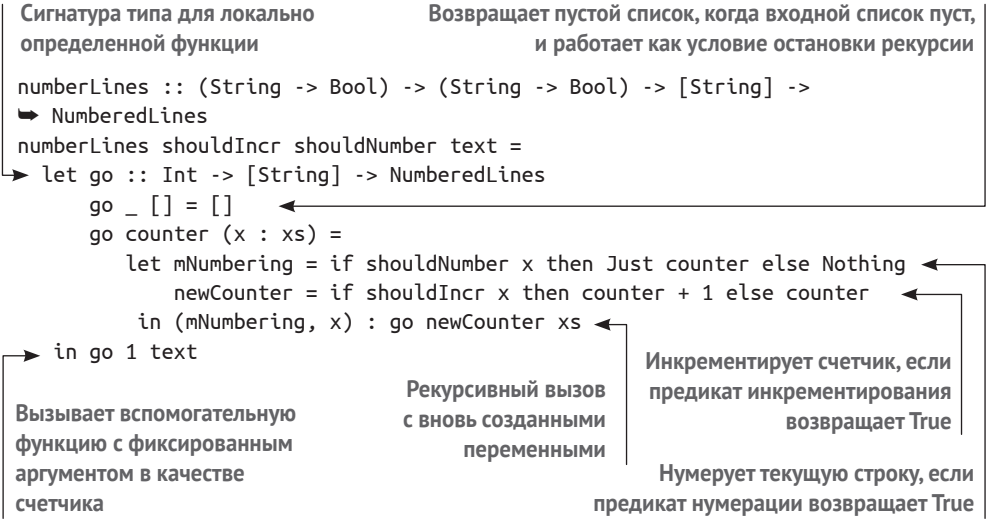
```
isNotEmpty :: String -> Bool
isNotEmpty str = not (isEmpty str)
```

Теперь можно приступить к модификации нашей функции нумерации. Поскольку мы хотим изменять поток управления на основе предикатов, применяемых к строкам, то тип аргументов нашей новой функции уже известен: `String -> Bool`. Нам понадобятся два предиката: один будет решать, следует ли инкрементировать счетчик, а другой – следует ли нумеровать строку. Сигнатура нашей функции выглядит так:

```
numberLines :: (String -> Bool) -> (String -> Bool) -> [String]
➥ -> NumberedLines
numberLines shouldIncr shouldNumber text = ...
```

Мы переименовали функцию, потому что теперь это общая функция нумерации строк. Предикаты теперь могут присутствовать в определении нашей функции! Мы можем использовать if-then-else в `let`-привязке внутри функции `go`, чтобы нумеровать строки, инкрементировать счетчик или не делать ничего. Код приведен в листинге ниже.

Листинг 4.5 Обобщенная функция высшего порядка для нумерации строк




```
numberLines :: (String -> Bool) -> (String -> Bool) -> [String] ->
➥ NumberedLines
numberLines shouldIncr shouldNumber text =
  let go :: Int -> [String] -> NumberedLines
      go _ [] = []
      go counter (x : xs) =
         let mNumbering = if shouldNumber x then Just counter else Nothing
             newCounter = if shouldIncr x then counter + 1 else counter
          in (mNumbering, x) : go newCounter xs
  in go 1 text
```

Предикаты могут влиять на поведение подсчета и нумерации, поэтому на основе этой функции мы можем строить другие варианты! Напишем функцию `numberAllLines`.

4.2.1 Частичное применение функции

Для этого нам понадобятся функции, которые всегда возвращают `True`, поскольку мы собираемся подсчитывать и нумеровать все строки. Написать такую функцию просто:

```
(\_ -> True)
```

Эта функция игнорирует свой вход и возвращает `True`. Но зачем останавливаться на этом? Ее можно было бы сделать чуть более общей, устроив так, чтобы она всегда возвращала некоторую константу и игнорировала второй аргумент:

```
const :: a -> b -> a
const x = (\_ -> x)
```

Теперь выражение вида `const True` эквивалентно тому, что мы видели раньше. Обратите внимание на сигнатуру типа. Мы использу-

ем две свободные переменные типа. Выбирая для второго аргумента тип `b`, мы говорим, что он может быть отличен от типа первого аргумента. Эту функцию можно переписать, включив игнорируемый аргумент в список аргументов:

```
const :: a -> b -> a
const x _ = x
```

Итак, мы пришли к общей функции, которая порождает константную унарную функцию! К счастью, эта функция уже есть в Haskell! И теперь мы можем построить функцию `numberAllLines` из функции `numberLines` в листинге 4.5 и `const`:

```
numberAllLines :: [String] -> NumberedLines
numberAllLines text = numberLines (const True) (const True) text
```

В определении этой функции странным может показаться выражение `const True`. Разве это не бинарная функция? Тогда почему только один аргумент? Этот прием называется *частичным применением функции*. В Haskell (и многих других функциональных языках) необязательно передавать функции все необходимые ей аргументы. Если какие-то аргументы отсутствуют, то результатом вычисления будет другая функция, которая нуждается в отсутствующих аргументах!

Имея выражение вида `(\x -> f x)`, мы можем выполнить *эта-редукцию*, которая сводит выражение к `f`. Это влечет важные синтаксические последствия! Для функций с большим количеством аргументов возможна повторная эта-редукция, что позволяет писать очень лаконичные определения функций. Но почему это работает в случае `const`? Давайте рассмотрим данный вопрос более пристально. Пусть имеется выражение

```
const True
```

Вспомните предыдущее определение: `const x = (\_ -> x)`. Если теперь заменить `x` заданным значением (`True`), то получится

```
(\_ -> True)
```

То есть вычисляется унарная функция, которая возвращает константное значение `True`, но каков тип этой функции? Его легко вывести из того, что функция `const` имеет тип `a -> b -> a`. Поскольку мы задали первый аргумент равным `True`, то знаем, что `a` должно иметь тип `Bool`. Стало быть, окончательный тип будет

```
b -> Bool
```

Эта простая функция может принимать аргумент произвольного типа, поскольку `b` – свободная переменная типа, и возвращает Boolean. Это имеет смысл, т. к. аргумент попросту игнорируется.

Мы даже можем еще раз произвести такую эта-редукцию для функции `numberAllLines`:

```
numberAllLines :: [String] -> NumberedLines
numberAllLines = numberLines (const True) (const True)
```

Мы снова проверяем, согласуется ли это с типами. Так как `numberLines` имеет тип `(String -> Bool) -> (String -> Bool) -> [String] -> NumberedLines`, а `const True` – тип `b -> Bool`, мы можем подставить `String` вместо `b` и найти, что сигнатура типа `numberLines (const True) (const True)` должна быть `[String] -> NumberedLines`, а это именно тот тип, который нам нужен!

> **СОВЕТ** Эта-редукция – очень удобный способ написания лаконичных определений функций. Она используется почти в любом коде на Haskell, который вам встретится. Хотя применять эту технику совершенно необязательно, знакомство с ней полезно, поскольку так вам будет проще понять код, написанный другими.

Теперь можно проверить, что функция `numberAllLines` делает то, что должна:

```
numberAllLines ["Hello", "", "World", "!"]
[(Just 1,"Hello"),(Just 2,""),(Just 3,"World"),(Just 4,"!")]
```

Действительно, все строки пронумерованы! Но наша цель состояла не в том, чтобы написать функцию, только и умеющую нумеровать все строки. Давайте-ка что-нибудь проигнорируем.

Упражнение. Использование эта-редукций

Проект из предыдущей главы был реализован без использования *эта-редукций*, чтобы не усложнять синтаксис. Но теперь, когда мы знаем, как эти редукции работают, просмотрите заново исходный код и решите, можно ли где-нибудь воспользоваться редукциями.

Сначала напишем функцию, которая по-прежнему увеличивает счетчик для каждой строки, но не нумерует пустые строки. Для этого мы можем снова воспользоваться предикатами совместно с общей функцией `numberLines`. Первый аргумент должен быть равен константе `True`, потому мы хотим инкрементировать счетчик всегда, а второй – быть предикатом, возвращающим `True` только для *непустых* строк. По счастью, такой предикат у нас уже есть:

```
numberNonEmptyLines :: [String] -> NumberedLines
numberNonEmptyLines = numberLines (const True) isNotEmpty
```

Мы снова используем эта-редукцию, чтобы не писать аргумент типа `[String]`. Протестируем это определение:

```
ghci> numberNonEmptyLines ["Hello", "", "World", "!"]
[(Just 1,"Hello"),(Nothing,""),(Just 3,"World"),(Just 4,"!")]
```

Счетчик инкрементируется всегда, но пустые строки не нумеруются. Напишем также версию, которая не инкрементирует счетчик для пустых строк. Сделать это просто: нужно лишь передать предикат `isNotEmpty` в обоих аргументах!

```
numberAndIncrementNonEmptyLines :: [String] -> NumberedLines
numberAndIncrementNonEmptyLines = numberLines isNotEmpty isNotEmpty
```

Протестируем и это определение:

```
ghci> numberAndIncrementNonEmptyLines ["Hello", "", "World", "!"]
[(Just 1,"Hello"),(Nothing,""),(Just 2,"World"),(Just 3,"!")]
```

Ну, вот и все! На основе общего определения в листинге 4.5 мы написали три функции, функциональность которых существенно различается. Плоды наших трудов подытожены в следующем листинге.

Листинг 4.6 Три варианта функций, нумерующих строки по разным условиям

```
numberAllLines :: [String] -> NumberedLines
numberAllLines = numberLines (const True) (const True)

numberNonEmptyLines :: [String] -> NumberedLines
numberNonEmptyLines =
  numberLines (const True) isNotEmpty

numberAndIncrementNonEmptyLines :: [String] -> NumberedLines
numberAndIncrementNonEmptyLines =
  numberLines isNotEmpty isNotEmpty
```

Нумерует все строки

Нумерует все непустые строки, но инкрементирует счетчик даже для пустых строк

Нумерует только непустые строки и счетчик инкрементирует только для них

Из этих определений видно, как легко расширять эти функции. Нужно только написать простые булевы предикаты для строк, и мы сможем решать, какие строки нумеровать, какие не нумеровать, для каких инкрементировать счетчик, а для каких – нет.

4.3 Алгебраические структуры данных как кодирование возможностей

Теперь у нас есть функция нумерации строк, и осталось только их напечатать. Последняя трудность, которую нам предстоит преодо-

леть, – дополнение номеров строк слева или справа. И тут мы тоже хотим отыскать общий подход. Поскольку мы знаем, что дополнять нужно будет слева или справа, то можем представить себе параметризованную функцию, которая выполняет то или иное дополнение на основе некоторого параметра. Но как должен выглядеть такой параметр?

4.3.1 Типы-суммы или помеченные объединения

Чтобы ответить на этот вопрос, вернемся к ключевому слову `data`, которое мы начали рассматривать при обсуждении типа `Maybe`. Мы узнали, что его можно использовать для определения новых типов данных с помощью конструкторов. Важное свойство этих конструкторов – возможность участвовать в сопоставлении с образцом. Благодаря этому легко использовать такой тип данных для создания простого *типа-суммы*, позволяющего выразить различные варианты дополнения. *Тип-сумма* (иногда его называют *помеченным объединением*) – это тип данных, состоящий из конечного числа конструкторов фиксированных типов, активным из которых в любой момент времени может быть только один. С этими типами ассоциированы имена (отсюда и «метка» в *помеченном объединении*). Желая представить разные виды дополнения строки, мы можем воспользоваться метками для указания того, с какой стороны дополнять.

Листинг 4.7 Простой тип для различения дополнения слева и справа

```
data PadMode = PadLeft | PadRight
```

Этот тип показан в листинге 4.7. Вы можете прочитать `PadMode` как *либо* `PadLeft`, *либо* `PadRight`. Наша функция дополнения может произвести сопоставление с образцом для этого типа и разветвиться по способу дополнения строки. Как и типы, созданные с помощью ключевого слова `type`, типы, созданные с помощью `data`, должны быть добавлены в список экспорта. Поскольку с этим типом связаны *конструкторы* (`PadLeft` и `PadRight`), их тоже следует экспортировать. Сделать это можно, добавив `PadMode (..)` в список экспорта.

Каким же образом мы дополняем строку? Если она короче указанной длины, то справа (в режиме `PadRight`) или слева (в режиме `PadLeft`) добавляется столько пробелов, чтобы длина результирующей строки была равна требуемой. Для этого можно воспользоваться функцией `replicate`:

```
ghci> :t replicate
replicate :: Int -> a -> [a]
ghci> replicate 10 ' '
"  "
```

Эта функция принимает целое число и некоторое значение произвольного типа и вычисляет список, содержащий это значение число раз, равное первому аргументу. А что, если первый аргумент отрицателен?

```
ghci> replicate (-1) 'a'
""
```

Как выясняется, `replicate` предусматривает такую возможность. Конечно, не бывает списков отрицательного размера, поэтому возвращается просто пустой список. Это важно, потому что позволяет вычислять дополнение вне зависимости от того, нужно ли добавлять хотя бы один пробел. Его можно вычислить как

```
replicate (<desired length> - <actual length>) ' '
```

Снова используя `let`-привязку, мы можем легко написать функцию, которая вычисляет разность между требуемой и фактической длиной и необходимое дополнение:

```
pad :: Int -> String -> String
pad n str =
  let diff = n - length str
      padding = replicate diff ' '
  in ...
```

Вычисляет разность между требуемой и фактической длиной

Вычисляет дополнение входной строки

Постойте-ка, мы же забыли про `PadMode`! Куда поместить этот аргумент? Правильный ответ – на первое место. Почему? Снова подумайте о частичном применении функции. Если режим является первым аргументом, то мы просто передаем его в качестве аргумента и получаем в ответ другую функцию, которая производит дополнение в заданном режиме! Скоро мы это увидим.

Зададимся еще одним вопросом: где производить сопоставление с образцом? До сих пор мы встречали его только в аргументах функций, но в данном случае это бесполезно, т. к. мы хотим использовать определения из `let`-привязки, но не повторять их. Поэтому познакомимся еще с одной синтаксической конструкцией – для сопоставления с образцом внутри функции в виде простого выражения с использованием ключевых слов `case` и `of`. Выглядит это так:

Начинает сопоставление <expression 1> с образцом

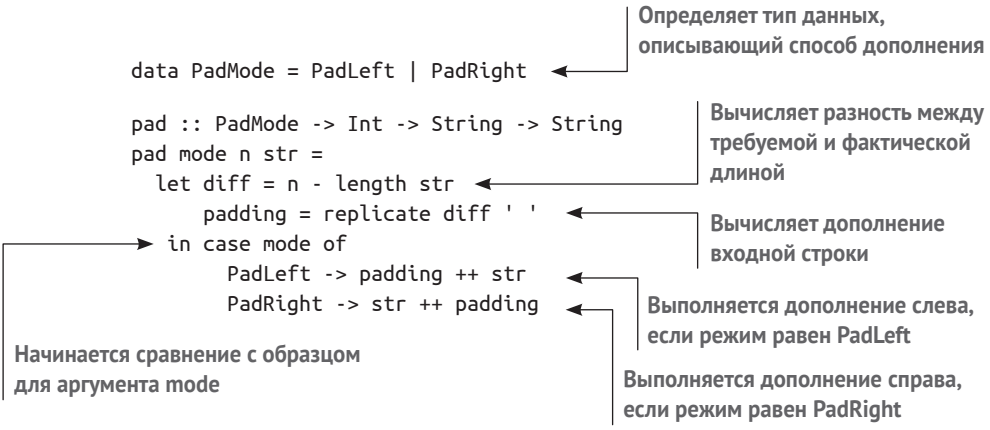
```
case <expression 1> of
  <pattern 1> -> <expression 2>
  <pattern 2> -> <expression 3>
  ...
```

Если сопоставляется образец <pattern 1>, то результатом выражения является результат вычисления <expression 2>

То же относится к <pattern 2> и <expression 3>

Образцы работают так же, как и раньше. Этот синтаксис можно использовать для сопоставления с режимом `PadMode`, которое мы хотим добавить в функцию `pad` в листинге 4.7. В итоге мы приходим к следующему определению функции, которая умеет изменять поведение в зависимости от аргумента типа `PadMode`.

Листинг 4.8 Обобщенная функция для дополнения слева или справа



И вот – мы написали функцию, выполняющую дополнение! Добавив все новые функции в список экспорта модуля, мы можем его протестировать:

```
ghci> pad PadLeft 10 "Pad me!"
"   Pad me!"
ghci> pad PadRight 10 "Pad me!"
"Pad me!   "
```

4.3.2 Не повторяйся

Теперь мы можем создать конкретные функции на основе этой очень общей функции, снова воспользовавшись частичным применением функции. Это показано в листинге 4.9. Почему это разумный подход к написанию кода? Прежде всего нам не пришлось повторяться, а в программировании это всегда хорошо. Кроме того, представьте, что мы захотели создать еще режим `PadCenter` для центрирования строки. В таком случае нужно было бы только расширить тип данных и добавить еще один образец. Все остальные определения можно было бы использовать, как раньше. Это экономит время, потому что не приходится снова и снова реализовывать одно и то же. В таком (довольно простом) примере этот довод может показаться надуманным, но при реализации более сложных функций это действительно помогает сэкономить. У использования дополнительного параметра `mode` есть еще одно преимущество. Допустим, мы хотим

написать функцию, которая пользуется функцией `pad` (а мы хотим ей воспользоваться), но режим дополнения заранее не известен. Как ее параметризовать? Конечно, можно было бы использовать в качестве аргумента функцию, которая производит дополнение, но гораздо понятнее передать параметр `Pad-mode`. Так мы сразу понимаем, для чего служит аргумент.

Листинг 4.9 Функции дополнения слева и справа, основанные на обобщенной функции дополнения

```
padLeft :: Int -> String -> String
padLeft = pad PadLeft

padRight :: Int -> String -> String
padRight = pad PadRight
```

Определяет функцию дополнения слева на основе общей функции выравнивания

Определяет функцию дополнения справа на основе общей функции дополнения

Теперь, когда мы умеем дополнять номера, осталось только преобразовать список `NumberedLines` во что-то, что можно напечатать.

> **Упражнение. Центрирование**
>
> Мы обсудили, как можно было бы расширить тип данных и функцию в листинге 4.8, чтобы реализовать центрирование, когда входная строка располагается в середине отведенного ей поля. Сделайте это! Для деления целых чисел можете воспользоваться функцией `div`.

4.3.3 Функция zip

Для преобразования списка типа `NumberedLines`, созданного с помощью функции `numberLines`, нам понадобится еще одна функция, которая преобразует этот синоним ассоциативного списка во что-то, что проще напечатать, а точнее в список строк. Для этого нужно собрать все номера строк, вычислить максимальную длину номера, соответственно дополнить и выровнять номера и дописать их в начала соответствующих им строк. Все бы ничего, но мы имеем дело со списком кортежей. Необходимо расчленить этот список, выделив номера, а затем снова соединить два списка.

Сначала обсудим, что мы уже сделали, и (в очередной раз) обобщим это! Наша функция `numberLines` пользуется булевыми предикатами для порождения списка кортежей (типа `(Maybe Int, String)`) из простых строк. Как сделать это в общем виде? В предположении, что имеется два списка, которые нужно объединить в список кортежей, Haskell предоставляет удобную функцию `zip` типа `[a] -> [b]`

`-> [(a, b)]`. Она ведет себя как застежка-молния: берет два списка и объединяет соответственные элементы в кортежи, останавливаясь, когда более короткий список закончится. Это наглядно изображено на рис. 4.3. Рассмотрим несколько примеров.

```
ghci> zip [1,2,3] ["Hello", "World", "!"] :: [(Int, String)]
[(1,"Hello"),(2,"World"),(3,"!")]
ghci> zip [1..10] ['a'..'z'] :: [(Int, Char)]
[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e'),(6,'f'),(7,'g'),(8,'h'),(9,'i'),
➥ (10,'j')]
ghci> zip [1..10] [] :: [(Int, Int)]
[]
```

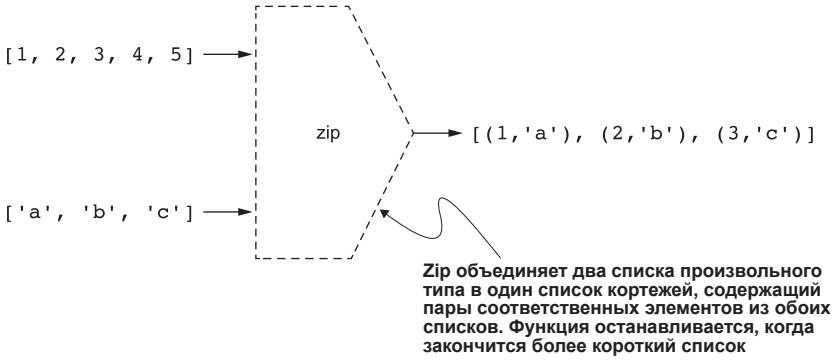


Рис. 4.3 Взаимодействие побочных эффектов, действий и чистого кода

У функции `zip` есть квазиобратная функция – `unzip`. Если `zip` используется для застегивания, то `unzip` – для расстегивания.

```
ghci> unzip [(1,'a'), (2,'b'), (3,'c')] :: ([Int], String)
([1,2,3],"abc")
ghci> unzip (zip [1..5] ['a'..'c'] :: ([Int], String))
([1,2,3],"abc")
```

Но, как показывает этот пример, `unzip` не является истинно обратной `zip` функцией, потому что `zip` может отбрасывать элементы (из более длинного списка). Эти функции очень полезны при работе с ассоциативными списками, поскольку дают чистый и безопасный способ конструирования и деконструирования последних. Это делает их подходящими кандидатами для расчленения нашего списка типа `NumberedLines`, т. к. этот тип – не что иное, как синоним `[(Maybe Int, String)]`! Мы могли бы заново изобрести колесо, написав специальную рекурсивную функцию для деконструирования списка, но это не только чревато ошибками, но и заставит создать кучу функций, которые делают, по существу, одно и то же. Благодаря функциям, предоставляющим какую-то базовую функциональ-

ность, мы избежим повторения и облегчим чтение кода наших собственных функций.

Чтобы узнать самый длинный номер строки, мы должны собрать все номера, вычислить их строковые представления (с помощью `show`), вычислить длины строк и взять их максимум. Это можно сделать довольно лаконично с помощью функции `maximum`, уже имеющейся в Haskell, в сочетании с `map`:

```
ghci> xs = [1..1000000] :: [Int]
ghci> maximum (map length (map show xs))
7
```

Внутренний вызов `map` преобразует список числовых значений в список значений типа `String`. Внешний вызов `map` вычисляет длину каждой строки, а `maximum` вычисляет максимальный элемент в списке, порожденном внешним `map`!

Умея вычислять максимальную длину номера строки, мы можем написать функцию, вычисляющую дополнения для номеров строк. Но как объединить эти выровненные номера с самими строками? Использовать `zip` нельзя, т. к. мы хотим включить номера прямо в значения типа `String`. Что нам нужно, так это обобщенная функция `zip`, позволяющая управлять способом объединения элементов. И Haskell снова приходит на помощь, предоставляя функцию `zipWith`!

```
ghci> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
ghci> zipWith (+) [1..10] [1..3] :: [Int]
[2,4,6]
ghci> zipWith (\x y -> (show x) ++ ": " ++ y) [1,2,3 :: Int]
➥ ["One", "Two", "Three"] :: [String]
["1: One","2: Two","3: Three"]
```

`zipWith` – еще один пример функции высшего порядка, поскольку мы можем модифицировать ее поведение, передав функцию по своему выбору. Как легко догадаться по названию, эта функция работает *в точности* как `zip`, только вместо создания кортежей вызывает заданную нами функцию, которой передает два соответственных элемента.

Упражнение. Функция zipWith

Как и `map`, `zip` – типичная вездесущая функция, которую любой пишущий на Haskell должен знать наизусть. Чтобы лучше познакомиться с ней, реализуйте ее, применив рекурсивный алгоритм. А затем обобщите ее и напишите `zipWith`. Дополнительный вопрос: сможете ли вы реализовать `zip` с помощью `zipWith`?

4.3.4 Работа с отсутствующими значениями

И последняя тонкость: наши номера строк имеют не числовой тип, а `Maybe Int`! Следовательно, описанные выше методы не работают, как это ни грустно. Так в чем же проблема? В том, что внутри `map`, примененной к номерам строк, мы должны были бы всюду проверять, какой конструктор нам подсунули: `Just` или `Nothing`. По этой причине мы и изучали функцию `maybe`. С ее помощью можно легко написать функцию `map`, умеющую преобразовывать список элементов типа `Maybe Int` в список `String`:

```
ghci> ys = [Just 1, Just 2, Nothing, Just 4] :: [Maybe Int]
ghci> map (maybe "" show) ys
["1","2","","4"]
```

Здесь мы снова воспользовались частичным применением функции, чтобы немного сократить определение. Располагая всеми этими функциями, мы наконец можем написать функцию, которая создает изящно оформленные строковые представления нашего списка типа `NumberedLines`. Она показана в листинге ниже.

Листинг 4.10 Функция преобразования нумерованных строк в отформатированные

```
prettyNumberedLines :: PadMode -> NumberedLines -> [String]
prettyNumberedLines mode lineNums =
  let (numbers, text) = unzip lineNums
      numberStrings = map (maybe "" show) numbers
      maxLength = maximum (map length numberStrings)
      paddedNumbers = map (pad mode maxLength) numberStrings
  in zipWith (\n l -> n ++ " " ++ l) paddedNumbers text
```

Преобразует список кортежей в кортеж списков

Объединяет выровненные номера строк с самими строками

Преобразует все номера строк в строковое представление и дополняет их до максимальной длины

Вычисляет максимальную длину номеров строк

Преобразует номера строк в строковое представление, а отсутствующие значения (Nothing) в пустые строки

Здесь мы видим, что каждое определение в `let`-привязке является частью определения всей функции. Каждое привязанное значение – кусочек головоломки, который занимает свое место в окончательном определении, благодаря чему составляется полная картина. Также мы видим, какие разные вычисления нужны для выполнения

операции в целом. Расчленение списка типа `NumberedLines`, нумерация строк, нахождение максимальной длины номеров строк и дополнение номеров – вот шаги, которые нужно выполнить, чтобы собрать все вместе и получить красиво отформатированные строки. Когда будете писать свои функции для решения более сложной задачи, обязательно постарайтесь разбить задачу на более мелкие части. Этот метод «разделяй и властвуй» прекрасно работает в сочетании с синтаксисом `let`!

> **ПРИМЕЧАНИЕ** Префикс `pretty` обычно используется для функций, которые преобразуют данные в *понятную человеку* форму.

Эта функция также служит прекрасным примером *декларативного стиля программирования*. Мы определяем несколько промежуточных результатов, которые могут ссылаться друг на друга. И лишь последнее выражение завершает определение всей функции. При таком стиле программирования не делается никаких предположений о том, в каком порядке эти определения используются для вычисления фактических значений. Этот вопрос мы изучим более подробно в последующих главах.

4.3.5 Печать списка значений с помощью mapM

У нас есть почти все части программы. Последнее, что осталось сделать, – печать всех отформатированных строк на экране. Для этого нам нужен способ напечатать каждую из списка строк. На ум приходит функция `map`, которая умеет делать нечто подобное: вычислять *чистую* функцию для каждого элемента списка. Однако сейчас мы хотим выполнить *действие* для каждого элемента. Поэтому наша функция `map` работать не будет, т. к. она умеет вычислять действие, но не выполнять его. По счастью, есть и другие функции.

Haskell предлагает две функции, предоставляющие ту же функциональность, что и `map` в контексте действий. Называются они `mapM` и `mapM_`, и каждая принимает в качестве аргумента унарную функцию, которая порождает действия, и вычисляет действие применительно к списку элементов. Переданная функция вычисляется для каждого элемента, причем действия выполняются *по порядку*. Это именно то, что можно использовать в сочетании с функцией `putStrLn` для нашего списка значений типа `String`. Но какую функцию выбрать? Попробуем обе:

```
ghci> mapM putStrLn ["Hello", "World", "!"]
Hello
World
!
```

```
[(),(),()]
ghci> mapM_ putStrLn ["Hello", "World", "!"]
Hello
World
!
```

Странно. Почему-то `mapM` печатает строки и затем возвращает `[(),(),()]` (список единичных типов), тогда как `mapM_`, похоже, не возвращает ничего. Строго говоря, она возвращает `()` (но GHCi просто скрывает его от нас). В чем разница между функциями? `mapM` работает так же, как `map`. Возвращенные действиями значения заменяют входные значения в исходном списке. Поскольку `putStrLn` возвращает единичный тип, каждое вычисление дает `()`, и эти значения собираются в список. `mapM_` – вариант `mapM`, предназначенный специально для действий, вывод которых нас не интересует. Как уже отмечалось, единичный тип не несет никакой информации. Это значит, что его можно просто отбросить, что `mapM_` и делает, когда игнорирует вывод и просто возвращает один единичный тип.

Упражнение: монадическое отображение

Для простоты можно предположить, что типы обеих функций `map` выглядят так:

```
mapM :: (a -> IO b) -> [a] -> IO [b]
mapM_ :: (a -> IO b) -> [a] -> IO ()
```

Напишите эти функции самостоятельно! Сначала внимательно проанализируйте, как была устроена наша чистая функция `map`, а затем взгляните на пример `interactiveLines` в начале этой главы. Функции `mapM` и `mapM_` можно написать так же!

Поскольку нас интересует только печать списка строк, подходит функция `mapM_`. Ее очень легко использовать для печати номеров строк!

4.4 От библиотеки к исполняемому файлу

Вот теперь можно собрать все вместе! Мы можем модифицировать программу из предыдущей главы, так чтобы она читала строки из указанного пользователем файла, нумеровала их, создавала отформатированные версии пронумерованных строк, а затем выводила их на экран, как в листинге ниже.

Листинг 4.11 Действие main, содержащее базовую функциональность утилиты нумерации строк

```
main :: IO ()
main = do
  cliArgs <- getArgs
  let mFilePath = parseArguments cliArgs
  maybe
    (printHelpText "Missing filename")
    ( \filePath -> do
        fileLines <- readLines filePath
        let numbered = numberAllLines fileLines
            prettyNumbered = prettyNumberedLines
➥ PadLeft numbered
        mapM_ putStrLn prettyNumbered
    )
    mFilePath
```

Читает аргументы, с которыми была вызвана программа

Разбирает аргумент, содержащий имя файла

Печатает справку, если задан не ровно один аргумент

Читает строки из файла

Нумерует каждую строку

Создает красиво отформатированное представление пронумерованных строк

Печатает пронумерованные строки на экране

Собрав все вместе, мы можем наконец-то протестировать программу! Мы можем пронумеровать строки! Взгляните на следующий пример:

```
shell $ stack run -- testFile.txt
 1 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas a
 2 mattis nisi, eleifend auctor erat. Vivamus nisl sapien, suscipit non
 3 gravida sed, mattis a velit. Fusce non iaculis urna, a volutpat leo.
 4 Ut non mauris vel massa tincidunt mattis. Sed eu viverra lectus.
 5 Donec pulvinar justo quis condimentum suscipit. Donec vel odio eu
 6 odio maximus consequat at at tellus. Ut placerat suscipit vulputate.
 7 Donec eu eleifend massa, et aliquet ipsum.
 8
 9 Mauris eget massa a tellus tristique consequat. Nunc tempus sit amet
10 est sit amet malesuada. Curabitur ultrices in leo vitae tristique.
11 Suspendisse potenti. Nam dui risus, gravida eu scelerisque sit amet,
12 tincidunt id neque. In sit amet purus gravida, venenatis lectus sit
13 amet, gravida massa. In iaculis commodo massa, in viverra est mollis
14 et. Nunc vehicula felis a vestibulum egestas. Phasellus eu libero sed
15 odio facilisis feugiat id quis velit. Proin a ex dapibus, lacinia dui
16 at, vehicula ipsum.
```

Здесь мы видим, что в режиме `PadLeft` строки выравниваются на правую границу. Утилита делает именно то, что от нее ожидается. Осталось только добавить дополнительные параметры.

4.4.1 Задание и разбор параметров командной строки

Наконец, мы хотим добавить дополнительные параметры программы, а именно:

- `--reverse` – изменить порядок нумерации на противоположный;
- `--skip-empty` – опустить нумерацию пустых строк;
- `--left-align` – выравнивать номера строк на левую границу.

Чтобы представить эти параметры в программе, мы можем воспользоваться алгебраическим типом данных. Это показано в листинге 4.12. Новым является ключевое слово `deriving`, которое мы рассмотрим подробнее в следующей главе. Пока не будем вдаваться в детали, но для проверки равенства с данными этого типа оно нам все же понадобится.

Листинг 4.12 Тип данных, используемый для представления параметров программы нумерации строк

```
data LineNumberOption
  = ReverseNumbering  <-- Представляет --reverse
  | SkipEmptyLines    <-- Представляет --skip-empty
  | LeftAlign         <-- Представляет --left-align
  deriving (Eq)       <-- Произведен от класса типа Eq
```

На основе этого типа данных мы можем написать функцию, которая разбирает параметры, преобразуя их к этому типу. Мы снова воспользуемся типом `Maybe`, чтобы показать, что разбор может завершиться неудачно. Код функции приведен в листинге ниже.

Листинг 4.13 Функция, которая разбирает строковые параметры и преобразует их к новому типу данных

```
lnOptionFromString :: String -> Maybe LineNumberOption
lnOptionFromString "--reverse" = Just ReverseNumbering   | Преобразует строку
lnOptionFromString "--skip-empty" = Just SkipEmptyLines  | в соответствующий ей
lnOptionFromString "--left-align" = Just LeftAlign       | конструктор
lnOptionFromString _ = Nothing  <-- Разбор неудачен, вернуть Nothing
```

Предположим, что порядок задания параметров и имени файла фиксирован: сначала параметры, в конце имя файла. В таком случае разбор оказывается немного проще. Чтобы обобщить нашу функцию разбора, мы можем воспользоваться функцией `reverse`, дабы сначала изменить порядок параметров в списке. Поскольку мы знаем, что имя файла – последний параметр, в обращенном списке оно должно быть первым, благодаря чему использовать образец `x:xs` очень просто! При разборе остальных аргументов можно использовать функцию `mapMaybe` (типа `(a -> Maybe b) -> [a] -> [b]`) из модуля `Data.Maybe`, чтобы применить к ним функцию `lnOptionFromString`. Напомним, что `mapMaybe` просто игнорирует все значения, равные `Nothing` (это значит, что параметры, которые не удалось разобрать, игнорируют-

ся). Собрав все вместе, мы можем вернуть путь к файлу и список параметров программы. Это показано в листинге ниже.

Листинг 4.14 Функция, которая разбирает список аргументов командной строки

```
import Data.Maybe

...

parseArguments :: [String] -> (Maybe FilePath, [LineNumberOption])
parseArguments args = case reverse args of
  [] -> (Nothing, [])
  (filename : options) ->
    ( Just filename,
      mapMaybe lnOptionFromString options
    )
```

Применяет сопоставление с образцом к обращенному списку аргументов

Если список аргументов пуст, то не возвращает ни имени файла, ни параметров

Возвращает имя файла, если оно задано

Разбирает дополнительные аргументы, игнорируя ошибки и не проверяя дубликаты

Наконец, мы можем включить этот код в действие `main`, показанное в листинге 4.11. Чтобы упростить задачу, нужно отступить на шаг и подумать, как лучше изменить *поток управления* в нашей программе. Поток определяется в основном используемыми функциями. Если мы хотим изменить поток управления на основе каких-то переменных (например, параметров), то можем просто выключить одни функции и включить другие. В Haskell функции являются значениями. Это значит, что их можно передавать в качестве аргументов (функциям высшего порядка) и присваивать переменным. Но почему бы, вместо того чтобы передавать нашим функциям параметры программы, просто не включать нужные функции в зависимости от параметров? Например:

```
let numberFunction =
      if SkipEmptyLines `elem` options
        then numberNonEmptyLines
        else numberAllLines
```

Проверяет, был ли задан параметр SkipEmptyLines

Выбирает нужную функцию в зависимости от параметра

Здесь предполагается, что `options` имеет тип `[LineNumberOption]` и представляет параметры, полученные от функции в листинге 4.14. Как видим, мы «сохраняем» одну из двух функций в переменной `numberFunction`. Точнее, мы изменяем значение идентификатора `numberFunction` в зависимости от результата вычисления булева выражения. И теперь можем включить параметры в действие `main`, как показано в листинге 4.15. Результат выполнения этой программы может выглядеть так:

```
shell $ stack run -- --skip-empty --left-align --reverse testFile.txt
16 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas a
15 mattis nisi, eleifend auctor erat. Vivamus nisl sapien, suscipit non
14 gravida sed, mattis a velit. Fusce non iaculis urna, a volutpat leo.
13 Ut non mauris vel massa tincidunt mattis. Sed eu viverra lectus.
12 Donec pulvinar justo quis condimentum suscipit. Donec vel odio eu
11 odio maximus consequat at at tellus. Ut placerat suscipit vulputate.
10 Donec eu eleifend massa, et aliquet ipsum.
8 Mauris eget massa a tellus tristique consequat. Nunc tempus sit amet
7 est sit amet malesuada. Curabitur ultrices in leo vitae tristique.
6 Suspendisse potenti. Nam dui risus, gravida eu scelerisque sit amet,
5 tincidunt id neque. In sit amet purus gravida, venenatis lectus sit
4 amet, gravida massa. In iaculis commodo massa, in viverra est mollis
3 et. Nunc vehicula felis a vestibulum egestas. Phasellus eu libero sed
2 odio facilisis feugiat id quis velit. Proin a ex dapibus, lacinia dui
1 at, vehicula ipsum.
```

4.4.2 Итоги проекта

Посмотрим, что мы написали. Полный модуль `Main` для проекта показан в листинге 4.15, а модуль `Lib` – в листинге 4.17.

Листинг 4.15 Заголовок, типы данных и вспомогательные функции в модуле Main

```
module Main (main) where

import Data.Maybe
import Lib
import System.Environment

data LineNumberOption
  = ReverseNumbering
  | SkipEmptyLines
  | LeftAlign
  deriving (Eq)

lnOptionFromString :: String -> Maybe LineNumberOption
lnOptionFromString "--reverse" = Just ReverseNumbering
lnOptionFromString "--skip-empty" = Just SkipEmptyLines
lnOptionFromString "--left-align" = Just LeftAlign
lnOptionFromString _ = Nothing

printHelpText :: String -> IO ()
printHelpText msg = do
 putStrLn (msg ++ "\n")
  progName <- getProgName
  putStrLn ("Usage: " ++ progName ++ " <options> <filename>")
  putStrLn "\n"
  putStrLn " Options:"
  putStrLn "   --reverse         - Reverse the numbering"
```

Импортирует модули, необходимые для наших определений

Определяет тип данных, представляющий аргументы командной строки

Определяет функцию для разбора одного аргумента командной строки

Определяет функцию вывода справки

Возвращает имя, под которым вызывалась программа

```
  putStrLn "   --skip-empty      - Skip numbering empty lines"
  putStrLn "   --left-align      - Use left-aligned line numbers"

parseArguments :: [String] -> (Maybe FilePath, [LineNumberOption])
parseArguments args = case reverse args of
  [] -> (Nothing, [])
  (filename : options) ->
    ( Just filename,
      mapMaybe lnOptionFromString options
    )
```

Определяет функцию для разбора имени файла и параметров, заданных в аргументах командной строки

Программа начинается с чтения и разбора аргументов командной строки. Последний аргумент интерпретируется как имя файла, а остальные – как факультативные параметры. Если имя файла не задано, то печатается справка, в противном случае читаются строки файла. Параметры нужны, чтобы изменить поведение программы, это делается путем выбора подходящих функций.

Приложение создает нумерованные строки, преобразует их в понятную человеку форму и печатает, после чего завершается. Это показано в следующем листинге.

Листинг 4.16 Действие main для утилиты нумерации строк

```
readLines :: FilePath -> IO [String]
readLines filePath = do
  contents <- readFile filePath
  return (lines contents)

main :: IO ()
main = do
  cliArgs <- getArgs

  let (mFilePath, options) = parseArguments cliArgs

  numberFunction =
    if SkipEmptyLines `elem` options
      then numberNonEmptyLines
      else numberAllLines

  padMode =
    if LeftAlign `elem` options
      then PadRight
      else PadLeft

  go filePath = do
    fileLines <- readLines filePath
    let numbered = numberFunction fileLines
        prettyNumbered = prettyNumberedLines
    ➥padMode numbered
              revNumbered = numberFunction
```

Читает содержимое файла и разбивает его на строки

Читает аргументы, с которыми была вызвана программа

Печатает номера строк в естественном или инвертированном порядке в зависимости от параметра --reverse

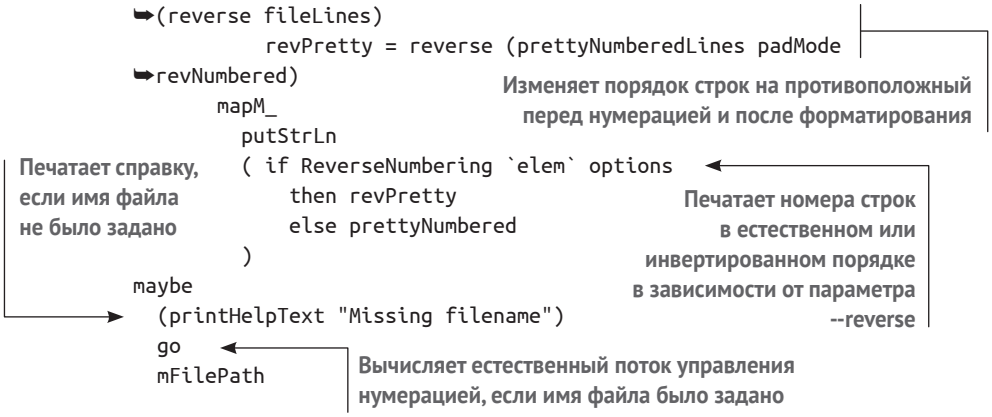
Разбирает аргументы и выделяет имя файла и дополнительные параметры

Выбирает режим дополнения в зависимости от параметра --left-align

Выбирает функцию, выполняющую нумерацию, в зависимости от параметра --skip-empty

Читает содержимое файла и разбивает его на строки

Нумерует строки с помощью выбранной функции нумерации и преобразует их в понятную человеку форму



Действие `main` вполне подходит для выполнения главной цели программы. Будет ли оно выполняться, зависит от существования файла с указанным в командной строке именем.

Упражнение. Направления развития

Как вы, наверное, поняли, мы не использовали функцию `numberAndIncrementNonEmptyLines`, которая не только оставляет пустые строки непронумерованными, но и не учитывает их при нумерации. Кроме того, наш способ разбора аргументов нельзя назвать совершенным. Мы не проверяем допустимость аргументов и не предоставляем параметр `--help` для отображения справки. Добавьте все это в программу!

Мы также можем подумать о дополнительных параметрах и функциональности нашей скромной утилиты. Например, можно было бы добавить параметр, подавляющий саму печать пустых строк или включающий режим увеличения счетчика только для абзацев. Количество вариантов безгранично! Если вам нужен источник вдохновения, взгляните на параметры утилиты GNU `nl`. Проявите фантазию!

Библиотечный модуль включает определения типов данных для нумерации строк и соответствующие вспомогательные функции. Список экспорта модуля должен выглядеть так:

```
module Lib
  ( NumberedLine,
    NumberedLines,
    isEmpty,
    isNotEmpty,
    numberLines,
    numberAllLines,
    numberNonEmptyLines,
    numberAndIncrementNonEmptyLines,
    prettyNumberedLines,
```

```
    PadMode (..),
    pad,
    padLeft,
    padRight,
  )
where
```

Определения, относящиеся к пронумерованным строкам, приведены в следующем листинге.

Листинг 4.17 Заголовок, типы данных и вспомогательные функции в библиотечном модуле

```
import Data.Char
```
Импортирует модуль Data.Char

```
type NumberedLine = (Maybe Int, String)

type NumberedLines = [NumberedLine]
```
Определяет типы для нумерации строк

```
isEmpty :: String -> Bool
isEmpty str =
  null str
    || all (\s -> not (isPrint s) || isSeparator s) str
```
Определяет функции, которые можно использовать в булевых предикатах для решения о том, пуста ли строка

```
isNotEmpty :: String -> Bool
isNotEmpty str = not (isEmpty str)
```
Определяет функции, которые можно использовать в булевых предикатах для решения о том, пуста ли строка

Там же находятся функции для создания значений типа `NumberedLines` из списка строк. Они составляют интерфейс для взаимодействия с нашим модулем.

Листинг 4.18 Функции для создания нумерованных строк из списка строк

Определяет функцию для нумерации строк с использованием двух предикатов для инкрементирования счетчика строки и отображения номеров

```
numberLines :: (String -> Bool) -> (String -> Bool) -> [String] ->
➥ NumberedLines
numberLines shouldIncr shouldNumber text =
  let go :: Int -> [String] -> NumberedLines
      go _ [] = []
      go counter (x : xs) =
        let mNumbering = if shouldNumber x then
➥ Just counter else Nothing
            newCounter = if shouldIncr x then
➥ counter + 1 else counter
         in (mNumbering, x) : go newCounter xs
    in go 1 text
```
Локально определяет рекурсивную вспомогательную функцию

Вычисляет номер строки, а также следующий номер на рекурсивном шаге с учетом булевых предикатов

```
numberAllLines :: [String] -> NumberedLines
numberAllLines = numberLines (const True) (const True)

numberNonEmptyLines :: [String] -> NumberedLines
numberNonEmptyLines = numberLines (const True) isNotEmpty

numberAndIncrementNonEmptyLines :: [String] -> NumberedLines
numberAndIncrementNonEmptyLines = numberLines isNotEmpty isNotEmpty
```

Определяет конкретные функции для создания номеров строк на основе более общего определения

Наконец, мы имеем функции, которые преобразуют пронумерованные строки в понятную человеку форму. Эта часть модуля показана в листинге ниже. Заметим, что в принципе этот код можно было бы повторно использовать и в других программах! Именно поэтому он реализован в виде библиотечного модуля.

Листинг 4.19 Функции для создания отформатированных строк, готовых к печати

Определяет функцию для преобразования пронумерованных строк в понятную человеку форму

Вычисляет максимальную длину номеров строк

Разбивает пронумерованные строки на номера и строки

```
prettyNumberedLines :: PadMode -> NumberedLines -> [String]
prettyNumberedLines mode lineNums =
  let (numbers, text) = unzip lineNums
      numberStrings = map (maybe "" show) numbers
      maxLength = maximum (map length numberStrings)
      paddedNumbers = map (pad mode maxLength) numberStrings
  in zipWith (\n l -> n ++ " " ++ l) paddedNumbers text

data PadMode = PadLeft | PadRight

pad :: PadMode -> Int -> String -> String
pad mode n str =
  let diff = n - length str
      padding = replicate diff ' '
  in case mode of
       PadLeft -> padding ++ str
       PadRight -> str ++ padding

padLeft :: Int -> String -> String
padLeft = pad PadLeft

padRight :: Int -> String -> String
padRight = pad PadRight
```

Преобразует номера строк в строковое представление

Дополняет номера строк до максимальной длины и объединяет их с самими строками

Определяет тип данных для различения дополнения слева и справа

Определяет функцию для дополнения строки до указанной длины пробелами

Определяет функции для дополнения слева или справа на основе более общего определения

Добавляет дополнение слева или справа в зависимости от режима PadMode

Важно понимать, что мы инкапсулируем *чистый код* в библиотечном модуле и включаем его в *нечистый код* в исполняемом модуле. Мы не против смешения чистоты с нечистотой, но самые важные функции, которые реализуют основную функциональность, охраняем от любых побочных эффектов!

Резюме

- `readFile` и `writeFile` используются для чтения и записи файлов.
- Функции могут принимать в качестве аргументов другие функции, способные влиять на потоки управления и данных.
- `let` можно использовать для определения вспомогательных функций внутри определений функций, что бывает полезно для сокрытия от внешней функции дополнительных аргументов, необходимых для рекурсивных вызовов.
- *Эта-редукции* можно использовать для уменьшения числа параметров функции с целью создания более лаконичных определений.
- *Частичное применение функции* позволяет создавать одни функции из других, предоставляя не все необходимые аргументы.
- `data` используется для определения *алгебраических типов данных*, которые являются помеченными объединениями, позволяющими моделировать типы с различимыми значениями.
- `zip` и `zipWith` применяются для поэлементного объединения двух списков.
- Функцию `mapM` можно использовать с действиями так же, как `map` используется с чистыми функциями. `mapM_` – специальная версия `mapM`, которая игнорирует возвращенные действиями результаты.

5 Слова и графы

Краткое содержание этой главы:

- моделирование структуры графа в Haskell;
- гарантирование инвариантов типов данных посредством сокрытия их конструирования и модификации;
- повторное использование реализаций с минимальными накладными расходами;
- построение простых преобразований строк.

В предыдущем проекте мы создали минималистский клон популярной утилиты UNIX. Серьезное дело! Но жизнь не ограничивается написанием качественных утилит для нумерации строк на экране терминала. Давайте немного развлечемся. А как люди развлекаются? Играют в игры, конечно!

Есть забавная игра – *цепочка слов*, в которой от игрока требуется построить последовательность слов, меняя на каждом шаге одну букву. У этой игры много вариантов, мы займемся довольно сложным. Но зачем играть самому, когда это можно поручить компьютеру?

Создание искусственного интеллекта, пусть даже для детской игры, – это вам не легкая прогулка. Этот проект заставит нас задуматься об изобретательном использовании структур данных для решения задачи поиска. На первый взгляд, ничего трудного, но мы увидим, что и в детской игре встречаются подводные камни.

Мы начнем эту главу с обсуждения того, как можно моделировать графы с помощью типов Haskell и как создавать собственные модули. Затем мы изучим основы классов типов: что это такое, что они делают и как ими пользоваться. После этого мы создадим специальный тип данных для отображений, используя ассоциативные списки и список экспорта модулей для гарантирования инвариантов типа. С помощью отображений мы построим тип данных для графов и справочную таблицу перестановок слов из списка.

5.1 Построение графа

Игра в цепочку слов – это интересное упражнение, заставляющее игроков (а участвовать может сколько угодно человек) расширять свой словарный запас. У правил игры есть несколько вариантов, мы начнем с самого простого, а затем перейдем к обсуждению того, как можно было бы разработать для игры искусственный интеллект.

Игроки начинают с двух слов одинаковой длины. Одно можно считать начальным, другое – конечным. Задача в том, чтобы найти цепочку других слов, связывающую начальное слово с конечным и такую, что любые два соседних слова отличаются одной буквой. То есть игрок начинает с какого-то слова, изменяет в нем одну букву, получая тем самым следующее слово, и продолжает действовать таким образом, пока не придет к конечному слову. Если играет несколько человек, то побеждает тот, кто найдет самую короткую цепочку. Например: cat → sat → sag → dag → dog.

Игра забавная, но для компьютера является тривиальным упражнением, как мы скоро увидим. На самом деле это классическая *задача поиска*. Чтобы найти решение, мы должны отыскать в некоторой области (на множестве слов) путь из одного элемента в другой. Такого рода задачи возникают во многих приложениях:

- навигационных системах;
- базах данных;
- сетевых маршрутизаторах;
- решателях головоломок судоку.

Решая игру в цепочку слов, мы обретаем навык, который можно перенести в любую из этих областей. Для компьютера поиск – это поиск, что бы ни являлось его целью. Различие в том, как моделируется решение. Для нас слова связаны, потому что в игре есть определенные правила, а в навигационной системе пути определяются картой. Если убрать этот уровень абстракции, то задачи ничем не отличаются.

Чтобы задача не казалась слишком простой, мы немного усложним ее: позволим игроку не только менять одну букву, но и добав-

лять буквы, удалять буквы и произвольным образом менять порядок букв. При такой модификации игра становится гораздо интереснее, поскольку теперь можно искать пути между словами разной длины. Эти правила позволяют найти путь между словами *find* и *solution* (например, find → fins → ions → loins → tonsil → lotions → solution). Разложение данного решения на шаги показано на рис. 5.1.



Рис. 5.1 Решение для пары слов *find* и *solution* в модифицированной игре в цепочку слов

Эта модификация не только позволяет находить более интересные решения, но и несколько усложняет решение задачи на компьютере. Теперь придется подумать о производительности и хитроумных способах избавиться от ненужной работы.

5.1.1 Полиморфные типы

Подумаем о том, как построить для такой игры искусственный интеллект. Предполагается, что наш интеллект располагает полным списком английских слов. Чтобы найти правильную цепочку, ему придется отыскать путь от одного слова до другого, содержащий только допустимые преобразования. Для этого мы можем предполагать, что все слова являются вершинами графа, в котором два слова соединены ребром тогда и только тогда, когда одно достижимо из другого за один шаг. Такой граф для нескольких слов показан на рис. 5.2. Как видим, для нахождения цепочки от *cat* к *dog* нужно найти путь в этом графе.

Сразу же бросаются в глаза несколько путей от *cat* к *dog*. Один такой путь: cat → hat → heat → heal → held → dole → does → dogs → dog. Но это не самое короткое решение! Например, путь cat → cats → tags → goat → got → dog короче. Наш искусственный интеллект должен уметь находить кратчайший путь в таком *графе цепочек*.

Легко прикинуть, что должна делать программа.

1. Получить от пользователя начальное и конечное слова.
2. Построить граф цепочек на основе словаря.
3. Найти в графе цепочек кратчайший путь от одного слова к другому.

К счастью, задача о нахождении кратчайших путей очень хорошо изучена в информатике, так что тут проблем возникнуть не должно. А *настоящая* проблема – как вычислить граф цепочек. И тут мы должны ответить на вопрос, как представить *граф* в Haskell.

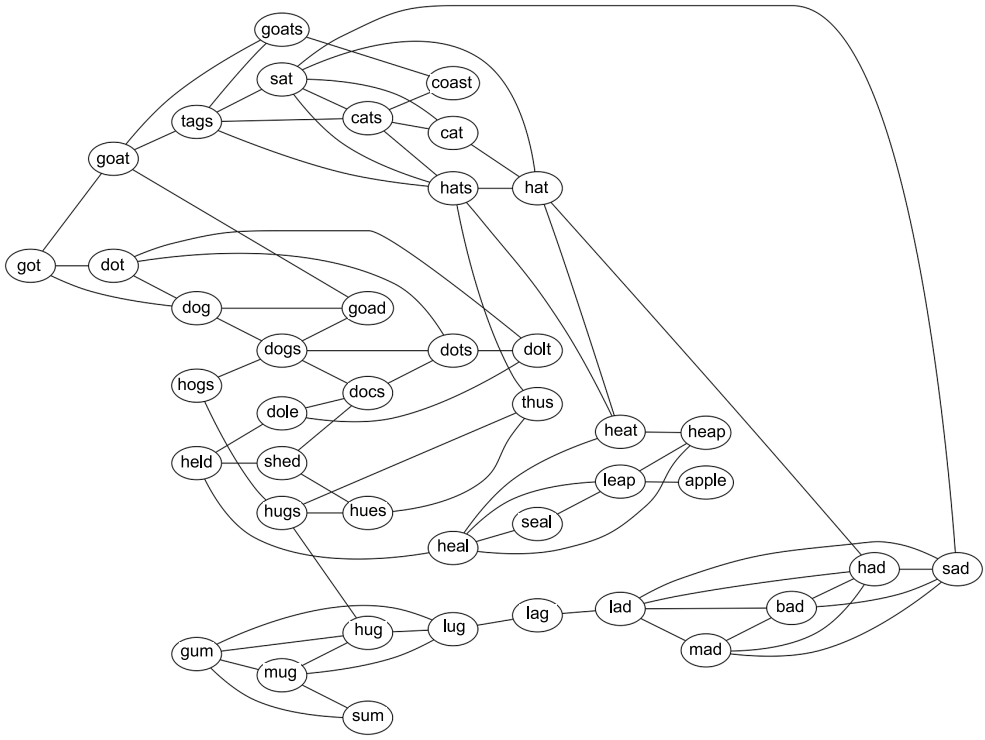


Рис. 5.2 Граф цепочек для подмножества английских слов

Вспомним, что такое графы. Граф состоит из *вершин* и соединяющих их *ребер*. Ребра могут быть ориентированными (т. е. путь по такому ребру может проходить только в одну сторону) или неориентированными. Кроме того, с ребром может быть связана стоимость, но в нашей задаче это не нужно. Мы трактуем все ребра, составляющие путь, одинаково: одно ребро представляет один шаг игры. Это дает нам много возможностей представить концепцию графа типом Haskell. Прежде всего какого рода элементы мы хотим хранить в графе? Какой *тип* будут иметь эти элементы? Можно было бы выбрать фиксированный тип, но в этом нет необходимости. В общем

случае элементы графа могут иметь разные типы, поэтому сделаем тип *полиморфным*:

```
type Graph a = ...
```

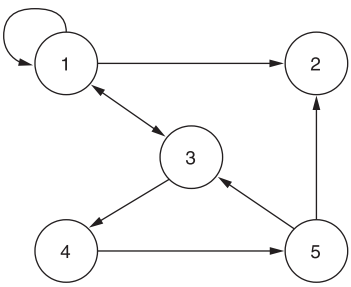
Это позволит нам более гибко решать, какие типы можно использовать в графах, но как смоделировать сам граф? Поскольку мы имеем дело с неориентированным графом и ребра не имеют стоимости, нас интересует только хранение информации о том, что две вершины соединены. Это можно сделать с помощью *списка смежности*. Такой список содержит все пары соединенных вершин. В предыдущей главе мы уже встречали тип, которым можно представить такую структуру данных: ассоциативный список!

```
type Graph a = [(a,a)]
```

Две вершины соединены ребром тогда и только тогда, когда в списке присутствует содержащий их кортеж. Это совсем простой тип, но у него есть несколько недостатков в плане производительности. В худшем случае для проверки существования ребра, а также для нахождения всех непосредственных потомков данного узла придется просмотреть весь список. То же самое касается вставки нового узла в список, поскольку необходимо проверять дубликаты. Для больших графов это не годится, а поскольку мы хотим работать с целыми словарями, от этой идеи придется отказаться. Для эффективного обхода нам нужна структура данных, допускающая быстрое индексирование. Гораздо лучшим решением может оказаться *карта смежности*, хотя многое зависит от ее реализации:

```
type DiGraph a = [(a, [a])]
```

Этот тип отображает вершину в список вершин; это подразумевает несколько ребер, но только в одном направлении! Поэтому тип называется `DiGraph`, чтобы подчеркнуть, что граф ориентированный. На рис. 5.3 показано, как карта представляет структуру конкретного графа.



Карта смежности	
1	1,2,3
2	
3	1,4
4	5
5	2,3

Рис. 5.3 Пример графа, построенного по карте смежности

Печально, но у этого типа есть недостаток, связанный с представлением неориентированных графов. Чтобы добавить одно неориентированное ребро, мы должны добавить два элемента в карту. В случае простого неориентированного графа с двумя соединенными вершинами `1` и `2` карта имела бы вид `[(1,[2]),(2,[1])]`.

Пока что удовлетворимся этим типом, потому что ориентированный граф удобен для нашей задачи поиска (как мы скоро увидим). Мы хотим создать не только этот тип, но и набор функций для работы с ним. Поэтому мы хотим собрать всю взаимосвязанную функциональность в один модуль – свой собственный! Начнем с создания нового проекта, который назовем `ladder`.

5.1.2 Знакомство с новым модулем

После создания нового проекта командой `stack new ladder` создадим файл в каталоге `src`, который будет играть роль модуля для типа графа. Назовем этот файл `Graph.hs` и будем хранить в нем все относящиеся к графу определения:

```
module Graph where
```

Напомним, что имя модуля должно коррелировать с именем файла.

> **ПРИМЕЧАНИЕ** Модули могут находиться в подкаталогах каталога исходного кода. Если модуль находится в файле вида `Foo/Bar/Module.hs`, то имя должно отражать этот факт: `Foo.Bar.Module`. Имена подкаталогов обычно начинаются с заглавной буквы.

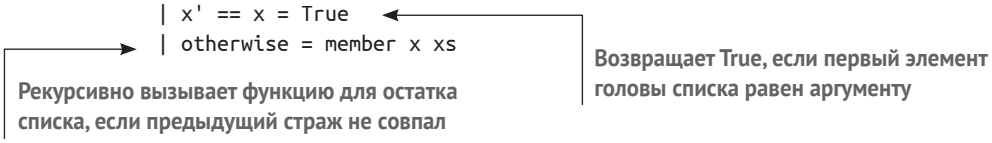
Теперь можно подумать о том, какие функции мы хотим реализовать. Во-первых, обратим внимание на построение графа. Как мы стали бы добавлять в граф одну вершину? Достаточно ли просто добавить в список ассоциативный кортеж? Не совсем, потому что мы хотим быть уверены, что один и тот же ключ не встречается в списке дважды. Напомним, что ассоциативный список работает как отображение. В нашем случае вершины отображаются на другие вершины, с которыми они соединены. Поэтому для начала нужно написать функцию, которая ищет ключ в отображении, а затем использует его в нашей функции вставки. Эта функция показана в листинге ниже.

Листинг 5.1 Функция, проверяющая, присутствует ли ключ в ассоциативном списке

```
member _ [] = False
member x ((x', _) : xs)
```

Возвращает False, потому что в пустом ассоциативном списке нет ключей

Сравнивает кортеж, состоящий из входного списка и первого элемента, называя первый элемент кортежа x'



Во втором образце этой функции мы видим пример сопоставления с образцом для кортежа в списке. Первый элемент кортежа (который сам по себе является первым элементом списка) назван `x'`, а второй элемент игнорируется.

Выражение типа для этой функции намеренно опущено, чтобы мы задумались над тем, как этот тип должен выглядеть. Мы хотим определить функцию для общих ассоциативных списков (чтобы впоследствии использовать это определение для нашего типа графа). Следовательно, тип ассоциативного списка и искомого элемента должен быть полиморфным:

```
member :: a -> [(a, b)] -> Bool
```

Это выражение вроде бы имеет смысл. Ключи и искомый элемент должны быть одного типа, тогда как вторые элементы кортежей могут иметь любой тип. Однако при попытке присвоить этот тип функции и откомпилировать код (например, командой `stack repl`) мы получаем ошибку!

```
...
   • No instance for (Eq a) arising from a use of '=='
     Possible fix:
      add (Eq a) to the context of
        the type signature for:
          member :: forall a b. a -> [(a, b)] -> Bool
...
```

О чем это? Ошибка связана с использованием функции `(==)`. Мы предполагаем, что тип ключей свободно полиморфен, но как мы можем быть уверены, что тип функции используется в сочетании со значениями, допускающими сравнение? Как в общем случае узнать, значения каких типов можно сравнивать?

> **ПРИМЕЧАНИЕ** В Haskell операторы вроде `==` или `&&` являются функциями, и мы можем использовать их, как любые другие! Компилятор интерпретирует их как записанные в инфиксной нотации, но ничто не мешает обращаться к ним, как к другим функциям, заключив оператор в скобки, например `(==)`. А значит, их можно передавать функциям высшего порядка (например, результатом вычисления `zipWith (==) [1,2,3] [1,1,3]` является `[True,False,True]`).

Чтобы разобраться в этой головоломке, мы должны поговорить о *классах типов*. Что это такое?

5.1.3 Класс типов Eq и ограничения типа

Начнем с простого вопроса: какие свойства есть у типов? Сами типы – это просто имена для коллекции значений (в редких случаях значения даже могут отсутствовать). Но из того, что значение принадлежит некоторому типу, еще не следует, что мы знаем, какие операции над ним можно производить. В большинстве языков для некоторых типов подразумеваются определенные операции, например оператор равенства в C и Java определен для всех примитивных типов данных. Но в Haskell необходимо явно определять операции для типов. Это делается с помощью *классов типов*; они содержат сигнатуры функций, которые должны быть определены для *экземпляра* этого класса. Мы можем проверить это в GHCi:

```
ghci> :info Eq
type Eq :: * -> Constraint
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
```

Это говорит нам, что в классе типов `Eq` определено два метода: для сравнения на равенство (`==`) и на неравенство (`/=`). Из сигнатуры типа оператора равенства (`a -> a -> Bool`) мы сразу можем сделать вывод, что разрешено сравнивать только значения *того же типа*, для которого существует *экземпляр* этого класса типов. Например, сравнивать `Int` с `Float` нельзя!

Экземпляр типа этого класса должен предоставлять только реализацию одного из них, на что указывает прагма `MINIMAL`. Отсутствующий метод выводится автоматически путем отрицания результата имеющегося метода. Таким образом, когда мы определяем оператор равенства (или неравенства) для типа, мы автоматически получаем и противоположную функциональность!

Экземпляры, существующие в нашей области видимости, перечислены в том же списке, что эта сигнатура. Вот некоторые из них:

```
instance Eq Bool
instance Eq Char
instance Eq Integer
instance Eq Int
instance Eq Float
```

Здесь мы видим знакомые типы, с которыми уже работали. Каждый экземпляр определяет, как проверять на равенство значения соответствующих типов. Далее мы видим

```
instance Eq a => Eq (Maybe a)
instance Eq a => Eq [a]
```

У некоторых экземпляров имеются *ограничения типа* в качестве предусловий. Эти два примера говорят, что если у некоторого типа `a` имеется экземпляр класса типов `Eq`, то он есть и у `Maybe a` и `[a]` для того же типа. Это означает, что `Maybe Float` и `[Int]` также можно сравнивать!

> **ПРЕДУПРЕЖДЕНИЕ** Терминология для классов типов частично пересекается с объектно ориентированным программированием. Однако эти концепции ни в коем случае не следует путать! Классы типов нельзя инстанцировать, получая объекты, а методы ведут себя не так, как методы классов, поскольку их реализация в разных типах может быть различна. Классы типов больше походят на объектно ориентированные интерфейсы, а не классы.

Существует еще много классов типов, и с некоторыми из них мы познакомимся в свое время, а сейчас сконцентрируемся на нахождении выражения типа для функции `member`. Если мы хотим указать, что тип искомого элемента должен допускать сравнение на равенство, то он должен иметь экземпляр класса типов `Eq`. Это можно сделать, добавив *ограничение типа* к нашему выражению типа:

```
member :: Eq a => a -> [(a, b)] -> Bool
```

Ограничение определяет, какими свойствами должны обладать используемые нами полиморфные типы. Заметим, что когда мы именуем типы явно (например, `Int`), добавлять ограничение не нужно, потому что уже известно, какими экземплярами обладает этот тип. Также заметим, что методы класса типов неявно уже содержат свое ограничение типа:

```
ghci> :type (==)
(==) :: Eq a => a -> a -> Bool
```

Если мы собираемся использовать `(==)`, то полиморфный тип должен иметь экземпляр класса типов `Eq`. Вот почему мы получили ошибку компиляции!

> **ПРИМЕЧАНИЕ** В предыдущих главах мы иногда сознательно опускали выражения типа, чтобы наши функции компилировались. Мы делали так, чтобы Haskell мог вывести ограничения самостоятельно. В большинстве случаев механизм вывода типа способен на это.

5.1.4 Функция flip

Закончив экскурсию, мы можем наконец реализовать функцию добавления новой вершины в граф. Чтобы не путаться в именах, мы определим новую функцию, `hasNode`, которая отличается от `member` только порядком аргументов. Теперь функция добавления новой вершины просто проверяет, существует ли уже такая вершина в графе, и если нет, то добавляет ее без исходящих ребер. Соответствующий код показан в листинге ниже.

Листинг 5.2 Функция, проверяющая, присутствует ли ключ в ассоциативном списке

```
member :: Eq a => a -> [(a, b)] -> Bool
member _ [] = False
member x ((x', _) : xs)
  | x' == x = True
  | otherwise = member x xs

hasNode :: Eq a => DiGraph a -> a -> Bool
hasNode = flip member

addNode :: Eq a => DiGraph a -> a -> DiGraph a
addNode graph node
  | graph `hasNode` node = graph
  | otherwise = (node, []) : graph
```

Определяет функцию, проверяющую, присутствует ли ключ в ассоциативном списке

Определяет функцию, проверяющую, присутствует ли ключ в ориентированном графе

Меняет порядок аргументов member, порождая определения hasNode

Возвращает неизмененный граф, если он уже содержит добавляемую вершину

Возвращает граф с добавленной вершиной, если ее еще не было в графе

Обратите внимание на порядок аргументов наших функций. У `hasNode` граф является первым аргументом, так чтобы ее можно было записывать в инфиксной нотации. Для изменения порядка аргументов мы воспользовались функцией `flip`:

```
flip :: (a -> b -> c) -> b -> a -> c
```

Эта функция принимает произвольную бинарную функцию и порождает функцию с переставленными аргументами! Обратите внимание, что `a` и `b` – аргументы функции и что, пользуясь частичным применением функции, `flip`, получающая одну бинарную функцию, создает новую функцию типа `b -> a -> c`.

Упражнение. Функция lookup

Модуль `Data.List` (как и уже импортированный модуль `Prelude`) предоставляет функцию `lookup` для поиска ключей в ассоциативном списке. Воспользовавшись ей, перепишите функцию `member`.

Глядя на выражения типа, мы видим, что должны добавить `Eq` в ограничение для каждой функции, поскольку мы хотим использовать другие функции (например, `(==)`) с тем же ограничением внутри их определений. Мы должны гарантировать, что типы аргументов наших функций удовлетворяют всем ограничениям, подразумеваемым в соответствующих определениях. Если бы мы не добавили ограничение `Eq`, это означало бы, что можно использовать произвольные типы, даже те, которые нельзя сравнивать на равенство, внутри функции, где такое сравнение производится, а это недопустимо.

> **СОВЕТ** Иногда вы можете встретить функцию, для которой в ограничении типа указан незнакомый вам класс типов. Я считаю, что самый быстрый способ познакомиться с классом – узнать, какие у него есть методы и экземпляры, – воспользоваться GHCi для инспекции класса. Это позволяет сделать команда `:i <имя класса>`.

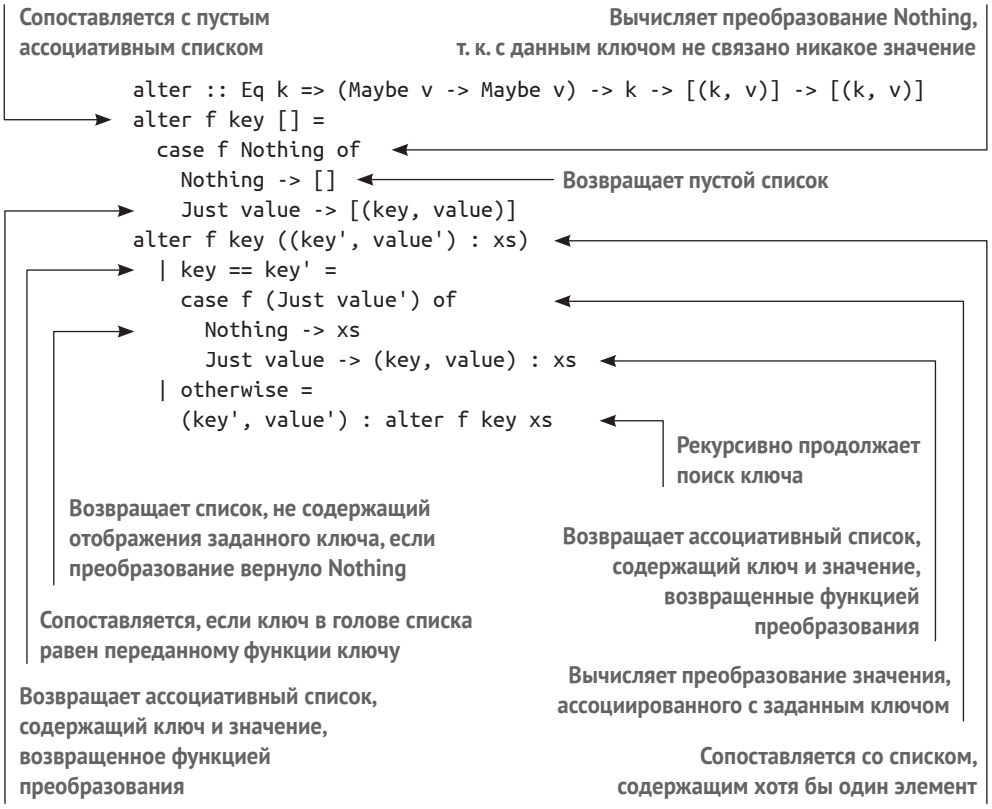
Теперь, зная, как использовать классы типов, мы можем заняться более сложной задачей – построением графа. Нужно подумать, какие функции работы с ассоциативными списками следует добавить в модуль.

5.2 Инкапсуляция реализации

Для построения графа мы хотим написать новую функцию `addEdge`, которая добавляет одно ребро между двумя вершинами графа. Для этого необходимо проверить, присутствует ли начальная вершина в графе. Мы должны будем либо добавить эту вершину в граф, снабдив ее списком ребер, содержащим только конечную вершину, либо – если вершина существует – модифицировать список исходящих из нее ребер. Однако при этом нужно каким-то образом *изменить* карту. Чтобы упростить задачу, мы хотим создать функцию, вносящую произвольные изменения в карту. В идеале она должна уметь добавлять новые элементы, удалять элементы и модифицировать значение указанного ключа. Хорошая идея – использовать функцию высшего порядка, которая принимает функцию, управляющую изменением в структуре данных.

Эта функция должна принимать информацию о том, существует ли уже значение для заданного ключа, и передавать информацию о том, должно ли это значение быть модифицировано или удалено, нашей функции высшего порядка. Оба шага можно выполнить с использованием типа `Maybe`. Чтобы лучше разобраться, взгляните на реализацию ниже.

Листинг 5.3 Функция модификации ассоциативного списка



Функция `f` определяет, что делать. Она получает `Nothing`, если для данного ключа не существует значения, и `Just` с определенным значением в противном случае. Функция возвращает либо `Nothing`, означающее, что если для ключа существует значение, то его следует удалить, либо `Just`, содержащее обновление значения в карте.

Если список пуст, то заданный ключ отсутствует, поэтому с ним точно не связано никакое значение. В таком случае мы передаем `f` `Nothing` и либо оставляем список пустым, либо добавляем в него новую запись. Если список не пуст, то мы рекурсивно ищем заданный ключ и, найдя его, проверяем результат функции. `Nothing` означает, что запись следует удалить из карты, а значение, обернутое конструктором `Just`, – что ее следует обновить. Чтобы назначение типов и значений было понятнее, мы назвали свободные переменные-типы `k` и `v`, т. е. *key* (ключ) и *value* (значение).

> **ПРИМЕЧАНИЕ** Легко видеть, что мы никогда не «модифицируем» ассоциативный список в функции `alter`. Функция строит совершенно новый список, содержащий нужную нам модификацию. А сам список *неизменяемый*. Так проявляется *чистота* Haskell, и именно так мы решаем проблему состояния.

5.2.1 Добавление и удаление записей

Посмотрим, как эту функцию можно использовать для удаления, обновления и добавления записей в наш список. Протестируем ее на простом ассоциативном списке:

```
ghci> myAssocList = [(1,1), (2,2), (3,3)] :: [(Int, Int)]
```

Удаление работает благодаря тому, что переданная в качестве аргумента функция возвращает `Nothing`. Если мы всегда будем возвращать `Nothing` (используя `\x -> Nothing` или `const Nothing`), то сможем воспользоваться функцией для удаления записи, связанной с данным ключом:

```
ghci> alter (const Nothing) 1 myAssocList
[(2,2),(3,3)]
```

Новое значение добавляется путем возврата конструктора `Just`, обертывающего значение, которое мы хотим связать с ключом:

```
ghci> alter (const (Just 4)) 4 myAssocList
[(1,1),(2,2),(3,3),(4,4)]
```

Использовать эту функцию можно также для обновления уже существующих записей. Мы можем воспользоваться функцией `maybe`, чтобы либо передать `Nothing`, если функции было передано `Nothing`, либо изменить значение, если функции было передано значение, обернутое `Just`:

```
ghci> alter (maybe Nothing (const (Just 0))) 1 myAssocList
[(1,0),(2,2),(3,3)]
ghci> alter (maybe Nothing (const (Just 0))) 4 myAssocList
[(1,1),(2,2),(3,3)]
```

Если мы не хотим явно проверять, существует ли уже значение в ассоциативном списке, то оно будет добавлено автоматически, если не существует. Это весьма полезная функция, на основе которой мы можем построить много разных утилит для работы с ассоциативным списком.

Упражнение. Другая реализация alter

В разделе 3.3.2 мы узнали о функции `maybe` для работы со значениями типа `Maybe`. Однако в нашей реализации `alter` мы прибегли к ручному сопоставлению с образцами. Перепишите ее, используя вместо этого `maybe`!

Прежде чем добавлять эти функции в состав вспомогательных, нам надо бы навести порядок в коде. Ясно, что мы определяем функ-

ции не для графов, а для ассоциативных списков общего вида. Так определим в коде новый модуль!

5.2.2 Использование списков экспорта для сокрытия конструкторов

Чтобы лучше организовать функциональность, добавим в каталог `src` новый модуль, `AssocMap`, и переместим функции `member` и `alter`. Поместим этот модуль в подкаталог `src`, который назовем `Data`. Тем самым мы хотим подчеркнуть, что создаваемый модуль нужен для определения типа данных и связанных с ним функций. После этого проект будет выглядеть следующим образом:

```
ladder
├── CHANGELOG.md
├── LICENSE
├── README.md
├── Setup.hs
├── app
│ └── Main.hs
├── package.yaml
├── src
│ ├── Data
│ │ └── AssocMap.hs
│ ├── Graph.hs
│ └── Lib.hs
├── stack.yaml
├── stack.yaml.lock
├── test
│ └── Spec.hs
└── test.cabal
```

Цель новой структуры – не просто разделить код для графов и ассоциативных списков, но и гарантировать соблюдение инвариантов. Мы должны быть уверены, что каждый ключ встречается в карте только один раз. В идеале мы хотим выполнять модификацию карты только в ее собственном модуле. В противном случае мы не сможем гарантировать, что другие разработчики, пользующиеся нашим типом, тоже будут следить за соблюдением этого инварианта. Они могли бы построить значение следующим образом:

```
badAssocMap = [(1,1), (1,2)] :: [(Int, Int)]
```

И тогда функции `alter` и `member` перестанут работать. Следовательно, нам хотелось бы как-то скрыть тот факт, что этот тип является просто списком. Для этого мы создадим новый тип с собственным конструктором:

```
data AssocMap k v = AssocMap [(k, v)]
```

Этот тип отличается тем, что содержит всего лишь конструктор, у которого, в свою очередь, есть одно-единственное поле. Для определения типа мы можем воспользоваться еще одним ключевым словом, `newtype`:

```
newtype AssocMap k v = AssocMap [(k, v)]
```

Существуют технические причины, по которым нужно использовать именно `newtype`, а не `data`. Более подробное объяснение различий и тонкостей, связанных с этим ключевым словом, см. в приложении B.

> **ПРИМЕЧАНИЕ** Можно (и так обычно и делают) поименовать конструктор в определении `newtype` так же, как сам тип. Это не обязательно, и вы всегда можете выбрать для конструктора другое имя, но в крупных проектах следование этому соглашению делает понятнее, какое значение с каким типом ассоциировано!

Так как же скрыть конструктор типа от любопытных взглядов извне? Ответ дает *список экспорта*. С его помощью мы можем управлять тем, что экспортируется из модуля, а что остается неизвестным стороннему наблюдателю. Если мы хотим экспортировать тип без конструкторов, то просто записываем его имя, например:

```
module Data.AssocMap
  ( AssocMap,
    member,
    alter,
  )
Where
```

Здесь мы экспортируем тип `AssocMap` и функции `member` и `alter`. Однако конструктор типа `AssocMap` мы не экспортируем. Если бы мы захотели экспортировать также конструктор, то могли бы написать `AssocMap (..)`, чтобы экспортировать все конструкторы этого типа.

Но теперь перед нами встает проблема – наши функции были написаны не для типа `AssocMap`, а для простого списка кортежей. Их необходимо переписать! Вариантов два:

1. добавить новый конструктор в каждое выражение, где обрабатываются ассоциативные списки;
2. построить для каждой функции, работающей с новым типом, обертку, в которой используются старые функции работы со списком.

Первый вариант, пожалуй, более канонический, потому что мы хотим определить функции для типа. Зато второй позволяет строить функции для типа из любой функции, работающей со списком кортежей! Мы выберем второй вариант, поскольку заодно он упрощает чтение кода.

Это также удачная возможность познакомиться с еще одной деталью синтаксиса Haskell: ключевым словом `where`. Мы можем определять внутренние функции, используя слово `where` точно так же, как поступали с ключевым словом `let`, за одним *исключением*: определение находится *после* места использования. Это показано на рис. 5.4.



Рис. 5.4 Фраза `where`

Листинг 5.4 Структура модуля для `AssocMap`

```
module Data.AssocMap
  ( AssocMap,                    <- Экспортирует тип AssocMap, но не его конструктор
    member,                      | Экспортирует функции,
    alter,                       | объявленные в этом модуле
  )
where

newtype AssocMap k v = AssocMap [(k, v)]   <- Определяет новый тип для карты,
                                              построенной на основе
                                              ассоциативного списка

member :: Eq k => k -> AssocMap k v -> Bool
member key (AssocMap xs) = member' key xs
  where
    member' :: Eq k => k -> [(k, v)] -> Bool
    member' _ [] = False
    member' x ((x', _) : xs)
      | x' == x = True
      | otherwise = member' x xs

alter :: Eq k => (Maybe v -> Maybe v) -> k -> AssocMap k v -> AssocMap k v
alter f key (AssocMap xs) = AssocMap (alter' f key xs)
  where
    alter' :: Eq k => (Maybe v -> Maybe v) -> k -> [(k, v)] -> [(k, v)]
    alter' f key [] =
      case f Nothing of
        Nothing -> []
        Just value -> [(key, value)]
    alter' f key ((key', value') : xs)
      | key == key' =
        case f (Just value') of
          Nothing -> xs
          Just value -> (key, value) : xs
      | otherwise =
        (key', value') : alter' f key xs

Разворачивает и снова
обертывает ассоциативный
список в новый тип
и использует для него
написанные ранее функции

Создает локальные
определения, которые
могут использоваться
в выражении перед
ключевым словом where

Создает локальные определения, которые могут
использоваться в выражении перед ключевым словом where
```

Итак, мы надлежащим образом инкапсулировали наш тип и функции, защитив их от доступа извне! Но увы, теперь нашим типом вообще нельзя воспользоваться. Почему? Вопрос: а как создать новый список? Напомним, у нас нет конструктора для этого типа, а значит, мы не можем сконструировать никакого значения типа `AssocMap k v`! Ну и как обойти эту засаду?

Мы могли бы предоставить функцию, которая преобразует ассоциативный список в `AssocMap`, но проще предоставить значение для пустой карты и функции для добавления и удаления записей. По счастью, написанная нами функция `alter` позволяет реализовать это вообще без усилий! Новые функции показаны в листинге ниже.

Листинг 5.5 Определения для создания пустой карты, а также функции удаления и вставки

```
empty :: AssocMap k v
empty = AssocMap []

delete :: Eq k => k -> AssocMap k v -> AssocMap k v
delete = alter (const Nothing)

insert :: Eq k => k -> v -> AssocMap k v -> AssocMap k v
insert key value = alter (const (Just value)) key
```

Определяет пустую карту

Определяет функцию, которая удаляет из карты ключ и значение

Определяет функцию, которая вставляет или обновляет значение, связанное с ключом в карте

Не забудем добавить новые определения в список экспорта:

```
module Data.AssocMap
  ( AssocMap,
    empty,
    member,
    alter,
    delete,
    insert,
  )
where
```

Вот теперь можно попробовать! Загрузим проект в GHCi и протестируем его:

```
ghci> insert 'a' "Hello" (insert 'b' "World" empty)
<interactive>:70:1: error:

  • No instance for (Show (AssocMap Char String))
    arising from a use of 'print'
  • In a stmt of an interactive GHCi command: print it
```

Ой! Похоже, мы снова забыли о классе типов.

5.2.3 Класс типов Show

Класс типов `Show` предоставляет функцию `show`, которая преобразует тип Haskell в `String`. Именно эту функцию использует GHCi для отображения вычисленных им значений, но у нашего типа нет экземпляра этого класса.

> **ПРЕДУПРЕЖДЕНИЕ** Функция `show` не предназначена для порождения понятных человеку представлений значений в Haskell! Класс типов `Show` является двойственным к классу типов `Read`, предоставляющему автоматически сгенерированные парсеры значений. Для большинства типов `Show` можно вывести автоматически.

По счастью, мы можем вывести его автоматически! Некоторые классы типов (например, `Eq` и `Show`) допускают автоматический вывод разумных реализаций. Чтобы вывести экземпляр класса типов для типа, используется ключевое слово `deriving`:

```
newtype AssocMap k v = AssocMap [(k, v)]
  deriving (Show)
```

Выведенные экземпляры для `Show` порождают значения типа `String`, которые выглядят практически так же, как значения, которые написали бы мы сами. Заметим, что это работает, только если внутри нашей карты используются типы, которые, в свою очередь, имеют экземпляры класса типов `Show`.

После этого небольшого изменения типа мы наконец можем протестировать функции:

```
ghci> insert 'a' "World" (insert 'b' "Hello" empty)
AssocMap [('b',"World"),('a',"Hello")]
ghci> delete 'a' (insert 'a' "Delete me!" empty)
AssocMap []
```

Попутно заметим, что можно также вывести экземпляр `Eq`! В этом случае равенство (`==`) определяется как *структурная эквивалентность*. Два значения типа равны, если их конструкторы, а значит, и их поля эквивалентны. Приведем несколько примеров:

```
ghci> data X = A | B Int | C Int Int deriving Eq
ghci> A == A
True
ghci> B 1 == B 2
False
ghci> B 1 == B 1
True
ghci> B 1 == C 1 2
```

```
False
ghci> C 1 2 == C 2 2
False
ghci> C 1 2 == C 1 2
True
```

Теперь определим функции для поиска значений, ассоциированных с ключом. В конце концов, именно в этом и состоит цель карты. Для функции `lookup` используется тот же метод, что и раньше: обертывание функции, работающей со списками, новой функцией. Кроме того, определим функцию, которая предоставляет значение по умолчанию, в случае когда ключ отсутствует. Это избавит нас от трудностей впоследствии. Код функций приведен в листинге 5.6.

Мы пишем функцию `lookup`, которая может конфликтовать с одноименной функцией, уже автоматически импортированной из `Prelude`. Чтобы обойти эту проблему, мы можем скрыть функцию `lookup` с помощью следующего предложения импорта:

```
import Prelude hiding (lookup)
```

> **ПРИМЕЧАНИЕ** Если мы не хотим скрывать функцию от импорта (потому что собираемся ее использовать), то должны будем всюду указывать перед именем функции имя модуля. В нашем случае это выглядело бы так: `Prelude.lookup` и `Data.AssocMap.lookup`.

Листинг 5.6 Определения для функций поиска в картах, построенных на основе ассоциативных списков

Сопоставляет со значением AssocMap и выполняет поиск в содержащемся внутри списке

```
lookup :: (Eq k) => k -> AssocMap k v -> Maybe v
lookup key (AssocMap xs) = lookup' key xs
  where
    lookup' _ [] = Nothing
    lookup' key ((key', value) : xs)
      | key == key' = Just value
      | otherwise = lookup' key xs

findWithDefault :: (Eq k) => v -> k -> AssocMap k v -> v
findWithDefault defaultValue key map =
  case lookup key map of
    Nothing -> defaultValue
    Just value -> value
```

Возвращает Nothing, если список пуст

Рекурсивно ищет ключ в ассоциативном списке и возвращает обернутое Just значение, если ключ найден

Возвращает значение по умолчанию, если поиск завершился неудачно

Возвращает найденное значение

Чудесно! Мы написали модуль для карт, построенных на основе ассоциативных списков. Гарантируется, что функции в этом модуле не нарушают инвариант, согласно которому каждый ключ должен

встречаться только один раз, и что карта не может быть «повреждена» извне, поскольку конструктор не экспортируется из модуля. Таким образом, ни сопоставление с образцом, ни конструирование новых карт невозможно в обход наших функций. Теперь можно реализовать граф, пользуясь этим определением карты.

5.3 Использование и повторное использование кода

В модуле `Graph` мы можем импортировать тип данных карты, но нужно помнить, что он экспортирует функцию `lookup`, которая будет конфликтовать с `lookup` из модуля `Prelude`. Это проблема, и ее серьезность только возрастает в более крупных проектах, где модуль может импортировать сотни других модулей.

5.3.1 Квалифицированный импорт

Элегантный способ борьбы с проблемой `lookup` – использование *квалифицированного импорта*:

```
import qualified Data.AssocMap as M
```

Здесь импортируется тип `Data.AssocMap`, которому присваивается имя `M`, поскольку мы не хотим каждый раз выписывать имя целиком. Ключевое слово `qualified` заставляет нас обращаться к функциям и определениям из этого модуля, указывая имя модуля перед их именами. То есть мы больше не можем просто использовать имя `alter`, а должны обращаться к функции по имени `M.alter`. Поскольку с каждым импортом связывается уникальное имя, мы избегаем конфликтов имен, а заодно делаем понятнее, в каком контексте используются некоторые определения!

Теперь воспользуемся таким импортом, чтобы написать определения для ориентированного графа. Сначала определим тип и пустой граф:

```
type DiGraph a = M.AssocMap a [a]
empty :: DiGraph a

empty = M.empty
```

Далее нужно предоставить следующую функциональность:

- добавление ребра в граф;
- добавление нескольких ребер в граф;
- получение всех вершин, связанных с данной.

Для добавления ребра нужно изменить карту смежности. Если записи еще нет, то добавляется вершина, с которой связана одна вершина. В противном случае вершина добавляется к уже существующим вершинам, с которыми связана начальная вершина. Нужно только следить за тем, чтобы в списке связанных вершин не оказалось дубликатов. Для этого можно удалить все дубликаты, и, к счастью, в модуле `Data.List` для этого уже существует функция, `nub`!

```
ghci> import qualified Data.List as L
ghci> L.nub [1,1,1,2,3,4,1,1,1,2,3,4] :: [Int]
[1,2,3,4]
```

Теперь можно написать функцию для добавления ребра. Если ребро отсутствует в графе, то оно добавляется. В противном случае в список связанных вершин добавляется новая вершина, и удаляются дубликаты. Несложно добавить более одного ребра. Нужно только добавить в граф список ребер, вызывая функцию для каждого добавляемого ребра. Кроме того, мы хотим определить функцию, которая преобразует ассоциативный список вершин и списки вершин в граф; ее реализация должна походить на реализацию функции добавления нескольких ребер. Для получения всех вершин, связанных с данной, нужно просто произвести поиск в карте! Если вершина не найдена, мы возвращаем пустой список, поскольку с отсутствующей вершиной ничего не может быть связано! Все эти функции показаны в листинге ниже.

Листинг 5.7 Функции для добавления ребер в ориентированный граф и получения множества вершин, связанных с данной

```
import qualified Data.List as L
...
addEdge :: (Eq a) => (a, a) -> DiGraph a -> DiGraph a
addEdge (node, child) = M.alter insertEdge node
  where
    insertEdge Nothing = Just [child]
    insertEdge (Just nodes) =
      Just (L.nub (child : nodes))

addEdges :: (Eq a) => [(a, a)] -> DiGraph a -> DiGraph a
addEdges [] graph = graph
addEdges (edge : edges) graph = addEdge edge (addEdges edges graph)

buildDiGraph :: (Eq a) => [(a, [a])] -> DiGraph a
buildDiGraph nodes = go nodes M.empty
  where
    go [] graph = graph
```

Импортирует модуль Data.List

Если вершина отсутствует в графе, то новая вершина-потомок образует список связанных вершин и добавляется в граф

Изменяет карту, образующую граф

Рекурсивно добавляет все ребра в граф

Добавляет вершину в множество других связанных вершин

Возвращает неизмененный граф, если добавлять в него нечего

```
    go ((key, value) : xs) graph = M.insert key value
➥ (go xs graph)

children :: (Eq a) => a -> DiGraph a -> [a]
children = M.findWithDefault []
```

Рекурсивно добавляет все пары связанных вершин в граф

Возвращает вершины, связанные с указанной, или пустой список, если вершина отсутствует в графе

Теперь в нашем арсенале есть несколько функций для работы с ориентированными графами. Посмотрим на некоторые из них в действии:

```
ghci> import Graph
ghci> g = addEdges [(1,1), (1,2), (3,2), (3,1)] Graph.empty
ghci> children 3 g :: [Int]
[2,1]
ghci> children 2 g :: [Int]
[]
ghci> children 2 (addEdge (2,1) g) :: [Int]
[1]
```

Преимущество нашей реализации в том, что она абсолютно не зависит от того, как реализована карта. Мы могли бы заменить `AssocMap` другим типом, при условии что в нем имеются совместимые функции.

> **Упражнение. Обобщение функций**
>
> При внимательном изучении функций работы с графами можно заметить, что реализации `addEdges` и `buildDiGraph` очень похожи – и это настораживает. Попробуйте обобщить эти функции, предложив функцию высшего порядка, которая работает со списками, организованными в похожую структуру, и применить эту функцию к обоим определениям. Кроме того, мы не написали функции для удаления вершин и ребер из графа. Реализуйте их самостоятельно.

Теперь мы можем уверенно сказать, что научились строить графы. Вооружившись этими знаниями, построим граф цепочек.

5.3.2 Построение карт перестановок

Напомним, что граф цепочек содержит все слова из заданного словаря, а ребра проведены между словами, которые находятся на расстоянии одного шага в игре в цепочки слов. Шагом называется комбинация следующих преобразований в любом порядке:

- добавление буквы в слово;
- удаление буквы из слова;
- произвольное изменение порядка букв в слове.

Преобразование должно приводить к слову, присутствующему в словаре игры. Это непростая вычислительная задача! Каждое возможное слово необходимо искать во всем словаре. Поскольку буквы можно переставлять любым способом, количество слов, подлежащих проверке, равно факториалу. Для больших слов это реальная проблема.

Очень помог бы какой-то кеш или фильтр, позволяющий быстро определить, какие перестановки допустимы, а какие – нет. В идеале мы вообще не хотели бы вычислять перестановки слова, но как это сделать? Предположим, что мы готовы просмотреть словарь один раз, имея целью сохранить перестановки каждого слова. Какие общие свойства имеются у этих перестановок? Все они состоят из одних и тех же букв! Поэтому после сортировки они должны быть одинаковы. Мы можем просмотреть словарь и создать карту слов и их перестановок, отсортировав каждое слово и используя результат как ключ, с которым ассоциируется список слов. Если проделать это для каждого слова, то будет создано отображение, или карта, ассоциирующая слова с их допустимыми перестановками!

Реализовывать эту идею *карт перестановок* мы начнем с создания модуля, который назовем `PermutationMap`. Этот тип отображает строку (отсортированное представление перестановок) на ее допустимые перестановки:

```
type PermutationMap = M.AssocMap String [String]
```

Операции для работы с этой картой такие же, как для типа `AssocMap`, но мы должны гарантировать, что при любой операции с ключом он сначала сортируется. Для этого можно воспользоваться функцией `sort` из модуля `Data.List`. Как и раньше, мы пользуемся квалифицированным импортом из модулей `Data.AssocMap`, `Data.List` и `Data.Maybe`. Однако из `Data.Maybe` мы хотим импортировать только одну функцию и сделаем это с помощью *списка импорта*. Такой тип списка действует по аналогии со списком экспорта, ограничивая множество импортируемых определений. Код этого модуля показан ниже.

Листинг 5.8 Модуль для карт перестановок

```
module PermutationMap
  ( PermutationMap,
    empty,
    member,
    alter,
    delete,
    insert,
    lookup,
    findWithDefault,
    createPermutationMap,
```

```
  )
where

import qualified Data.AssocMap as M
import qualified Data.List as L
import Data.Maybe (fromMaybe)
import Prelude hiding (lookup)

type PermutationMap = M.AssocMap String [String]

empty :: PermutationMap
empty = M.empty

member :: String -> PermutationMap -> Bool
member key = M.member (L.sort key)

alter ::
  ( Maybe [String] ->
    Maybe [String]
  ) ->
  String ->
  PermutationMap ->
  PermutationMap
alter f key = M.alter f (L.sort key)

delete :: String -> PermutationMap -> PermutationMap
delete key = M.delete (L.sort key)

insert :: String -> [String] -> PermutationMap -> PermutationMap
insert key = M.insert (L.sort key)

lookup :: String -> PermutationMap -> Maybe [String]
lookup key = M.lookup (L.sort key)

findWithDefault :: [String] -> String -> PermutationMap -> [String]
findWithDefault defaultValue key map =
  fromMaybe defaultValue (PermutationMap.lookup key map)
```

Квалифицированно импортирует модули и одну функцию из модуля Data.Maybe

Импортирует модуль Prelude без функции lookup, чтобы избежать конфликта имен

Определяет тип PermutationMap как синоним типа AssocMap

Определяет пустую карту перестановок

Общие функции из AssocMap будут использоваться для доступа к карте перестановок только после сортировки ключа

Поиск со значением по умолчанию

По сути дела, тип `PermutationMap` – просто специализированная версия `AssocMap`, а благодаря нашей полиморфной реализации мы вправе использовать конкретные типы. Кроме того, тип `PermutationMap`, как и `Graph`, полностью независим от реализации карты!

Упражнение. Сортировка строк

Взгляните на выражение типа для функции `sort`. Почему мы можем использовать ее для значений типа `String`? А почему можно использовать наши полиморфные функции из `AssocMap` в новом модуле? Попытайтесь понять, почему типы совместимы, а для этого исследуйте их в GHCi (командами `:type` и `:info`).

Далее мы хотим написать функцию, которая принимает список слов и строит по нему карту перестановок. Можно предполагать, что все слова записаны строчными буквами. Каждое слово необходимо добавить в карту. Мы ожидаем, что будет несколько слов с одинаковым ключом (поскольку в этом и заключается смысл перестановки слов), поэтому слова нужно будет добавлять в список, на который указывает ключ. Соответствующий код приведен ниже.

Листинг 5.9 Функция, которая строит карту перестановок по списку строк

Вызывает вспомогательную функцию с пустой картой

Определяет вспомогательную функцию для добавления каждого нового слова в карту

```
createPermutationMap :: [String] -> PermutationMap
createPermutationMap = go empty
  where
    go permMap [] = permMap
    go permMap (x : xs) = go (insertPermutation x permMap) xs

    insertPermutation word = alter (insertList word) word

    insertList word Nothing = Just [word]
    insertList word (Just words) = Just (word : words)
```

Изменяет карту, используя слово в качестве ключа и функцию добавления слова в список уже встречавшихся значений

Определяет функцию, которая либо создает список из одного слова, если такого слова еще нет в карте, либо добавляет слово в уже существующий список

Эта реализация очень напоминает функцию `addEdges` в листинге 5.7, но на этот раз мы не разделяем функции. Вместо этого мы храним все необходимые определения локально с помощью `where`. Также видно, что механизм функции `go` кажется знакомым. И правда, существует функция с точно таким поведением, но знакомство с ней мы отложим до главы 8!

5.3.3 Создание карты перестановок по словарю

Теперь мы можем протестировать нашу новую функцию:

```
ghci> words = ["traces", "reacts", "crates", "caster", "tool", "loot",
➥ "cat"]
ghci> pm = createPermutationMap words
ghci> pm
AssocMap [("acerst",["caster","crates","reacts","traces"]),("loot",
➥ "loot","tool"]),("act",["cat"])]
ghci> PermutationMap.lookup "tool" pm
Just ["loot","tool"]
ghci> PermutationMap.lookup "reacts" pm
Just ["caster","crates","reacts","traces"]
```

Как видим, каждое слово встречается в списке, ассоциированном с его отсортированным вариантом. После построения карты мы можем быстро получить все перестановки слова, встречавшиеся в исходном списке слов. Вместо вычисления всех перестановок слов и проверки каждой из них нужно произвести один простой список в карте!

5.4 Параметризация преобразований

Теперь, когда у нас есть нечто вычислительно реализуемое, можно приступить к построению графа цепочек. Для этого создадим еще один модуль, `Ladder`, который содержит функции, относящиеся только к рассматриваемой задаче. Для начала определим тип для списка слов:

```
type Dictionary = [String]
```

Это нужно для того, чтобы отличать словарь от обычного списка строк. Мы предполагаем, что словарь содержит слова, записанные строчными буквами. Теперь мы можем написать действие ввода-вывода, которое читает файл со словами и разбирает его как словарь. Мы уже умеем разбивать строки файла на слова, но в записях словаря нужно оставить только строчные буквы. Для этого нам пригодится функция `filter` из модуля `Data.List`. Она получает функцию типа `a -> Bool`, которая работает как булев предикат для элементов списка. Возвращаются только элементы, для которых этот предикат равен `True`, например:

```
ghci> filter (\x -> x <= 5) [1..10]
[1,2,3,4,5]
ghci> filter even [1..10]
[2,4,6,8,10]
```

С помощью этой функции мы можем профильтровать строки, оставив только строчные буквы; для этого мы должны для каждой литеры проверить, является ли она частью алфавита строчных латинских букв:

```
ghci> filter (\x -> x `elem` ['a'..'z']) "hello world."
"helloworld"
```

Это позволяет отфильтровать все остальные литеры. Кроме того, необходимо удалить дубликаты, воспользовавшись функцией `nub`. Код, который все это делает, показан в листинге ниже. Мы также квалифицировали импорт из модулей `Data.List`, `Graph` и `PermutationMap`.

Листинг 5.10 Действие ввода-вывода, которое читает словарь из файла

```
module Ladder
  ( Dictionary,
    readDictionary,
  )
where

import qualified Data.List as L
import qualified Graph as G
import qualified PermutationMap as PM

type Dictionary = [String]

readDictionary :: FilePath -> IO Dictionary
readDictionary filepath = do
  dictionaryContent <- readFile filepath
  let
    lines = L.lines dictionaryContent
    words = L.map (L.filter (`L.elem` ['a' .. 'z'])) lines
  return (L.nub words)
```

Читает файл

Разбивает содержимое файла на строчки

Фильтрует строчки, оставляя только строчные буквы

Возвращает список профильтрованных слов без дубликатов

Здесь мы также видим, как пользоваться инфиксной нотацией для функции (в данном случае `L.elems`) и эта-редукцией, чтобы избавиться от лямбда-абстракции.

Теперь мы хотим использовать слова из словаря для построения графа цепочек. Для этого нужно вычислить все возможные модификации слова (путем добавления букв, удаления букв и изменения одной буквы) и воспользоваться построенной картой перестановок, чтобы вычислить все осмысленные перестановки вновь образованных слов. Функция `buildDiGraph` строит список вершин и соединяющих их ребер путем вычисления всех допустимых новых слов для каждого слова в словаре. В предположении, что мы уже имеем функцию `computeCandidates`, которая возвращает все возможные кандидаты для данного слова, окончательная функция показана в листинге ниже.

Листинг 5.11 Функция построения графа цепочек по словарю

Строит ориентированный граф из словарных слов и слов, связанных с ними по правилам игры

```
mkLadderGraph :: Dictionary -> G.DiGraph String
mkLadderGraph dict = G.buildDiGraph nodes
  where
    map = PM.createPermutationMap dict
    nodes =
      L.map (\w -> (w, computeCandidates map w)) dict
```

Строит карту перестановок по словарю

Отображает каждое слово в список, содержащий его самого и все связанные с ним слова

Однако у нас еще нет необходимой функции `computeCandidates`! Зная слово, мы должны сначала добавить к нему произвольную букву, а затем с помощью карты перестановок получить все допустимые перестановки. Это немного упрощает нашу работу, потому что не имеет значения, в какое место добавлять новую букву. Как добавить букву в строку? Одну из возможностей дает функция `map`:

```
ghci> map (\x -> x : "word") ['a' .. 'z']
["aword","bword","cword","dword","eword","fword","gword","hword","iword",
➥ "jword","kword","lword","mword","nword","oword","pword","qword","rword",
➥ "sword","tword","uword","vword","wword","xword","yword","zword"]
```

Удалить букву также можно с помощью `map`, поскольку мы можем поочередно удалять каждую букву, пользуясь функцией `delete` из модуля `Data.List`. Отметим, что при этом удаляется только первое вхождение удаляемого элемента:

```
ghci> map (\x -> delete x "word") "word"
["ord","wrd","wod","wor"]
```

Опять же нас не интересует, какая буква удаляется, потому что карта перестановки все равно будет сортировать слова при поиске! Для вычисления слов, в которых нужно заменить одну букву, обе операции необходимо скомбинировать, поскольку мы не хотим добавлять только что удаленную букву. Но такой код может оказаться довольно громоздким, поэтому мы воспользуемся другой синтаксической конструкцией для списков: списковыми включениями.

5.4.1 Списковые включения

Списковое включение (list comprehension) позволяет очень просто записывать определения списков. Оно состоит из двух частей: в левой описывается, как строятся элементы списка, а в правой находятся так называемые генераторы и стражи. Например:

```
ghci> [ x + 1 | x <- [1..10] ] :: [Int]
[2,3,4,5,6,7,8,9,10,11]
ghci> [ x + 1 | x <- [1..10], x <= 5 ] :: [Int]
[2,3,4,5,6]
```

Как видим, левая часть вычисляется для каждого элемента генератора, удовлетворяющего сторожевому условию! Если задать несколько генераторов, то левая часть будет вычисляться для декартова произведения генераторов:

```
ghci> [ (x, y) | x <- [1,2], y <- ['a', 'b', 'c']] :: [(Int, Char)]
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c')]
```

С помощью таких списковых включений можно построить модификации слов. Для заданного слова `word` мы можем вычислить возможные новые слова следующим образом:

```
added = [x : word | x <- ['a' .. 'z']]
removed = [delete x word | x <- word]
modified =
  [x : delete y word | x <- ['a' .. 'z'], y <- word, x /= y]
```

Здесь мы видим, как списковые включения можно использовать, чтобы объединить функциональность `map` и `filter`. Кроме того, они позволяют комбинировать списки с декартовым произведением. Количество размерностей произвольно, т. к. можно задать несколько генераторов. Такой способ писать определения функций, работающих со списками, позволяет сделать определения немного короче и потенциально проще для чтения. Впрочем, это дело вкуса.

> **ПРИМЕЧАНИЕ** Списковые включения можно использовать также для сопоставления с образцом. Неудачное сопоставление в генераторе считается пропущенным значением. Таким образом, функцию `catMaybes` можно было бы определить так: `catMaybes xs = [ x | Just x <- xs ]`. Любое значение, не сопоставившееся с `Just x`, просто отбрасывается.

Теперь функцию можно дописать до конца, воспользовавшись определениями выше. Заодно мы можем удалить все дубликаты отсортированных слов, чтобы уменьшить количество операций поиска в карте. И напоследок следует удалить из списка исходное слово, потому что на каждом ходе игры в цепочку слов необходимо изменять слово. Полный исходный код функции приведен в следующем листинге.

Листинг 5.12 Функция вычисления допустимых кандидатов для следующего хода

```
computeCandidates :: PM.PermutationMap -> String -> [String]
computeCandidates map word =
  let candidates = modified ++ removed ++ added ++ [word]
      uniques = L.nub [L.sort w | w <- candidates]
      perms = L.concatMap (\x ->
➥ PM.findWithDefault [] x map) uniques
   in L.delete word perms
  where
    added = [x : word | x <- ['a' .. 'z']]
```

Сортирует всех возможных кандидатов и удаляет дубликаты

Вычисляет все допустимые перестановки для каждого возможного кандидата

Удаляет исходное слово из списка допустимых кандидатов

Вычисляет строки, отличающиеся от исходного слова одной добавленной буквой

```
Вычисляет строки, отличающиеся от исходного слова
одной удаленной буквой
      removed = [L.delete x word | x <- word]
      modified =
        [x : L.delete y word |
➥ x <- ['a' .. 'z'], y <- word, x /= y]
Вычисляет строки, отличающиеся от исходного слова
одной измененной буквой
```

Здесь используется ранее не встречавшаяся функция `concatMap`. Она тесно связана с функцией `concat`, которая просто принимает список списков и *разглаживает* его, создавая путем конкатенации один список, который содержит все элементы исходных списков. `concatMap` – комбинация `map` с последующей `concat`. Это полезно в ситуациях, когда результатом функции `map` является список, но мы хотим объединить все элементы этих списков в один. Поскольку такая ситуация возникает довольно часто, функция включена в библиотеку.

```
ghci> concat [[1,2,3], [4,5,6]] :: [Int]
[1,2,3,4,5,6]
ghci> concat ["Hello", " ", "World"] :: String
"Hello World"
ghci> concatMap (\x -> [1..x]) [1..5] :: [Int]
[1,1,2,1,2,3,1,2,3,4,1,2,3,4,5]
```

Теперь мы можем построить граф цепочек для заданного словаря! Вот небольшой пример:

```
ghci> mkLadderGraph ["cat", "cats", "act", "dog"]
AssocMap [("dog",[]),("act",["cat","cats"]),("cats",["act","cat"]),
➥ ("cat",["act","cats"])]
```

Здесь мы передали функции словарь из четырех слов. Видно, что слова *cat*, *cats* и *act* достижимы за один шаг друг из друга, тогда как *dog* недостижима ни из какой вершины и не имеет соседей в графе. После того как мы научились строить структуру для поиска решения, можно заняться проблемой, составляющей суть нашего искусственного интеллекта: задачей поиска!

Резюме

- Алгебраические типы данных могут содержать свободные переменные типа, которые можно использовать для создания структуры данных, содержащей произвольные типы.
- Имя модуля должно совпадать с путем к файлу в проекте.

- Класс типов `Eq` используется для предоставления функций (`(==)` и `(/=)`) сравнения значений на равенство и неравенство.
- Функция `flip` используется для перестановки аргументов бинарной функции.
- Функции высшего порядка могут полностью изменять свое поведение в зависимости от функции, переданной им в качестве аргумента.
- Мы можем исключать конструкторы типа из списка экспорта модуля, чтобы запретить создание значений типа извне модуля.
- Класс типов `Show` предоставляет функцию `show`, которая используется для преобразования значений в строки.
- Квалифицированный импорт применяется, чтобы заставить нас указывать имя модуля (или его псевдоним) при использовании определений из него.
- Списковые включения предлагают специальный синтаксис для фильтрации, отображения и комбинирования списков.

6 Решение игры в цепочки слов

Краткое содержание этой главы:

- реализация сложных алгоритмов с помощью нескольких вспомогательных функций;
- тонкости использования переменных типов и их области видимости;
- профилирование производительности приложения и добавление в проект внешних зависимостей.

В предыдущей главе мы описали игру в цепочки слов и обсудили, как реализовать структуру данных для представления графов. Мы даже умеем строить граф, представляющий все возможные исходы игры.

Теперь мы собираемся заняться задачей о вычислении решения в этом графе. Мы достигнем своей цели, воспользовавшись алгоритмом поиска, который гарантирует нахождение кратчайшего решения игры.

Мы начнем эту главу с рассмотрения реализации алгоритма поиска в ширину для нашей игры. Затем обсудим переменные типа и языковые расширения. И завершим эту главу разговором о профилировании проекта и о том, как повысить его производительность.

6.1 Поиск в ширину

Как построить граф цепочек для заданного словаря, мы уже знаем; осталась единственная вычислительная задача – произвести в нем поиск. К алгоритму предъявляется дополнительное требование: он должен находить кратчайший путь, соответствующий оптимальной цепочке слов. Наш граф специальный в том смысле, что стоимости всех ребер одинаковы, т. е. на них не проставлены веса. В таком случае гарантируется, что кратчайший путь будет найден алгоритмом *поиска в ширину*.

6.1.1 Общее описание алгоритма

Давайте подумаем о графе. Поиск пути начинается с некоторой вершины. Из нее мы должны посетить всех соседей, а затем повторить этот процесс для каждого еще не посещенного соседа новой вершины. В процессе поиска мы создаем упорядочение на *уровнях* вершин. Такое упорядочение показано на рис. 6.1.

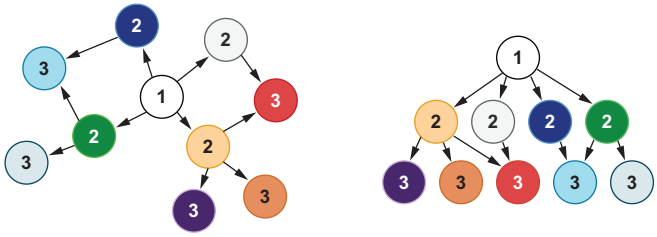


Рис. 6.1 Упорядочение вершин графа, индуцированное поиском в ширину

Для реализации *поиска в ширину* мы должны одновременно обновить рубеж, состоящий из посещаемых на данном шаге вершин. Мы не можем просто осуществить рекурсивный поиск в каждой вершине, потому что это был бы *поиск в глубину*, при котором мы посещаем не всех соседей данной вершины, а продолжаем поиск с первого соседнего узла. То есть в процессе поиска мы должны запоминать вершины, посещаемые на данном шаге, и соответственно обновлять их. Это показано на рис. 6.2.

Для решения игры мы должны не только установить существование пути, но и предъявить все лежащие на нем вершины. Для этого нужно хранить историю просмотренных вершин и запоминать предшественников вершин в процессе поиска. При этом нужно позаботиться о том, чтобы никакая вершина не посещалась дважды, потому что у каждой вершины должен быть только один предшественник; в противном случае нам пришлось бы еще раз произвести поиск во всем графе, чтобы найти тот самый путь, по которому мы

уже прошли. Чтобы противостоять этому, мы можем удалять каждую просмотренную вершину из графа во время поиска. Годится ли это для наших целей? Да! Кратчайший путь из одной вершины в другую не может проходить ни через какую вершину дважды, потому что, удалив из пути вершины, в которые мы заходили между двумя посещениями одной и той же вершины, мы получили бы более короткий путь из начальной вершины в конечную. На рис. 6.3 показано создание этого *графа предшествования*.

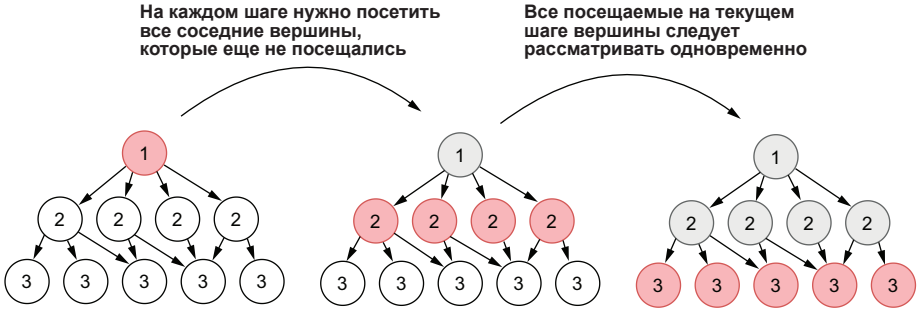


Рис. 6.2 Пример поиска в ширину

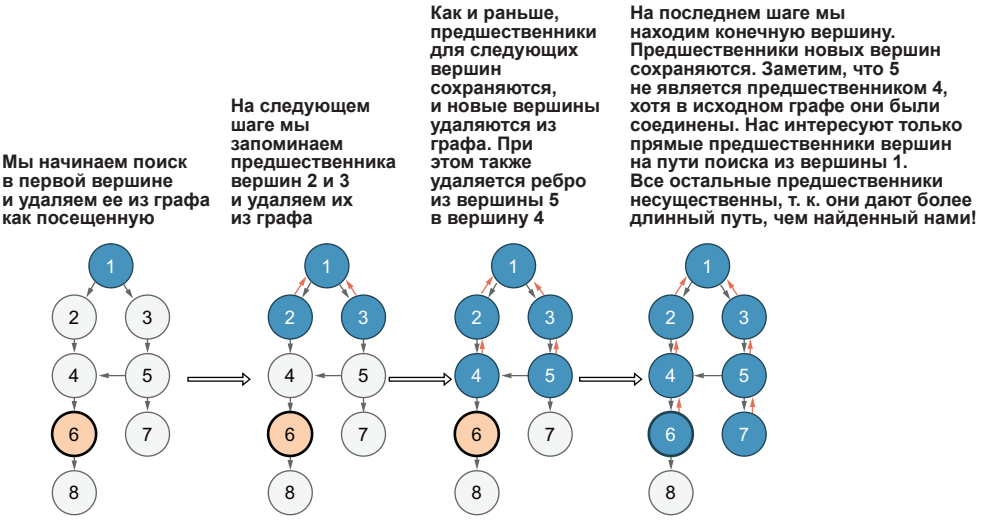


Рис. 6.3 Пример поиска в графе (пути из 1 в 6) и построения графа предшественников на каждом шаге

Повторим. Наш алгоритм поиска должен выполнять следующие действия.

1. Начать с графа и начальной вершины как исходного рубежа.
2. Собрать все соседние вершины для каждой вершины, принадлежащей рубежу.

3. Удалить текущий рубеж из графа.
4. Сохранить все вершины, принадлежащие рубежу, в качестве предшественников соответствующих соседних вершин.
5. Проверить, является ли конечная вершина элементом множества соседних вершин.
   a. Если да, то поиск завершен, и для нахождения пути достаточно проследовать назад по графу предшествования.
   b. Если нет, продолжить поиск с шага 2, считая множество соседних вершин новым рубежом.

Для удаления сразу нескольких вершин нам понадобится новая функция. По аналогии с предыдущими реализациями, мы можем рекурсивно вызывать функцию `delete` из нашего модуля `AssocMap` для каждого элемента, который должен быть удален из графа. Код функции приведен ниже.

Листинг 6.1 Функция удаления нескольких вершин из графа

```
deleteNodes :: Eq a => [a] -> DiGraph a -> DiGraph a
deleteNodes [] graph = graph
deleteNodes (x : xs) graph = M.delete x (deleteNodes xs graph)
```

Возвращает сам граф, если никаких вершин удалять не нужно

Удаляет одну вершину из графа и рекурсивно применяется к остатку графа

Из нашего описания алгоритма поиска ясно, что необходимо обрабатывать некий вид состояния. Мы должны следить за рубежом, а также за графом, в котором производится поиск (поскольку мы удаляем из него вершины), и за предшественниками.

6.1.2 Отслеживание состояния поиска

По счастью, у нас уже есть тип, пригодный для наших целей: `DiGraph`. Таким образом, мы можем представить состояние поиска как отдельный тип, содержащий рубеж в качестве списка элементов, наш граф и еще один граф, используемый для запоминания предшественников:

```
type SearchState a = ([a], DiGraph a, DiGraph a)
```

Дополнительно определим тип для результата поиска. Поиск либо безуспешен, либо успешен, и в таком случае возвращаются найденные предшественники. Предполагается, что граф предшествования позволит найти решение путем обратного прохода по нему.

```
data SearchResult a = Unsuccessful | Successful (DiGraph a)
```

В нашем алгоритме нужно выполнить две задачи:

1. произвести поиск и соответственно обновить состояние;

2 выполнить обратный проход по графу предшествования, чтобы получить найденный путь.

Схематически наша функция могла бы выглядеть так:

```
bfsSearch :: Eq a => DiGraph a -> a -> a -> Maybe [a]
bfsSearch graph start end
  | start == end = Just [start]
  | otherwise =
    case bfsSearch' ([start], graph, empty) of
      Successful preds -> Just (findSolution preds)
      Unsuccessful -> Nothing
```

Предполагается, что наша функция ищет в графе путь, соединяющий начальную вершину с конечной. Мы возвращаем значение типа `Maybe`, потому что поиск может завершиться неудачно, и в таком случае мы, конечно, вернем `Nothing`. Теперь сосредоточимся на реализации `bfs-Search'`. Эта функция будет работать с состоянием поиска и возвращать результат, содержащий признак успеха и предшественников. Текущий рубеж будет удален из графа функцией `deleteNodes`. При сборе вершин, соединенных с каждой вершиной текущего рубежа (дочерних вершин), мы должны профильтровать их, исходя из членства в уже обновленном графе с удаленными вершинами, поскольку не хотим добавлять в рубеж вершины, уже не входящие в граф. Чтобы добавить все новые вершины в качестве предшественников, мы напишем вспомогательную функцию, которая добавляет список кортежей, состоящих из вершин и их соседей с обращенными ребрами, т. е. будем добавлять в граф ребра, обозначающие, какая вершина какой предшествует при поиске. Обе эти функции (определенные локально в функции `bfsSearch`) показаны в листинге ниже.

Листинг 6.2 Вспомогательные функции для выполнения поиска в ширину

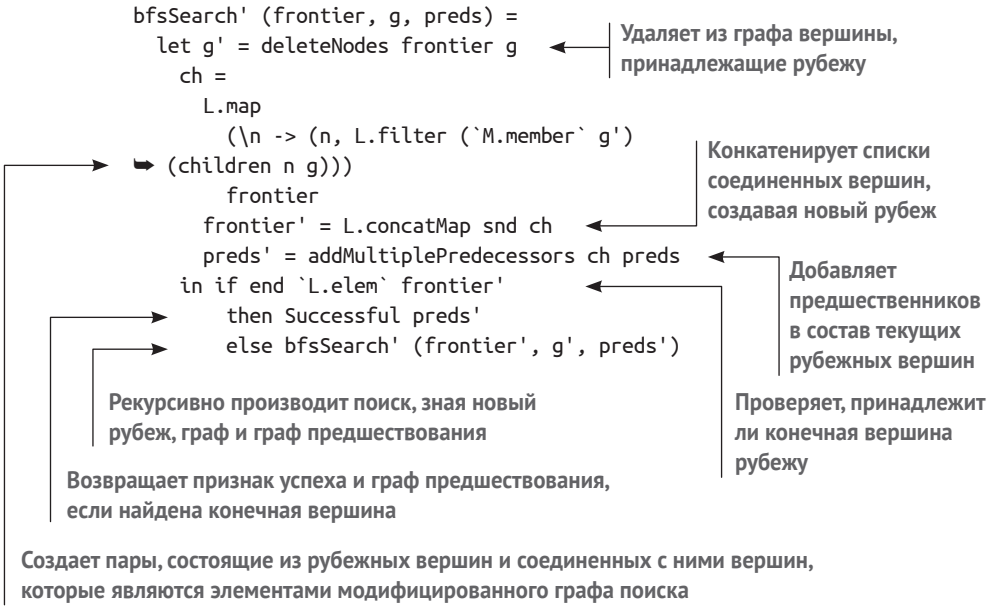
Вызывает вспомогательную функцию для каждой пары, состоящей из вершины и соединенных с ней вершин

```
addMultiplePredecessors :: Eq a => [(a, [a])] -> DiGraph a -> DiGraph a
addMultiplePredecessors [] g = g
addMultiplePredecessors ((n, ch) : xs) g =
  addMultiplePredecessors xs (go n ch g)
  where
    go n [] g = g
    go n (x : xs) g = go n xs (addEdge (x, n) g)

bfsSearch' :: Eq a => SearchState a -> SearchResult a
bfsSearch' ([], _, preds) = Unsuccessful
```

Возвращает признак неудачи, если рубеж пуст, поскольку поиск был безуспешен

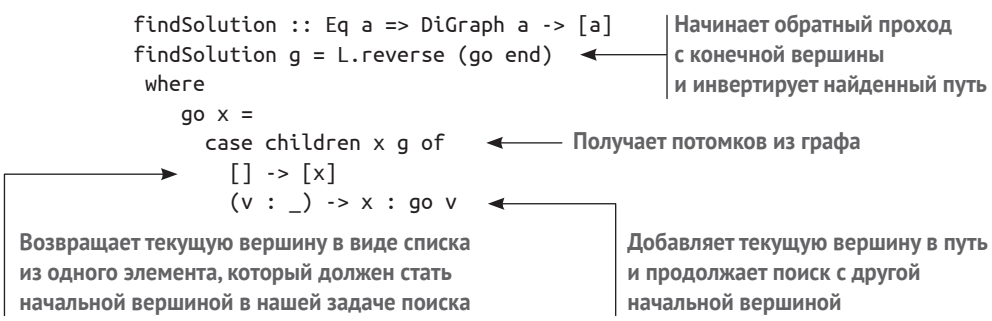
Рекурсивно добавляет каждую вершину в качестве предшественника исходной вершины



6.1.3 Нахождение решения с помощью обратного прохода

Последний кусочек этого пазла – алгоритм обратного прохода для поиска решения с помощью графа предшествования. Мы знаем, что все вершины этого графа имеют только одного предшественника, за исключением начальной вершины, у которой предшественников нет. Это позволяет рекурсивно пройти назад от конечной вершины к начальной. Поиск останавливается, когда предшественников не осталось. Код алгоритма показан в листинге ниже. Заметим, что вспомогательная функция, осуществляющая поиск, порождает решение в обратном порядке. Кроме того, алгоритм работает, только если решение существует, и выполняются сделанные нами предположения о графе предшествования!

Листинг 6.3 Алгоритм обратного прохода для поиска пути в графе предшествования



Теперь можно собрать все вместе! Добавив определения из листингов 6.2 и 6.3 в заготовку кода с помощью фразы `where`, мы приходим к окончательному определению алгоритма поиска. Однако код не компилируется, и выдается такая ошибка:

```
• Couldn't match expected type 'a1' with actual type 'a'
  'a1' is a rigid type variable bound by
    the type signature for:
      findSolution :: forall a1. Eq a1 => DiGraph a1 -> [a1]
    at .../ladder/src/Graph.hs:...
  'a' is a rigid type variable bound by
    the type signature for:
      bfsSearch :: forall a. Eq a => DiGraph a -> a -> a -> Maybe [a]
    at .../ladder/src/Graph.hs:...
```

По какой-то причине типы `bfsSearch` и `findSolution` не согласованы – но почему? Разве оба не полиморфны? У обоих одинаковые ограничения и имя типа, так что вроде бы они должны быть совместимы.

6.2 Область видимости переменной типа

Эта ошибка может привести в замешательство, а чтобы понять, в чем дело, нужно немного заняться теорией. Проблема в том, как Haskell обращается с переменными типа и их областями видимости.

6.2.1 Навешивание квантора всеобщности

Чтобы разобраться в этой ошибке, взглянем еще раз на выражение типа. Haskell скрывает от нас, что показанного ниже выражения типа не существует:

```
const :: a -> b -> a
```

Haskell видит его немного иначе:

```
const :: forall a b. a -> b -> a
```

Это называется *навешиванием квантора всеобщности* и по умолчанию неявно выполняется на самом внешнем уровне всех сигнатур типа, которые содержат свободные переменные типа. Ключевое слово `forall` привносит новые переменные типа в область видимости объявления функции. Для кода, внешнего по отношению к объявлению функции, это можно рассматривать как обещание: для всех типов, в которых вместо `a` и `b` что-то подставлено, объявление имеет место. Немного поэкспериментировав с функцией, легко убедиться, что это обещание выполняется:

```
ghci> const (1 :: Int) ("Hello" :: String)
1
ghci> const (True :: Bool) (3.1415 :: Float)
True
ghci> const (() :: ()) ((\x -> x) :: (a -> a))
()
```

Мы можем заменить `a` и `b` произвольными типами, и результат будем тем же. Для внешнего по отношению к объявлению функции кода это действительно обещание, но для определений внутри него оно является ограничением. И это понятно, потому что конкретные типы выбираются не объявлением функции, а вызываемой стороной! Внутри объявления функции типы фиксированы (в сообщениях об ошибках употребляется слово *rigid*).

```
f :: a -> a
f x = y
 where y = x
```

Тип в этом примере будет неявно изменен на `forall a. a -> a`; таким образом, переменная типа `a` вводится и фиксируется для объявления. Haskell выведет, что `x` имеет тип `a`, и поскольку `y = x`, то `y` также должна иметь тип `a`. Типы корректны. Но что, если мы добавим к `x` выражение типа, объявляющее ее как переменную типа `a`?

```
f :: a -> a
f x = y
 where y = (x :: a)
```

Возникнет та же самая ошибка. Но почему? После неявного навешивания квантора всеобщности код будет выглядеть так:

```
f :: forall a. a -> a
f x = y
 where y = (x :: forall a. a)
```

Выражение `forall a. a -> a` ограничивает переменную `a`, делая ее *произвольной, но фиксированной* в объявлении функции. И это распространяется также на объявления во фразе `where`! Однако `forall a. a` для `x` дает обещание, что `x` может быть произвольного типа. Это обещание дается оставшейся части определения функции и в конечном итоге приводит к несовместимости типов! `x` не может быть одновременно фиксированного и произвольного типа! При проверке этих свойств компилятор не принимает во внимание имена типов!

6.2.2 Расширения языка

Именно эта проблема и возникла в нашей функции поиска. По счастью, мы можем попросить Haskell применять *лексические области видимости* к переменным типа; это означает, что когда переменная типа вводится с помощью `forall`, ее можно повторно использовать в типах внутри объявления функции, и она будет обозначать один и тот же тип. Активировать это поведение можно с помощью *расширения языка*. Такие расширения позволяют изменять поведение компилятора Haskell глобально (с помощью флагов компилятора) или на уровне одного файла. Нас в данном случае интересует расширение языка `ScopedTypeVariables`. Чтобы активировать его, добавим следующую строку в начало нашего модуля:

```
{-# LANGUAGE ScopedTypeVariables #-}

module Graph (...) where
```

Теперь мы можем *явно* использовать слово `forall` в определениях типов, изменив его поведение: `forall` теперь вводит переменные типа с лексической областью видимости! Это позволяет написать показанную выше функцию, явно навесив квантор всеобщности на переменную типа в самой внешней сигнатуре типа, как показано ниже.

Листинг 6.4 Пример использования переменной типа с лексической областью видимости

```
f :: forall a. a -> a
f x = y
 where y = (x :: a)
```

Вводит переменную типа a, которая будет использоваться в объявлении функции

Ссылается на введенную переменную типа и может не включать повторно квантора всеобщности

Приятный побочный эффект заключается в том, что ограничения на тип *переносятся* на другие определения, так что мы должны применить ограничение только в самой внешней сигнатуре типа! Заметим также, что определения функций без явного использования `forall` ведут себя так же, как и раньше.

> **ВАЖНО** Использование `forall` перегружено, и его смысл зависит от активного расширения языка. Помимо `ScopedTypeVariables`, есть еще расширения `RankNTypes` и `ExistentialQuantification`, которые оказывают существенное влияние на работу системы типов. В общем случае использовать `forall` явно следует только тогда, когда необходимо! Впрочем, расширение `ScopedTypeVariables` относительно безопасно и во многих проектах активируется глобально.

6.2.3 Использование переменных типа с лексической областью видимости

После модификации типов для нашей функции поиска мы приходим к окончательному (и успешно компилируемому) определению. Полный исходный код приведен в листинге ниже. Обратите внимание на явное `forall` в самой внешней сигнатуре типа и на то, что ограничение типа `Eq a` встречается только в этой сигнатуре, потому что переменная типа `a` теперь доступна во всем объявлении функции.

Листинг 6.5 Нахождение кратчайших путей в ориентированных графах с одинаковой стоимостью ребер

Определяет тип для состояния поиска в ширину, включающий список рубежных вершин, граф, в котором производится поиск, и граф предшествования

Определяет тип результата поиска, включающий признак успеха или неудачи и граф предшествования, по которому находится решение

```
type SearchState a = ([a], DiGraph a, DiGraph a)

data SearchResult a = Unsuccessful |
➥ Successful (DiGraph a)

bfsSearch :: forall a. Eq a => DiGraph a
➥ -> a -> a -> Maybe [a]
bfsSearch graph start end
  | start == end = Just [start]
  | otherwise =
    case bfsSearch' ([start], graph, empty) of
          Successful preds -> Just (findSolution preds)
          Unsuccessful -> Nothing
  where
    findSolution :: DiGraph a -> [a]
    findSolution g = L.reverse (go end)
      where
        go x =
          case children x g of
            [] -> [x]
            (v : _) -> x : go v

addMultiplePredecessors :: [(a, [a])] -> DiGraph a -> DiGraph a
addMultiplePredecessors [] g = g
```

Возвращает тривиальное решение, когда начальная вершина совпадает с конечной

Возвращает найденное решение, осуществляя обратный проход по графу предшествования

Производит поиск кратчайшего пути из начальной вершины в конечную

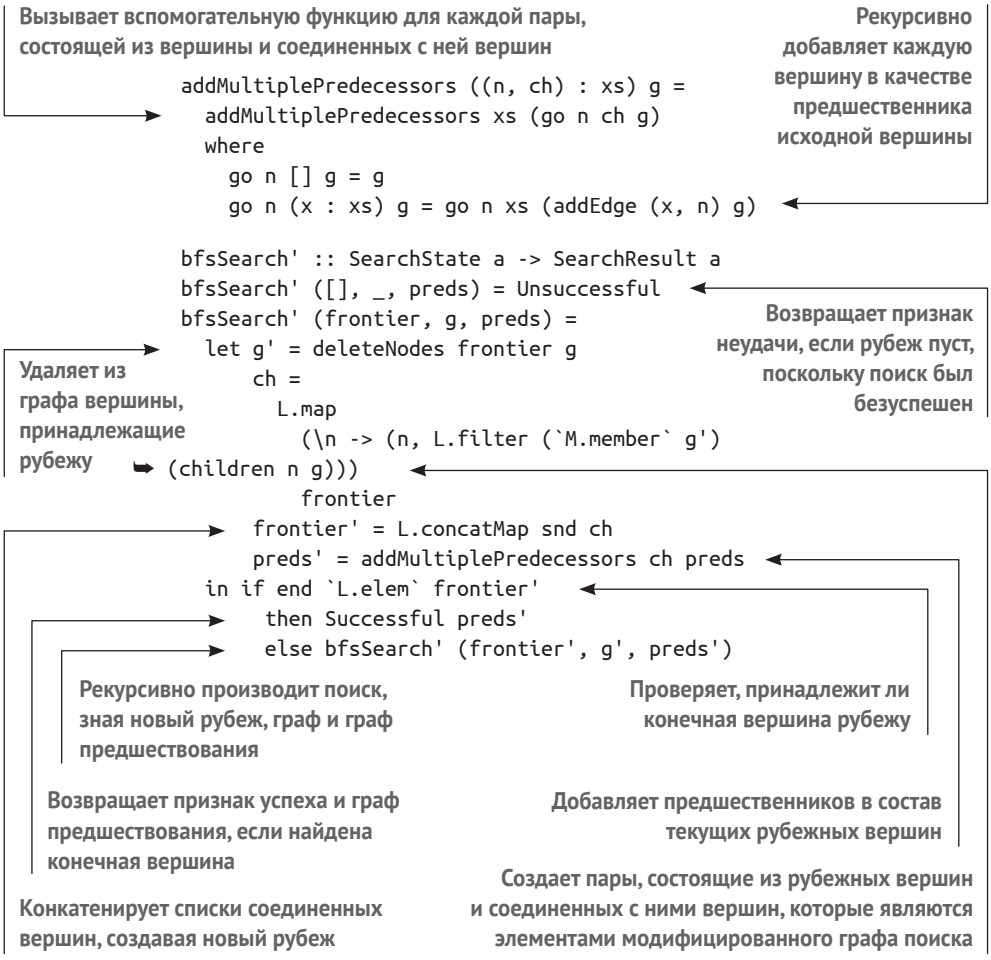
Явно вводит переменную типа a с лексической областью видимости

Получает потомков из графа

Добавляет текущую вершину в путь и продолжает поиск с другой начальной вершиной

Начинает обратный проход из конечной вершины и инвертирует найденный путь

Возвращает текущую вершину в виде списка из одного элемента, который должен стать начальной вершиной в нашей задаче поиска



Мы построили алгоритм поиска в ширину в ориентированном графе, который находит кратчайший путь, запоминая посещенные вершины, модифицированный граф и ранее найденных предшественников. Граф предшествования строится из посещенных вершин, а затем используется для нахождения собственно решения посредством обратного прохода.

Упражнение. Другие алгоритмы поиска

Для поиска кратчайшего пути в графе мы воспользовались *поиском в ширину*; однако существуют и другие алгоритмы поиска. Если нам нужно найти *любой* путь, а его длина не важна, то подойдет *поиск в глубину*. Кроме того, алгоритм двунаправленного поиска в ширину быстрее обыкновенного, потому что выполняет два поиска одновременно: из начальной вершины в конечную и из конечной в начальную. Когда оба

поиска встретятся, решение найдено. Реализуйте оба этих алгоритма. (Заметим, однако, что реализовать двунаправленный поиск в ширину нелегко.)

Теперь мы умеем строить граф цепочек и находить в нем кратчайший путь. А значит, есть все кусочки для построения программы!

6.3 Повышение производительности с помощью хеш-таблиц

Написать функцию для решения игры в цепочки слов довольно просто. Нужен только словарь и два слова: начальное и конечное. Для построения графа цепочек мы воспользуемся функцией `mkLadderGraph`, а для нахождения решения – нашим алгоритмом поиска. Код в том виде, в каком он хранится в модуле `Ladder`, показан в листинге ниже.

Листинг 6.6 Функция поиска оптимального решения игры в цепочки слов

```
ladderSolve :: Dictionary -> String -> String -> Maybe [String]
ladderSolve dict start end =
  let g = mkLadderGraph dict
   in G.bfsSearch g start end
```

Строит граф цепочек по словарю

Производит поиск в построенном графе

Модуль `Main` нашей программы можно сделать относительно простым. Как и в предыдущей главе, мы печатаем текст справки, если количество аргументов не соответствует ожиданиям. В противном случае строим словарь с помощью действия `readDictionary` и решаем задачу с помощью функции `ladderSolve`. После этого печатаем решение и его длину. Полный код модуля приведен в следующем листинге.

Листинг 6.7 Модуль `Main` для программы решения игры в цепочки слов

```
module Main (main) where

import Ladder
import System.Environment

printHelpText :: String -> IO ()
printHelpText msg = do
```

Импортирует все определения из модуля Ladder

Определяет действие для печати справки

```
  putStrLn (msg ++ "\n")
  progName <- getProgName
  putStrLn ("Usage: " ++ progName ++ " <filename> <start> <end>")

main :: IO ()
main = do
  args <- getArgs
  case args of
    [dictFile, start, end] -> do
      dict <- readDictionary dictFile
      case ladderSolve dict start end of
        Nothing -> putStrLn "No solution"
        Just sol -> do
          print sol
          putStrLn ("Length: " ++ show (length sol))
    _ -> printHelpText "Wrong number of arguments!"
```

Читает аргументы программы

Все хорошо, если задано ровно три аргумента

Читает словарь из указанного файла

Производит поиск кратчайшего пути и печатает результат

Действие `print` – это просто комбинация `show` и `putStrLn`, оно печатает значение на `stdout`. Теперь приложение можно протестировать!

Для этого в репозитории кода уже есть два готовых словаря: `small_dictionary.txt` на 200 слов и `large_dictionary.txt` на 58 110 слов. Находясь в каталоге проекта, мы можем вызвать программу следующим образом:

```
shell $ stack run -- path/to/small_dictionary.txt cat flower
["cat","oat","lot","volt","love","vowel","lower","flower"]
Length: 8
shell $ stack run -- path/to/small_dictionary.txt dog book
["dog","dot","lot","tool","look","book"]
Length: 6
```

Неплохо! А почему бы не протестировать ее со словарем побольше? Увы, мы не сможем увидеть функциональность программ во всем ее блеске, потому что, по крайней мере на моей машине, она не выводит никакого результата. Она работает слишком долго. Нужно улучшать положение.

6.3.1 Анализ производительности с помощью профилирования

Прежде чем решать, что именно улучшать, нужно проанализировать программу и понять, какая операция занимает так много времени. Для этого мы подвергнем программу профилированию. Но предварительно нужно откомпилировать ее командой `stack` с флагом `--profile`. При этом в GHC активируется несколько параметров, позволяющих профилировать приложение во время работы. После этого нужно включить в среде исполнения базовое профилирование

времени и памяти с помощью флагов `+RTS -p -RTS`. `+RTS` и `-RTS` обозначают начало и конец аргументов, предназначенных среде исполнения Haskell, а не самому приложению. Полная командная строка выглядит так:

```
shell $ stack run --profile -- \
            path/to/small_dictionary.txt dog book +RTS -p -RTS
```

После завершения программы будет создан файл, который в нашем случае называется `ladder-exe.prof` и содержит информацию о профилировании:

```
Mon Jul 25 16:22 2022 Time and Allocation Profiling Report (Final)

          ladder-exe +RTS -N -p -RTS path/to/small_dictionary.txt dog book

        total time =         0.03 secs   (100 ticks @ 1000 us, 8 processors)
        total alloc =  46,192,080 bytes  (excludes profiling overheads)

COST CENTRE                  MODULE         SRC
       ➥ %time %alloc

lookup.lookup'               Data.AssocMap  src/Data/AssocMap.hs:(54,5)-(5
➥ 7,34) 54.0 0.1
computeCandidates.uniques    Ladder         src/Ladder.hs:19:7-50
➥ 21.0 26.9
member.member'               Data.AssocMap  src/Data/AssocMap.hs:(24,5)-(2
➥ 7,32) 14.0 0.0
alter.alter'                 Data.AssocMap  src/Data/AssocMap.hs:(33,5)-(4
➥ 3,40) 5.0 44.7
lookup                       PermutationMap src/PermutationMap.hs:31:1-34
➥ 3.0 12.3
readDictionary               Ladder         src/Ladder.hs:(10,1)-(14,22)
➥ 2.0 0.2
computeCandidates.perms      Ladder         src/Ladder.hs:20:7-69
➥ 0.0 1.7
computeCandidates.modified   Ladder         src/Ladder.hs:(25,5)-(26,58)
➥ 0.0 7.1
computeCandidates.canditates Ladder         src/Ladder.hs:18:7-57
➥ 0.0 2.5
```

Это краткий перечень *центров затрат* нашей программы, которые состоят в основном из функций. Он говорит, сколько времени проведено в каждой функции и сколько было выделено памяти. Отсюда видно, что очень много времени тратится в функции `lookup` из модуля `AssocMap`. Больше половины времени – на одну эту функцию! Оно и понятно, потому что мы постоянно ищем значения в графе, который представляет собой просто структуру `AssocMap`. Так что если мы хотим ускорить работу программы, то должны оптимизировать эту структуру!

Но вот вопрос: что не так с этой функцией? Это простой поиск в ассоциативном списке, образующем нашу карту. В худшем случае нам придется обойти весь список при каждом поиске! Понятно, что чем больше словарь, тем больше граф, а значит, мы будем тратить больше времени на поиск в нем значений. Печально, но такова реальность – ассоциативные списки по природе своей работают медленно. Чтобы хоть как-то ускориться, нам нужно полностью заменить их.

6.3.2 Добавление зависимостей в проект

Отличным кандидатом будет *хеш-таблица*, известная быстрым доступом. Но на этот раз мы не будем программировать собственную хеш-таблицу, а воспользуемся готовой! Для этого нужно добавить в проект зависимости `unordered-containers` и `hashable`. Чтобы это сделать, отредактируем свой файл `package.yml`. Интересующая нас часть должна выглядеть так:

```
dependencies:
- base >= 4.7 && < 5
- unordered-containers
- hashable
```

Теперь `stack` автоматически скачает и соберет зависимости вместе с нашим проектом. После чего мы сможем заменить наш модуль `Data.AssocMap` модулем `Data.HashMap.Lazy`. Кроме того, в модулях `Graph` и `PermutationMap` нужно использовать в наших типах `M.HashMap` вместо `M.AssocMap`. Чтобы нечто могло выступать в роли ключа в `HashMap`, оно должно иметь экземпляр класса типов `Hashable`. Поэтому мы должны изменить сигнатуру типа в модуле `Graph`, включив `Hashable a` в ограничения типа. Этот класс предоставляет метод хеширования, который используется в `HashMap`, и может быть импортирован из модуля `Data.Hashable`.

> **ПРИМЕЧАНИЕ** Вовсе не совпадение, что замена `AssocMap` на `HashMap` прошла так гладко – ведь мы специально реализовали те же самые функции, что имеются в модуле `HashMap`. Если вы поняли, как работают функции из `AssocMap`, то уже знаете, как работает модуль `HashMap`!

Снова откомпилировав и выполнив программу, мы сможем работать даже с большим словарем! Еще раз взглянем на центры затрат, запустив программу в режиме профилирования.

```
computeCandidates.uniques   Ladder              src/Ladder.hs:19:7-50
➥ 67.4 34.7
readDictionary              Ladder              src/Ladder.hs:(10,1)-(
➥ 14,22) 4.7 0.2
```

```
liftHashWithSalt.step       Data.Hashable.Class    src/Data/Hashable/Clas
➥ s.hs:656:9-46 4.7 7.2
liftHashWithSalt            Data.Hashable.Class    src/Data/Hashable/Clas
➥ s.hs:(653,5)-(656,46) 4.7 0.0
lookup#                     Data.HashMap.Internal  Data/HashMap/Internal.
➥ hs:597:1-82 2.3 1.3
insert'.go                  Data.HashMap.Internal  Data/HashMap/Internal.
➥ hs:(759,5)-(788,76) 2.3 1.1
```

Поиск стал таким быстрым, что основным пожирателем времени теперь кажется вычисление уникальных ключей поиска в карте перестановок! По счастью, от этого легко избавиться, потому что мы и добавили-то эту функцию, только чтобы минимизировать количество операций поиска в карте перестановок. Ну, а раз реализация хеш-таблицы настолько быстрая, то и необходимость в ней отпала. Еще одно улучшение производительности! После еще одного профилирования приложения, на этот раз с большим словарем, мы получаем удивительный результат:

```
readDictionary              Ladder                 src/Ladder.hs:(10,1)-(14
➥ ,22) 95.7 5.5
readDictionary.words        Ladder                 src/Ladder.hs:13:7-60
➥ 1.3 6.0
alter                       PermutationMap         src/PermutationMap.hs:22
➥ :1-36 0.4 18.5
readDictionary.lines        Ladder                 src/Ladder.hs:12:7-39
➥ 0.4 14.5
insert'.go                  Data.HashMap.Internal  Data/HashMap/Internal.hs
➥ :(759,5)-(788,76) 0.3 5.8
```

Теперь мы тратим больше всего времени на чтение файла! Это отличная новость, потому что она означает, что алгоритм более-менее оптимизирован. Однако можно улучшить даже чтение файла. До сих пор мы радостно использовали убийцу производительности, даже не зная об этом. И зовут преступника `String`. Проблема кроется в конструировании строк, которые являются не чем иным, как списком значений типа `Char`. У этого списка есть две основные беды:

- он расположен в куче, поэтому доступ к памяти довольно медленный;
- он реализован как связный список, что плохо с точки зрения локальности кеша.

Когда производительность выходит на первое место, типа `String` следует избегать. Подходящие замены – тип `Text` из пакета `text` или тип `ByteString` из пакета `bytestring`. Оба предлагают более производительное, упакованное представление строк. Но у них есть недостаток – невозможно использовать обычные функции работы со списками, что делает эти типы несколько менее переносимыми. Мы не будем здесь вдаваться в детали замены типа `String` чем-то более

быстрым, потому что выигрыш окажется несущественным. Но любопытные читатели могут найти в репозитории оптимизированную версию этого проекта!

> **ВАЖНО** При проектировании программ, для которых важна производительность, замена типов должна быть *последним средством*. Правильный выбор алгоритма и структур данных гораздо важнее технических деталей.

6.3.3 Ленивое вычисление

Обсуждение производительности мы завершим более пристальным взглядом на процесс вычисления в Haskell. В отличие от других языков, вычисления в Haskell являются *ленивыми*. Это значит, что выражения вычисляются не сразу, а только тогда, когда это *необходимо*. Рассмотрим пример:

```
ghci> :{
ghci| const :: a -> b -> a
ghci| const x _ = x
ghci| :}
ghci> x = (1000 :: Int)^(100000000 :: Int)
ghci> y = 0 :: Int
ghci> const x y
0
```

Функция `const` имеет два аргумента, и второй из них отбрасывается. Что происходит в нашем вычислении? Выражение `const 0 1000^100000000` сводится к `0`, потому что `const x _` сводится к `x`. Второй аргумент – гигантское число, на вычисление которого ушло бы много времени, попросту отбрасывается и, стало быть, не вычисляется. В приложении B приведены дополнительные сведения на эту тему.

В нашем алгоритме ленивым вычислениям тоже нашлось место. Мы не строим граф целиком перед поиском решения, вместо этого он строится в процессе поиска! Вычисляются лишь необходимые элементы графа. Однако это работает, только если все наши структуры данных *ленивые*. Списки и определенные нами типы являются ленивыми по умолчанию. Воспользовавшись хеш-таблицами для повышения производительности, мы сознательно импортировали ленивую версию `HashMap`, которая не вычисляет значения принудительно.

> **ПРИМЕЧАНИЕ** Может показаться, что *ленивое вычисление* в общем случае предпочтительнее своей противоположности – *строгого вычисления*, но это не так. Некоторые алгоритмы, например алгоритм поиска в этой главе, выигрывают от ленивых вычислений, тогда как в других производительность разительно падает. Кроме того, откладывание вычислений

может вызвать неожиданный расход памяти, поэтому некоторые разработчики полностью запрещают его в своих проектах с помощью таких расширений языка, как `StrictData`.

Подведем итоги сделанному. Мы разработали искусственный интеллект, который по заданному словарю умеет находить кратчайшее решение модифицированной игры в цепочки слов. Для этого мы реализовали алгоритм создания графа, представляющего все решения игры, и посредством поиска в этом графе нашли решение. В процессе работы мы написали свою версию карты на основе ассоциативных списков и поместили всю функциональность этого типа в отдельный модуль, допускающий повторное использование. На базе этой реализации мы создали другой тип карты, который используется для быстрого нахождения допустимых перестановок слов. Граф моделировался как карта смежности, а для карты использовалась написанная нами реализация. Для поиска кратчайшего пути мы применили поиск в ширину. После тестирования и профилирования программы мы улучшили ее производительность, заменив карту на основе ассоциативных списков хеш-таблицей.

Теперь мы готовы принять участие в чемпионате по игре в цепочки слов! … Если, конечно, он существует.

Резюме

- На свободные переменные типы неявно навешивается квантор всеобщности.
- Переменные с навешенным квантором всеобщности – это обещание, данное части программы вне объявления о том, что любую переменную, удовлетворяющую определенным ограничениям, можно будет использовать с данным объявлением.
- Вызываемый функцией код решает, какие конкретные типы будут использованы в выражении, содержащем переменные с навешенными кванторами всеобщности.
- Чтобы изменить поведение языка по умолчанию, можно использовать расширения, которые активируются программой `{-# LANGUAGE … #-}`.
- Чтобы включить в проекте профилирование, нужно вызвать команду `stack` с флагом `--profile`.
- Ограничители `+RTS` и `-RTS` позволяют передать параметры среде выполнения Haskell при запуске приложения.
- Для добавления внешних зависимостей в stack-проекты следует включить имена пакетов в секцию `dependencies` файла `package.yml`.
- Ленивое вычисление означает, что выражения вычисляются только в момент, когда возникает необходимость в их результате.

7 Работа с CSV-файлами

Краткое содержание этой главы:

- синтаксис записей в Haskell и его применение;
- использование *умных конструкторов* для безопасного создания структур данных, обладающих некоторым свойством;
- создание экземпляров классов типов;
- определение собственных классов типов с реализациями по умолчанию.

В предыдущей главе мы занимались задачей поиска и написали искусственный интеллект для детской игры. В этой же главе мы намерены заняться серьезным бизнесом, а что может быть серьезнее в бизнесе, чем электронные таблицы? Не волнуйтесь, мы не будем непосредственно работать ни с какими данными. Будучи программистами, мы, очевидно, хотим автоматизировать такие задачи, а самый простой способ работы с данными, хранящимися в электронной таблице, – файлы в формате значений через запятую (CSV).

В этой главе мы поговорим о том, как разбирать такие файлы, извлекать из них данные и эффективно работать со структурированными данными. Мы узнаем, как обобщить дописывание и вырезание данных, а также как классы типов помогают писать программы.

Мы начнем эту главу с введения в CSV-файлы и их моделирование с помощью синтаксиса записей в Haskell. Узнаем, как тип `Either` позволяет кодировать ошибки. Мы рассмотрим оператор «доллар» и некоторые расширения языка для упрощения синтаксиса. Далее создадим собственные классы типов, чтобы обобщить рассмотренное применение наших типов данных. Попутно мы узнаем о классах типов `Semigroup` и `Monoid` – что они представляют и для чего используются.

7.1 Моделирование CSV-данных

В этой главе будем рассматривать работу с CSV-файлами из командной строки. Хотя для человека их разбор не представляет особых трудностей, по мере увеличения размера работать с ними становится очень неудобно. Поэтому для манипулирования данными нам зачастую приходится прибегать к какому-нибудь инструменту с графическим интерфейсом. Но можно сделать лучше! У нашей программы будет много возможностей для отображения и преобразования табличных данных, в том числе:

- чтение CSV-файлов и их печать в виде ASCII-таблицы с поддержкой файлов с заголовком и без;
- объединение двух CSV-файлов;
- фильтрация строк таблицы, удовлетворяющих условию поиска;
- вырезание из таблицы заданного диапазона столбцов;
- подсчет непустых строк в указанном столбце и печать результата в виде таблицы;
- в качестве альтернативы запись результата преобразования в CSV-файл.

Для примера посмотрим, как выглядел бы результат нашей программы (которую мы назовем `csview`) после чтения файла, поиска строки и вырезания некоторых столбцов:

```
shell $ head -n 4 cities.csv
"LatD","LatM","LatS","NS","LonD","LonM","LonS","EW","City","State"
   41,    5,   59, "N",      80,   39,   0, "W", "Youngstown", OH
   42,   52,   48, "N",      97,   23,  23, "W", "Yankton", SD
   46,   35,   59, "N",     120,   30,  36, "W", "Yakima", WA
shell $ csview --in=cities.csv --with-header --search=Ya --slice="8,10"
------------------
| City    | State |
+---------+-------+
| Yankton | SD    |
| Yakima  | WA    |
------------------
```

Прежде чем приступать к написанию инструмента, нужно разобраться в этом файловом формате.

CSV-файлы – это простые текстовые файлы, строки которых называются *записями данных*. Записи содержат значения и разделены запятыми, так что CSV-файл состоит из строк, в которых значения перечислены через запятую. Кроме того, первая строка может (хотя это не обязательно) содержать заголовок с названиями столбцов. Довольно просто, не правда ли?

Увы, хотя формат документирован (предложенная спецификация содержится в RFC 4180), складывается впечатление, что каждая реализация вводит свои правила построения CSV-файлов. Перечислим несколько типичных ограничений, действующих в отношении CSV-файлов. Они должны:

- содержать простой текст в общепринятой кодировке (UTF-8, ASCII и т. д.);
- содержать в каждой строке одну запись;
- использовать один на весь файл разделитель полей в записи (и это необязательно должна быть запятая);
- содержать одинаковое число полей в каждой записи;
- могут содержать строку заголовка;
- поля могут заключаться в двойные кавычки.

Поэтому программа для просмотра и анализа данных в таких файлах должна поддерживать ряд параметров, описывающих различные способы разбора содержимого. Давайте обсудим, что должно делать наше приложение. Мы хотим, чтобы оно читало CSV-файлы и печатало их в форме таблицы. Кроме того, мы хотим иметь возможность искать значения и вычислять статистику данных. Порядок работы будет выглядеть примерно так:

1. Прочитать аргументы и вывести из них параметры разбора и запрашиваемую функциональность.
2. Разобрать CSV-файл, заполнив структуру данных.
3. Выполнить поиск и (или) вычислить статистику данных.
4. Записать данные обратно в файл.
5. Факультативно напечатать красиво отформатированную информацию.

Прежде чем заняться чтением аргументов командной строки, мы должны определить, как представлять CSV-файлы в программе, и обсудить их разбор!

7.1.1 Синтаксис записей

Как всегда, начнем с создания нового проекта командой `stack new csview`. Мы уже сказали, что CSV-файлы содержат текст. В этом проекте мы хотим отойти от работы с типом `String` и использовать бо-

лее распространенный способ работы с текстом в Haskell: тип `Text`, который представляет собой более производительную, упакованную версию строки литер. Для работы с этим типом нужно сначала включить пакет `text` в проект. Для этого отредактируем секцию `dependencies` в файле `package.yaml`. Попутно можем изменить имя исполняемого файла и удалить суффикс `-exe`. Впоследствии мы захотим установить это приложение на свою машину, поэтому о хорошем имени стоит позаботиться заранее!

```
dependencies:
- base >= 4.7 && < 5
- text
...
executables:
  csview:
    main:         Main.hs
    ...
```

Тип данных `Text` можно импортировать из модуля `Data.Text`, который содержит много функций для работы с текстом, поскольку использовать функции из модуля `Data.List` мы больше не можем. Для преобразования данных из `String` в `Text` и обратно служат функции `pack` и `unpack`:

```
ghci> import Data.Text
ghci> myString = "Hello Text"
ghci> :t pack myString
pack myString :: Text
ghci> :t unpack (pack myString)
unpack (pack myString) :: String
```

Модуль содержит многие функции, которые заменяют то, что мы обычно получаем из `Data.List`, например `null` или `length`. Чтобы сэкономить место, отсылаем читателей к документации по модулю (https://mng.bz/1a8n).

Теперь мы готовы подумать о моделировании CSV-файлов в своей программе. Поскольку CSV-файлы, вообще говоря, не типизированы, содержащиеся в них значения можно рассматривать как текст вне зависимости от того, что в них находится. Однако позже мы будем предполагать, что CSV-файлы содержат числовую или текстовую информацию, чтобы можно было строить такие алгоритмы, как поиск строки и суммирование чисел. Столбцы, равно как и необязательный заголовок файла, можно представлять простыми списками. Таким образом, можно было бы создать тип вроде:

```
type Column = [Text]

type Csv = (Maybe [Text], [Column])
```

В этом типе первый элемент кортежа – факультативный (представлен типом `Maybe`) заголовок, второй представляет столбцы файла, а каждый столбец является списком полей. На этой основе мы можем определить функции для получения значений:

```
header :: Csv -> Maybe [Text]
header = fst

columns :: Csv -> [Column]
columns = snd
```

Метод `snd` мы встречали в главе 6. Напомним, что `fst` извлекает первый элемент кортежа, а `snd` – второй. Определения выглядят неплохо, но можно сделать лучше. Налицо несколько проблем.

- Тип является синонимом типа, поэтому мы не можем скрыть его строение с помощью списка экспорта модуля.
- При каждом расширении типа нам придется изменять функции, обращающиеся к его элементам, или добавлять новые.
- Без комментариев непонятно, что именно представляют поля кортежа.

К счастью, с этими проблемами можно справиться, воспользовавшись *синтаксисом записей* в Haskell. Он позволяет давать имена полям конструкторов данных. Посмотрим, как это выглядит в действии, для чего определим тип как положено и попутно выведем экземпляр `Show`. Ниже приведен код типа и нового модуля `Csv.Types`.

Листинг 7.1 Тип данных для содержимого CSV-файла с использованием синтаксиса записей

```
module Csv.Types where

import qualified Data.Text as T

type Column = [T.Text]

data Csv = Csv
  { csvHeader :: Maybe [T.Text],
    csvColumns :: [Column]
  }
  deriving (Show)
```

Квалифицированно импортирует Data.Text и задает псевдоним T

Определяет тип Csv с единственным конструктором

Определяет два поля для конструктора Csv

Выводит экземпляр Show для нового типа

Мы импортируем `Data.Text` квалифицированно, потому что в нем много функций, имена которых конфликтуют с `Prelude`. Как видим, синтаксис записи позволяет идентифицировать поля конструкторов данных по именам! Есть и дополнительный эффект: имена полей играют роль функций, извлекающих соответствующее поле из типа данных. Они называются *селекторами полей*:

```
ghci> :{
ghci| data Record = Record
ghci|   { field1 :: Int,
ghci|     field2 :: String
ghci|   }
ghci| :}
ghci> :t field1
field1 :: Record -> Int
ghci> :t field2
field2 :: Record -> String
ghci> field1 (Record {field1 = 1, field2 = ""})
1
```

Это означает, что если бы мы расширили этот тип, добавив еще одно поле, то была бы автоматически добавлена функция доступа к этому полю! Кроме того, теперь наш тип снабжен минимальной документацией, поскольку имена говорят о назначении полей. Наконец, поскольку у нас теперь имеется конструктор типа, мы можем исключить его из списка экспорта модуля (как поступали в главе 5 для типа `AssocMap`), чтобы предотвратить конструирование значений с недопустимыми данными.

> **ПРИМЕЧАНИЕ** Поскольку имена полей порождают функции, имена таких функций могут конфликтовать. Если в записях имеются поля с одинаковыми именами, то импортировать их в модули нужно очень осторожно. Принято добавлять к имени поля префикс с какой-нибудь идентифицирующей аббревиатурой (например, акронимом имени типа, скажем, `csvHeader` или `csv_header`, а не просто `header`).

При конструировании значений с помощью синтаксиса записей мы можем явно именовать поля, как в предыдущем примере, или просто записывать значения как поля конструктора в том же порядке, в каком поля определены:

```
ghci> data Record = Record {field1 :: Int, field2 :: String} deriving Show
ghci> Record {field1 = 100, field2 = "Hello"}
Record {field1 = 100, field2 = "Hello"}
ghci> Record 100 "Hello"
Record {field1 = 100, field2 = "Hello"}
```

Если заданы не все поля, то мы увидим такое же поведение, как для конструкторов алгебраических типов данных, с которыми мы уже знакомы. Они частично применяются, и их вычисление порождает функцию, которая вычисляет тип записи, после того как будут заданы все недостающие поля:

```
ghci> :t Record
Record :: Int -> String -> Record
```

```
ghci> :t Record 100
Record 100 :: String -> Record
```

Отметим, что конструирование типа данных с заданием как неименованных, так и именованных полей не поддерживается:

```
ghci> Record 100 {field2 = "Hello"}

<interactive>:25:8: error:
    • Couldn't match expected type 'Int' with actual type 'Record'
    • In the first argument of 'Record', namely
        '100 {field2 = "Hello"}'
      In the expression: Record 100 {field2 = "Hello"}
      In an equation for 'it': it = Record 100 {field2 = "Hello"}
```

Записи также можно модифицировать, явно указав имена полей и значение каждого из них:

```
ghci> :{
ghci| f :: Record -> Record
ghci| f rec = rec {field1 = 0}
ghci| :}
ghci> f (Record 100 "Hello")
Record {field1 = 0, field2 = "Hello"}
```

Синтаксис записей можно использовать в сочетании с несколькими конструкторами, используя разные поля для каждого из них, но этой возможности мы здесь касаться не будем. Однако скажем, что это работает аналогично записям с одним конструктором, только добавляется проблема частичных селекторов полей, которые мы рассмотрим в главе 10 и которых в общем случае лучше избегать.

7.1.2 Кодирование ошибок типом Either

В начале этой главы мы говорили, что впоследствии захотим искать в CSV-файлах текст или вычислять суммы. Для этого нужно уметь различать значения, представляющие текст и числа. Это можно сделать, заведя новый тип для кодирования полей данных, как показано в листинге ниже.

Листинг 7.2 Тип данных для полей в CSV-таблице

```
data DataField
  = IntValue Int
  | TextValue T.Text
  | NullValue
  deriving (Eq, Show)
```

Определяет конструктор значений типа Int

Определяет конструктор значений типа Text

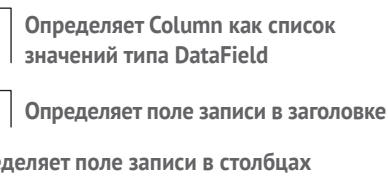
Определяет конструктор пустых значений

Таблица типа `Csv` заполняется не произвольными данными, а значениями типа `DataField`. Финальная версия типа `Csv` показана ниже.

Листинг 7.3 Тип данных для CSV-таблиц

```
type Column = [DataField]

data Csv = Csv
  { csvHeader :: Maybe [T.Text],
    csvColumns :: [Column]
  }
  deriving (Show)
```



Это позволяет нам включать в таблицы текстовые и числовые значения, даже смешивая их в одном столбце.

Далее мы поговорим о конструкторах типа `Csv` и о том, как гарантировать, что с их помощью конструируются только осмысленные данные.

7.2 Умные конструкторы

Иногда мы хотим задавать для типов свойства, которые не могут быть проверены самой системой типов. Например, число полей в заголовке должно быть равно числу столбцов, а во всех столбцах должно быть одинаковое число элементов. В главе 5 при работе с типом `AssocMap` мы явно запретили конструировать его значения напрямую, разрешив использовать для этой цели только определенные экспортируемые нами функции. Но в случае CSV мы хотели бы разрешить конструирование, но при этом включить проверку ошибок. Тогда нам не придется проверять инварианты типа в каждой функции, которая с ним работает. Это подводит нас к теме *умных конструкторов*.

7.2.1 Проверка свойства на этапе конструирования

Гарантировать свойства типа `Csv` мы можем, разрешив конструировать его значения только специальной функцией, которая проверяет корректность аргументов. Обычно эта функция аварийно завершает программу в случае ошибки, и мы тоже предоставим такую возможность, но дополнительно хотим иметь *безопасную* версию. Как должна выглядеть эта функция?

```
mkCsv :: Maybe [T.Text] -> [Column] -> ???
```

`Maybe Csv` кажется разумным выбором, но было бы невежливо просто молча завершиться с ошибкой, не дав пользователю возможности узнать, что не так. Мы хотели бы возвращать либо конкретную ошибку, либо значение типа `Csv`. Такой тип мог бы иметь вид

```
data ErrorOrCsv a = Error String | Value Csv
```

По счастью, уже существует тип с нужной нам функциональностью – `Either`!

```
data Either a b = Left a | Right b
```

Его можно использовать по-разному, но чаще всего – для кодирования возможной ошибки. В таком случае `Right` кодирует корректное значение, а `Left` – ошибку. Если нам нужно получить либо сообщение об ошибке, либо значение, то следует написать:

```
type ErrorOrValue a = Either String a
```

Именно этим типом мы и воспользуемся в нашей функции! Легко проверить, совпадает ли длина заголовка с числом столбцов во входных данных. Кроме того, нужно проверить, что все столбцы имеют одинаковую длину. Для этого нам пригодится функция `nubBy` из модуля `Data.List`. Она работает почти так же, как `nub`, но получает в качестве предиката бинарную функцию, которая проверяет равенство двух значений в списке. Если длина результата этой функции меньше или равна 1, то мы знаем, что число элементов во всех столбцах одинаково. Это показано в листинге ниже.

Листинг 7.4 Функция безопасного конструирования значения типа Csv

Возвращает сообщение об ошибке, если длина заголовка некорректна

Возвращает сообщение об ошибке, если длины столбцов некорректны

```
mkCsv :: Maybe [T.Text] -> [Column] -> Either String Csv
mkCsv mHeader columns
  | not headerSizeCorrect =
      Left "Size of header row does not fit number of columns"
  | not columnSizesCorrect =
      Left "The columns do not have equal sizes"
  | otherwise = Right Csv
➥ {csvHeader=mHeader, csvColumns=columns}
  where
    headerSizeCorrect =
      M.maybe True (\h -> L.length h == L.length columns) mHeader
    columnSizesCorrect =
      L.length (L.nubBy (\x y -> length x == length y) columns) <= 1
```

Проверяет, что размер заголовка равен количеству столбцов

Проверяет, что все столбцы имеют одинаковый размер

Возвращает значение CSV, передавая поля в качестве аргументов

Здесь мы предполагаем, что `M` – квалифицированный импорт `Data.Maybe`. Далее в этой книге принято несколько соглашений о квалифицированном импорте: `T` заменяет `Data.Text`, `L` – `Data.List`, `M` – `Data.Maybe`, `E` – `Data.Either`.

7.2.2 Предоставление небезопасной альтернативы

Написанную выше функцию можно считать «безопасной», потому что она не завершает программу, когда некоторые (или все) аргументы некорректны. Как и в случае `Maybe`, мы предусмотрели возможность ошибки в типе. Однако эта функция требует, чтобы при каждом ее использовании производилось сопоставление с образцом, даже если мы уже проверили, что все инварианты выполнены. Чтобы упростить себе жизнь, мы напишем «небезопасную» версию этой функции, которая аварийно завершает программу, когда аргументы не такие, какими должны быть.

> **ПРИМЕЧАНИЕ** В некоторых кодовых базах к имени «безопасной» функции прибавляется специальный префикс, а в имени «небезопасной» функции его нет. Мы будем добавлять к имени небезопасных функций префикс `unsafe`, так что, встретив имя `unsafeName`, вы должны понимать, что использование функции чревато серьезной опасностью.

Для этого будем использовать функцию `error`, которая возбуждает исключение:

```
ghci> :t error
error :: GHC.Stack.Types.HasCallStack => [Char] -> a
ghci> error "Oh no!"
*** Exception: Oh no!
CallStack (from HasCallStack):
  error, called at <interactive>:... in interactive:Ghci1
```

Тип `Maybe` находится в отдельном модуле `Data.Maybe`, который экспортирует функцию `maybe` для упрощения работы, и точно так же обстоит дело с типом `Either`:

```
ghci> import Data.Either
ghci> :t either
either :: (a -> c) -> (b -> c) -> Either a b -> c
```

Функция `either` принимает две функции – для случаев `Left` и `Right`, – чтобы в конечном итоге породить значение некоторого общего типа. В нашем случае мы можем написать конструктор `unsafeMkCsv`, воспользовавшись функцией `error` для случая `Left` и функцией `id` для случая `Right`, которая просто возвращает свой аргумент. Это показано в листинге ниже.

Листинг 7.5 Небезопасная функция, которая аварийно завершает программу, если переданы недопустимые аргументы для значения CSV

```
unsafeMkCsv :: Maybe [T.Text] -> [Column] -> Csv
unsafeMkCsv header columns =
  E.either error id (mkCsv header columns)
```

Либо возбуждает исключение, получив ошибку от безопасной mkCsv, либо возвращает результат

Эта функция либо возвращает корректный результат типа `Csv`, либо завершает программу.

> **ПРЕДУПРЕЖДЕНИЕ** Функцию `error` следует использовать только в случае серьезной ошибки в программе, после которой восстановление невозможно. Не надо изобильно рассыпать ее по коду, потому что она аварийно завершает программу.

Теперь можно протестировать умные конструкторы:

```
ghci> :set -XOverloadedStrings
ghci> int x = IntValue x
ghci> mkCsv (Just ["First", "Second"]) [[int 1, int 2], [int 3, int 4]]
Right (Csv {csvHeader = Just ["First","Second"], csvColumns =
➥ [[IntValue 1,IntValue 2],[IntValue 3,IntValue 4]]})
ghci> mkCsv (Just ["First", "Second"]) [[int 1, int 2], [int 3]]
Left "The columns do not have equal sizes"
ghci> mkCsv (Just ["First"]) [[int 1, int 2], [int 3, int 4]]
Left "Size of header row does not fit number of columns"
ghci> unsafeMkCsv Nothing [[int 1, int 2], [int 3]]
*** Exception: The columns do not have equal sizes
```

Но какой конструктор использовать? `unsafeMkCsv` может возбуждать исключение, которое (если только не перехвачено) завершит программу. Если мы предпочитаем безопасность, то следует использовать `mkCsv`. Конструктор `unsafeMkCsv` полезен только тогда, когда мы заведомо знаем, что ничего плохого не произойдет, потому что проверили где-то в другом месте.

Из примера выше также видно, что мы можем как обычно использовать значения типа `String` для объявления значений типа `Text`. Почему это возможно, мы скоро узнаем. Но сначала поговорим об одном виде синтаксического сахара.

7.2.3 Оператор доллар

Функция `unsafeMkCsv` демонстрирует часто встречающуюся практику написания кода. Ей передается несколько аргументов, причем последний сам является выражением:

```
E.either error id (mkCsv header columns)
```

Эта конструкция может оказаться громоздкой, если последнее выражение длинное. К счастью, можно воспользоваться оператором, которого мы еще не видели: `$`. Этому оператору недостает сложности, но это с лихвой компенсируется элегантностью и простотой. Ниже показано его полное определение.

Листинг 7.6 Оператор доллар в Haskell

```
($) :: (a -> b) -> a -> b
($) f x = f x
```

Оператор `$` просто применяет функцию к значению! Его можно использовать для элегантной записи применения функции без использования скобок в последнем аргументе:

```
ghci> either (+1) (*2) (Right 100) :: Int
200
ghci> either (+1) (*2) $ Right 100 :: Int
200
```

Это очень удобно, когда аргумент оказывается длинным выражением:

```
someFunc ... =
  someOtherFunc ... $
    anotherFunc ... $
      yetAnotherFunc ... $
        andYetAnotherFunc ...
```

Однако это еще не вся история:

```
ghci> :i ($)
($) :: (a -> b) -> a -> b
infixr 0 $
```

Захотев получить дополнительную информацию об этом операторе, мы видим странное замечание: `infixr 0 $`. Оно говорит нам, что этот оператор записывается в *инфиксной* форме и является *правоассоциативным* (о чем свидетельствует буква `r` после `infix`) с *приоритетом* 0. Правоассоциативность означает, что если имеется несколько операторов с одинаковым приоритетом, то они группи-

руются справа налево. Приоритет 0 означает, что прочие операторы более приоритетны, чем $ (если только они сами не имеют приоритет 0). В чем польза?

```
ghci> :i (++)
(++) :: [a] -> [a] -> [a]
infixr 5 ++
ghci> map (*10) [1..5] ++ [6..10] :: [Int]
[10,20,30,40,50,6,7,8,9,10]
ghci> map (*10) $ [1..5] ++ [6..10] :: [Int]
[10,20,30,40,50,60,70,80,90,100]
```

Здесь мы видим, что поскольку оператор дописывания в конец (++) имеет приоритет 5, то он *связывает сильнее*, чем $. Следовательно, вычисление производится раньше, чем результат передается вызову функции map.

> **ПРИМЕЧАНИЕ** Оператор $ встречается в коде на Haskell сплошь и рядом. Поэтому так важно знать о его тонкостях и употреблении. Однако использовать его вовсе не обязательно. Если вам больше нравятся скобки, применяйте их сколько душе угодно.

В этой книге мы еще не раз столкнемся с этим оператором. Если он кажется вам непонятным, то помните, что всегда можно переписать выражение, добавив скобки.

Теперь мы можем конструировать значения типа Csv и предоставить внешнему миру безопасный интерфейс, гарантирующий, что ошибочные значения не просочатся!

7.3 Использование классов типов

Далее мы хотим написать функции, которые будут лучше работать с CSV-таблицами, а именно определять количество столбцов и строк. Работа с синтаксисом записей может оказаться несколько запутанной, поэтому мы познакомимся с несколькими расширениями языка, способными немного упростить ее. По умолчанию при работе с записями мы должны выписать поля, к которым собираемся обратиться, и дать им имена:

```
f Csv {csvHeader=h, csvColumns=c} = ...
```

Разрешается опускать ненужные поля:

```
f Csv {csvColumns=c} = ...
```

Но что, если мы не хотим давать полю новое имя?

```
f Csv {csvColumns=csvColumns} = ...
```

Хорошо было бы избежать присваивания этому полю избыточного имени. Это позволяет сделать расширение `NamedFieldPuns`:

```
f Csv {csvColumns} = ...
```

Мы все равно должны перечислить все поля, которые собираемся использовать. Если в записи много полей, то это становится слишком громоздко. Однако расширение языка `RecordWildCards` позволяет сделать *все* поля доступными по именам:

```
f Csv {..} = ...
```

Теперь мы можем легко обращаться к полям записи, не выписывая их! Это позволяет писать функции для вычисления количества строк и столбцов. При этом мы можем предполагать, что число элементов во всех столбцах одинаково (это инвариант нашего типа). Поэтому вычислить количество строк, зная столбцы, тривиально. Для определения количества столбцов нужно либо взять количество элементов заголовка, либо (если заголовок равен `Nothing`) подсчитать сами столбцы. Это легко сделать с помощью функции `maybe`, как показано ниже.

Листинг 7.7 Функции для вычисления количества строк и столбцов в CSV-файле

```
numberOfRows :: Csv -> Int
numberOfRows Csv {..} =
  case csvColumns of
    [] -> 0
    (x : _) -> length x

numberOfColumns :: Csv -> Int
numberOfColumns Csv {..} = length csvColumns
```

Извлекает все поля из записи CSV-файла и делает их доступными последующей части функции

Возвращает 0, поскольку если нет столбцов, то нет и строк

Возвращает длину первого столбца

Возвращает размер заголовка, если он существует, или количество столбцов в противном случае

Теперь, имея несколько функций для построения и описания CSV-таблиц, мы можем задуматься об их преобразовании. Мы хотим иметь возможность объединять таблицы, разрезать их и вычислять для них статистику. Начнем с объединения.

7.3.1 Полугруппа и моноид

Две объединяемые CSV-таблицы могут быть совершенно различны, т. е. количество строк может не совпадать, и одна может иметь заголовок, а другая нет. Это показано на рис. 7.1.

Чтобы исправить ситуацию, мы должны заполнить ячейки пустыми значениями. Заголовки можно просто оставить пустыми, а для полей данных использовать `NullValue`. Наша функция будет объединять заголовки, проверяя наличие заголовка хотя бы в одной таблице. Если его нет нигде, то и в новой таблице заголовка не будет. В противном случае отсутствующий заголовок будет заменен списком пустых строк нужной длины. При добавлении столбцов более короткие дополняются пустыми ячейками (`NullValue`). Код показан в листинге 7.8.

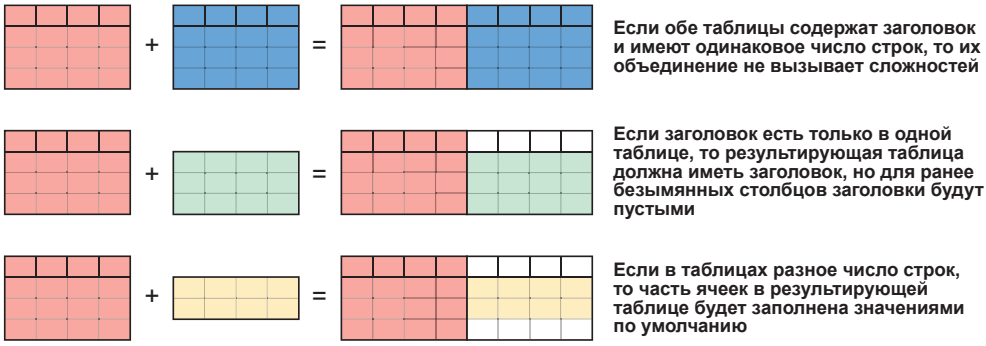


Рис. 7.1 Различные случаи при объединении двух таблиц

Листинг 7.8 Функция объединения двух CSV-таблиц

С помощью ограничения типа говорит, что для переменной типа добавляемых Csv-значений должен существовать экземпляр класса HasDefault

```
appendCsv :: Csv -> Csv -> Csv
appendCsv a b =
  Csv
    { csvHeader =
        if M.isNothing (csvHeader a) && M.isNothing (csvHeader b)
          then Nothing
          else Just $ header' a ++ header' b,
      csvColumns = appendColumns (csvColumns a) (csvColumns b)
    }
  where
    header' csv =
```

Добавленный заголовок устанавливается в Nothing, если ни у одной из объединяемых таблиц нет заголовка

Объединяет заголовки, подставляя вместо отсутствующих значений пустые строки

Вычисляет заголовок Csv-значения или возвращает список пустых строк размера, равного числу столбцов, если заголовок отсутствует

```
    M.fromMaybe
      (L.replicate (numberOfColumns csv) "")
      (csvHeader csv)

appendColumns colsA colsB =
  map (\cols -> cols ++ fillA) colsA
    ++ map (\cols -> cols ++ fillB) colsB
  where
    fillA = replicate (numberOfRows b - numberOfRows a) NullValue
    fillB = replicate (numberOfRows a - numberOfRows b) NullValue
```

Добавляет в конец столбцы, дописывая в ячейки более коротких столбцов значения по умолчанию

Сразу видно, что этот алгоритм работает только потому, что, по предположению, все столбцы имеют одну и ту же длину. В противном случае пришлось бы определять длину максимального столбца и дополнять все столбцы до этой длины.

Упражнение. Соединение CSV

`appendCsv` заполняет таблицу пустыми ячейками, если размеры двух CSV-файлов не совпадают. Можно было бы вместо этого просто отрезать строки у CSV-файла с более короткими столбцами. Реализуйте такой вариант объединения!

Заметим, что эти стратегии напоминают операции *внешнего* и *внутреннего соединения*, знакомые по базам данным. Однако мы соединяем CSV-файлы не по какому-то ключу или предикату, а просто дописывая в конец столбцы. Напишите функции, которые выполняют соединения структур типа `Csv`. Эта функция должна принимать булев предикат, который решает, соединять строки или нет.

Обратите внимание еще на одно свойство этой функции. Желая объединить три значения типа `Csv`, мы можем написать `appendCsv a (appendCsv b c)`. Однако порядок не важен: с тем же успехом можно было бы написать `appendCsv (appendCsv a b) c`. Это свойство, именуемое *ассоциативностью*, является фундаментальным математическим понятием. Оно позволяет не обращать внимания на порядок выполнения операций. Оно настолько важно, что в Haskell имеется специальный класс для типов, им обладающих: `Semigroup`.

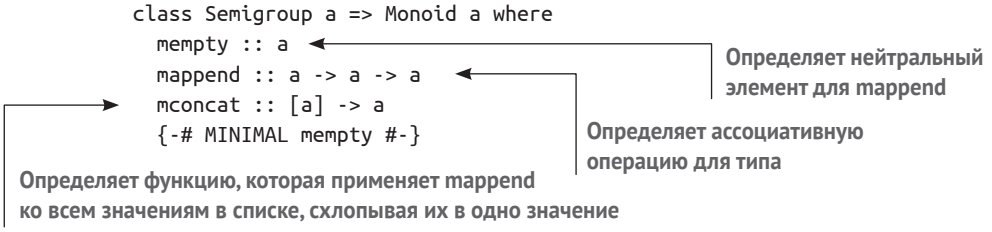
Самая важная операция этого класса типов – бинарная операция `<>`. По соглашению, это должна быть ассоциативная функция, т. е. равенство `a <> (b <> c) == (a <> b) <> c` должно иметь место для всех значений `a`, `b` и `c`, принадлежащих типу. Однако компилятор не проверяет и не доказывает выполнение этого свойства. Проверка возлагается на вас.

Для успешной компиляции важно только складывать операнды в правильном порядке, т. е. выражение `a <> b`, вычисляющее конечный результат, нельзя заменить на `b <> a`.

Расширением класса типов `Semigroup` является класс типов `Monoid`, показанный ниже.

Листинг 7.9 Сигнатура класса Monoid

```
class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
  {-# MINIMAL mempty #-}
```



`mappend` – такая же функция, как `<>`, и по умолчанию определена как `mappend = (<>)`. Однако `Monoid` также определяет константу `mempty`, являющуюся *нейтральным элементом* относительно `mappend`; это означает, что равенства `mappend mempty a == a` и `mappend a mempty == a` имеют место для любого `a`. `mconcat` подозрительно напоминает написанную нами ранее функцию комбинирования. Взглянем на `Monoid` в действии:

```
ghci> [1,2,3] <> mempty <> [4,5,6] :: [Int]
[1,2,3,4,5,6]
ghci> mconcat [[1,2,3], mempty, [4,5,6]] :: [Int]
[1,2,3,4,5,6]
```

Ну разве не похоже? У списков имеется экземпляр `Semigroup` и `Monoid`:

```
instance Semigroup [a] where
  (<>) = (++)

instance Monoid [a] where
  mempty = []
  mconcat xss = [x | xs <- xss, x <- xs]
```

Это означает, что и со значениями типа `String` можно работать так же!

```
ghci> "Hello" <> " " <> "World" :: String
"Hello World"
ghci> mconcat ["Hello", " ", "World"] :: String
"Hello World"
```

Однако существуют типы, для которых мы не можем определить `Semigroup` или `Monoid` из-за наличия *нескольких* ассоциативных функций! Типичным примером является тип `Int`, потому что в определении можно было бы использовать и сложение, и умножение. Но однозначного канонического выбора не существует. Именно поэто-

му имеются типы `Sum` и `Product`, которые обертывают `Int` и выбирают подходящую функцию для своих экземпляров типа классов.

> **ПРЕДУПРЕЖДЕНИЕ** Хотя в настоящее время `mappend` можно реализовать иначе, чем `<>`, никогда так не делайте. В будущем `mappend`, возможно, будет исключена из этого класса типов и заменена на `<>`.

Теперь, когда мы разобрались с этими классами типов, почему бы не реализовать для них экземпляры? Бинарной ассоциативной функцией `Semigroup` уже является `appendCsv`. Осталось только определить нейтральное значение `mempty` для `appendCsv`. Объединение с каким значением типа `Csv` не изменяет любое другое значение? Разумеется, с *пустым*, т. е. с таблицей, не имеющей ни заголовка, ни столбцов. Важно также, что у нашей функции имеется ограничение типа, которому должен удовлетворять экземпляр класса. Код показан ниже.

Листинг 7.10 Экземпляр `Semigroup` и `Monoid` для типа `Csv`

```
instance Semigroup Csv where
  (<>) = appendCsv

instance Monoid Csv where
  mempty = Csv {csvHeader = Nothing, csvColumns = []}
```

Определяет appendCsv в качестве бинарной ассоциативной функции

Определяет пустое значение типа Csv

Теперь мы можем использовать также функцию `mconcat` для конкатенации списка значений типа `Csv`!

> **Упражнение. Законы классов типов**
>
> Наше определение экземпляров `Csv` корректно, но как в этом удостовериться? Для этого нужно внимательно взглянуть на функцию `appendCsv`, а конкретно проверить, как дописываются данные. Почему наше определение `mempty` корректно? Попытайтесь установить, почему выполняется закон для этих классов типов!

С комбинированием нескольких CSV-таблиц мы разобрались, а как насчет их разрезания? Для этого нам придется обобщить способ разрезания структур данных.

7.3.2 Класс типов IsString

Как мы видели, значения типа `String` можно комбинировать методами из классов типов `Semigroup` и `Monoid`. Оказывается, что это работает и для значений типа `Text`:

```
ghci> import Data.Text
ghci> pack "Some text" :: Text
"Some text"
ghci> pack "Hello" <> pack " " <> pack "World" :: Text
"Hello World"
```

Чтобы упростить эту операцию, рассмотрим еще один класс типов, `IsString`. С его помощью записывать значения типа `Text` можно так же, как значения типа `String`:

```
ghci> :set -XOverloadedStrings
ghci> "Some text" :: Text
"Some text"
ghci> "Hello" <> " " <> "World" :: Text
"Hello World"
```

Имея это расширение, мы можем использовать типичный синтаксис строковых литералов для записи значений любых типов, имеющих экземпляр `IsString`. В частности, это относится к типам `String` *и* `Text`! Вот почему мы часто видим, что строковые значения конкатенируются оператором `<>`, а не `++`.

Но как это работает? Если активировано расширение `OverloadedStrings`, то значения можно записывать как строки, которые позднее будут преобразованы в настоящий тип методом `fromString` из класса типов `IsString`:

```
class IsString a where
    fromString :: String -> a
```

Следовательно, мы можем так записывать все типы, для которых определен экземпляр `IsString` (из модуля `Data.String`). Что особенно удобно, к таким типам относится `Text` и, кстати, `ByteString` тоже.

> **ПРИМЕЧАНИЕ** Помимо `OverloadedStrings` и `IsString`, существует еще расширение `OverloadedLists` и класс типов `IsList`, которые позволяют записывать списки и преобразовывать их в другие типы. Это очень полезно при определении типов данных, похожих на списки, но требующих громоздкого синтаксиса для записи литералов.

Поскольку это расширение ничему не вредит и чрезвычайно полезно при разработке проектов с участием таких типов, мы хотим активировать его глобально. Это делается в файле `package.yaml` нашего проекта. Туда мы можем добавить список активируемых по умолчанию расширений. И заодно добавим вышеупомянутые расширения `RecordWildCards` и `NamedFieldPuns`, поскольку они также универсально полезны:

```
default-extensions:
  - OverloadedStrings
  - RecordWildCards
  - NamedFieldPuns
```

Таким образом, мы активировали эти расширения для всех модулей проекта (включая внешние библиотеки).

> **ПРИМЕЧАНИЕ** Haskell поддерживает специальное «издание» языка, в котором активировано много расширений, считающихся настолько безопасными, что их можно использовать всегда. Найти его можно по адресу https://mng.bz/2gwm.

Получив опыт работы с готовыми классами типов, обратим внимание на написание своих собственных!

7.4 Создание нового класса типов

Когда класс типов вроде `Semigroup` можно использовать для комбинирования данных, он может оказаться полезным и для определения способа разрезания данных. Практически нас интересует выделение частей таблицы и работа с ними как с новыми таблицами. Это можно распространить и на другие структуры данных, например списки. И тогда возникает вопрос: можно ли обобщить разрезание структуры данных?

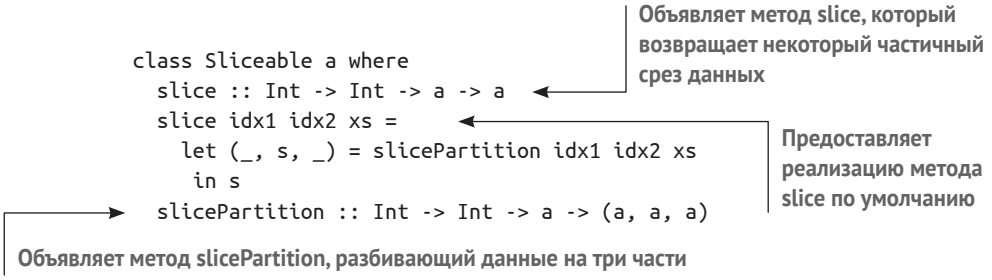
Преследуя эту цель, мы начнем с создания модуля `Data.Sliceable` в новом каталоге `src/Data`. Этот модуль будет содержать весь код, необходимый для поддерживающих разрезание данных. Мы хотим создать в этом модуле новый класс типов и определить в нем операции, нужные для разделения структуры данных на части.

Разрезание данных означает выделение некоторого диапазона внутри структуры данных. Мы будем предполагать, что структура допускает индексирование. Тогда диапазон можно определить двумя индексами. Значение при этом разбивается на три части, но нас будет интересовать только средняя.

7.4.1 Класс типов для вырезания структур данных

Теперь можно подумать о методах, которые должен раскрывать наш класс типов. Особенно важен метод `slice`, позволяющий вырезать часть структуры данных, ограниченную двумя индексами. Однако поскольку разбиение дает интересующий нас срез задаром, стоит также завести метод `slicePartition`. И тогда метод `slice` можно будет реализовать на основе метода `slicePartition`! Реализация такого класса показана в листинге ниже.

Листинг 7.11 Определение класса для допускающих вырезание данных




```
class Sliceable a where
  slice :: Int -> Int -> a -> a
  slice idx1 idx2 xs =
    let (_, s, _) = slicePartition idx1 idx2 xs
     in s
  slicePartition :: Int -> Int -> a -> (a, a, a)
```

Поскольку этот класс вводит общую концепцию, мы можем поместить его в модуль `Data.Sliceable`. Здесь `Sliceable` – имя нашего класса, а `a` – переменная типа, для которой мы сейчас определяем выражения типа. Лексическая область видимости этой переменной – класс, так что если некий тип содержит переменную `a`, то она связана с типом, для которого будет позднее создан экземпляр этого класса. Далее определяется два метода: `slice` и `slicePartition`.

В этом коде мы также видим, как определять поведение метода по умолчанию. Объявленные в классе методы известны другим определениям. Определенная нами реализация `slice` – это реализация по умолчанию, которая автоматически выводится, если явно не определена в экземпляре.

> **ПРИМЕЧАНИЕ** Реализации методов по умолчанию можно переопределять. Это полезно, когда для конкретного типа удается найти более производительную реализацию.

Загрузив новый модуль и запросив информацию о нашем классе типов в GHCi, мы увидим следующее:

```
ghci> import Data.Sliceable
ghci> :i Sliceable
type Sliceable :: * -> Constraint
class Sliceable a where
  slice :: Int -> Int -> a -> a
  slicePartition :: Int -> Int -> a -> (a, a, a)
  {-# MINIMAL slicePartition #-}
```

Как и для многих других классов, мы видим минимальное определение, содержащее только наш метод `slicePartition`, потому что `slice` можно вывести из него! Теперь можно сконструировать экземпляр для списков, воспользовавшись функциями `take` и `drop`; первая выбирает из списка определенное число элементов и возвращает их, вторая удаляет определенное число элементов из списка и возвращает то, что осталось. *Срезом* списка между двумя индексами на-

зывается список, который содержит элементы, начиная с позиции первого индекса в исходном списке и кончая позицией второго, но не включая элемент в этой позиции. В результате мы получаем список, длина которого равна разности индексов. Код показан в следующем листинге.

Листинг 7.12 Экземпляр класса типов `Sliceable` для списков

```
instance Sliceable [a] where
  slicePartition idx1 idx2 xs =
    ( take idx1 xs,
      take (idx2 - idx1) $ drop idx1 xs,
      drop idx2 xs
    )
```

Определяет функцию slicePartition, которая разбивает список на три части

Определяет первую часть списка как множество элементов до начала среза

Определяет сам срез

Определяет третью часть списка как множество элементов после среза

Это определение автоматически дает функцию вырезания!

```
ghci> import Data.Sliceable
ghci> slicePartition 2 8 [0..9] :: ([Int], [Int], [Int])
([0,1],[2,3,4,5,6,7],[8,9])
ghci> slice 2 8 [0..9] :: [Int]
[2,3,4,5,6,7]
```

Кроме того, мы можем найти еще один экземпляр этого класса для `Maybe`. Если тип, обернутый `Maybe`, допускает вырезание, то мы можем определить срез для `Maybe` от этого типа. Либо части разбиения помещаются в `Just`, либо каждая часть равна `Nothing`. Экземпляр определен в листинге ниже.

Листинг 7.13 Экземпляр класса типов `Sliceable` для `Maybe`

```
instance Sliceable a => Sliceable (Maybe a) where
  slicePartition idx1 idx2 Nothing =
    (Nothing, Nothing, Nothing)
  slicePartition idx1 idx2 (Just xs) =
    let (hd, s, tl) = slicePartition idx1 idx2 xs
     in (Just hd, Just s, Just tl)
```

Определяет экземпляры для любого Maybe, параметр-тип которого также имеет экземпляр Sliceable

Возвращает Nothing в качестве всех частей разбиения, если на вход было подано Nothing

Вызывает slicePartition для значения, обернутого Just

Обертывает результат slicePartition конструкторами Just

Как и функции, классы типов могут иметь ограничения типа. Мы воспользовались этим в листинге 7.13, желая выразить тот факт, что значения, обернутые `Maybe`, должны допускать вырезание, чтобы можно было предоставить реализацию для конкретного типа `Maybe`.

Поэтому мы можем вызывать метод `slicePartition`, предоставляемый экземпляром `Sliceable a`, в нашем определении экземпляра класса для `Sliceable (Maybe a)`.

Теперь мы можем использовать `slice` для любого значения `Maybe a`, для которого `a` имеет экземпляр `Sliceable`:

```
ghci> slicePartition 2 8 $ (Just [0..9] :: Maybe [Int])
(Just [0,1],Just [2,3,4,5,6,7],Just [8,9])
ghci> slice 2 8 $ (Just [0..9] :: Maybe [Int])
Just [2,3,4,5,6,7]
ghci> slice 2 8 $ (Nothing :: Maybe [Int])
Nothing
```

Оба экземпляра важны для нашего типа `Csv`, потому что он, по сути дела, состоит из списка, обернутого `Maybe`, и списка списков. При вырезании частей CSV-таблиц мы разрезаем таблицы по границам столбцов и создаем новые значения на основе этих срезов. Это показано на рис. 7.2.

Значение типа `Csv` можно разрезать, по отдельности разрезав заголовок и столбцы. Для этого годятся уже созданные нами экземпляры! Код показан в листинге 7.14.

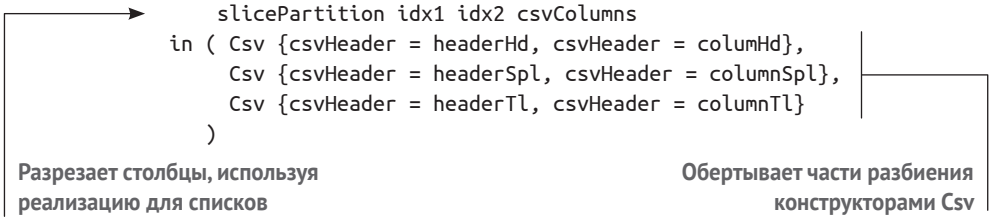


Рис. 7.2 Разрезание CSV-таблиц

Листинг 7.14 Экземпляр класса типов `Sliceable` для CSV-таблиц

```
instance Sliceable Csv where
 slicePartition idx1 idx2 Csv {..} =
  let (headerHd, headerSpl, headerTl) =
       slicePartition idx1 idx2 csvHeader
      (columHd, columnSpl, columnTl) =
```

Разрезает заголовок, используя реализацию для Maybe



Благодаря этому коду мы теперь можем разрезать произвольные значения типа `Csv`.

> **Упражнение. Отображение срезов**
>
> Метод `slicePartition` годится и для написания других функций, поскольку он разбивает значение на меньшие части, которые можно использовать независимо друг от друга. Одна такая функция – `sliceMap`, которая отображает срез, но не другие части разбиения, а затем снова склеивает части. Напишите эту функцию полиморфным образом. В ограничении типа вам понадобится второй класс типов.
>
> Аналогично напишите функцию `sliceDelete`, которая удаляет срез из значения. Тут тоже придется подумать о классах типов.

7.4.2 Реэкспорт модулей

Реализовав часть функциональности приложения, мы должны еще раз рассмотреть структуру проекта и подумать, какие определения мы хотим импортировать в другие модули – и в какие именно.

В Haskell импорт одних модулей в другие – удобный способ совместного использования кода, но без проблем не обходится. Конфликты имен и – что еще хуже – циклический импорт может помешать успешной компиляции проекта. Чтобы не попасться в эти ловушки, имеет смысл собирать модули, работающие с одними и теми же концепциями, в отдельные файлы для реэкспорта. В нашем проекте мы соберем функциональность, относящуюся к CSV, в каталоге Csv, который будет содержать модули для работы с типом `Csv`. В этом каталоге мы можем сначала определить модуль `Types`, который будет экспортировать все необходимые типы; `Csv` – один из таких типов. Затем нужно будет определить модуль `Conversion`, используемый для определения различных преобразований данных для нашей программы. Структура каталога будет выглядеть так:

```
src
├── Csv
│   ├── Conversion.hs
│   └── Types.hs
└── Data
    └── Sliceable.hs
```

Теперь мы хотим поместить на верхний уровень модуль `Csv`, раскрывающий наружу модули из подкаталога `Csv`. Сделать это можно с помощью списка экспорта, сославшись на модуль посредством ключевого слова `module`, как показано в листинге ниже.

Листинг 7.15 Реэкспорт модулей

```
module Csv
  ( module Csv.Conversion,
    module Csv.Types,
  )
where

import Csv.Conversion
import Csv.Types
```

Реэкспортирует импортированные модули из модуля Csv

Импортирует модули в модуль Csv

Это позволит нам импортировать только один модуль и при этом получить все необходимые определения, не беспокоясь о циклическом импорте. В этом модуле может также находиться код, нуждающийся в определениях из нескольких модулей.

Пришло время протестировать функции. Но прежде нам нужно заполучить в программе CSV-таблицы. Поэтому в следующей главе мы изучим разбор CSV-файлов.

Резюме

- Записи можно использовать для моделирования сложных данных с автоматически генерируемыми функциями доступа.
- Тип `Either` можно использовать для кодирования программных ошибок, что дает возможность порождать значение ошибки, например сообщение о ней.
- Умные конструкторы – это функции, которые используются для построения значения, а также для проверки свойств этих значений.
- Расширения языка `RecordWildCards` и `NamedFieldPuns` можно активировать, чтобы упростить работу с записями.
- `Semigroup` и `Monoid` – классы типов, определяющие ассоциативную функцию в типе.
- Благодаря классам типов мы можем обобщить поведение различных типов и сделать код, в котором они используются, более общим.
- Реэкспорт модулей можно использовать, чтобы собрать определения, экспортируемые из нескольких модулей, в одном модуле.

8 Программа для работы с CSV-файлами

Краткое содержание этой главы:

- разбор числовых значений;
- определение и использование сверток структур данных;
- использование класса типов `Functor` и его возможности обработки ошибок;
- создание простого интерфейса для разбора командной строки.

В предыдущей главе мы начали разрабатывать каркас приложения для работы с CSV-файлами. В этой главе мы завершим проект и представим функциональность для разбора и печати данных типа `Csv`.

Мы продолжим с того места, где остановились, и продемонстрируем такие важные концепции, как класс типов `Functor` и свертка, чтобы познакомить вас с основополагающими идеями функционального программирования. И завершим главу, применив все, чему научились, к созданию короткой программы, в которой будет использоваться наша библиотека. И наконец, мы установим собранный двоичный файл локально на наш компьютер.

8.1 Разбор данных

Следующий наш шаг – наделение программы возможностью читать и разбирать CSV-файлы. Для этого нам понадобится функция, умеющая читать содержимое такого файла, выделять разделители и поля и записывать результат в структуру данных.

Разбор CSV-файлов – трудная задача, потому что каждое приложение, похоже, делает это по-своему, особенно в части выбора разделителей и символов новой строки. Заголовки CSV-файлов также факультативны, но не существует надежного способа определить, имеет файл заголовок или нет. Поэтому придется задавать параметры разбора. Смоделировать параметры можно с помощью типов, как показано в следующем листинге.

Листинг 8.1 Параметры для разбора CSV-файлов

```
data Separators = Separators
  { sepLineSeparator :: Char,
    sepFieldSeparator :: Char
  }
  deriving (Eq, Show)

data HeaderOption = WithHeader | WithoutHeader
  deriving (Eq, Show)

data CsvParseOptions = CsvParseOptions
  { cpoSeparators :: Separators,
    cpoHeaderOption :: HeaderOption
  }
  deriving (Eq, Show)

defaultSeparators :: Separators
defaultSeparators =
  Separators
  { sepLineSeparator = '\n',
    sepFieldSeparator = ','
  }

defaultOptions :: CsvParseOptions
defaultOptions =
  CsvParseOptions
  { cpoSeparators = defaultSeparators,
    cpoHeaderOption = WithoutHeader
  }
```

Определяет тип данных для разделителей строк и полей

Определяет параметр для заголовков

Определяет комбинацию параметров разбора

Определяет разделители по умолчанию: знак новой строки для разделения строк и запятую для разделения полей

Определяет параметры разбора CSV-файлов по умолчанию

Эти параметры находятся в новом модуле `Csv.Parsing`, который мы также можем реэкспортировать из модуля `Csv`. Впоследствии мы научимся задавать эти параметры в командной строке. А в качестве

разумного умолчания будем предполагать, что файл не содержит заголовка, а его поля разделяются запятыми.

Что до правил разбора, то нам нужно будет принять определенные предположения и правила обращения с CSV-файлами. Как было отмечено в самом начале, мы считаем, что:

- CSV-файлы содержат одну запись в каждой строке, а между строками находятся разделители;
- записи разбиты на поля, разделенные одним и тем же символом;
- число полей во всех записях одинаково;
- может присутствовать факультативная строка-заголовок.

Кроме того, мы будем *пропускать пустые строки*, не считая, что они нарушают структуру файла. Также мы будем игнорировать поля в кавычках и экранированные двойные кавычки, которые могут встречаться в закавыченных полях, потому что их разбор сделал бы задачу слишком сложной для учебного упражнения.

Тем не менее мы хотим, чтобы наш парсер выдавал полезную информацию, когда файл не удается разобрать. А именно парсер должен сообщать, что обнаружил строки, содержащие разное число полей. Хотя наши умные конструкторы уже дают такую возможность, мы все же хотим получать сведения о номере строки, при разборе которой произошла ошибка.

Для этого мы должны сначала разбить текст на строки, затем эти строки разбить на поля и проверить, все ли длины строк одинаковы. Первым делом озаботимся разбиением значения типа `Text` на строки и поля. К счастью, модуль `Data.Text` уже предоставляет функцию `split`, которая разбивает `Text` по булеву предикату, и функцию `splitOn`, разбивающую по заданной подстроке:

```
ghci> import qualified Data.Text as T
ghci> :t T.split
T.split :: (Char -> Bool) -> T.Text -> [T.Text]
ghci> :t T.splitOn
T.splitOn :: T.Text -> T.Text -> [T.Text]
ghci> T.split (== '\n') "a\nb\nc"
["a","b","c"]
ghci> T.splitOn "\n" "a\nb\nc"
["a","b","c"]
```

Таким образом, для разбиения значений типа `Text` на строки и строк на поля можно воспользоваться функцией `splitOn` с разделителями, определенными в типе `CsvParseOptions`. Далее займемся разбором произвольного текста с целью выделения осмысленных данных.

8.1.1 Разбор числовых значений

В процессе разбора мы имеем дело только с типом `Text`. Но в конечном итоге мы хотели бы преобразовать `Text` из разобранной CSV-таблицы в нечто, с чем можно работать, например в числовые и текстовые значения (т. е. в значения типа `DataField`).

Для преобразования в тип `Int` можно воспользоваться функцией `readMaybe` из модуля `Text.Read`. Эта функция умеет разбирать значения с помощью экземпляра класса типов `Read`, преобразуя их из `String` в значение подходящего типа или возвращая `Nothing`, когда преобразовать не получается:

```
ghci> import Text.Read
ghci> :t readMaybe
readMaybe :: Read a => String -> Maybe a
ghci> readMaybe "100" :: Maybe Int
Just 100
ghci> readMaybe "abc" :: Maybe Int
Nothing
```

Это можно использовать для написания функции, которая преобразует `Text` в значения типа `DataField`. Кроме того, мы предоставим функцию, которая преобразует `DataField` обратно в `Text`. Код обеих функций показан ниже.

Листинг 8.2 Функции преобразования `DataField` в `Text` и обратно

```
textToDataField :: T.Text -> DataField
textToDataField "" = NullValue
textToDataField raw =
  let mIntVal = readMaybe (T.unpack raw)
   in maybe (TextValue raw) IntValue mIntVal

dataFieldToText :: DataField -> T.Text
dataFieldToText (IntValue i) = T.pack $ show i
dataFieldToText (TextValue t) = t
dataFieldToText NullValue = ""
```

Возвращает NullValue, если вход пустой

Пытается преобразовать значение типа Text в Int

Возвращает значение, которое было обернуто конструктором TextValue

Возвращает пустую строку типа Text

Возвращает IntValue, если разбор был успешен, иначе возвращает вход, преобразованный в TextValue

Преобразует IntValue в Text, для чего сначала преобразует числовое значение в String, а затем упаковывает строку

Заметим, что механизм вывода типа способен определить, что полиморфная функция `readMaybe` должна вернуть `Maybe Int`, поскольку к правильно разобранному значению применяется `IntValue`.

8.2 Свертка структур данных

Чтобы убедиться в корректности содержимого файла, мы должны проверить, что число полей во всех строках одинаково. Сначала разберемся с одной строкой. После разбиения файла на строки мы получаем список строк, подлежащих проверке. Конечно, мы знаем, как решить задачу с помощью рекурсии и сопоставления с образцом, но давайте отступим на шаг назад. Мы уже сталкивались с этой проблемой бессчетное число раз. Наша стратегия обхода структуры данных, подобной списку, и вычисления при этом какого-то результата формулируется следующим образом:

- рекурсивно перечислить все элементы структуры данных;
- хранить некоторое состояние в аргументе-аккумуляторе;
- вычислять новый аккумулятор на каждом шаге по элементу из структуры данных и аккумулятору на предыдущем шаге;
- когда структура данных окажется пустой (выполнен последний шаг рекурсии), вернуть аккумулятор.

Это очень действенная стратегия – настолько действенная, что используется повсеместно в функциональном программировании, потому что устраняет необходимость циклов. Она хорошо известна под названием *свертка*[1]. Прежде чем разбираться в том, что можно делать с помощью свертки, рассмотрим эту идею на шутливом примере.

8.2.1 Концепция свертки

Представьте, что перед вами тропинка. На тропинке разбросаны леденцы разных сортов (конечно, в фантиках). Как это смоделировать? Можно с помощью списка:

```
data Candy
  = Lemon
  | Apple
  | Coffee
  | Caramel
  deriving (Eq, Show)

type CandyTrail = [Candy]
```

Теперь предположим, что вы идете по тропинке и для каждой найденной конфетки должны решить, что с ней делать. Собранные конфетки вы держите в руке, и, наткнувшись на следующую конфетку, вы принимаете решение, что делать дальше, исходя из того, что

[1] Часто употребляется также термин редукция. – *Прим. перев.*

у вас уже есть, и того, что нашли. Очевидно, что в начале прогулки в руке у вас ничего не было, а в конце у вас будет только то, что вы в ней держите. Это можно представить так:

```
walkOnTrail :: (a -> Candy -> a) -> a -> CandyTrail -> a
walkOnTrail _ hand [] = hand
walkOnTrail f hand (x : xs) = walkOnTrail f (f hand x) xs
```

Имея эту функцию высшего порядка, вы можете выполнить несколько действий. Например, можно собирать только леденцы с фруктовым вкусом или оставить только последние пять, потому что больше в руке не помещается:

```
isFruity :: Candy -> Bool
isFruity c = c == Lemon || c == Apple

collectFruits :: CandyTrail -> [Candy]
collectFruits =
  walkOnTrail
    ( \hand c ->
        if isFruity c
          then hand ++ [c]
          else hand
    )
    []

collectLastFive :: CandyTrail -> [Candy]
collectLastFive =
  walkOnTrail
    ( \hand c ->
        if length hand == 5
          then tail hand ++ [c]
          else hand ++ [c]
    )
    []
```

Теперь представим, что леденцовая тропинка – это простой список (или вообще любая структура данных, которую можно обойти, как список). Обобщение функции `walkOnTrail` подводит нас к определению *левой свертки*. Это определение для списков показано в следующем листинге.

Листинг 8.3 Функция левой свертки для списков

```
foldLeft :: (b -> a -> b) -> b -> [a] -> b
foldLeft _ z [] = z
foldLeft f z (x : xs) = foldLeft f (f z x) xs
```

Возвращает аккумулятор для пустого списка

Рекурсивно сворачивает остаток списка с вновь вычисленным аккумулятором

Эта свертка называется *левой*, потому что элементы сворачиваются слева направо. Можно также создать *правую свертку*, если сначала обойти список, а потом применить функцию. Это показано в листинге ниже.

Листинг 8.4 Функция правой свертки для списков

```
foldRight :: (a -> b -> b) -> b -> [a] -> b
foldRight _ z [] = z
foldRight f z (x : xs) = f x $ foldRight f z xs
```

Возвращает аккумулятор для пустого списка

Применяет функцию к голове и рекурсивно свернутому хвосту списка

Свертка – настолько общая концепция, что для нее имеется специальный класс типов: `Foldable`. Этот класс предоставляет правую свертку `foldr`, а также много вспомогательных функций, например `null`, `length`, `elem`, `sum`, `maximum` и `minimum`. Но в минимальном определении требуется реализовать только функцию `foldr`, поскольку все остальные можно вывести из нее!

Упражнение. Сила сворачиваемости

Рассмотрим класс типов `Foldable` внимательнее. Для начала попробуйте реализовать его вспомогательные функции, используя `foldr` и ничего более. Эти функции можете реализовать только для списков. Затем возьмите простое двоичное дерево и реализуйте для него экземпляр `Foldable`. Что такое левая и правая свертка применительно к деревьям?

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

Свертку проще всего представить себе для списков и других линейных структур, потому что элементы выстроены по порядку, допуская сворачивание слева направо или в обратном направлении. Модули `Data.List` и `Data.Text` предоставляют реализации `foldl` и `foldr` для списков и текста соответственно. Однако свертку можно определить и для многих других структур данных.

> **ПРИМЕЧАНИЕ** Свертка имеется и в других, нефункциональных языках программирования, например `reduce` в Python и Java и `accumulate` в C++.

Хотя термины *левая* и *правая* применительно к свертке наводят на мысль о том, что элементы вычисляются соответственно слева направо или справа налево, на самом деле это не так. Обе функции свертки вычисляются слева направо. Разница в том, что `foldl` *ассоциирует* аккумулирующую функцию слева, а `foldr` – справа. Сво-

рачивая структуру данных, мы, по сути дела, вычисляем член выражения, в которое входят элементы этой структуры и некоторая функция. Поскольку вычисления ленивые, выражение не вычисляется, пока какое-то событие не заставит это сделать. Можно сказать, что выражения выглядят, как в примере ниже:

```
ghci> foldl (\a d -> "(" <> a <> "+" <> show d <> ")") "0" [1..9 :: Int]
"(((((((((0+1)+2)+3)+4)+5)+6)+7)+8)+9)"
ghci> foldr (\d a -> "(" <> show d <> "+" <> a <> ")") "0" [1..9 :: Int]
"(1+(2+(3+(4+(5+(6+(7+(8+(9+0)))))))))"
```

Здесь хорошо видна левая ассоциативность `foldl` и правая ассоциативность `foldr`. С помощью свертки мы строим новое выражение по структуре данных. Это показано на рис. 8.1.

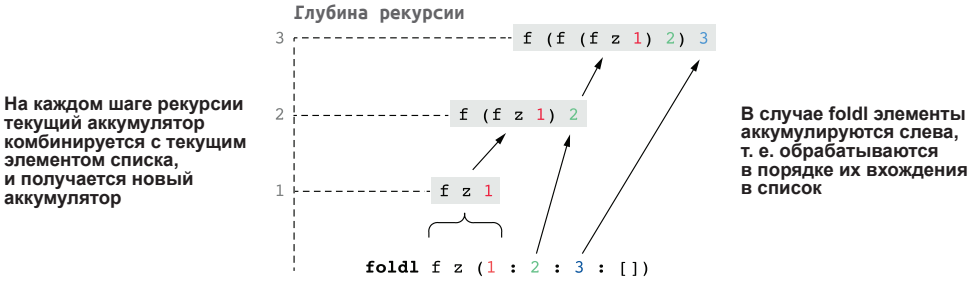


Рис. 8.1 Пример построения выражения с помощью свертки

У ленивости Haskell могут быть неожиданные последствия, обсуждаемые в приложении B. В двух словах, она может приводить к катастрофически неэффективному использованию памяти. Для борьбы с этими эффектами иногда предлагаются строгие версии сверток. В модулях `Data.List` и `Data.Text` они называются `foldl'`.

> **ПРИМЕЧАНИЕ** Есть общее правило: если аккумулирующая функция ленивая (точнее, нестрогая по второму аргументу), то следует использовать `foldr`, в противном случае – `foldl'`.

Кроме того, бывает, что мы не хотим задавать «начальное значение» аккумулятора, а предпочитаем использовать для этой цели первый элемент структуры данных. Для такого варианта свертки многие модули предоставляют функцию с суффиксом `1` (например, `foldl1`), но при этом требуется, чтобы структура имела хотя бы один элемент.

> **ПРЕДУПРЕЖДЕНИЕ** Функции вида `foldl1` частичные и возбуждают исключение, если аргумент вообще не содержит элементов.

Рассматривая реализации в прошлых проектах, мы понимаем, что могли бы использовать свертку в очень многих функциях. В главе 5 даже было упражнение на тему обобщения рекурсии в функциях `addEdges` и `buildDiGraph`. Для его решения нужно реализовать свертку, которая выполняет некоторое действие для одного элемента из списка!

8.2.2 Структура данных для разбора

Теперь, зная, что такое свертки и как с ними работать, мы можем воспользоваться этими знаниями для проверки строк CSV-файла. Для этого мы должны обойти все строки, разбить их на поля, а затем проверить длину списка полей, отслеживая при этом номер текущей строки. Конечно, добавление номеров строк к самим строкам – детская забава для любого, кто прочел главу 3. Нет ничего проще, чем воспользоваться функцией `zip` для добавления номеров и функцией `filter` для избавления от пустых строк. Наша функция разбора, которую мы назвали `parseCsv`, получает параметры в виде значения типа `CsvParseOptions` и исходное содержимое файла типа `Text`, а возвращает либо значение типа `Csv`, либо сообщение об ошибке. Ее результат можно представить типом `Either`.

Чтобы убедиться в корректности файла, мы возьмем первую строку, а затем проверим, что все остальные строки содержат столько же полей, сколько она. Функция разбора приведена в листинге ниже.

Листинг 8.5 Функция разбора CSV-файла

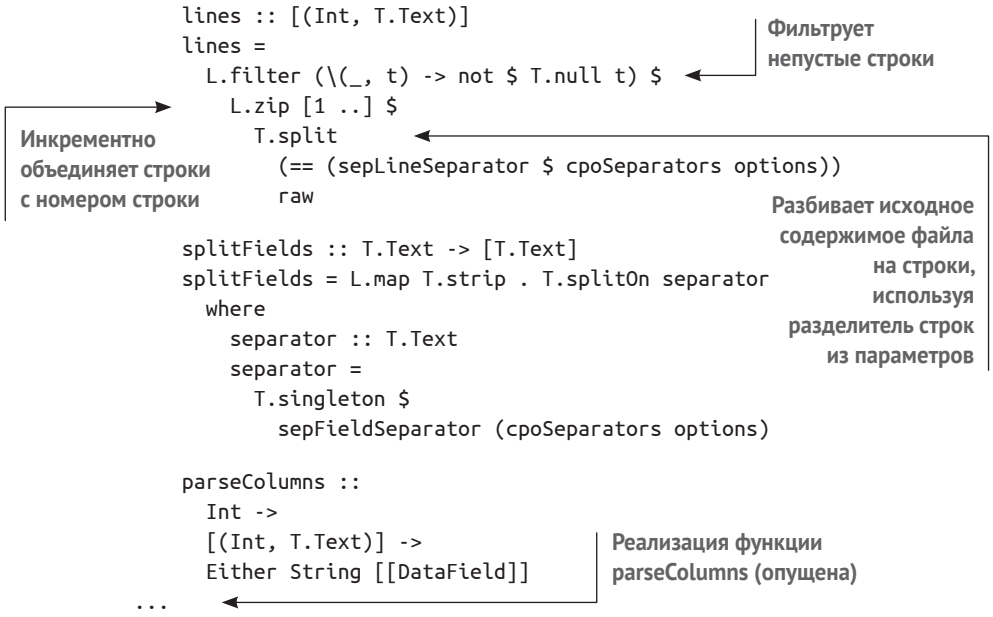
```
parseCsv ::
  CsvParseOptions ->
  T.Text ->
  Either String Csv
parseCsv options raw = case lines of
  [] -> mkCsv Nothing []
  ((_, firstLine) : rest) ->
    let expectedLength = length $ splitFields firstLine
     in case cpoHeaderOption options of
          WithHeader ->
            let headerFields = splitFields firstLine
             in unsafeKkCsv (Just headerFields)
                  <$> parseColumns expectedLength rest
          WithoutHeader ->
             unsafeMkCsv Nothing <$>
  ➥ parseColumns expectedLength lines
  where
```

Определяет ожидаемую длину строки таблицы как количество полей в первой строке файла вне зависимости от того, представляет она заголовок или нет

Возвращает пустое значение Csv, если строк нет

Создает Csv с заголовком из позитивного (Right) результата, возвращенного parseColumns

Создает Csv без заголовка из позитивного (Right) результата, возвращенного parseColumns



В этой функции мы используем оператор <$> (операторную форму функции `fmap`), чтобы передать `Left` из `parseColumns` в наше возвращаемое значение. Чтобы лучше понять, что делают этот оператор и функция `fmap`, нам нужно познакомиться с классом типов `Functor`.

8.2.3 Класс типов Functor

Откуда берется функция `fmap`? Что означает ее имя? Кое-что о ней может рассказать GHCi:

```
ghci> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
ghci> :i fmap
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Имя `fmap` связано с классом типов `Functor`. Ее тип до странности напоминает тип `map` для списков:

```
ghci> :t map
map :: (a -> b) -> [a] -> [b]
ghci> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
```

Так какова же цель `Functor`? Он позволяет определять типы, допускающие отображение. По большей части это контейнеры, которые содержат допускающие отображение значения. Точнее, эти типы

содержат свободные переменные типа, которые можно отобразить. Чего мы еще не видели в выражениях типа, так это типов вроде `f a`, где `f` – тип функтора, а `a` – его переменная типа. Какой тип можно было бы подставить вместо `f`? Например, `Maybe`, потому что он содержит свободную переменную типа. Другой кандидат – списки! Оба этих типа имеют экземпляр класса типов `Functor`. Мы уже видели `fmap` для `Maybe` – ничего особенного. Для списков `fmap` еще скучнее – это просто `map`!

Но вернемся к обсуждению типов с переменными типа. Не всякий тип обладает таким свойством (например, `Int`). Это означает, что мы не можем создать экземпляр `Functor` для `Int`. О том же говорит и вывод GHCi:

```
ghci> :i fmap
type Functor :: (* -> *) -> Constraint
```

Здесь перед нами *род* (kind) типов `Functor`. Род для типов – то же, что тип для значений. Как всякое значение имеет тип, так всякий тип имеет род. Роды, с которыми мы встречаемся в повседневном программировании, легко читаются:

- `*` (читается как *тип*) – *мономорфный тип*, например `Int`, `Maybe String` или `Maybe [Bool]`;
- `* -> *` – тип, принимающий параметр.
  - Обобщая, тип, принимающий два параметра, обозначается `* -> * -> *`, что можно прочитать как `* -> (* -> *)` (тип, который принимает параметр и результатом вычисления которого является тип, принимающий другой параметр);
- `Constraint` – тип, используемый в качестве ограничения, например в ограничениях классов.

Теперь мы понимаем, как читать `(* -> *) -> Constraint`. Это тип, который принимает другой тип, который принимает тип в качестве параметра и становится `Constraint`. Давайте исследуем роды более подробно в GHCi с помощью команды `:kind` (или сокращенно `:k`):

```
ghci> :kind Int
Int :: *
ghci> :kind []
[] :: * -> *
ghci> :kind [Int]
[Int] :: *
ghci> :kind Functor []
Functor [] :: Constraint
```

Это тоже важное различие. `Int` – тип, а `[]` – параметризованный тип. С другой стороны, `Functor []` – не тип, а `Constraint`, т. е. мы не можем использовать его для определения новых типов данных!

```
ghci> newtype MyFun = MyFun (Functor [])

<interactive>...: error:
    • Expected a type, but 'Functor []' has kind 'Constraint'
    • In the type '(Functor [])'
      In the definition of data constructor 'MyFun'
      In the newtype declaration for 'MyFun'
```

Когда мы рассуждаем о том, какие типы допускают создание экземпляров, нужно обязательно смотреть на род `Functor (* -> *)`! Инстанцировать можно только классы типов с типами этого рода. Это прямая противоположность `Semigroup`, `Monoid` и нашему классу `Sliceable`, которые имеют род `* -> Constraint`.

8.2.4 Использование свертки при разборе

Вот теперь мы можем заняться функцией `parseColumns`, которая должна разбивать строки на поля и проверять количество полей в каждой строке. Кроме того, мы хотим преобразовать разобранные строки в столбцы, которые можно использовать в нашем типе `Csv`. Как уже было сказано, свертка может оказаться подходящим решением для этой функции. В процессе свертывания мы перебираем все строки по порядку, преобразуем их в поля и проверяем длину, возвращая либо `Left` с сообщением об ошибке, либо `Right` в случае успеха. В нашем случае мы должны проверить, была ли ранее обнаружена ошибка, и если да, вернуть ее; в противном случае мы продолжаем аккумулировать разобранные строки. Для преобразования строк в столбцы результат нужно «повернуть». Список строк становится списком столбцов, если поменять местами измерения. Это может сделать функция `transpose` из модуля `Data.List`:

```
ghci> import Data.List (transpose)
ghci> transpose [[1,2,3], [4,5,6]] :: [[Int]]
[[1,4],[2,5],[3,6]]
```

И снова мы можем использовать `fmap` для передачи состояния ошибки наружу. Код функции показан ниже.

Листинг 8.6 Функция разбора строк текста и преобразования их в столбцы CSV-таблицы

```
parseColumns ::
  Int ->
  [(Int, T.Text)] ->
  Either String [[DataField]]
parseColumns expectedLength lines =
  let textColumns =
        L.transpose
```

Транспонирует результат Right

```
        <$> L.foldl' parseRow (Right []) lines
  in fmap (L.map (L.map textToDataField)) textColumns
where
  parseRow ::
    Either String [[T.Text]] ->
    (Int, T.Text) ->
    Either String [[T.Text]]
  parseRow mRows (lNum, line) =
    E.either
      Left
      ( \rows ->
          let fields = splitFields line
            in if length fields /= expectedLength
                then
                  Left $
                    "Number of fields in line "
                      <> show lNum
                      <> " does not match"
                      <> " expected length of "
                      <> show expectedLength
                      <> "! Actual length is "
                      <> show (length fields)
                      <> "!"
                else Right $ rows ++ [fields]
      )
      mRows
```

Сворачивает строки функцией parseRow с пустым начальным значением

Возвращает значение Left, если предыдущим значением было Left, в противном случае обрабатывает строку

Разбивает строку на поля

Возвращает Left с сообщением об ошибке, если количество полей в строке не совпадает с ожидаемым

Возвращает Right и добавляет поля к уже разобранным строкам

Собирая весь уже написанный код вместе, получаем функцию `parseCsv`, которая умеет разбирать простой `Text` на значения типа `Csv`! Для удобства можно написать две функции, которые обертывают различные параметры функцией `parseCsv`: одну для разбора с параметрами по умолчанию и заголовком, другую для разбора с параметрами по умолчанию и без заголовка. Для этого нужно изменить созданную ранее запись `defaultOptions` – мы зададим новое значение в записи в фигурных скобках после идентификатора записи. Код показан в листинге ниже.

Листинг 8.7 Вспомогательные функции для разбора CSV-таблиц

```
parseWithHeader :: T.Text -> Either String Csv
parseWithHeader =
  parseCsv (defaultOptions
➥ {cpoHeaderOption = WithHeader})

parseWithoutHeader :: T.Text -> Either String Csv
parseWithoutHeader = parseCsv defaultOptions
```

Разбирает CSV-файл с разделителями по умолчанию и заголовком

Разбирает CSV-файл с разделителями по умолчанию и без заголовка

Теперь можно попробовать разобрать какие-нибудь файлы! В репозитории кода к этой главе есть несколько CSV-файлов для тестирования функциональности.

```
ghci> file <- readFile "cities.csv"
ghci> fileText = T.pack file
ghci> Right csv = parseWithHeader fileText
ghci> csvHeader csv
Just ["LatD","LatM","LatS","NS","LonD","LonM","LonS","EW","City","State"]
ghci> numberOfRows csv
128
```

Мы проделали долгий путь и научились разбирать и проверять файлы, но пока что умеем *интерпретировать* значения в файлах только как `Text`. Для более сложных задач этого недостаточно. Далее мы хотим преобразовать эти текстовые значения во что-то, с чем удобнее работать.

8.3 Печать CSV-таблицы

Наша программа была бы неполна без возможности вывести CSV-таблицу на экран терминала или в файл. Для этого мы создадим модуль `Csv.Print`, который будет содержать всю функциональность, необходимую для печати и записи в файл. Сначала подумаем, как преобразовать значение типа `Csv` в формат CSV и записать его в файловую систему.

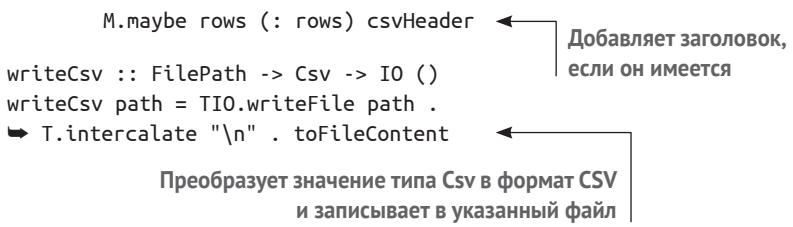
Для этого нам нужно преобразовать столбцы `Csv` в строки с помощью `transpose`. Далее мы должны преобразовать `DataField` в `Text` и объединить поля строк с помощью функции `intercalate` из модуля `Data.Text`. Она объединяет все элементы списка значений типа `Text`, располагая между каждой парой соседних элементов указанное значение того же типа. Мы можем воспользоваться этим, чтобы разделить поля запятыми, а строки – знаком новой строки! Затем встает вопрос, как записать `Text` в файл. Модуль `Data.Text.IO` предоставляет функции `readFile` и `writeFile` для работы с `Text` вместо `String`! Функции записи `Csv` в файл показаны в листинге ниже. Модуль `Data.Text.IO` импортируется как `TIO`.

Листинг 8.8 Функции для записи значения типа Csv в файл

```
toFileContent :: Csv -> [T.Text]
toFileContent Csv {..} =
  let rows = L.map (L.map dataFieldToText)
➥ $ L.transpose csvColumns
   in L.map (T.intercalate ",") $
```

Преобразует строки CSV в список текстовых строк

Разделяет столбцы запятыми



В этом определении мы видим еще один новый оператор: точку (`.`). Это оператор *композиции функций*. Говоря по-простому, он получает в качестве аргументов две функции и сначала применяет вторую, а потом первую. Определение очень простое:

```
ghci> (.) f g x = f (g x)
ghci> f = (+1) . (*3)
ghci> f 10
31
```

Это позволяет сцеплять функциональность, по-прежнему используя частичное применение функции. Определения при этом получаются очень короткими. Кроме того, этот оператор можно использовать для композиции функций, которые хранятся в структурах данных:

```
ghci> f = foldl (flip (.)) id
ghci> f [(+1), (*100), (/50)] 1
4.0
```

Поскольку показанную выше композицию функций иногда трудно сходу осмыслить, дадим более развернутое объяснение. В *функциональном* программировании *функции* являются полноправными объектами, что позволяет обращаться с ними как со значениями. Мы можем передавать их другим функциям, получать их от других функций, присваивать им имя, помещать их в структуры данных и вычислять новые функции на основе существующих. Часто упор делается на *композиции* функций. По правде говоря, все, что мы делаем, когда программируем декларативно, – это композиция функций, поскольку мы не определяем последовательность операций, как в императивном языке, а описываем, как результаты одних функций передаются другим функциям. Делается ли это с помощью let-привязок или операторов точки и доллара, не суть важно. Именно это мы и наблюдаем в предыдущем примере, где конечное действие ввода-вывода, заключающееся в записи `Csv` в файл, вычисляется на основе кучки других функций. Как это записывается – явно или с помощью свертки, – несущественно.

Обратите внимание, как удачно расширение языка `RecordWildCards` сочетается с операторами (`$` и `.`). Истинная суть Haskell (и функционального программирования вообще) – композиция функций,

и здесь мы имеем прекрасный пример композиции разной функциональности, которая в целом оказывается гораздо больше, чем сумма ее частей!

Наконец, мы хотим овладеть искусством *красивой печати*, т. е. отображения информации в удобном человеку виде. Разумеется, понятие о красоте субъективно. Поэтому мы хотим сделать эту часть главы преимущественно упражнением. Мы обсудим общую архитектуру, но фактическую реализацию оставим читателю.

Поговорим о том, как должен выглядеть *интерфейс* красивой печати. Проблема в том, что тип `Csv` параметрический. Он может содержать любой тип, в т. ч. такой, который напечатать нельзя. Поэтому мы должны ограничиться только типами, которые как минимум допускают преобразование в `Text`. Кроме того, мы хотели бы печатать сводную информацию о каждом столбце. В типе `Csv` это не моделируется явно. Чтобы смоделировать это дополнительное ограничение и провести более точное различие между обычными значениями и значениями, допускающими красивую печать, мы определим новый тип, показанный в листинге ниже.

Листинг 8.9 Функции для записи значений типа `Csv` в файл

```
data PrettyCsv = PrettyCsv
  { pcHeader :: Maybe [T.Text],
    pcColumns :: [[T.Text]],
    pcSummaries :: Maybe [T.Text]
  }
deriving (Eq, Show)
```

Определяет поле для красиво напечатанного заголовка

Определяет поле для красиво напечатанных столбцов

Определяет поле для факультативной сводной информации о каждом столбце

Отметим использование аббревиатуры типа (`pc`) в качестве префикса, чтобы избежать конфликтов с именами полей в типе `Csv`. С помощью этого нового типа мы можем преобразовать тип `Csv` в тип `PrettyCsv`, который затем использовать для добавления сводной информации и форматирования вывода, которым пользователи будут любоваться. Но красота – вещь субъективная, поэтому я не хочу высекать какой-то определенный стандарт в камне. Придумать алгоритмы красивой печати – дело *ваше*. Для этого вы можете воспользоваться шаблоном кода, показанным в следующем листинге.

Листинг 8.10 Функции для записи значений типа `Csv` в файл

```
fromCsv :: Csv -> PrettyCsv
fromCsv = ...

withSummaries ::
  PrettyCsv ->
```

Преобразует `Csv` в `PrettyCsv`

```
  [T.Text] ->
  Either String PrettyCsv
withSummaries = ...

pretty :: PrettyCsv -> String
pretty = T.unpack . prettyText

prettyText :: PrettyCsv -> T.Text
prettyText = ...
```

Безопасно добавляет сводную информацию в PrettyCsv, перезаписывая прежнюю и проверяя, что количество таких сводок совместимо с количеством столбцов

Преобразует PrettyCsv в удобный для восприятия Text

В исходном коде проекта имеются возможные реализации этих функций, но это не мешает вам попробовать написать собственную.

8.3.1 Операции с CSV-таблицами

До сих пор мы реализовали несколько полезных операций с CSV-таблицами и научились показывать их на экране с дополнительной сводной информацией. Пришло время наполнить эти сводки столбцов чем-то полезным. Для этого мы хотели бы *сворачивать* наши значения типа `Csv` по столбцам, чтобы получить из совокупности значений в столбце одно *сводное* значение. Это можно сделать, свернув их по отдельности и отобразив результат. См. листинг ниже.

Листинг 8.11 Свертка столбцов CSV-таблиц в одно значение

```
foldCsv :: (DataField -> b -> b) -> b -> Csv -> [b]
foldCsv f z (Csv {csvColumns}) = map (foldr f z) csvColumns
```

Отображает свертки столбцов и возвращает результат

Жаль, но мы не можем использовать эту свертку для создания экземпляра `Foldable Csv`, потому что `Csv` имеет род `*`, а не `* -> *`.

Еще одна операция, полезная при работе с этими данными, – поиск по строкам таблицы. По аналогии с функцией `filter` для списков, мы можем определить функцию фильтрации для `Csv`. Для этого нужно транспонировать столбцы в строки, отфильтровать их, а затем выполнить обратное транспонирование. Это показано в листинге ниже.

Листинг 8.12 Функция фильтрации CSV-таблиц

Извлекает столбцы из аргумента Csv

Транспонирует столбцы в строки

```
filterCsv :: (DataField -> Bool) -> Csv -> Csv
filterCsv p csv@(Csv {csvColumns}) =
  let rows = L.transpose csvColumns
      filtered = L.filter (any p) rows
   in csv {csvColumns = L.transpose filtered}
```

Фильтрует строки по предикату

Строит профильтрованный Csv из оставшихся после фильтрации строк

`any` – функция для списков, которая возвращает `True`, если заданный предикат истинен хотя бы для одного элемента списка. В нашем коде мы принимаем все строки, для которых хотя бы одно поле удовлетворяет предикату.

В этом определении мы видим еще один синтаксический трюк, припасенный Haskell. Конструкция `@` называется *привязкой образца*; она связывает имя с частью образца. В нашей функции второй аргумент, с которым производится сопоставление, связывается с именем `csv`; таким образом, «несопоставленное» значение становится доступно в определении функции наряду с полем `csvColumns`.

Теперь, построив две очень общие функции, мы можем определить конкретную функциональность. Для значений типа `Csv` мы можем с помощью `foldCsv` подсчитать все непустые значения, поскольку пустые обозначаются как `NullValue`. Аналогично мы можем подсчитать точное число вхождений конкретного значения `DataField`. Наконец, мы хотим определить фильтр, которому удовлетворяют все элементы `Csv`, содержащие некоторый `Text`. Чтобы это работало, полиморфный тип CSV должен иметь экземпляр `ToText`. Реализации этих функций показаны в следующем листинге.

Листинг 8.13 Операции с CSV-таблицами на основе свертки и фильтрации

```
countNonEmpty :: Csv -> [Int]
countNonEmpty = foldCsv f 0
  where
    f NullValue acc = acc
    f _ acc = acc + 1

countOccurences :: DataField -> Csv -> [Int]
countOccurences df =
  foldCsv (\x acc -> if x == df then acc + 1 else acc) 0

searchText :: T.Text -> Csv -> Csv
searchText t = filterCsv (\f -> dataFieldToText f `contains` t)
  where
    contains = flip T.isInfixOf
```

Подсчитывает все непустые значения, сворачивая по каждому столбцу и увеличивая аккумулятор на 1 всякий раз, как значение не равно NullValue

Ищет поля, содержащие указанный Text

Подсчитывает число полей в столбце, имеющих указанное значение

Эти функции теперь послужат основой для CSV-операций! Мы можем поместить их в новый модуль, `Csv.Operations`, и реэкспортировать его из модуля `Csv`.

Упражнение. Сводки и фильтры

Разумеется, существует множество других сводок и фильтров, только и ждущих, чтобы их реализовали! Например, можно было бы создать сводку, которая подсчитывает, сколько раз указанный `Text` встречается в столбце или сколько в столбце числовых значений, больших или меньших заданной величины.

Особо отважные могут испытать на деле пакет `regex` и фильтровать `Csv` по регулярному выражению. Документация по пакету находится по адресу https://hackage.haskell.org/package/regex.

8.4 Простой парсер командной строки

В этом проекте мы реализовали много средств для работы с CSV-файлами. Мы умеем разбирать их, вырезать части, собирать из частей, отображать и записывать в файл или печатать на терминале. Но пока что все это разрозненные фрагменты функциональности, которыми невозможно воспользоваться. Теперь посмотрим, как превратить их в полноценный инструмент.

Для этого снова обратимся к разбору аргументов командной строки, но на этот раз воспользуемся вновь приобретенными навыками для написания простого парсера, который будет сообщать, заданы в командной строке определенные параметры или нет.

Сначала обернем функцию `getArgs` из модуля `System.Environment`, так чтобы она возвращала `Text` вместо `String`. Взглянем на ее тип:

```
getArgs :: IO [String]
```

Мы видим, что должны отобразить список, возвращенный действием ввода-вывода, и упаковать все элементы типа `String`, приведя их к типу `Text`. Мы уже знаем, как использовать для этой цели `do`-нотацию, но существует более простой способ: тип `IO` обладает экземпляром `Functor`! Это значит, что мы можем отобразить внутренний тип и значение действия ввода-вывода, воспользовавшись `fmap` и `<$>`. В данном случае мы хотим отобразить функцию в действие ввода-вывода, которое упаковывает список `String`. Код выглядит так:

```
getArguments :: IO [T.Text]
getArguments = map T.pack <$> getArgs
```

Результат `getArgs` отображается с помощью `T.pack`. Напомним, что оператор `<$>` – это инфиксная форма `fmap`. Он заставляет `map T.pack` воздействовать непосредственно на аргументы. Однако из типа `fmap` ясно следует, что результат при этом сам является действием вво-

да-вывода. Это общее правило. Когда `fmap` применяется к спискам, результатом тоже является список, а когда она применяется к `Maybe`, снова получается `Maybe`.

Исходя из этого, мы можем создать дополнительные действия ввода-вывода для разбора аргументов. Мы оставим их реализацию читателю, хотя в репозитории проекта имеется полный исходный код. Здесь же мы обсудим, как следует структурировать и форматировать аргументы.

Все аргументы нашей программы должны начинаться с `--`. Аргумент, не являющийся булевым флагом, должен иметь вид `--argument-name=value`, где `value` может иметь зависящую от задачи форму. Существуют следующие типы аргументов:

- `Bool` – равен `True`, если аргумент присутствует, и `False` в противном случае (например, `--argument-name`);
- `Char` – представляет введенную литеру `Char`, но только если она одна (например, `--argument-name=,`, но нет `--argument-name=abc`);
- `Text` – представляет введенный `Text`;
- `Interval` – интервал, представленный двумя целыми числами `Int` через запятую (например, `--argument-name=1,5`).

Соответствующие функции, которые должны находиться в своем собственном модуле, – это:

```
getBool :: T.Text -> IO Bool
getChar :: T.Text -> IO (Maybe Char)
getText :: T.Text -> IO (Maybe T.Text)
getInterval :: T.Text -> IO (Maybe (Int, Int))
```

В наших примерах эти функции реализованы в модуле `Util.Arguments`, на который мы ссылаемся в коде по квалифицированному имени `Args`. Реализация этих функций поручается вам (хотя если вам неохота, можете скопировать их из репозитория). Полезно будет написать общую вспомогательную функцию, которая читает из аргументов значения, поскольку все аргументы имеют вид `--key=value`.

> **ПРИМЕЧАНИЕ** Конечно, немногие реализуют парсеры аргументов самостоятельно, поскольку на эту тему существует немало библиотек. Одна из них – `optparse-applicative`, которую вы могли бы использовать в этом проекте. Мы будем рассматривать ее в последующих главах.

8.4.1 Поддержка флагов и более сложных аргументов

Покончив с разбором аргументов, можно подумать о том, как собрать воедино всю разработанную функциональность. Для этого мы применим средства работы с аргументами, чтобы определить

поведение парсера, способы преобразования данных и способ их печати или записи в файл. Мы собираемся поддержать следующие аргументы:

- `--in=<path>` – использовать в качестве основного CSV-файл с путем `<path>`;
- `--append=<path>` – использовать CSV-файл с путем `<path>` в качестве дополнительного, который дописывается справа к данным из основного файла;
- `--field-separator=<char>` – разделитель полей в основном и дополнительном файлах;
- `--with-header` – парсер должен предполагать наличие заголовков в основном и дополнительном файлах;
- `--slice=<x>,<y>` – разрезать результирующий CSV-файл в местах, определяемых заданными индексами;
- `--search=<term>` – оставить только строки, содержащие указанный терм;
- `--count-non-empty` – добавить в красивую выдачу сводную информацию о количестве непустых полей в каждом столбце;
- `--no-pretty` – подавить красивую выдачу CSV, которая по умолчанию включена;
- `--out=<path>` – путь к результирующему CSV-файлу (после всех преобразований); если равен `-`, то вывод производится на `stdout`.

Поскольку мы хотим разрешить чтение из нескольких CSV-файлов и использовать дополнительные флаги, то всю функциональность чтения файла имеет смысл поместить в отдельное действие ввода-вывода. Оно создает параметры разбора на основе заданных аргументов ввода-вывода и соответственно разбирает файл. Код этого действия приведен в листинге ниже.

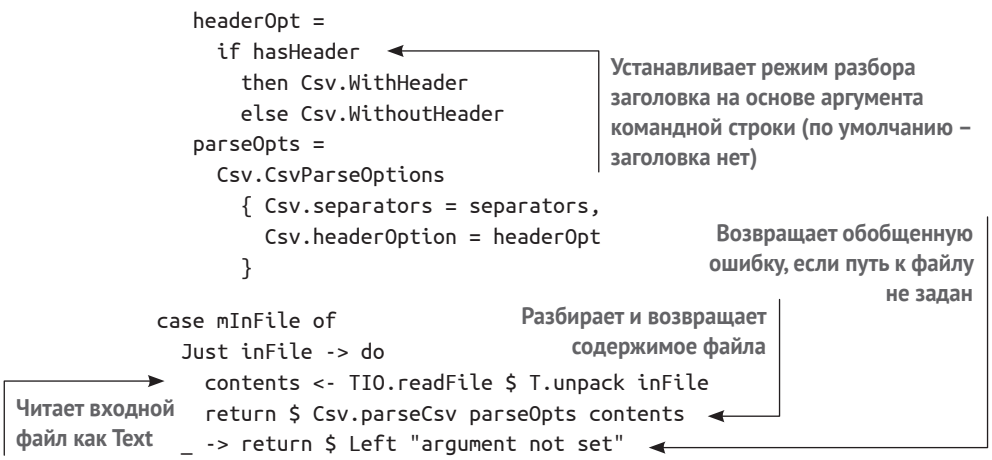
Листинг 8.14 Действие чтения файла с параметрами разбора, определяемыми аргументами командной строки

```
parseInFile :: T.Text -> IO (Either String (Csv.Csv T.Text))
parseInFile key = do
  mInFile <- Args.getText key
  mFieldSep <- Args.getChar "field-separator"
  hasHeader <- Args.getBool "with-header"

  let separators =
        Csv.defaultSeparators
          { Csv.fieldSeparator =
              M.fromMaybe
                (Csv.fieldSeparator Csv.defaultSeparators)
                mFieldSep
          }
```

Читает путь и параметры разбора, заданные аргументами командной строки

Переустанавливает разделитель полей, если он задан аргументом, иначе использует значение по умолчанию



Имея эту функцию, можно задуматься и о написании действия `main`. Ясно, что сначала нужно проверить, был ли задан аргумент `in`. Если нет, то мы просто печатаем сообщение об ошибке и завершаем программу:

```
main :: IO ()
main = do
  mCsv <- parseInFile "in"
  case mCsv of
    Left _ -> putStrLn "no input file given (do so with --in=...)"
    Right csv -> do
    ...
```

Следующий шаг потруднее. Нам предстоит выполнить преобразования разобранного значения типа `Csv`. Как к этому подойти? Мы могли бы проверить каждый аргумент и по результатам проверок произвести преобразования, но можно поступить более элегантно. Поскольку функции – это обычные значения, мы можем работать с ними, как с любыми другими данными. В частности, их можно собирать (помещать в список) и вычислять по ним новые значения (в данном случае – функции). Эту стратегию мы и примем: отобразим найденные аргументы на операции, а затем объединим их в единую операцию преобразования, которую и применим к значению типа `Csv`. Код показан в следующем листинге.

Листинг 8.15 Чтение необязательных аргументов и вывод из них поведения

```
...
Right csv -> do
  mAppend <- eitherToMaybe <$> parseInFile "append"
  mSliceInterval <- Args.getInterval "slice"
  mSearch <- Args.getText "search"
```

Производит поиск аргументов, дающий значения типа Maybe

```
  let mAppendOp = fmap (flip (<>)) mAppend
      mSliceOp = fmap (uncurry slice) mSliceInterval
      mSearchOp = fmap Csv.searchText mSearch
      transformOp =
        foldl
          (\t mOp -> (M.fromMaybe id mOp) . t)
          id
          [mAppendOp, mSliceOp, mSearchOp]
      dataCsv = Csv.toDataCsv $ transformOp csv
...

eitherToMaybe :: Either b a -> Maybe a
eitherToMaybe (Left _) = Nothing
eitherToMaybe (Right x) = Just x
```

Отображает значения аргументов типа Maybe на функции, обернутые Maybe

Строит композицию функций по порядку, подставляя id, если функции не существует

Применяет построенную функцию к входному Csv и преобразует его в DataCsv

Встречающаяся в коде функция `uncurry` принимает бинарную функцию и создает унарную функцию, принимающую кортеж из двух аргументов и дающую тот же результат. Двойственная ей функция называется `curry`. Здесь мы используем `uncurry`, чтобы заставить функцию `slice` принимать интервальный кортеж, хранящийся в `mSliceInterval`.

> **ПРИМЕЧАНИЕ** Имя `curry` происходит от концепции *карирования*, согласно которой любую функцию с несколькими аргументами можно переписать в виде последовательности унарных функций. Вспомните пример из второй главы, где мы изучали, как сконструировать бинарную функцию из двух анонимных унарных функций (`f = \x → \y → …`). Именно эта идея и позволяет нам записать любую функцию с конечным числом аргументов в виде последовательности унарных функций. Она была введена в обиход математиком Хаскеллом Карри, в честь которого была названа не только она, но и язык программирования, изучаемый в этой книге.

Итак, мы раскрыли функциональность нашей программы посредством аргументов командной строки, и теперь осталось только позаботиться о представлении данных внешнему миру. Мы хотим, чтобы пользователь мог записывать CSV-таблицу в файл или на `stdout` для последующих манипуляций с данными. Также у пользователя должна быть возможность получить красиво отформатированные данные. По счастью, для всего этого нам не придется изучать новых заковыристых механизмов. Все сводится к сопоставлению с образцами и незначительной помощи со стороны функции, которая предоставляет синтаксически изящный способ факультативно выполнить действие ввода-вывода:

```
when :: Bool -> IO () -> IO ()
when True act = act
when False _ = return ()
```

```
unless :: Bool -> IO () -> IO ()
unless b = when (not b)
```

С помощью `unless` мы можем записать сравнительно длинное выражение, не обертывая его `case` или `if`. Код показан в листинге ниже.

Листинг 8.16 Чтение факультативных аргументов и определение по ним поведения

```
mOut <- Args.getText "out"
case mOut of
  Just "-" -> TIO.putStrLn $ Csv.toFileContent dataCsv
  Just fp -> Csv.writeCsv (T.unpack fp) dataCsv
  _ -> do
    countNonEmpty <- Args.getBool "count-non-empty"
    let mSummary =
          if countNonEmpty
            then Just $ Csv.countNonEmpty dataCsv
            else Nothing
    noPrettyOut <- Args.getBool "no-pretty"
    unless noPrettyOut $
      TIO.putStrLn $
        Csv.prettyText $
          maybe
            id
            (flip Csv.withSummaries)
            mSummary
            (Csv.fromCsv dataCsv)
```

Печатает представление файла в виде CSV-таблицы на stdout

Записывает CSV-таблицу в файл

Читает аргумент из командной строки

Вычисляет факультативную сводку, если задан аргумент count-non-empty

Останавливает выполнение, если задан аргумент no-pretty

Печатает красивое представление CSV-таблицы

Либо оставляет CSV-таблицу без изменения, если сводка не вычислялась, либо добавляет сводку в красивое представление

На этом мы завершаем реализацию. Мы написали библиотеку, умеющую разбирать CSV-файлы, строя модель данных, на основе которой можно проектировать различные алгоритмы. Мы познакомились с такими концепциями работы с данными, как классы типов `Semigroup` и `Monoid`, и даже реализовали собственные классы для вырезания данных и преобразования в тип `Text` и обратно. Затем мы раскрыли функциональность внешнему миру, добавив возможность конфигурирования.

Упражнение. Бесчисленные опции

Теперь ваша очередь. Мы заложили основу для инструмента, который умеет читать, преобразовывать и записывать CSV-файлы, но раскрыли не весь его потенциал. Займитесь развитием программы, реализуйте дополнительные фильтры, стратегии поиска и способы вывода данных, прочитанных из CSV-файлов. Вот несколько идей для начала:

- соединение CSV-таблиц;
- фильтрация по числовым значениям (`> n` и `< n`);
- настраиваемое форматирование вывода в файл;
- более одного основного и дополнительного файла;
- поиск и замена строк;
- экспорт CSV-таблицы в команды SQL CREATE;
- поддержка дополнительных типов данных, например чисел с плавающей точкой и булевых значений;
- автоматическое определение разделителей на основе анализа данных.

Более отважные читатели могут попробовать свои силы в реализации средств, требующих больше работы:

- преобразование CSV-файлов в формат JSON и наоборот;
  - для этого вам может пригодиться пакет `aeson`;
- интерактивный режим с командным интерфейсом для исследования больших CSV-таблиц;
  - ознакомьтесь с пакетом `brick`, позволяющим строить командные интерфейсы пользователя;
  - в нем имеется тип `Table`, который можно использовать для вывода таблиц на экран.

Наконец, мы хотим *установить* программу на свой компьютер! Конечно, этот шаг необязателен, но если мы написали полезную вещь, то пренебрегать им не стоит. Команда `stack install` собирает программу и устанавливает ее в локальный каталог. Записав путь к этому каталогу в переменную окружения `PATH`, вы сможете пользоваться программой! В нашем случае исполняемый файл называется `csview`.

Резюме

- Функцию `readMaybe` из модуля `Text.Read` можно использовать для безопасного разбора числовых значений.
- Свертка используется для редуцирования сложных данных в одно значение.
- `foldl` ассоциирует свою функцию слева, а `foldr` – справа.
- Функторы используются для отображения данных, содержащихся в типе.
- Роды соотносятся с типами так же, как типы со значениями.
- `(<$>)` – инфиксная версия `fmap`.
- Оператор `(.)` используется для композиции функций: `(.) f g x = f (g x)`.

9 QuickCheck и случайные тесты

Краткое содержание этой главы:

- работа со случайными значениями;
- построение простого каркаса для тестирования свойств;
- использование популярного каркаса тестирования QuickCheck;
- адаптация генераторов случайных значений к специальным тестам;
- включение комплектов тестов в программные проекты.

В предыдущих главах мы писали на Haskell код для различных приложений. Но не задавались вопросом, а правилен ли написанный нами код. То ли он делает, что должен? Верные ли результаты порождают наши алгоритмы? В программной инженерии ответ на этот вопрос часто дает *тестирование*, и именно этим мы займемся в настоящей главе. Для знакомства с тестированием мы возьмем популярную библиотеку *QuickCheck* и изучим тестирование свойств в контексте Haskell; нас будет интересовать формализация свойств корректности для чистого функционального кода.

Мы начнем эту главу знакомством с концепцией *тестирования свойств*, после чего реализуем собственный каркас тестирования.

По ходу дела мы узнаем, как работать со случайными значениями в Haskell. После этого будет приведено введение в QuickCheck. В этой главе рассматривается, как писать свойства, как использовать класс типов `Arbitrary` и подстраивать поведение тестирования к своим потребностям. В завершение мы покажем, как включать комплекты тестов QuickCheck в проект, созданный программой Stack.

9.1 Как тестировать

Тестирование возможно на разных уровнях. При *комплексном тестировании* рассматривается взаимодействие различных компонентов программы, при *системном тестировании* проверяется, выполнены ли все требования к готовой системе, а при *автономном тестировании* (unit testing) внимание уделяется меньшим «единицам» кода, поведение которых тестируется по отдельности. Это позволяет писать независимые тесты, проверяющие корректность функционирования небольших компонентов. Графически это показано на рис. 9.1.



Рис. 9.1 Пирамида тестирования

Достоверность интеграционных и системных тестов зависит от корректности, уже установленной автономными тестами. Если мы не уверены, что самые мелкие компоненты системы работают, как должны, то и нет смысла тестировать более крупные компоненты. Таким образом, степень доверия, внушаемая тестами, зависит от полноты и тщательности автономных тестов на самом низком уровне. В этой главе мы сосредоточимся на автономном тестировании одиночных функций.

Однако в связи с этим возникает вопрос, *как* тестировать код. Самый простой способ – предъявлять примеры, т. е. задавать какие-то входные значения и проверять, что после выполнения кода получился ожидаемый результат. Но у такого способа есть много недостатков:

- каждый тест нужно готовить вручную;
- покрытие тестами зависит от количества написанных тестов;
- пограничные случаи должны быть выявлены и описаны вручную (но иногда об этом забывают).

Все это чревато ошибками. Если бы могли представить себе полный набор тестов во время написания кода, то и тестировать было бы ни к чему. В конечном итоге мы пишем тесты для функциональности, которая, как мы уже знаем, работает правильно (особенно если проектировщик тестов эту функциональность и разрабатывал). То есть тесты – это страховка от регрессии. А вот *обнаруживаются* ошибки при тестировании редко.

Тут-то и вступает в игру тестирование свойств. Если мы формализуем правильность функциональности в виде математического свойства, которое должно быть истинным на входе и на выходе, то можем протестировать свой код путем случайного генерирования входов и проверки выполнения свойства для каждого из них. Тогда у нас будет больше шансов обнаружить ошибки, т. к. мы не обязаны сами определять тестовые входные данные.

9.1.1 Тестирование свойств

Для начала подумаем о том, что мы имеем в виду, говоря о свойствах. *Свойство* – эта характеристика данных, которую можно вычислить и проверить. Проще говоря, любую функцию, которая принимает какой-то аргумент и возвращает булево значение, можно считать свойством. Свойства, используемые для тестирования функциональности программы, могут быть сколь угодно сложными. Это может быть простое сравнение значений базовых типов, проверка того, что данные подчиняются некоторому набору правил, или комбинирование имеющихся свойств для построения новых. Тестиро-

вание на основе равенства, когда выход кода проверяется на равенство ожидаемому значению, – очень простая форма свойств.

Изучение этой техники тестирования мы начнем с примера. Создадим новый проект командой `stack`. Допустим, что требуется протестировать некую функцию сортировки. Для такой функции свойством может быть тот факт, что на выходе всегда получается отсортированный список. Нетрудно написать функцию, которая проверяет, что список отсортирован (в порядке возрастания), а также функцию, которая проверяет, что заданная функция порождает отсортированный список по заданному входу. Обе функции показаны в листинге ниже.

Листинг 9.1 Предикаты отсортированности списков

```
sorted :: Ord a => [a] -> Bool
sorted [] = True
sorted [_] = True
sorted (x : y : xs) = x <= y && sorted (y : xs)

sorts :: Ord a => ([a] -> [a]) -> [a] -> Bool
sorts f input = sorted $ f input
```

Возвращает True, если в списке меньше двух элементов

Проверяет, что первые два элемента отсортированы, а затем рекурсивно проверяет, что остаток списка также отсортирован

Возвращает True, если заданная функция для заданного входа порождает отсортированный выход

Теперь мы можем воспользоваться свойством `sorts` для тестирования функций на входных данных. `sorted` – это свойство, а `sorts` проверяет, обладает ли результат функции на заданных входных данных свойством `sorted`. Давайте протестируем функцию `sort` из модуля `Data.List`.

```
ghci> import Data.List (sort)
ghci> sort `sorts` [5,4,3,2,1 :: Int]
True
ghci> sort `sorts` [1,3,2,5,4 :: Int]
True
ghci> sort `sorts` ([] :: [Int])
True
```

Похоже, работает правильно! Однако при построении комплекта тестов мы не хотели бы выписывать такие тесты самостоятельно. Мы хотим рандомизировать вход, но само понятие случайности требует, чтобы мы покинули мир чистых функций и создали что-то другое.

9.1.2 Генерирование случайных значений

Поскольку тестирование свойств – рандомизированный метод тестирования, мы потратим довольно много времени на генерирование

случайных входных данных. Важно поддержать не только широкое разнообразие типов, для которых имеются генераторы случайных значений, но и их модификацию. Построение новых генераторов для сложных типов данных должно быть простым, а хороший каркас облегчит эту задачу.

На данный момент нас интересует создание случайных списков для тестирования нашего свойства применительно к функции `sort`. Со случайными значениями мы пока не имели дела, а в языках, подобных Haskell, это вещь нетривиальная, потому что для вычисления новых случайных чисел генератор должен хранить состояние. В языках с побочными эффектами реализовать случайные значения просто (с помощью глобального управляемого состояния), но в Haskell все по-другому. Выход функции определяется *только* ее входом!

Ну и как же нам обеспечить случайность? Чтобы получить представление об этом, сначала рассмотрим простой пример. Пусть требуется написать функцию, генерирующую случайное значение. Мы знаем, что для возврата случайного значения ей на вход нужно подать какое-то случайное состояние; однако потом должно быть возвращено новое случайное состояние. В противном случае функция будет раз за разом вызываться с одним и тем же случайным состоянием. Если состояние модифицируется чистой функцией (например, выполняется псевдослучайное преобразование), то потом можно будет использовать новое состояние. Подавая это состояние обратно на вход функции, мы сможем создать последовательность случайных значений. Это показано в листинге ниже.

Листинг 9.2 Функции, явно работающие со случайным состоянием

```
newtype RandomState = RandomState Int
➥ deriving (Eq, Show)

randomInt :: RandomState -> (Int, RandomState)
randomInt (RandomState rs) = (newRs, RandomState newRs)
where
  newRs = (1103515245 * rs + 12345) `mod` (2 ^ 31)

randomIntList :: RandomState -> Int -> [Int]
randomIntList rs n
  | n <= 0 = []
  | otherwise =
    let (v, rs') = randomInt rs
     in v : randomIntList rs' (n - 1)
```

Определяет тип случайного состояния как Int

Возвращает случайное значение и новое случайное состояние

Вычисляет новое случайное состояние

Рекурсивно строит список случайных значений по заданному случайному состоянию

Функция `randomInt` использует *линейный конгруэнтный генератор* для вычисления псевдослучайных значений. Подавая состояние обратно на вход функции, мы можем создать поток случайных значений. Именно эта идея применяется в функции `randomIntList` для создания списка случайных целых чисел:

```
ghci> randomIntList (RandomState 100) 5
[829870797,1533044610,1478614675,1357823696,413847241]
```

Это один из способов генерирования случайных чисел, но недостаточно общий. Нам нужна возможность описывать случайные значения, не ограничивающиеся числами. И кроме того, мы не должны реализовать ничего из этого сами, поскольку все уже написано. В нашем распоряжении ассортимент функций из модуля `System.Random` для получения случайных значений различных типов.

Для использования этого модуля мы должны добавить `random` в состав зависимостей нашего пакета. Секция зависимостей должна выглядеть как-то так:

```
dependencies:
- random >= 1.2.1.1
```

Модуль `System.Random` предоставляет самые разные функции, типы и классы типов для генерирования случайных значений. Однако некоторые его части объявлены нерекомендуемыми, так что использовать их не следует. Самый современный (на момент написания книги) подход представлен в модуле `System.Random.Stateful`, который дает доступ ко многим типам генераторов и способам взаимодействия с ними. Но нас будет интересовать только `StdGen` – типичный генератор псевдослучайных чисел.

9.1.3 Random и Uniform

Генератор `StdGen` по способу использования очень похож на созданный нами `RandomState`:

```
ghci> import System.Random
ghci> import System.Random.Stateful
ghci> g = mkStdGen 100
ghci> g
StdGen {unStdGen = SMGen 16626775891238333538 2532601429470541125}
ghci> random g :: (Int, StdGen)
(9216477508314497915,StdGen {unStdGen = SMGen 712633246999323047
➥ 2532601429470541125})
```

Как видим, мы можем создать `StdGen` с начальным значением (в данном случае `100`) и использовать функцию `random` для генерирования кортежа, состоящего из случайного числа и модифицированного `StdGen`. `random` – одна из четырех важных функций, получающих случайные значения от генератора:

- `random :: (Random a, RandomGen g) => g -> (a, g);`
- `randomR :: (Random a, RandomGen g) => (a, a) -> g -> (a, g);`

- `uniform :: (Uniform a, RandomGen g) => g -> (a, g)`;
- `uniformR :: (UniformRange a, RandomGen g) => (a, a) -> g -> (a, g)`.

Если `random` генерирует случайные значения с *неизвестным* распределением, то `uniform` генерирует равномерно распределенные случайные значения заданного типа. Именно поэтому классы типов для обеих функций различны. У обеих функций имеются также версии с именами, заканчивающимися на `R`, которые генерируют случайные значения в заданном диапазоне.

Поскольку и `Random`, и `Uniform` имеют много экземпляров для различных значений, мы можем использовать этот факт для написания полиморфных функций `random`. Мы не обязаны писать их конкретно для типа `Int` или вообще числовых значений. Поэтому ничто не мешает обобщить наш список случайных значений. Кроме того, генератор можно использовать для порождения списка случайных значений случайной же длины (но все же ограниченной сверху). Это показано в следующем листинге.

Листинг 9.3 Функции, порождающие случайные списки случайного размера

```
randomListN :: (Random a) => StdGen -> Int -> ([a], StdGen)
randomListN gen n
  | n <= 0 = ([], gen)
  | otherwise =
    let (v, gen') = random gen
        (xs, gen'') = randomListN gen' (n - 1)
     in (v : xs, gen'')

randomList :: (Random a) => StdGen -> Int -> ([a], StdGen)
randomList gen maxVal = randomListN gen' n
  where
    (n, gen') = uniformR (0, maxVal) gen

randomList' :: (Random a) => StdGen -> ([a], StdGen)
randomList' = flip randomList 100
```

Генерирует случайное значение и модифицирует генератор

Рекурсивно строит список случайных значений

Возвращает случайный список случайного размера от 0 до 100

Генерирует случайное целое число от 0 до заданной максимальной величины и модифицирует генератор

Генерирует случайный список случайного размера с модификацией генератора

Как и в случае `RandomState`, мы передаем генератор рекурсивным вызовам функции для порождения новых случайных значений. В конце возвращается модифицированный генератор, чтобы его можно было использовать дальше; однако каждый раз передавать и возвращать генераторы несколько громоздко. Хотелось бы сделать вещь, типичную для большинства языков программирования: завести глобальный генератор случайных значений.

9.1.4 Использование глобального генератора случайных значений

Кажется, что в языке, подобном Haskell, случайные значения с состоянием невозможны, но, по счастью, модуль `System.Random` предлагает такую функциональность. Ниже приведены упрощенные сигнатуры двух интересующих нас действий:

- `getStdGen :: IO StdGen`;
- `setStdGen :: StdGen -> IO ()`.

Они позволяют читать и изменять глобальный `StdGen` в действиях ввода-вывода. Глобальный `StdGen` уже инициализирован в начале работы программы. Однако здесь нас поджидает ловушка!

```
ghci> g <- getStdGen
ghci> g
StdGen {unStdGen = SMGen 9118772029923285118 15072976524409580889}
ghci> random g :: (Int, StdGen)
(7972809293454871112,StdGen {unStdGen = SMGen 5745004480623314391
➥ 15072976524409580889})
ghci> getStdGen
StdGen {unStdGen = SMGen 9118772029923285118
➥ 15072976524409580889}
```

`getStdGen` возвращает не ссылку на глобальный генератор, а копию. Поэтому после модификации не забывайте записать измененный генератор назад в глобальное состояние.

> **ПРИМЕЧАНИЕ** Глобальные ресурсы можно реализовать в действиях ввода-вывода с помощью типа `IORef`, как в случае `getStdGen` и `setStdGen`. Это изменяемые ссылки, которые можно использовать для чтения, сохранения и модификации глобально доступных данных (в случае если переменная типа `IORef` глобальна). Однако реализация небезопасна относительно конкурентного выполнения, так что использовать `IORef` не рекомендуется, и упомянут он только для информации. Вместо этого следует использовать `applyAtomicGen` и `globalStdGen`, как будет показано ниже в этой главе.

С помощью глобального генератора мы можем написать действие ввода-вывода, которое *не* нуждается в передаче `StdGen` в качестве аргумента, а вместо этого самостоятельно получает глобальный генератор, применяет функцию `randomList'`, записывает модифицированный генератор обратно и только потом возвращает случайный список. Код показан ниже.

Листинг 9.4 Действие ввода-вывода, генерирующее случайные списки

```
randomListIO :: (Random a) => IO [a]
randomListIO = do
  g <- getStdGen
  let (xs, g') = randomList' g
  setStdGen g'
  return xs
```

Получает глобальный stdGen

Генерирует случайный список и модифицированный stdGen

Возвращает случайный список

Сохраняет модифицированный stdGen в глобальном генераторе

Теперь можно использовать это действие для генерирования случайных значений разных типов, не заботясь ни о передаче, ни об инициализации генератора:

```
ghci> randomListIO :: IO [Bool]
[True,False,True,True,True,True,True,False,False,False,True,True,True,
➥ True,False,False,True,True,False,False,False,True,False,False,False,
➥ True,False,False,True,True,False,False,False,True,True,False,True,
➥ True,True,False,True,True,True,False,False,False,True,True,False,
➥ False,True,False,False,False,False,False,False,False,False,False,
➥ False,False,False,False,False,False,True,False,True,False,False,True,
➥ True,True,True,False,True,True,False,False,False,True,True,False,
➥ False,True,False]
ghci> randomListIO :: IO [Int]
[3691255068706279388,-2800187626527368700,3917553492683069837,
➥ 4227660806806667542,3268527085094962027,-5741538607687626546,
➥ 2150401921944371087,-94788782940232908,-7272037394668750655,
➥ 6615906112287910112,-7876828434837980022,-3967934342698301492,
➥ -8761894913785559922,3300716267348171691,2746716191340705260,
➥ 501540442220827077,7580997570714960595,4369396272232835442,
➥ 4464186059485273848,-1170916352399186521,-9175236237513240717,
➥ -3497815391797361407,-8229136869197540017,7239141001949992062,
➥ 6119772250924788401,-4616451576842552626,-3121301812412732082,
➥ 1811224167447568155,1914103012390535081]
```

Однако в написанном нами коде используются нерекомендованные функции, которые не следует вызывать в конкурентной программе. Они упоминаются во многих онлайновых пособиях и старых источниках, но модуль `System.Random.Stateful` предлагает новый, гораздо более безопасный интерфейс. Он предполагает использование типа `AtomicGenM` и генератора `globalStdGen`, который, к счастью, имеет тип `AtomicGenM StdGen`. Чтобы использовать вместе с ним функцию `random` или ей подобную, нам понадобится функция `applyAtomicGen`:

```
ghci> applyAtomicGen random globalStdGen :: IO Int
-8749052575918541620
```

```
ghci> applyAtomicGen (uniformR (0, 100)) globalStdGen :: IO Float
60.164364
ghci> applyAtomicGen randomList' globalStdGen :: IO [Bool]
[True,False,False,True,True,False,False,False,True,True,True,True,True,
➥ False,True,False,False,False,False,True,True,True,False,True,True,
➥ False,False,False,True,False,False,True,False,True,True,False,True,
➥ True,True,False,False,False,False,True,True,True,False,False,False,
➥ True,False,False,False,True,False,False,True,True,True]
```

Эта функция выполняет по существу то же действие, что наша `randomListIO`. Она читает глобальный генератор, генерирует значение, модифицирует его и записывает назад в глобальную ссылку. Однако все это она делает *атомарно*, отсюда и название. Поэтому этот генератор безопасно использовать в конкурентном контексте. Так как `globalStdGen` – константа, мы можем написать небольшую вспомогательную функцию, чтобы генерировать случайные значения было удобнее. Она показана в следующем листинге.

Листинг 9.5 Действие ввода-вывода, которое применяет функцию к глобальному генератору псевдослучайных значений

```
applyGlobalStdGen :: (StdGen -> (a, StdGen)) -> IO a
applyGlobalStdGen f = applyAtomicGen f globalStdGen
```

Применяет функцию, отображающую StdGen в значение, а новый StdGen – в глобальный StdGen программы, и возвращает сгенерированное случайное значение

Теперь мы можем легко (и безопасно) генерировать случайные значения в действиях ввода-вывода, что позволяет написать простой тест свойств.

9.1.5 Простой тест свойств

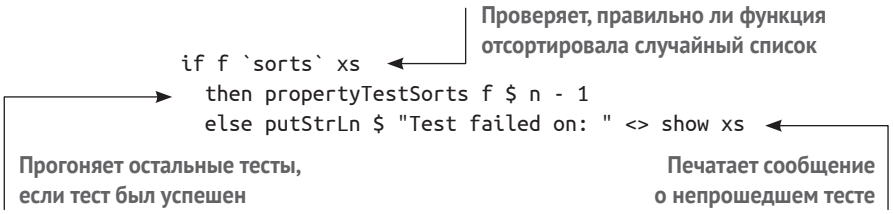
Для построения теста свойств нам осталось только собрать все части в действие ввода-вывода, которое генерирует определенное число тестов и прогоняет на них заданную функцию, печатая вход, если будет найден контрпример. Это показано в следующем листинге.

Листинг 9.6 Пример теста свойств, который проверяет, что функция правильно сортирует входные данные

```
propertyTestSorts :: ([Int] -> [Int]) -> Int -> IO ()
propertyTestSorts f n
  | n <= 0 = putStrLn "Tests successful!"
  | otherwise = do
    xs <- applyGlobalStdGen randomList'
```

Печатает сообщение об успехе, если больше тестов не осталось

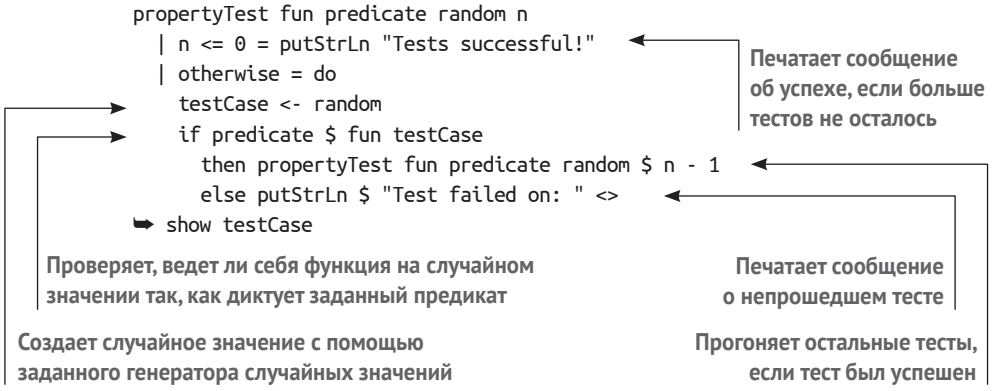
Создает случайный список



Мы можем обобщить понятие теста свойств, включив в него функцию, подлежащую тестированию, предикат, проверяющий эту функцию, и действие ввода-вывода, которое генерирует случайное значение в качестве аргумента. Соответствующий код показан в листинге ниже.

Листинг 9.7 Обобщенное действие для тестирования свойства

```
propertyTest :: Show a => (a -> b) -> (b -> Bool) -> IO a -> Int -> IO ()
propertyTest fun predicate random n
  | n <= 0 = putStrLn "Tests successful!"
  | otherwise = do
    testCase <- random
    if predicate $ fun testCase
      then propertyTest fun predicate random $ n - 1
      else putStrLn $ "Test failed on: " <>
➥ show testCase
```



Итак, мы пришли к очень общему и полезному определению обертки, которую можно использовать для конструирования любых тестов свойств. Сделаем тривиальное предположение, что функция `sort` действительно сортирует список, а функция `id` – нет:

```
ghci> propertyTest sort sorted (applyGlobalStdGen randomList' ::
➥ IO [Int]) 100
Tests successful!
ghci> propertyTest id sorted (applyGlobalStdGen randomList' ::
➥ IO [Int]) 100
Test failed on: [8872,9320,331,9174,7295,7971,7136,4134,7692,1235,8030,
➥ 3595,9078,4121,2722,23,3092,998,1005,2172,795,8495,1960,7359,7353,
➥ 16,3706,4893,6819,3984,7179,8933,8109,951,6330,542,4480,2864,2411,
➥ 2531,8971,2380,4653,1649,224,3466,5091,1834,5436,2803,7251,445,2795,
➥ 8274,7546,3241,3971,3624,955,6299,7857,1152,5028,2899,686,355,6695,
➥ 7098,6713]
```

Теперь, имея базовое представление о том, как реализуется тест свойств, обсудим некоторые проблемы, присущие нашему подходу:

1. Мы должны строить специальный генератор случайных значений для каждого типа данных, который желаем протестировать.
2. Каждый генератор случайных значений должен быть явно определен, и на него необходимо явно ссылаться при прогоне тестов.
3. Тест не допускает предусловий, которым должны удовлетворять входные данные, чтобы тест можно было прогонять.
4. Тест не возвращает результат, обозначающий успех или неудачу.

Проницательный читатель сразу же укажет решение первой проблемы: класс типов! Вторую проблему можно решить, предоставив генераторы случайных значений, *допускающие композицию*; это значит, что существует некий способ взять уже имеющиеся генераторы и создать из них новый. Поговорим об этом подробнее.

9.1.6 Определение постусловий для случайных значений

Модуль `System.Random` предоставляет класс типов `Random`, который можно использовать с уже рассмотренными нами `StdGen` для генерирования случайных значений:

```
ghci> applyGlobalStdGen random :: IO Int
-7583503057972408353
ghci> applyGlobalStdGen random :: IO Bool
False
ghci> applyGlobalStdGen random :: IO Double
0.5396729243902469
```

Haskell автоматически будет использовать корректную реализацию, зависящую от типа, который мы собираемся протестировать. Это пример *полиморфизма по типу возвращаемого значения*. Правильная реализация выбирается на основе типа, ожидаемого *в точке вызова*. Тип может выбрать вызывающая функцию сторона! Хотя параметрический полиморфизм имеется во многих других языках, выбор типа часто производится при создании значения, а не при его потреблении.

Класс типов `Random` уже имеет много экземпляров для базовых типов. Но не для списков. Кроме того, этот класс типов не позволяет легко компоновать генераторы (например, предоставив предусловия для генерируемых значений). Чтобы решить эти проблемы, мы хотим обернуть обобщенные действия ввода-вывода новым типом, обозначающим, что результат генерируется случайным образом. Мы сделаем это, создав с помощью `newtype` тип, единственное поле которого – действие ввода-вывода. А затем напишем более общую версию функции `randomListIO`. Это показано в листинге ниже.

Листинг 9.8 Тип и умные конструкторы для генераторов случайных значений

```
newtype RandomIO a = RandomIO {runRandomIO :: IO a}

one :: Random a => RandomIO a
one = RandomIO $ applyGlobalStdGen random

some :: Random a => RandomIO [a]
some = RandomIO $ do
  n <- applyGlobalStdGen $ uniformR (0, 100)
  replicateIO n $ runRandomIO one
```

Определяет тип с единственным полем, которое инкапсулирует действие ввода-вывода

Создает RandomIO, используя глобальный StdGen для порождения случайного значения некоторого типа

Генерирует список случайного размера и использует его для порождения случайных элементов

Тип называется `RandomIO`, потому что инкапсулирует действие ввода-вывода, способное порождать случайные значения. `one` и `some` – умные конструкторы, которые создают значения `RandomIO` для типов, имеющих экземпляр класса типов `Random`. `replicateIO` – функция, которая повторяет действие ввода-вывода заданное число раз и агрегирует результаты в списке. Код приведен ниже.

Листинг 9.9 Функция агрегирования, повторяющая действие ввода-вывода

```
replicateIO :: Int -> IO a -> IO [a]
replicateIO n act
  | n <= 0 = return []
  | otherwise = do
    x <- act
    xs <- replicateIO (n - 1) act
    return $ x : xs
```

Возвращает пустой список, если заданный размер равен нулю или отрицателен

Вызывает действие

Рекурсивно повторяет действие

Предпосылает результату действия результаты рекурсивного повторения и возвращает окончательный список

Теперь `RandomIO` можно встроить в любое действие ввода-вывода, обратившись к полю `runRandomIO`, а затем просто вызвав действие ввода-вывода:

```
ghci> runRandomIO one :: IO Float
0.12762898
ghci> runRandomIO some :: IO [Bool]
[True,True,True,True,False,False,False,False,False,False,False,True,False,
➥ True,False,False,True,False,True,True,True,True,False,True,False,
➥ False,True,False,True,False,True]
```

Снова воспользовавшись полиморфизмом по типу возвращаемого значения, мы можем применить `RandomIO` для генерирования значений разных типов. Однако для целей тестирования желательно более точно определить, какого вида значения генерировать.

> **ПРИМЕЧАНИЕ** Когда действия или вычисления обертываются `newtype`, поле в новом конструкторе часто имеет префикс `run`. Так как запись содержит только одно поле, действие можно обернуть, используя один лишь конструктор (без указания поля). В общем случае это немного упрощает чтение кода.

Для этого мы можем написать функцию, которая модифицирует существующий `RandomIO`, отфильтровывая значения, которые не должны генерироваться. Таким образом, мы можем скомпоновать наши умные конструкторы с функциями, которые описывают, как должно выглядеть случайное значение. С этой целью напишем функцию, которая неограниченное число раз повторяет вызов генератора, пока некий предикат не вернет `True`. С ее помощью мы уже можем образовывать новые генераторы, порождающие только неотрицательные и непустые значения. Это показано в листинге ниже.

Листинг 9.10 Функция, модифицирующая результат генератора случайных значений

Инкапсулирует следующее действие ввода-вывода конструктором RandomIO

Выполняет предоставленное RandomIO для генерирования случайного значения

```
suchThat :: RandomIO a -> (a -> Bool) -> RandomIO a
suchThat rand pred = RandomIO $ do
  val <- runRandomIO rand
  if pred val
    then return val
    else runRandomIO $ suchThat rand pred

nonNegative :: (Num a, Ord a, Random a) => RandomIO a
nonNegative = one `suchThat` (> 0)

nonEmpty :: Random a => RandomIO [a]
nonEmpty = some `suchThat` (not . null)
```

Возвращает значение, если оно обладает заданным свойством, в противном случае повторяет действие

Повторно выполняет RandomIO, пока не будет сгенерировано неотрицательное значение

Повторно выполняет RandomIO, пока не будет сгенерирован непустой список

`suchThat` – прекрасный пример того, как мы разворачиваем `RandomIO`, чтобы использовать его в действии ввода-вывода, а затем снова помещаем внутрь `RandomIO`, обертывая весь блок `do` конструктором. С помощью этого приема мы можем создать генератор чи-

таемых значений типа `String`. Увы, экземпляр `Random Char` по умолчанию порождает много непечатаемых символов. Но мы можем использовать функции `isAscii` и `isControl` из модуля `Data.Char`, чтобы проверить, можно ли напечатать и произнести составляющие строку литеры. Пользуясь функцией `all`, можно проверить, что этот предикат истинен для всех элементов `String`. Однако при выполнении такого генератора обнаруживается проблема:

```
ghci> import Data.Char
ghci> asciiString = some `suchThat`
➥ all (\c -> isAscii c && not (isControl c))
ghci> runRandomIO asciiString
""
ghci> runRandomIO asciiString
""
ghci> runRandomIO asciiString
""
```

Что происходит? Оказывается, что заданный нами предикат настолько специфичен, что скорее породит пустой список, чем желательное нам значение. Все дело в том, что значения типа `Char` попрежнему генерируются случайно. И всего одно неподходящее `Char` заставляет повторять весь процесс генерирования!

Упражнение. Функтор для случайных значений

`suchThat` действует на наш генератор примерно как фильтр. А вот чего мы до сих пор не сделали, так это эквивалент функции `map` для наших случайных значений. Чтобы обобщить все по максимуму, мы могли бы реализовать экземпляр класса типов `Functor` для `RandomIO`, где `fmap` применяет к случайному значению какую-то функцию. Реализуйте такой экземпляр!

По счастью, это можно исправить, создав сначала генератор печатаемых значений `Char`, а затем используя его для построения генератора `asciiString`. Ограничимся только символами ASCII. Нормальные (нерасширенные) литеры – либо печатаемые, либо управляющие символы, – поэтому достаточно отфильтровывать последние во время дополнительной проверки символа на принадлежность кодировке ASCII. Чтобы построить `String` с помощью генератора `Char`, мы напишем функцию, которая работает в точности как `some`, но пользуется генератором, переданным в качестве аргумента, вместо `randomIO`. Это показано в листинге ниже. Аналогично мы можем создать генератор значений типа `String`, содержащих только буквы.

Листинг 9.11 Генераторы случайных ASCII-символов и строк

```
asciiChar :: RandomIO Char
asciiChar = one `suchThat`
➥ (\c -> isAscii c && not (isControl c))

letterChar :: RandomIO Char
letterChar = asciiChar `suchThat` isLetter

manyOf :: RandomIO a -> RandomIO [a]
manyOf rio = RandomIO $ do
  n <- applyGlobalStdGen $ uniformR (0, 100)
  replicateIO n (runRandomIO rio)

asciiString :: RandomIO String
asciiString = manyOf asciiChar

letterString :: RandomIO String
letterString = manyOf letterChar
```

Определяет генератор случайных печатаемых ASCII-символов, требуя, чтобы сгенерированный Char принадлежал диапазону ASCII и не был управляющим символом

Определяет генератор случайных букв, еще сильнее ограничивая генератор ASCII-символов

Генерирует случайный список с помощью заданного RandomIO

Определяет генераторы случайных значений типа String, которые содержат только определенные символы, путем композиции других генераторов с помощью manyOf

Теперь мы умеем порождать случайные значения типа `String` для тестирования!

```
ghci> runRandomIO asciiString
"N:=P!RVyo7O{=QU6"
ghci> runRandomIO asciiString
"&q~,=Dm:"
ghci> runRandomIO letterString
"EdgcdiBHQcwIJAkCKlhemiaUWBMeNxrHOkWigRuktokfMyxXzYklFBO"
ghci> runRandomIO letterString
"uEETBtLDCIneVmHFNvICxPLuEwSoSnj"
```

Теперь мы можем модифицировать функцию `propertyTest`, так чтобы она использовала наш генератор `RandomIO`, и при этом упростить ее, разрешив использовать только один предикат, который определяет, прошел ли тест, возвращая значение типа `Bool`. Это позволит тестировать более сложные случаи, где требуется следить за правильностью более общего вычисления, а не только за результатом одной функции. Окончательный код функции приведен в листинге ниже.

Листинг 9.12 Действие для тестирования свойств с применением специализированных генераторов случайных значений

```
propertyTest :: Show a => (a -> Bool) -> RandomIO a -> Int -> IO ()
propertyTest predicate random n
  | n <= 0 = putStrLn "Tests successful!"
  | otherwise = do
    testCase <- runRandomIO random
```

Печатает сообщение об успехе, если больше тестов не осталось

Создает случайное значение с помощью заданного генератора

```
    if predicate testCase
      then propertyTest predicate
➥ random $ n - 1
      else putStrLn $ "Test failed on: " <> show testCase
```

Проверяет, ведет ли себя функция на случайном значении так, как диктует заданный предикат, и либо прогоняет остальные тесты, либо печатает контрпример

Итак, успешно справившись с задачей построения генераторов случайных значений и вспомогательных функций для их конструирования и модификации, мы можем наконец воспользоваться функцией `propertyTest` для тестирования ранее написанного кода и посмотреть, как она с этим справится.

Упражнение. Эффективные ASCII-строки

Наш способ определения `letterString` крайне неэффективен. Вместо того чтобы производить случайный выбор из множества желательных значений `Char`, мы выбираем из огромного множества, а затем отфильтровываем ненужное. Чтобы сделать его более эффективным, а заодно и более общим, напишите функцию, которая принимает непустой список элементов и случайным образом выбирает один из них. Затем воспользуйтесь этой функцией для создания новых генераторов ASCII-символов и букв, на этот раз работающих быстрее:

```
elements :: [a] -> RandomIO a
elements [] = error "elements cannot work with an empty list!"
elements xs = ...
```

9.2 Рандомизированное тестирование

После всестороннего рассмотрения генераторов случайных значений мы можем подумать о том, как *правильно* обращаться со *свойствами*. То есть как писать свойства таким образом, чтобы они обеспечили корректность программы.

Но прежде чем начать, мы должны структурировать проект. Для этого поместим весь написанный к этому момент тестовый код в модуль `Test.SimpleCheck`. Он послужит нам каркасом для тестирования. Сделав это, мы можем скопировать код из проекта на тему шифра Цезаря из главы 2 в свой исходный код и приступить к написанию тестов для него.

Какого рода свойства нужно тестировать? Разумеется, это сильно зависит от спецификации программы. В нашем случае требуется просто убедиться, что наши базовые предположения о коде выполнены. Одно такое предположение заключалось в том, что двойное

применение шифра ROT13 дает исходное значение, т. е. шифр ROT13 симметричен. Это можно выразить в нашем каркасе тестирования. Свойство функции состоит в том, что для любого входного значения `s` должно иметь место равенство `s == rot13 (rot13 s)`. Это легко выразить с помощью функции, которую мы можем передать нашей функции `propertyTest`, как показано в листинге ниже.

Листинг 9.13 Проверка свойства, выражающего симметрию функции ROT13

```
propRot13Symmetrical :: IO ()
propRot13Symmetrical =
  propertyTest
    (\s -> s == rot13 (rot13 s))
    asciiString
    100
```

Проверяет, что двойное применение rot13 к значению оставляет это значение неизменным

Использует для тестирования генератор ASCII-строк

Задает количество тестов – 100

Выбор генератора случайных значений для этого свойства важен! В главе 2 мы говорили, что свойство симметрии имеет место для *букв*. Однако в этом тесте проверяется более общий случай. Рассматриваются не только буквы, но и любые печатаемые литеры ASCII. Следовательно, этот тест проверяет не только, что `rot13` симметричен, но и что это свойство справедливо для подаваемых на вход значений типа `Char`. По счастью, мы написали `rot13` именно так, поэтому тест проходит!

```
ghci> propRot13Symmetrical
Tests successful!
```

Рассмотренный выше тест является *тестом спецификации*. Тот факт, что шифр ROT13 симметричен, – часть его спецификации. Это свойство гарантирует, что функция `rot13` правильно реализует спецификацию. Теперь мы можем переработать `rot13` и лежащие в его основе механизмы, не опасаясь нарушить спецификацию. Если мы где-то напутаем, то тест об этом скажет (хочется надеяться)!

> **ПРИМЕЧАНИЕ** Полезно давать тестам свойств специальные имена. В некоторых кодовых базах имена таких функций начинаются префиксом `prop` или даже `prop_` (хотя линтеры не приветствуют использование знака подчеркивания в идентификаторах). Если кодовая база содержит как тесты свойств, так и автономные тесты на основе примеров, то для последних иногда употребляются имена с префиксом `test` или `test_`.

Мы можем добавить дополнительные тесты для написанных нами вспомогательных функций. Конечно, для этого проекта писать комплект тестов – перебор, но это хорошая возможность познакомиться с методикой тестирования. Но нам придется изучить некоторые вспомогательные функции более внимательно и решить, можем ли мы проверить их функциональность.

Прежде всего рассмотрим функцию `alphabetRot`, которая поворачивает алфавит. Свойство заключается в том, что после поворота новое возвращенное функцией значение типа `Char` должно быть элементом исходного алфавита. Это мы можем проверить! Однако вход этой функции несколько сложнее, чем то, что мы видели до сих пор. Функция принимает `Alphabet` (псевдоним типа `String`), `Int` и литеру `Char`, которая должна быть элементом алфавита в первом аргументе. Легко видеть, что у нас нет готового генератора, который поставлял бы все это. Однако мы можем написать новый специально для этого теста – он будет создавать непустой алфавит и выбирать аргумент `Char` из этого алфавита. Код показан ниже.

Листинг 9.14 Проверка свойства, выражающего замкнутость функции поворота алфавита

```
propAlphabetRotClosed :: IO ()
propAlphabetRotClosed = propertyTest prop gen 100
  where
    prop (alphabet, n, c) =
      let c' = alphabetRot alphabet n c
       in c' `elem` alphabet

    gen = RandomIO $ d o
      alphabet <-
        runRandomIO $
          asciiString `suchThat` (not . null)
      n <- runRandomIO $ elements [-100 .. 100]
      c <- runRandomIO $ elements alphabet
      return (L.nub alphabet, n, c)
```

Выполняет проверку свойства на 100 тестах

Определяет проверяемое тестом свойство – после поворота символ принадлежит тому же алфавиту

Генерирует непустой алфавит, содержащий литеры ASCII

Выбирает случайное значение от –100 до 100

Выбирает случайное значение из сгенерированного алфавита

Удаляет из алфавита дубликаты и возвращает сгенерированные значения

В этом тесте генератор выполняет всю трудную работу, подготавливая разумные условия для теста. Прежде чем передать тесту аргументы, генератор исключает из алфавита дубликаты, так что мы можем предполагать, что ни один элемент не встречается в нем дважды. Аргумент `Char` должен выбираться *случайно* из случайного алфавита. Это значит, что даже если один и тот же случайный алфавит сгенерирован дважды, велики шансы, что из него будет выбран другой `Char`.

9.2.1 Преимущество ссылочной прозрачности

Мы хотим немного отвлечься на теоретическое обсуждение того, *почему* тестирование свойств эффективно конкретно в Haskell. До сих пор мы эксплуатировали чрезвычайно важное свойство языка, чтобы сделать тесты полезными: *чистоту*. Это значит, что написанные на нем функции свободны от побочных эффектов. На результат функции не может повлиять *ничто, кроме ее входа*. А следовательно, простого наблюдения входа и выхода достаточно, чтобы судить о правильности функции, потому что больше никакого выхода или поведения, за которым имело бы смысл наблюдать, в системе нет. Кроме того, достаточно спроектировать генератор случайных значений для функции, потому что не существует никакого окружения, которое нужно было бы искусственно настраивать для применимости теста. Корректность теста определяется только правильным выбором случайных значений и правильной интерпретацией тестируемого свойства.

Это свойство также называется *ссылочной прозрачностью*. Любое ссылочно-прозрачное выражение (например, применение чистой функции) можно заменить значением, полученным в результате его вычисления. Мы должны быть уверены только в одном – что значения выражений совпадают с ожидаемыми.

В качестве примера рассмотрим функцию `isMisc` из главы 2. Она говорит, что литера не является ни буквой, ни цифрой. Кроме нее, в проекте есть еще несколько предикатов (`isLower`, `isUpper` и `isDigit`), которые определяют, является ли литера строчной буквой, заглавной буквой или цифрой. Отсюда можно сделать вывод, что `isMisc` равен `True` тогда и только тогда, когда ни один из трех других предикатов не равен `True`. При реализации шифра Цезаря мы просто предполагали, что это так. Однако это легко проверить с помощью теста свойства, показанного в листинге ниже.

Листинг 9.15 Тест свойства для проверки инварианта предиката

```
propIsMiscIsDefault :: IO ()
propIsMiscIsDefault =
  propertyTest
    (\c -> isMisc c == not
➥ (isLower c || isUpper c || isDigit c))
    one
    10000
```

Проверяет, что isMisc истинен тогда и только тогда, когда никакой другой предикат от Char истинным не является

Задает число тестов – 10 000

Использует обобщенный генератор одного случайного значения типа Char

Тестируемое свойство – простой предикат, проверяющий эквивалентность `isMisc` и логического терма. Снова благодаря чистоте Haskell эквивалентность термов можно проверить с помощью простого оператора `==`. Важно отметить, что эти термы *эквивалентны*

только *с точки зрения наблюдателя*. Они порождают одинаковые выходы для одинаковых входов, но технически имеют разное поведение и по-разному потребляют память. Но это нормально, потому что в чистом коде эквивалентности с точки зрения наблюдателя достаточно, когда речь заходит о правильности программы. Наши функции управляются только своими входными данными и не могут делать ничего, кроме вычисления выхода. Теперь, получив представление о тестировании свойств, мы готовы усовершенствовать методики тестирования, перейдя от собственного примитивного каркаса к настоящему лидеру тестирования: каркасу QuickCheck.

9.3 Каркас тестирования QuickCheck

Мы начали эту главу с разработки собственного «каркасика» тестирования: SimpleCheck. Однако ему присущ ряд проблем, которые трудно игнорировать:

- конструировать генераторы случайных значений сложно и трудоемко;
- реализовать композицию свойств нелегко;
- невозможно заглянуть внутрь тестируемых данных (на каких значениях производится тестирование и как часто);
- невозможно модифицировать тесты, задав предусловия;
- поведение наших генераторов случайных значений фиксировано, и изменить его непросто;
- не делается попыток минимизировать количество непрошедших тестов, их может стать много, а понять, в чем дело, трудно.

Из-за всего этого наш каркас было бы трудно использовать в реальных проектах. Однако решение есть: каркас тестирования под названием *QuickCheck*. Разработанный Коэном Классеном и Джоном Хьюзом из Технического университета Чалмерса в 1999 году, QuickCheck популяризовал идею тестирования свойств, и теперь существуют его реализации на многих других языках – функциональных (OCaml, Clojure, Scala) и нефункциональных (C++, Java, TypeScript). Его дизайн очень близок к нашему каркасу SimpleCheck. Он предлагает:

- функции для тестирования свойств;
- комбинаторы для композиции и модификации свойств;
- вспомогательную функциональность для отладки поведения тестирования;
- типы и классы типов для генерирования случайных значений.

Прежде чем переходить непосредственно к примерам тестирования свойств, мы хотим дать краткий обзор этой библиотеки и ее функциональности.

Но сначала посмотрим, как включить QuickCheck в проект. Поскольку QuickCheck находится в репозитории Stackage, мы можем просто добавить его в свой файл `package.yaml`, в секцию `dependencies`. Мы добавим строку `- QuickCheck >= 2.0`, чтобы скачивать версию QuickCheck не ниже `2.0`. При сборке приложения или запуске REPL `stack` автоматически разрешит эту зависимость. Секция `dependencies` должна выглядеть примерно так:

```
dependencies:
- base >= 4.7 && < 5
- QuickCheck >= 2.0
- random >= 1.2.1.1
```

Разобравшись с тем, как запустить каркас QuickCheck, перейдем к рассмотрению его самых важных концепций.

9.3.1 Тип Property в QuickCheck

Первым делом остановимся на самом простом строительном блоке: функции `quickCheck`. Она получает какое-то свойство и тестирует его, создавая действие ввода-вывода. Два важных экземпляра такого свойства – `Bool` и тип `Property`, определенный в QuickCheck. Кроме того, такое свойство может быть функцией, возвращающей свойство! Следовательно, функция, которая возвращает `Bool` (как простой предикат), является свойством.

> **ПРИМЕЧАНИЕ** На внутреннем уровне свойство должно обладать экземпляром класса типов `Testable`. Этот класс определяет, как преобразовать тип в свойство. Здесь мы не будем его рассматривать, поскольку нам нет нужды конструировать собственные экземпляры для него, но, встречая `Testable prop` в документации по QuickCheck, полезно вспоминать о *свойстве*.

Воспользуемся этим для тестирования симметричности функции `rot13`:

```
ghci> import Test.QuickCheck
ghci> prop_rot13Symm s = s == rot13 (rot13 s)
ghci> :t prop_rot13Symm
prop_rot13Symm :: String -> Bool
ghci> quickCheck prop_rot13Symm
+++ OK, passed 100 tests.
```

QuickCheck успешно протестировал это свойство. Отметим, что он каким-то *волшебным* образом создал данные для нашей функции. Но этим мы займемся позже. А пока продолжим изучение проверки свойств.

> **ПРИМЕЧАНИЕ** Свойства QuickCheck обычно записываются с префиксом `prop_`. Так QuickCheck сможет автоматически прогнать тесты всех найденных в модуле свойств. Некоторые линтеры Haskell запрещают использовать знак _ в идентификаторах, но для свойств QuickCheck мы сделаем исключение.

Что, если бы мы захотели протестировать с помощью этого свойства неправильную реализацию?

```
ghci> symbols = upperAlphabet ++ lowerAlphabet ++ digits
ghci> rot13' = map $ (\ch -> if ch `elem` symbols then alphabetRot symbols
  ➥ 13 ch else ch)
ghci> prop_rot13Symm s = s == rot13' (rot13' s)
ghci> quickCheck prop_rot13Symm
*** Failed! Falsified (after 2 tests and 1 shrink):
"a"
```

В этом примере мы намеренно реализовали `rot13` неправильно и тут же использовали эту функцию в свойстве. По существу, мы внесли ошибки во время рефакторинга (а тестируемое свойство не изменилось). QuickCheck сообщает, что тест не прошел (а перед этим два теста прошли), и завершается на входе `"a"`. Очень похоже на нашу прежнюю функцию тестирования свойств!

Еще одна важная концепция – *сокращение* (shrink). Если тест не проходит, QuickCheck автоматически пытается уменьшить его, т. е. брать числа поменьше, включать в списки и отображения меньше элементов и т. д. Конечно, и сокращенное значение должно быть контрпримером, иначе QuickCheck просто сообщит нам о большем значении. Это делается, чтобы предъявляемые контрпримеры были понятнее. В нашем случае свойство обнаруживает проблему при повороте на один символ. Но могло бы случиться, что она проявляется при использовании огромной строки, состоящей из разных символов. Мы видим это при тестировании того же свойства с помощью нашего каркаса SimpleCheck:

```
ghci> propertyTest prop_rot13Symm asciiString 100
Test failed on: "JQCj_s&S>mOSLnPW5XXIqi9+*IF4WLbTW$&hW?4FTg\\:
➥ !a-vo_QfR/cy4,w2.Xg~5R0sCsX==-KPyD^Wy\\"
```

Нет нужды спрашивать, какой результат теста проще отладить. Еще одна замечательная особенность QuickCheck – то, что для многих типов сокращение детерминированно. Многократный прогон теста всякий раз будет давать один и тот же контрпример!

```
ghci> quickCheck prop_rot13Symm
*** Failed! Falsified (after 2 tests and 1 shrink):
"a"
ghci> quickCheck prop_rot13Symm
```

```
*** Failed! Falsified (after 5 tests and 1 shrink):
"a"
ghci> quickCheck prop_rot13Symm
*** Failed! Falsified (after 4 tests and 2 shrinks):
"a"
```

Это гарантирует гораздо лучшую повторяемость тестов и помогает классифицировать типичные ошибки в функциях.

Настало время протестировать что-то более сложное. Мы можем записать свойство замкнутости функции `alphabetRot`. Оно показано в листинге ниже.

Листинг 9.16 Свойство замкнутости функции поворота алфавита в QuickCheck

```
prop_alphabetRotClosed :: Alphabet -> Int -> Char -> Bool
prop_alphabetRotClosed alphabet n c =
  let c' = alphabetRot alphabet n c
   in c' `elem` alphabet
```

Определяет свойство, которое проверяет, что повернутая литера по-прежнему принадлежит тому же алфавиту

Проверим это свойство:

```
ghci> quickCheck prop_alphabetRotClosed
*** Failed! Falsified (after 1 test and 1 shrink):
""
0
'a'
```

Тут надо обратить внимание на две вещи. Во-первых, QuickCheck умеет генерировать аргументы для функции с тремя аргументами! Наши функции могут иметь сколько угодно аргументов, и все они будут генерироваться независимо. И второе – наше свойство некорректно! Но почему? Как видим, для свойства были заданы аргументы `alphabet = ""`, `n = 0` и `c = 'a'`. Проблема в том, что `c` должно быть элементом `alphabet` и, что еще важнее, что `alphabet` не должен быть пустым! И как же решить эту проблему?

В своем каркасе мы решили ее путем модификации генератора случайных значений, так чтобы он порождал только правильные значения. То же самое мы теперь должны сделать в QuickCheck.

9.4 Генерирование случайных значений для тестирования

Генерирование случайных значений лежит в основе QuickCheck, потому что без него тестировать свойства бессмысленно. Кроме того, библиотека умеет автоматически выводить правильные генераторы просто из типов. Как это реализовано?

Ответ дает класс типов `Arbitrary`:

```
ghci> :i Arbitrary
type Arbitrary :: * -> Constraint
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a -> [a]
  {-# MINIMAL arbitrary #-}
```

Он определяет функцию `arbitrary`, используемую для порождения случайных значений, и функцию `shrink`, которую мы рассмотрим позже. Из минимального определения видно, что `shrink` можно вообще игнорировать (пока)! А что такое `arbitrary` и что такое `Gen`?

9.4.1 Генератор случайных значений Gen

`Gen` очень похож на наш тип `RandomIO`. Он определяет, как генерировать случайные значения конкретного типа. Мы можем использовать функцию `generate` для порождения случайных значений с помощью `Gen` в действии ввода-вывода:

```
ghci> generate arbitrary :: IO Int
15
ghci> generate arbitrary :: IO [Bool]
[True,True,True,False,True,False]
ghci> generate arbitrary :: IO (Maybe Float)
Just (-9.993277)
ghci> generate arbitrary :: IO (Either String Char)
Right '.'
```

Здесь мы видим, что генераторы могут порождать значения разных типов. Можно также определить собственные типы, пользуясь вспомогательными функциями, предоставляемыми QuickCheck. Ниже мы приведем их неполный перечень:

- `choose :: Random a => (a, a) -> Gen a` – ведет себя как `randomRIO` и может использоваться для генерирования числовых значений, принадлежащих интервалу;
- `chooseAny :: Random a => Gen a` – преобразует случайное значение из экземпляра `Random` в `Gen` того же типа;
- `oneof :: [Gen a] -> Gen a` – случайно выбирает генератор из заданного списка;
- `elements :: [a] -> Gen a` – порождает генератор, который случайно выбирает элемент из заданного списка;
- `suchThat :: Gen a -> (a -> Bool) -> Gen a` – модифицирует генератор, так чтобы он пропускал нежелательные элементы, удовлетворяющие булеву предикату;

- `listOf :: Gen a -> Gen [a]` – генерирует список случайной длины с элементами, полученными от генератора;
- `listOf1 :: Gen a -> Gen [a]` – то же, что `listOf`, но не генерирует пустых списков;
- `vectorOf :: Int -> Gen a -> Gen [a]` – генерирует список случайных значений заданной длины;
- `vector :: Arbitrary a => Int -> Gen [a]` – то же, что `vectorOf`, но генератор элементов списка выводится на уровне типа;
- `shuffle :: [a] -> Gen [a]` – порождает случайные перестановки заданного списка;
- `sublistOf :: [a] -> Gen [a]` – порождает случайные подсписки заданного списка.

Это не все вспомогательные функции, предоставляемые QuickCheck, но, пожалуй, наиболее полезные. Среди них мы видим своего старого знакомого, `suchThat`! С помощью этих функций мы можем определять новые генераторы. Но как именно это выглядит?

Упражнение. Комбинаторы генераторов

Мы уже реализовали `suchThat` в своем каркасе SimpleCheck. Теперь самое время скопировать другие функции из QuickCheck, чтобы лучше прочувствовать, как вычисляются значения. Следите, чтобы сигнатуры типов были точно такими же (только вместо `Gen` используйте `RandomIO`), и попытайтесь добиться максимальной производительности.

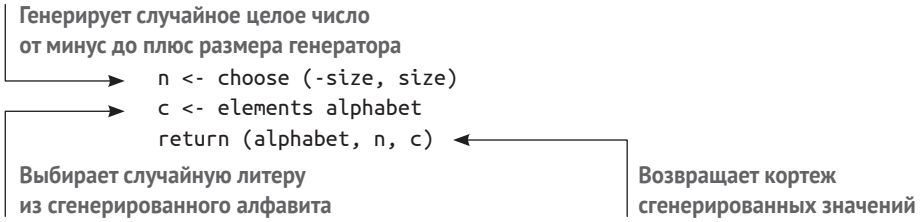
Определение генераторов очень похоже на определение `RandomIO`. Для их создания мы можем использовать `do`-нотацию, с которой познакомились, изучая действия ввода-вывода. Эта нотация поможет нам воспроизвести генератор, которым мы пользовались выше для свойства алфавита. Использовать `runRandomIO` внутри `do`-нотации генератора необязательно. Мы можем просто использовать другой генератор, например `arbitrary` или `choose`, и получить от него нужные значения. Как и раньше, необходимо возвращать кортеж, потому что генерирование одного значения зависит от других. Код приведен в листинге ниже.

Листинг 9.17 Генератор QuickCheck, порождающий данные для функции поворота алфавита

```
gen :: Gen (Alphabet, Int, Char)
gen = do
  size <- getSize
  alphabet <- arbitrary `suchThat` (not . null)
```

Определяет внутренний размер генератора

Создает непустой алфавит



Небольшое отличие этого генератора от предыдущего заключается в использовании генератора `getSize` и его применении для определения `n`. Все генераторы в QuickCheck имеют *внутренний размер*, который определяет некоторые параметры генерируемых значений. Например, стандартный генератор `Int` порождает значения, абсолютная величина которых не больше внутреннего размера генератора. По умолчанию размер генератора равен 30, но его можно изменить функцией `resize`, например:

```
ghci> import Control.Monad
ghci> import Test.QuickCheck
ghci> sample = replicateM 10
ghci> sample $ generate (arbitrary :: Gen Int)
[0,-8,12,9,-9,29,-15,19,-28,-26]
ghci> sample $ generate (resize 100 $ arbitrary :: Gen Int)
[-69,-38,13,-29,40,79,-33,-59,-28,-33]
ghci> sample $ generate (resize 10000 $ arbitrary :: Gen Int)
[270,-5063,-121,-5645,-6974,9861,-1830,-5653,4493,9132]
```

Это решает проблему в случае, когда мы ради производительности хотим управлять размером генератора извне. Примером может служить `listOf`, где внутренний размер определяет максимальный размер списка. То же относится к `arbitrary`, параметризованному типом `String`. При конструировании своих генераторов мы должны учитывать этот размер и по возможности использовать `getSize`. При вызове одного генератора из другого размер передается, поэтому мы можем строить одни генераторы из других, не думая об изменении внутреннего размера (кроме тех случаев, когда хотим сделать это явно).

> **ПРИМЕЧАНИЕ** Функция `sample` уже имеется в QuickCheck, но она производит выборку переменного размера (от 0 до 20) из генератора. Ей можно воспользоваться, чтобы быстро узнать, как выглядят сгенерированные значения и как поведение генератора изменяется в зависимости от размера.

Итак, мы создали генератор для своей задачи, пора применить его на деле. Желая выполнить конкретный генератор для какого-то свойства, мы можем использовать функцию `forAll`, которая позволяет явно задать генератор тестов данного свойства. Результатом

этой функции снова является свойство. Это позволяет переписать тест свойства замкнутости `alphabetRot`, пользуясь только средствами QuickCheck, как показано в листинге ниже.

Листинг 9.18 Свойство, описывающее замкнутость функции `alphabet`

```
prop_alphabetRotClosed :: Property
prop_alphabetRotClosed =
  forAll gen prop
  where
    prop :: (Alphabet, Int, Char) -> Bool
    prop (alphabet, n, c) =
      let c' = alphabetRot alphabet n c
       in c' `elem` alphabet

    gen :: Gen (Alphabet, Int, Char)
    gen = do
      size <- getSize
      alphabet <- arbitrary `suchThat` (not . null)
      n <- choose (-size, size)
      c <- elements alphabet
      return (L.nub alphabet, n, c)
```

Определяет свойство, подлежащее тестированию предоставленным генератором

Определяет свойство, проверяющее, что повернутый символ принадлежит тому же алфавиту

Возвращает текущий внутренний размер генератора

Генерирует случайный непустой алфавит

Удаляет из алфавита дубликаты и возвращает сгенерированные значения

Выбирает случайное значение из сгенерированного алфавита

Выбирает случайное значение в диапазоне, зависящем от внутреннего размера генератора

Эта реализация должна находиться в новом модуле, чтобы не конфликтовать с предыдущей. Заметим, что хотя код похож на тот, что был написан раньше, он допускает более выразительное тестирование, чем прежде.

Мы видели, как создавать свои генераторы, но, очевидно, не хотим делать это для каждого теста. Разработав собственные типы, мы хотели бы создать экземпляр класса типов `Arbitrary`, чтобы было проще осуществлять композицию генераторов. Этот вопрос мы и рассмотрим далее.

9.4.2 Пример: AssocMap

Вернемся мысленно к главе 5, где ввели тип `AssocMap`, тип полиморфного отображения. Его мы и хотим протестировать следующим. Однако для этого нам понадобится порождать случайные значения этого типа. Класс типов `Arbitrary` специально предназначен для этой цели, поэтому мы хотим создать для него экземпляр этого типа!

Вспомним, как работал тип `AssocMap`. Он использовал список кортежей `[(k, v)]` для установления отображения между *ключа-*

ми и *значениями*. Его полное определение показано в следующем листинге.

Листинг 9.19 Определение AssocMap из главы 5

```
newtype AssocMap k v = AssocMap [(k, v)]
  deriving (Show)
```

Каждый ключ может встречаться только один раз, поэтому если мы хотим генерировать произвольные значения этого типа, то должны соблюдать это свойство. Мы можем воспользоваться тем фактом, что `Gen` уже имеет экземпляр `Functor`, поэтому можно применить `fmap` и `<$>` для модификации сгенерированных значений. Раз так, то мы можем сгенерировать список ключей без дубликатов и на его основе создать столько же значений, после чего объединить ключи и значения в отображение. Код генератора `AssocMap` показан ниже.

Листинг 9.20 Генератор ассоциативных отображений на основе QuickCheck

```
import qualified Data.List as L
import Test.QuickCheck

...

genAssocMap :: (Eq k, Arbitrary k, Arbitrary v) => Gen (AssocMap k v)
genAssocMap = do
  keys <- L.nub <$> arbitrary
  vals <- vectorOf (L.length keys) arbitrary
  return $ AssocMap (L.zip keys vals)
```

Импортирует необходимые модули

Создает случайный список ключей без дубликатов

Создает столько же значений, сколько имеется ключей

Возвращает значение, сгенерированное AssocMap, комбинируя сгенерированные ключи и значения

Ограничение для этого типа важно. Поскольку мы хотим использовать *произвольные* ключи и значения, у типов должен иметься экземпляр класса типов `Arbitrary`. Функция `arbitrary` позволяет использовать в нашем генераторе AssocMap *любой* тип, для которого существует генератор `arbitrary`. Теперь мы можем протестировать этот генератор и посмотреть, что он порождает для разных типов:

```
ghci> generate genAssocMap :: IO (AssocMap Int Bool)
AssocMap [(0,True),(-1,False),(-23,True),(19,False),(14,False),(2,False)]
ghci> generate genAssocMap :: IO (AssocMap Int (Either Bool Int))
AssocMap [(-10,Left True),(-20,Right 19)]
ghci> generate genAssocMap :: IO (AssocMap Bool [Float])
AssocMap [(True,[7.2917347,17.285803,-12.867395,-3.3579175])]
```

Итак, мы покончили с первой частью написания экземпляра `Arbitrary`. Однако и вторую часть – функцию `shrink` – игнорировать не следует.

9.4.3 Сокращение тестов

При тестировании, особенно когда тест не проходит, нам чаще всего не интересны чрезмерно сложные и запутанные контрпримеры свойств. Однако если входы для тестов свойств выбираются случайным образом, то в конечном итоге будут генерироваться большие контрпримеры.

В QuickCheck встроен механизм *сокращения*, задача которого – уменьшить найденный контрпример. После того как найден вход, для которого тест не проходит, он подвергается сокращению, и проверяется, по-прежнему ли имеет место контрпример. Это повторяется до тех пор, пока значение поддается сокращению, не переставая быть контрпримером. По завершении процедуры мы можем считать, что нашли минимальный вход, на котором тест свойства не проходит. Эта идея показана на рис. 9.2.

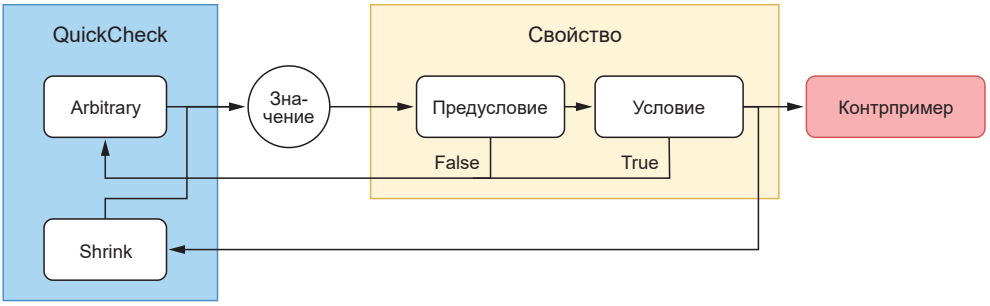


Рис. 9.2 Взаимодействие QuickCheck, свойств и сокращения

Для каждого типа данных, для которого QuickCheck умеет генерировать случайные данные, она также умеет сокращать значения, потому что в классе типов `Arbitrary` определена функция, реализующая сокращение, – `shrink`. Ее сигнатура типа `a -> [a]` говорит нам нечто очень важное: это *чистая функция*, которая генерирует список сокращенных значений по входу! Таким образом, ее выход зависит только от входа, а значит, функция детерминированна! В ней нет ничего случайного. Именно поэтому контрпримеры, генерируемые QuickCheck, иногда кажутся детерминированными; на основе входа мы можем вернуть предопределенные кандидаты для последующего испытания с помощью QuickCheck. Хороший пример дает функция `shrink` для типа `Char`:

```
ghci> shrink 'H'
"abchABC"
ghci> shrink 'A'
"abc"
ghci> shrink 'h'
```

```
"abc"
ghci> shrink '\n'
"abcABC123 "
ghci> shrink '1'
"abcABC"
```

Здесь мы видим, как при последовательном сокращении значений `Char` в какой-то момент появляется `'"abc"'`. Это делается намеренно, потому что если `'H'` – контрпример, то таковым может оказаться и любое значение из `'"abchABC"'`. Не менее важно, что `shrink` не должна возвращать свой вход как часть сокращенных значений. В противном случае сокращение могло бы зациклиться!

Ну и как нам сократить тип `AssocMap`? Если мы говорим, что типы ключа и значения имеют экземпляр `Arbitrary`, то это также означает, что кортеж и список кортежей таких типов имеют такой экземпляр. Поэтому мы можем просто сократить список в нашем типе `AssocMap`, но с осторожностью. `shrink` потенциально может изменять значения, так что нарушится наш инвариант: никакой ключ не должен встречаться дважды. Поэтому при сокращении нужно позаботиться о том, чтобы каждый ключ встречался только один раз. Реализация сокращения показана в листинге ниже.

Листинг 9.21 Функция `shrink` для ассоциативных отображений

```
shrinkAssocMap ::
  (Eq k, Arbitrary k, Arbitrary v) =>
  AssocMap k v ->
  [AssocMap k v]
shrinkAssocMap (AssocMap xs) =
  L.map
    (AssocMap . L.nubBy (\(k1, _) (k2, _) -> k1 == k2))
    (shrink xs)
```

Создает новые значения AssocMap из сокращенных кандидатов путем удаления ключей-дубликатов

Сокращает значения, присутствующие в заданном отображении, и порождает на их основе кандидаты

Мы можем протестировать эту функцию сокращения на произвольном `AssocMap`:

```
ghci> am <- generate (genAssocMap :: Gen (AssocMap Int Int))
ghci> am
AssocMap [(-26,11),(6,3),(12,-24),(-24,-4),(30,12),(-5,27),(-18,9)]
ghci> shrinkAssocMap am
[AssocMap [],AssocMap [(-24,-4),(30,12),(-5,27),(-18,9)],...
```

Это позволит получить гораздо более простые контрпримеры, если мы напишем свойство, которому `AssocMap` не удовлетворяет! Для этого осталось положить последний кусочек пазла: сконструировать экземпляр `Arbitrary (AssocMap k v)`, пользуясь написанными функциями. Код показан ниже.

Листинг 9.22 Экземпляр класса типов `Arbitrary` для ассоциативных отображений

```
instance (Eq k, Arbitrary k, Arbitrary v) => Arbitrary (AssocMap k v)
  where
    arbitrary = genAssocMap
    shrink = shrinkAssocMap
```

Использует genAssocMap в качестве генератора случайных значений

Использует shrinkAssocMap в качестве функции сокращения для этого типа

Этот экземпляр позволяет нам генерировать случайные значения типа `AssocMap` внутри свойств. Кроме того, наш тип теперь можно использовать в объемлющем генераторе для порождения еще более сложных значений.

Упражнение. Свойства для ассоциированных списков

Мы реализовали экземпляр класса типов `Arbitrary` для `AssocMap`, теперь настала ваша очередь написать свойства для типа, которые можно протестировать. Важно убедиться, что тип выполняет свою основную функцию – является отображением, поэтому необходимо проверить следующее:

- вставленное значение можно найти;
- при попытке вставить другое значение с уже существующим ключом старое значение перезаписывается;
- удаление ключа работает, как положено;
- `empty` действительно пустое;
- изменение отображения не приводит к нарушению инварианта.

Реализуйте свойства QuickCheck, тщательно проверяющие этот тип. Если сможете придумать дополнительные свойства, реализуйте и их тоже!

Итак, мы познакомились с основами QuickCheck, научились писать свойства и генераторы и знаем, как работает сокращение. Теперь займемся вопросом об использовании каркаса QuickCheck и включении его в свои проекты.

9.5 Практическое использование тестирования свойств

До сих пор мы изучали различные концепции QuickCheck: как тестируются свойства и как генерируются случайные значения. Теперь мы хотим сосредоточиться на том, как использовать эту библиотеку для тестирования кода и на что обращать внимание при проектировании тестов.

Начнем с интерактивного использования рассмотренной выше функции `quickCheck`. Для этого вернемся к тестированию функций сортировки и прежде всего нашего предиката `sorts`. Протестируем его с помощью QuickCheck:

```
ghci> quickCheck $ sorts sort
+++ OK, passed 100 tests.
ghci> quickCheck $ sorts id
+++ OK, passed 100 tests.
```

Минуточку, что такое? Что-то явно не так, функция `id` никак не может сортировать список! Требуется расследование.

9.5.1 Уровень детализации и отчеты о покрытии

Чтобы начать расследование, мы можем использовать несколько *модификаторов*, которые изменяют порядок выполнения теста свойства. Первый называется `verbose` и позволяет получить детальную информацию обо всех тестах:

```
ghci> quickCheck . verbose $ sorts sort
Passed:
[]

Passed:
[()]

Passed:
[()]

Passed:
[(),()]

Passed:
[(),()]

...

+++ OK, passed 100 tests.
```

Уже неправильно. Тестируемые списки содержат значения единичного типа! Понятно, что список, содержащий только значения такого типа, сортируется тривиально, т. к. у этого типа имеется единственный представитель. Нам необходимо явно указать тип, который QuickCheck должна использовать при тестировании.

> **ПРИМЕЧАНИЕ** То, что здесь произошло, называется *выбором типа по умолчанию*. В некоторых ситуациях, когда неясно, какой тип использовать, GHC выбирает тип автоматически. При тестировании нашего свойства по умолчанию был выбран тип элементов `()`. Обычно GHC об этом предупреждает.

Это можно сделать, явно указав имя свойства и сигнатуру типа для него:

```
ghci> :{
ghci| prop_sortSorts :: [Int] -> Bool
ghci| prop_sortSorts xs = sort `sorts` xs
ghci| :}
ghci> quickCheck . verbose $ prop_sortSorts
Passed:
[]

Passed:
[]

Passed:
[]

Passed:
[]

Passed:
[0,3]

Passed:
[4,-3]

...

+++ OK, passed 100 tests.
```

Вот теперь мы видим разумные значения! Но поскольку QuickCheck увеличивает размер генератора медленно, он начинает с многократного тестирования одного и того же тривиального значения: пустого списка! Разумеется, это плохой тест. Нужно более внимательно исследовать, на каких данных мы тестируем. Для этого мы можем воспользоваться функцией `collect` для получения статистики, которую мы хотели бы вычислять для входных данных. В нашем случае интерес представляет количество тестов с пустыми данными:

```
ghci> :{
ghci| prop_sortSorts :: [Int] -> Property
ghci| prop_sortSorts xs = collect (null xs) $ sort `sorts` xs
ghci| :}
ghci> quickCheck prop_sortSorts
+++ OK, passed 100 tests:
95% False
 5% True
```

Интересно! Получается, что 5 % наших тестов пустые. Чуть более информативным вывод можно сделать, воспользовавшись функцией `label`, которая назначает метки различным статистикам:

```
ghci> :{
ghci| prop_sortSorts :: [Int] -> Property
ghci| prop_sortSorts xs = label (if null xs then "empty" else "not empty")
➥ $ sort `sorts` xs
ghci| :}
ghci> quickCheck prop_sortSorts
+++ OK, passed 100 tests:
95% not empty
 5% empty
```

Заметим, что в обоих случаях сигнатура типа нашего свойства изменилась с `[Int] -> Bool` на `[Int] -> Property`. Это связано с тем, что `collect` и `label` порождают значение типа `Property`. Теперь вопрос: как улучшить этот тест? Тестировать тривиальные случаи бесполезно. Мы хотим, чтобы свойство тестировалось на списках, содержащих не менее двух элементов. Можно было бы изменить генератор тестов, но это слишком сложное решение для данного случая. Проще использовать *предусловие*. Как мы добавляли предусловия в SimpleCheck, так можем сделать это и для QuickCheck. Для добавления предусловий в свойство служит оператор `==>`:

```
ghci> :{
ghci| prop_sortSorts :: [Int] -> Property
ghci| prop_sortSorts xs = length xs >= 2 ==> sort `sorts` xs
ghci| :}
ghci> quickCheck prop_sortSorts
+++ OK, passed 100 tests; 43 discarded.
```

Вывод изменился. Теперь QuickCheck сообщает, что 43 теста были отброшены! Эти тесты не удовлетворяют предусловию, поэтому полностью проигнорированы. Это делается, в основном чтобы улучшить покрытие нашего свойства тестами. Не имеет смысла тестировать функцию на тривиальных входных данных, потому что вероятность найти таким образом ошибку низкая. Однако совсем исключать их тоже не всегда правильно. Предусловие хорошо, когда мы хотим полностью исключить некоторые тесты, поскольку они не отражают спецификацию или не имеют отношения к тестируемой функциональности. Покрытие тестами – нечто совершенно иное.

QuickCheck располагает функциями для этой цели. Одна из них называется `cover` и сообщает кое-какую информацию о покрытии:

```
ghci> :{
ghci| prop_sortSorts :: [Int] -> Property
ghci| prop_sortSorts xs = cover 25 (length xs >= 2) "non-trivial" $
➥ sort `sorts` xs
ghci| :}
ghci> quickCheck prop_sortSorts
+++ OK, passed 100 tests (93% non-trivial).
```

`cover` принимает три аргумента: число от 0 до 100, определяющее минимально необходимое количество успешных тестов; условие, при котором значение считается прошедшим тест, и метку для прошедших тест значений. Затем `cover` собирает статистику о количестве прошедших тестов и соответственно помечает ее.

> **ПРИМЕЧАНИЕ** Функция `cover` не приводит к отказу теста свойства! Она только *предупреждает* о том, что требуемое покрытие не достигнуто. Если вы хотите, чтобы тест считался непрошедшим, то оберните его функцией `checkCoverage`, которая преобразует предупреждение в отказ.

9.5.2 Модификация параметров теста

Наконец, мы хотим взглянуть, как еще можно модифицировать поведение `QuickCheck`. Мы уже видели, как `verbose` увеличивает уровень детализации QuickCheck, однако другие модификаторы позволяют оказать более значительное влияние на поведение тестирования:

- `verbose :: Testable prop => prop -> Property` – увеличивает уровень детализации теста свойств и может использоваться в сочетании с `quickCheck` или в качестве вспомогательной функции `verboseCheck`;
- `verboseShrinking :: Testable prop => prop -> Property` – включает информацию о сокращении в вывод;
- `noShrinking :: Testable prop => prop -> Property` – подавляет сокращения для данного свойства;
- `withMaxSuccess :: Testable prop => Int -> prop -> Property` – определяет, сколько тестов должно быть успешно выполнено, по умолчанию 100;
- `within :: Testable prop => Int -> prop -> Property` – считает, что тест свойства не прошел, если он не завершился за указанное число микросекунд.

Модификаторы `withMaxSuccess` и `within` особенно важны для тестирования. Иногда 100 тестов недостаточно, а иногда мы хотим, чтобы лимитированный по времени код выполнялся быстро. Однако все это зависит от теста. Далее мы поговорим о том, как включать свойства и проверки в проекты.

9.5.3 Построение комплектов тестов

На этапе инициализации нового проекта командой `stack new` программа Stack не только создает структуру для нашей библиотеки и исполняемого файла. Она также создает файл `Spec.hs` в подкаталоге *test*. Эта часть проекта зарезервирована для кода, который предположительно будет тестировать библиотеку и исполняемый файл.

Комплект тестов можно создать командой `stack test`, но откуда Stack знает, прошел тест или нет? По умолчанию это определяется кодом выхода программы, откомпилированной с использованием `Spec.hs` в качестве главного модуля. И как же мы можем писать комплекты тестов с такими вводными?

Во-первых, нам нужен какой-то шаблон для построения комплектов тестов, который понуждает программу возвращать код неудачи, если тест не проходит. Очень простой шаблон показан в листинге ниже.

Листинг 9.23 Шаблон комплекта тестов, используемого вместе со Stack

```
module Main where

import System.Exit (exitFailure, exitSuccess)

main :: IO ()
main = do
  success <- ...
  if success
    then exitSuccess
    else exitFailure
```

Импортирует необходимые действия для различных статусов выхода

Использует тот или иной статус выхода в зависимости от успешности теста

Теперь вопрос в том, как выполнить сами тесты. Для этого QuickCheck не предлагает четкой структуры, но предоставляет ряд очень полезных функций. Из них самая полезная `quickCheckAll`. Она собирает все свойства, определенные в модуле, и применяет `quickCheck` к каждому из них. Как, спросите вы? Для этого используется расширение *Template Haskell* и схема именования свойств.

Template Haskell – расширение GHC, позволяющее применять метапрограммирование и напрямую манипулировать абстрактным синтаксическим деревом. Его можно использовать, чтобы автоматически сгенерировать функции, которые иначе пришлось бы писать вручную. Перечисление всех свойств с префиксом `prop_` и вызов для них `quickCheck` – одно из преобразований, применяемых QuickCheck. Таким образом, модуль может определить набор свойств и предоставить способ прогона всех содержащихся в нем тестов (с помощью `quickCheckAll`). В нашем главном модуле мы можем собрать тесты из комплектов тестов, а затем проверить их результат.

Для этого мы хотим создать еще два модуля наряду со `Spec.hs`, которые будут нашими комплектами тестов. Модули `SuiteOne.hs` и `SuiteTwo.hs` будут содержать какие-то посторонние свойства с различными модификаторами и поведениями. Оба экспортируют действие с именем `runTests`, которое использует Template Haskell для сбора всех свойств в модуле. Затем мы можем сослаться на эти комплекты из главного модуля. Код первого комплекта тестов приведен в листинге ниже.

Листинг 9.24 Пример комплекта тестов

```
{-# LANGUAGE TemplateHaskell #-}

module SuiteOne where

import Test.QuickCheck

prop_true :: Int -> Bool
prop_true = const True

prop_false :: Int -> Bool
prop_false = const False

return []

runTests :: IO Bool
runTests = $quickCheckAll
```

Активирует расширение языка TemplateHaskell

Определяет свойство, проверка которого всегда успешна

Определяет свойство, проверка которого всегда неудачна

Для обхода ошибки в системе шаблонов

Выполняет quickCheck для всех свойств с префиксом prop_

Чтобы все это работало, необходимо активировать расширение языка `TemplateHaskell`. Знак доллара перед `quickCheckAll` – специальный синтаксис, используемый для Template Haskell. В коде есть какое-то странное предложение `stray return []`. Это рекомендуемый способ обхода ошибки в системе шаблонов. По аналогии с этим комплектом, мы можем реализовать еще один, с более сложными свойствами. Он показан ниже.

Листинг 9.25 Еще один пример комплекта тестов

```
{-# LANGUAGE TemplateHaskell #-}

module SuiteTwo where

import Test.QuickCheck

prop_addPos :: Int -> Int -> Property
prop_addPos x y =
  withMaxSuccess 500 $
    x > 0 && y > 0 ==> x + y > 0

prop_multZero :: Int -> Property
prop_multZero x =
  noShrinking $
    cover 95 (x /= 0) "non-zero" $ x * 0 == 0

return []

runTests :: IO Bool
runTests = $quickCheckAll
```

Активирует расширение языка TemplateHaskell

Определяет свойство с модифицированным количеством тестов

Определяет свойство, в котором не используется сокращение и определены требования к покрытию

Для обхода ошибки в системе шаблонов

Выполняет quickCheck для всех свойств с префиксом prop_

Конечно, свойства могут исчисляться сотнями, и у нас никогда не должно возникать необходимости записывать их проверку вручную.

Если мы хотим расширить комплект тестов, то просто формулируем еще одно свойство, и при следующем запуске `stack test` оно проверяется автоматически. Теперь мы можем включить комплекты тестов в главный модуль. Это показано в листинге ниже.

Листинг 9.26 Пример главного модуля, используемого вместе с двумя комплектами тестов

```
module Main where

import qualified SuiteOne as S1
import qualified SuiteTwo as S2
import System.Exit (exitFailure, exitSuccess)

main :: IO ()
main = do
  s1success <- S1.runTests
  s2success <- S2.runTests
  if s1success && s2success
    then exitSuccess
    else exitFailure
```

Импортирует демонстрационные комплекты тестов

Импортирует необходимые действия по разным кодам выхода

Выполняет комплекты тестов

Возвращает код выхода, отражающий успешность тестов

Функция `runTests` возвращает булево значение, сообщающее, как завершились тесты. Но важнее то, что она выводит полезную информацию при прогоне теста. Таким образом, порядок прогона тестов важен. Тест, который должен появляться в распечатке первым, должен идти первым в действии ввода-вывода `main`.

Упражнение. Закорачивание прогона тестов

В своей реализации мы не использовали никакого закорачивания логики, чтобы досрочно прекратить тестирование, если один комплект завершается неудачно еще до начала второго. Мы тестируем *все* свойства и только потом решаем, успешно завершился тест или нет. Это не то поведение, которое мы хотели бы иметь на этапе разработки. Добавьте аргумент, который разрешает или запрещает быстрый отказ, если комплект тестов завершается неудачно.

Для передачи аргументов тесту можно использовать `--test-arguments`:

```
shell $ stack test --test-arguments "--my-cool-flag"
```

Наконец, пришло время выполнить наш тест. В нашем случае проект называется `properties`, поэтому комплект назван `properties-test`:

```
shell $ stack test
properties> test (suite: properties-test)
```

```
=== prop_true from test/SuiteOne.hs:7 ===
+++ OK, passed 100 tests.

=== prop_false from test/SuiteOne.hs:10 ===
*** Failed! Falsified (after 1 test):
0

=== prop_addPos from test/SuiteTwo.hs:7 ===
+++ OK, passed 500 tests; 1758 discarded.

=== prop_multZero from test/SuiteTwo.hs:12 ===
+++ OK, passed 100 tests (97% non-zero).

properties> Test suite properties-test failed
Test suite failure for package properties-0.1.0.0
    properties-test: exited with: ExitFailure 1
Logs printed to console
```

Это довольно грубый способ выполнения тестов. Мы могли бы также создать дополнительные комплекты на основе файла `package.yaml` или воспользоваться библиотекой *Tasty*, которая позволяет комбинировать QuickCheck, *SmallCheck* (еще одна библиотека тестирования) и ручные автономные тесты в комплекты, которые можно перечислять независимо друг от друга. Но для небольших проектов это, пожалуй, слишком, а нашего метода вполне достаточно.

> **Упражнение. Комплекты тестов в проекте**
>
> Теперь, когда вы знаете, как включать комплекты тестов в проекты, настала ваша очередь заняться этим для всех наших проектов! Подумайте, какие свойства можно придумать для различных встречавшихся выше случаев, и убедитесь, что все работает, как задумано! Тут нельзя однозначно сказать, что правильно, а что неправильно; просто проявите фантазию, создайте экземпляры `Arbitrary` для наших типов и посмотрите, какие свойства сможете предложить.

Отметим также, что для Haskell существуют и другие каркасы тестирования, например *Hedgehog*, похожий на QuickCheck, и *HUnit*, ориентированный больше на автономные тесты на основе примеров. У каждого каркаса есть свойственные только ему возможности, и в зависимости от размера и функциональности проекта вы можете предпочесть тот или иной.

9.5.4 Эффективность тестирования

Мы уже рассмотрели преимущества чистых функций Haskell с точки зрения тестопригодности, но хотим завершить главу чуть более развернутым обсуждением этого вопроса. Зачем вообще мы пишем тес-

ты программного обеспечения? Чего мы пытаемся достичь? Самое очевидное свойство программы, которое мы хотим обеспечить, – *правильность*. Мы хотим, чтобы программы порождали результаты в соответствии со спецификацией. Например, генератор простых чисел, естественно, должен порождать только простые числа. И еще мы хотим гарантировать, что свойства программы не изменяются со временем. После рефакторинга кода все тесты, написанные ранее для программы, по-прежнему должны успешно проходить.

Это вселяет уверенность в программах. Мы можем быть уверены, что они порождают правильные результаты и что мы можем продолжать работу над кодом без добавления регрессионных тестов. Именно это – уверенность – является главным преимуществом Haskell. Статические типы, чистые функции и четкое отделение побочных эффектов от программирования без таковых – вот что делает Haskell в высшей степени тестопригодным, а сами тесты – *эффективными*.

Поскольку чистые функции не имеют побочных эффектов, они являются просто *преобразованиями* данных. Тестирование свойств этих преобразований должно гарантировать сохранение свойств: если данные на входе обладают какими-то свойствами, то и на выходе они должны обладать какими-то свойствами. В этой главе мы видели, как реализовать такие тесты:

- мы используем предусловие или специально разработанный генератор, чтобы гарантировать свойства входных данных;
- свойства, которые мы хотим гарантировать, проверяются для результата преобразования:
  - для функций проверяется правильность результатов;
  - для типов данных проверяется правильность конструирования и сохранение инвариантов в процессе преобразований;
- мы компонуем свойства в комплекты тестов, чтобы не терять уверенности в правильности взаимодействия функций и типов данных на уровне одного модуля или всей системы.

Хотя другие языки программирования также позволяют разработчикам писать тесты, такой высокой степени уверенности редко можно достичь при столь малом объеме тестирования, как в этой главе. Тесты свойств в чистом функциональном коде сродни *формальной верификации* в том смысле, что формализованные свойства занимают место наблюдения за поведением программ. В результате мы получаем программное обеспечение, которое гораздо легче поддается рефакторингу, расширению и масштабированию. В таких процессах, как *непрерывная интеграция* и *непрерывная доставка*, эти тесты образуют основу успешного функционирования и обычно избавляют вас и вашу компанию от различных мрачных сценариев.

Резюме

- Для использования случайных значений в чистых функциях требуется, чтобы генератор случайных значений был аргументом и частью возвращаемого значения.
- Доступ к глобальному генератору случайных значений из действий ввода-вывода возможен с помощью `AtomicGenM` и `globalStdGen`.
- Классы типов `Random` и `Uniform` дают широкий спектр типов, для которых возможно генерирование случайных значений.
- При тестировании свойств формальные характеристики и случайные входные данные используются для нахождения ошибок в реализациях.
- Для простоты написания случайные генераторы должны допускать композицию.
- QuickCheck автоматизирует выбор генераторов случайных значений посредством использования системы типов и классов типов.
- Класс типов `Arbitrary` не только предоставляет генератор случайных значений для типа, но также умеет сокращать значение до более компактного и простого.
- У генераторов в QuickChecks имеется внутренний размер, что позволяет создавать случайные значения разной сложности.
- Функцию `quickCheckAll` можно использовать для автоматического сбора свойств, присутствующих в модуле, и выполнения для них тестов.
- Тестирование свойств в чистом функциональном коде гарантирует соблюдение формальных свойств типов данных, функций и модулей.

10 Цифровая музыкальная шкатулка

Краткое содержание этой главы:

- корректная работа с числовыми значениями и их классами типов;
- работа с бесконечными списками;
- моделирование предметно-ориентированной задачи с помощью типов данных Haskell;
- построение абстракций и высокоуровневых структур над низкоуровневыми реализациями.

До сих пор мы обсуждали очень серьезные темы. Античные схемы шифрования, кратчайшие пути и операции с CSV-файлами – нешуточное дело. В этой главе мы собираемся немного порезвиться и раскрепостить свою творческую натуру, занявшись сочинением музыки! Не расстраивайтесь, если под рукой нет музыкального инструмента или вы вообще не знаете, как создается музыка. Мы воспользуемся мощью Haskell, поручив ему не только изготовить музыкальный синтезатор, но и заняться самой композицией, так что мы без труда создадим очередной музыкальный шедевр, который слушатели будут помнить в веках.

Моделирование реальных данных в программах – обычное дело для любого программиста. В этой главе мы научим вас размышлять об абстрагировании деталей и моделировании части реальной си-

стемы (западной теории музыки) с помощью типов и значений. Такой вид моделирования возникает всякий раз, как вы собираетесь написать программу, включающую какую-то бизнес-логику или концепции, напрямую не связанные со спецификациями и архитектурой программы. Haskell также способен к созданию в высшей степени абстрактных моделей, в которых опущены все детали, не имеющие отношения к рассматриваемому сценарию. Именно это мы и собираемся свершить в настоящей главе.

Чтобы осуществить свои музыкальные мечты, мы будем моделировать в программе звук и напишем код, способный его порождать. По ходу дела мы узнаем о специфике работы Haskell с числовыми значениями, а также столкнемся с предметно-ориентированными задачами, которые нам предстоит решить. Имея функции и типы для создания звука, мы затем построим над ними абстракции, чтобы реализовать в программе сочинение музыки, не забивая голову низкоуровневыми деталями. Это заставит нас задуматься о том, что такое правильная абстракция и как определить преобразования, берущие на себя решение наиболее трудных частей задачи.

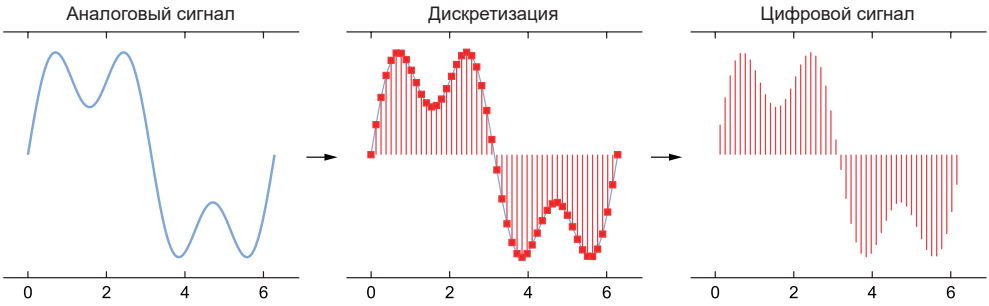
10.1 Моделирование звука с помощью чисел

Самый важный вопрос, на который нам предстоит ответить, – как программа создает звуки? И вообще что такое звук? В реальном мире *звуки* – это колебания воздуха, достигающие нашей барабанной перепонки и порождающие возбуждение в мозгу, проявляющееся в виде чувства *слуха*. Для компьютера все проще. Если говорить о монофоническом аудио, то звук – это сигнал. Если требуется два канала (левый и правый), то звук состоит из двух сигналов. Но что такое сигнал? В аналоговом мире сигнал – это изменяющееся во времени значение со сколь угодно малой разрешающей способностью. Можно считать, что это непрерывная математическая функция времени. В цифровом мире мы используем дискретные отсчеты для аппроксимации аналогового сигнала. Отсчет – это просто одно числовое значение. Последовательность отсчетов – это дискретизированный сигнал. См. рис. 10.1.

Как видно из этого рисунка, точность воспроизведения аналогового сигнала ограничена *частотой дискретизации* (сколько производится отсчетов сигнала в секунду) и *разрешающей способностью* (сколько битов используется для представления одного отсчета). Каковы разумные значения этих двух характеристик? Благодаря теореме отсчетов Найквиста–Шеннона[1] мы знаем, что если требу-

[1] В отечественной литературе это утверждение называется теоремой Котельникова. – *Прим. перев.*

ется восстановить сигнал с известной максимальной частотой, то частота дискретизации должна быть как минимум вдвое больше этой частоты.



Звуковой сигнал – это продолжающееся изменение амплитуды (представленной напряжением, которое преобразуется в давление звуковой волны в динамиках) со временем. В аналоговой области сигнал – не набор отдельных значений, а непрерывная функция

В цифровой области непрерывный сигнал дискретизируется (квантуется) путем отбора значений непрерывного сигнала в конечном числе точек, зачастую находящихся на фиксированном расстоянии друг от друга

Дискретизированный сигнал состоит из упорядоченной последовательности значений, представляющих исходный непрерывный сигнал. Если имеется достаточное число точек, то по ним можно реконструировать исходный сигнал. Именно так представляются цифровые звуковые сигналы

Рис. 10.1 Дискретизация аналогового сигнала

Для звуковых сигналов частоты больше 22 кГц несущественны, поскольку человек их не слышит (и звуковая аппаратура, как правило, неспособна воспроизводить такие частоты). Отсюда и возникает широко распространенная частота дискретизации 44.1 кГц, поскольку теоретически она позволяет идеально восстановить звуковой сигнал. А как насчет разрешающей способности? Она в основном влияет на *отношение сигнал–шум* для нашего сигнала, т. е. говорит, какая часть интересующего нас сигнала пробивается через фоновый шум, вызванный ошибками дискретизации. Для музыки применяются значения от 8 до 32 бит. В файловом формате WAV, который мы будем рассматривать в этой главе, обычно используется 16 бит.

10.1.1 Зоопарк классов числовых типов

Сначала поговорим о том, как представить дискретизированный сигнал. Один отсчет, как было сказано, – это просто числовое значение. Для его представления можно использовать тип `Double`. Тогда сигнал – не что иное, как список таких отсчетов, `[Double]`. И теперь мы должны решить, какую частоту дискретизации хотим поддержать. Более высокая частота 92 кГц могла бы помочь в сглаживании верхних частот, но мы остановимся на стандартной частоте 44.1 кГц, стандартно применяемой при записи музыки на CD-диск.

Мы также можем определить некоторые другие типы, чтобы было проще обсуждать предмет. Поскольку нам придется задавать частоты и промежутки времени, определим псевдонимы `Hz` и `Seconds` типа `Double`. И теперь можно написать первые вспомогательные функции, которые будут сообщать, сколько отсчетов сигнала заданной частоты необходимо взять в периоде заданной длительности. Позже нам это понадобится. Код типов и функций приведен в листинге ниже.

Листинг 10.1 Типы и вспомогательные функции для работы с сигналами

```
type Sample = Double

type Signal = [Sample]

type Hz = Double

type Seconds = Double

sampleRate :: Double
sampleRate = 44100

samplesPerPeriod :: Hz -> Int
samplesPerPeriod hz = round $ sampleRate / hz

samplesPerSecond :: Seconds -> Int
samplesPerSecond duration = round $
➥ duration * sampleRate
```

Определяет псевдонимы типов для отсчетов, сигналов, частоты в герцах и секунд

Определяет константу, равную частоте дискретизации

Определяет число отсчетов, необходимых для представления одного периода заданной частоты

Вычисляет число отсчетов, необходимое для представления заданного количества секунд

Здесь мы видим, как используется функция `round`. Она принимает значение, имеющее экземпляр класса типов `RealFrac`, и возвращает значение, имеющее экземпляр класса типов `Integral`. Это означает, что мы можем взять значение типа `Float` или `Double` и преобразовать его в значение типа `Int` или `Integer`. Мы используем ее, потому что число отсчетов должно быть целым. Не бывает половины отсчета. Смысл обеих функций можно уяснить, поняв тот простой факт, что частота дискретизации равна числу отсчетов в секунду. Период частоты `n` герц в точности равен `1 / n`. Умножение на частоту дискретизации дает число отсчетов, помещающихся в один период частоты.

Прежде чем продолжить, скажем несколько слов о классах `RealFrac` и `Integral`. Что это такое и для чего они используются? Существует целый зоопарк классов числовых типов, и эти два – его обитатели. Мы уже обсуждали класс типов `Num`, который определяет основные арифметические операции, однако в нем нет операции деления, что странно, поскольку мы видели две функции, которые ее выполняют:

```
ghci> :t div
div :: Integral a => a -> a -> a
ghci> :t (/)
(/) :: Fractional a => a -> a -> a
```

Эти функции являются частями классов типов `Integral` и `Fractional` соответственно. Класс `Integral` представляет функции для вычисления частного и остатка от деления целых чисел, а класс `Fractional` – функции для деления и вычисления обратного для чисел с плавающей точкой:

```
ghci> :i Integral
type Integral :: * -> Constraint
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer
  {-# MINIMAL quotRem, toInteger #-}
instance Integral Word
instance Integral Integer
instance Integral Int
ghci> :i Fractional
type Fractional :: * -> Constraint
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
instance Fractional Float
instance Fractional Double
```

В классе `Integral` есть важная функция `toInteger`, которая позволяет преобразовывать типы в значения `Integer`. Удивительно, но мы встречаем дополнительные классы типов в ограничении для `Integral`, а именно классы `Real` и `Enum`:

```
ghci> :i Real
type Real :: * -> Constraint
class (Num a, Ord a) => Real a where
  toRational :: a -> Rational
  {-# MINIMAL toRational #-}
ghci> :i Enum
type Enum :: * -> Constraint
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]
  {-# MINIMAL toEnum, fromEnum #-}
```

Типы, имеющие экземпляр класса `Real`, можно преобразовать в `Rational`; этот тип, определяющий отношение двух целых чисел, мы рассмотрим далее в данной главе.

> **ПРИМЕЧАНИЕ** С математической точки зрения, имя типа класса `Real` неудачно, потому что теоретически иррациональные числа, например `pi` из модуля `Prelude`, также являются частью класса и могут быть преобразованы в рациональное число с помощью `toRational`. Это момент, в котором теория расходится с практикой.

Класс `Enum` определяет перечисление типов; он используется в синтаксисе списков, когда определяется диапазон. Более внимательно мы рассмотрим его в главе 15.

```
ghci> toRational (5 :: Double)
5 % 1
ghci> toRational (5.5 :: Double)
11 % 2
ghci> [1..10] :: [Int]
[1,2,3,4,5,6,7,8,9,10]
ghci> enumFromTo 1 10 :: [Int]
[1,2,3,4,5,6,7,8,9,10]
```

Теперь можно взглянуть на класс `RealFrac`, который объединяет функции для преобразования дробей в целые значения:

```
ghci> :i RealFrac
type RealFrac :: * -> Constraint
class (Real a, Fractional a) => RealFrac a where
  properFraction :: Integral b => a -> (b, a)
  truncate :: Integral b => a -> b
  round :: Integral b => a -> b
  ceiling :: Integral b => a -> b
  floor :: Integral b => a -> b
  {-# MINIMAL properFraction #-}
```

Функции `round`, `ceiling` и `floor` – наш путь к преобразованию чисел с плавающей точкой в целые. `round` возвращает целое, ближайшее к аргументу, тогда как `ceiling` – наименьшее целое, большее или равное аргументу, а `floor` – наибольшее целое, меньшее или равное аргументу. Еще один подкласс `Fractional` – `Floating`, он содержит несколько математических функций, в частности `sin`, `log` и `sqrt` – для вычисления синуса, логарифма и квадратного корня соответственно.

Хотя этот зоопарк классов типов кажется запутанным, мы можем черпать утешение в том факте, что обычно используемые нами числовые типы, а именно `Integer`, `Float` и `Double`, имеют их экземпляры, когда в этом есть смысл. Это позволяет нам преобразовывать `Int` в `Float` с помощью `fromIntegral` и `Double` в `Integer` с помощью `round`:

```
ghci> fromIntegral (1 :: Int) :: Float
1.0
ghci> round (3.1415 :: Double) :: Integer
3
```

Однако `Floating` не имеет экземпляров для целых значений. Поэтому, желая вычислить синус целого числа, мы должны выполнить явное преобразование, т. к. неявного приведения типов в Haskell попросту не существует:

```
ghci> sin (fromIntegral (1 :: Int)) :: Double
0.8414709848078965
```

Мы не рассмотрели класс `RealFloat`, который можно использовать для доступа к низкоуровневым компонентам чисел с плавающей точкой, поскольку к нашим проектам он не имеет отношения.

10.1.2 Создание периодических функций

Рассмотрев тонкости классов числовых типов, мы можем перейти к созданию звука. В музыке нас интересует *тембр* – количественная характеристика звуков. Но прежде чем спускаться в кроличью нору обертонов, мы можем срезать путь и посмотреть, как это делают другие синтезаторы. Разные характеристики звуков можно получить с помощью разных форм сигналов. Четыре самые распространенные формы сигнала показаны на рис. 10.2.

На этих рисунках изображен один период сигнала. Повторение этих периодов определяется длительностью сигнала и частотой, которую мы пытаемся генерировать. Если время фиксировано, то изменение частоты будет растягивать или сжимать сигнал. Функции `samplesPerPeriod` и `samplesPerSecond` можно использовать для вычисления того, сколько отсчетов необходимо для периода заданной частоты. Повторное взятие этих отсчетов на протяжении всего сигнала даст нам тон некоторой частоты на заданном промежутке времени. Варьирование частот порождает музыку! Для наших целей значения отсчета изменяются от `-1` до `1`. Об их модуляции мы подумаем позже, когда дело дойдет до управления громкостью сигнала. А пока нам нужно реализовать сами формы сигнала.

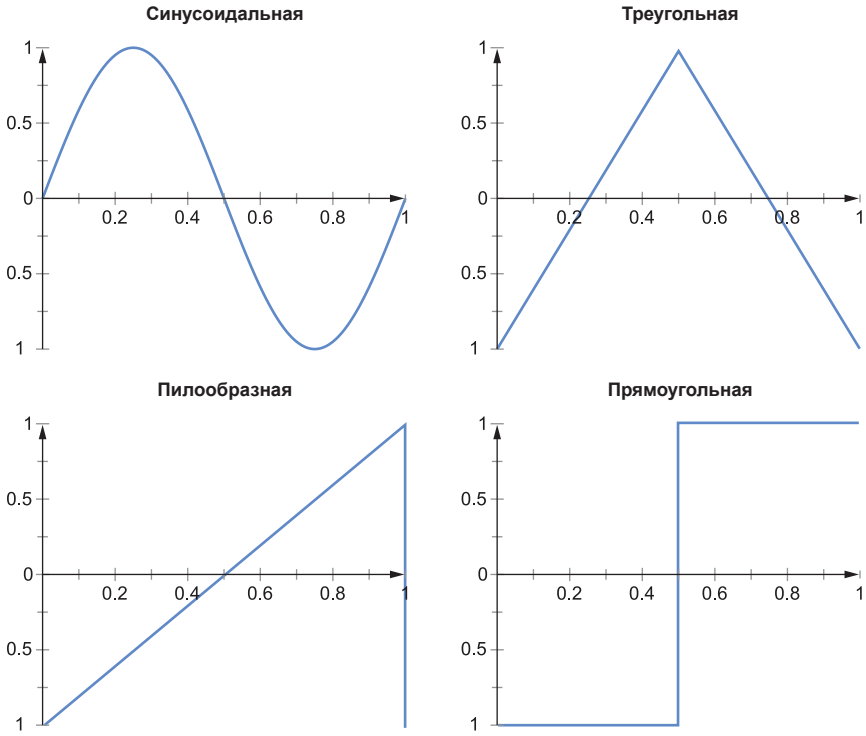


Рис. 10.2 Четыре стандартные формы сигнала

В нашем случае форма сигнала – это просто функция, которая принимает числовой аргумент от `0` до `1` и возвращает значение отсчета для сигнала в этой точке. Этот параметр можно использовать для «сканирования» формы сигнала. Чем быстрее мы сканируем (т. е. чем больше расстояние между временными шагами), тем короче период и тем выше частота. Поскольку формы сигнала на рис. 10.2 – обычные математические функции, мы можем смоделировать их в коде. Синусоидальный сигнал можно сгенерировать с помощью функции `sin` из модуля `Prelude`. Прямоугольный сигнал сравнивает входной параметр с порогом: он равен `-1`, когда вход меньше или равен `0.5`, и `1` в противном случае. Пилообразные и треугольные формы несколько сложнее, потому что необходимо создавать наклонные прямые. Пилообразный сигнал имеет уравнение `2 * t - 1`. Треугольный сигнал состоит из двух частей. Первая половина – пила половинной длины, вычисляемая по формуле `4 * t - 1`. Во второй половине знак вычисленного значения противоположный, поскольку прямая зеркально отражается относительно вертикальной прямой. Поэтому мы прибавляем смещение `+3`, чтобы значения попадали в диапазон от `1` до `-1`. Код показан в листинге ниже.

Листинг 10.2 Типы и вспомогательные функции для работы с сигналами

```
type Wave = Double -> Sample

sin :: Wave
sin t = Prelude.sin $ 2 * pi * t

sqw :: Wave
sqw t
  | t <= 0.5 = -1
  | otherwise = 1

saw :: Wave
saw t
  | t < 0 = -1
  | t > 1 = 1
  | otherwise = (2 * t) - 1

tri :: Wave
tri t
  | t < 0 = -1
  | t > 1 = -1
  | t < 0.5 = 4 * t - 1
  | otherwise = -4 * t + 3
```

Определяет тип для представления форм сигнала в виде функции

Определяет синусоидальную форму, используя константу pi из модуля Prelude

Определяет прямоугольную форму с 50-процентным рабочим циклом

Определяет пилообразную форму

Определяет треугольную форму

Все эти функции определены только для входных значений от `0` до `1`. Для значений вне этого диапазона функции постоянны. Но нам неважно, что делают функции за пределами области определения.

Теперь мы можем воспользоваться этими функциями для вычисления числа отсчетов, необходимого для построения функций, описывающих тишину (т. е. сигнала с постоянным значением 0) и тон, который порождает заданную частоту на протяжении заданного периода. Для этого вычислим, сколько отсчетов нужно для генерирования, и перечислим их, начиная с 0. Затем разделим эти значения по модулю числа отсчетов, необходимого для получения периода правильной частоты. При этом будет создан список входных параметров, который мы затем можем передать любой из функций типа `Wave`. В результате получится наш тон. Код приведен в следующем листинге.

Листинг 10.3 Функции, порождающие сигналы заданной формы

```
silence :: Seconds -> Signal
silence t = replicate (samplesPerSecond t) 0

tone :: Wave -> Hz -> Seconds -> Signal
tone wave freq t = map wave periodValues
```

Возвращает сигнал с постоянным значением 0 и заданной длительности

Отображает функцию wave на несколько значений для создания нужного сигнала

```
  where
    numSamples = samplesPerPeriod freq
    periodValues =
     map
      (\x -> fromIntegral (x `mod` numSamples) / fromIntegral numSamples)
      [0 .. samplesPerSecond t]
```

Вычисляет необходимое число отсчетов для одного периода wave

Создает список, содержащий входные параметры функции, для создания тона заданной длительности и частоты

Мы снова должны воспользоваться функцией `fromIntegral` для деления чисел с плавающей точкой, а не целых. В этой функции мы вручную гарантируем, что функции `wave` подаются повторяющиеся значения на протяжении заданного промежутка времени. Выглядит сложнее, чем необходимо, потому что мы могли бы просто вычислить `wave` один раз, а затем повторить ее. В следующем упражнении у вас будет шанс реализовать эту идею.

Упражнение. Генератор с регулируемым рабочим циклом

Выше мы реализовали, в частности, прямоугольную форму сигнала. Она особенная в том смысле, что является так называемой импульсной волной с 50-процентным рабочим циклом, т. е. время, в течение которого сигнал имеет «низкий» уровень, равно времени, в течение которого его уровень «высокий». Можно определить импульсные волны с любым рабочим циклом, а когда они используются в музыке, характеристики звука изменяются вместе с рабочим циклом! Реализуйте обобщенную импульсную волну с регулируемым рабочим циклом.

Наконец, мы можем использовать эти функции для создания реальных звуков! Для этого воспользуемся библиотекой *HCodecs*, которая умеет читать и записывать файлы в формате WAV – формате несжатого аудио, который могут воспроизводить почти все аудиоплееры. Чтобы сделать эту библиотеку доступной, добавим `HCodecs` и `array` в секцию `dependencies` нашего файла `package.yml`. В этой библиотеке есть функция `exportFile`, которая позволяет записывать аудиоданные в WAV-файл. Секция `dependencies` должна выглядеть так:

```
dependencies:
- base >= 4.7 && < 5
- HCodecs >= 0.5.2
- array >= 0.5.4
```

Далее мы должны сконструировать значение `Audio` – запись, содержащую информацию о частоте дискретизации, отсчетах и числе каналов. Для этого сначала преобразуем наш `Signal` в нечто, что

называется `SampleData` и на внутреннем уровне представляет собой *массивы* для хранения аудиоданных. Массивы имеются в Haskell, но в этой книге не рассматриваются, поэтому мы не станем на них задерживаться. Нам важно, что для работы с ними необходимо добавить `array` в секцию `dependencies`. Затем мы можем воспользоваться функцией `listArray` для построения массива из списка; она принимает кортеж с описанием границ массива и данные из списка, которые мы собираемся конвертировать. Нужно только, чтобы в кортеже описывался диапазон от 0 до длины списка минус 1. Чтобы преобразовать наш тип `Signal` в отсчеты, понятные HCodecs, воспользуемся функцией `fromSample`. Полный код показан в следующем листинге.

Листинг 10.4 Функция для записи аудиоданных в WAV-файл

```
import Data.Array.Unboxed (listArray)
import Data.Audio
import Data.Int (Int16)

signalToSampleData :: Signal -> SampleData Int16
signalToSampleData signal =
  listArray (0, n) $ map fromSample signal
  where
    n = length signal - 1
```

Импортирует функцию listArray для распакованных массивов

Импортирует функции и типы из библиотеки HCodecs

Импортирует тип Int16

Преобразует Signal в массив 16-разрядных отсчетов

В процессе преобразования `Signal` в `SampleData` мы должны следить за тем, чтобы наши значения попадали в диапазон от `-1` до `1`, для чего можно воспользоваться функцией, обрезающей значения. Однако мы не хотим выходить на эти пределы, поэтому нужно будет немного уменьшить уже обрезанный сигнал. С помощью функций `min` и `max` можно зажать сигнал в требуемый диапазон. См. листинг 10.5.

Листинг 10.5 Функция ограничения сигнала

```
limit :: Signal -> Signal
limit = map (min threshold . max (-threshold) . (* threshold))
  where
    threshold = 0.9
```

Здесь мы видим композицию трех операций внутри функции `map`. Отображаемые значения сначала умножаются на пороговую величину `threshold`, а затем зажимаются с помощью `max` и `min`.

Теперь можно собрать функции воедино и создать действие ввода-вывода, которое записывает наш `Signal` в WAV-файл, используя тип и функцию из библиотеки HCodecs. Его код приведен ниже.

Листинг 10.6 Функция для записи сигнала в WAV-файл

```
import Util.Types
import qualified Codec.Wav
import Data.Audio

writeWav :: FilePath -> Signal -> IO ()
writeWav filePath signal = do
  putStrLn $
    "Writing " ++ show (length signal) ++ " samples to " ++ filePath
  let sampleData = signalToSampleData $ limit signal
      audio =
        Audio
          { Data.Audio.sampleRate = round Util.Types.sampleRate,
            channelNumber = 1,
            sampleData = sampleData
          }
  Codec.Wav.exportFile filePath audio
```

Импортирует функции из библиотеки HCodecs

Ограничивает абсолютные значения отсчетов и преобразует сигнал в данные отсчетов, понятные HCodecs

Создает аудиоданные, которые можно экспортировать в виде WAV-файла

Экспортирует аудиоданные в WAV-файл с указанным именем

Воспользуемся этими функциями для создания WAV-файла:

```
ghci> writeWav "4waves.wav" $ concatMap (\w -> tone w 220 5)
➥ [Sound.Synth.sin, tri, saw, sqw]
```

В результате будет создан файл 4waves.wav, содержащий 20-секундное аудио, разбитое на сигналы четырех форм, каждый из которых воспроизводится на частоте 220 Гц в течение 5 секунд. Поскольку наш сигнал – это простой список, мы можем использовать такие функции, как `(++)`, `concat` или `concatMap`, для конкатенации сигналов.

Как выглядит наш звуковой файл, можно посмотреть в любом свободном аудиоредакторе, например *Audacity*. См. рис. 10.3.

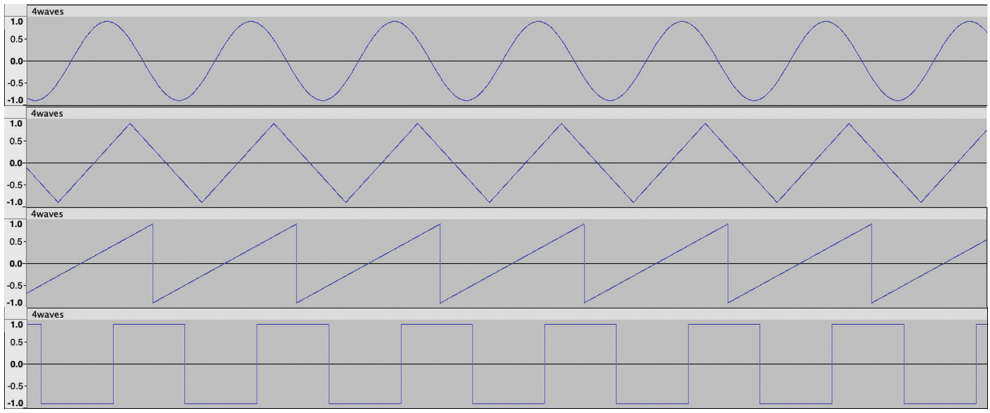


Рис. 10.3 Формы сигналов в экспортированном аудиофайле

Воспользовавшись аудиоплеером (или редактором), мы можем также прослушать файл. Но будьте осторожны: звук очень громкий и пока не слишком приятный.

10.2 Работа с бесконечными списками

Далее мы хотим придать нашим тонам немного благозвучности. Экспериментируя с нашим генератором тонов, мы, наверное, слышали «щелчки», когда тон прекращается. Это из-за того, что наша функция внезапно обрывается по истечении заданного времени. Это хорошо видно на рис. 10.4. Необходимо как-то сгладить тон, особенно в конце сигнала, чтобы избавиться от этого щелчка.

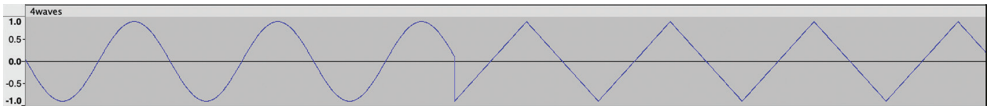


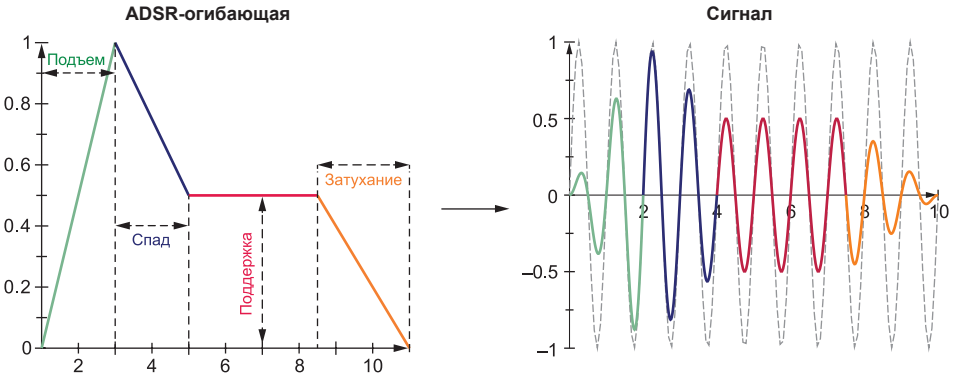
Рис. 10.4 Пример внезапного изменения формы сигнала, вызывающего щелчок при воспроизведении звука

10.2.1 Подъем, спад, поддержание и затухание

В большинстве синтезаторов это достигается с помощью огибающей, которая формирует амплитуду сигнала, зачастую сводя его на нет в начале и в конце. В своем синтезаторе мы хотим реализовать очень распространенную ADSR-огибающую (attack, decay, sustain, release – подъем, спад, поддержание, затухание). Эта огибающая позволяет сигналу некоторое время нарастать (подъем), затем на протяжении некоторого времени он спадает до фиксированной величины, поддерживаемой некоторое время, после чего медленно затухает. Такая огибающая показана на рис. 10.5 вместе с ее воздействием на сигнал.

Теперь возникает новый вопрос: как представить ADSR-огибающую? Поскольку огибающая должна воздействовать на амплитуду нашего сигнала, можно рассматривать ее как список значений, на которые умножается сигнал (см. рис. 10.5). Так что снова решением будет простой тип `[Double]`.

Для представления параметров огибающей мы можем использовать запись. Все участки – подъем, спад, поддержание и затухание – имеют длительность. Их графики линейны, хотя в других синтезаторах они могут быть иными. Впрочем, график на участке поддержания – плато. Как долго длится это плато, зависит от продолжительности модулируемого сигнала и остальных длительностей. Код параметров такой огибающей показан в листинге ниже.



ADSR-огибающая описывает изменение амплитуды сигнала со временем. Она состоит из участков подъема, спада, поддержания и затухания и оказывает влияние на характеристики порождаемого сигнала

Огибающую можно представлять себе как функцию, которая определяет амплитуду сигнала в данный момент. Умножение сигнала на огибающую дает сформированный сигнал с приятными слуху характеристиками

Рис. 10.5 ADSR-огибающая и ее воздействие на сигнал

Листинг 10.7 Тип для представления параметров ADRS-огибающей

```
data ADSR = ADSR
  { attack :: Seconds,
    decay :: Seconds,
    sustain :: Double,
    release :: Seconds
  }
  deriving (Show)
```

Определяет типы параметров ADRS-огибающей

Определяет новый тип данных для ADRS-огибающей

Выводит экземпляр класса типов Show

Теперь можно подумать о том, как применить эти параметры к сигналу. Мы должны сгенерировать список множителей для наших отсчетов от 0 до 1, поэтому можем использовать хорошо известные операции над списками.

Линейные функции, которые нам предстоит сгенерировать, можно вычислить с помощью списковых выражений, поделив каждое значение на максимум:

```
ghci> map (/ 10) [1..10] :: [Double]
[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]
```

Это уже может служить в качестве кривой подъема. Для спада нужно нечто похожее, только линейная функция должна убывать от 1 до заданного значения.

```
ghci> s = 0.5 :: Double
ghci> map (\x -> (x / 10) * (1 - s) + s) [1..10] :: [Double]
[0.55,0.6,0.65,0.7,0.75,0.8,0.85,0.9,0.95,1.0]
```

После обращения этого списка мы будем иметь функцию спада! Однако нужно позаботиться о том, чтобы ее можно было соединить

с функцией подъема. Кривая подъема уже включает значение 1, и повторять его на кривой спада нет никакой необходимости. Чтобы исправить это, начнем нумерацию элементов списка с 0 и закончим желаемой длиной минус 1:

```
ghci> map (\x -> (x / 10) * (1 - s) + s) [0..9] :: [Double]
[0.5,0.55,0.6,0.65,0.7,0.75,0.8,0.85,0.9,0.95]
```

Мы построили участок огибающей, спадающий до поддерживаемого значения `0.5`. А поскольку на участке поддержания значение фиксировано, его можно просто повторить в списке. Но только необходимо посчитать, сколько значений нам нужно, а это определяется количеством отсчетов на участке затухания, который еще предстоит добавить. Мы могли бы повторно применить логику, использованную при построении кривой подъема, к построению кривой затухания, а затем модифицировать участок поддержания. Но как объединить эти кривые?

10.2.2 Построение бесконечных списков и работа с ними

Возвращаясь к функциям, которые мы использовали для списков, можно вспомнить о функции `zipWith`, которая сшивает два списка с помощью указанной функции, останавливаясь по достижении конца *более короткого* списка. Есть также функция `zipWith3`, работающая с тремя, а не с двумя списками. Мы можем трактовать подъем, спад и поддержание как один список, а затухание – как другой. Их следует объединить с сигналом. Нам осталось только построить кривую затухания. Конечно, мы могли бы проделать кучу вычислений, связанных с длиной сигнала и количеством значений на каждом участке, но процесс можно упростить, воспользовавшись *бесконечными списками*.

Ленивость Haskell имеет интересное последствие – можно создавать бесконечные определения, вычисление которых не бесконечно. Определение списков могут быть бесконечно рекурсивно, но это не вызывает никаких проблем, потому что функции не обязаны вычислять все значения (каковых бесконечно много):

```
ghci> ones = 1 : ones :: [Int]
ghci> take 10 ones
[1,1,1,1,1,1,1,1,1,1]
```

При условии что используются функции, которые не вычисляют весь список, например `take`, `takeWhile`, `(!!)`, `head` и другие, мы можем спокойно работать с такими списками и даже отображать, фильтровать и сшивать их. Это распространяется и на другие типы данных, которые можно генерировать бесконечно, например на графы.

Что касается списков, то мы можем легко использовать знакомые выражения диапазона для создания бесконечных списков, просто опустив максимальное значение. Таким образом, не составляет никакого труда сгенерировать все натуральные или нечетные числа с помощью очень лаконичных выражений:

```
ghci> take 10 [1..] :: [Int]
[1,2,3,4,5,6,7,8,9,10]
ghci> take 10 [1,3..] :: [Int]
[1,3,5,7,9,11,13,15,17,19]
```

> **ПРЕДУПРЕЖДЕНИЕ** Бесконечные структуры данных – это круто, но *опасно*. Следует внимательно следить за тем, чтобы не вызывались функции, которые пытаются вычислить их целиком. Правила работы с данными, которые мы выработали для себя, могут быть нарушены, причинив немало горя, если нет гарантии, что данные конечны. Если вызвать `length` для бесконечного списка, то она будет работать вечно и, стало быть, подвесит нашу программу!

Нам нужно повторить некоторые фиксированные значения (1 в начале кривой затухания и на участке поддержания), для чего можно воспользоваться удачно названной функцией `repeat`, которая возвращает бесконечный список, содержащий единственное значение:

```
ghci> take 10 $ repeat 1
[1,1,1,1,1,1,1,1,1,1]
```

Мы можем использовать тот факт, что `zipWith` останавливается после исчерпания *более короткого списка*, для сшивки бесконечных списков с конечными:

```
ghci> zipWith (+) [1,2,3] (repeat 1)
[2,3,4]
```

Возвращаясь к использованию огибающей, мы можем использовать один список, который объединяет кривые подъема и спада, а затем бесконечно повторяет плато на участке поддержания. Для второго списка мы можем построить кривую затухания так же, как кривую подъема, и повторить 1, взяв из кривой ровно столько элементов, сколько необходимо для сигнала, и обратив их. Перемножение этих кривых с исходным сигналом даст желаемый результат.

Упражнение. Циклический обход списка

Функция `repeat` создает бесконечный список, содержащий единственное значение. А функция `cycle` принимает список в качестве аргумента и обходит его, снова и снова возвращаясь из конца в начало. В нашей функции `tone` мы вручную вычисляли повторяющуюся форму сигнала, применив `mod`. Теперь, используя `cycle` и `take`, перепишите функцию `tone`, вычислив форму сигнала один раз, а затем организовав циклический обход.

Реализация этой огибающей показана в листинге ниже. Функция вычисляет и применяет кривую к сигналу, переданному в качестве аргумента. Использование `zipWith3` гарантирует, что действие распространится на весь сигнал, потому что участок поддержания бесконечно длинный, а длина участка затухания совпадает с длиной сигнала. Таким образом, эта функция правильна по построению.

Листинг 10.8 Функция, применяющая ASDR-огибающую к сигналу

```
import Util.Types

adsr :: ADSR -> Signal -> Signal
adsr (ADSR a d s r) signal =
  zipWith3
    (\adsCurve rCurve sample ->
➥ adsCurve * rCurve * sample)
    (att ++ dec ++ sus)
    rel
    signal
  where
    attackSamples = fromIntegral $ samplesPerSecond a
    decaySamples = fromIntegral $ samplesPerSecond d
    releaseSamples = fromIntegral $ samplesPerSecond r

    att = map (/ attackSamples) [0.0 .. attackSamples]
    dec =
      reverse $
        map
          (\x -> ((x / decaySamples) * (1 - s)) + s)
          [0.0 .. decaySamples - 1]
    sus = repeat s
    rel =
      reverse $
        take
          (length signal)
          (map (/ releaseSamples)
➥ [0.0 .. releaseSamples] ++ repeat 1.0)
```

Умножает комбинацию кривых подъема, спада и затухания на плато поддержания и на сигнал

Конкатенирует кривые подъема и спада с плато поддержания

Вычисляет, сколько отсчетов необходимо для каждой кривой

Вычисляет кривую подъема как линейную функцию

Обращает линейную функцию спада

Вычисляет кривую спада как линейную функцию, изменяющуюся от уровня поддержания до 1

Генерирует бесконечный список отсчетов на плато поддержания

Обращает кривую затухания

Берет столько значений из кривой затухания, сколько отсчетов в сигнале

Вычисляет кривую затухания как линейную функцию, изменяющуюся от 0 до 1, и бесконечное число 1

Теперь мы можем применить эту огибающую к нашему тону и впервые услышать, как он звучит:

```
ghci> params = ADSR 0.1 0.2 0.5 0.1
ghci> writeWav "adsrSignal.wav" $ adsr params (tone tri 550 1)
```

При открытии результирующего файла в аудиоредакторе мы увидим форму сигнала. Участок линейного подъема достигает полной громкости за 100 мс. Затем сигнал линейно спадает на протяжении 200 мс до уровня поддержания, равного половине максимальной громкости (`0.5`). На участке затухания громкость сигнала падает до 0 за 100 мс. Все это показано на рис. 10.6.

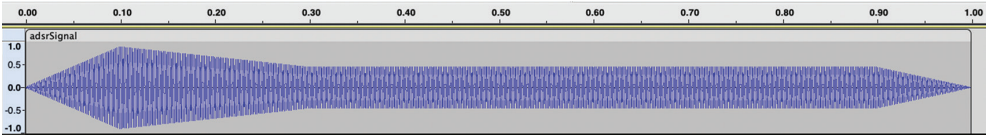


Рис. 10.6 Сигнал после применения ADSR-огибающей

Пользуясь такой огибающей, мы можем формировать тона и более радикально. Можно создавать «щипковые» звуки, используя быстрый подъем, спад и затухание и оставив уровень поддержания очень низким. Можно создавать «нарастающие» звуки, продлевая участки подъема и спада. Разобравшись с генерированием тона и его оконтуриванием, мы можем приступить к созданию источников шума, которые в мире синтезаторов называются генераторами.

10.3 Управление синтезом

У синтезаторов всего одна задача: принимать управляющие сигналы (с клавиатуры, от электронной аппаратуры или из цифрового последовательного порта) и порождать соответствующие звуки. Мы также хотели бы управлять нашим шумом, а именно высотой тона, длительностью и моментом начала звучания – для создания музыки.

10.3.1 Селекторы частичных полей

Для этого мы будем кодировать события генераторов тона. Событие – это либо воспроизведение тона в конкретный момент времени в течение заданного промежутка на заданной частоте, либо тишина, которая также имеет начало и продолжительность. Эти события кодируются типом с применением синтаксиса записей. Затем можно определить функции, которые отличают тона от тишины и вычисляют момент конца события. Код приведен в листинге ниже.

Листинг 10.9 Функция применения ADSR-огибающей к сигналу

```
module Composition.Performance where

import Util.Types

data Event
  = Tone {freq :: Hz, start :: Seconds,
➥ duration :: Seconds}
  | Silence {start :: Seconds, duration :: Seconds}
  deriving (Show)

isTone :: Event -> Bool
isTone Tone {} = True
isTone _ = False

isSilence :: Event -> Bool
isSilence Silence {} = True
isSilence _ = False

end :: Event -> Seconds
end e = start e + duration e
```

Определяет конструктор тонов с заданными частотой, временем начала и длительностью

Определяет конструктор тишины с заданными временем начала и длительностью

Выводит экземпляр класса типов Show для типа Event

Определяет вспомогательную функцию, определяющую, представляет ли Event тон

Определяет вспомогательную функцию, определяющую, представляет ли Event тишину

Определяет вспомогательную функцию, вычисляющую время конца события

В этом типе заслуживает внимания использование полей. В главе 7 мы видели, что синтаксис записи позволяет по заданному полю записи автоматически создать функцию, которая извлекает значение этого поля. Так же обстоит дело, когда в нашем типе имеется несколько конструкторов, использующих синтаксис записи:

```
ghci> :t start
start :: Event -> Seconds
ghci> :t duration
duration :: Event -> Seconds
ghci> :t freq
freq :: Event -> Hz
```

Если одно и то же поле используется в нескольких конструкторах, то сгенерированные функции, называемые *селекторами полей*, способны разобраться и извлечь нужное поле. Именно поэтому в определении функции `end` допустимо использовать `start` и `duration`:

```
ghci> start $ Tone 0 1 2
1.0
ghci> start $ Silence 1 2
1.0
```

Однако поле `freq` особое, потому что встречается не во всех конструкторах. Это делает его *селектором частичного поля*:

```
ghci> freq $ Tone 440 1 2
440.0
ghci> freq $ Silence 1 2
*** Exception: No match in record selector freq
```

По сути дела, селекторы частичных полей являются частичными функциями, т. е. функциями, которые определены не для всех значений. Поэтому использовать их опасно, и лучше бы держаться от них подальше.

> **ПРИМЕЧАНИЕ** При наличии флага компиляции `-Wpartial-field` GHC будет автоматически предупреждать о наличии селекторов частичных полей. Такого рода предупреждения всегда полезны. Бывает так, что вы не собирались заводить частичное поле, а сделали это непреднамеренно в ходе рефакторинга полей другой записи, и тогда предупреждение поможет вам вернуться на путь истинный!

Мы не хотим раскрывать миру частичные функции. И как решить эту проблему? Один из способов – воспользоваться расширением языка `NoFieldSelectors`, которое подавляет автоматическое создание функций для полей, определенных в модуле записей. Это дает возможность определять собственные функции для доступа к полям:

```
{-# LANGUAGE NoFieldSelectors #-}

module Composition.Performance
  ( Event (..),
    start,
    end,
  )
where

...

data Event
  = Tone {freq :: Hz, start :: Seconds, duration :: Seconds}
  | Silence {start :: Seconds, duration :: Seconds}
  deriving (Show)

start :: Event -> Seconds
start (Tone _ s _) = s
start (Silence s _) = s

duration :: Event -> Seconds
duration (Tone _ _ d) = d
duration (Silence _ d) = d
```

Активирует расширение языка NoFieldSelectors

Экспортирует тип Event и его конструкторы

Экспортирует наши собственные функции селекции полей, определенные на верхнем уровне

Определяет функции для доступа к полям записи Event

Теперь наш модуль экспортирует функции `start` и `duration`, работающие для обоих конструкторов. А опасный селектор частичного поля скрыт.

10.3.2 Функция как тип

Располагая определением событий, мы можем подумать о создании компонентов, которые порождают сигналы на их основе. В своем коде мы хотим назвать их `Oscillator` из-за связи с генераторами в музыкальных синтезаторах. По большей части генераторы имеют предопределенную форму сигнала и напрямую связаны с какой-то огибающей для оконтуривания. Наши генераторы имеют тип `newtype`, который обертывает функцию `Event -> Signal` новым типом. Теперь можно построить функцию, которая комбинирует функцию `Wave` с параметрами `ADSR` для создания генератора тонов по событиям. Этот код показан в листинге ниже.

Листинг 10.10 Функция определения генераторов

```
module Sound.Synth (
  ...
) where

import Composition.Performance
...

newtype Oscillator = Osc {playEvent :: Event -> Signal}

osc :: Wave -> ADSR -> Oscillator
osc wave adsrParams = Osc oscf
  where
    oscf (Silence _ t) = silence t
    oscf (Tone f _ t) = adsr adsrParams $ tone wave f t
```

Обертывает созданную функцию конструктором Osc

Определяет тип генераторов с селектором поля

Создает тишину заданной длительности

Создает сигнал, содержащий тон заданной частоты и заданной длительности, и оконтуривает его с помощью ADSR-огибающей

Как видите, параметр `start` наших событий игнорируется. Это объясняется тем, что мы не хотим думать о том, как сигналы комбинируются внутри генератора. В конце концов, синтезатор может иметь несколько генераторов, воспроизводящих свои тона, иногда в разное время, а иногда с перекрытием. Чтобы правильно обработать все это, нам впоследствии понадобится какая-то система управления, но пока можно начать с построения генераторов со своими собственными характеристиками, определяемыми формой сигнала и параметрами ADSR. В листинге ниже мы видим три кандидата на роль генераторов шума. Можете поэкспериментировать с формами сигналов и параметрами ADSR и создать свои собственные музыкальные шкатулки.

Листинг 10.11 Функция определения генераторов

```
module Sound.Synth (
  ...
) where

import Prelude hiding (sin)
import qualified Prelude

...

piano :: Oscillator
piano = osc saw $ ADSR 0.01 0.6 0.3 0.2

ocarina :: Oscillator
ocarina = osc sin $ ADSR 0.01 0.3 0.7 0.01

violin :: Oscillator
violin = osc saw $ ADSR 2 2 0.1 0.2

pluck :: Oscillator
pluck = osc sqw $ ADSR 0.01 0.05 0.0 0.01

bass :: Oscillator
bass = osc tri $ ADSR 0.001 0.2 0.9 0.1
```

Определяет острый генератор с быстрым нарастанием амплитуды и длинным участком поддержания

Определяет генератор наполненного звука с почти постоянной громкостью

Определяет острый генератор с длительным подъемом и спадом

Определяет щипковый генератор без участка поддержания

Определяет генератор спокойного и громкого звука, который можно использовать для создания глубоких тонов

Используя селектор полей в нашем типе `Oscillator`, мы можем испытать эти генераторы:

```
ghci> oscs = [piano, ocarina, violin, pluck, bass]
ghci> signal = concatMap (`playEvent` (Tone 440 0 1)) oscs
ghci> writeWav "oscillators.wav" signal
```

Мы видим очень заметное изменение звучания при смене формы сигнала или его оконтурирования. На этом наша работа по созданию части синтезатора, связанной с генерированием звука, окончена.

Упражнение. Эффект тремоло

Мы реализовали базовую функциональность генерирования звука. Настоящие синтезаторы предлагают гораздо большую гибкость в плане формирования звуков, реализации фильтров и других типов огибающих. Расширьте возможности генератора, добавив модуляцию, когда другой сигнал (низкой частоты 0–10 Гц) оказывает влияние на какую-то характеристику нашего генератора. Если эта характеристика – громкость, то мы получаем *эффект тремоло*. Реализуйте этот эффект, чтобы придать генерируемому звуку большую глубину.

Располагая генератором шумов, мы можем приступить к сочинению музыки: как моделировать и группировать ноты и как воспроизводить наши композиции.

10.4 Нотные модели

Если мы хотим создавать музыкальные композиции, то должны уметь классифицировать высоту и длительность звука относительно какой-то глобальной системы измерения времени. Мы не собираемся превращать эту главу в скучную лекцию по теории музыки, а потому опустим большую часть таких деталей, но вопрос о том, как моделировать теорию музыки, будет нам интересен.

10.4.1 Класс типов для высоты звука

Сначала займемся высотой. Мы уже имеем способ дискретизации высоты для нашего типа `Oscillator`, это поле `Hz`, частота сигнала. Однако Бетховен и Моцарт в своих сочинениях использовали не частоты, а музыкальные ноты. Какова связь между ними и тем, что умеет создавать наш `Oscillator`? Это мы и намереваемся выяснить.

Мы понимаем, что в какой-то момент нам нужно будет преобразовать ноты в частоты. Для этого создадим класс типов, показанный в листинге ниже.

Листинг 10.12 Класс типов, которые можно преобразовать в частоту

```
{-# LANGUAGE TypeSynonymInstances #-}
module Composition.Pitch where

class Pitchable a where
  toFrequency :: a -> Hz

instance Pitchable Hz where
  toFrequency = id
```

Определяет класс типов, допускающих преобразование в частоту

Создает реализацию функции toFrequency для типа Hz

Класс состоит всего из одной функции, которая умеет преобразовывать тип в частоту. Первый экземпляр этого класса уже имеется – для самого типа `Hz`. Конечно, частоту можно преобразовать в частоту, не делая вообще ничего, поэтому в данном случае реализацией `toFrequency` является тождественная функция.

На клавиатуре (фортепьянной, а не компьютерной) музыкальные ноты разделены полутонами. Это показано на рис. 10.7. Перемещая палец с одной клавиши на соседнюю, мы изменяем звучащую ноту на один полутон. Имеет смысл создать тип для полутонов, но как он должен выглядеть?

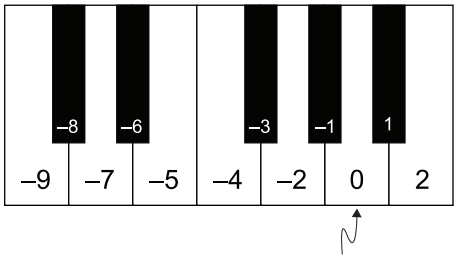


Рис. 10.7 Музыкальная клавиатура с расстояниями от концертного ля, выраженными в полутонах

Большинство музыкальных произведений, исполняемых сегодня, следует одному и тому же строю. Этот строй определяет концертное ля как частоту 440 Гц. Если полутона определяются расстоянием от концертного ля, то для вычисления частоты ноты можно пользоваться формулой на рис. 10.8.

$$f = 440\,Hz * 2^{s/12}$$

Рис. 10.8 Формула вычисления частоты (*f*) по заданному расстоянию в полутонах от концертного ля (*s*) в 12-тоновой равномерной темперации

Это подводит нас к определению полутонов для нашего синтезатора. Полутон – это просто значение типа `Integer`, задающее расстояние до концертного ля. Вычисление, необходимое для создания экземпляра `Pitchable Semitone`, показано на рис. 10.8. А соответствующий код приведен в листинге ниже.

Листинг 10.13 Тип для полутонов

```
{-# LANGUAGE TypeSynonymInstances #-}
module Composition.Pitch where

...

type Semitone = Integer

instance Pitchable Semitone where
  toFrequency semitone =
    440 * (2.0 ** (fromInteger semitone / 12))
```

Активирует расширение языка TypeSynonymInstances, разрешающее экземпляры классов для синонимов типов

Создает синоним типа для полутонов

Вычисляет частоту ноты, отстоящей от концертного ля на заданное число полутонов

Теперь можно и повычислять. Ожидается, что когда мы отступаем на одну октаву вверх или вниз от заданного полутона, частота должна соответственно увеличиться или уменьшиться вдвое:

```
ghci> toFrequency (0 :: Semitone)
440.0
ghci> toFrequency (12 :: Semitone)
880.0
```

```
ghci> toFrequency (-12 :: Semitone)
220.0
ghci> toFrequency (5 :: Semitone)
587.3295358348151
ghci> toFrequency (5+12 :: Semitone) / 2
587.3295358348151
```

Итак, мы составили первое представление о задании тонов и о том, как связаны между собой их высоты. Однако Бетховен и Моцарт в своих сочинениях использовали не полутона, а *музыкальные ноты*. Пора заняться и этой, последней, проблемой.

Упражнение. Тест для полутонов

Итак, прибавление или вычитание 12 полутонов должно приводить к увеличению или уменьшению частоты вдвое, верно? Так это же прямо-таки формальное свойство! Напишите с помощью QuickCheck тест вычислений частоты в типе `Semitone`. Какая проблема возникает при конструировании этого теста? Как ее решить?

Открою маленький секрет: ноты – не что иное, как причудливые названия полутонов. *Концертное ля* называется также *A4*, где *A* – название полутона в октаве (ля), а *4* – номер октавы. Количество октав и полутонов ограничено, потому что человеческое ухо способно воспринимать звуки примерно в диапазоне от 20 Гц до 20 кГц. Все, что меньше 20 Гц, воспринимается не как высота звука, а как ритм, а большую часть частот выше 20 кГц мы не воспринимаем, хотя их воспринимают наши домашние питомцы.

Но вернемся к нотам. Полутонам в октаве на клавиатуре соответствуют названия нот. Ноты, разделенные интервалом в полтона, называются также хроматическими. Их названия показаны на рис. 10.9.

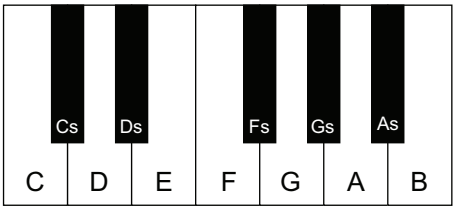


Рис. 10.9. Музыкальная клавиатура и названия нот[1]

[1] Соответствие английских и русских названий нот: C – до, D – ре, E – ми, F – фа, G – соль, A – ля, B – си. Соответственно, Cs, Ds ... – до-диез, ре-диез и т. д. – *Прим. перев.*

Как их представить? Теоретически можно было бы перечислить *все* возможные значения в гигантском помеченном объединении:

```
data Chromatic
  = C0 | Cs0 | D0 | Ds0 | E0 ...
  | C1 | Cs1 | D1 ...
  ...
  | C8 | Cs8 ...
```

Однако его не только трудно выписать, но и сопоставление с таким образцом выполнять проблематично, поэтому мы хотим представлять ноту номером октавы и смещением от начала этой октавы в полутонах (этому смещению и сопоставляется название ноты). Итак, имеет смысл завести два типа: для названий хроматических нот и самих хроматических нот. Число, ассоциированное с хроматической нотой, не может быть отрицательным, поэтому мы воспользуемся типом `Natural` из модуля `Numeric.Natural`. Этот тип кодирует натуральные, т. е. неотрицательные, числа. Код нового типа данных приведен в следующем листинге.

Листинг 10.14 Типы для хроматических нот

```
module Composition.Pitch where

...

import Numeric.Natural (Natural)

...

data ChromaticName = A | As | B | C | Cs | D | Ds | E | F | Fs | G | Gs
  deriving (Show, Eq)

data Chromatic
  = Chromatic ChromaticName Natural
  deriving (Show, Eq)
```

Импортирует определение типа Natural

Определяет имена хроматических нот в виде помеченного объединения

Выводит экземпляры различных классов типов для типа ChromaticName

Определяет тип для хроматических нот

Выводит экземпляры классов типов Show и Eq для типа Chromatic

Увы, запись значений этого типа получается довольно громоздкой. Мы можем определить вспомогательные функции для каждого возможного названия ноты, которые требуют только задания октавы, которой принадлежит нота:

```
a :: Natural -> Chromatic
a = Chromatic A

as :: Natural -> Chromatic
as = Chromatic As

...
```

```
gs :: Natural -> Chromatic
gs = Chromatic Gs
```

И последний шаг – преобразовать эти ноты в частоты. Поскольку это не что иное, как альтернативные названия полутонов, а как вычислять частоты, соответствующие полутонам, мы уже знаем, перед нами стоит задача преобразовать хроматические ноты в полутона. Мы знаем, что `A4` должен соответствовать `0` и что отклонение от `A` на одну ноту равно одному полутону. Следовательно, для вычисления полутона нужно прибавить смещение ноты от начала октавы, умноженное на номер октавы, минус 4 × 12 (потому что в октаве 12 полутонов). Смещения легко перечислить. И мы можем использовать это для вычисления частоты, как показано в листинге ниже.

Листинг 10.15 Преобразование хроматических нот в полутона и частоты

```
chromaticToSemitone :: Chromatic -> Semitone
chromaticToSemitone (Chromatic name oct) =
  (12 * (fromIntegral oct - 4)) + noteOffset name
  where
    noteOffset C = -9
    noteOffset Cs = -8
    noteOffset D = -7
    noteOffset Ds = -6
    noteOffset E = -5
    noteOffset F = -4
    noteOffset Fs = -3
    noteOffset G = -2
    noteOffset Gs = -1
    noteOffset A = 0
    noteOffset As = 1
    noteOffset B = 2

  instance Pitchable Chromatic where
    toFrequency = toFrequency . chromaticToSemitone
```

Вычисление Semitone по хроматической ноте путем прибавления смещения ноты в пределах октавы к номеру полутона, соответствующему началу октавы

Определяет смещения, соответствующие названиям хроматических нот

Вычисляет частоту по хроматической ноте, преобразуя ее в значение типа Semitone и используя реализацию toFrequency в этом типе

Как и раньше, мы можем проверить справедливость нашего свойства частот и октав для этого нового типа:

```
ghci> toFrequency (a 4)
440.0
ghci> toFrequency (a 5)
880.0
ghci> toFrequency (a 3)
220.0
ghci> toFrequency (c 2)
65.40639132514966
ghci> toFrequency (c 3) / 2
65.40639132514966
```

Используя эти типы в качестве строительных блоков, мы можем создать свои первые мелодии:

```
ghci> melody = map toFrequency [c 4, e 4, g 4, d 5, c 5 :: Chromatic]
ghci> signal = concatMap (\f -> playEvent piano $ Tone f 0 0.7) melody
ghci> writeWav "melody.wav" signal
```

Все замечательно, только как-то монотонно. Каждая нота звучит одно и то же время. Мы могли бы варьировать длительность нот, но у нас нет элегантного способа это записать.

Упражнение. Свойства для различных высот

Напишите свойства QuickCheck для этого нового типа, реализовав также экземпляр `Arbitrary Chromatic`, и проверьте, правильно ли работает преобразование в `Semitone` и в `Hz`.

Чего не хватает в нашей композиции, так это возможности указывать длительность ноты, и этим мы займемся далее.

10.4.2 Tun Ratio

Музыкальные ноты характеризуются длительностью. Длительность ноты определяет, какую часть такта занимает нота. Целая нота занимает целый такт, половинная нота – половину такта, четвертная нота – четверть такта и т. д. Длительности всегда положительны, потому что «отрицательная часть» не имеет смысла. Вопрос: как представить их в коде?

Хотя для представления длительностей можно было бы использовать тип `Double` или `Float`, можно принять куда более явный подход и воспользоваться типом `Ratio` из модуля `Data.Ratio`. Дробь (ratio) – именно то, что нам надо; она состоит из числителя и знаменателя. Это значит, что дроби можно представить с произвольной точностью, если тип числителя и знаменателя может быть бесконечно большим. Кроме того, `Ratio` реализует класс типов `Num`, поэтому дроби можно использовать в вычислениях.

При работе с типом `Ratio` основным оператором является `(%)`, который определяет рациональное число с помощью числителя и знаменателя:

```
ghci> import Data.Ratio
ghci> :k Ratio
Ratio :: * -> *
ghci> 1 % 2 :: Ratio Int
1 % 2
ghci> 3 % 2 + 1 % 2 :: Ratio Int
```

```
2 % 1
ghci> 4 % 5 - 2 % 10 :: Ratio Int
3 % 5
```

Как видим, родом `Ratio` является `* -> *`, т. е. он параметризуется другим типом. Оператор `(%)` гарантирует, что можно использовать только целые типы. Поэтому можно сконструировать `Ratio Int` и `Ratio Integer`, но не `Ratio Double`:

```
ghci> :t (%)
(%) :: Integral a => a -> a -> Ratio a
```

Мы уже встречались с типом `Ratio`, когда говорили о функции `toRational`: тип `Rational` – не что иное, как псевдоним `Ratio Integer`:

```
ghci> :i Rational
type Rational :: *
type Rational = Ratio Integer
```

Нам нужно запретить отрицательные значения на уровне типа, поэтому имеет смысл использовать тип `Natural`, который, на наше счастье, тоже имеет экземпляр класса типов `Integral`. Пользуясь этим, мы можем сконструировать много длительностей нот, известных в западной теории музыки.

> **ВАЖНО** Тип `Natural` кажется разумным выбором, когда нужно представить неотрицательные числовые значения, но не всегда это так. Если в процессе вычислений с числами типа `Natural` возникнет отрицательное значение (а это возможно, потому что существует экземпляр `Num Natural`, и, значит, такие числа можно вычитать), то будет возбуждено исключение. Это может привести к краху программы в самый неподходящий момент!

Типы для длительностей нот, а также некоторые константы приведены в следующем листинге.

Листинг 10.16 Преобразование хроматических нот в полутона и частоты

```
module Composition.Notelength where

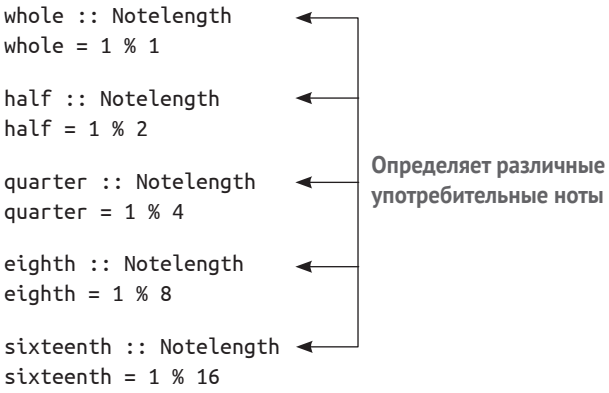
import Data.Ratio (Ratio, (%))
import Numeric.Natural (Natural)

type Notelength = Ratio Natural
```

Импортирует тип Ratio и оператор (%) из модуля Data.Ratio

Импортирует тип Natural из модуля Numeric.Natural

Определяет тип для длительностей нот в виде отношения натуральных чисел



Однако в западной музыкальной нотации существуют еще и модификаторы длительностей нот. Из них нас интересуют *ноты с точкой* и *мультиоли*. Ноты с точкой можно считать продлением оригинальной ноты. Если рядом с нотой стоит точка, то ее длительность увеличивается в полтора раза. Точек может быть две и даже три. Мультиоли интересны тем, что позволяют создавать рваные ритмы, отличающиеся от нормальных длительностей. Доли для нот, записанных в виде мультиоля, распределяются по-разному. Они также параметризуются некоторой константой; три для «триолей» и пять для «квинтолей» – наиболее распространенные значения. Длительности нот умножаются на два, поделенное на значение мультиоля, т. е. для триолей на две трети, а для квинтолей на две пятых.

Человек не способен интерпретировать и исполнить ноты слишком сложной длительности, но у нашего синтезатора нет этих жалких биологических ограничений. Поэтому мы разрешим произвольное число точек и любые мультиоли. Реализовать модификаторы можно с помощью простых функций, показанных в листинге ниже.

Листинг 10.17 Преобразование хроматических нот в полутона и частоты

```
dots :: Natural -> Notelength -> Notelength
dots n x = x + x * (1 % 2 ^ n)

dotted :: Notelength -> Notelength
dotted = dots 1

doubleDotted :: Notelength -> Notelength
doubleDotted = dots 2

tripleDotted :: Notelength -> Notelength
tripleDotted = dots 3

tuplet :: Natural -> Notelength -> Notelength
tuplet n x = x * (2 % n)
```

```
triplet :: Notelength -> Notelength
triplet = tuplet 3

quintuplet :: Notelength -> Notelength
quintuplet = tuplet 5
```

Теперь мы можем модифицировать длительности нот, что позволит создавать гораздо более интересные ритмические рисунки.

```
ghci> nl = 1 % 2 :: Notelength
ghci> dotted nl
3 % 4
ghci> triplet nl
1 % 3
```

10.4.3 Различные способы возведения в степень

В листинге 10.17 стоит обратить внимание на использование оператора (^). Ранее мы использовали для вычисления частот оператор (**), но теперь используем (^), хотя тот и другой обозначают возведение в степень. А если вы еще недостаточно запутались, то имейте в виду, что есть и третий оператор возведения в степень, выглядящий как смайлик счастья: (^^). Да что ж это такое? Зачем так много? А у каждого из них есть только ему присущая особинка. Для начала получим информацию об их типах:

```
ghci> :t (**)
(**) :: Floating a => a -> a -> a
ghci> :t (^)
(^) :: (Num a, Integral b) => a -> b -> a
ghci> :t (^^)
(^^) :: (Fractional a, Integral b) => a -> b -> a
```

Из одних лишь сигнатур мы видим, что (**) следует использовать, когда показатель степени – число с плавающей точкой, (^) и (^^) – когда показатель степени целый, однако только (^^) можно использовать, когда основание дробное. Но самое главное отличие – точность и допустимые значения показателя степени. (^) не допускает отрицательных показателей. Для целых показателей лучше использовать (^) и (^^), поскольку они дают большую точность, но только не для отрицательных показателей:

```
ghci> (1.2 :: Double) ** (5 :: Double)
2.4883199999999994
ghci> (1.2 :: Double) ^ (5 :: Int)
2.48832
ghci> (1.2 :: Double) ^^ (5 :: Int)
2.48832
ghci> (1.2 :: Double) ^ (-5 :: Int)
```

```
*** Exception: Negative exponent
ghci> (1.2 :: Double) ^^ (-5 :: Int)
0.4018775720164609
ghci> (1.2 :: Double) ** (-5 :: Double)
0.40187757201646096
```

Как видите, все донельзя запутано, и правильный выбор оператора не всегда очевиден. Вообще говоря, `(^)` используется для целых, `(^^)` для рациональных, а `(**)` для чисел с плавающей точкой, но все же возможны исключения, когда показатели степени отрицательны.

Теперь, когда мы неплохо освоились с длительностями нот, нужно сделать последний шаг и связать эти значения с нашим вторым типом `Seconds`. В западной теории музыкальной композиции обычно указываются размер и что-то вроде числа ударов в минуту (BPM). Размер обычно описывает, как делить и подсчитывать такты, а BPM дает точную меру времени. Но знать число ударов в минуту недостаточно, нужно еще знать количество ударов в целой ноте.

Мы не хотим иметь дело с размером, т. к. он несуществен для сочинения музыки (мы же пишем музыку не на листе бумаги), поэтому нам нужно поддержать только понятие BPM и количество ударов в целой ноте. Эту информацию можно представить отдельным типом, как показано в листинге ниже.

Листинг 10.18 Типы для информации о темпе

```
type BPM = Double
```

Определяет тип BPM как Double

```
data TempoInfo = TempoInfo
  { beatsPerMinute :: BPM,
    beatsPerWholeNote :: Double
  }
```

Определяет тип, который описывает количество ударов в минуту и количество ударов в целой ноте

Имея тип `TempoInfo`, мы наконец можем навести мост между реальным миром (измеряемым в минутах) и нашими композициями (целыми нотами). Для этого мы можем определить, сколько ударов укладывается в конкретной длительности ноты (являющейся частью целой ноты), сколько секунд занимает удар и сколько секунд занимает нота конкретной длительности. Код показан в листинге ниже.

Листинг 10.19 Функции для преобразования длительностей нот в единицы времени

```
module Composition.Notelength where

import Util.Types (Seconds)
import Data.Ratio (Ratio, (%), denominator, numerator)
```

```
...

timePerBeat :: BPM -> Seconds
timePerBeat bpm = 60.0 / bpm

timePerNotelength :: TempoInfo -> Notelength -> Seconds
timePerNotelength
  (TempoInfo beatsPerMinute beatsPerWholeNote)
  noteLength =
  let beatsForNoteLength = beatsPerWholeNote *
➥ toDouble noteLength
   in beatsForNoteLength * timePerBeat beatsPerMinute
  where
    toDouble :: Ratio Natural -> Double
    toDouble r =
      (fromInteger . toInteger $ numerator r)
        / (fromInteger . toInteger $ denominator r)
```

Вычисляет число секунд в ударе для заданного значения BPM

Вычисляет количество ударов в заданной длительности ноты

Вычисляет количество секунд в заданной длительности ноты

Преобразует Ratio Natural в Double путем преобразования числителя и знаменателя в Double и их деления друг на друга

Здесь мы используем функции `numerator` и `denominator`, импортированные из `Data.Ratio`, чтобы получить доступ к числителю и знаменателю значения типа `Ratio`. Для преобразования `Natural` в `Double` мы сначала должны преобразовать значение в `Integer`, а затем `Integer` в `Double`.

Включив в каркас композиции высоту и темп, мы можем приступить к приданию формы нашим композициям, разработав собственную структуру для соотнесения нот и пауз. Этим мы займемся в следующей главе.

Резюме

- Классы типов `Integral` и `Fractional` определяют функции для деления целых чисел и чисел с плавающей точкой.
- Класс типов `Enum` определяет, как создавать перечисления значений типов, и используется в синтаксисе диапазона списка.
- Функции `round`, `ceiling` и `floor` из класса типов `RealFrac` используются для преобразования чисел с плавающей точкой в целые, а функция `fromIntegral` – для преобразования целых в числа с плавающей точкой.
- Можно лениво строить списки с бесконечным числом элементов, которые создаются только в момент доступа к ним.
- С бесконечными списками следует обращаться аккуратно, чтобы ни в коем случае не вычислять их целиком.

- Наличие нескольких конструкторов в типе записи может стать причиной появления селекторов частичных полей, если некоторое поле присутствует не во всех конструкторах.
- Расширение языка `NoFieldSelectors` позволяет подавить автоматическое создание селекторов полей.
- Тип `Ratio` определяет отношение чисел.
- При возведении в степень оператор `(^)` используется для целых, `(^^)` – для рациональных чисел, а `(**)` – для чисел с плавающей точкой.

11 Программирование музыкальных композиций

Краткое содержание этой главы:

- построение полиморфной структуры данных, содержащей данные разных типов;
- интерпретация сложных структур данных для вычисления дополнительной информации;
- реализация предметно-ориентированного языка на Haskell.

В предыдущей главе мы заложили основы музыкального синтезатора. Мы представили типы для абстрагирования аудиосигналов и информации о высоте звука. С помощью библиотеки HCodec мы смогли записать аудиоданные в файл.

Оставив технические детали позади, мы теперь поговорим о сочинении музыки внутри программы. Очевидно, мы хотим записывать ноты не на листе бумаги, а прямо в программе. По сути дела, откомпилированная программа на Haskell становится композицией, включающей синтезатор для ее воспроизведения! Можно считать, что это наша личная цифровая музыкальная шкатулка.

В этой главе мы соберем эту шкатулку, спроектировав тип для представления композиций. По ходу дела мы научимся конструировать неоднородные структуры данных, объединяющие данные разных типов. Определив музыкальные структуры и способ их интерпретации, мы тем самым спроектируем предметно-ориентированный язык музыкальных композиций, написанный на Haskell!

11.1 Полиморфные структуры данных с несколькими типами

Итак, как представлять композиции в программе? В музыке мы различаем ноты и паузы. Инструмент либо воспроизводит ноты, либо молчит на протяжении некоторого промежутка времени. Ноты и паузы собираются либо в последовательность, либо в группы.

На рис. 11.1 показана мелодия – последовательность нот, а на рис. 11.2 – аккорд – группа нот, звучащих одновременно. Однако последовательности нот также могут находиться в составе последовательности или группы. И группы можно выстраивать в последовательность. Например, несколько аккордов можно воспроизводить последовательно, как показано на рис. 11.2. И как же все это смоделировать?

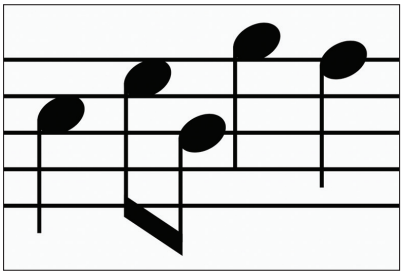


Рис. 11.1 Мелодия – это последовательность нот

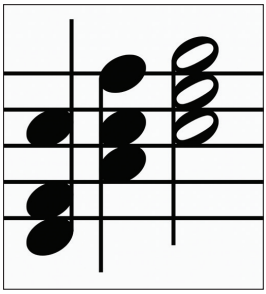


Рис. 11.2 Последовательность аккордов – одновременно звучащих нот

Наша структура данных нуждается в рекурсии или вложенности. В противном случае мы не сможем представить, например, последовательность последовательностей. Кроме того, нужно подумать, чего мы хотим достичь. Впоследствии нам не хотелось бы записывать данные этого типа в терминах его конструкторов, мы предпочли бы использовать операторы для конструирования нот, пауз, последовательностей и групп. Мы хотим иметь возможность смешивать все эти элементы, например помещать их в группу, которая должна воспроизводиться одновременно. Следовательно, моделируемая сущ-

ность должна быть представлена одним типом данных. Тип, удовлетворяющий всем этим требованиям, мог бы выглядеть так:

```
data NoteStructure a
  = Note Notelength a
  | Pause Notelength
  | Sequence [NoteStructure a]
  | Group [NoteStructure a]
```

Однако тут возникает проблема. В предыдущей главе мы использовали класс типов `Pitchable`, чтобы определить типы, для которых можно вычислить высоту звука. Это типы `Hz`, `Semitone` и `Chromatic`. Нашему синтезатору безразлично, с чем работать – с частотами или хроматическими нотами, – поэтому смешение этих типов имеет смысл. Но тип `NoteStructure` параметризован только одним типом. Это значит, что мы можем определить `NoteStructure Hz` и `NoteStructure Chromatic`, но не `NoteStructure`, в котором может быть смесь значений типа `Hz` и `Chromatic`.

То же ограничение относится и к спискам. Мы считаем, что эти структуры данных должны быть *однородны*, т. к. содержат значения единственного типа. Если бы мы хотели поместить данные разных типов в одну структуру данных, то структура стала бы *неоднородной*. Но есть ли такая штука в Haskell?

11.1.1 Навешивание квантора существования

Еще раз вспомним, как работают типы Haskell, и рассмотрим пример:

```
ghci> data D a = D a
ghci> :t [D (1 :: Int), D (2 :: Int)]
[D (1 :: Int), D (2 :: Int)] :: [D Int]
ghci> :t [D (1 :: Int), D (2 :: Float)]

<interactive>:1:19: error:
    • Couldn't match expected type 'Int' with actual type 'Float'
    • In the first argument of 'D', namely '(2 :: Float)'
      In the expression: D (2 :: Float)
      In the expression: [D (1 :: Int), D (2 :: Float)]
```

Здесь определяется простой тип данных с одним конструктором и переменной типа. Мы можем определить однородный список типа `[D Int]`, в котором все значения должны иметь тип `D Int`. Если мы попытаемся поместить в список значения типа `D Int` и `D Float`, то возникнет конфликт типов, поскольку `Int` – не то же самое, что `Float`, а значит, `D Int` – не то же самое, что `D Float`. Можно ли как-то скрыть параметр типа? Если бы нам удалось это сделать, то мы могли бы просто создать список типа `[D]`, и никаких ошибок типизации не

возникло бы. Оказывается, что это позволяет сделать расширение языка `ExistentialQuantification`.

Если это расширение активировано, то становится возможно определять типы данных, которые могут использовать переменные типа в *правой части* определения без упоминания их в *левой части*:

```
ghci> :set -XExistentialQuantification
ghci> data E = forall a. MkE a
ghci> :t MkE
MkE :: a -> E
ghci> :t MkE (1 :: Int)
MkE (1 :: Int) :: E
ghci> :t MkE (1 :: Float)
MkE (1 :: Float) :: E
ghci> :t [MkE (1 :: Int), MkE (2 :: Float)]
[MkE (1 :: Int), MkE (2 :: Float)] :: [E]
```

Такие несвязанные переменные типа вводятся ключевым словом `forall`. Как видим, у типа `E` нет переменной типа, но его конструктор имеет тип `a -> E`. Поэтому он может принимать значение *любого типа* и порождать значение типа `E`. Это означает, что вне зависимости от того, что мы передадим конструктору `MkE`, на выходе всегда будет один и тот же тип, а раз так, то мы можем помещать эти значения в список и тем самым конструировать *неоднородный* список.

Но как работать с таким типом? Как получить доступ к значениям, переданным конструктору? Для этого нужно написать функцию типа `E -> a`, где `a` – то же, что `a`, использованное в конструкторе `MkE`. Увы, это невозможно, потому что тип `E` не раскрывает это `a` – ведь именно этого мы и пытались достичь. Проблема в том, что мы ничего не знаем об `a`. Оно могло бы быть чем угодно.

Это можно исправить, предоставив ограничение типа для переменной типа с навешенным квантором существования:

```
ghci> data ShowData = forall a. (Show a) => ShowThis a
ghci> :{
ghci| showThis :: ShowData -> String
ghci| showThis (ShowThis x) = show x
ghci| :}
ghci> showValues = [ShowThis (1 :: Int), ShowThis ("Hi" :: String)]
ghci> :t showValues
showValues :: [ShowData]
ghci> map showThis showValues
["1","\"Hi\""]
```

В этом примере мы можем использовать `show` для значений внутри конструктора `ShowThis`, поскольку из определения типа `ShowData` ясно следует, что значения внутри конструктора имеют экземпляр класса типов `Show`. Следовательно, мы знаем, что элементы списка

значений типа `ShowData` могут быть неоднородны, но, безусловно, их можно использовать как аргументы `showThis`.

11.1.2 Использование типов с квантором существования

Мы можем воспользоваться этим, чтобы заставить `NoteStructure` принимать значения разных типов. Для этого сначала заведем новый тип для общего описания высоты звука. Нам понадобятся следующие расширения:

```
{-# LANGUAGE ExistentialQuantification #-}
{-# LANGUAGE StandaloneDeriving #-}
module Composition.Pitch where
```

Теперь мы можем определить тип `Pitch`, который содержит значения, имеющие экземпляр `Pitchable`. Затем определим экземпляр `Pitchable Pitch`, который применяет функцию `toFrequency` к значению внутри значения типа `Pitch`. Код показан ниже.

Листинг 11.1 Тип данных для структурирования нот

Определяет тип для высоты звука, который можно использовать для обертывания любого значения типа, имеющего экземпляры Pitchable и Show

```
data Pitch = forall a. (Pitchable a, Show a) => Pitch a

deriving instance Show Pitch

instance Pitchable Pitch where
  toFrequency (Pitch p) = toFrequency p
```

Выводит экземпляр Show для типа Pitch

Определяет экземпляр Pitchable Pitch

Мы также используем `Show` в качестве ограничения, так что можем вывести экземпляр `Show` для `Pitch`. Для типов данных с навешенными кванторами существования мы вынуждены использовать *автономный вывод* (standalone deriving), активируемый расширением языка `StandaloneDeriving`, которое позволяет записывать фразу `deriving` отдельно от определения типа данных. Теперь мы можем использовать этот тип для обертывания любого типа `Pitchable` и строить из них структуры данных, содержащие значения разных типов:

```
ghci> pitches = [Pitch (a 4), Pitch (440 :: Hz), Pitch (0 :: Semitone)]
ghci> :t pitches
pitches :: [Pitch]
ghci> pitches
[Pitch (Chromatic A 4),Pitch 440.0,Pitch 0]
ghci> map toFrequency pitches
[440.0,440.0,440.0]
```

Созданный нами список высот звука неоднороден, но мы по-прежнему можем преобразовывать их в частоты функцией `toFrequency`.

Теперь можно избавиться от переменной типа в нашей структуре `NoteStructure` и заменить ее типом `Pitch`, как показано в листинге ниже.

Листинг 11.2 Тип данных для структурирования нот

```
data NoteStructure
  = Note Notelength Pitch
  | Pause Notelength
  | Sequence [NoteStructure]
  | Group [NoteStructure]
  deriving (Show)
```



Имея этот тип, мы теперь должны найти способ транслировать написанные с его помощью композиции в нечто, из чего можно создавать звуки. У нас уже есть тип для звуков – `Event`. Пришло время включить `NoteStructure` в несколько событий. Для этого определим тип для исполнения музыки, который будет описывать несколько событий.

```
module Composition.Performance where

type Performance = [Event]
```

Этот тип должен содержать все события (с правильными моментами начала), так что мы можем подать их нашему типу `Oscillator` и услышать сладчайшую музыку.

11.2 Интерпретация структур

Для этого нам придется рекурсивно разбирать структуру `NoteStructure`, хранить длительности и помещать элементы в список. Поскольку это нужно делать рекурсивно, функция, которую мы собираемся написать, должна иметь аргумент `Seconds` – текущее время интерпретации. Кроме того, нам понадобится аргумент `TempoInfo`, чтобы знать, сколько времени должна длиться нота или пауза, т. к. тип `Event` понимает только `Seconds` как единицу измерения времени. В итоге нам нужно передавать не только `Performance`, но и `Seconds`, т. к. при рекурсивном разборе понадобится информация об истекшем времени. Разбор последовательностей и групп сворачивается

по элементам в соответствующих списках. Для последовательности начало элемента определяется уже истекшим временем, а возвращенное значение типа `Seconds` равно концу последнего порожденного события `Event`. В группе все элементы начинаются в одно и то же время, а возвращенное значение типа `Seconds` равно концу самого долгого из порожденных событий `Event`. Памятуя обо всем этом, мы можем написать функцию, показанную в следующем листинге.

Листинг 11.3 Функция для преобразования структур нот и пауз в допускающие исполнение события

```
module Composition.Performance where

...

import Composition.Note
import Composition.Notelength
import Composition.Pitch
import Data.List (foldl')

...

structureToPerformance ::
  TempoInfo ->
  Seconds ->
  NoteStructure ->
  (Seconds, Performance)
structureToPerformance tempoInfo start structure =
  case structure of
    (Note length pitch) ->
      let freq = toFrequency pitch
          duration = timePerNotelength tempoInfo length
       in (start + duration,
➥ [Tone {freq, start, duration}])
    (Pause length) ->
      let duration = timePerNotelength tempoInfo length
       in (start + duration,
➥ [Silence {start, duration}])
    (Group structs) -> foldl' f (start, []) structs
      where
        f (durAcc, perf) struct =
          let (dur, tones) =
➥ structureToPerformance tempoInfo start struct
```

Вычисляет длительность ноты или паузы по ее длине и информации о темпе

Возвращает конец звучания ноты и значение Performance с единственной нотой в его событии Event

Вычисляет длительность ноты или паузы по ее длине и информации о темпе

Определяет частоту по информации о высоте звука данной ноты

Рекурсивно вычисляет Performance и время его окончания по заданным структурам, используя заданное время начала как начало этого конкретного исполнения

Сворачивает рекурсивные структуры в последовательности или группы с заданным временем начала

Возвращает конец паузы и значение Performance с тишиной в его событии Event

```
          in (max dur durAcc, perf ++ tones)
  (Sequence structs) -> foldl' f (start, []) structs
    where
      f (durAcc, perf) struct =
        let (newdur, tones) = structureToPerformance tempoInfo durAcc
➥ struct
         in (newdur, perf ++ tones)
```

Рекурсивно вычисляет Performance и его время окончания и добавляет значения Event в аккумулятор, не придерживаясь какого-то определенного порядка

Рекурсивно вычисляет Performance и его время окончания по текущей структуре в свертке, используя текущий момент в качестве начала этого конкретного исполнения

Сворачивает рекурсивные структуры в последовательности или группы с заданным временем начала

Возвращает максимум заданных моментов времени, порождает конечный момент на основе рекурсивного вызова и добавляет значения Event в аккумулятор, не придерживаясь какого-то определенного порядка

Эта функция не располагает значения типа `Event` в правильном порядке, определяемом их временем начала, и нуждается в специальном аргументе `start`, который должен быть равен `0`, когда `NoteStructure` преобразуется впервые. Кроме того, она возвращает кортеж с некоторой длительностью, которая не важна для нас, если мы просто хотим воспроизвести эти события. Таким образом, мы можем написать обертывающую функцию, которая игнорирует этот возвращаемый кортеж и правильно сортирует значения. Она показана в листинге ниже.

Листинг 11.4 Обертывающая функция для преобразования нотных структур в исполнение

```
...
import Data.List (sortBy, foldl')
...

toPerformance :: TempoInfo -> NoteStructure -> Performance
toPerformance tempoInfo =
  sortBy (\x y -> compare (start x) (start y))
    . snd
    . structureToPerformance tempoInfo 0
```

Сортирует значения Event в результирующем значении Performance по времени начала исполнения

Читает второе значение в возвращенном кортеже

Вызывает structureToPerformance, передавая 0 в качестве времени начал

Благодаря этой функции мы перебросили мост между нашими композициями, которые описываются значениями типа `NoteStructure`, и значениями типа `Performance`, исполняемыми `Oscillator`.

> **Упражнение. Объединение периодов тишины**
>
> Наша функция `toPerformance` работает хорошо, но есть одна возможность для улучшения. Паузы всегда означают отсутствие звука – тишину. Наш тип `NoteStructure` позволяет создавать последовательность периодов тишины. Конечно, несколько периодов тишины можно объединить в один, более длительный. Напишите функцию типа `Performance → Performance`, которая именно это и делает, а затем включите ее вызов в функцию `toPerformance`. Подумайте, как протестировать эту функцию. Можете ли вы придумать для нее свойство QuickCheck?

Мы еще не реализовали тип `Oscillator`, способный исполнять `Performance` целиком. Это последний фрагмент пазла, который осталось положить перед реализацией композиции.

11.2.1 Микширование сигналов

Чтобы и дальше экспериментировать с типом `Performance`, мы хотим абстрагировать его с помощью простого класса типов. Мы сделаем это, чтобы впоследствии можно было добавлять другие источники шума. В конце концов, не только генераторы умеют порождать звуки. Класс типов показан в листинге ниже.

Листинг 11.5 Класс типов для преобразования `Performance` в сигнал

```
module Sound.Sound where

import Util.Types
import Composition.Performance

class Performer p where
  play :: p -> Performance -> Signal
```

Функция `play` впоследствии будет применяться к значению `Performance` для конструирования сигнала. Как это будет работать? Стоящая перед нами проблема заключается в том, что события могут естественным образом перекрываться. Создаваемый нами синтезатор полифонический, т. е. может воспроизводить несколько нот одновременно. Поэтому нам нужно сначала подумать, как объединить несколько сигналов в один. Для этого мы хотим написать функцию, которая умеет складывать сигналы без отсечения. *Отсечение* (clipping) имеет место, когда уровень отсчета оказывается больше `1` или

меньше -1. В таком случае наша функция `limit` срезает сигнал, что приводит к искажениям. Этого следует избегать. Поэтому при сложении сигналов мы должны гарантировать, что отсчет не выйдет за максимальный или минимальный уровень. Этого можно добиться путем деления суммы сигналов на их количество. Само сложение сигналов похоже на функцию `zipWith (+)` с тем отличием, что мы не можем останавливаться после окончания короткого сигнала. Необходимо включать и более длинный. Но как сложить произвольное число сигналов – ведь функции `zipWithN` не существует? Ответ прост: в очередной раз воспользоваться сверткой. Добавляя каждый сигнал, один за другим, в свертку, мы можем сложить все, а затем разделить отсчеты на количество сложенных сигналов. Это показано в листинге ниже.

Листинг 11.6 Функция микширования нескольких сигналов в один

```
module Sound.Sound where

...
import Data.List (foldl')
...

mix :: [Signal] -> Signal
mix signals = (/ n) <$> foldl' addSignals [] signals
  where
    n :: Double
    n = fromIntegral $ length signals

    addSignals :: Signal -> Signal -> Signal
    addSignals xs [] = xs
    addSignals [] ys = ys
    addSignals (x : xs) (y : ys) = (x + y) :
➥ addSignals xs ys
```

Определяет количество сигналов и преобразует его в тип Double

Сворачивает сигналы путем их сложения и последующего деления отсчетов на количество сигналов

Возвращает время окончания самого долгого сигнала

Складывает первые отсчеты двух сигналов и рекурсивно прибавляет остальные

Разобравшись с микшированием сигналов, мы должны решить, как идентифицировать перекрывающиеся значения типа `Event`, чтобы их можно было сгруппировать, воспроизвести с помощью `Oscillator`, а затем снова микшировать сигналы. Сделанное нами – это метод генерирования полифонических звуков, т. е. одновременного воспроизведения нескольких звуков. В большинстве синтезаторов на количество полифонически воспроизводимых звуков налагаются аппаратные ограничения. Наш синтезатор способен на бесконечную полифонию, при условии что мы дадим ему достаточно времени для вычисления звуков!

Вернемся к одновременно исполняемым событиям. Чтобы определить, перекрываются ли два значения типа `Event`, мы можем вос-

пользоваться селектором поля `start` и функцией `end` для проверки того, принадлежит ли начало одного события интервалу другого. Если да, то события перекрываются. Функция для выполнения этой проверки показана в листинге 11.7.

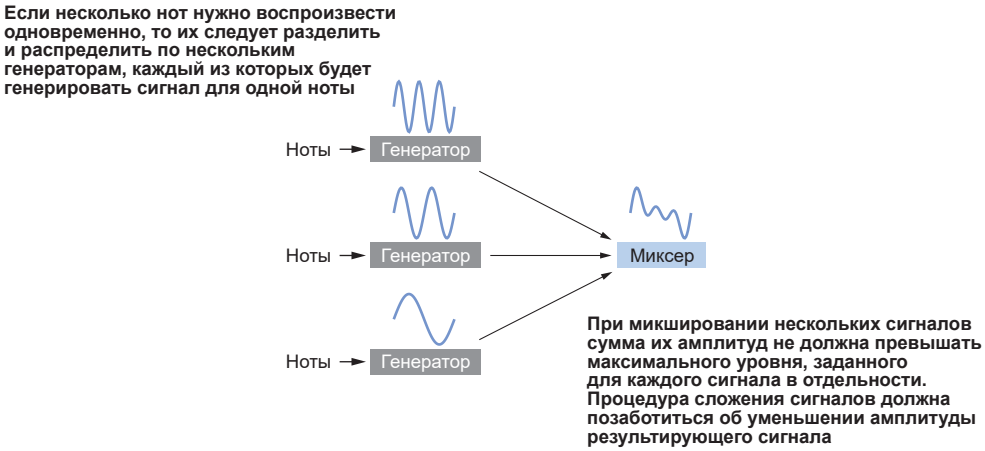


Рис. 11.3 Идея генерирования полифонических аудиосигналов

Листинг 11.7 Функция проверки перекрытия событий

```
module Composition.Performance where

...

overlaps :: Event -> Event -> Bool
overlaps e1 e2 =
  start e1 `between` (start e2, end e2)
    || start e2 `between` (start e1, end e1)
  where
    between x (a, b) = x >= a && x <= b
```

Проверяет, попадает ли начало первого события в интервал второго

Проверяет, попадает ли начало второго события в интервал первого

Определяет функцию, которая возвращает True, если заданное значение принадлежит заданному диапазону

Имя вспомогательной функции `between` выбрано так, чтобы ее можно было записывать в инфиксном стиле, – так код будет удобнее читать.

Упражнение. Свойства функций работы со звуком

Как и многие другие, наши функции `mix` и `overlaps` обладают некоторыми свойствами. Функция `mix` особенно важна из-за вышеупомянутой проблемы отсечения. Напишите какие-нибудь свойства QuickCheck, которые проверяли бы правильность этой функции.

Имея эти функции, мы можем приступить к написанию экземпляра `Performer` для типа `Oscillator`.

11.2.2 Полифонические группы

На пути преобразования `Performance` в `Signal` есть несколько ловушек, которых мы хотели бы избежать. Прежде всего список значений типа `Event` упорядочен. Поэтому направление его свертывания важно. Во-вторых, при группировке событий нужно учитывать их перекрытие. При этом создаются непересекающиеся группы событий, между которыми могут быть неявные паузы, т. е. одно событие может заканчиваться задолго до начала другого. Необходимо также гарантировать, что в процессе микширования сигналов не происходит отсечения, но об этом уже заботится наша функция `mix`.

Сначала поговорим о группировке событий. Сворачивая события слева направо, мы можем начать с первого встретившегося события и поместить его в свою собственную группу. Для следующего события в `Performance` мы сначала должны проверить, перекрывается ли оно с каким-нибудь событием в текущей группе. Если нет, мы можем добавить его в эту группу. Если же да, то нужно начать новую группу, включив в нее это событие. Этот процесс повторяется для всех событий. Описанный алгоритм обладает важным свойством – он не изменяет порядок событий внутри групп. Код станет понятнее, если мы воспользуемся списковым включением и функцией `or`:

```
ghci> :t or
or :: Foldable t => t Bool -> Bool
ghci> :t and
and :: Foldable t => t Bool -> Bool
```

Функции `or` и `and` понять легко: они сворачивают значение типа `Foldable`, содержащее булевы значения, строя их дизъюнкцию или конъюнкцию соответственно. В сочетании со списковыми включениями это позволяет динамически порождать список булевых значений и применять к ним `or` или `and`:

```
ghci> xs = [Tone 0 0 1, Tone 0 2 1]
ghci> x1 = Tone 0 2 1
ghci> x2 = Tone 0 1.5 0.1
ghci> or [ x1 `overlaps` e | e <- xs ]
True
ghci> or [ x2 `overlaps` e | e <- xs ]
False
```

Поскольку результатом вычисления этих выражений является единственное значение типа `Bool`, мы можем использовать их в качестве *стражей* в наших функциях! Это позволяет различать случаи в довольно сложных свойствах списков.

Осталось решить, как нам воспроизводить эти группы событий. Поскольку между ними могут быть неявные паузы, необходимо следить за временем. В рекурсивной функции это можно сделать с помощью аргумента по аналогии с тем, что мы делали в функции `structureToPerformance`. Далее мы должны пробежаться по списку (слева направо) и воспроизводить события с помощью `Oscillator`. Если группа событий почему-либо была неправильно упорядочена, то пришлось бы возвращаться назад во времени и воспроизводить тон, который уже, возможно, был заполнен тишиной. В таком случае ничего не остается, кроме как завершиться с ошибкой `error`. Но если наша реализация корректна, то ничего подобного случиться не может.

После группировки и генерирования сигналов по событиям останется только микшировать все сигналы в терминах их конструкторов. По счастью, это простое применение функции `mix`. Полный код этой функции представлен ниже.

Листинг 11.8 Экземпляр класса типов для исполнения с помощью генератора

```
module Sound.Sound where

...
import Sound.Synth
...

instance Performer Oscillator where
  play (Osc oscf) perf = mix $ fmap
➥ (playEvents 0) eventGroups
    where
      eventGroups :: [[Event]]
      eventGroups = foldr insertGroup [] perf
        where
          insertGroup x [] = [[x]]
          insertGroup x (es : ess)
            | or [x `overlaps` e | e <- es] =
➥ es : insertGroup x ess
            | otherwise = (x : es) : ess

      playEvents :: Seconds -> [Event] -> Signal
      playEvents _ [] = []
      playEvents curTime (event : xs)
        | curTime < ts =
          concat
            [ silence (ts - curTime),
```

Преобразует каждую группу событий в Signal и микширует эти сигналы

Сворачивает значения типа Event в Performance слева направо

Добавляет одну группу с одним элементом в аккумулятор, если он пуст

Рекурсивно добавляет текущее событие Event в другую группу, если оно перекрывается с каким-либо другим событием в текущей группе в аккумуляторе

Добавляет текущее событие Event в текущую группу, если оно не перекрывается ни с каким другим событием в ней

Возвращает пустой сигнал, если больше не осталось событий для воспроизведения

Добавляет необходимый промежуток тишины к сигналу, а затем воспроизводит следующее событие, если текущая временная метка еще не дошла до его начала

```
            oscf event,
            playEvents te xs
          ]
      | curTime == ts =
➥ oscf event ++ playEvents te xs
      | otherwise = error "Event occurs in the past!"
      where
        ts = start event
        te = end event
```

Рекурсивно преобразует остаток группы в сигнал

Возбуждает ошибку, если текущая временная метка оказалась дальше начала следующего события

Преобразует следующее событие в сигнал и рекурсивно дописывает остаток, если текущая временная метка равна началу этого события

Обратите внимание, как мы закоротили выполнение в функции `playEvents`. Если в группе не осталось событий, то мы возвращаем пустой `Signal`, потому что функция `mix` позаботится о правильной длительности.

Имея эту функцию, мы наконец можем воспроизвести значения типа `NoteStructure`, а также нашу первую композицию.

```
ghci> melody1 = Sequence [Note whole (Pitch $ c 4), Pause half,
➥ Note whole (Pitch $ f 4)]
ghci> melody2 = Sequence [Note whole (Pitch $ f 4),
➥ Note half (Pitch $ g 4), Note whole (Pitch $ a 4)]
ghci> melody3 = Sequence [Note whole (Pitch $ f 3),
➥ Note half (Pitch $ c 3), Note whole (Pitch $ c 4)]
ghci> group = Group [melody1, melody2, melody3]
ghci> perf = toPerformance (TempoInfo 120 4) group
ghci> perf
[Tone {freq = 261.6255653005986, start = 0.0, duration = 2.0},Tone {freq
  ➥ = 349.2282314330039, start = 0.0, duration = 2.0},Tone {freq =
  ➥ 174.61411571650194, start = 0.0, duration = 2.0},Silence {start
  ➥ = 2.0, duration = 1.0},Tone {freq = 391.99543598174927, start
  ➥ = 2.0, duration = 1.0},Tone {freq = 130.8127826502993, start
  ➥ = 2.0, duration = 1.0},Tone {freq = 349.2282314330039, start
  ➥ = 3.0, duration = 2.0},Tone {freq = 440.0, start = 3.0, duration
  ➥ = 2.0},Tone {freq = 261.6255653005986, start = 3.0, duration = 2.0}]
ghci> signal = play piano perf
ghci> writeWav "performance.wav" signal
Writing 220500 samples to performance.wav
```

Здесь мы видим, как три мелодии объединяются в одну группу. Ноты воспроизводятся одновременно, а результирующий сигнал не отсекается.

Чудненько! Но, как мы видим, написать даже такую простенькую композицию – тяжкое занятие. Мы не хотим выписывать типы данных Haskell. Нам нужно более удобное для написания и понимания представление того же самого, в идеале вообще абстрагирующее стоящий за ним тип данных.

11.3 Реализация предметно-ориентированного языка

Теперь мы наконец-то готовы перейти к обсуждению вопроса о сочинении музыки в нашей программе. Очевидно, нас не интересует ручное выписывание конструкторов в виде гигантского синтаксического дерева безумной сложности. А хотели бы мы создать *предметно-ориентированный язык* (DSL).

Haskell позволяет нам определять свои операторы с собственными правилами взаимодействия с другими данными. Кроме того, он позволяет задавать приоритеты операторов, что делает синтаксис более понятным. Поэтому Haskell является отличной платформой для разработки предметно-ориентированных языков для специализированного применения.

> **ПРИМЕЧАНИЕ** Строго говоря, предметно-ориентированный язык, непосредственно встраиваемый в некоторый язык программирования, как в этой главе, следует называть *встраиваемым предметно-ориентированным языком* (EDSL). Но во имя краткости мы опускаем это дополнительное различие.

Мы воспользуемся этим для определения синтаксиса, позволяющего легко сочинять музыку на основе типа `NoteStructure`, который композитор (пользователь DSL) не только не будет никогда видеть, но даже и знать о его существовании. Первым делом надо придумать, как записывать ноты. Наш текущий метод несколько многословен: `Note whole (Pitch $ e 4)`. Но раз конструктор такой разговорчивый, то можно создать заменяющий его оператор:

```
module Composition.Note where

...

(.|) :: (Pitchable a, Show a) => a -> Notelength -> NoteStructure
(.|) p l = Note l (Pitch p)

infixr 4 .|
```

Мы выбрали в качестве оператора `.|`, потому что он немного напоминает музыкальную ноту. Наш оператор по умолчанию определен как *инфиксный*. Первый аргумент располагается перед оператором, а второй – после него. Замена конструктора важна, потому что пользователь DSL вообще не должен иметь дело с типом `NoteStructure`. Явно мы даем доступ только к типам, имеющим экземпляры `Pitchable` и `Show`, потому что таково ограничение конструктора `Pitch`. И снова мы видим, что тип `NoteStructure` неоднородный, т. к.

мы передаем полиморфный аргумент (a) функции, но эта переменная типа не присутствует в результате функции!

11.3.1 Упрощение синтаксиса

Новый оператор позволяет записывать ноты кратко. Однако длительности (`whole`, `half` и т. д.) по-прежнему очень пространные. Чтобы уменьшить длины идентификаторов, мы можем создать короткие имена для констант типа `Notelength`:

```
module Composition.Notelength where

...

wn :: Notelength
wn = whole

hn :: Notelength
hn = half

...

sn :: Notelength
sn = sixteenth
```

После этого простого шага записывать ноты стало очень легко:

```
ghci> c 4 .| wn
Note (1 % 1) (Pitch (Chromatic C 4))
ghci> f 8 .| triplet sn
Note (1 % 24) (Pitch (Chromatic F 8))
ghci> (400 :: Hz) .| (100000 % 100001)
Note (100000 % 100001) (Pitch 400.0)
```

То же самое можно проделать с паузами, создав однобуквенный синоним конструктора `Pause`:

```
p :: Notelength -> NoteStructure
p = Pause
```

Вот мы и позаботились еще об одном конструкторе.

11.3.2 Пользовательские операторы для списокоподобных структур данных

Далее рассмотрим типы `Sequence` и `Group`. В общем случае мы хотим, чтобы пользователь нашего DSL:

1. не думал о списках;
2. не думал о том, как комбинировать последовательности и группы;
3. не испытывал затруднений с повторением.

Это значит, что мы должны полностью абстрагировать понятия последовательностей и групп. Этого можно добиться, определив операторы, позволяющие комбинировать ноты и паузы. Сначала разберемся с последовательностями. Если встретились две последовательности, мы просто объединяем их в новую последовательность. Если только один аргумент нашего оператора является последовательностью, то мы добавляем в нее второй аргумент. Если ни один аргумент не является последовательностью, то мы создаем новую последовательность. Код оператора показан в листинге ниже.

Листинг 11.9 Инфиксный оператор для построения последовательностей композиций

```
(<~>) :: NoteStructure -> NoteStructure -> NoteStructure
(<~>) (Sequence xs) (Sequence ys) = Sequence $ xs ++ ys
(<~>) (Sequence xs) x = Sequence $ xs ++ [x]
(<~>) x (Sequence xs) = Sequence $ x : xs
(<~>) a b = Sequence [a, b]

infixr 3 <~>
```

Соединяет две последовательности

Добавляет второй аргумент в последовательность

Создает новую последовательность из обоих аргументов

Это дает нам возможность избежать выписывания конструктора `Sequence` *и* необходимого ему списка:

```
ghci> c 4 .| wn <~> e 4 .| wn <~> g 4 .| wn
Sequence [Note (1 % 1) (Pitch (Chromatic C 4)),Note (1 % 1)
➥ (Pitch (Chromatic E 4)),Note (1 % 1) (Pitch (Chromatic G 4))]
```

Забавно, что функция для групп очень, очень похожа (если не полностью идентична). Ее код представлен ниже.

Листинг 11.10 Инфиксный оператор для построения групп композиций

```
(<:>) :: NoteStructure -> NoteStructure -> NoteStructure
(<:>) (Group xs) (Group ys) = Group $ xs ++ ys
(<:>) (Group xs) x = Group $ xs ++ [x]
(<:>) x (Group xs) = Group $ x : xs
(<:>) a b = Group [a, b]

infixr 2 <:>
```

Соединяет две группы

Добавляет второй аргумент в группу

Создает новую группу из обоих аргументов

Конечно же, это сходство – не случайность. Поскольку структуры `Sequence` и `Group` идентичны, не следует удивляться тому, что и операторы для них также идентичны. Единственное различие – имя конструктора, потому что именно оно определяет, как данные будут интерпретироваться дальше.

ПРИМЕЧАНИЕ Идеи DSL, предложенного в этой главе, частично основаны на проекте Haskore, где используются похожие операторы.

Теперь, когда операторы имеются, осталось ответить только на один вопрос. Каков результат выражения `c 4 .| wn <~> e 4 .| wn <:> g 4 .| wn`? Смысл вопроса – в *приоритете операторов*. Что приоритетнее: `<~>` или `<:>`? Это не простой вопрос, потому что ответ зависит от того, как будет использоваться DSL, но мы решили отдать приоритет оператору `<~>`. А как этого добиться?

11.3.3 Объявления приоритета операторов

В Haskell имеется синтаксис для объявления правил назначения приоритетов. Мы уже встречали его при обсуждении оператора `$` в главе 7. Объявление начинается одним из ключевых слов `infixr`, `infixl` или `infix`, которое определяет, является оператор правоассоциативным, левоассоциативным или вообще неассоциативным. Далее следует приоритет от 0 до 9. Объявление завершается обозначением оператора, о котором идет речь. Этот синтаксис называется *объявлением приоритета операторов*.

Мы сделаем все операторы правоассоциативными. Оператор `(.|)` должен иметь более высокий приоритет, чем все остальные, потому что он не используется для комбинирования композиций, а образует атомарные строительные блоки. Далее, `(<~>)` приоритетнее, чем `(<:>)`. Эти правила описываются следующими объявлениями:

```
infixr 4 .|
infixr 3 <~>
infixr 2 <:>
```

Мы позаботились почти обо всех желательных свойствах нашего DSL. Осталось еще упростить повторение. Если мы хотим повторить мелодию или аккорд, то по-прежнему должны копировать объявления. Конечно, можно придумать *новые* операторы для повторения! Эти операторы могут копировать заданную структуру `NoteStructure` заданное число раз и обертывать их либо `Sequence`, либо `Group`. Это показано в листинге ниже:

Листинг 11.11 Операторы для повтора композиций

```
(<~|) :: NoteStructure -> Natural -> NoteStructure
(<~|) (Sequence xs) n = Sequence $ concat [xs | _ <- [1 .. n]]
(<~|) struct n = Sequence $ [struct | _ <- [1 .. n]]

(|~>) :: Natural -> NoteStructure -> NoteStructure
```

```
(|~>) = flip (<~|)

(<:|) :: NoteStructure -> Natural -> NoteStructure
(<:|) (Group xs) n = Group $ concat [xs | _ <- [1 .. n]]
(<:|) struct n = Group $ [struct | _ <- [1 .. n]]

(|:>) :: Natural -> NoteStructure -> NoteStructure
(|:>) = flip (<:|)
```

Чтобы сделать синтаксис чуть более гибким, существуют операторы, в которых число повторений указывается слева и справа. Поскольку повторение должно «обертывать» другие композиции, приоритет этих операторов должен быть меньше, чем всех остальных:

```
infixr 1 <~|
infixr 1 |~>
infixr 1 <:|
infixr 1 |:>
```

Испытаем наш новый язык в деле:

```
ghci> c 4 .| wn <~> e 4 .| wn <:> g 4 .| wn <~| 2
Sequence [Group [Sequence [Note (1 % 1) (Pitch (Chromatic C 4)),
➥ Note (1 % 1) (Pitch (Chromatic E 4))],Note (1 % 1)
➥ (Pitch (Chromatic G 4))],Group [Sequence [Note (1 % 1)
➥ (Pitch (Chromatic C 4)),Note (1 % 1) (Pitch (Chromatic E 4))],
➥ Note (1 % 1) (Pitch (Chromatic G 4))]]
```

Как видим, «программу» стало гораздо легче читать и писать, и при этом не требуется ничего знать о наших структурах данных.

Упражнение. Изменение приоритетов

Мы решили, что оператор (<~>) приоритетнее, чем (<:>). Но проблемы можно было бы избежать, создав синонимы для обоих операторов с переставленными приоритетами. Тогда композитору не нужно было бы ставить скобки – достаточно переключиться на другой оператор! Реализуйте эти синонимы.

Ну вот, наша цифровая музыкальная шкатулка и готова!

Подведем итоги сделанному. Во-первых, мы реализовали типы и функции для предметно-ориентированной задачи: генерирование музыкальных звуков. Мы смоделировали в своей программе свойства реального мира в виде типов данных, а также технологию простого синтезатора. Попутно мы создали небольшой каркас для построения синтезированных инструментов, который легко расширить.

Упражнение. Добавление семплера

Наш примитивный синтезатор умеет не все. Единственный его генератор звуков – тип `Oscillator`, который умеет создавать аккорды и мелодии, но оставляет желать много большего, когда речь заходит о перкуссии. Где же коровий колокольчик? В электронной музыке эту проблему часто решают с помощью семплеров – устройств, умеющих семплировать звук другого инструмента (например, барабанов) и воспроизводить его в музыкальной аранжировке. Реализуйте такой тип `Sampler` для нашего каркаса синтезатора. Придется проявить изобретательность. Вы должны будете подумать, как дать этому компоненту доступ к звуковым файлам, как активировать воспроизведение этих файлов и как включить все это в наш DSL.

Во-вторых, мы смоделировали еще одну предметно-ориентированную задачу: сочинение музыки. Мы воспользовались различными типами данных и функциями, чтобы абстрагировать смысл музыкальных композиций в нечто такое, чем было бы проще управлять из кода.

В-третьих, мы создали предметно-ориентированный язык для сочинения музыки. Он абстрагирует наши типы данных, так чтобы их можно было использовать, даже не понимая внутренней реализации. Его можно расширить, поддержав больше способов композиции, например алеаторику (специальный термин для выражения понятия случайности). Поскольку язык встроен в Haskell, мы можем добавлять в него любую функциональность, реализованную в каркасе.

Осталось только протестировать наш синтезатор и сочинить какую-нибудь музыку. В репозитории есть небольшой образчик!

Резюме

- Расширение языка `ExistentialQuantification` можно использовать для создания неоднородных структур данных, скрывающих параметры типов внутри типов.
- Чтобы неоднородный тип был полезен, необходимо задать ограничения типа, так чтобы было ясно, что можно делать с конкретным значением этого типа.
- Структуры данных могут выступать в роли описания результата, которое еще нужно интерпретировать.
- Новые операторы можно реализовать как обычные функции.
- Предметно-ориентированные языки можно реализовать на Haskell, написав специальные операторы для определения структуры данных, которые затем интерпретируются.
- Ассоциативность и приоритеты операторов можно определить с помощью объявлений `infix`, `infixl` и `infixr`.

12 Разбор пиксельных данных

Краткое содержание этой главы:

- написание парсеров текста с нуля;
- комбинирование эффектообразующих действий;
- теория, лежащая в основе функторов, аппликативных функторов и монад;
- построение парсеров двоичных данных с помощью библиотеки Attoparsec.

В предыдущих главах мы уже касались вопроса о чтении данных, будь то просто строки файла или более сложные форматы, например CSV. Мы написали функции чтения данных, специализированные для решаемой задачи. Однако обычно в таких ситуациях поступают иначе. Какой бы язык мы ни использовали, обычно мы пишем для его разбора парсер. В Haskell процесс разработки парсеров едва ли не уникален – так чисто простые строительные блоки компонуются в более крупные парсеры.

В этой главе мы напишем упрощенную библиотеку для чтения PNM-файлов, семейства простых форматов изображений. При этом мы научимся не только компоновать эффектообразующие действия, но и обрабатывать различные ошибки и отказы, которые могут

встретиться по ходу дела. Для этого мы будем пользоваться *аппликативными функторами* и *монадами*, поймем, как они работают и почему являются важным строительным блоком в Haskell. После этого познакомимся с широко используемой библиотекой комбинаторов Attoparsec и с ее помощью создадим высокопроизводительные парсеры файлов изображений.

Мы начнем эту главу определением задачи, которую собираемся решить. Затем напишем собственную библиотеку разбора, чтобы хотя бы приступить к ее решению. А напоследок применим полученные знания, воспользовавшись уже имеющейся высококачественной библиотекой для написания парсеров.

12.1 Написание парсера

С появлением фотографии нас стало интересовать не только запечатление окружающего мира, но и модификация изображений после их создания. Это относится не только к фотографии, но и к любому изобразительному искусству. Современным фотографам и художникам одинаково повезло, потому что большинство их работ переместилось в цифровую область, где изображения – не более чем биты и байты. А компьютеры – мастера преобразования и модификации этих битов и байтов с целью создания чего-то совершенно нового.

Но ведь мы, программисты, можем заставить компьютеры делать все, что нам нужно. А значит, пришло время написать программу, умеющую преобразовывать данные изображений! Для этого мы должны решить, какие форматы изображений будем поддерживать. В наши дни нет недостатка в сложных форматах с высокой степенью сжатия, но мы предпочтем старую школу, сориентировав свою библиотеку на форматы из проекта Netpbm.

12.1.1 Переносимые изображения из прошлого

В 1980-х годах поделиться изображением было несколько сложнее, чем просто загрузить его в облако. Если вы хотели чем-то поделиться, то должны были отправить это по электронной почте, которая тогда не жаловала двоичные данные. Пересылаемое изображение необходимо было представить в старом добром 7-битовом коде ASCII. Чтобы решить эту проблему, Джеф Посканзер изобрел семейство файловых форматов, которые можно было распространять в ASCII или в двоичном виде и которые не содержали избыточных метаданных, что делало их удобными для разбора и понимания.

К этим форматам – иногда их называют «portable anymap format» (PNM) – относятся *portable pixmap format* (PPM), *portable graymap format* (PGM) и *portable bitmap format* (PBM). Они используются со-

ответственно для хранения цветных, полутоновых и черно-белых изображений. Существует также комплект инструментов для работы с этими форматами, *Netpbm*, который позволяет преобразовывать их в более употребительные форматы, и наоборот.

Кратко рассмотрим, как эти форматы работают. Каждый из них содержит заголовок, а в нем обязательное магическое число – сигнатуру. Сигнатура определяет, какой формат используется и в какой кодировке: ASCII или двоичной. Расширение файла обычно не используется для определения формата. Далее в заголовке хранятся ширина и высота изображения. Для форматов PPM и PGM есть еще четвертое поле – максимальное значение одной точки данных. Взгляните на пример в следующем листинге.

Листинг 12.1 Простой PBM-файл с комментариями в заголовке

```
P1  # PBM in ASCII
2 2 # 2x2 pixel size
1 0
0 1
```

Как видим, заголовок может содержать комментарии, начинающиеся знаком #. Они всегда записаны в кодировке ASCII, вне зависимости от сигнатуры. В этом примере определено растровое изображение размеров 2×2 пикселя. Следующие далее двоичные данные представляют значения этих пикселей. В увеличенном масштабе изображение выглядит, как показано на рис. 12.1.

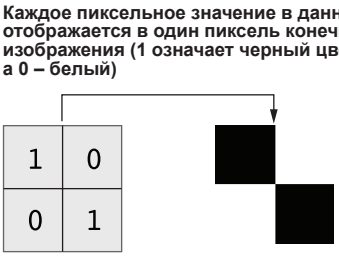


Рис. 12.1 Пример переносимого растрового изображения, увеличенного на 150 %

В табл. 12.1 представлены эти файловые форматы, их сигнатуры и описание содержащихся в них данных. Видно, что у каждого формата есть две сигнатуры: для версии ASCII и двоичной версии.

> **ПРИМЕЧАНИЕ** Заголовок форматов PNM всегда представлен в кодировке ASCII!

Для растровых и полутоновых файлов одно значение после заголовка соответствует одному пикселю изображения. Но в формате

пиксельной карты (pixmap) пиксель состоит из трех значений: для красного, зеленого и синего каналов.

Таблица 12.1. Сигнатуры форматов PNM

Формат	Сигнатуры		Расширение	Данные изображения (для одного пикселя)
	ASCII	Двоичный		
Portable bitmap format (PBM)	`P1`	`P4`	`.pbm`	`0` (черный) или `1` (белый)
Portable graymap format (PGM)	`P2`	`P5`	`.pgm`	От `0` до `maximum value`, где `maximum value` (максимальная величина) задана в заголовке
Portable pixmap format (PPM)	`P3`	`P6`	`.ppm`	От `0` до `maximum value` для каждого RGB-канала, где `maximum value` (максимальная величина) задана в заголовке

Задание максимальной величины в заголовке показано в листинге 12.2. Для записи элементов заголовка нет строгих правил. Их можно поместить на одной строке, но обычно сигнатуру, размеры изображения и максимальную величину располагают в разных строках. Кроме того, данные изображения можно располагать в отдельных строках, но это необязательно. Для удобства восприятия рекомендуется, чтобы длина строки не превышала 76 знаков, но и этой рекомендацией можно пренебречь, если наша единственная цель – хранить данные в виде, удобном компьютеру для чтения.

Листинг 12.2 Простой PGM-файл с комментариями в заголовке

```
P2 10 14 # PGM в коде ASCII, 10x14 пикселей
9        # Значения от 0 (черный) до 9 (белый)
9 6 3 4 9 9 9 9 9 9
8 4 2 2 6 9 9 9 9 9
8 6 9 5 4 9 9 9 9 9
8 9 9 9 4 9 9 9 9 9
9 9 9 9 4 7 9 9 9 9
9 9 9 9 3 6 9 9 9 9
9 9 9 6 1 3 9 9 9 9
9 9 9 4 0 3 8 9 9 9
9 9 7 0 4 4 8 9 9 9
9 9 4 1 7 5 6 9 9 9
9 7 1 3 9 6 5 9 9 8
9 5 0 7 9 8 3 8 9 6
8 1 2 8 9 8 3 3 4 4
5 2 5 9 9 9 7 2 1 7
```

Результат отображения этого PGM-файла показан на рис. 12.2. С помощью программ *Netpbm* и *ImageMagick* легко преобразовать эти файлы в другие, более употребительные форматы, например jpeg или png.

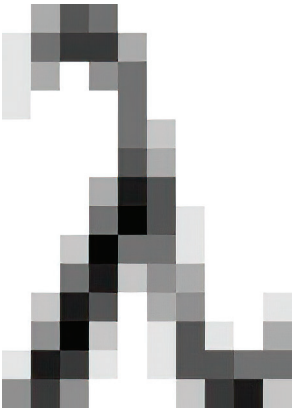


Рис. 12.2 Пример переносимого полутонового изображения, увеличенный на 150 %

Получив представление о том, как эти форматы работают, мы можем подумать, как читать такие файлы из программы.

12.1.2 Как разбирать файл

Для чтения информации – неважно, из файлов, из сетевых пакетов или из последовательного порта – мы должны описать ожидаемую структуру. Это относится в равной мере к чтению CSV-файлов, кода на языке программирования или (как в нашем случае) файлов изображений. Разбор этой структуры часто производится с помощью парсеров. Парсер – это программа или часть программы, которая транслирует строку символов или байтов в данные, допускающие интерпретацию указанной программой. У парсера есть две основные обязанности:

- проверять правильность структуры данных и сообщать об ошибках;
- преобразовывать строки символов или байтов в платформенные типы данных.

Теоретически для достижения этих целей можно написать простые функции, как мы делали в главе 8, когда разбирали CSV-файлы. Но тут есть недостаток – только глядя на определение парсера, трудно понять, чего он ожидает. А поскольку мы не хотим это повторять и собираемся реализовать парсер на Haskell, то добавим еще две цели:

- функции парсера должны допускать композицию;
- определение парсера должно проливать свет на ожидаемую структуру.

Итак, давайте напишем общий парсер на Haskell. Начните новый проект командой `stack new` и добавьте следующие расширения языка по умолчанию:

```
default-extensions:
- OverloadedStrings
- RecordWildCards
- NamedFieldPuns
```

Чтобы начать работу, мы должны определить тип для парсера. По мере необходимости мы будем изменять этот тип.

Поскольку парсер преобразует вход в какой-то платформенный тип данных, он мог бы быть простой функцией:

```
type Parser a = Text -> a
```

Наш упрощенный парсер будет работать с данными типа `Text` из модуля `Data.Text`. Если бы мы захотели написать парсер сигнатур формата PNM, то могли бы сделать это следующим образом:

```
module Graphics.PNM.Types where

import Parser.Core

data MagicNumber = P1 | P2 | P3 | P4 | P5 | P6 deriving (Eq, Show)

magicNumberP :: Parser MagicNumber
magicNumberP "P1" = P1
magicNumberP "P2" = P2
magicNumberP "P3" = P3
magicNumberP "P4" = P4
magicNumberP "P5" = P5
magicNumberP "P6" = P6
```

Наш парсер, который мы назвали `magicNumberP`, может прочитать сигнатуру из `Text`. Далее в этой главе мы предполагаем, что для всего проекта активировано расширение языка `OverloadedStrings`. Однако очевидно, что в нашем парсере сопоставление с образцом неполное, что приведет к краху программы, если при разборе возникнет ошибка. Следовательно, в нашем типе должна быть предусмотрена возможность ошибки.

> **ПРИМЕЧАНИЕ** Как для переменных типа `Maybe` и `Either` принято употреблять префикс `m`, так для парсеров обычен постфикс `P`!

В предыдущих главах мы узнали, что это можно сделать с помощью типа `Either`. Это также дает нам возможность вернуть сообщение об ошибке в парсере. Поэтому изменим тип следующим образом:

```
type ErrorMessage = String
type Parser a = Text -> Either ErrorMessage a
```

Новый парсер сигнатур выглядит так:

```
magicNumberP :: Parser MagicNumber
magicNumberP "P1" = Right P1
magicNumberP "P2" = Right P2
magicNumberP "P3" = Right P3
magicNumberP "P4" = Right P4
magicNumberP "P5" = Right P5
magicNumberP "P6" = Right P6
magicNumberP t =
  Left $
    "Could not parse \""
      ++ T.unpack t
      ++ "\" as magic number"
```

Эта функция не приводит к краху программы в случае ошибки, а возвращает сообщение о ней:

```
ghci> magicNumberP "P3"
Right P3
ghci> magicNumberP "Not a magic number"
Left "Could not parse \"Not a magic number\" as magic number"
```

На первый взгляд, работает, но возникает проблема: как скомпоновать этот парсер с другими, чтобы прочесть весь заголовок PNM-файла? Мы хотим, чтобы результатом разбора разных частей заголовка были разные типы, игнорируя при этом пробелы, знаки новой строки и комментарии. Это показано на рис. 12.3.

Каждая часть данных разбирается в значение какого-то типа Haskell. Ожидается, что сигнатура, ширина и высота должны присутствовать, но максимальной величины может и не быть, поэтому оно представлено типом Maybe. Пробелы и комментарии игнорируются, поэтому они не порождают типа, несущего информацию

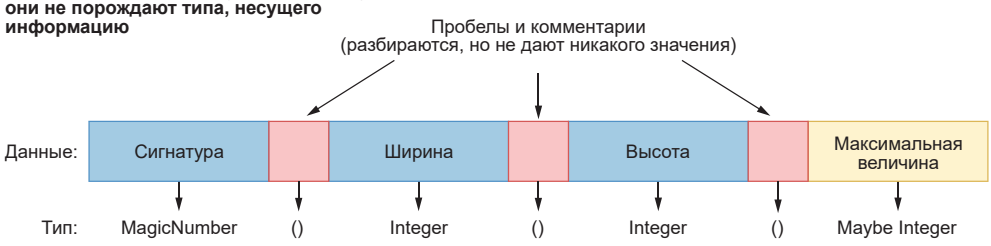
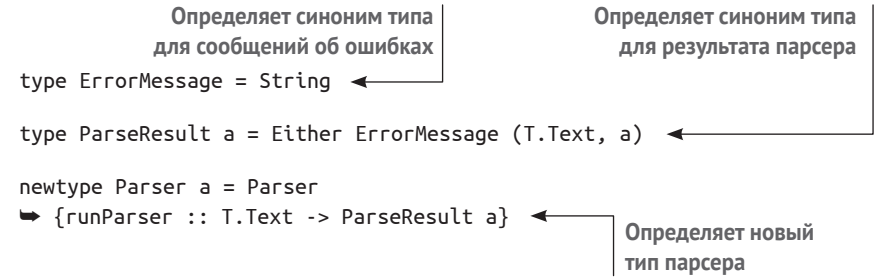


Рис. 12.3 Показано, какие части разобранной информации из файла порождают значения, а какие игнорируются

Наш парсер `magicNumberP` умеет разбирать первую часть заголовка, но после него мы уже не можем разобрать прочие части файла. А все из-за того, что парсер не возвращает остаток входных данных, чтобы мы могли продолжить разбор. Эту проблему можно решить,

разрешив парсеру возвращать не только результат, но и текст, оставшийся после разбора. И этот текст мы сможем передать другому парсеру. Окончательный тип нашего парсера показан в листинге ниже. Тип `ParseResult` содержит остаток входных данных и результат разбора. Кроме того, мы обернем парсер конструкцией `newtype` с полем для доступа к внутренней функции.

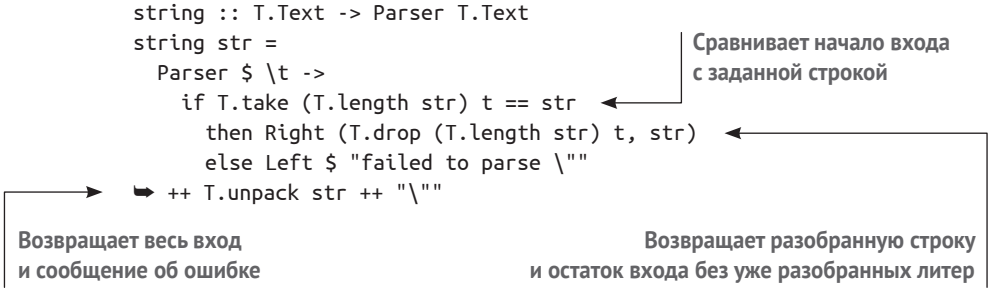
Листинг 12.3 Тип данных для парсеров



Далее в этой главе мы предполагаем, что наши модули импортируют модуль `Data.Text` с квалифицированным именем `T`. Мы могли бы уже сейчас переписать `magicNumerP`, но прежде должны подумать о базовых строительных блоках нашего парсера. Мы уже знаем, что должны выделять из фиксированных строк сигнатуры, так почему бы не создать общий парсер для строк?

Для этого можно написать функцию, которая читает столько литер, какова длина указанной строки, и сравнивает их с этой строкой. Если строки равны, то литеры можно удалить из входных данных и вернуть результат. Эта функция показана в листинге ниже.

Листинг 12.4 Парсер выделения фиксированных строк



Этот парсер можно применить к разбору сигнатур в заголовке:

```
ghci> runParser (string "P1") "P1 100 100"
(" 100 100",Right "P1")
ghci> runParser (string "P1") "P2 100 100"
("P2 100 100",Left "failed to parse \"P1\"")
```

Мы видим, что парсер завершается неудачно, если вход не начинается с заданной строки, и что он возвращает остаток входа в случае успеха. Однако этого недостаточно для определения парсера `MagicNumber`, потому что разобранную строку нужно еще преобразовать в тип Haskell. Что нам нужно, так это преобразовать наш `Parser Text` в `Parser MagicNumber`. Это похоже на то, что умеет делать `Functor`, и на самом деле экземпляр `Functor Parser` – именно то, что здесь необходимо, потому что мы хотим преобразовать внутреннее значение парсера.

В определении функции `fmap`, которая необходима для этого экземпляра, мы можем воспользоваться тем фактом, что экземпляры `Functor Either` и `Functor ((,) a)` уже существуют. Определение экземпляра показано в листинге ниже.

Листинг 12.5 Экземпляр функтора для парсеров

```
instance Functor Parser where
  fmap f p = Parser $ \t -> fmap (fmap f) (runParser p t)
```

Выполняет парсер для некоторого входа и отображает заданную функцию только в правильные результаты

Почему это работает? Поскольку `fmap` применяется к значениям `Right` типа `Either` и к последнему значению в кортеже, мы можем преобразовать тип и значение результата разбора:

```
ghci> fmap (+1) (1,1) :: (Int, Int)
(1,2)
ghci> fmap (+1) (Right 1) :: Either Int Int
Right 2
ghci> fmap (+1) (Left 1) :: Either Int Int
Left 1
```

Напомним, что `Right` означает, что наш парсер успешно завершил разбор, и второе значение в кортеже – это результат разбора. Поэтому внешняя `fmap` применяется к типу `Either`, а внутренняя `fmap f` – к кортежу внутри конструктора `Right`.

Для наших сигнатур мы можем определить такие парсеры:

```
ghci> p1P = fmap (const P1) $ string "P1"
ghci> :t p1P
p1P :: Parser MagicNumber
ghci> runParser p1P "P1 100 100"
Right (" 100 100",P1)
ghci> runParser p1P "P2 100 100"
Left "failed to parse \"P1\""
```

Всякий раз, как мы хотим применить выражение вида `fmap (const X)`, мы можем упростить его до оператора из класса типов `Functor`, кото-

рый именно это и делает. Оператор (<$) предоставляет постоянное значение для функторного отображения. С его помощью мы можем определить наши парсеры:

```
magicNumberP1P :: Parser MagicNumber
magicNumberP1P = P1 <$ string "P1"

...

magicNumberP6P :: Parser MagicNumber
magicNumberP6P = P6 <$ string "P6"
```

Теперь у нас есть парсер для каждой сигнатуры. Чтобы разобрать весь заголовок файла, мы должны подумать о комбинировании парсеров. Как из этих разрозненных частей построить более крупную структуру?

12.1.3 Композиция эффектов

Предположим, что мы пытаемся разбирать файлы, в начале которых может быть не любая сигнатура, а только фиксированная, скажем P1. В этом заголовке нет максимальной величины для данных изображения, поэтому заголовок состоит только из сигнатуры, ширины и высоты. Для простоты забудем о комментариях и предположим, что заголовок состоит из одной строки, в которой значения разделены пробелами. Тогда заголовок имеет вид:

```
P1 100 100
```

Чтобы разобрать произвольное число пробелов, мы можем построить парсер, ведущий себя как функция takeWhile, известная нам по модулям Data.List и Data.Text, которая читает вход до тех пор, пока для литер остается истинным некоторый предикат. Поскольку наши функции работают со значениями типа Text, мы можем использовать функции из Data.Text для построения парсера. Это показано в следующем листинге.

Листинг 12.6 Парсер, который читает вход, пока выполняется предикат

```
takeWhile :: (Char -> Bool) -> Parser T.Text
takeWhile p = Parser $ \t -> Right
➥ (T.dropWhile p t, T.takeWhile p t)
```

Читает литеры, пока выполняется заданный предикат, и удаляет прочитанные литеры из остатка входа, подлежащего дальнейшему разбору

С их помощью мы можем сконструировать парсеры пробелов, а также строк, состоящих из одних цифр:

```
spaces :: Parser T.Text
spaces = takeWhile (' ' ==)
```

Используя `takeWhile` и `fmap`, мы также можем определить парсер целых чисел:

```
integer :: Parser Integer
integer = read . T.unpack <$> takeWhile (`elem` ['0' .. '9'])
```

Проблема в том, что этот парсер аварийно завершает программу, если вход пуст. Улучшенный парсер числовых значений мы напишем позже.

Теперь мы можем определить тип для заголовка:

```
data Header = Header
  { width :: Integer,
    height :: Integer
  }
  deriving (Show)
```

Мы хотели бы создать `Parser Header` и придумать, как комбинировать компоненты `spaces` и `integer`, чтобы затем построить парсер для фактического типа данных. Спецификация ожидаемой структуры заголовка могла бы выглядеть так:

```
Header <_> (string "P1" <_> spaces <_> integer) <_> (spaces <_> integer)
```

Это позволило бы нам указать, как читать сигнатуру, ширину и высоту и какие результаты разбора будут использоваться в качестве полей типа данных `Header`. Оператор (`<_>`) – просто маркер. Нам нужно будет придумать, какие операторы подставить вместо него. Но в любом случае они должны будут распространять ошибки из любого частичного парсера в конечный.

Чтобы лучше понять, в чем проблема, посмотрим, что такое `Header`. Это конструктор, но также функция типа `Integer → Integer → Header`. Представим на секунду, что мы могли заменить тип на `Parser Integer → Parser Integer → Parser Header`. Это позволило бы нам подставить `integer` вместо обоих аргументов, и в итоге получился бы `Parser Header`.

Отсюда можно сделать вывод, что нам нужно организовать последовательность применения парсеров и собрать результаты. Под *сбором* мы имеем в виду любой вид агрегирования результатов, в т. ч. игнорирование каких-то результатов или какой-то способ их комбинирования. По существу, мы хотим применить функцию (в данном случае `Header`) к нескольким значениям `Parser`.

Класс `Functor` уже предоставляет нам функцию `fmap` для применения унарной функции к `Parser`. А что произойдет, если мы применим `Header` к `integer`?

```
ghci> :t Header <$> integer
Header <$> integer :: Parser (Integer -> Header)
```

Получается `Parser` с функцией в качестве внутреннего типа (напомним, что оператор `<$>` – просто инфиксная версия `fmap`). Как теперь продолжить? Как можно применить этот парсер? Здесь в игру вступает новый класс типов: `Applicative`.

Аппликативные функторы, или просто *аппликативы*, – это обобщение функторов, при котором разрешено применение функции в функторном контексте к другому значению в этом контексте, что позволяет выстраивать последовательности таких операций. Чтобы было яснее, рассмотрим определение класса `Applicative`:

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
  {-# MINIMAL pure, ((<*>) | liftA2) #-}
```

Самые важные функции в этом классе типов – `pure` и `(<*>)`. `pure` позволяет взять значение и поместить его в функторный контекст, обозначенный `f`. `(<*>)` применяет функцию в этом контексте к значению в нем. По существу, мы могли бы переписать тип этого оператора как `f (a → b) → (f a → f b)`, что означало бы, что оператор преобразует функцию внутри контекста в функцию вне его. Именно в этом и состоит особенность *аппликативных функторов*. *Функтор* может преобразовать обычную функцию в функцию, работающую с функториально обернутыми значениями, как показано на рис. 12.4.

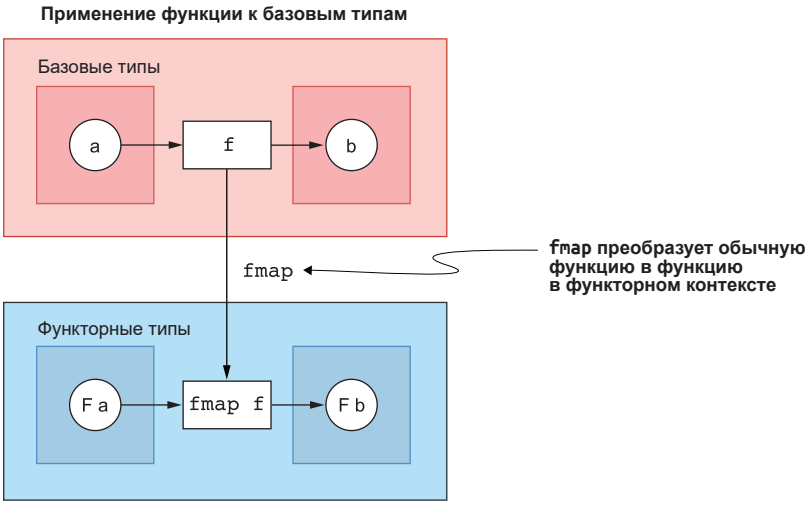


Рис. 12.4 Абстрактный взгляд на функциональность функторов

С другой стороны, аппликативы берут функцию в функторном контексте и опускают ее в обычный контекст, где с ней возможна композиция. Это показано на рис. 12.5.

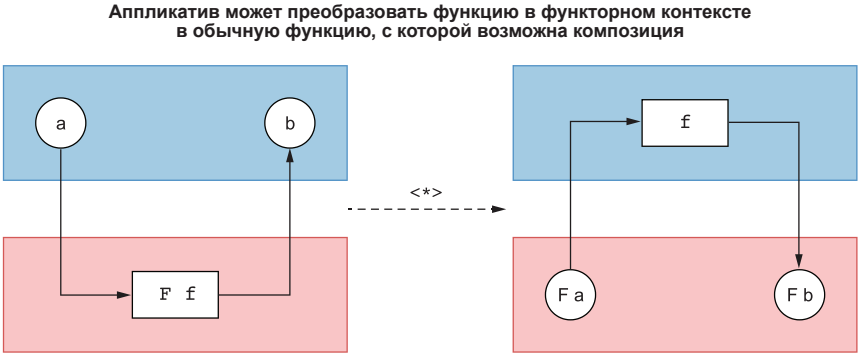


Рис. 12.5 Абстрактный взгляд на функциональность аппликативов

Прежде чем начать беспокоиться о других функциях, рассмотрим пример. Тип `Maybe` имеет экземпляр `Applicative`, где `(<*>)` применяет функцию, если она обернута конструктором `Just`, к обернутому значению. Если какое-либо из значений `Maybe` равно `Nothing`, то конечным результатом также будет `Nothing`:

```
ghci> Just (+1) <*> Just 1 :: Maybe Int
Just 2
ghci> Just (+1) <*> Nothing :: Maybe Int
Nothing
ghci> Nothing <*> Just 1 :: Maybe Int
Nothing
ghci> Nothing <*> Nothing :: Maybe Int
Nothing
ghci> pure 100 :: Maybe Int
Just 100
```

Поскольку мы теперь можем без труда компоновать функции, то можем применять их и к другим значениям в контексте `Maybe`:

```
ghci> add3 x y z = x + y + z
ghci> pure add3 <*> Just 1 <*> Just 2 <*> Just 3 :: Maybe Int
Just 6
```

Функция `pure` для поднятия функции в контекст встречается весьма часто, однако благодаря использованию частичного применения функции (как мы видели в нашем примере `Header`) и `fmap` удается добиться чуть более понятного синтаксиса:

```
ghci> add3 <$> Just 1 <*> Just 2 <*> Just 3 :: Maybe Int
Just 6
```

Но какие еще есть функции в классе `Applicative`? `liftA2` поднимает любую бинарную функцию, так что она работает с аргументами в функторном контексте:

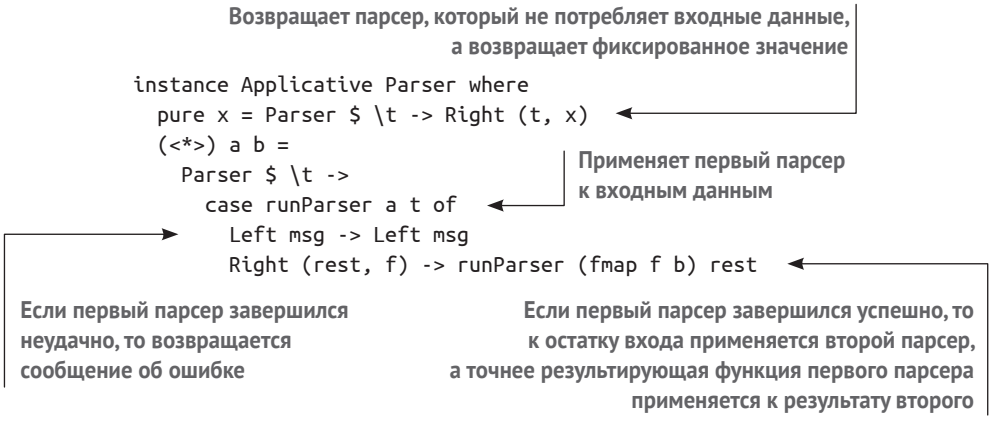
```
ghci> liftA2 (+) (Just 1) (Just 2) :: Maybe Int
Just 3
```

Особенность `(<*)` и `(*>)` в том, что они отбрасывают соответственно первый или второй аргумент, но при этом вычисляют его эффекты. Под этим мы понимаем, что результат этого аргумента существен для конечного результата. В случае `Maybe`, если какой-то аргумент оказывается равным `Nothing` в процессе выстраивания их в последовательность, то и весь терм будет равен `Nothing`. Это верно и для следующих отбрасывающих версий `(<*>)`:

```
ghci> Just (1 :: Int) <* Just (2 :: Int)
Just 1
ghci> Just 1 <* Nothing :: Maybe Int
Nothing
```

Какое отношение это имеет к нам? Класс `Applicative` дает возможность строить последовательность парсеров. Входные данные будут разбираться путем последовательного применения парсеров к входу, оставленному предыдущими парсерами. На первый взгляд может показаться странным, что мы должны писать функцию для `Parser` от значения функции, но все обретает смысл, стоит только применить ее. В следующем листинге показан код экземпляра `Applicative Parser`.

Листинг 12.7 Экземпляр `Applicative` для типа парсера



`pure` создает парсер, который вообще не потребляет входные данные, а возвращает фиксированное значение. Оператор `(<*>)` сначала применяется к входу первого парсера; если этот шаг разбора

неудачен, то мы возвращаем `ErrorMessage` и остаток входа. В противном случае для остатка выполняется второй парсер, который применяет функцию, возвращенную первым парсером, с помощью `fmap`.

Теперь можно поэкспериментировать с последовательностью парсеров. Например, мы можем нормализовать пропуск между словами:

```
ghci> :{
ghci| helloWorldP = (\x _ z -> x <> " " <> z)
ghci|       <$> string "Hello"
ghci|       <*> spaces
ghci|       <*> string "World"
ghci| :}
ghci> runParser helloWorldP "Hello World"
Right ("","Hello World")
ghci> runParser helloWorldP "Hello      World"
Right ("","Hello World")
ghci> runParser helloWorldP "Helloooo World"
Left "failed to parse \"World\""
```

Здесь мы видим, как каждый парсер работает с входом, оставленным предыдущим парсером. Такое же определение можно было бы написать и с `(<*)`:

```
ghci> :{
ghci| hw = (\x y -> x <> " " <> y)
ghci|       <$> string "Hello"
ghci|       <* spaces
ghci|       <*> string "World"
ghci| :}
```

И теперь мы наконец-то можем дописать до конца парсер заголовка. Для этого нам понадобятся написанные ранее `Header` и `(<$>)`, а также оператор `(<*>)`:

```
ghci> :{
ghci| headerP = Header
ghci|           <$> (string "P1" *> spaces *> integer)
ghci|           <*> (spaces *> integer)
ghci| :}
ghci> runParser headerP "P1 100 200"
Right ("",Header {width = 100, height = 200})
ghci> runParser headerP "P2 100 200"
Left "failed to parse \"P1\""
ghci> runParser headerP "P1 100 "
Right ("",Header {width = 100, height = *** Exception:
➥ Prelude.read: no parse
```

Такой *аппликативный* стиль написания парсеров широко распространен, а значит, заслуживает более внимательного изучения.

Класс `Applicative` (как и `Functor`) часто встречается при чтении кода на Haskell. Он так популярен благодаря способности комбинировать эффекты, что помогает писать код, сохраняющий возможность композиции, но при этом строго соблюдающий семантические правила. И все это заслуга одной лишь системы типов! Многие встречавшиеся нам ранее типы, в частности `Maybe`, `Either` и даже `IO`, имеют экземпляры этого класса.

Упражнение. Моноидальные и аппликативные функторы

Аппликативные функторы эквивалентны так называемым *(слабым) моноидальным функторам*. Мы можем использовать их, чтобы объединить два функторных значения в кортеж, тем самым скомбинировав их:

```
class Functor f => Monoidal f where
  unit :: f ()
  (**) :: f a -> f b -> f (a, b)
```

Чтобы этот класс был эквивалентен `Applicative`, должна быть возможность преобразовать любой экземпляр одного класса в другой. Покажите, что это возможно!

Итак, мы написали парсер упрощенного заголовка. Но пока еще наш парсер понимает только одну сигнатуру. В нашем примере мы также могли бы разрешить сигнатуру `P4`, но при этом парсер выдал бы ошибку. Нам нужен способ кодирования альтернатив в парсере. Ответ дает еще один класс типов.

12.1.4 Выбор варианта

Реализацию `(<*>)` в нашем парсере можно было интерпретировать как семантику *и затем*. Первый парсер разбирает вход, и затем то же самое делает второй. Но чтобы представить в коде альтернативы, нам нужно реализовать семантику *а иначе*. Это значит, что если первый парсер завершается ошибкой, то те же самые входные данные (из которых ничего не изъято) должны быть переданы второму парсеру, который, хочется надеяться, разберет их успешно.

Именно такая семантика требуется от экземпляров класса типов `Alternative`. Взгляните на его определение:

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
  some :: f a -> f [a]
  many :: f a -> f [a]
  {-# MINIMAL empty, (<|>) #-}
```

Минимальное определение (состоящее из `empty` и `(<|>)`) подозрительно похоже на определение `Monoid`, да они и вправду аналогичны! `empty` – нейтральный элемент для `(<|>)` – ассоциативной бинарной операции для `Alternative`. На самом деле `Alternative` – это моноид на множестве аппликативных функторов!

Если `(<*>)` выдает ошибку, когда ошибку выдает его единственный аргумент, то `(<|>)` выдает ошибку, только если ошибку выдают оба аргумента. Поэкспериментируем с этой семантикой, снова воспользовавшись типом `Maybe`:

```
ghci> Just 1 <|> Just 2 :: Maybe Int
Just 1
ghci> Just 1 <|> Nothing :: Maybe Int
Just 1
ghci> Nothing <|> Just 2 :: Maybe Int
Just 2
ghci> Nothing <|> Nothing :: Maybe Int
Nothing
```

Если рассматривать `Nothing` как «ошибку», то теперь мы понимаем, как должен вести себя наш парсер. Чтобы он выдавал более осмысленные сообщения об ошибках, мы можем написать функцию, модифицирующую сообщение парсера об ошибке. Она показана в листинге ниже.

Листинг 12.8 Функция, модифицирующая сообщение парсера об ошибке разбора

```
modifyErrorMessage ::
  (ErrorMessage -> ErrorMessage) ->
  Parser a ->
  Parser a
modifyErrorMessage f p =
  Parser $ \t -> case runParser p t of
    Left msg -> Left $ f msg
    result -> result
```

Выполняет парсер для заданного входа

Модифицирует сообщение об ошибке, если разбор был неудачен

Успешный результат возвращается без изменения

Имея эту функцию, мы можем создать экземпляр `Alternative Parser`, показанный в листинге 12.9. Чтобы функция `empty` была нейтральным элементом, она должна всегда возвращать ошибку, поэтому мы можем написать парсер, который даже не пытается потребить вход, а всегда возвращает какое-то значение `Left` в качестве своего результата. Что касается `(<|>)`, то мы должны выполнить первый парсер, а в случае неудачи выполнить второй, используя функцию `modifyErrorMessage` для модификации возможного сообщения об ошибке путем приписывания в его начало сообщения первого парсера.

Листинг 12.9 Экземпляр `Alternative` для парсера

```
instance Alternative Parser where
  empty = Parser $ \_ -> Left "empty alternative"
  (<|>) a b =
    Parser $ \t ->
      case runParser a t of
        Left msg -> runParser (modErr msg b) t
        right -> right
    where
      modErr msg =
        modifyErrorMessage
➥ (\msg' -> msg ++ " and " ++ msg')
```

Выполняет первый парсер

Выполняет второй парсер, модифицируя сообщение об ошибке в случае неудачи

Возвращает результат, если первый парсер завершил разбор успешно

Присовокупляет заданное сообщение об ошибке к модифицируемому сообщению

Имея такой экземпляр, мы теперь можем закодировать выбор. Интересно, что так можно поступить с глубоко вложенными парсерами. То есть если парсер успешно разобрал значительную часть входа, а потом обнаружил, что дальше продолжать не может, то оператор (`<|>`) позволяет откатиться и дать шанс другому парсеру. Рассмотрим пример:

```
ghci> import Control.Applicative
ghci> runParser (string "A" <|> string "B") "A"
Right ("","A")
ghci> runParser (string "A" <|> string "B") "B"
Right ("","B")
ghci> runParser (string "A" <|> string "B") "C"
Left "failed to parse \"A\" and failed to parse \"B\""
```

Обратите внимание на импорт модуля `Control.Applicative`. Все рассмотренные выше классы (`Functor`, `Applicative` и `Alternative`) определены там. Этот импорт необходим нам, чтобы можно было использовать оператор (`<|>`).

Экземпляр `Alternative` невероятно полезен сам по себе, но ко всему прочему он предоставляет еще две функции: `some` и `many`. Обе повторно применяют заданный аппликативный функтор, пока это еще возможно, – в нашем случае до тех пор, пока разбор не завершится неудачно. Результаты собираются в список. Единственное различие между этими функциями состоит в том, что `many` никогда не возвращает ошибку. Если первое же применение оказывается неудачным, то возвращается пустой список. С другой стороны, вызов `some` считается успешным, только если было хотя бы одно успешное применение. Поэтому `some` означает *один или больше*, а `many` – *нуль или больше*.

Упражнение. Many и some

Функцию `many` можно определить следующим образом:

```
many f = some f <|> pure []
```

Либо мы получаем один или более элементов, либо возвращаем пустой список. А как можно определить `some`? Попробуйте реализовать ее.

С помощью этих двух функций мы теперь можем реализовать парсер, который разбирает некоторую лексему сколь угодно много раз:

```
ghci> runParser (many $ string "A") "AAA"
Right ("",["A","A","A"])
ghci> runParser (many $ string "A") ""
Right ("",[])
ghci> runParser (some $ string "A") "AAA"
Right ("",["A","A","A"])
ghci> runParser (some $ string "A") ""
Left "failed to parse \"A\""
```

Вспомните наше определение функции `spaces`, которая была реализована с помощью `takeWhile`. Если бы мы хотели создать парсер, гарантирующий, что входные данные содержат по меньшей мере один пробел, то без всякого труда могли бы сделать это с помощью `some`, как показано в следующем листинге.

Листинг 12.10 Парсер, разбирающий как минимум один пробел

```
someSpaces :: Parser T.Text
someSpaces = T.concat <$> some (string " ")
```

Разбирает по меньшей мере один пробел и конкатенирует результирующие строки в одну большую, иначе завершается с ошибкой

И последнее, что мы должны обсудить, – как записывать случаи, когда имеется много альтернативных парсеров. Действительно ли мы хотим комбинировать их с помощью оператора (`<|>`) в один большой терм? Лучше было бы просто определить список парсеров, а затем пробовать один за другим. Это позволяет сделать функция `asum` из модуля `Data.Foldable`. Она сворачивает заданную структуру, применяя семантику экземпляра `Alternative` для данного типа:

```
ghci> :i asum
asum :: (Foldable t, Alternative f) => t (f a) -> f a
ghci> runParser (asum [string "A", string "B"]) "A"
Right ("","A")
```

```
ghci> runParser (asum [string "A", string "B"]) "B"
Right ("","B")
ghci> runParser (asum [string "A", string "B"]) "C"
Left "failed to parse \"A\" and failed to parse \"B\" and
➥ empty alternative")
```

Отметим, что сообщение об ошибке заканчивается строкой `empty alternative`. Это сообщение об ошибке, выдаваемое нейтральным элементом нашей реализации `Alternative Parser`! Функция `asum` действительно комбинирует все элементы структуры с семантикой моноида, заданной классом `Alternative`, т. е. сворачивает их оператором (`<|>`), считая последним элемент `empty` (каковой является умолчанием для пустой структуры).

> **ПРЕДУПРЕЖДЕНИЕ** Функции `some` и `many` определены в терминах функций, определенных в классах `Alternative` и `Applicative`. Передав им константный аргумент, который всегда завершается успешно, вы зациклите программу! Пример – `some (Just 1)`.

Мы можем написать псевдоним этой удобной функции, чтобы сделать свой код понятнее:

```
choice :: [Parser a] -> Parser a
choice = asum
```

С альтернативами мы разобрались, и осталось положить последний кусочек пазла. Разбор заголовка PNM-файла уже почти возможен. Мы умеем кодировать выбор сигнатуры, пропускать любое число пробелов и читать целые числа. Не хватает совсем немного – как сказать, что разбирать максимальную величину точек данных нужно, только если сигнатура отлична от `P1` и `P4`. Чтобы решить проблемы компонуемости и обработки ошибок, мы перешли от `Functor` к `Applicative`. Однако проблема, которая стоит перед нами сейчас, требует еще одного шага.

12.1.5 Введение в монады

Пока что наш парсер зависит только от читаемых данных, но не от результатов предыдущих парсеров. Поэтому практически невозможно заставить парсер учитывать в своих действиях контекст и дополнительную информацию, содержащуюся в уже разобранных данных. Рассмотрим проблему, возникающую при разборе заголовка PNM-файла:

```
P2 3 2 1
1 0 1
0 1 0
```

Этот заголовок описывает полутоновое изображение с максимальной величиной `1`. Следовательно, единственно возможные значения – `1` и `0`. Парсер такого файла должен прочитать сигнатуру, сделать на ее основе вывод о необходимости прочитать максимальную величину, а затем перейти к чтению данных. Если же сигнатура равна `P1`, то чтение максимальной величины пропускается, и сразу читаются данные. Нам нужно различать случаи в зависимости от прочитанной сигнатуры. Сначала создадим более полный тип для заголовка PNM-файла, показанный в листинге ниже.

Листинг 12.11 Тип для заголовка PNM-файла

```
data Header = Header
  { magicNumber :: MagicNumber,
    width :: Integer,
    height :: Integer,
    maxVal :: Maybe Integer
  }
  deriving (Eq, Show)
```

Определяет поле сигнатуры заголовка

Определяет поля ширины и высоты изображения

Определяет поле необязательной максимальной величины данных

Выводит экземпляры Eq и Show

Мы должны разобрать сигнатуру, ширину и высоту. А дальше есть выбор. Либо мы не разбираем максимальную величину, потому что она не нужна, либо читаем ее. Решение зависит от ранее разобранной сигнатуры.

Однако *аппликативы не позволяют этого сделать*! Они дают возможность компоновать парсеры, но не различать варианты. И изменить поведение в зависимости от уже прочитанных значений так не получится. Чтобы решить проблему, нам придется ввести еще одно понятие. *Аппликативы* были обобщением *функторов*, а теперь мы рассмотрим обобщение самих *аппликативов*: класс *Monad*.

Монады – центральная концепция в Haskell и, пожалуй, одна из самых полезных и интересных концепций функционального программирования вообще. Они позволяют обертывать значения монадическим типом и обрабатывать их произвольными функциями. Как и *аппликативы*, *монады* допускают композицию этих функций в соответствии с правилами, которые мы можем определить для нашего сценария. Рассмотрим класс типов:

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  {-# MINIMAL (>>=) #-}
```

В Haskell все монады одновременно являются *аппликативными функторами*. Самой важной и единственно обязательной для экзем-

пляра Monad функцией является (>>=), в разговорной речи называемая bind. Она получает монадическое значение m a и функцию, которая действует на значение внутри этого монадического значения, возвращая новое монадическое значение m b, возможно, другого типа. Это значение возвращается из функции bind. Вторая функция (>>) похожа на (>>=). Она вычисляет первое монадическое значение, но игнорирует результат и возвращает свой второй аргумент. Функция return не просто похожа на pure из класса Applicative, она, по существу, и есть pure, обертывающая значение монадическим значением.

> **ПРИМЕЧАНИЕ** Раньше мы называли результаты pure значениями в функторном контексте. Теперь мы называем их монадическими значениями, т. к. эти термины взаимозаменяемы. Иногда мы называем монадические значения m a *действиями*, потому что потенциально они могут изменять какое-то состояние в стоящей за ними монадической структуре. Эти действия могут производить *эффекты*. Важно различать вычисление и невычисление действия, потому что разница может оказаться весьма существенной.

Чтобы лучше прочувствовать этот класс, снова рассмотрим пример экземпляра – Monad Maybe:

```
ghci> return 1 :: Maybe Int
Just 1
ghci> (Just 1) >>= (\x -> return (x + 1)) :: Maybe Int
Just 2
ghci> (Just 1) >>= (\x -> return (x + 1)) >>= (\x -> return (x * 2))
➥ :: Maybe Int
Just 4
ghci> Nothing >>= (\x -> return (x + 1)) :: Maybe Int
Nothing
```

Как (<*>) порождает Nothing, если аргумент равен Nothing, так и (>>=) делает то же самое для первого аргумента! Здесь мы видим, как можно использовать функцию bind для доступа к значениям внутри монады (в данном случае Maybe). Функция, переданная bind, может иметь сколь угодно сложное поведение и использовать любые языковые средства, при условии что сигнатуры типов соответствуют требованиям. Это означает, что мы можем использовать bind внутри функции, переданной bind, что позволяет комбинировать произвольное число значений, которые были обернуты монадическим типом. Например, взгляните на функцию, показанную в следующем листинге.

Листинг 12.12 Функция для монадического сложения числовых значений

```
monadd :: (Num a, Monad m) => m a -> m a -> m a
monadd xm ym =
  xm >>=          | Привязывает значение внутри
    (\x ->        | монадического значения xm к x
      ym >>=      | Привязывает значение внутри
        (\y ->    | монадического значения ym к y
          return $ x + y  <-- Возвращает сумму x и y в качестве
        )                     нового монадического значения
    )
```

Эта функция получает два монадических значения, разворачивает их с помощью (>>=) и возвращает их сумму в виде нового монадического значения:

```
ghci> monadd (Just 1) (Just 2) :: Maybe Int
Just 3
ghci> monadd Nothing (Just 2) :: Maybe Int
Nothing
ghci> monadd (Just 1) Nothing :: Maybe Int
Nothing
ghci> monadd Nothing Nothing :: Maybe Int
Nothing
```

Пользуясь функцией bind, мы можем строить и компоновать произвольную логику при работе с вышеупомянутыми значениями. И стоит отметить, что нам нет нужды беспокоиться о состояниях ошибки. Как видно из листинга 12.12, можно не думать о том, что значение равно `Nothing` (в данном случае – при использовании `Maybe`). Функция bind гарантирует, что доступ к значению имеется, потому что в противном случае результатом вычисления терма просто является `Nothing`. Это свойство делает монады отличным кандидатом для обработки *побочных эффектов*.

> **ПРИМЕЧАНИЕ** Иногда говорят, что Haskell свободен от побочных эффектов. Это не так. Если бы это было правдой, то в Haskell не было бы ввода-вывода. Просто в Haskell нет произвольных побочных эффектов, а есть только контролируемые, которые четко отделены от чистого кода системой типов!

На самом деле мы работаем с монадами еще с главы 3! Ведь `IO` – тоже монада. Побочные эффекты, порождаемые при вводе-выводе, контролируются и обрабатываются внутри функции bind. Функция `return`, которую мы использовали совместно с вводом-выводом, –

это в точности `return` из класса типов `Monad`. Оказывается, что мы использовали функцию bind для ввода-вывода, даже не подозревая об этом. `do`-нотация, предоставляемая Haskell, – синтаксический сахар для функций bind!

```
m >>= f
```

Этот простой терм эквивалентен

```
do x <- m
   f x
```

Другую функцию bind транслировать еще проще:

```
m1 >> m2
```

Этот терм превращается в

```
do _ <- m1
   m2
```

Обратите внимание, как хорошо в этом фрагменте видно, что первый аргумент будет вычисляться. Понимая связь между bind и `do`-нотацией, мы легко перепишем пример из листинга 12.12, используя `do`-нотацию:

```
monadd' :: (Num a, Monad m) => m a -> m a -> m a
monadd' xm ym = do
  x <- xm
  y <- ym
  return $ x + y
```

`do`-нотация проясняет, когда и как используются значения, полученные вычислением монадических значений.

Прежде чем переходить к более глубокому обсуждению монад, следует взглянуть, как их можно использовать в нашем парсере. Для создания экземпляра `Monad Parser` мы должны определить функции `return` и `(>>=)`. Для реализации `return` мы можем просто использовать `pure` из нашего экземпляра `Applicative`. Функция bind должна вычислить парсер, переданный в качестве первого аргумента, а затем функцию, переданную в качестве второго аргумента, передав ей результат парсера, и, наконец, выполнить парсер, являющийся результатом вычисления этой функции, для оставшихся входных данных. Это показано в следующем листинге. Если первый парсер завершается неудачно, то мы должны закоротить выполнение, потому что не существует значения, которое можно было бы передать функции. Поэтому мы просто возвращаем в этой точке ошибку.

Листинг 12.13 Экземпляр `Monad` для типа парсера

```
instance Monad Parser where
  return = pure
  (>>=) p f =
    Parser $ \t ->
      case runParser p t of
        Left msg -> Left msg
        Right (rest, x) -> runParser (f x) rest
```

Определяет return как pure из экземпляра Applicative

Выполняет первый парсер для заданного входа

Возвращает ошибку разбора, если первый парсер завершился неудачно

Применяет заданную функцию к результату первого парсера и выполняет результирующий парсер

Теперь мы можем воспользоваться do-нотацией для записи наших парсеров!

```
helloWorldP = do
  h <- string "Hello"
  _ <- someSpaces
  w <- string "World"
  return $ h <> " " <> w
```

Сравните это с аппликативной нотацией, показанной выше в этой главе, – функциональность одинакова. Как и следовало ожидать, неудача разбора внутри блока do приводит к отказу всего парсера:

```
ghci> runParser helloWorldP "Hello World"
Right ("","Hello World")
ghci> runParser helloWorldP "Hello World"
Right ("","Hello World")
ghci> runParser helloWorldP "Bye Bye World"
Left "failed to parse \"Hello\""
```

Однако мы вводили монады не для того, чтобы заменить *аппликативы*, а для того, чтобы их расширить. Точнее, нам нужна возможность принимать решения на основе разобранных значений. В частности, мы могли бы преобразовать успешно разобранную строку и тем самым улучшить наш прежний парсер `integer`.

С помощью do-нотации мы можем читать числовые значения так, как делали это и раньше. Затем, используя `readMaybe`, мы можем обнаружить, завершилось ли преобразование в значение типа `Integer` успешно или неудачно. Код при этом мог бы выглядеть, как показано ниже.

Листинг 12.14 Парсер значений типа Integer

```
import Text.Read (readMaybe)

...

integer :: Parser Integer
integer = do
  intStr <- takeWhile (`elem` ['0' .. '9'])
  case readMaybe (T.unpack intStr) of
    Just value -> return value
    Nothing ->
      Parser $ \_ ->
        Left $
          "Could not convert \""
            ++ T.unpack intStr
            ++ "\" as an integer"
```

Читает цифры из входа

Пытается преобразовать прочитанную строку в Integer

В случае успеха возвращает преобразованное значение

В случае неудачи возвращает сообщение об ошибке

Теперь этот парсер безопасно читает литеры из входа и преобразует ошибку преобразования в ошибку разбора. Это можно сделать, потому что монады позволяют воздействовать на значение, получившееся в результате вычисления предшествующего действия.

И наконец-то мы готовы построить парсер заголовка. Мы читаем сигнатуру, по меньшей мере один пробел, ширину, по меньшей мере один пробел, высоту и затем факультативно – в зависимости от сигнатуры – дополнительные пробелы и максимальную величину. Все это делает парсер, показанный в листинге ниже.

Листинг 12.15 Парсер заголовка PNM-файла

```
headerP :: Parser Header
headerP = do
  magicNumber <-
    choice
      [ magicNumberP1P,
        magicNumberP2P,
        magicNumberP3P,
        magicNumberP4P,
        magicNumberP5P,
        magicNumberP6P
      ]
  _ <- someSpaces
  width <- integer
  _ <- someSpaces
  height <- integer
  maxVal <-
    if magicNumber == P1 || magicNumber == P4
      then return Nothing
```

Разбирает любую сигнатуру

Разбирает по меньшей мере один пробел и игнорирует результат

Разбирает числовые значения ширины и высоты

Разбирает по меньшей мере один пробел и игнорирует результат

Разбирает числовые значения ширины и высоты

Возвращает Nothing, если сигнатура показывает, что в заголовке не должно быть максимальной величины

```
    else Just <$> (someSpaces *> integer)
return Header {..}
```

Разбирает по меньшей мере один пробел, а затем Integer, которое обертывается конструктором Just

Конструирует полное значение типа Header по разобранным данным

В представленном здесь парсере заголовков используются все рассмотренные к этому моменту концепции. Парсеры сигнатур, введенные выше в этой главе, комбинируются с помощью `choice`. Выбирается первый в списке парсер, который сумел разобрать сигнатуру. Поскольку они разбирают разные строки и никакой неоднозначности быть не может, порядок их записи безразличен. Однако если бы один парсер разбирал только префикс другого парсера, то порядок был бы важен. Для разбора пробелов между значениями используется парсер `someSpaces`, но сами пробелы игнорируются.

> **ПРИМЕЧАНИЕ** Мы используем `_ ← someSpaces` для игнорирования пробелов, указывая, что результирующее значение не следует присваивать никакой переменной, отсюда и `_`. Так поступать необходимо, потому что результат `someSpaces` – не `()`; в противном случае мы могли бы утратить возможность привязки значения. Это можно упростить, воспользовавшись функцией `void` из модуля `Control.Monad`, чтобы отобразить результат в `()`. Следовательно, код можно переписать в виде `void someSpaces`, сделав явным тот факт, что результат действия игнорируется.

Ширина и высота разбираются парсером `integer`, показанным в листинге 12.14. Парсер максимальных величин конструируется в новом блоке `do`. Здесь мы можем вести себя по-разному в зависимости от того, какое значение было разобрано раньше. В случае когда разбирать максимальную величину нужно, мы также разбираем пробелы, но игнорируем их, используя оператор `(*>)` из экземпляра `Applicative`. Стоит отметить, что если любой из этих парсеров завершается ошибкой, то ошибкой завершается и весь парсер. Это важное свойство монад, которое упрощает процесс написания более сложных приложений, поскольку нам не нужно думать об эффектах и ошибках, которые обрабатываются экземпляром класса типов `Monad`.

12.1.6 Обсуждение монад

Этот раздел был бы неполон без дополнительного обсуждения монад. Сделаем несколько замечаний о функторах, аппликативах и монадах:

- они помогают отделять чистый код от эффектообразующего средствами системы типов;

- это единственный метод обработки эффектов;
- не всегда они обладают ссылочной прозрачностью;
- они подчиняются определенным правилам.

Один из самых важных атрибутов аппликативов и монад – то, что они помогают нам управлять эффектами и контролировать их из чистого кода. Если судить только по определениям, то классы, которые мы видели, чистые, но их действия чистыми быть не обязаны. Монада `IO` – самый яркий пример. Внутри нее мы можем взаимодействовать с окружением. А это эффект! Однако это не то, что мы обычно считаем побочным эффектом, потому что он имеет место не внутри наших чистых функций, а вне их – в монаде `IO`. Отсюда следует логический вывод: чтобы написанная на Haskell программа делала что-то полезное, весь наш код должен находиться внутри монады `IO`. Именно поэтому функция `main` имеет тип `IO ()`.

Возникает вопрос: почему мы не можем вырваться из монад? Почему нет функции типа `(Monad m) => m a -> a`, которая давала бы доступ к значению внутри монады из чистого кода? Потому что это противоречило бы самой идее монад! Методы, предоставляемые классом `Monad`, позволяют только вносить значения в монадический контекст (с помощью `return`), а извлекать их можно не иначе, как с помощью функций bind. Это сделано намеренно. Мы не должны иметь возможности вырваться за пределы монады без тщательного контроля над управляемым состоянием.

> **ПРИМЕЧАНИЕ** Структура, которая явно позволяет вырваться из монады, называется *комонадой*, для нее существует отдельный пакет в репозитории Hackage.

В качестве примера рассмотрим монаду `Maybe`. Обычно в случае ошибки она принимает значение `Nothing`. Функцию типа `Maybe a -> a` для этого типа написать невозможно, не сделав ее частичной. На самом деле функция `fromMaybe` дает ту же самую функциональность, но требует значения по умолчанию. Поэтому вырваться из монады `Maybe` можно, но о возможном ошибочном состоянии мы должны позаботиться сами. Функция, позволяющая вырваться из `Maybe` небезопасно, называется `fromJust`, она возвращает ошибку, если значение равно `Nothing`.

Это свойство почти справедливо для ввода-вывода. Функция `unsafePerformIO` может вырваться из монады `IO`. Однако ее использование может привести к неожиданным эффектам, например эффекты могут порождаться не по порядку или применяться более одного раза. В более сложном коде `unsafePerformIO` может даже приводить к ошибкам типизации на этапе выполнения.

> **ПРЕДУПРЕЖДЕНИЕ** Без слов понятно, что использовать функцию `unsafePerformIO` не следует. Большинство решаемых ей проблем можно решить и другими средствами. Если вы ее все-таки используете, то должны хорошо понимать, что делаете, и гарантировать, что это не станет причиной проблем в вашей программе.

Монады – не единственный способ обработки эффектов, а лишь один из возможных. В других языках реализованы собственные системы эффектов, зачастую с частичной поддержкой. Java требует от программиста указывать, какие исключения может возбуждать метод. Затем вызывающая функция должна либо обработать исключение, либо указать, что она тоже может возбуждать его. Это способ обработки эффектов в языке с исключениями.

Одна из причин, по которой мы хотим, чтобы функции были чистыми, – ссылочная прозрачность. В двух словах это означает, что применение функции можно заменить возвращенным ей значением, и результат программы от этого не изменится. Вот простой пример:

```
f x = x + 1
```

Вычисление терма `f 1` всегда дает `2`, поэтому мы спокойно можем подставить `2` вместо `f 1`. Все вхождения `f 1` в программу могут быть заменены на `2`. Если бы все наши функции были ссылочно-прозрачными, то мы могли бы заменить все их применения результатом того применения, которое произошло раньше. К сожалению, не все, что мы делаем, обладает свойством ссылочной прозрачности. Тривиальные примеры – монада `IO` и действия вроде `getLine`. Чтение данных, введенных пользователем, или данных из файла, взаимодействие с сетевым или иным аппаратным интерфейсом, генерирование случайных чисел – все это не ссылочно-прозрачные действия. По счастью, мы можем (и так и делаем) изолировать их в монаде `IO` и других. Монады вроде `Maybe` являются ссылочно-прозрачными, но в общем случае это не так.

> **ПРИМЕЧАНИЕ** У класса `Applicative` больше экземпляров, чем у `Monad`. Об этом важно помнить при написании общих функций для монадических структур. Иногда мы можем написать более общую функцию, предполагая только наличие экземпляра `Applicative`, что позволит применять наш код к большему числу типов.

12.1.7 Как правильно ошибаться

И последнее, что нам нужно обсудить, – это ошибки. Ошибки – неотъемлемая часть человеческой природы, и то же самое справедливо для разбора. Да, разбор может завершиться неудачно. И мы даже мо-

делировали неудачный разбор в наших функциях. Функции `string` и `integer` могут возвращать ошибку. Нельзя ли обобщить это поведение? Можно – с помощью дополнительных классов типов.

Первый такой класс предлагает нам модуль `Control.Monad.Fail`, и называется он `MonadFail`. Он был написан для решения проблемы, которая возникает при сопоставлении с образцом внутри монады. Рассмотрим пример:

```
mhead :: (Monad m) => m [a] -> m a
mhead m = do
  (x : _) <- m
  return x
```

Это определение монадической версии `head` не компилируется! Причина – неявное и неполное сопоставление с образцом для списка, которое может завершиться неудачно. В таком случае неясно, как войти во внутреннее состояние ошибки в монаде. Тут-то и приходит на помощь класс типов `MonadFail`, который предоставляет именно эту функциональность в виде метода `fail`. Класс довольно простой:

```
class Monad m => MonadFail m where
  fail :: String -> m a
```

`fail` получает сообщение об ошибке (которое может по желанию полностью игнорировать) и должен предоставить монадическое значение, реагирующее на ошибку. Для `Maybe` это вполне тривиально: `fail` просто игнорирует сообщение и возвращает `Nothing`.

Теперь мы можем изменить тип нашей монадической функции `head`:

```
mhead :: (MonadFail m) => m [a] -> m a
mhead m = do
  (x : _) <- m
  return x
```

Она компилируется, так что можно подвергнуть ее испытанию:

```
ghci> mhead (return [1]) :: Maybe Int
Just 1
ghci> mhead (return []) :: Maybe Int
Nothing
ghci> mhead (return [1]) :: IO Int
1
ghci> mhead (return []) :: IO Int
*** Exception: user error (Pattern match failure in 'do' block at ...
```

В случае монады `IO` мы получаем исключение. Мы еще не говорили о том, как обрабатывать исключения, но сделаем это в главе 14.

Чтобы обработать это исключение и ошибки вообще, мы можем предоставить для своего типа `Parser` экземпляр `MonadFail`, как показано в листинге ниже. Мы уже видели, как кодировать такое ошибочное состояние в определении `empty`, а как оформлять отказ – в парсере `integer`.

Листинг 12.16 Экземпляр `MonadFail` для типа парсера

```
instance MonadFail Parser where
  fail s = Parser $ \t -> Left s
```

Определяет метод отказа, при котором сообщение об ошибке обертывается типом результата парсера

Это позволяет использовать функцию `fail` всегда, когда мы не можем продолжить разбор. С ее помощью можно сделать код парсера `integer` чище:

```
integer :: Parser Integer
integer = do
  intStr <- takeWhile (`elem` ['0' .. '9'])
  case readMaybe (T.unpack intStr) of
    Just value -> return value
    Nothing ->
      fail $
        "Could not convert \""
          ++ T.unpack intStr
          ++ "\" as an integer"
```

Другой способ наделения монад способностью к отказу дает тип классов `MonadPlus`. Как `Alternative` является моноидом для аппликативных функторов, так `MonadPlus` – моноид для монад! Его определение очень похоже на определение `Alternative`:

```
class (Alternative m, Monad m) => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Оказывается, что реализацией `mzero` по умолчанию является функция `empty`, а реализацией `mplus` по умолчанию – `(<|>)`, обе из типа классов `Alternative`. Если мы готовы использовать эти реализации по умолчанию при определении экземпляра `MonadPlus Parser`, то достаточно одной строки:

```
instance MonadPlus Parser
```

Однако иногда желательно изменить сообщение об ошибке для метода `mzero`. Как это делается, показано в следующем листинге.

Листинг 12.17 Экземпляр MonadPlus для типа парсера

```
import Control.Monad

instance MonadPlus Parser where
  mzero = modifyErrorMessage (const "mzero") empty
```

Возвращает парсер с другим сообщением об ошибке по умолчанию

Использование `empty` в этом определении намеренное, потому что мы хотим отразить изменения в определении `empty` в `mzero`. Обе должны вести себя похоже. `MonadPlus`, как и `Alternative`, позволяет реализовать отказ, выбор одной из монад (`msum`) и фильтрацию по результатам действий (`mfilter`):

```
ghci> import Control.Monad
ghci> smallerTen = mfilter (< 10) integer
ghci> greaterTwenty = mfilter (> 20) integer
ghci> px = msum [smallerTen, greaterTwenty]
ghci> px' = modifyErrorMessage (const "value not < 10 or > 20") px
ghci> runParser px' "1"
("",Right 1)
ghci> runParser px' "100"
("",Right 100)
ghci> runParser px' "15"
("15",Left "value not < 10 or > 20")
```

Заметим, что эти функции работают не только для нашего типа `Parser`. Они работают для *всех монад*. Многие функции, как и `msum` и `mfilter`, определены в модулях `Control.Monad` и `Control.Applicative`, что позволяет лучше осуществлять композицию функциональности.

На этом мы завершаем обсуждение монад. Хотя это кажется отвлечением от основной темы, понимать их очень важно. Монадическое программирование – парадигма, которая помогает держать эффекты под контролем, а Haskell уникален тем, что поддерживает его на уровне ядра языка.

Пока что наш парсер умеет обрабатывать заголовок файла изображения. Пора уже научить его читать весь файл целиком.

12.2 Разбор по-крупному

Мы показали, как писать допускающие композицию и отказоустойчивые парсеры. Кроме того, мы видели, как реализовать парсер заголовков PNM-файлов. Пришло время построить парсер для всего файла изображения, т. е. после разбора заголовка продолжить разбор данных. В процессе разбора мы должны убедиться, что данных не слишком мало и не слишком много и что не превышена максимальная величина, если она была задана в заголовке.

Кроме того, предстоит обсудить проблему, которую мы до сих пор игнорировали. В зависимости от сигнатуры данные могут быть за-

даны не в виде текста, а в виде байтов. Наш парсер этого не поддерживает. Есть и другие ограничения.

- `Parser` не отслеживает текущую позицию в разбираемых входных данных.
  - Поэтому он не знает, сколько данных уже разобрано.
  - В сообщениях об ошибке нет указания места, в котором произошла синтаксическая ошибка.
- Для разбора должен присутствовать весь вход.
  - Поэтому потоковый ввод (из файлов или из сети) не поддерживается.
  - Мы обязаны загрузить все входные данные в память, прежде чем начинать разбор.
- Выбирая срез из входа, мы копируем все оставшиеся данные.
- Поддерживаются только входные данные типа `Text`.
  - Двоичные данные разобрать нельзя.
  - Парсер не может обнаружить проблемы кодировки и завершается с сообщением об ошибке общего характера.

Решение этих проблем не тривиально, поэтому мы переключим внимание на новый вид парсера, который решает их все разом. Проблема грамматического разбора в мире Haskell очень хорошо изучена. Результатом этой работы стало несколько библиотек. Из них наибольшим авторитетом пользуется, пожалуй, библиотека *Parsec*, предлагающая парсер, очень похожий на написанный нами. Основная идея этой библиотеки в том, что монадические парсеры можно построить из *комбинаторов парсеров*. Комбинаторы являются полноправными функциями. Мы уже использовали их, не зная, что они так называются. Это `takeWhile`, `choice` и монадические функции bind.

Упражнение. Реализация комбинаторов

Много полезных комбинаторов имеется в библиотеке *parser-combinators*. Например, в ней есть функция `between`, которая применяет парсеры к началу, середине и концу входа, и `skipMany`, которая пропускает нуль или более успешных применений парсера. Вот несколько примеров:

```
ghci> import Control.Monad.Combinators
ghci> parenInteger = between (string "(") (string ")") integer
ghci> runParser parenInteger "(123)"
("",Right 123)
ghci> runParser parenInteger "(123"
("",Left "failed to parse \")\"")
ghci> skipSpaces = skipMany $ string " "
ghci> integerWithSpaces = between skipSpaces skipSpaces integer
ghci> runParser integerWithSpaces " 123 "
("",Right 123)
```

Посмотрите, какие функции предоставляет библиотека, и попробуйте реализовать их самостоятельно!

К числу самых современных библиотек комбинаторов парсеров для Haskell относятся *Megaparsec* и *Attoparsec*. Megaparsec – швейцарский нож для библиотек такого типа. Ее можно использовать для разбора текста или байтов, она поддерживает очень детальные сообщения об ошибках и предлагает широкий спектр методов для более точного управления поведением парсеров. С другой стороны, Attoparsec, узкоспециализированная библиотека для построения парсеров байтовых данных с упором на производительность. Ее «фишка» – встроенная поддержка инкрементного ввода, благодаря чему большие файлы можно разбирать, не загружая содержимое файла целиком в память с самого начала. Однако следует отметить, что парсеры поддерживают возврат, поэтому после того как парсер увидит все входные данные, они целиком останутся в памяти.

Мы сосредоточимся на библиотеке Attoparsec и с ее помощью напишем парсер PNM-файлов. На самом деле переход будет простым, потому что написанные с применением этой библиотеки парсеры в основных чертах ведут себя, в точности как наш. Как и наша библиотека, Attoparsec поддерживает функцию `string`. Но парсеры работают с типом `ByteString`, а не `Text`. Этот тип предлагает такие же функции, как `Text`, которые входят в пакет `bytestring`, поэтому, подставив один пакет вместо другого, мы можем работать так же, как работали с текстовыми данными.

`ByteString` – списковидная структура данных, элементами которой являются значения типа `Word8`, т. е. 8-разрядные целые без знака. Иными словами, литера – это просто целое число от `0` до `255`. Модуль `Data.ByteString.Internal` предоставляет функции `c2w` и `w2c` для преобразования `Char` в `Word8` и обратно на случай, если мы захотим использовать обычный тип `Char` в сочетании с `ByteString`. На наше счастье, экземпляр `IsString ByteString` уже существует, поэтому мы можем записывать обычные строки и интерпретировать их как `ByteString`.

12.2.1 Введение в библиотеку Attoparsec

Чтобы можно было использовать новый тип и новую библиотеку, мы должны включить пакеты `bytestring` и `attoparsec` в секцию `dependencies` файла `package.yaml`. Сделав это, поэкспериментируем с новыми типами. В Attoparsec парсер состоит из очень простых строительных блоков. Основной тип – `Parser`, он используется для разбора значений типа `ByteString`. И этот тип, и большая часть функций, необходимых для разбора, импортируются из модуля `Data.Attoparsec.ByteString`. Существует также модуль `Data.Attoparsec.Text`, который содержит эквивалентные функции и типы для работы со значениями типа `Text` – как в нашем собственном парсере. Но поскольку нам предстоит работать с двоичными данными, мы выберем первый вариант.

```
ghci> import Data.Attoparsec.ByteString as AP
ghci> AP.parse (AP.string "Hello") "Hello"
Done "" "Hello"
ghci> AP.parse (AP.string "Hello") "hello"
Fail "hello" [] "string"
ghci> AP.parse (AP.string "Hello") "He"
Partial _
```

Мы импортируем в GHCi модуль с квалифицированным именем, чтобы им можно было пользоваться, не опасаясь конфликтов с модулем нашего парсера. Как видим, модуль экспортирует функцию `string` типа `ByteString -> Parser ByteString`, работающую точно так же, как наша функция `string`. Функция `parse` имеет тип `Parser a -> ByteString -> Result a`, т. е. применяет парсер к заданному входу. Тип `Result` немного отличается от нашего. В нем есть три конструктора: `Done` для случая успешного разбора, `Fail` – на случай ошибки и `Partial` – если парсер, строго говоря, не завершился ошибкой, но для завершения разбора ему не хватило данных. `Done` и `Fail` в своем первом поле содержат неразобранный остаток входа, а `Partial` – функцию продолжения, которую можно применить к остатку входа:

```
ghci> Partial c = AP.parse (AP.string "Hello") "He"
ghci> Partial c2 = c "l"
ghci> c2 "lo"
Done "" "Hello"
```

В этом примере мы игнорируем предупреждения компилятора о неполном сопоставлении с образцом. Как видим, функции в частичном результате можно использовать для инкрементного разбора входа. В этой главе такая возможность нас не интересует, мы разбираем только вход целиком, поэтому можем использовать функцию `parseOnly` типа `Parser a -> ByteString -> Either String a`, очень похожую на нашу функцию `runParser`. Работают они одинаково. Результат `parseOnly` не может иметь тип `Partial`, он либо успешный, либо неудачный.

Для построения парсера в нашем распоряжении имеется целый набор функций:

- `word8 :: Word8 -> Parser Word8` – читает конкретный байт из входа;
- `anyWord8 :: Parser Word8` – читает любой байт из входа;
- `string :: ByteString -> Parser ByteString` – читает конкретную строку байтов;
- `skipWhile :: (Word8 -> Bool) -> Parser ()` – пропускает байты, пока заданный предикат удовлетворяется;
- `take :: Int -> Parser ByteString` – читает заданное количество байтов;
- `takeWhile :: (Word8 -> Bool) -> Parser ByteString` – читает байты, пока заданный предикат удовлетворяется.

Как видим, большая часть этих функций специализирована для работы с `Word8`. Если мы хотим вместо него использовать `Char`, то должны импортировать модуль `Data.Attoparsec.ByteString.Char8`, который содержит функции для разбора букв и цифр в кодировке ASCII и вообще упрощает работу со значениями типа `Char`.

12.2.2 Разбор изображений

Теперь можно серьезно подойти к разбору данных изображения. Сначала определим необходимые типы данных. У нас уже есть тип для заголовка, но нужно внести в него небольшое исправление. Максимальная величина данных в PNM-файле равна **65535**, т. е. максимальное 16-разрядное целое без знака. Его можно представить типом `Word16` из модуля `Data.Word`. Поэтому мы заменим тип `Maybe Integer` факультативной максимальной величины на `Maybe Word16`. Сами данные – множество значений типа `Word16`. Мы могли бы поискать более компактный способ хранения значений пикселей, потому что тратить по 16 бит на каждый пиксель растрового изображения расточительно, но сейчас наша цель – не оптимизация каркаса. Типы показаны в листинге ниже.

Листинг 12.18 Типы для разбора PNM-файлов

```
data MagicNumber = P1 | P2 | P3 | P4 | P5 | P6
  deriving (Eq, Show)

data Header = Header
  { magicNumber :: MagicNumber,
    width :: Integer,
    height :: Integer,
    maxVal :: Maybe Word16
  }
  deriving (Eq, Show)

data RawPixel
  = Single Word16
  | RGB Word16 Word16 Word16
  deriving (Eq, Show)

newtype RawData = RawData
  { pixels :: [RawPixel]
  }
  deriving (Eq, Show)

data RawPnm = RawPnm
  { header :: Header,
    imageData :: RawData
  }
  deriving (Eq, Show)
```

Тип для сигнатур

Тип для заголовка PNM-файла

Определяет пиксель, содержащий одно значение (полутоновый или черно-белый)

Определяет пиксель, содержащий значения красного, зеленого и синего каналов

Тип для хранения исходных пикселей

Тип для разобранного PNM-файла

Для построения парсера заголовка с применением Attoparsec необходимо внести небольшое изменение в наш первоначальный парсер. Мы воспользуемся возможностью заменить написанные нами функции теми, что предоставляет библиотека. В листингах ниже предполагается, что импортированы следующие модули:

```
import Data.Attoparsec.ByteString
import qualified Data.Attoparsec.ByteString.Char8 as C8
import qualified Data.ByteString as BS
```

Теперь можно приступить к разбору заголовка. Разбор сигнатуры точно такой же, как в нашей реализации парсера. Поскольку библиотечный тип `Parser` имеет экземпляры `Functor`, `Applicative` и `Monad`, эквивалентные нашим определениям, мы можем использовать оператор `(<$)` для разбора входа и возврата `MagicNumber`, как показано в листинге ниже.

Листинг 12.19 Парсеры для разбора PNM-файлов

```
magicNumberP1P :: Parser MagicNumber
magicNumberP1P = P1 <$ string "P1"

...

magicNumberP6P :: Parser MagicNumber
magicNumberP6P = P6 <$ string "P6"
```

Разбирает «P1» и возвращает значение P1

Разбирает «P6» и возвращает значение P6

Таким образом, мы можем разобрать фиксированную строку и мгновенно транслировать ее в значение Haskell.

> **ПРИМЕЧАНИЕ** Расширение `OverloadedStrings` позволяет опускать использование функции `string` для парсеров фиксированных строк. Достаточно просто написать `P1 <$ "P1"`. Однако это менее понятно, чем явное использование `string`, потому-то мы и решили отказаться от такого короткого пути в этой главе.

Модуль `Data.Attoparsec.ByteString.Char8` также предоставляет замену парсеру `integer` в виде парсера `decimal` типа `Integral a => Parser a`, который разбирает десятичные значения. Проблема в том, как пропустить пробельные символы, к которым в формате PNM относятся `' '`, `\n`, `\r`, `\f`, `\v` и `\t`. Как и раньше, мы можем воспользоваться функцией `satisfy` и проверять, совпадает ли данная литера `Char` с одним из пробельных символов. Однако в некоторых местах требуется наличие хотя бы одного пробельного символа. Для этого можно использовать функцию `skipMany1`, которая применяет заданный парсер по меньшей мере один раз и игнорирует прочитанные значения. Это показано в листинге ниже.

Листинг 12.20 Парсер для пропуска пробельных символов

```
import Control.Monad (void)

...

whitespace :: Parser ()
whitespace =
  void $
    C8.satisfy $
      \c ->
        c == ' '
          || c == '\n'
          || c == '\t'
          || c == '\r'
          || c == '\v'
          || c == '\f'

whitespaces :: Parser ()
whitespaces = skipMany1 whitespace
```

Игнорирует результат, отображая его в ()

Читает один символ, удовлетворяющий предикату

Определяет предикат для пробельных символов в PNM-файлах

Пропускает по меньшей мере один пробельный символ

Далее мы хотим разбирать (и игнорировать) комментарии, которые могут присутствовать в заголовке. Комментарий начинается с литеры `%` и заканчивается знаком новой строки. Для разбора можно воспользоваться парсером типа `char :: Char -> Parser Char`, который читает литеру и вызывает `takeWhile` для чтения литер до встречи знака новой строки. Код показан ниже.

Листинг 12.21 Парсер для комментариев

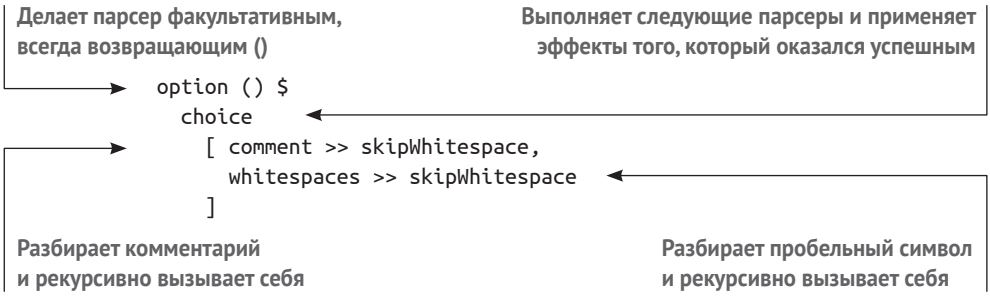
```
comment :: Parser BS.ByteString
comment = C8.char '%' >> C8.takeWhile (/= '\n')
```

Читает литеру %, а затем продолжает чтение, пока не встретится знак новой строки. Возвращает строку, прочитанную после %

Заметим, что этот парсер требует наличия комментария, поэтому если мы захотим использовать его для разбора факультативного комментария, то должны будем игнорировать ошибки. В нашем примере мы хотим игнорировать пробельные символы и комментарии в заголовке. Увы, мы не знаем, когда они встретятся, поэтому должны игнорировать их между сигнатурой и другими читаемыми значениями. Для этого построим еще один парсер, который пропускает лишнюю информацию.

Листинг 12.22 Парсер для пропуска пробельных символов и комментариев

```
skipWhitespace :: Parser ()
skipWhitespace =
```



Как видно из листинга 12.22, построить такой парсер можно, реализовав выбор между чтением комментария или пробельных символов с последующим рекурсивным выбором того же парсера для чтения следующего комментария или множества пробельных символов. Однако что будет, если мы встретим реальные данные, так что оба варианта завершатся неудачно? Чтобы учесть такую возможность, мы воспользуемся функцией `option :: Alternative f => a -> f a -> f a`, которая пытается применить действие во втором аргументе, а если это не получается, предоставляет значение по умолчанию в первом аргументе.

Теперь мы можем скомбинировать все эти парсеры и разобрать новый тип заголовка, показанный в листинге 12.18. Между значимыми элементами может находиться произвольное число комментариев и пробельных символов, которые пропускаются парсером `skipWhitespace`. В конце заголовка мы разбираем один пробельный символ, потому что спецификация требует, чтобы заголовок завершался таковым. На этот раз мы используем функцию `decimal` из модуля `Data.Attoparsec.ByteString.Char8` для разбора значений типа `Word16`. А в остальном отличий от предыдущей реализации немного. Код нового парсера показан в листинге ниже.

Листинг 12.23 Парсер заголовка PNM-файла с применением Attoparsec

```
headerP :: Parser Header
headerP = do
  skipWhitespace
  magicNumber <-                    Разбирает сигнатуру PNM-файла
    choice
      [ magicNumberP1P,
        magicNumberP2P,
        magicNumberP3P,
        magicNumberP4P,
        magicNumberP5P,
        magicNumberP6P
      ]
  skipWhitespace
```

```
width <- C8.decimal
skipWhitespace
height <- C8.decimal
maxVal <- do
  if magicNumber == P1 || magicNumber == P4
    then return Nothing
    else Just <$> (skipWhitespace *> C8.decimal)
  whitespace
  return Header {..}
```

Разбирает десятичные значения ширины и высоты

Разбирает факультативное десятичное значение максимальной величины

Разобравшись с заголовком, перейдем к разбору данных изображения.

12.2.3 Выбор формата

В разных файловых форматах PNM данные кодируются по-разному. Вспомним эти форматы. Для сигнатур `P1` (bitmap), `P2` (graymap) и `P3` (pixmap) данные представлены в кодировке ASCII. В черно-белых изображениях (bitmap) есть всего два возможных значения, `1` и `0`, а в полутоновых (graymap) и цветных (pixmap) допустимое значение зависит от максимальной величины, указанной в заголовке. Пиксели представлены одним десятичным значением в полутоновых изображениях и тремя десятичными значениями в цветных. Для сигнатур `P4` (bitmap), `P5` (graymap) и `P6` (pixmap) действуют аналогичные правила, однако данные закодированы в двоичном виде. В двоичном черно-белом изображении каждый *бит* соответствует одному пикселю, причем пиксели следуют в том же порядке, что биты, т. е. файл нужно читать побайтно, разбирая биты от старшего к младшему. Если число подлежащих чтению пикселей не делится на 8, то лишние биты игнорируются. Для двоичных полутоновых и цветных изображений действуют те же правила, что для их аналогов в кодировке ASCII. Однако применяется специальное правило, зависящее от максимальной величины. По умолчанию все значения однобайтовые. Если же максимальная величина больше 255, то значение не помещается в один байт, и, следовательно, для получения значения каждого пикселя нужно прочитать два байта.

Начнем с написания парсеров для файлов в кодировке ASCII. Пока нас интересует только разбор одиночных пикселей, а затем мы воспользуемся им для разбора всех данных. Из предыдущего обсуждения мы знаем, что разбор данных пикселей черно-белого изображения сводится к разбору констант (`1` и `0`). При чтении полутоновых и цветных изображений нужно будет разбирать десятичные значения. Мы не будем сразу проверять, превышена максимальная величина, указанная в заголовке, или нет. Нас интересует только формальный разбор данных, присутствующих в файле. Код этих парсеров показан в листинге ниже.

Листинг 12.24 Парсеры данных изображения в кодировке ASCII

```
p1PixelP :: Parser RawPixel
p1PixelP =
  Single
    <$> choice
      [ 1 <$ C8.char '1',
        0 <$ C8.char '0'
      ]

p2PixelP :: Parser RawPixel
p2PixelP = Single <$> C8.decimal

p3PixelP :: Parser RawPixel
p3PixelP = do
  r <- C8.decimal
  whitespaces
  g <- C8.decimal
  whitespaces
  b <- C8.decimal
  return $ RGB r g b
```

Разбирает только литеры 1 и 0 и отображает их в соответствующие значения

Разбирает одно десятичное значение

Разбирает три десятичных значения как значение одного RGB-пикселя

Парсер `decimal` типа `Integral a => Parser a` умеет разбирать любые целые значения. Haskell способен вывести, что разбираемое значение в наших экземплярах имеет тип `Word16`, потому что именно этот тип задан в конструкторах. В реализациях `p1PixelP` и `p2PixelP` мы используем оператор `(<$>)` для применения конструктора одного значения к разбираемым данным. В случае `p3PixelP` мы используем `do`-нотацию для разбора произвольного числа пробельных символов между десятичными значениями, потому что в спецификации формата файла не сказано, что число пробелов между значениями фиксировано.

Покончив с пикселями в кодировке ASCII, мы можем заняться двоичными форматами. Сначала обсудим полутоновые и цветные изображения, потому что у черно-белых есть особенности. Для этих двух форматов нам нужно по два парсера для каждой сигнатуры, потому что для получения значения читается либо один байт (`Word8`), либо два байта (`Word16`). Attoparsec уже предоставляет парсер `anyWord8 :: Parser Word8`, который читает один байт. Но парсера `anyWord16` за нас никто не написал, а значит, это придется сделать самостоятельно.

Для написания этого парсера нам нужно подумать, как объединить несколько байтов в одно слово. Допустим, имеется два байта, записанных подряд. Мы хотим прочитать их как `Word16`. Мы можем прочитать сначала первый, а потом второй байт. Для объединения их придется интерпретировать как 16-разрядные слова, а затем сдвинуть первый байт на 8 позиций влево. Тогда этот байт займет первые 8 бит 16-разрядного слова, оставив столько же места для бай-

та, прочитанного вслед за ним. В предположении, что после сдвига значения освободившиеся позиции содержат `0`, мы можем вставить в них второй байт, выполнив операцию поразрядного ИЛИ.

Такие *низкоуровневые* операции предоставляет модуль `Data.Bits`. Там имеется класс типов `Bits`, который определяет поразрядные операции для разных типов данных, в т. ч. для `Word8` и `Word16`. Там же мы находим метод `shift :: a -> Int -> a` для сдвига значения влево и оператор `(.|.) :: a -> a -> a` для выполнения поразрядной операции ИЛИ. Поэкспериментируем:

```
ghci> import Data.Word
ghci> import Data.Bits
ghci> byte1 = 1 :: Word8
ghci> byte2 = 1 :: Word8
ghci> shift byte1 8
0
ghci> shift (fromIntegral byte1 :: Word16) 8
256
ghci> byte1_16 = fromIntegral byte1 :: Word16
ghci> byte2_16 = fromIntegral byte2 :: Word16
ghci> shift byte1_16 8 .|. byte2_16
257
```

Как видим, сдвиг `Word8`, содержащего один единичный бит (в младшем разряде), на 8 позиций влево дает `0`, т. к. мы вышли за границы байта. Но если преобразовать его в `Word16` с помощью функции `fromIntegral`, то место для сдвинутого бита появится. Также мы видим, что сдвиг одного байта на 8 позиций и последующее поразрядное ИЛИ со вторым байтом объединяет два байта в нужном нам порядке. Это показано на рис. 12.6.

Мы можем воспользоваться этим для построения парсера `anyWord16`, который будет читать два байта и объединять их, как описано выше. Реализация показана в следующем листинге.

Листинг 12.25 Парсеры 16-разрядных слов

```
import Data.Bits

...

anyWord16 :: Parser Word16
anyWord16 = do
  h <- fromIntegral <$> anyWord8
  l <- fromIntegral <$> anyWord8
  return $ shift h 8 .|. l
```

Читает 8-разрядное слово и преобразует его в 16-разрядное

Объединяет два значения в одно 16-разрядное слово

Мы снова пользуемся функцией `fromIntegral` для преобразования `Word8` в `Word16`, чтобы поразрядные операции могли работать. Разобравшись с этой функциональностью, мы можем построить парсеры

полутоновых и цветных изображений в 8- и 16-разрядном вариантах. Их реализация, показанная в листинге ниже, достаточно прямолинейна, потому что теперь не нужно думать о пробелах.

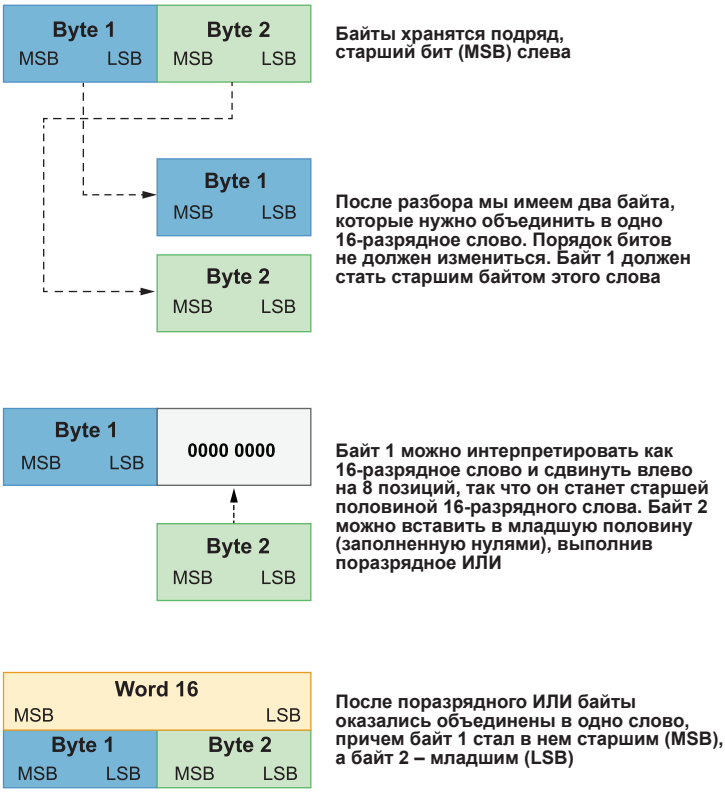


Рис. 12.6 Процесс объединения двух байтов в одно 16-разрядное слово

Листинг 12.26 Парсеры двоичных 8- и 16-разрядных PNM-файлов

```
p5PixelWord8P :: Parser RawPixel
p5PixelWord8P = Single <$> (fromIntegral <$> anyWord8)

p5PixelWord16P :: Parser RawPixel
p5PixelWord16P = Single <$> anyWord16

p6PixelWord8P :: Parser RawPixel
p6PixelWord8P = RGB <$> p <*> p <*> p
  where
    p = fromIntegral <$> anyWord8

p6PixelWord16P :: Parser RawPixel
p6PixelWord16P = RGB <$> anyWord16 <*>
➥ anyWord16 <*> anyWord16
```

Читает одно 8-разрядное слово

Читает одно 16-разрядное слово

Читает три 8-разрядных слова

Читает три 16-разрядных слова

Осталось только разобрать двоичные черно-белые изображения. В этом случае файлы нужно разбирать побитово, т. к. каждый пиксель занимает один бит. Мы не умеем читать файл побитово, поэтому должны подумать, как извлечь значения из прочитанных байтов. Снова нам поможет класс типов `Bits` из модуля `Data.Bits`, который предоставляет необходимую функцию, `testBit :: a -> Int -> Bool`, позволяющую опросить бит в указанной позиции байта. С помощью этой функции мы можем разобрать один байт и вернуть восемь закодированных в нем значений пикселей. Это показано в следующем листинге.

Листинг 12.27 Парсер двоичных черно-белых изображений

```
p4PixelP ::
  Parser
    ( RawPixel,
      RawPixel,
      RawPixel,
      RawPixel,
      RawPixel,
      RawPixel,
      RawPixel,
      RawPixel
    )
p4PixelP = do
  w <- anyWord8
  let parseBitAtIndex n =
        Single $ if testBit w n then 1 else 0

return
  ( parseBitAtIndex 7,
    parseBitAtIndex 6,
    parseBitAtIndex 5,
    parseBitAtIndex 4,
    parseBitAtIndex 3,
    parseBitAtIndex 2,
    parseBitAtIndex 1,
    parseBitAtIndex 0
  )
```

Читает одно 8-разрядное слово

Определяет функцию для выделения указанного бита из 8-разрядного слова

Возвращает все биты разобранного слова по отдельности

В этой реализации мы разбираем каждый бит по отдельности и возвращаем содержащий их кортеж, потому что точно знаем, сколько битов в значении типа `Word8`. Разбор производится от старшего бита к младшему, как указано в спецификации формата PNM. Таким образом, мы получаем доступ к каждому битовому значению. Однако при разборе всего файла нужно игнорировать лишние биты, не являющиеся частью изображения (если число пикселей в изображении не делится нацело на 8). Имея парсеры одиночных пикселей, мы можем теперь сконструировать парсер всего тела файла изображения.

12.2.4 Собираем все парсеры вместе

Парсер тела файла изображения можно построить, выбирая для чтения данных нужный парсер из числа описанных в предыдущем разделе в зависимости от уже разобранного заголовка, а именно содержащихся в нем сигнатуры и максимальной величины. Поскольку все парсеры, кроме парсера двоичного черно-белого изображения, читают одно значение типа `RawPixel`, мы можем просто повторить применение парсера столько раз, сколько пикселей имеется в файле (ширина, помноженная на высоту). Кроме того, мы построим парсер пробельных символов, который сможем использовать для данных в кодировке ASCII. Для черно-белых изображений пробельные символы не нужны, но мы все равно разрешим их, например для разбора знаков новой строки. Для двоичных полутоновых и цветных изображений нужный парсер выбирается в зависимости от максимальной величины. Единственный специальный случай – обработка двоичных черно-белых изображений, которые должны разбираться по-другому. Этот случай мы скоро рассмотрим. А пока обсудим реализацию парсера данных изображения, приведенную в следующем листинге.

Листинг 12.28 Парсер данных изображения в PNM-файле

```
dataP :: Header -> Parser RawData
dataP (Header {..}) = do
  pixels <- case (magicNumber, maxVal) of
    (P1, _) -> count' $ whitespaces' *> p1PixelP
    (P2, _) -> count' $ whitespaces' *> p2PixelP
    (P3, _) -> count' $ whitespaces' *> p3PixelP
    (P4, _) -> p4DataP width' height'
    (P5, Just mv) ->
      count' $
        if mv <= 255
          then p5PixelWord8P
          else p5PixelWord16P
    (P6, Just mv) ->
      count' $
        if mv <= 255
          then p6PixelWord8P
          else p6PixelWord16P
    _ ->
      error $
        "Internal error: "
          ++ show magicNumber
          ++ " without maximum value in header!"
```

Разбирает данные изображения в формате, определенном сигнатурой в заголовке, повторяя нужный парсер фиксированное число раз

Разбирает одиночные пиксели, учитывая разрядность

Возбуждает ошибку, если заголовок некорректен

Разбирает данные изображения в формате, определенном сигнатурой в заголовке, повторяя нужный парсер фиксированное число раз

```
  return $ RawData pixels
    where
      whitespaces' = option () whitespaces
      count' = count (width' * height')
      width' = fromInteger width
      height' = fromInteger height
```

Определяет парсер, который факультативно пропускает пробельные символы

Определяет комбинатор, который выполняет парсер столько раз, сколько пикселей должно быть в изображении

Функция `count'` применяет заданный парсер столько раз, сколько необходимо. Ветвление `case` по сигнатуре и максимальной величине дает нужное действие, которое вычисляется для чтения значений `pixels`. Наш тип допускает отсутствие максимальной величины для сигнатур `P5` и `P6`. Однако это не разрешено и должно приводить к внутренней ошибке. Если такое происходит, то парсер заголовка должен был выдать ошибку. Так что если мы столкнулись с этой проблемой, значит, парсер заголовка написан неправильно.

Как видим, все наши парсеры используются в сочетании с `count'`, за исключением парсера двоичных черно-белых изображений. Это объясняется тем, что мы получаем от парсера не один `RawPixel`, а сразу восемь. Мы должны построить алгоритм, который выделяет значения из байта, но останавливается, как только разобрано достаточно значений. Кроме того, двоичные черно-белые изображения содержат растр значений. Число битов в каждой строке равно ширине изображения. Однако эти строки упакованы в байты! Это значит, что каждая строка может содержать лишние биты, которые, согласно спецификации, следует игнорировать. Для этого мы можем определить рекурсивное действие, которое разбирает каждую строку отдельно, рекурсивно читая нужное число битов и игнорируя остальные. Затем мы сможем конкатенировать строки в один большой список значений типа `RawPixel`. Эта реализация показана ниже.

Листинг 12.29 Парсер данных двоичного черно-белого изображения

```
p4DataP :: Int -> Int -> Parser [RawPixel]
p4DataP width height = do
  rows <- count height (reverse <$> readRow width [])
  return $ concat rows
  where
    readRow n pxs
    | n <= 0 = return pxs
    | otherwise = do
      (b7, b6, b5, b4, b3, b2, b1, b0) <- p4PixelP
      case n of
```

Читает столько строк, какова высота изображения

Возвращает прочитанные пиксели по окончании чтения

Разбирает байт и выделяет значения пикселей

```
1 -> return $ b7 : pxs
2 -> return $ b6 : b7 : pxs
3 -> return $ b5 : b6 : b7 : pxs
4 -> return $ b4 : b5 : b6 : b7 : pxs
5 -> return $ b3 : b4 : b5 : b6 : b7 : pxs
6 -> return $ b2 : b3 : b4 : b5 : b6 : b7 : pxs
7 -> return $ b1 : b2 : b3 : b4 : b5 : b6 : b7 : pxs
8 -> return $ b0 : b1 : b2 : b3 : b4 : b5 : b6 : b7 : pxs
_ ->
  readRow
    (n - 8)
    (b0 : b1 : b2 : b3 : b4 : b5 : b6 : b7 : pxs)
```

Рекурсивно разбирает дополнительные байты, пока не будет прочитано столько пикселей, сколько должно быть в строке

Возвращает правильное число битов, если от парсера осталось получить меньше одного байта

Здесь нужно внимательно относиться к порядку битов. Мы разбираем биты от старшего к младшему, но в результирующий список для каждой строки добавляем в обратном порядке. Затем эти списки инвертируются. В `readRow` мы следим за тем, сколько элементов нужно прочитать, и, когда останется меньше девяти, включаем в результат ровно столько, сколько необходимо.

> **ПРИМЕЧАНИЕ** Значения добавляются в обратном порядке ради производительности. В противном случае мы не могли бы дописывать элемент в начало с помощью конструктора `(:)`, а должны были бы использовать функцию добавления в конец `(++)`, которая работает дольше, потому что должна обойти весь список перед добавлением новых элементов. Для больших списков ее рекурсивное выполнение быстро становится запретительно дорогим. Дешевле добавлять элементы в обратном порядке, а затем инвертировать весь список.

Прочитав все строки, мы можем конкатенировать значения и получить полные данные типа `[RawPixel]`. Теперь, когда реализован разбор для всех форматов, можем собрать все кусочки вместе.

Функции `headerP` и `dataP` дают парсеры заголовка и тела файла изображения. Объединив их, получим парсер PNM-файлов общего вида. Он показан в следующем листинге.

Листинг 12.30 Парсер и функция для разбора PNM-файлов

```
pnmP :: Parser RawPnm
pnmP = do
  header <- headerP
  imageData <- dataP header
  option () whitespaces
  return RawPnm {..}
```

Разбирает заголовок

Разбирает данные изображения на основе заголовка

Пропускает хвостовые пробелы

```
parsePnm :: BS.ByteString -> Either String RawPnm
parsePnm = parseOnly (pnmP <* endOfInput)

readPnmFile :: FilePath -> IO (Either String RawPnm)
readPnmFile path = parsePnm <$> BS.readFile path
```

Разбирает изображение в формате PNM, хранящееся в строке байтов, ожидая, что парсер дойдет до конца строки, в противном случае завершается неудачно

Разбирает изображение в формате PNM, ожидая, что парсер дойдет до конца файла, в противном случае завершается неудачно

Прочитав заголовок и данные, мы дополнительно пропускаем пробельные символы, которые теоретически могли присутствовать в файле. Делается это для того, чтобы мы были уверены, что дошли до конца файла. В функции `parsePnm` мы явно говорим, что разбор должен завершаться неудачно, если вход прочитан не полностью; это делает функция `endOfInput`. Если после применения `pnmP` парсер остановился не в конце файла, то `endOfInput` аварийно завершит программу.

Упражнение. Разбор нескольких изображений

Показанный в этой главе парсер умеет читать только одно изображение из PNM-файла. Однако спецификация допускает наличие нескольких изображений в одном файле. Сможете ли вы модифицировать наши функции разбора, так чтобы реализовать это требование?

Если вы хотите испытать наш новый парсер в деле, в репозитории кода к этой книге имеются примеры изображений разного формата и размера. Мы написали парсер формата PNM, который умеет читать данные изображений из файлов. В следующей главе займемся параллельной обработкой этих данных, чтобы повысить скорость работы.

Резюме

- Функторы используются для ссылочно-прозрачной модификации значения, являющегося результатом вычисления эффекта.
- Аппликативные функторы можно использовать для композиции эффектообразующих действий, например обработки входных данных или разбора содержимого файлов.
- Класс типов `Alternative` наделяет аппликативный функтор способностью выбирать между успешным и неудачным действиями, например он позволяет парсеру вернуть сообщение об ошибке, возникшее глубоко внутри рекурсивно устроенного разбора.

- Монады можно использовать для композиции эффектообразующих действий таким образом, что результаты предыдущих действий влияют на новый результат. При этом действия сцепляются, но возможность единообразной обработки ошибок сохраняется.
- Класс `MonadFail` наделяет монадический тип возможностью отказа, это используется, чтобы явно перевести монаду в ошибочное состояние.
- Класс `MonadPlus` – монадический эквивалент `Alternative`, который наделяет монады возможностью выбирать между успешным и неудачным действиями.
- Для выполнения поразрядных операций служат функции из модуля `Data.Bits`. Они применяются для манипулирования числами и строками байтов на низком уровне.

13 Параллельная обработка изображений

Краткое содержание этой главы:

- работа с обобщенными алгебраическими типами данных;
- как безопасно избежать полиморфизма по возвращаемому значению;
- реализация обобщенного алгоритма преобразования данных изображения;
- повышение производительности за счет распараллеливания.

В предыдущей главе мы написали парсер простого формата изображений PNM. Мы рассмотрели чтение таких файлов, но пока не умеем их преобразовывать и записывать обратно в файловую систему. В этой главе мы наверстаем упущенное.

Мы расскажем о том, как систему типов Haskell можно использовать для создания типа данных, способного динамически сохранять данные разных типов, не прибегая к параметрическому полиморфизму, но тем не менее сохраняя возможность извлечь информацию о типе этих данных впоследствии. Затем мы определим способ проверки разобранных данных, а также общий способ отображения данных разных типов. Наконец, мы поговорим о том, как распарал-

лелить работу и тем самым более чем вдвое повысить производительность нашей программы ценой всего одной строчки кода.

13.1 Предоставление информации о типе вызывающей стороне

Предыдущая глава была посвящена разбору изображений в формате PNM. Теперь, когда изображения загружены в программу, мы могли бы что-то сделать с ними. Давайте их обработаем! Точнее, мы хотим придумать общий способ изменять пиксели изображения из программы. Это позволит нам писать различные фильтры, например:

- преобразование изображения в полутоновое;
- пикселизация;
- размытие.

Для ускорения работы мы будем использовать распараллеливание на многоядерных процессорах. В конечном итоге мы хотим сохранить изображение (обработанное или нет) в формате PNG на диске, для чего воспользуемся библиотекой.

13.1.1 Проблемы полиморфизма по возвращаемому значению

Прежде всего вспомним, на чем мы остановились. Мы написали парсер для чтения файлов в формате PNM, который читает данные и порождает представление в Haskell с помощью типа `RawPnm`. Введенный нами тип показан в листинге ниже.

Листинг 13.1 Типы для разбора PNM-файлов

```
data MagicNumber = P1 | P2 | P3 | P4 | P5 | P6
  deriving (Eq, Show)

data Header = Header
  { magicNumber :: MagicNumber,
    width :: Integer,
    height :: Integer,
    maxVal :: Maybe Word16
  }
  deriving (Eq, Show)

data RawPixel
  = Single Word16
  | RGB Word16 Word16 Word16
  deriving (Eq, Show)

newtype RawData = RawData
```

Тип сигнатур

Тип заголовка PNM-файла

Определяет пиксель, состоящий из одного значения (полутоновый или черно-белый)

Определяет пиксель, состоящий из трех значений для красного, зеленого и синего каналов

Тип для хранения исходных пикселей

```
  { pixels :: [RawPixel]
  }
  deriving (Eq, Show)

data RawPnm = RawPnm    <-- Тип разобранного PNM-файла
  { header :: Header,
    imageData :: RawData
  }
  deriving (Eq, Show)
```

Эти типы дают представление исходных данных, прочитанных из файла. Значения поступают непосредственно от парсера, т. е. никакой проверки не производилось. И это первое, о чем мы должны позаботиться. В наследство от предыдущей главы нам осталась структура модуля для разбора формата PNM, показанная в листинге ниже.

Листинг 13.2 Структура модуля, образовавшаяся в главе 12

```
src/
└── Graphics
├── PNM
│   ├── Parsing.hs
│   └── Types.hs
└── PNM.hs
```

Теперь мы хотим создать модуль `Graphics.PNM.Validation`, который будет предоставлять функции, занимающиеся проверкой. Напомним, что есть три разновидности формата PNM: черно-белый, полутоновый и цветной, и каждый из них допускает представление в кодировке ASCII или в двоичной форме. Конкретный вариант определяется сигнатурой в заголовке файла. В соответствии с этим форматом пиксели кодируются либо одним числом (в черно-белом и полутоновом изображении), либо тремя числами (в цветном изображении). Кроме того, в заголовке файла может быть задана максимальная величина цвета.

Написанный в предыдущей главе парсер не проверяет разобранные данные. Процедура проверки должна включать следующие шаги:

- число прочитанных пикселей равно произведению ширины и высоты, заданных в заголовке;
- типы прочитанных пикселей соответствуют сигнатуре в заголовке;
- цвета пикселей не превосходят максимальной величины, заданной в заголовке.

Этот процесс должен вернуть некоторый тип, с которым можно безопасно работать. Как должен выглядеть такой тип? И прежде всего как должен выглядеть тип наших пикселей? Поскольку мы знаем,

что PNM-файлы кодируются 8 или 16 битами, необходимо поддержать это различие. В нашем парсере 8-разрядные значения кодируются типом `Word16`, что несколько расточительно, поэтому не будем продолжать эту практику. Типы пикселей могли бы выглядеть так:

```
import Data.Word (Word8, Word16)

data Pixel8Bit = Pixel8Bit Word8 Word8 Word8

data Pixel16Bit = Pixel16Bit Word16 Word16 Word16
```

Однако изобретать колесо нет необходимости. Эти типы уже предоставляет пакет JuicyPixels, которым мы позже воспользуемся для преобразования данных из формата PNM в формат PNG! Поэтому имеет смысл включить типы из этого пакета в наш тип изображения.

Для этого добавим зависимость `JuicyPixels` в раздел `dependencies` файла `package.yaml`. Теперь мы можем использовать типы `PixelRGB8` и `PixelRGB16` из модуля `Codec.Picture`. Дополнительно модуль предоставляет типы `Pixel8` и `Pixel16` для полутоновых пикселей, они нам также пригодятся:

```
ghci> :t PixelRGB8
PixelRGB8 :: Pixel8 -> Pixel8 -> Pixel8 -> PixelRGB8
ghci> :t PixelRGB16
PixelRGB16 :: Pixel16 -> Pixel16 -> Pixel16 -> PixelRGB16
ghci> :i Pixel8
type Pixel8 :: *
type Pixel8 = Word8
ghci> :i Pixel16
type Pixel16 :: *
type Pixel16 = Word16
```

Как мы позднее будем использовать этот пакет для записи данных в формате PNG? Для этого библиотека предоставляет тип `DynamicImage`, который служит для внутреннего представления сохраняемого изображения в терминах глубины цвета, цвета и модели – полутоновой или цветной. Чтобы правильно использовать этот тип, мы должны запоминать тип используемых пикселей.

Теперь можно задать вопрос о том, как построить тип изображения. Простой тип, содержащий основную информацию об изображении, мог бы выглядеть так:

```
import qualified Codec.Picture as P

data PnmImage = PnmImage
  { width :: Int,
    height :: Int,
    pixels :: Either [P.PixelRGB8] [P.PixelRGB16]
  }
```

Однако если мы также хотим сохранять полутоновые пиксели, то понадобится тип, аналогичный `Either`, который дает нам четыре варианта. Глядя в этом направлении, мы могли бы остановиться на следующем типе:

```
data ArbitraryPixels
  = Pixels8Bit [P.PixelRGB8]
  | Pixels16Bit [P.PixelRGB16]
  | GrayPixels8Bit [P.Pixel8]
  | GrayPixels16Bit [P.Pixel16]

data PnmImage = PnmImage
  { width :: Int,
    height :: Int,
    pixels :: ArbitraryPixels
  }
```

На первый взгляд, решение неплохое, но практически оно бесполезно. Прежде всего невозможно гарантировать, что некоторая функция принимает изображения только определенного пиксельного типа, потому что у типа `Image` нет параметра. Работа с этим типом неизбежно приведет нас к необходимости сопоставления с конструктором `pixels` для выяснения внутреннего типа. Кроме того, у формата PNM имеется расширение, допускающее использование 32-разрядных пикселей. Если бы мы захотели поддержать его, то должны были бы еще расширить этот тип и все функции, которые производят сопоставление с ним. Так что это решение непрактично.

Мы можем решить проблему, добавив к типу `Image` параметр. Тогда можно будет указать, с изображением какого типа мы имеем дело, не выходя за рамки системы типов:

```
data PnmImage px = PnmImage
  { width :: Int,
    height :: Int,
    pixels :: [px]
  }
```

Однако возникает новая проблема. Как должна выглядеть функция проверки, которую мы собираемся написать? Каким должен быть ее тип? В голову приходит что-то вроде:

```
validatePnm :: RawPnm -> Either String (PnmImage px)
```

Выглядит неплохо, но только до тех пор, пока мы не задумаемся о том, где принимается решение о типе переменной `px`. Мы не можем принять это решение внутри `validatePnm`, потому что в Haskell применяется полиморфизм по типу возвращаемого значения, о чем

мы говорили в главе 9. Это значит, что сторона, *вызывающая* функцию, принимает решение о типе возвращаемого ей значения! Однако это полностью расходится с нашими целями. Вызывающая сторона не может еще до проверки знать ожидаемый тип изображения. Функция `validatePnm` должна принять решение о виде изображения, проверить его корректность, а затем произнести свой вердикт. Как же передать ответственность за выбор типа этой функции?

13.1.2 Обобщенные алгебраические типы данных

Проблема, стоящая перед нами, очень проста. Вызывающая сторона никаким способом не может принять решение о типе функции проверки до ее вызова, поэтому ее тип должен быть полиморфным. Однако когда мы захотим поработать с результатом разбора, нужно будет этот тип *уточнить*. Это значит, что нам необходим способ взять полиморфный тип и конкретизировать его. Такая функциональность предоставляется расширением системы типов Haskell, которое называется *обобщенными алгебраическими типами данных* (generalized algebraic data types – GADT). Оно активируется с помощью расширения языка `GADTs`.

GADT во многом эквивалентны обычным алгебраическим типам данных, с которыми мы все время работаем. Они состоят из конструкторов, которые состоят из полей. Однако есть и различия, и первое из них – синтаксис использования GADT:

```
{-# LANGUAGE GADTs #-}

...

data MyAdt a
  = AdtCons1 Int Float String
  | AdtCons2 Bool a

data MyGADT a where
  GadtCons1 :: Int -> Float -> String -> MyGADT a
  GadtCons2 :: Bool -> a -> MyGADT a
```

Эти два типа эквивалентны. `MyAdt` и `MyGadt` имеют эквивалентные конструкторы с эквивалентными полями и (что самое важное) эквивалентными типами. Как видим, оба конструктора нашего GADT имеют тип `MyGADT a`. В точности так это работает для наших ADT, которые всегда имеют тип `MyAdt a`.

Однако GADT могут иметь тип, зависящий от используемого конструктора! Немного изменим определение:

```
data MyGADT a where
  GadtCons1 :: Int -> Float -> String -> MyGADT Bool
  GadtCons2 :: Bool -> a -> MyGADT String
```

Теперь конструктор решает, каким должен быть полиморфный тип (a).

```
ghci> :t GadtCons1 1 1.0 "Hi!"
GadtCons1 1 1.0 "Hi!" :: MyGADT Bool
ghci> :t GadtCons2 True (1 :: Int)
GadtCons2 True (1 :: Int) :: MyGADT [Char]
```

Чем это может помочь? Это дает возможность определять функции, которые полиморфны, но ограничены типом GADT:

```
data GiveMe a where
  SomeString :: GiveMe String
  SomeInt :: GiveMe Int

giveMe :: GiveMe a -> a
giveMe SomeString = "Hello"
giveMe SomeInt = 1
```

Это определение `giveMe` вполне допустимо, хотя и возвращает *разные типы* в зависимости от сопоставленного конструктора. Это возможно благодаря типу `GiveMe`. Поскольку `SomeString` имеет тип `GiveMe String`, это означает, что `giveMe SomeString` должно иметь тип `String`, т. к. вместо `a` в `GiveMe a` подставляется `String` из `GiveMe String`. Haskell понимает, как статически вывести нужный тип на основе сопоставленного конструктора. Потому-то определение проходит проверку типа и работает, как ожидается:

```
ghci> giveMe SomeString
"Hello"
ghci> giveMe SomeInt
1
```

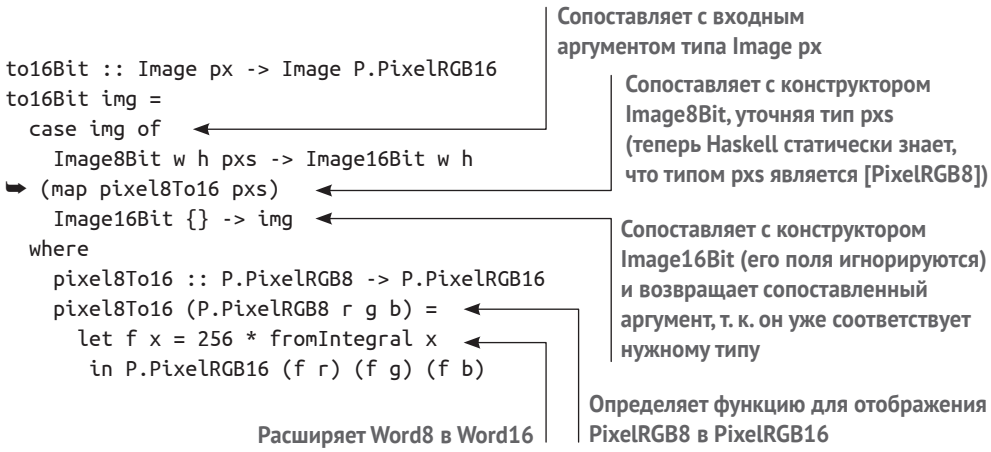
GADT – важное дополнение к классам типов. Если классы типов *отображают типы в термы*, то GADT *отображают термы в типы*. Классы типов позволяют писать полиморфный код и при этом использовать конкретную реализацию. GADT умеют *уточнять полиморфный тип* при сопоставлении с их конструкторами. Именно поэтому наше определение `giveMe` работает, как мы хотели.

Мы можем воспользоваться этим в своем коде, чтобы дать стороне, вызывающей функцию проверки, возможность определить, какой из пиксельных типов используется. Предположим, что тип изображения используется следующим образом:

```
data Image px where
  Image8Bit ::
    Int ->
    Int ->
    [P.PixelRGB8] ->
```

```
    Image P.PixelRGB8
  Image16Bit ::
    Int ->
    Int ->
    [P.PixelRGB16] ->
    Image P.PixelRGB16
```

Теперь можно написать функции для значений этого типа, которые сопоставляются с конструкторами. Отличие от ADT в том, что при сопоставлении с GADT уточняется тип внутри сопоставленной ветви. Рассмотрим пример:



Как видно из этого примера, аргумент (типа `Image px`) полиморфный. Однако наша функция может привести этот полиморфный тип к `PixelRBG16`, не жертвуя безопасностью типов. Haskell умеет статически определять, какие конструкторы существуют, какие типы из них вытекают и правильно ли мы используем эти типы.

13.1.3 Tun Vector

Теперь можно задуматься о проектировании типа для наших изображений. Он должен хранить ширину и высоту изображения, а также его данные, представленные некоторым типом значений пикселей. В предыдущей главе мы использовали для хранения пикселей простой список. Списки хороши для небольших объемов данных, но обладают плохими характеристиками производительности, когда требуется доступ к произвольному элементу. Поскольку впоследствии это будет важно, нужно подумать о какой-нибудь другой структуре данных.

Нам подходит тип данных `Vector` из пакета `vector`. Этот тип похож на массивы, имеющиеся в других языках, в том смысле, что для доступа к значениям по индексу требуется постоянное время. Кро-

ме того, такие типичные операции, как `take`, `drop` и `splitAt`, выполняются за постоянное время. Однако операции добавления новых элементов или конкатенации нескольких векторов в типе `Vector` выполняются медленнее. И еще этот тип похож на обычные массивы в других языках тем, что в процессе динамического роста массивов данные приходится снова и снова копировать в памяти.

В нашей библиотеке мы хотим хранить данные изображения и затем эффективно строить над ними отображения. Для этого нам нужно уметь читать одиночные пиксели и комбинировать их. Поэтому большая часть работы будет связана с *чтением*, а не *записью* данных, что делает тип `Vector` подходящим кандидатом.

Для добавления пакета нужно включить `vector` в секцию `dependencies` файла `package.yaml`. Значения типа `Vector` можно создавать функциями `cons` и `snoc`, которые добавляют элементы в начало или в конец соответственно, или просто путем преобразования списка функцией `fromList`:

```
ghci> import Data.Vector as V
ghci> :t V.empty
V.empty :: Vector a
ghci> :t V.cons
V.cons :: a -> Vector a -> Vector a
ghci> V.cons 1 V.empty :: V.Vector Int
[1]
ghci> V.fromList [1..10] :: V.Vector Int
[1,2,3,4,5,6,7,8,9,10]
```

Очень полезно сначала лениво построить список, а затем преобразовать его в вектор. Так мы сможем воспользоваться многочисленными функциями, которые строят списки с помощью более простых функций (вспомните о комбинаторах парсеров из предыдущей главы), а затем построить более компактное представление, которое гораздо проще индексировать и читать.

> **ПРИМЕЧАНИЕ** Векторы не всегда являются структурно ленивыми. Это значит, что обычно мы не можем создать бесконечное значение типа `Vector`, как в случае списков. Следующий код будет «пожирать» память, пока программа не «грохнется»: `V.take 100 $ V.concat [V.empty, (V.fromList [1..])]`.

Теперь мы можем добавить новый тип в наш модуль `Graphics.PNM.Types` из предыдущей главы. Этот тип будет предоставлять четыре конструктора для кодирования полутоновых и цветных изображений в 8- и 16-разрядных вариантах. Каждый конструктор хранит информацию о ширине, высоте и пикселях. Последние кодируются типом `Vector`.

Листинг 13.3 Тип для хранения PNM-изображений

Активирует расширение языка GADTs, чтобы можно было использовать в коде обобщенные алгебраические типы данных

```
{-# LANGUAGE GADTs #-}

module Graphics.PNM.Types where

import qualified Codec.Picture as P
import qualified Data.Vector as V

...

data PnmImage px where
  PnmGray8Bit ::
    Int ->
    Int ->
    V.Vector P.Pixel8 ->
    PnmImage P.Pixel8
  PnmGray16Bit ::
    Int ->
    Int ->
    V.Vector P.Pixel16 ->
    PnmImage P.Pixel16
  PnmColor8Bit ::
    Int ->
    Int ->
    V.Vector P.PixelRGB8 ->
    PnmImage P.PixelRGB8
  PnmColor16Bit ::
    Int ->
    Int ->
    V.Vector P.PixelRGB16 ->
    PnmImage P.PixelRGB16
```

Импортирует необходимые модули

Определяет тип данных для кодирования проверенных PNM-изображений

Определяет конструкторы для различных видов PNM-изображений

Нельзя ли было использовать в этом определении синтаксис записи? Теоретически это можно делать в определениях GADT, но в этом контексте так не выйдет, потому что третье поле (пиксели) имеет разные типы в каждом конструкторе и не сопоставляется с полиморфным типом px. Однако нам необходимо написать несколько функций для доступа к ширине, высоте и пиксельным данным. Для наших пикселей эти функции работают, потому что полиморфны по переменной типа px. Они показаны в листинге ниже.

Листинг 13.4 Функции для доступа к полям PNM-изображений

```
pnmWidth :: PnmImage px -> Int
pnmWidth (PnmGray8Bit w _ _) = w
pnmWidth (PnmGray16Bit w _ _) = w
pnmWidth (PnmColor8Bit w _ _) = w
```

Функция выделения ширины PnmImage

```
pnmWidth (PnmColor16Bit w _ _) = w

pnmHeight :: PnmImage px -> Int
pnmHeight (PnmGray8Bit _ h _) = h
pnmHeight (PnmGray16Bit _ h _) = h
pnmHeight (PnmColor8Bit _ h _) = h
pnmHeight (PnmColor16Bit _ h _) = h

pnmPixels :: PnmImage px -> V.Vector px
pnmPixels (PnmGray8Bit _ _ pixels) = pixels
pnmPixels (PnmGray16Bit _ _ pixels) = pixels
pnmPixels (PnmColor8Bit _ _ pixels) = pixels
pnmPixels (PnmColor16Bit _ _ pixels) = pixels
```

Функция выделения высоты PnmImage

Функция выделения пикселей PnmImage

Вам, наверное, интересно, почему мы не вывели Show или Eq для нашего типа. Потому что с GADT нельзя использовать ключевое слово deriving. Если мы все-таки хотим выводить для них экземпляры классов типов, то должны использовать расширение StandaloneDeriving и выписывать отдельные выражения, как показано в листинге ниже.

Листинг 13.5 Вывод экземпляров классов типов для GADT

```
{-# LANGUAGE StandaloneDeriving #-}
...

deriving instance (Show px) => Show (PnmImage px)

deriving instance (Eq px) => Eq (PnmImage px)
```

Активирует расширения языка StandaloneDeriving

Выводит экземпляр Show для PnmImage px, при условии что экземпляр Show px существует

Выводит экземпляр Eq для PnmImage, px при условии что экземпляр Eq px существует

Имея код в листинге 13.5, мы можем считать, что с селекторами полей и экземплярами разобрались, и пока что остановиться на текущем определении типа PnmImage. И что, теперь мы можем, наконец, написать функцию проверки?

13.1.4 Динамические типы с навешенными кванторами существования

Увы, пока мы еще не готовы к написанию функции проверки. Идея включения GADT в том и заключалась, чтобы функция проверки могла принимать решение о типе пикселей. Однако полиморфизм по типу возвращаемого значения все еще стесняет нас, когда имеется функция, возвращающая PnmImage px. Использование GADT было только первой частью решения головоломки, поскольку оно дает вызывающей стороне возможность уточнить полиморфный тип.

Но теперь нам нужно придумать, как запретить вызывающей стороне выбирать этот тип. Требуется как-то скрыть параметр типа от возвращаемого функцией значения. Изящный выход, который мы уже обсуждали в главе 11, дает *еще одно* расширение системы типов Haskell: `ExistentialQuantification`.

Напомним, что с помощью этого расширения мы можем создавать типы данных, которые скрывают свои переменные типа от собственного типа. Это позволяет создать тип, который может хранить полиморфные типы, не будучи сам полиморфным. Единственное, о чем нужно позаботиться, – возможность снова извлечь информацию из этого типа. Обычно это решается с помощью ограничений типов, но в данной ситуации нам повезло чуть больше.

Листинг 13.6 Определение динамического типа данных для PNM-изображений

```
{-# LANGUAGE ExistentialQuantification #-}

...

data DynamicPnmImage = forall px.
➥ DynamicPnmImage (PnmImage px)
```

Активирует расширение языка ExistentialQuantification

Определяет тип, который может хранить значения PnmImage px для любого типа px

В листинге 13.6 мы видим, что ограничение типа в общем-то и не нужно. Почему? Не окажется ли наше значение в итоге бесполезным? Вовсе нет, поскольку мы все же знаем, что внутри конструктора `DynamicPnmImage` находится GADT `PnmImage`, а, как мы уже говорили, сопоставление с конструктором GADT уточняет его тип. Это значит, что после сопоставления со значением, находящимся внутри конструктора, мы будем знать, с какого вида изображением имеем дело. Мы не потеряли информацию о типах! Теперь мы наконец-то можем написать функцию проверки, которая принимает любой результат разбора и создает значение типа `DynamicPnmImage`, которое мы впоследствии сможем использовать, чтобы узнать, какого типа изображение хранится внутри.

13.2 Проверка разобранных данных

Для начала вспомним, что должна делать наша функция проверки. Она должна проверять следующие утверждения:

- число прочитанных пикселей равно произведению ширины и высоты, заданных в заголовке;

- типы прочитанных пикселей соответствуют сигнатуре в заголовке;
- цвета пикселей не превосходят максимальной величины, заданной в заголовке.

Написанный в предыдущей главе парсер не производит никаких проверок данных в типе `RawPnm`. Мы можем написать функцию, которая преобразует значения типа `RawPnm` в тип `DynamicPnmImage`, проверяя при этом выполнение сформулированных выше утверждений. Эту функцию мы хотим поместить в новый модуль `Graphics.PNM.Validation`, который впоследствии реэкспортируем из модуля `Graphics.PNM`.

Для начала подумаем, как преобразовать (и проверить) одиночные пиксели. Для черно-белых пикселей парсер должен возвращать значения `RawPixel` с помощью конструктора `Single`, при этом цвет будет иметь значение `0` или `1`. Эти пиксели мы отобразим на тип `Pixel8` из пакета JuicyPixels, предназначенный для 8-разрядных полутоновых пиксельных значений. Этот тип – просто псевдоним `Word8`, поэтому мы можем вернуть обычное число. Напомним, что `1` в формате PNM представляет черный цвет. Функция проверки одного черно-белого пикселя показана в листинге ниже.

Листинг 13.7 Функция преобразования исходного пикселя в черно-белый пиксель

```
module Graphics.PNM.Validation where

import Graphics.PNM.Types
import qualified Codec.Picture as P

bitmapPixel :: RawPixel -> Either String P.Pixel8
bitmapPixel (Single 1) = Right 0
bitmapPixel (Single 0) = Right 255
bitmapPixel p =
  Left $
    "Could not convert "
      <> show p
      <> " to a bitmap pixel"
```

Возвращает черный или белый пиксель в зависимости от заданного RawPixel

Возвращает сообщение об ошибке, если RawPixel не представляет черно-белый пиксель

Как и в предшествующих главах, мы используем тип `Either` для кодирования ошибок проверки. Заметим, что мы не проверяем, превосходит ли значение пикселя максимальную величину, потому что для черно-белых изображений максимальная величина не задается. Это свойство мы будем проверять позже.

Максимальная величина важна при работе с полутоновыми пикселями. В зависимости от максимальной величины вернуть следует `Pixel8` или `Pixel16`, но это решение принимается вне функции. Поскольку `Pixel16` – псевдоним `Word16`, мы можем вернуть любой тип. Проверив, что значение меньше или равно максимальной величине,

мы можем масштабировать значение пикселя, умножив его на отношение заданного значения к максимально возможному. Это показано в листинге ниже.

Листинг 13.8 Функция преобразования исходного пикселя в полутоновый пиксель

```
import Data.Word

...

graymapPixel ::
  (Integral px) =>
  Word16 ->
  RawPixel ->
  Either String px
graymapPixel maxVal pixel@(Single g)
  | g > maxVal =
    Left $
      "The pixel "
        <> show pixel
        <> " exceeds the maximum value "
        <> show maxVal
  | otherwise = Right v
  where
    factor = fromIntegral g / fromIntegral maxVal
    v = round $ fromIntegral maxVal * factor
graymapPixel _ p =
  Left $
    "Could not convert "
      <> show p
      <> " to a graymap pixel"
```

Проверяет, превосходит ли значение данного пикселя максимальную величину, и возвращает сообщение об ошибке

Возвращает масштабированное значение

Вычисляет отношение заданного значения к максимальной величине

Умножает максимальную величину на отношение заданного значения к максимальной величине

Возвращает сообщение об ошибке, если RawPixel не является представлением полутонового пикселя

Чтобы воспользоваться функцией `round`, мы должны добавить в наш код класс `Integral` в качестве ограничения типа. Как будет производиться округление, определяется типом возвращаемого значения, который выбирает вызывающая сторона.

Для цветных пикселей мы делаем по существу то же самое, только преобразование из типа `RawPixel` осуществляется для значений типа `RGB`. Тип результата также сильно отличается от предыдущего, потому что теперь мы должны вернуть значение типа `PixelRGB8` или `PixelRGB16`. Оба конструктора получают три значения (типа `Pixel8` или `Pixel16`):

```
ghci> import Codec.Picture
ghci> :t PixelRGB8
PixelRGB8 :: Pixel8 -> Pixel8 -> Pixel8 -> PixelRGB8
ghci> :t PixelRGB16
PixelRGB16 :: Pixel16 -> Pixel16 -> Pixel16 -> PixelRGB16
```

Чтобы сделать нашу функцию проверки максимально общей, мы можем потребовать, чтобы аргумент сопоставлялся с полиморфной версией этих конструкторов. Тогда вызывающая сторона сможет передать функции подходящий конструктор, указав тем самым, каким должен быть тип результата. Эта функция показана в листинге ниже.

Листинг 13.9 Функция преобразования исходного пикселя в цветной пиксель

```
pixmapPixel ::
  (Integral val) =>
  Word16 ->
  (val -> val -> val -> px) ->
  RawPixel ->
  Either String px
pixmapPixel maxVal f pixel@(RGB r g b)
  | r > maxVal || g > maxVal || b > maxVal =
    Left $
      "The pixel "
        <> show pixel
        <> " exceeds the maximum value "
        <> show maxVal
  | otherwise =
    Right $
      f (transform r) (transform g) (transform b)
  where
    factor x = fromIntegral x / fromIntegral maxVal
    transform x = round $ fromIntegral maxVal * factor x
pixmapPixel _ _ p =
  Left $
    "Could not convert "
      <> show p
      <> " to a pixmap pixel"
```

Проверяет, превосходят ли заданные значения красного, зеленого и синего каналов максимальную величину, и возвращает сообщение об ошибке

Возвращает масштабированные значения красного, зеленого и синего каналов и обертывает их функцией, переданной в качестве аргумента, для порождения окончательного результата

Умножает максимальную величину на отношение заданного значения к максимальной величине

Возвращает сообщение об ошибке, если RawPixel не является представлением цветного пикселя

Вычисляет отношение заданного значения к максимальной величине

Для построения функции проверки было бы полезно вывести вид изображения из сигнатуры. Мы можем написать для этого несколько простых вспомогательных функций (показаны ниже).

Листинг 13.10 Вспомогательные функции для проверки

```
indicatesBitmap :: MagicNumber -> Bool
indicatesBitmap P1 = True
indicatesBitmap P4 = True
indicatesBitmap _ = False

indicatesGraymap :: MagicNumber -> Bool
indicatesGraymap P2 = True
indicatesGraymap P5 = True
indicatesGraymap _ = False
```

Функция, проверяющая, что заданная сигнатура определяет черно-белое изображение

Функция, проверяющая, что заданная сигнатура определяет полутоновое изображение

```
indicatesPixmap :: MagicNumber -> Bool
indicatesPixmap P3 = True
indicatesPixmap P6 = True
indicatesPixmap _ = False
```

Функция, проверяющая, что заданная сигнатура определяет цветное изображение

```
imageSizeCorrect :: RawPnm -> Bool
imageSizeCorrect RawPnm {header, imageData} =
  let expectedNumPixels =
        fromIntegral $ width header * height header
      numPixels = length $ pixels imageData
 in numPixels == expectedNumPixels
```

Функция, проверяющая, что размеры изображения, указанные в заголовке, соответствуют числу пикселей

Для удобства мы добавили также функцию, которая проверяет, действительно ли число пикселей в изображении равно размеру, указанному в заголовке. Заметим, что это определение допустимо, только если активировано расширение `NamedFieldPuns`. Давайте подумаем, как проверить, что все пиксели в нашем изображении корректны. Мы уже написали функции для проверки одного пикселя, но нужно проверить все пиксели исходного изображения. Кроме того, мы хотим вернуть первую ошибку (значение `Left`), возникшую в процессе преобразования пикселей. По счастью, для этого не нужно писать специальную функцию, поскольку мы можем воспользоваться тем фактом, что `Either a` является *монадой*.

В предыдущей главе мы узнали, что монада `Maybe` всегда сводится к `Nothing`, если в какой-то точке вычисления возникает `Nothing`. Это напоминает `Either a` и конструктор `Left`. Поэтому мы должны отобразить наши функции преобразования пикселей на список пикселей и обрабатывать эти вычисления в монаде `Either a`. Оказывается, в главе 4 мы уже видели функцию, которая именно это и делает, `mapM`:

```
ghci> :t mapM
mapM :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
ghci> :{
ghci| f :: Int -> Either String Int
ghci| f 0 = Left "zero is not allowed"
ghci| f x = Right x
ghci| :}
ghci> mapM f [1..10]
Right [1,2,3,4,5,6,7,8,9,10]
ghci> mapM f [0..10]
Left "zero is not allowed"
```

Но вот чего мы раньше не обсуждали, так это полный тип функции `mapM`, поскольку тогда не знали об ограничениях типов и монадах. `mapM` применяется к значениям типа `Traversable` (которому принадлежат и списки) в монадическом контексте. Переданная `mapM` функция вычисляет действие монады (`m b`) для каждого элемента, а результаты собирает в `m (t b)`. По сути дела, `mapM` обходит наш список и применяет

действия монады одно за другим. Поскольку монада `Either a` закорачивается в `Left`, все наше вычисление поступает так же!

> **ПРИМЕЧАНИЕ** Строго говоря, мы могли бы заменить `mapM` на `traverse`, имеющую тип `Applicative f => (a -> f b) -> t a -> f (t b)`. Функция `mapM` функционально ей эквивалентна, но пользуется ограничением `Monad`, а не `Applicative`. Для `Either a` это не имеет значения.

Теперь все кусочки пазла на месте, и мы готовы написать функцию проверки. Она должна проверять все утверждения и обертывать корректный тип `PnmImage` типом `DynamicPnmImage`, если проверка завершилась успешно. Функция показана в листинге ниже.

Листинг 13.11 Функция проверки результатов разбора PNM-файла

```
import qualified Data.Vector as V

...

validatePnm :: RawPnm -> Either String DynamicPnmImage
validatePnm img@(RawPnm {header, imageData})
  | not $ imageSizeCorrect img =
    Left $
      "The number of pixels given does not match "
        <> "the expected size inferred from the header"
  | indicatesBitmap magicNum =
    case maxVal header of
      Nothing -> PnmGray8Bit `withTransform`
➥ bitmapPixel
      Just _ ->
        Left $
          "Image seems to be a bitmap image "
            <> "but has a maximum value set"
  | indicatesGraymap magicNum =
    case maxVal header of
      Nothing ->
        Left $
          "Image seems to be a graymap image "
            <> "but has no maximum value set"
      Just maxVal ->
        if maxVal <= 255
          then
            PnmGray8Bit
              `withTransform` graymapPixel maxVal
          else
            PnmGray16Bit
              `withTransform` graymapPixel maxVal
  | indicatesPixmap magicNum =
    case maxVal header of
      Nothing ->
        Left $
```

Возвращает PnmGray8Bit, содержащий исходные данные, преобразованные в черно-белые пиксели

Возвращает PnmGray8Bit или PnmGray16Bit (в зависимости от максимальной величины), содержащий исходные данные, преобразованные в полутоновые пиксели

Возвращает сообщение об ошибке, если утверждение не выполнено

```
        "Image seems to be a pixmap image "
          <> "but has no maximum value set"
    Just maxVal ->
      if maxVal <= 255
        then
          PnmColor8Bit
            `withTransform` pixmapPixel maxVal P.PixelRGB8
        else
          PnmColor16Bit
            `withTransform` pixmapPixel maxVal P.PixelRGB16
| otherwise = Left "Image seems to be of unknown type"
where
  magicNum = magicNumber header
  width' = fromIntegral $ width header
  height' = fromIntegral $ height header

  withTransform ::
    (Int -> Int -> V.Vector px -> PnmImage px) ->
    (RawPixel -> Either String px) ->
    Either String DynamicPnmImage
  withTransform c f =
    let mConvertedData = mapM f $ pixels imageData
     in fmap mkRes mConvertedData
     where
       mkRes = DynamicPnmImage . c width' height'
➥ . V.fromList
```

Возвращает PnmGray8Bit или PnmGray16Bit (в зависимости от максимальной величины), содержащий исходные данные, преобразованные в цветные пиксели

Возвращает сообщение об ошибке, если утверждение не выполнено

Преобразует пиксели изображения, применяя переданную в аргументе функцию, и останавливается немедленно, если в процессе преобразования возникает ошибка

Создает окончательный результат в случае успешного преобразования

Преобразует пиксели в Vector и обертывает его типом DynamicPnmImage, используя конструктор, переданный в качестве аргумента

Поначалу это определение вселяет страх, но все становится на свои места, стоит только понять функцию `withTransform`. Она получает два аргумента: `c` – функция для построения `PnmImage`; в этом качестве выступают только конструкторы типа `PnmImage`. Аргумент `f` – функция, которая используется для преобразования исходных данных изображения в пиксельный тип, необходимый конструктору, переданному в аргументе `c`. Функция `withTransform` записана в инфиксном стиле, чтобы код было проще понять. Конструктор используется в сочетании с заданной функцией, чтобы выполнить преобразование пиксельных данных.

Как видим, `validatePnm` умеет определять вид `PnmImage`, исходя из переданных ей данных (сигнатуры и данных изображения), и возвращает конкретный конструктор, обернутый типом `DynamicPnmImage`. Пиксели преобразуются соответственно. Заметим, что вызывающая эту функцию сторона не может определить, какой `PnmImage` ей вернули. Всю ответственность несет `validatePnm`.

13.3 Обобщенный алгоритм преобразования изображения

Поскольку у нас теперь есть способ проверки и кодирования изображений, хорошо бы научиться что-то делать с ними. Быть может, мы хотим пикселизировать изображение или создать фильтр размытия, только вот как к этому подступиться? Поскольку пиксельные данные могут иметь разный тип в зависимости от конструктора `PnmImage`, нам может понадобиться несколько версий фильтра. Существует ли обобщенный способ отображения пикселей, составляющих изображение?

Для ответа на этот вопрос мы должны научиться обобщенно работать с пикселями. Необходимо преобразовать конкретный пиксельный тип (например, `Pixel8` или `PixelRGB16`) во что-то более общее. Это означает отображение типа в какую-то функцию, осуществляющую преобразование. Как мы уже знаем, классы типов предназначены именно для этого. Поэтому определим тип, представляющий обобщенный пиксель, и класс типов, который предоставляет функции преобразования такого пикселя в обе стороны. См. листинг ниже.

Листинг 13.12 Обобщенный тип и класс типов для пикселей

```
module Graphics.PNM.Conversion where

type GenericRGB = (Double, Double, Double)

class GenericPixel px where
  toGenericRGB :: px -> GenericRGB
  fromGenericRGB :: GenericRGB -> px
```

Псевдоним типа для представления обобщенного пикселя тремя значениями типа Double

Определение нового класса типов

Объявляет функцию, которая преобразует полиморфный тип в GenericRGB, обернутый классом типов GenericPixel

Объявляет функцию, которая преобразует значение типа GenericRGB в полиморфный тип, обернутый классом типов GenericPixel

Мы можем поместить код в листинге 13.12 в новый модуль, который назовем `Graphics.PNM.Conversion`. Тип `GenericRGB` представляет красный, зеленый и синий каналы пикселя в диапазоне от `0` до `1`. Это упрощает вычисления со значениями пикселей, поскольку не нужно приводить целые значения к типу `Double` и обратно внутри отображения.

Но так как наши пиксели имеют целые типы, нужен способ привести их к диапазону, требуемому типом `GenericRGB`. Пиксели принадлежат типам `Word8` или `Word16`, поэтому мы можем написать функции масштабирования. Они показаны в листинге ниже.

Листинг 13.13 Вспомогательные функции для преобразования целых типов в `Double` и обратно

```
import Data.Word

...

clamp :: Double -> Double
clamp = max 0 . min 1

scaleDown8Bit :: Word8 -> Double
scaleDown8Bit v = clamp $ (fromIntegral v) / 255.0

scaleUp8Bit :: Double -> Word8
scaleUp8Bit v = round $ 255 * (clamp v)

scaleDown16Bit :: Word16 -> Double
scaleDown16Bit v = clamp $ (fromIntegral v) / 65535.0

scaleUp16Bit :: Double -> Word16
scaleUp16Bit v = round $ 65535 * (clamp v)
```

Ограничивает значение интервалом от 0 до 1

Приводит Word8 к интервалу от 0 до 1

Приводит значение в интервале от 0 до 1 к типу Word8

Приводит Word16 к интервалу от 0 до 1

Приводит значение в интервале от 0 до 1 к типу Word16

С помощью этих функций мы можем определить экземпляры для различных пиксельных типов. Однако следует помнить, что `Pixel8` и `Pixel16` – синонимы типов, поэтому для создания их экземпляров необходимо активировать расширение языка `TypeSynonymInstances`. Сами объявления экземпляров не вызывают трудностей. Так как технически мы имеем красный, зеленый и синий каналы, нужно просто объединить эти каналы и нормировать результат для получения одного полутонового канала. Теоретически можно было бы использовать более изощренный обобщенный тип для различения цветных и полутоновых пикселей, но мы хотим не усложнять код, а упростить определение отображений.

Листинг 13.14 Экземпляры класса типов `GenericPixel`

```
import qualified Codec.Picture as P

...

instance GenericPixel P.Pixel8 where
  toGenericRGB p = (p', p', p')
    where
      p' = scaleDown8Bit p
  fromGenericRGB (r, g, b) = p
    where
      p = scaleUp8Bit $ (r + g + b) / 3

instance GenericPixel P.Pixel16 where
  toGenericRGB p = (p', p', p')
```

Возвращает GenericRGB для одного целого значения

Возвращает одно целое значение для GenericRGB

Возвращает GenericRGB для одного целого значения

```
    where
      p' = scaleDown16Bit p / 3
  fromGenericRGB (r, g, b) = p
    where
      p = scaleUp16Bit $ (r + g + b) / 3

instance GenericPixel P.PixelRGB8 where
  toGenericRGB (P.PixelRGB8 r g b) =
    (scaleDown8Bit r, scaleDown8Bit g, scaleDown8Bit b)
  fromGenericRGB (r, g, b) =
    P.PixelRGB8
      (scaleUp8Bit r)
      (scaleUp8Bit g)
      (scaleUp8Bit b)

instance GenericPixel P.PixelRGB16 where
  toGenericRGB (P.PixelRGB16 r g b) =
    (scaleDown16Bit r, scaleDown16Bit g, scaleDown16Bit b)
  fromGenericRGB (r, g, b) =
    P.PixelRGB16
      (scaleUp16Bit r)
      (scaleUp16Bit g)
      (scaleUp16Bit b)
```

Возвращает одно целое значение для GenericRGB

Возвращает GenericRGB для RGB-пикселя

Возвращает RGB-пиксель для GenericRGB

Возвращает GenericRGB для RGB-пикселя

Возвращает RGB-пиксель для GenericRGB

Разобравшись с преобразованиями, можно задуматься о том, как должна выглядеть наша функция отображения. Было бы хорошо просто предоставить интерфейс для применения какой-то чистой функции к одному пикселю, но этого могло бы быть недостаточно для реализации более интересных операций над изображениями. Мы хотим реализовать отображающие *ядра*.

13.3.1 Применение матриц преобразования к отображениям

Ядра, которые иногда называют *сверточными матрицами*, применяются в обработке изображений для комбинирования нескольких пикселей в один. Ядро можно рассматривать как матрицу, которая «скользит» по изображению; при этом центр матрицы совмещается с каждым пикселем для вычисления его нового значения. Это показано на рис. 13.1. Применение такой свертки к изображению порождает новое изображение. Разные матрицы дают разные эффекты, например увеличение резкости или размытие. Даже более сложные операции, такие как выделение границ, можно выполнить с помощью ядер.

Однако мы хотим не просто создать скучное ядро, а обобщить его, написав функцию, которая сможет беспрепятственно прочитать любой пиксель изображения и породить новое значение. Так мы сможем не только эмулировать ядра, но и писать функции, которые делают то, что одни лишь ядра сделать не могут.

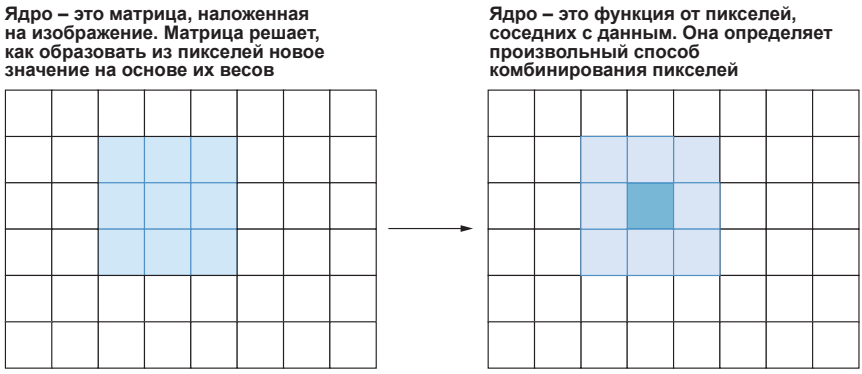


Рис. 13.1 Функциональность ядер, интерпретируемых как функции

Для этого мы создадим псевдоним типа для функции, способной решить эту задачу. Она принимает координаты текущего пикселя и функцию выборки произвольного пиксельного изображения. Если этой функции передать координаты пикселя за пределами изображения, то она должна вернуть не значение, а `Nothing`. А принимать эта функция будет текущий пиксель. Ее код показан в листинге 13.15.

Листинг 13.15 Синоним типа для обобщенного отображения пикселей

```
type PixelMapping =
  ( Int ->
    Int ->
    (Int -> Int -> Maybe GenericRGB) ->
    GenericRGB ->
    GenericRGB
  )
```

Координата x текущего пикселя

Координата y текущего пикселя

Функция, получающая пиксель изображения по его координатам

Текущий отображаемый пиксель изображения

Результирующий пиксель, оказавшийся в текущей позиции после отображения

При таком определении мы можем создать простое отображение каждого пикселя, ядро или что-то более сложное. Для построения функции отображения можно использовать функцию `imap` из модуля `Data.Vector`, которая будет принимать текущий индекс в процессе обхода всех пикселей:

```
ghci> import qualified Data.Vector as V
ghci> vec = V.fromList [1..10] :: V.Vector Int
ghci> V.imap (\i x -> (i, x)) vec
[(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(8,9),(9,10)]
```

Используя `imap` в сочетании с функциями из нашего класса `GenericPixel`, мы можем написать функцию отображения, которая при-

нимает `PixelMapping` и применяет его к пикселям, содержащимся в `DynamicPnmImage`, не затрагивая значение типа `PnmImage`. Ее код приведен в листинге ниже.

Листинг 13.16 Обобщенная функция, отображающая данные изображения

```
import qualified Data.Vector as V
import Graphics.PNM.Types

...

mapImagePixels ::
  PixelMapping ->
  DynamicPnmImage ->
  DynamicPnmImage
mapImagePixels f (DynamicPnmImage img) =
  case img of
    PnmGray8Bit _ _ pxs ->
      PnmGray8Bit `withPixels` mapping pxs
    PnmGray16Bit _ _ pxs ->
      PnmGray16Bit `withPixels` mapping pxs
    PnmColor8Bit _ _ pxs ->
      PnmColor8Bit `withPixels` mapping pxs
    PnmColor16Bit _ _ pxs ->
      PnmColor16Bit `withPixels` mapping pxs
  where
    w = pnmWidth img
    h = pnmHeight img

    withPixels ::
      (Int -> Int -> V.Vector px -> PnmImage px) ->
      V.Vector px ->
      DynamicPnmImage
    withPixels c pxs = DynamicPnmImage $ c w h pxs

    mapping :: GenericPixel px => V.Vector px -> V.Vector px
    mapping pixels =
      flip V.imap pixels $ \i px ->
        let x = i `mod` w
            y = i `div` w
            getPixel x' y' =
              toGenericRGB <$> pixels V.!? (y' * w + x')
            result = f x y getPixel $ toGenericRGB px
        in fromGenericRGB result
```

Сопоставляет с PnmImage и создает новый DynamicPnmImage, содержащий отображенные пиксели

Обертывает конструктор PnmImage и пиксели сопоставленного типа типом DynamicPnmImage с правильными шириной и высотой

Определяет функцию, которая безопасно обращается к пикселю внутри Vector и преобразует его в GenericRGB

Вычисляет координаты x и y по текущему индексу пикселя

Преобразует результат в правильный пиксельный тип

Вызывает функцию PixelMapping, передавая ей текущие координаты, функцию выборки произвольных пикселей из изображения и текущий пиксель

Функция `withPixels` используется так же, как рассмотренная выше функция `withTransform` в `validatePnm`. Обратите внимание на

использование `V.!?`, это безопасный вариант обычного индексирования типа `Vector`, который возвращает значение, обернутое `Maybe`:

```
ghci> import Data.Vector as V
ghci> vec = V.fromList [1..10] :: V.Vector Int
ghci> (V.!?) vec 0
Just 1
ghci> (V.!?) vec (-1)
Nothing
ghci> (V.!?) vec 10
Nothing
```

13.3.2 Экспорт изображений в формате PNG

Итак, мы научились работать с PNM-изображениями, остался последний шаг: записать их назад в файловую систему. Мы могли бы снова сериализовать данные в формате PNM, но хотим воспользоваться богатыми возможностями пакета JuicyPixels и записывать изображения в виде PNG-файлов. Для этого потребуется преобразовать наш тип `DynamicPnmImage` в тип `DynamicImage` из пакета.

Этот тип очень похож на наш, потому что содержит данные разных типов, различаемые по внутреннему типу пикселей. Однако это не GADT и не тип с навешенным квантором существования, а простой ADT с различными конструкторами для разных типов пикселей. Среди них есть конструкторы `ImageY8` и `ImageY16`, используемые для полутоновых изображений, а также конструкторы `ImageRGB8` и `ImageRGB16` для цветных изображений. Модуль `Codec.Picture` содержит функцию `generateImage` для построения изображений по функции; она имеет сигнатуру `Pixel px ? (Int -> Int -> px) -> Int -> Int -> Image px`. Эта функция получает функцию, которая возвращает пиксель с заданными координатами, а также ширину и высоту изображения. Сигнатура функции, передаваемой в первом аргументе, подозрительно похожа на сигнатуру функции извлечения произвольного пикселя, так что мы уже имеем представление о том, как ее писать. Ширина и высота изображения также известны, так что осталось только разветвиться по типу изображения и в зависимости от него построить правильное изображение. Код функции приведен ниже.

Листинг 13.17 Функция преобразования PNM-изображения в тип изображения из пакета JuicyPixels

```
dynamicPnmToDynamicImage :: DynamicPnmImage -> P.DynamicImage
dynamicPnmToDynamicImage (DynamicPnmImage img) =
  case img of
    PnmGray8Bit {} -> P.ImageY8 $ pnmToImage img
    PnmGray16Bit {} -> P.ImageY16 $ pnmToImage img
    PnmColor8Bit {} -> P.ImageRGB8 $ pnmToImage img
    PnmColor16Bit {} -> P.ImageRGB16 $ pnmToImage img
```

Сопоставляется с конструктором PnmImage и возвращает нужное изображение

```
  where
    pnmToImage :: P.Pixel px => PnmImage px -> P.Image px
    pnmToImage pnmImg =
      P.generateImage build (pnmWidth pnmImg) (pnmHeight pnmImg)
      where
        build x y =
          pnmPixels pnmImg V.! (y * (pnmWidth pnmImg) + x)
```

Определяет функцию извлечения пикселя с заданными координатами

Строит изображение по данным пикселей PnmImage

Здесь мы используем оператор `(!)` из модуля `Data.Vector`, потому что уверены, что индекс не выходит за границы вектора. В противном случае имела бы место ошибка в программе. Вы, наверное, недоумеваете, зачем мы повторяем `pnmToImage img` в каждом сопоставленном случае. Нельзя ли просто написать что-то вроде

```
dynamicPnmToDynamicImage (DynamicPnmImage img) =
  let img' = pnmToImage img
   in case img of
        PnmGray8Bit {} -> P.ImageY8 img'
        PnmGray16Bit {} -> P.ImageY16 img'
        PnmColor8Bit {} -> P.ImageRGB8 img'
        PnmColor16Bit {} -> P.ImageRGB16 img'
```

Нет, нельзя! Потому что сопоставления с GADT необычны. Они не просто уточняют тип – этот уточненный тип *действителен только* внутри выражения `case`, но не во внешней привязке `let` или `where`. Мы используем это уточнение, чтобы определить тип результирующего `pnmToImage`.

Наконец-то мы можем написать действие, которое записывает данные изображения в PNG-файл. Для этого используем функцию `writeDynamicPng` типа `FilePath -> DynamicImage -> IO (Either String Bool)`. Поскольку у нас уже есть функция для преобразования нашего `DynamicPnmImage` в `DynamicImage`, мы можем просто объединить обе функции, как показано в листинге ниже.

Листинг 13.18 Действие, сохраняющее динамическое PNM-изображение в формате PNG

```
writeAsPng ::
  FilePath ->
  DynamicPnmImage ->
  IO (Either String Bool)
writeAsPng path img =
  P.writeDynamicPng path $
    dynamicPnmToDynamicImage img
```

Функция `writeDynamicPng` возвращает ошибки в виде `Either`, если преобразовать или сохранить изображение не удалось. Таким обра-

зом, наш модуль `Graphics.PNM.Conversion` готов. Мы можем добавить новые функции в модуль `Graphics.PNM`, который реэкспортирует типы данных и функции из внутренних модулей.

Листинг 13.19 Модуль, который реэкспортирует определения из внутренних модулей

```
module Graphics.PNM
  ( module PNMC,
    module PNMP,
    module PNMT,
    module PNMV,
  )
where

import Graphics.PNM.Conversion as PNMC
import Graphics.PNM.Parsing as PNMP
import Graphics.PNM.Types as PNMT
import Graphics.PNM.Validation as PNMV
```

Экспортирует определения из импортированных модулей

Импортирует модули с указанием псевдонимов

Теперь можно перейти к интересной части – написанию функций отображения для нашей библиотеки, поскольку до сих пор мы лишь создавали инструменты. Пора бы уже ими воспользоваться.

13.4 Использование распараллеливания для преобразования данных

Давайте побалуемся – напишем различные отображения для нашей библиотеки обработки изображений. И будем тестировать их на изображении на рис. 13.2, которое имеется в репозитории этой книги.



Рис. 13.2 Изображение для экспериментов

Сначала напишем очень простое отображение, которое преобразует цветное изображение в полутоновое путем сложения красного, зеленого и синего каналов и нормирования суммы. Оно показано в листинге ниже, но большая часть возможностей нашего типа `PixelMapping` в нем игнорируется.

Листинг 13.20 Преобразование в полутоновое изображение

```
module Main where

import Graphics.PNM

grayScale :: PixelMapping
grayScale _ _ _ (r, g, b) = (v, v, v)
  where
    v = (r + g + b) / 3
```

Возвращает полутоновый пиксель

Складывает цветовые каналы и нормирует результат

Теперь займемся более интересным делом: пикселизацией изображения. Смысл в том, чтобы увеличить размер некоторых пикселей, перезаписав другие. Поскольку наш тип `PixelMapping` знает текущую позицию, он может проверить, для какого увеличенного пикселя он вызван, и извлечь этот пиксель. Но как определить увеличенный пиксель? Можно вычислить остаток от деления координат на фиксированную константу. Тогда мы сможем извлечь пиксель с отрицательным смещением от текущей позиции, как показано в листинге ниже.

Листинг 13.21 Пикселизация изображения

```
import Data.Maybe

...

pixelate :: Int -> PixelMapping
pixelate pixelSize x y getPx curPx =
  let xDiff = x `mod` pixelSize
      yDiff = y `mod` pixelSize
   in fromMaybe curPx $ getPx (x - xDiff) (y - yDiff)
```

Вычисляет разность между текущим и подлежащим увеличению пикселями

Извлекает подлежащий увеличению пиксель, по умолчанию подразумевая текущий

Это отображение, показанное в листинге 13.21, позволяет задать размер увеличенных пикселей. Применение его к оригинальному изображению с разными размерами дает изображения, показанные на рис. 13.3.

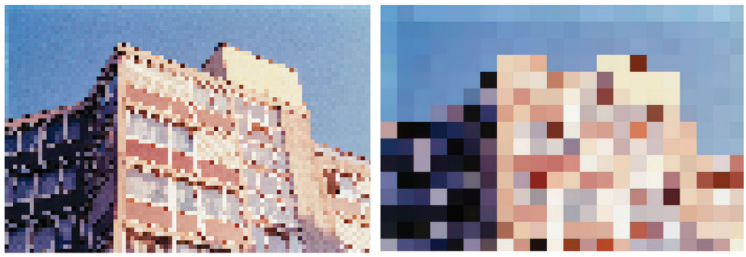


Рис. 13.3 Изображение, пикселизованное с помощью функций `pixelate 16` **и** `pixelate 64`

И последнее отображение, которое мы рассмотрим в этой главе, – *эффект размытия*. Есть много способов размыть изображение, но мы остановимся на самой бесхитростной реализации: прямоугольном размытии. Прямоугольным оно называется потому, что размытие достигается путем рисования прямоугольника заранее определенного размера вокруг размываемого пикселя и вычисления среднего по всем пикселям в этом прямоугольнике. В нашем типе `PixelMapping` это можно сделать, перебрав все пиксели внутри прямоугольника и вычислив средние каждого цветового канала. Отображение показано в следующем листинге.

Листинг 13.22 Прямоугольное размытие изображения

```
boxBlur :: Int -> PixelMapping
boxBlur boxSize x y getPx _ =
  let nbrs =
        catMaybes
          [ getPx (x + dx) (y + dy)
            | dx <- [(-boxSize) .. boxSize],
              dy <- [(-boxSize) .. boxSize]
          ]
      (rx, gx, bx) = unzip3 nbrs
      numPixs = fromIntegral $ length nbrs
  in ( sum rx / numPixs,
       sum gx / numPixs,
       sum bx / numPixs
     )
```

Отбрасывает пиксели, которые нельзя извлечь, потому что они лежат за пределами изображения

Перебирает пиксели внутри прямоугольника, окружающего текущий пиксель

Подсчитывает количество пикселей

Вычисляет среднее по цветовым каналам

Разбивает список (Double, Double, Double) на три списка, по одному на каждый цветовой канал

Применение этого отображения создает изображение, показанное на рис. 13.4. В его определении мы воспользовались функцией `unzip3` – вариантом `unzip` для списка трехэлементных кортежей, который возвращает три списка, содержащих первые, вторые и третьи элементы кортежей соответственно.



Рис. 13.4 Изображение, размытое функцией `boxBlur 8`

Отметим, что это отображение работает заметно дольше других. Поэтому оставшуюся часть главы мы посвятим его ускорению за счет *распараллеливания*.

13.4.1 Измерение времени

Для тестирования будем использовать показанный ниже главный модуль. Чтобы упростить код, программа принимает один аргумент и аварийно завершается при обнаружении ошибок во время разбора или проверки. В код зашито конкретное преобразование – прямоугольное размытие.

Листинг 13.23 Пример главного модуля для тестирования

```
module Main (main) where

...
import Graphics.PNM
import System.Environment (getArgs)
main :: IO ()
main = do
  [filePath] <- getArgs
  Right rawImg <- readPnmFile filePath
  let Right pnmImg = validatePnm rawImg
      pnmImg' = mapImagePixels (boxBlur 8) pnmImg
  _ <- writeAsPng "test.png" pnmImg'
  return ()
```

Выполнение этой программы для изображения large/p6.ppm из репозитория заняло 33 секунды на компьютере 2021 MacBook Pro. В этом можно убедиться, задав в командной строке флаг `-s`: `stack run-<path to the pnm image> +RTS -s`. Тогда вывод будет содержать информацию о времени работы программы:

```
INIT    time    0.000s  (  0.004s elapsed)
  MUT     time   23.866s  ( 28.727s elapsed)
  GC      time    2.794s  (  4.578s elapsed)
  EXIT    time    0.002s  (  0.009s elapsed)
  Total   time   26.662s  ( 33.318s elapsed)
```

Гм… не радует. Нельзя ли как-то ускорить, и если да, то как именно? Начнем с перечисления шагов программы:

1. Разобрать содержимое файла.
2. Проверить разобранные данные.
3. Отобразить данные изображения для получения нового.
4. Записать новое изображение в файл.

Проверка и запись изображения не настолько дороги, чтобы всерьез задуматься об их перепроектировании. Разбор требует какого-то времени, но не он главный подозреваемый, в чем легко убедиться, заменив `boxBlur` на `pixelate` в главном модуле. После этого программа работает меньше 2 секунд! Очевидно, что самая дорогостоящая часть – отображение.

Вернемся на шаг назад и проанализируем, что представляет собой `PixelMapping`. По сути дела, это не что иное, как простая *чистая* функция, с какими мы имеем дело, начиная с главы 2. Эта чистая функция применяется к каждому пикселю изображения, а значит, для изображения 1000×1000 пикселей нужно выполнить 1 миллион вычислений. Поскольку чистые функции не имеют побочных эффектов, порядок вычислений безразличен. Более того, отдельные вычисления не влияют друг на друга. Так почему бы не выполнять их параллельно?

13.4.2 Как устроен параллелизм?

Двоичный файл, являющийся результатом компиляции Haskell-программы, содержит не только наш код, но и обширную *систему времени выполнения*. В ее состав входит не только сборщик мусора, но и планировщик *зеленых потоков* (которые иногда называют *виртуальными*). Это не настоящие потоки, работающие на процессоре, а облегченные процессы внутри системы времени выполнения. У них есть много преимуществ:

- потоки можно планировать и быстро уничтожать;
- максимальное число потоков гораздо больше, чем позволила бы операционная система;
- обработка конкурентности и разделения ресурсов системой времени выполнения безопасна;
- система времени выполнения может принимать решения о планировании потоков с учетом доступных ресурсов;
- поведение системы времени выполнения настраивается с помощью параметров;
- имеются различные модели конкурентности.

Логично, что язык типа Haskell, где почти все, имеющее отношение к побочным эффектам, жестко контролируется, предоставляет абстракции для реализации безопасной многопоточности. Что касается конкурентности, нам доступно несколько пакетов асинхронных вычислений:

- пакет `async` – высокоуровневый интерфейс для безопасных асинхронных вычислений;
- пакет `stm` – программная транзакционная память внутри системы времени выполнения;
- модуль `Control.Concurrent` – низкоуровневый доступ к потокам.

Хотя все эти возможности интересны, к нам они не вполне приложимы, потому что нас интересует не *конкурентность*, а *параллелизм*. Разница в том, что нужные нам вычисления однотипны и должны работать параллельно. Для такого рода многопоточной модели имеется пакет `parallel`.

Этот пакет содержит модуль `Control.Parallel.Strategies`, который предлагает несколько способов выразить параллельное выполнение. Центральное понятие в нем – тип `Strategy`, который определяет, как следует организовать вычисление. `Strategy` – синоним функции типа `a -> Eval a`. `Eval` – это монада, которая описывает, как производить некоторое вычисление: что следует вычислять параллельно и какие вычисления по необходимости должны выполняться последовательно.

Для определения порядка вычислений в нашем распоряжении имеется несколько готовых стратегий:

- `r0` – не выполнять *никаких* вычислений;
- `rseq` – вычислять аргумент;
- `rpar` – параллельно вычислять аргумент в виртуальном потоке;
- `rdeepseq` – полностью вычислять аргумент.

В чем разница между `rseq` и `rdeepseq`? Ведь вроде бы обе вычисляют аргумент? Разница в том, что вычисление в Haskell означает не полное вычисление, а вычисление до так называемой *слабой головной нормальной формы*. Не вдаваясь глубоко в этот предмет, скажем лишь, что выражение приведено к слабой головной нормальной форме, если оно вычислено хотя бы до первого конструктора данных. Это значит, что список представлен в слабой головной нормальной форме, если он вычислен до первого конструктора `(:)` или `[]`. Однако вычислять значения в списке для этого необязательно! Это возможно благодаря концепции *переходников* (thunk) – элементов данных, которые не вычисляются, точнее вычисляются, только когда это необходимо. Это именно то, что делает Haskell ленивым.

В отличие от `rseq`, стратегия `rdeepseq` принудительно вычисляет *нормальную форму*, т. е. данные вычисляются полностью. Но для нас

самой важной является стратегия `grpar`, поскольку она выполняет параллельное вычисление.

13.4.3 Искры и контексты выполнения Haskell

Искры[1] (spark) – концепция, похожая на переходники. Если переходники заключают в себе невычисленные данные, то искры указывают на эти переходники. Вычисление искры означает вычисление переходника. Разница же в том, что искры могут вычисляться параллельно. Для этого система времени выполнения пользуется так называемыми *контекстами выполнения Haskell* (Haskell execution context – HEC). Грубо говоря, каждый контекст выполнения существует в своем собственном потоке операционной системы и управляется системой времени выполнения.

HEC'и умеют вычислять искры. Для этого они имеют доступ к *пулу искр*, заполняемому стратегией `grpar`. Планирование такого параллельного выполнения осуществляется путем *захвата работы* – это означает, что любой HEC, имеющий возможность взять новую искру, просто забирает ее из пула для вычисления. Эта идея показана на рис. 13.5.

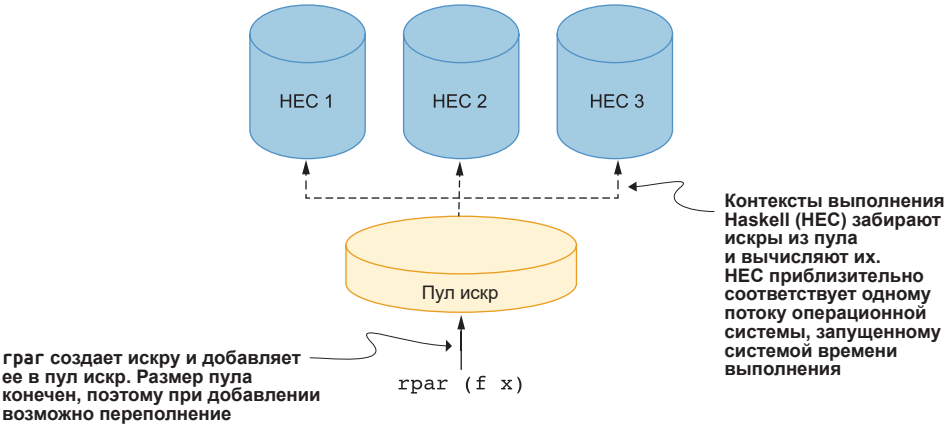


Рис. 13.5 Контексты выполнения Haskell и пулы искр

Конечно, на самом деле все несколько сложнее, поскольку искры могут быть вычислены преждевременно или убраны в мусор еще до

[1] Термин, конечно, не очень удачный, но эта метафора проведена в терминологии весьма последовательно (см. далее *fizzle* и *dud*), и я не счел себя вправе заменять ее чем-то другим. Такой неудачный опыт имел место, когда слово *thread* стали переводить как *поток* (безмерно перегрузив термин, т. к. тем же словом переводятся *flow*, *stream* и *flux*, что в ряде случаев приводит к путанице) вместо исходного значения *нить*. При этом была потеряна логическая связь между *thread* и *fiber* (волокно) – структурой, еще менее накладной, чем *thread*. – *Прим. перев.*

того, как HEC смог забрать их. Однако это детали, о которых заботится система времени выполнения, нас они не касаются.

Располагая этими знаниями, реализуем свое первое параллельное выполнение. Для этого нужно будет добавить пакет `parallel` в секцию `dependencies` файла `package.yaml`. Вычислением управляет монада `Eval`, а для его запуска используется функция `runEval`, которая получает значение от этой монады. Рассмотрим пример:

```
runEval $ do
  x' <- rpar x
  return x'
```

Это запускает параллельное выполнение `x` с помещением его в пул искр. Однако `return` не ждет завершения вычисления. Это означает, что если результирующее значение вычисляется кодом, вызывающим `runEval`, то искра становится недействительной. Мы говорим, что искра *потухла*. Если `x` уже было вычислено раньше вычисления `rpar`, то мы считаем его *муляжом*. Можно ли как-то гарантировать вычисление до `return`?

```
runEval $ do
  x' <- rpar x
  rseq x'
  return x'
```

В этом варианте мы сначала создаем искру, а затем принудительно запускаем последовательное вычисление еще до вызова `return`. Однако в этом нет никакого смысла, потому что хотели-то мы выполнить вычисление параллельно. Распространим наше вычисление на весь список. Модуль `Control.Parallel.Strategies` предлагает для этого несколько функций.

Одна из них, `evalList`, имеет тип `Strategy a -> Strategy [a]`. Мы можем передать стратегию, применяемую к каждому элементу списка, породив новую стратегию, которую затем можно будет использовать в сочетании со списком. Вызов мог бы выглядеть так:

```
map (^ 100) [1 .. 1000] `using` evalList rpar
```

Здесь вычисление, осуществляемое `map`, выполняется параллельно. Функция `using` из того же модуля вычисляет структуру данных, применяя указанную стратегию. Мы могли бы также записать это выражение с помощью функции `withStrategy`:

```
withStrategy (evalList rpar) $ map (^ 100) [1 .. 1000]
```

Параллельное вычисление спископодобных структур, безусловно, имеет смысл, особенно когда применение к одному элементу обходится очень дорого. Для нас это важно, потому что именно так об-

стоит дело с нашим типом `PixelMapping`! Однако мы имеем не список пикселей, а `Vector`, и возникает вопрос, как задать подходящую стратегию. Модуль предоставляет функцию `evalTraversable`, которая определяет способ применения стратегии к каждому элементу структуры типа `Traversable`, что прямо следует из ее типа: `Traversable t ? Strategy a -> Strategy (t a)`. Нам повезло, потому что экземпляр `Traversable Vector` уже существует!

Воспользуемся этим для параллельного вычисления `PixelMapping`! Для этого нужно добавить в нашу вспомогательную функцию `mapping` в `mapImagePixels` всего одну строчку кода, как показано в листинге ниже.

Листинг 13.24 Ошибочная стратегия параллельного отображения

```
import Control.Parallel.Strategies as S

...

mapping :: GenericPixel px => V.Vector px -> V.Vector px
mapping pixels =
  S.withStrategy (S.evalTraversable S.rpar) $
    flip V.imap pixels $ \i px ->
      let x = i `mod` w
          y = i `div` w
          getPixel x' y' =
            toGenericRGB <$> pixels V.!? (y' * w + x')
          result = f x y getPixel $ toGenericRGB px
       in fromGenericRGB result
```

Полностью распараллеленное вычисление вектора пикселей

Теперь можно выполнить программу повторно со следующими параметрами: `+RTS -N8 -s -RTS`. `-N8` говорит системе времени выполнения, что для вычисления следует использовать восемь процессоров, и, стало быть, будет задействовано восемь HEC. Благодаря флагу `-s` мы получим статистику вычисления искр.

> **ПРИМЕЧАНИЕ** Программу можно выполнить параллельно, потому что она была откомпилирована с флагом `-threaded`, который `stack` добавляет в конфигурацию сборки по умолчанию.

В результате запуска с такими параметрами мы получим следующую статистику:

```
SPARKS: 1318768 (10163 converted, 1210301 overflowed, 0 dud, 0 GC'd, 0 fizzled)
INIT time 0.000s ( 0.003s elapsed)
MUT time 23.793s ( 27.440s elapsed)
GC time 2.758s ( 3.972s elapsed)
EXIT time 0.000s ( 0.011s elapsed)
Total time 26.552s ( 31.427s elapsed)
```

Как видим, производительность программы почти не изменилась. Игра вообще не стоила свеч! Но погодите, давайте разберемся, что нам говорит статистика искр. Мы видим, что всего было создано 1 318 768 искр. Поскольку на вход программе было подано изображение размера 1397×944 пикселей, получается, что на каждый пиксель было создано по искре, так что отображение-то вроде работает. 10 163 из этих искр были *конвертированы* (converted), т. е. HEC взял их из пула и вычислил. Ни одна искра не потухла (fizzled), ни одна не была убрана в мусор и ни одна не сочтена *муляжом* (*dud*), т. е. наше отображение не порождает лишних искр. Искра *убирается в мусор* (GC'd), если еще до того, как HEC до нее добрался, оказалось, что вычислять ее не нужно. Самое ошеломительное число – 1 210 301 искра, которые что-то там *переполнили* (*overflowed*). Что бы это значило?

Ответ простой: пулы искр имеют фиксированный размер. Если за короткое время создается слишком много искр, то пул может оказаться заполненным, и происходит переполнение. В таком случае искры просто отбрасываются, и никакого распараллеливания не происходит. Вот в эту ловушку мы и попались! Наша программа фактически не вычисляет отображение параллельно.

И как избежать переполнения пула искр? Конечно, нужно создавать меньше искр, но при этом все равно вычислять параллельно. Нужно выполнять *больше работы* в каждой искре. Для этого мы можем разбить работу на крупные *блоки*, которые затем вычислять как искры.

Модуль `Control.Parallel.Strategies` предлагает функцию `parListChunk`, которая определяет для списков стратегию, работающую с блоками заданного размера. Однако эта функция применима только к спискам. Чтобы все же воспользоваться ей, мы должны будем преобразовать `Vector` в список, выполнить параллельное вычисление, а затем преобразовать список обратно в `Vector`. Напишем эту функцию и поместим ее в новый модуль `Data.Vector.Strategies`, полный код которого приведен в следующем листинге.

Листинг 13.25 Модуль блочного параллельного выполнения для типа `Vector`

```
module Data.Vector.Strategies (parVectorChunk) where

import Control.Parallel.Strategies
import qualified Data.Vector as V

parVectorChunk :: Int -> Strategy (V.Vector a)
parVectorChunk size =
  fmap V.fromList . parListChunk size rseq . V.toList
```

Импортирует необходимые модули

Определяет стратегию параллельного вычисления блоков вектора размера size

Почему мы используем стратегию `rseq`? Потому что `parListChunk` уже производит вычисления параллельно. Использование `rpar` привело бы к созданию искрой новой искры, а это бесполезная работа!

Теперь мы можем использовать `parVectorChunk` в нашей функции `mapping`. Но как определить, сколько блоков нам нужно? Увы, наука эта не точна, потому что искры обходятся не даром. Из-за расходов на управление и сборку мусора создание новых искр (даже если пул при этом не переполняется) может привести не к более быстрому, а к более медленному выполнению. Тут все зависит от того, сколько работы можно проделать в одной искре. В конечном итоге программист должен сам настраивать такие параметры и на тестах проверять, дают ли они какой-нибудь выигрыш.

Мы создадим порядка 1000 искр. Это можно сделать, вычислив размер блока по общему количеству пикселей в изображении.

Листинг 13.26 Распараллеленная функция `mapping` для отображения пикселей изображения

```
import Control.Parallel.Strategies as S
import qualified Data.Vector as V
import qualified Data.Vector.Strategies as V
...
  mapping :: GenericPixel px => V.Vector px -> V.Vector px
  mapping pixels =
    S.withStrategy (V.parVectorChunk
➥ ((w * h) `div` 1000)) $
      flip V.imap pixels $ \i px ->
        let x = i `mod` w
            y = i `div` w
            getPixel x' y' =
              toGenericRGB <$> pixels V.!? (y' * w + x')
            result = f x y getPixel $ toGenericRGB px
         in fromGenericRGB result
```

Импортирует модуль, содержащий параллельные стратегии

Импортирует Data.Vector и Data.Vector.Strategies в одно и то же пространство имен

Разбивает данные изображения примерно на 1000 блоков и вычисляет их параллельно

После этой модификации мы можем протестировать отображение для прямоугольного размытия. Статистика являет весьма приятное взгляду зрелище:

```
SPARKS: 1001 (1001 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time   0.000s  ( 0.004s elapsed)
  MUT     time  27.682s  ( 5.502s elapsed)
  GC      time   3.085s  ( 2.180s elapsed)
  EXIT    time   0.000s  ( 0.000s elapsed)
  Total   time  30.767s  ( 7.686s elapsed)
```

Мало того, что пропало переполнение, так еще и время выполнения снизилось с 33 до примерно 8 секунд, хотя *ни одна строчка кода не изменилась*!

На этом мы завершаем создание библиотеки параллельной обработки изображений. Мы успешно выполнили проверку PNM-файлов, применив типобезопасное динамическое кодирование данных. Кроме того, мы придумали общий метод отображения пикселей изображения для создания различных фильтров и эффектов. Наконец, мы значительно повысили производительность общего алгоритма, воспользовавшись встроенной в Haskell поддержкой распараллеливания.

Резюме

- *Обобщенные алгебраические типы* (GADT) используются для *уточнения* полиморфного типа путем определения конкретных типов для конструкторов полиморфных в остальном переменных типа.
- `Vector` – высокоуровневый тип, похожий на списки, но с характеристиками производительности, аналогичными массивам. Он обеспечивает очень быстрый доступ, но имеет недостаток – медленное динамическое выделение памяти при частом добавлении данных.
- Типы с навешенным квантором существования в сочетании с GADT позволяют обойти полиморфизм по типу возвращаемого значения, но при этом по-прежнему использовать параметрически полиморфные типы, что дает возможность принимать решение о типе результата внутри вызываемой функции, не оставляя его на усмотрение вызывающей стороны.
- Чистая функция, единообразно применяемая к коллекции данных, легко поддается распараллеливанию с применением стратегий вычисления, предоставляемых модулем `Control.Parallel.Strategies`.
- В Haskell встроена поддержка параллельного выполнения с помощью искр, которые можно создать посредством монады `Eval` и стратегии `rpar` или любой из готовых параллельных стратегий. Это позволяет прозрачно включать детерминированный параллелизм в наши программы без изменения алгоритмов.
- Создание слишком большого числа искр приводит к переполнению пула искр и делает распараллеливание неэффективным, но это можно исправить, предварительно разбив работу на крупные блоки, которые можно вычислять параллельно.

14 Файлы и исключения

Краткое содержание этой главы:

- работа с файлами на низком уровне;
- побайтное чтение данных из файла;
- исследование файловой системы и работа с исключениями;
- определение, возбуждение и перехват исключений.

До сих пор мы рассматривали проекты, связанные с чтением данных, их преобразованием и тем или иным способом вывода. Нас больше интересовал «счастливый путь», на котором взаимодействия со средой сведены к минимуму, так что об ошибках можно было не беспокоиться. В этой главе все изменится.

Проект, которым мы сейчас займемся, – это утилита файловой синхронизации, которая копирует файлы и подкаталоги из одного каталога в другой, но лишь те, которых еще нет в целевом каталоге. Для этого нам придется узнать об *исключениях* и их обработке – это и есть предмет настоящей главы.

14.1 Открытие и чтение файлов

Иногда нам нужно синхронизировать два каталога; это может быть резервное копирование на сетевое хранилище или синхронизация файлов в промежуточной и производственной среде. Обычно для этой цели используется программа `rsync`. Но мы хотим научиться создавать подобные инструменты самостоятельно. Программа должна обходить исходный каталог и копировать отсутствующие и измененные файлы в конечный каталог, как показано на рис. 14.1. Если файл имеется в конечном каталоге, но отсутствует в исходном, мы просто игнорируем его, не пытаясь ни удалить, ни переместить.

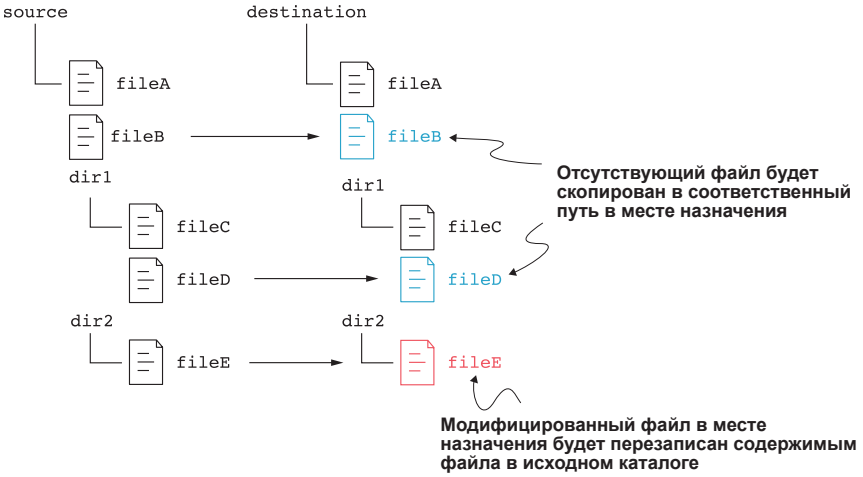


Рис. 14.1 Поведение файлового синхронизатора

Чтобы отличить наш инструмент синхронизации от других, мы дополнительно попробуем определять типы файлов и ограничиться синхронизацией файлов определенных типов. При написании утилиты нам придется взаимодействовать со средой несколькими способами:

- копировать файлы;
- читать метаданные из файлов;
- создавать каталоги;
- читать часть файла для определения его типа.

Прежде чем начинать проект, мы должны разобраться, как устроена работа с файлами в Haskell.

14.1.1 System.IO и Handle

До сих пор в наших проектах встречалась только самая простая работа с файлами. Нам приходилось либо читать весь файл в программу, либо записывать какие-то данные в новый файл.

Теперь мы хотим копнуть глубже. Сначала рассмотрим модуль `System.IO`, в котором определены многие операции чтения и записи файлов. Из важных типов в этом модуле отметим `Handle`, который содержит *описатель файла*, открытого процессом, и связанные с ним свойства, в том числе:

- доступен ли файл для чтения или записи;
- открыт или закрыт файл;
- текущие позиции чтения и записи файла;
- способ буферизации при записи.

Операции, определенные для этого типа, похожи на операции, применимые к файловым дескрипторам в Unix-подобных системах. Как и в Unix, описатели `stdout`, `stderr` и `stdin` автоматически открываются в начале работы программы, и с ними мы неявно взаимодействовали (например, используя функцию `putStrLn`, мы записывали в `stdout`). Однако важно иметь в виду, что этими описателями управляет система времени выполнения Haskell, и от операционной системы они, вообще говоря, не зависят.

> **ПРИМЕЧАНИЕ** При работе с типом `Handle` следует помнить, что некоторые виды поведения по умолчанию *платформенно-зависимы*, т. е. зависят от системы, в которой работает программа. Поэтому при работе на таком низком уровне необходима осторожность.

Так как же работать с этим типом? Модуль предоставляет функцию `openFile` типа `FilePath -> IOMode -> IO Handle`, которая создает описатель `Handle`, управляющий доступом к файлу по указанному пути. Аргумент типа `IOMode` управляет взаимодействием с `Handle`:

- `ReadMode` – открыть файл для чтения;
- `WriteMode` – открыть или создать файл для записи и, если он существует, стереть старое содержимое, сделав длину файла нулевой;
- `AppendMode` – открыть или создать файл для записи и установить позицию записи в конце файла;
- `ReadWriteMode` – открыть или создать файл для чтения и записи, не стирая содержимое.

Открыв описатель, мы можем поработать с ним. Ниже приведен пример сеанса GHCi:

```
ghci> import System.IO
ghci> h <- openFile "test.txt" WriteMode
ghci> hPutStrLn h "Hello World"
ghci> hClose h
ghci> h2 <- openFile "test.txt" ReadMode
ghci> hGetLine h2
```

```
"Hello World"
ghci> hClose h2
```

В предположении, что `test.txt` ранее не существовал, теперь он содержит короткое сообщение. Функция `hPutStrLn` записывает заданную строку в указанный `Handle`, а `hClose` закрывает его. Описатели закрываются по двум причинам:

- закрытие описателя сбрасывает его буфер на диск;
- несколько открытых описателей одного и того же файла могут привести к исключению.

Такое исключение легко вызвать, просто открыв один и тот же файл дважды – для чтения и для записи:

```
ghci> import System.IO
ghci> h1 <- openFile "test.txt" ReadMode
ghci> h2 <- openFile "test.txt" WriteMode
*** Exception: test.txt: openFile: resource busy (file is locked)
```

Поэтому важно держать описатели открытыми не дольше, чем необходимо. А что там насчет буферов? Посмотрим, как они работают.

14.1.2 Буферизация и поиск

С каждым описателем связан буфер, в котором изменения запоминаются перед тем, как записать их в файловую систему. Простая функция `hPutStrLn`, как правило, записывает строку не в файл, а только в буфер, который сбрасывается на диск функцией `hFlush` или в результате закрытия описателя функцией `hClose`. На поведение буферизации влияет функция `hSetBuffering`, получающая аргумент типа `BufferMode`, который может принимать следующие значения:

- `NoBuffering` – отключает буферизацию, т. е. данные записываются в файл немедленно;
- `LineBuffering` – буфер сбрасывается после того, как в него записан знак новой строки;
- `BlockBuffering` – буферизуется заданное или определенное системой по умолчанию количество элементов.

Заметим, что эти режимы задают *верхний предел* заполнения буфера. Он может сбрасываться на диск и чаще. Кроме того, у разных описателей разные умолчания:

```
ghci> import System.IO
ghci> h <- openFile "test.txt" ReadMode
ghci> hGetBuffering stdout
NoBuffering
ghci> hGetBuffering h
BlockBuffering Nothing
```

При работе с *допускающим поиск* описателем, все равно, для чтения или для записи, отслеживается позиция в ассоциированном с ним файле. Эта позиция автоматически продвигается вперед при чтении и записи, но ее можно установить и вручную. Описатели вроде `stdout` не допускают поиска, поэтому невозможно откатиться на предыдущую позицию, т. к. все, что вы в них записали, в конечном счете будет напечатано.

Но описатели, ассоциированные с файлом, допускают поиск. Для получения и установки позиции описателя служат соответственно функции `hTell` и `hSeek`:

```
ghci> import System.IO
ghci> h <- openFile "test.txt" WriteMode
ghci> hTell h
0
ghci> hPutStr h "Hello"
ghci> hTell h
5
ghci> hPutStr h " "
ghci> hTell h
6
ghci> hPutStr h "World"
ghci> hTell h
11
ghci> hSeek h AbsoluteSeek 6
ghci> hPutStr h "Haskell"
ghci> hClose h
```

В результате в файл будет записана строка `"Hello Haskell"`, потому что мы *искали* позицию первой буквы слова `"World"`, а затем перезаписали ее функцией `hPutStr`. Функция `hSeek` поддерживает несколько режимов поиска, определяемых параметром `SeekMode`:

- `AbsoluteSeek` – точная позиция описателя задается смещением от начала файла;
- `RelativeSeek` – позиция описателя задается смещением от текущей позиции;
- `SeekFromEnd` – позиция описателя задается смещением от конца файла.

Хотя мы и сообщили эти детали для полноты картины, часто пользоваться ими нам не придется, потому что буферизация и поиск – примеры того, чего мы всеми силами пытаемся избежать: побочных эффектов! При работе с описателями мы почти всегда зависим от их внутреннего состояния. Операции с описателями не являются ссылочно-прозрачными, поэтому использовать их опасно, если не следить за позициями, режимом буферизации и состоянием, поскольку можно нарваться на исключения. Ниже мы обсудим обработку таких

исключений, а пока хотим воспользоваться описателями для определения типа файла.

14.2 Чтение байтов из файла

Во многих файловых форматах имеется сигнатура (иначе называемая магическим числом), которая позволяет понять, с какими данными работает программа. Мы хотим написать действие ввода-вывода, которое будет разбирать его сигнатуру и таким образом определять тип заданного файла.

Создадим новый проект командой `stack new` и добавим в секцию `dependencies` пакет `bytestring`, потому что собираемся разбирать отдельные байты файла, а для этого нужен модуль `Data.ByteString`. Затем создадим модуль `System.FileType` в каталоге `src`.

> **ПРИМЕЧАНИЕ** Модуль `System.IO` предоставляет действия для работы со значениями типа `Handle` и `String`. Поскольку `String` – не самый эффективный тип данных, приложения, которым нужна высокая производительность, должны отдавать предпочтение типу `Data.ByteString` из пакета `bytestring` или типу `Data.Text.IO` из пакета `text`, которые работают с описателями непосредственно.

Можно было бы потратить всю жизнь на ознакомление с сигнатурами наиболее употребительных файловых типов, но мы ограничимся только несколькими типами, показанными в табл. 14.1.

Таблица 14.1 Сигнатуры некоторых файловых форматов

Формат файла	Расширение файла	Сигнатура (шестнадцатеричная)	Категория
Растр	`.bmp`	`42 4d`	Изображение
PNG	`.png`	`89 50 4e 47 0d 0a 1a 0a`	Изображение
JPEG	`.jpg, .jpeg`	`ff d8 ff`	Изображение
bzip2	`.bz2`	`42 5a 68`	Сжатый
gzip	`.gz`	`1f 8b`	Сжатый

Теперь можно приступить к написанию модуля. Сначала определим тип для категорий файловых типов:

```
data FileType
  = Other
  | Image
  | Compressed
  deriving (Show, Eq)
```

Для нашего действия ввода-вывода необходима функциональность из модулей `Data.ByteString` и `System.IO`. Ее можно импортировать следующим образом:

```
import qualified Data.ByteString as BS
import System.IO (IOMode (ReadMode), hClose, openFile)
```

Модуль `Data.ByteString` предоставляет несколько действий ввода-вывода, работающих с типом `Handle`. Нам понадобится `hGet` типа `Handle → Int → IO ByteString`. Это действие читает не более указанного числа байтов из `Handle` в `ByteString`. Важно, что она не завершается ошибкой, встретив конец файла (EOF). С ее помощью мы можем прочитать минимальное число байтов, необходимое для проверки сигнатуры, а затем применить сопоставление с образцом байтового представления `ByteString` в виде `[Word8]`. Преобразованием займется функция `unpack`. В листинге ниже показана первая версия этого действия.

Листинг 14.1 Пример небезопасного чтения из описателя файла

```
determineFileType :: FilePath -> IO FileType
determineFileType path = do
  hdl <- openFile path ReadMode
  signature <- BS.hGet hdl 8
  let fileType = case BS.unpack signature of
       0x42 : 0x4d : _ -> Image
       0x89 : 0x50 : 0x4e : 0x47 : 0x0d : 0x0a : 0x1a : 0x0a
➥ : _ -> Image
       0xff : 0xd8 : 0xff : _ -> Image
       0x42 : 0x5a : 0x68 : _ -> Compressed
       0x1f : 0x8b : _ -> Compressed
       _ -> Other
  hClose hdl
  return fileType
```

Открывает описатель файла с указанным именем

Читает первые восемь байт из описателя

Определяет тип файла, сравнивая прочитанные байты с сигнатурами

Закрывает описатель

Возвращает результат

Технически закрывать `Handle` необязательно, потому что сборщик мусора сделает это, когда будет убирать описатель. Однако поступать так считается дурным тоном, потому что мы не можем знать, когда именно у GC дойдут до этого руки. А тем временем мы могли бы снова открыть другой описатель того же файла, что привело бы к проблемам.

> **ПРИМЕЧАНИЕ** Использовать `unpack` при сравнении префиксов значений типа `ByteString` необязательно. Можно было бы вместо этого вызвать функцию `isPrefixOf` из модуля `Data.ByteString`. Для крупных блоков данных это заметно быстрее.

Еще одна проблема – исключения. Что, если наш код возбудит исключение перед вызовом `hClose`? Это общая проблема захвата и освобождения ресурсов.

14.2.1 Захват ресурсов и функция bracket

Эта проблема настолько широко распространена, что Haskell предоставляет специальную функцию `bracket` для ее решения. Она импортируется из модуля `Control.Exception` и имеет тип `IO a → (a → IO b) → (a → IO c) → IO c`. Ей передаются три действия:

- `a` – захватывает ресурс и передает его сначала `w`, а затем `r`;
- `w` – использует захваченный ресурс и порождает результат;
- `r` – освобождает захваченный ресурс.

Функция `bracket` специальная в том смысле, что гарантирует вызов действия освобождения ресурса, даже если во время работы действия было возбуждено исключение. Воспользовавшись этим, мы можем переписать `determineFiletype` и на этом покончить с новым модулем.

Листинг 14.2 Полный код модуля `System.FileType`

```
module System.FileType
  ( FileType (..),
    determineFileType,
  )
where

import Control.Exception (bracket)
import qualified Data.ByteString as BS
import System.IO (IOMode (ReadMode), hClose, openFile)

data FileType
  = Other
  | Image
  | Compressed
  deriving (Show, Eq)

determineFileType :: FilePath -> IO FileType
determineFileType path =
  bracket (openFile path ReadMode) hClose $ \hdl -> do
    signature <- BS.hGet hdl 8
    return $ case BS.unpack signature of
      0x42 : 0x4d : _ -> Image
      0x89 : 0x50 : 0x4e : 0x47 : 0x0d : 0x0a : 0x1a : 0x0a
➥ : _ -> Image
      0xff : 0xd8 : 0xff : _ -> Image
      0x42 : 0x5a : 0x68 : _ -> Compressed
      0x1f : 0x8b : _ -> Compressed
      _ -> Other
```

Определяет список экспорта для модуля

Импортирует необходимые функции и типы

Определяет тип для категорий файлов

Обертывает открытие и закрытие описателя файла функцией bracket

Читает первые восемь байт открытого файла

Определяет тип файла, сравнивая прочитанные байты с сигнатурами, и возвращает результат

В листинге 14.2 видно, как `bracket` делает код модуля чище. Благодаря порядку аргументов функции становится возможно быстро писать обертки для разных ресурсов. Таким образом, код для работы

с внешними ресурсами – файлами, сетевыми соединениями, базами данных и т. д. – оказывается более безопасным.

> **ПРИМЕЧАНИЕ** Идиома `bracket (openFile path mode) hClose` встречается очень часто, и ее можно заменить функцией `withFile path mode` из модуля `System.IO`, эквивалентной, но более удобочитаемой.

Теперь, понимая, как читать и записывать файлы, перейдем к изучению вопроса о работе с каталогами.

14.3 Работа с файловой системой и исключения

До сих пор мы видели только один метод ссылки на файлы: использование типа `FilePath` для задания пути к файлу. Однако разрабатываемый нами инструмент должен иметь дело в основном с каталогами. Нам необходим способ их создания и перечисления содержимого.

Для этого мы будем использовать модули `System.Directory` и `System.FilePath`, для доступа к которым нужно добавить в файл `package.yaml` пакеты `directory` и `filepath`. Первый модуль содержит системно-независимые функции для создания, удаления, перемещения и получения содержимого каталогов, а второй – служебные функции для создания путей к файлам и разбиения их на части.

14.3.1 Модули System.Directory и System.FilePath

Познакомимся с применением вышеупомянутых модулей `System.Directory` и `System.FilePath`. Пусть имеется каталог с такой структурой:

```
foo
├── bar
│   ├── fileB.txt
│   └── fileC.txt
├── baz
│   ├── fileD.txt
│   └── fileE.txt
└── fileA.txt
```

Вот как мы можем с ним взаимодействовать с помощью функций из этих модулей:

```
ghci> import System.Directory as Dir
ghci> import System.FilePath as FP
ghci> Dir.listDirectory "foo"
```

```
["fileA.txt","baz","bar"]
ghci> Dir.listDirectory ("foo" </> "bar")
["fileB.txt","fileC.txt"]
ghci> Dir.listDirectory ("foo" </> "baz")
["fileD.txt","fileE.txt"]
ghci> map FP.dropExtension <$> Dir.listDirectory "foo"
["fileA","baz","bar"]
```

Функция `listDirectory` перечисляет содержимое каталога, но возвращает только базовые имена файлов и каталогов. Оператор `(</>)` помогает соединить их, не думая о разделителях компонентов пути. Модуль `System.FilePath` содержит много функций для работы с файлами и их расширениями, в т. ч. `dropExtension` и `takeExtension`. В последней строке мы видим важное ограничение на функции из модуля `System.FilePath`: они не знают, что является файлом, а что – нет. Модуль манипулирует строками, а обязанность правильно их использовать возлагается на нас.

14.3.2 Перечисление файлов и каталогов

Нам требуется получать отдельно списки файлов и каталогов, а затем копировать файлы и обходить каталоги. Откуда нам знать, какой путь ведет к файлу, а какой к каталогу? Для этого предназначены функции `doesFileExist` и `doesDirectoryExist` из модуля `System.Directory`:

```
ghci> Dir.listDirectory "foo"
["fileA.txt","baz","bar"]
ghci> Dir.doesFileExist ("foo" </> "fileA.txt")
True
ghci> Dir.doesFileExist ("foo" </> "bar")
False
ghci> Dir.doesDirectoryExist ("foo" </> "bar")
True
```

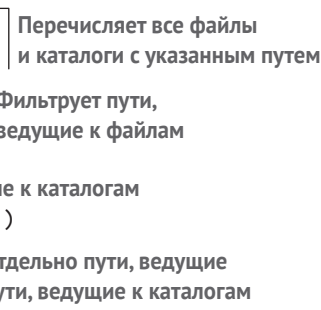
Обе функции имеют тип `FilePath -> IO Bool`, поэтому нельзя просто использовать функцию `filter` для проверки путей. Нам необходим монадический эквивалент этой функции, который дает модуль `Control.Monad`. Подобно тому, как `mapM` является монадической версией `map`, монадическая версия `filter` называется `filterM`, а ее тип – `Applicative m => (a -> m Bool) -> [a] -> m [a]`.

Листинг 14.3 Действие для раздельного перечисления файлов и каталогов

```
module System.Directory.Safe where

import System.Directory as Dir
import System.FilePath ((</>))
import Control.Monad (filterM)
```

```
listFilesAndDirectories :: FilePath -> IO ([FilePath], [FilePath])
listFilesAndDirectories pathPrefix = do
  paths <- Dir.listDirectory pathPrefix
  files <-
    filterM
      (Dir.doesFileExist . (pathPrefix </>))
      paths
  dirs <-
    filterM
      (Dir.doesDirectoryExist . (pathPrefix </>))
      paths
  return (files, dirs)
```



Теперь мы можем использовать эту функцию, чтобы обойти дерево из нашего примера и получить список находящихся в нем файлов. Пути *относительные*, т. е. путь, возвращенный `listFilesAndDirectories`, становится правильным только после дописывания в его начало пути к каталогу, который использовался для его получения:

```
ghci> listFilesAndDirectories "foo"
(["fileA.txt"],["baz","bar"])
ghci> listFilesAndDirectories $ "foo" </> "bar"
(["fileB.txt","fileC.txt"],[])
ghci> listFilesAndDirectories $ "foo" </> "baz"
(["fileD.txt","fileE.txt"],[])
ghci> listFilesAndDirectories $ "foo" </> "foo"
*** Exception: foo/foo: getDirectoryContents:openDirStream:
➥ does not exist (No such file or directory)
```

Отметим важное свойство этой функции. Она возбуждает исключение, если указанного пути не существует. Это подводит нас к неудобной части работы с файловой системой: исключениям.

> **Упражнение. Все файлы в каталоге**
>
> Пользуясь функциями из модулей `System.Directory` и `System.FilePath`, напишите действие ввода-вывода, которое рекурсивно перечисляет все файлы в заданном каталоге. Ее применение к нашему примеру `foo` должно возвращать плоский список файлов в произвольном порядке:
>
> ```
> ghci> listFilesRecursively "foo"
> ["foo/fileA.txt","foo/baz/fileD.txt","foo/baz/fileE.txt",
> ➥ "foo/bar/fileB.txt","foo/bar/fileC.txt"]
> ```

14.3.3 Основы исключений

В этой книге мы видели несколько подходов к обработке ошибок. Мы кодировали факультативные значения и результаты неудачных

вычислений, возвращая значение типа `Maybe` или `Either`. Мы также использовали действие `fail` из класса типов `MonadFail` в главе 12, где реализовали экземпляр, чтобы парсер мог завершиться с ошибкой, не утянув за собой всю программу.

Напомним, что класс `MonadFail` предоставляет действие `fail`, которое приводит к неудачному завершению монадического вычисления с заданным сообщением об ошибке:

```
class Monad m => MonadFail m where
  fail :: String -> m a
```

Заметим, что действие возвращает `m a` для произвольного `a`. Как реализация может изобрести значение? В случае парсера мы возвращали значение типа `Either String a`, которое было создано конструктором `Left` со значением типа `String`. Но какого рода ошибку может породить монада `IO`?

```
ghci> fail "Whoops" :: IO ()
*** Exception: user error (Whoops)
```

Оказывается, что `fail` в монаде `IO` просто возбуждает исключение, что выглядит вполне разумно, если взглянуть на сигнатуры типов обычных функций для возбуждения исключений:

```
ghci> import Control.Exception
ghci> :t throw
throw :: Exception e => e -> a
ghci> :t throwIO
throwIO :: Exception e => e -> IO a
```

Функции `throw` и `throwIO` способны выдать значения любого типа. Это похоже на две другие встречавшиеся нам функции, способные возбуждать исключения:

```
ghci> :t error
error :: GHC.Stack.Types.HasCallStack => [Char] -> a
ghci> :t undefined
undefined :: GHC.Stack.Types.HasCallStack => a
```

Исключения в чистом функциональном языке программирования, каковым является Haskell, – штука хитрая. Можно сказать, что исключения не укладываются в чистый функциональный дизайн, потому что являются *побочными эффектами*. Они необратимо нарушают поток управления программы, потому что немедленно передают управление от функции вызывающей ее стороне. До сих пор мы не имели дело с исключениями, т. к. в основном ограничивались чистыми функциями.

Однако исключения – неотъемлемая часть работы с системой вне нашей программы. Файлы, сети и другие процессы – примеры рабо-

ты с окружением, которая в любой момент может завершиться неудачно. Файлы могут исчезать, сетевые подключения могут рваться, другие процессы могут «падать». Исключения – это реальная часть внешнего мира, поэтому о них нужно знать.

14.4 Возбуждение и перехват исключений

Рассмотрим модуль `Control.Exception`, в котором определены типы и функции для работы с исключениями. `Exception` – класс типов, описывающий, как преобразовать значение в тип `SomeException` и обратно, а также как отобразить его:

```
class (Typeable e, Show e) => Exception e where
  toException :: e -> SomeException
  fromException :: SomeException -> Maybe e
  displayException :: e -> String
```

`SomeException` – тип с навешенным квантором существования, похожий на наш тип `DynamicPnmImage`, введенный в предыдущей главе:

```
ghci> :i SomeException
type SomeException :: *
data SomeException = forall e. Exception e => SomeException e
```

Скоро мы узнаем о некоторых распространенных типах исключений, а пока создадим свой собственный. Мы можем определить новый тип и вывести для него экземпляр класса типов `Exception`, а заодно класса `Show`. Для этого нам понадобится расширение языка `DeriveAnyClass`, которое подсказывает компилятору, как создавать экземпляры классов вроде `Exception` с пустым определением. Это работает, потому что у `Exception` нет даже минимального определения, так что `DeriveAnyClass` порождает пригодный для работы экземпляр:

```
ghci> :set -XDeriveAnyClass
ghci> data SomeFailure = SomeFailure String deriving (Show, Exception)
ghci> throw (SomeFailure "Whoops")
*** Exception: SomeFailure "Whoops"
```

Отлично! Мы умеем возбуждать исключения! Но тогда нужен и какой-то способ перехватывать и реагировать на них. Самая простая функция для перехвата исключения называется `catch`, она имеет тип `Exception e => IO a -> (e -> IO a) -> IO a`:

```
ghci> :t catch
catch :: Exception e => IO a -> (e -> IO a) -> IO a
ghci> throw (SomeFailure "Whoops") `catch` \(SomeFailure s)
➥ -> putStrLn ("Caught " ++ s)
Caught Whoops
```

Заметим, что явно мы перехватываем только исключения типа `SomeFailure` – и никакие другие. Это значит, что исключение, возбужденное функцией `error`, а также `undefined`, не будет перехвачено функцией `catch`, как, вообще говоря, и любое исключение, не принадлежащее типу `SomeFailure`:

```
ghci> catchSomeFailure act = act `catch` \(SomeFailure s)
➥ -> putStrLn ("Caught " ++ s)
ghci> catchSomeFailure (throw $ SomeFailure "Whoops")
Caught Whoops
ghci> catchSomeFailure (error "Whoops")
*** Exception: Whoops
CallStack (from HasCallStack):
  error, called at <interactive>:15:19 in interactive:Ghci6
ghci> catchSomeFailure undefined
*** Exception: Prelude.undefined
CallStack (from HasCallStack):
  undefined, called at <interactive>:16:18 in interactive:Ghci6
```

Однако можно перехватить *все* исключения, передав `catch` функцию обработки типа `SomeException` с навешенным квантором существования, который может содержать любое исключение. Обычно это не рекомендуется, потому что в большинстве случаев все исключения – слишком широкая категория, ведь исключения могут возникать по самым разным причинам – от прерывания пользователем до асинхронного потока управления. Позже мы познакомимся с некоторыми видами исключений.

> **ПРИМЕЧАНИЕ** Может показаться, что для перехвата нескольких типов исключений разумно использовать цепочку вызовов `catch` с разными типами. Однако ничего хорошего в этом нет: исключения, возбужденные обработчиком исключений, вызванным предшествующей `catch`, будут перехвачены последующей `catch`. Это значит, что мы перехватываем слишком много исключений: не только те, что могли возникнуть в интересующем нас выражении, но и те, что могут возбуждаться обработчиками исключений – в итоговом потоке управления будет трудно разобраться. Решение проблемы дает функция `catches` из модуля `Control.Exception`, которая позволяет определять несколько обработчиков для одного выражения.

Также при работе с исключениями полезна функция `try` типа `Exception e => IO a -> IO (Either e a)`. Она выполняет действие ввода-вывода и возвращает встретившееся исключение внутри конструктора `Left` типа `Either`. Если исключения не возникло, то мы получаем значение действия в конструкторе `Right`. Таким образом, мы можем пытаться выполнить небезопасные операции, потенциально

способные возбудить исключение, превратив их в более безопасные версии, которые кодируют исключения в своем типе возвращаемого значения.

Теперь мы можем обезопасить свою функцию `listFilesAndDirectories`, воспользовавшись `try`. Важно определить, исключения какого типа перехватывать. Поскольку `listFilesAndDirectories` – действие в монаде `IO`, ожидаемое исключение имеет тип `IOException`. Исключения такого типа *всегда* могут быть возбуждены монадой `IO`, он кодирует ошибки, относящиеся к файловой системе и сетям. Перехватывая такие исключения и кодируя их типом возвращаемого значения, мы создаем безопасную версию функции. Поэтому мы можем назвать ее именем, отражающим безопасность использования, как показано в листинге ниже.

Листинг 14.4 Безопасная функция перечисления, кодирующая исключения в типе возвращаемого значения

```
import Control.Exception

...

safeListFilesAndDirectories ::
  FilePath -> IO (Either IOException ([FilePath], [FilePath]))
safeListFilesAndDirectories = try .
➥ listFilesAndDirectories
```

Вызывает функцию listFilesAndDirectories и перехватывает исключения типа IOException

Важно понимать, что тип возвращаемого функцией значения определяет, какие виды исключения могут быть перехвачены `try`. Перехватываются только исключения типа `IOException`, а все остальные просто возбуждаются повторно.

Когда впоследствии мы будем использовать эту функцию в своей реализации, мы поместим ее вместе с другими в модуль, который содержит и экспортирует только безопасные функции для работы с файловой системой. И назовем этот модуль `System.Directory.Safe`.

14.4.1 Обработка ошибок

Мы написали функцию для перечисления файлов и каталогов. Еще нам нужна возможность копировать файлы из одного места в другое. Модуль `System.Directory` предоставляет два действия для копирования файла. Первое, `copyFile`, копирует файл, сохраняя права доступа к нему в операционной системе, но с потерей всех остальных метаданных. Второе, `copyFileWithMetadata`, копирует файл вместе со всеми метаданными. В своем приложении мы хотим сохранить метаданные, чтобы лучше сравнивать синхронизируемые файлы.

Ни то, ни другое действие не создает каталог, если он не существует, так что об этом придется позаботиться нам. Для этой цели

отлично подойдет действие `createDirectoryIfMissing`. Оно получает путь к файлу и булево значение, указывающее, должно ли действие создавать также родительские каталоги на пути к файлу. Пользуясь действиями `createDirectoryIfMissing` и `copyFileWithMetadata`, мы можем написать действие, которое создает конечный каталог (если он отсутствует) и копирует в него файл:

```
safeCopyFileAndCreateDir ::
  FilePath -> FilePath -> IO (Either IOException ())
safeCopyFileAndCreateDir src dst = try $ do
  Dir.createDirectoryIfMissing True $ takeDirectory dst
  Dir.copyFileWithMetadata src dst
```

Функция `takeDirectory` нужна, чтобы исключить имя файла из конечного пути. Действие замечательное, но есть одна проблема. Предположим, что приложение копирует большой файл и пользователь прервал его выполнение. В таком случае уже скопированный кусок останется. Не такого поведения вы ожидаете от инструмента синхронизации!

> **ПРИМЕЧАНИЕ** `copyFileWithMetadata` копирует файлы не атомарно, т. е. может случиться, что копирование будет частично выполнено, а затем произойдет отказ – и на диске останется поврежденный файл. Однако `copyFile` гарантирует, что файл либо скопирован целиком, либо не скопирован вовсе. Если вам нужен больший контроль над атомарностью при работе с файлами, поинтересуйтесь модулем `UnliftIO.IO.File` из пакета `unliftio`.

Мы можем обработать исключения, инициированные пользователем, перехватывая конкретный подтип так называемых *асинхронных исключений*. Это противоположность *синхронным исключениям*, которые код возбуждает в том же потоке, в котором работает сам. Асинхронные же исключения возбуждаются в другом потоке. Обычно они используются одним потоком программы, чтобы остановить выполнение другого потока – для этого асинхронное исключение возбуждается с помощью `throwTo`. Подробно эту тему мы обсуждать не будем, но затронуть ее в контексте обработки прерываний от пользователя придется, поскольку асинхронные прерывания может возбуждать и операционная система.

У типа `AsyncException` четыре конструктора:

- `StackOverflow` – возникает, когда переполняется стек потока;
- `HeapOverflow` – возникает, когда переполняется куча программы;
- `ThreadKilled` – возникает, когда поток должен быть остановлен (обычно возбуждается другими потоками);
- `UserInterrupt` – возникает, когда пользователь прерывает выполнение программы (например, нажав **Ctrl**+**C**).

Мы хотим перехватывать исключения, созданные конструктором `UserInterrupt`, и удалять частично скопированный файл. Для такого точного управления предназначены действия `catchJust` и `tryJust`, позволяющие решить, какое исключение перехватывать, с помощью предиката. `catchJust` имеет тип `Exception e => (e -> Maybe b) -> IO a -> (b -> IO a) -> IO a`; здесь первый аргумент – предикат, принимающий исключение, которое могло быть возбуждено действием во втором аргументе, он возвращает обернутое конструктором `Just` значение, которое передается обработчику исключения в третьем аргументе. Предикат может определить, какой тип исключения он получил, а затем вернуть результат в зависимости от содержащихся в исключении данных. Если предикат вернул истину, то обработчик должен что-то сделать с исключением.

Внутри предиката мы можем произвести сопоставление со значением ожидаемого типа `AsyncException` и проверить, создано ли оно конструктором `UserInterrupt`. Это показано в следующем листинге.

Листинг 14.5 Функция безопасного копирования файла с метаданными

```
safeCopyFileAndCreateDir ::
  FilePath -> FilePath -> IO (Either IOException ())
safeCopyFileAndCreateDir src dst = try $ do
Dir.createDirectoryIfMissing True $ takeDirectory dst
catchJust
  ( \(ex :: AsyncException) -> case ex of
      UserInterrupt -> Just ()
      _ -> Nothing
  )
  (Dir.copyFileWithMetadata src dst)
  (const $ Dir.removeFile dst >>
➥ throwIO UserInterrupt)
```

Перехватывает исключение типа IOException

Создает конечный каталог, если его еще не существует

Возвращает Just(), если было возбуждено исключение UserInterrupt

Копирует файл из исходного каталога в конечный с сохранением метаданных

Если было перехвачено исключение UserInterrupt, то удаляет частично скопированный файл, а затем заново возбуждает UserInterrupt

Перехватив исключение `UserInterrupt`, мы удаляем файл из конечного каталога, а затем повторно возбуждаем `UserInterrupt`, поскольку, возможно, захотим вообще прервать работу программы. Заметим, что для задания типа внутри предиката требуется расширение языка `ScopedTypeVariables`.

Функции, о которых мы говорили, могут находиться в вышеупомянутом модуле `System.Directory.Safe`. Чтобы не было сомнений по поводу его использования, начало модуля должно выглядеть следующим образом:

```
{-# LANGUAGE ScopedTypeVariables #-}

module System.Directory.Safe
```

```
  ( safeListFilesAndDirectories,
    safeCopyFileAndCreateDir,
)
where

import Control.Exception
import Control.Monad (filterM)
import qualified System.Directory as Dir
import System.FilePath (takeDirectory, (</>))
```

Располагая модулями `System.FileType` и `System.Directory.Safe`, мы теперь можем написать приложение. Но нужно еще подумать о задании конфигурации, состоянии и выводе с помощью нескольких действий ввода-вывода.

Резюме

- Тип `Handle` дает низкоуровневый доступ к файлам и может быть использован для чтения байтов из файла.
- Функция `bracket` позволяет безопасно захватить и освободить ресурс даже при наличии исключений.
- Как монадическая версия `map` называется `mapM`, так монадическая версия `filter` называется `filterM` и имеет тип `Applicative m =\> (a -> m Bool) -> [a] -> m [a]`.
- Исключения – это обычные типы, имеющие экземпляр класса типов `Exception`, их можно использовать для изменения потока управления в программе.
- Расширение языка `DeriveAnyClass` позволяет компилятору выводить экземпляры классов типов для произвольного типа, при условии что минимальный набор определений для экземпляра пуст.
- Исключения можно перехватывать с помощью функций `catch` или `try`, но если мы хотим перехватить сразу несколько исключений, то следует использовать функцию `catches`.
- Асинхронные исключения возбуждаются одним потоком в других потоках, чтобы изменить их поток управления.

15 Применение трансформеров для синхронизации

Краткое содержание этой главы:

- использование монадных трансформеров для расширения возможностей монад;
- структурирование приложения с помощью стека монадных трансформеров с допускающим только чтение окружением, журналом событий и состоянием;
- разбор аргументов командной строки приложения с помощью пакета `optparse-applicative`.

В предыдущей главе мы начали разрабатывать приложение для синхронизации каталогов. Мы рассмотрели чтение и запись файлов, а также обработку исключений. Теперь мы хотим воспользоваться этой функциональностью в приложении.

Мы хотим использовать *монадные трансформеры*, чтобы упростить эффектообразующие вычисления, которые имеют доступ к данным, допускающим только чтение, могут сами модифицировать состояние и собирать результаты вывода, а не только полагаться на монаду `IO`. Мы также научимся разбирать аргументы командной строки с помощью пакета `optparse-applicative`.

15.1 Монадные трансформеры

Сначала определим требования к нашему приложению синхронизации каталогов. Пользователь может задать один исходный и один конечный каталог, а также ограничить типы копируемых файлов. Кроме того, должна быть возможность наложить ограничение на глубину рекурсивной синхронизации и на количество скопированных файлов. В процессе синхронизации мы должны будем отслеживать состояние, чтобы не превысить заданные пределы.

Было бы хорошо получить отчет, содержащий список файлов, обработанных приложением, с информацией об успешности копирования каждого файла, потому что теоретически при копировании могли возникнуть исключения. Также можно игнорировать файлы, которые не изменились – ведь в этом и состоит смысл приложения.

Можно выделить три способа взаимодействия с данными. Заданная пользователем конфигурация – это данные, которые не изменяются, но доступны из любого места приложения. Можно считать, что это статическое окружение, к которому мы хотим иметь доступ для чтения. Информация о глубине рекурсии и уже синхронизированных файлах – это состояние, которым должна оперировать наша программа. Список файлов и совершенных над ними операций – это вывод, который мы хотим накапливать в процессе работы приложения. Для взаимодействия с данными такими разными способами мы будем использовать технику *монадных трансформеров*, помогающую создавать мощные абстракции.

15.1.1 Чтение окружения с помощью ReaderT

Сначала посмотрим, как обычно реализуется что-то типа конфигурации, допускающей чтение. Начнем с различных типов приложения в новом модуле `App.Types`. Именно там будет находиться тип конфигурационных данных.

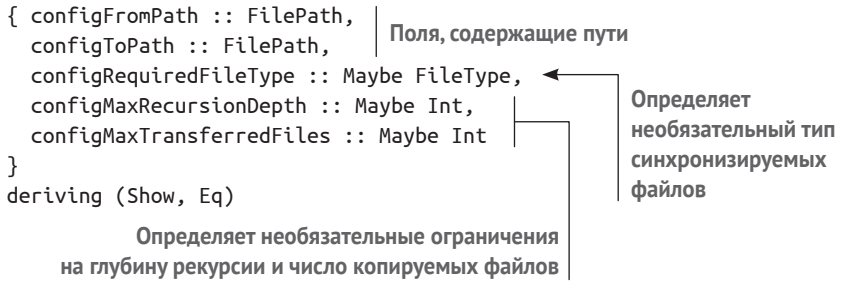
Как уже было сказано, в конфигурации нашего приложения должны присутствовать начальный и конечный каталоги, а также тип синхронизируемых файлов и ограничения на глубину рекурсии и число файлов. Тип такой конфигурации мог бы выглядеть, как показано в листинге ниже.

Листинг 15.1 Тип конфигурации в приложении для синхронизации файлов

```
module App.Types where

import System.FileType (FileType)  <-- Импортирует FileType

data Config = Config
```



Далее встает вопрос о том, как читать эту конфигурацию в наших действиях. Простое решение – передать `Config` в качестве аргумента, а затем использовать расширение языка `RecordWildCards` для доступа к полям конфигурации:

```
action :: Config -> ... -> IO ()
action cfg@(Config {..}) ... = ...
```

В этом действии мы можем обращаться ко всем полям конфигурации и передавать ее другим действиям в аргументе `cfg`. На первый взгляд, ничего плохого, но несколько громоздко. Мы должны задавать конфигурацию и ее тип в каждом действии, где хотим ее использовать, что ведет к многократному повторению кода.

Предпочтительнее было бы прочитать окружение один раз, а не задавать его для каждого действия. Такое поведение реализует монада `Reader`. Как следует из имени, `Reader` обладает способностью *читать из окружения*, но не изменять его.

Чтобы ей воспользоваться, добавим пакет `transformers` в файл `package.yaml` и импортируем модуль `Control.Monad.Trans.Reader`:

```
ghci> import Control.Monad.Trans.Reader
ghci> :{
ghci| withConstantAdded :: Int -> Reader Int Int
ghci| withConstantAdded x = do
ghci|   env <- ask
ghci|   return $ env + x
ghci| :}
ghci> runReader (withConstantAdded 1) 10
11
```

Тип `Reader r a` имеет две переменные типа: `r` – читаемое окружение, `a` – тип, возвращаемый в результате работы `Reader`. Функция `runReader` выполняет `Reader` с заданным окружением. В примере выше мы передаем ей просто `Int`, а с помощью `ask` можем прочитать окружение внутри монады `Reader`:

```
ghci> :{
ghci| addIfNotNeg :: Int -> Reader Int Int
ghci| addIfNotNeg x = do
```

```
ghci|   env <- ask
ghci|   if env < 0
ghci|     then return env
ghci|     else withConstantAdded x
ghci| :}
ghci> runReader (addIfNotNeg 10) 1
11
ghci> runReader (addIfNotNeg 10) (-1)
-1
```

Очевидное преимущество использования монады `Reader` в том, что нет необходимости передавать окружение из одного действия в другое, но возможность обращаться к нему при этом сохраняется. Однако монада `Reader` еще не идеальна. Она не обладает возможностями `IO`, поэтому воспользоваться ей для наших целей невозможно! А вот здорово было бы как-нибудь взять монаду `IO` и наделить ее способностями монады `Reader`, правда?

Именно для этого предназначен пакет `transformers`! Он определяет *монадные трансформеры*, которые помогают придать существующим монадам новые свойства. Один из них, `ReaderT`, позволяет сплавить `Reader` с другими монадами, например `IO`.

> **ПРИМЕЧАНИЕ** Имена монадных трансформеров обычно заканчиваются буквой `T`, поэтому, видя лишь тип, мы можем сказать, что он расширяет некоторую базовую монаду.

Тип `ReaderT r m a` имеет следующие переменные типа:

- `r` – окружение, к которому имеет доступ `ReaderT`;
- `m` – базовая монада `ReaderT`;
- `a` – результат выполнения `ReaderT`.

Если `m` удовлетворяет ограничению типа `Monad m`, то и весь тип `ReaderT` ему удовлетворяет. Поэтому `ReaderT Config IO a` – монада, для которой базовой является `IO` и которая имеет доступ к значению `Config` в качестве окружения. Используя действие `ask` (имеющее тип `Monad m => ReaderT r m r`), мы можем опросить окружение:

```
ghci> import Control.Monad.Trans.Reader
ghci> type App a = ReaderT Config IO a
ghci> import Control.Monad.IO.Class (liftIO)
ghci> :{
ghci| example :: App ()
ghci| example = do
ghci|   env <- ask
ghci|   let toPath = configToPath env
ghci|       fromPath = configFromPath env
ghci|    in liftIO $ putStrLn (fromPath <> " -> " <> toPath)
ghci| :}
```

```
ghci> cfg = Config "src" "dst" Nothing Nothing Nothing
ghci> runReaderT example cfg
src -> dst
```

В этом примере определен новый тип App, представляющий наше приложение. Действие example читает пути из заданного Config и печатает их. Отметим, что здесь мы вынуждены использовать liftIO, потому что putStrLn – действие ввода-вывода, но мы-то находимся внутри App, а не IO. Если мы хотим использовать действия ввода-вывода внутри App, то должны *поднять* их в действие App, что и делает функция liftIO, определенная в классе типов MonadIO:

```
ghci> import Control.Monad.IO.Class
ghci> :i MonadIO
type MonadIO :: (* -> *) -> Constraint
class Monad m => MonadIO m where
  liftIO :: IO a -> m a
  {-# MINIMAL liftIO #-}
```

Монады, реализующие экземпляр MonadIO, могут выполнять любое действие IO с помощью liftIO. ReaderT реализует MonadIO, если базовая монада реализует MonadIO и IO реализует MonadIO.

Но что, если базовая монада – не IO? Сможем ли мы поднять действия из нее в трансформер? Оказывается, что эта концепция принципиально присуща монадным трансформерам и кодифицирована в классе типов MonadTrans:

```
ghci> import Control.Monad.Trans.Class
ghci> :i MonadTrans
type MonadTrans :: ((* -> *) -> * -> *) -> Constraint
class MonadTrans t where
  lift :: Monad m => m a -> t m a
  {-# MINIMAL lift #-}
```

Как видим, функция lift принимает монаду и *поднимает* ее в контекст трансформера (обозначенный t).

Таким образом, ReaderT позволяет прочитать Config в наших действиях ввода-вывода, но как насчет обработки состояния внутри этих действий? Существует ли трансформер для этого?

15.1.2 StateT и WriterT

Да, трансформер для обработки состояния существует и называется StateT. Он работает очень похоже на ReaderT. Тип StateT s m a имеет три переменные типа:

- s – состояние, которым управляет StateT;
- m – базовая монада StateT;
- a – результат выполнения StateT.

`StateT` ведет себя похоже в определениях экземпляров `Monad` и `MonadIO`. Различие между `ReaderT` и `StateT` заключается в основном в функциях, которые можно использовать для доступа к управляемым данным. Если `ReaderT` располагает функцией `ask`, то `StateT` предоставляет три разные функции:

- `get :: Monad m => StateT s m s` – получает текущее состояние монады `StateT`;
- `put :: Monad m => s -> StateT s m ()` – устанавливает новое значение в качестве состояния монады `StateT`;
- `modify :: Monad m => (s -> s) -> StateT s m ()` – модифицирует существующее состояние монады `StateT`.

Есть еще несколько функций, но эти три самые важные. С их помощью мы можем беспрепятственно модифицировать состояние внутри произвольных монад. Разумеется, в нашем случае используется монада `IO`:

```
ghci> import Control.Monad
ghci> import Control.Monad.Trans.State.Strict
ghci> import Control.Monad.IO.Class (liftIO)
ghci> import System.Directory.Safe
ghci> import System.FilePath
ghci> type App a = StateT Int IO a
ghci> :{
ghci| example :: FilePath -> App ()
ghci| example path = do
ghci|   mFilesAndDirs <-
ghci|     liftIO $ safeListFilesAndDirectories path
ghci|   case mFilesAndDirs of
ghci|     Left _ -> return ()
ghci|     Right (_, dirs) -> do
ghci|       modify (\x -> x + (length dirs))
ghci|       if null dirs
ghci|         then return ()
ghci|         else forM_ (map (path </>) dirs) example
ghci| :}
ghci> execStateT (example "foo") 0
2
```

Теперь возникает вопрос, как должно выглядеть состояние нашего приложения. Мы уже упоминали, что должны отслеживать глубину рекурсии и количество уже скопированных файлов. Ниже показан подходящий для этого тип.

Листинг 15.2 Тип и вспомогательные функции для статистики времени выполнения

```
data Stats = Stats                    <-- Тип для статистики времени выполнения
  { statsRecursionDepth :: Int,
    statsTransferredFiles :: Int
```

```
  }
  deriving (Show, Eq)

modifyRecursionDepth :: (Int -> Int) -> Stats -> Stats
modifyRecursionDepth f (Stats recDepth files) =
  Stats (f recDepth) files

modifyTransferredFiles :: (Int -> Int) -> Stats -> Stats
modifyTransferredFiles f (Stats recDepth files) =
  Stats recDepth (f files)
```

Функции для модификации конкретных полей типа Stats

Дополнительно мы определяем функции модификации различных полей, в основном чтобы упростить работу с функцией `modify`. Поскольку позже мы будем запускать приложение с пустым состоянием, было бы хорошо определить `mempty`, поэтому мы можем реализовать также экземпляры `Semigroup` и `Monoid` для этого типа. Они показаны в листинге ниже.

Листинг 15.3 Экземпляры `Semigroup` и `Monoid` для типа `Stats`

```
instance Semigroup Stats where
  a <> b =
    Stats
      { statsRecursionDepth = max `on`
➥ statsRecursionDepth,
        statsTransferredFiles = (+) `on`
➥ statsTransferredFiles
      }
    where
      f `on` field = f (field a) (field b)

instance Monoid Stats where
  mappend = (<>)
  mempty =
    Stats
      { statsRecursionDepth = 0,
        statsTransferredFiles = 0
      }
```

Вычисляет максимальную из двух глубин рекурсии

Вычисляет количество скопированных файлов

Определяет вспомогательную функцию для применения функции к заданному полю для обоих аргументов

Использует ассоциативную функцию из экземпляра Semigroup в качестве mappend

Определяет пустое значение Stats

Итак, состояние нашего приложения определено. Последний кусочек пазла – отчет о том, что происходило с файлами в процессе их обработки. Нашей монаде нужно как-то собирать данные для этого отчета «в сторонке», одновременно с основными вычислениями.

Вы могли бы предположить, что это тоже можно сделать посредством вышеупомянутого состояния, и технически это будет правильно. Однако имеется еще один трансформер, позволяющий сделать это гораздо элегантнее: `WriterT`. Он позволяет аккумулировать вывод во время выполнения других эффектообразующих вычислений. Тип `WriterT w m a` состоит из следующих типов:

- `w` – вывод, собираемый `WriterT`;
- `m` – базовая монада `WriterT`;
- `a` – результат выполнения `WriterT`.

Чтобы это работало, вывод обычно нуждается в экземпляре `Monoid`, потому что различные части вывода собираются по одной и объединяются с помощью функции `mappend`, рассмотренной в главе 7:

```
ghci> import System.Directory.Safe
ghci> import Control.Monad
ghci> import Control.Monad.IO.Class (liftIO)
ghci> import Control.Monad.Trans.Writer.Strict
ghci> type App a = WriterT [FilePath] IO a
ghci> :{
ghci| example path = do
ghci|   mFilesAndDirs <- liftIO $ safeListFilesAndDirectories path
ghci|   case mFilesAndDirs of
ghci|     Left _ -> return ()
ghci|     Right (_, dirs) -> do
ghci|       tell dirs
ghci|       if null dirs
ghci|         then return ()
ghci|         else forM_ (map (path </>) dirs) example
ghci| :}
ghci> runWriterT (example "foo")
((),["baz","bar"])
```

Этот тип предоставляет функцию `tell` типа `Monad m => w -> WriterT w m ()`, которая добавляет вывод в аккумулятор. Теперь можно подумать о том, каким должен быть тип отчета. В общем случае синхронизация файлов может завершиться успешно, неудачно или быть необязательной (потому что файл уже синхронизирован). Поэтому журнал должен содержать три разных типа с соответствующими параметрами.

Листинг 15.4 Определение записей отчета в приложении синхронизации файлов

```
import Control.Exception

...

data SuccessEntry = SuccessEntry
  { successFromPath :: FilePath,
    successToPath :: FilePath
  }
  deriving (Show, Eq)

data FailureEntry = FailureEntry
  { failureFilePath :: FilePath,
```

Тип для успешно синхронизированного файла

Тип для неудачной синхронизации

```
    failureException :: IOException
  }
  deriving (Show, Eq)

data SkipEntry = SkipEntry
  { skipFilePath :: FilePath,
    skipReason :: String
  }
  deriving (Show, Eq)

data LogEntry
  = Success SuccessEntry
  | Failure FailureEntry
  | Skipped SkipEntry
  deriving (Show, Eq)
```

Тип для пропущенного файла

Тип записи отчета

Как видим, даже исключения, в частности `IOException`, – просто данные, которые можно хранить в виде значений и передавать из одного места в другое. Итак, мы можем определить простой тип журнала с помощью ключевого слова `newtype`:

```
newtype Log = Log [LogEntry]
```

Однако теперь мы должны вручную определить экземпляры `Semigroup` и `Monoid` для нового типа, несмотря на то что в типе списка они уже определены. Определение функций этих экземпляров будет заключаться в разворачивании типа `Log`, вызове соответствующих функций и последующем обертывании их конструктором `Log`. По счастью, Haskell уже предлагает метод автоматического вывода таких экземпляров из определений `newtype`. Для этого нам понадобятся два расширения языка:

```
{-# LANGUAGE DerivingStrategies #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
```

Первое позволяет сообщить Haskell, как выводить классы типов из типа путем указания *стратегии вывода* после слова `deriving`, например:

```
ghci> :set -XDerivingStrategies
ghci> newtype Foo = Foo String deriving newtype Show
ghci> show $ Foo "Hey, where did Foo go?"
"\"Hey, where did Foo go?\""
```

Задав стратегию `newtype` в конструкции `deriving newtype`, мы просим компилятор использовать экземпляр базового типа `newtype` для конструирования нового экземпляра. В нашем примере для получения экземпляра `Show Foo` используется экземпляр `Show` для типа `String`. Поэтому конструктор не показан. Если мы хотим использовать стратегию вывода по умолчанию, то можем либо вообще не ука-

зывать ее, либо указать стратегию `stock`, чтобы было понятно, что нам нужно поведение, предоставляемое Haskell по умолчанию.

> **ПРИМЕЧАНИЕ** Haskell также предоставляет стратегии вывода `anyclass` и `via`, которые используются в сочетании с расширениями языка `DeriveAnyClass` и `DerivingVia` для классов, имеющих пустое минимальное определение, и для вывода экземпляров класса для типов с одинаковыми представлениями времени выполнения.

По умолчанию Haskell умеет выводить только экземпляры `Eq`, `Ord`, `Enum` и `Bounded` с помощью стратегии `newtype`. Чтобы компилятор мог вывести больше экземпляров, необходимо расширение языка `GeneralizedNewtypeDeriving`. С его помощью мы, наконец, сможем вывести экземпляры `Semigroup` и `Monoid` для нашего типа `Log`, как показано в листинге ниже.

Листинг 15.5 Определение типа журнала, которое можно использовать совместно с трансформером `WriteT`

```
newtype Log = Log [LogEntry]
  deriving stock (Show, Eq)
  deriving newtype (Semigroup, Monoid)
```

Определяет тип Log как newtype для [LogEntry]

Определяет экземпляры для Show и Eq, используя стратегию вывода по умолчанию

Определяет экземпляры Semigroup и Monoid, используя экземпляры для базового типа списка

Упражнение. Использование модуля Sequence для повышения производительности

Все это выглядит прекрасно, но, увы, в наше приложение вкралась проблема производительности. Мы используем действие `tell`, т. к. знаем, что какие-то данные добавляются в аккумулируемый `WriterT` вывод с помощью `mappend`, которая для обычного списка совпадает с функцией дописывания `(++)`. Поскольку мы хотим добавлять элементы по одному в конец ранее собранных результатов, приходится каждый раз обходить весь список, так что для аккумулирования `n` элементов вывода нужно будет обойти список `n` раз, что ведет к квадратичному времени выполнения. Поэтому простой список не подходит для больших объемов данных. В качестве альтернативы можно использовать тип `Seq` из модуля `Data.Sequence`, входящего в состав пакета `containers`. Он поддерживает дописывание значений в конец последовательности за постоянное время, что избавляет нас от квадратичного роста. Далее в этой главе мы продолжим использовать списки, но вы все же попробуйте тип `Seq`. (Указание: много кода менять не придется.)

Теперь мы можем использовать функции из обоих классов типов для аккумулирования вывода в нашем трансформере `WriterT`!

15.1.3 Организация стека трансформеров с помощью RWST

Как мы видели, `ReaderT` можно использовать, чтобы дать монаде возможность читать из окружения, `StateT` – для управления состоянием, к которому можно обратиться внутри монады, а `WriterT` – чтобы наделить монаду способностью аккумулировать вывод в процессе других вычислений. Однако мы хотим объединить всю эту функциональность в одной монаде. Возможно ли это?

Разумеется – впрочем, мы уже к этому привыкли. Поскольку монадный трансформер преобразует `Monad` и порождает тоже `Monad`, мы можем сочетать несколько трансформеров. Такую организацию часто называют *стеком трансформеров*:

```
ghci> import Control.Monad.Trans.Class (lift)
ghci> import Control.Monad.Trans.Reader
ghci> import Control.Monad.Trans.Writer
ghci> type Foo a = ReaderT Int (WriterT String IO) a
ghci> foo = (ask >>= lift . tell . show) :: Foo ()
ghci> runWriterT (runReaderT foo 1)
((),"1")
```

Как видим, чтобы использовать `tell` из `WriterT`, мы должны сначала *поднять* его в монаду `ReaderT`. Теперь представим себе стек всех трех описанных выше трансформеров. Тогда мы должны были бы использовать `lift` дважды при использовании действий из самой внутренней монады.

По счастью, такое использование трех трансформеров встречается довольно часто, поэтому пакет `transformers` предлагает трансформер `RWST`, который представляет собой комбинацию всех трех и не опирается на стек трансформеров.

> **ПРИМЕЧАНИЕ** Рассмотренные нами монадные трансформеры очень широко распространены. Но есть и много других, а в реальном коде нередко можно встретить сложные стеки трансформеров.

Определение `RWST r w s m a` является комбинацией рассмотренных выше трансформеров:

- `r` – окружение, к которому имеет доступ `RWST`;
- `w` – вывод, аккумулируемый `RWST`;
- `s` – состояние, которым управляет `RWST`;
- `m` – базовая монада `RWST`;
- `a` – результат выполнения `RWST`.

Итак, в качестве типа для нашего приложения можно взять просто RWST, используя в нем типы, с которыми мы хотели работать в стеке трансформеров. Поскольку мы имеем дело с файловой системой, базовой монадой является IO. Окончательный тип показан в листинге ниже.

Листинг 15.6 Определение монады приложения

```
import Control.Monad.Trans.RWS.Strict  <-- Импортирует определение RWST

...

type App a = RWST Config Log Stats IO a  <-- Определяет тип приложения в терминах RWST
```

Заметим, что поскольку RWST сам является монадой, мы можем использовать App точно так же, как другие трансформеры до этого момента. Определившись с типом, можно приступать к реализации логики приложения.

15.2 Реализация приложения

Код нашего приложения будет находиться в отдельном модуле App.Impl. Если предыдущий модуль предоставлял определения типов, то новый будет содержать саму реализацию. Он начинается следующими предложениями импорта:

```
import App.Types
import Control.Exception (SomeException, catch)
import Control.Monad (forM_, when)
import Control.Monad.IO.Class
import Control.Monad.Trans.RWS.Strict
import qualified System.Directory as Dir
import qualified System.Directory.Safe as Dir
import System.Exit (ExitCode (ExitFailure), exitWith)
import System.FilePath ((</>))
import System.FileType (determineFileType)
import System.IO (hPutStrLn, stderr)
```

Заметим, что модули System.Directory и System.Directory.Safe импортируются с одинаковыми квалифицированными именами. Это позволяет употреблять одно и то же имя Dir в ссылках на символы из обоих модулей.

Но начнем по порядку: как мы можем выполнить приложение. Поскольку App представляет собой RWST, мы можем использовать функцию runRWST типа RWST r w s m a -> r -> s -> m (a, s, w). Она принимает RWST, окружение, из которого предстоит читать, и состояние, а возвращает результат, состояние и вывод, аккумулированный ба-

зовой монадой. Давайте ей воспользуемся! Однако что такое в нашем случае результат `a`? Поскольку наши действия не вычисляют полезный результат (а служат только для порождения побочных эффектов в виде копирования файлов), мы можем принять в качестве `a` единичный тип `()`. Итак, мы уже знаем, какие типы передать `runRWST`. Поскольку `Stats`, наш тип состояния, имеет экземпляр класса `Monoid`, мы можем использовать `mempty` для задания начального состояния. Окружение (наш тип `Config`) нужно будет передать извне. Зная все это, мы можем написать вспомогательную функцию, которая будет запускать `App` с заданным `Config` и возвращать `Log` в монаде `IO`. Полный код показан в листинге ниже.

Листинг 15.7 Функция для выполнения `App` в монаде `IO`

```
runApp :: App () -> Config -> IO Log
runApp app config = do
  (_, _, syncLog) <-
    runRWST app config mempty
      `catch` exceptionHandler
  return syncLog
  where
    exceptionHandler :: SomeException -> IO a
    exceptionHandler ex = do
      hPutStrLn stderr $
        "Syncing interrupted by exception: "
          ++ show ex
      exitWith $ ExitFailure 1
```

Выполняет App с пустым состоянием и заданной конфигурацией

Перехватывает все исключения

Возвращает аккумулированный Log

Печатает исключение на stderr

Завершает программу с кодом 1, индицирующим ошибку

В коде мы перехватываем исключения и в случае ошибки выходим из программы и выводим сообщение на `stderr`, чтобы пользователь понимал, что происходило в момент аварийного завершения.

Теперь время подумать о действиях, которые должен выполнять тип `App`:

- проверять, нуждаются ли два файла в синхронизации;
- проверять тип файла;
- копировать файл из одного каталога в другой, следя за количеством уже скопированных файлов;
- обходить подкаталоги исходного каталога, следя за глубиной рекурсии.

Будем реализовывать все по порядку. Файлы нуждаются в синхронизации, если конечного файла не существует или у него другое время модификации. Для получения времени модификации файла предназначено действие `getModificationTime` из модуля `System.Directory`. Поскольку это действие ввода-вывода, мы должны прибегнуть к `liftIO`, чтобы его можно было использовать в монаде `App`. То же самое относится к действию `doesFileExist`, которое

позволяет узнать, существует ли конечный файл. Код приведен в следующем листинге.

Листинг 15.8 Действие, проверяющее, одинаково ли время модификации у двух файлов

```
data FileDiff
  = Matching
  | ModificationTimeDiff
  | DestinationMissing
  deriving (Show, Eq)
```

Тип, определяющий, различаются ли два файла, и если да, то чем именно

```
checkFileDiff :: FilePath -> FilePath -> App FileDiff
checkFileDiff src dst = do
  dstExists <- liftIO $ Dir.doesFileExist dst
  if not dstExists
    then return DestinationMissing
    else do
      srcModTime <- liftIO $ Dir.getModificationTime src
      dstModTime <- liftIO $ Dir.getModificationTime dst
      if srcModTime == dstModTime
        then return Matching
        else return ModificationTimeDiff
```

Проверяет существование конечного файла

Проверяет, отличается ли время модификации файлов

С помощью этого действия мы можем проверить, нужно ли синхронизировать файлы.

> **СОВЕТ** Просто сравнить время модификации недостаточно для уверенности в идентичности двух файлов. Теоретически надо бы еще проверить, что совпадают размеры файлов, и дополнительно вычислить контрольные суммы, чтобы гарантировать неизменность содержимого. Но для простоты мы решили ограничиться проверкой времени модификации.

Еще следует проверить, удовлетворяет ли тип исходного файла ограничению, заданному в `Config`. Сначала нужно прочитать поле `configRequiredFileType` конфигурации `Config` нашего приложения. Мы уже видели функцию `ask` для опроса окружения, но есть более простой способ прочитать только одно поле из типа: функция `asks`, имеющая тип `(Monoid w, Monad m) => (r -> a) -> RWST r w s m a`. В развернутой форме код опроса выглядит так:

```
config <- ask
let mRequiredFileType = configRequiredFileType config
```

Его можно записать короче:

```
mRequiredFileType <- configRequiredFileType <$> ask
```

И преобразовать к виду:

```
mRequiredFileType <- asks configRequiredFileType
```

Существует также двойственная функция, которая проделывает с `get` то же, что `asks` с `ask`, т. е. позволяет читать поля по одному из состояния нашего приложения `App`. Используя `asks`, мы можем записать функцию, которая проверяет, совпадает ли тип файла с указанным при запуске приложения. См. код ниже.

Листинг 15.9 Действие, проверяющее, совпадает ли тип файла с заданным в конфигурации (если он был задан)

```
checkFileType :: FilePath -> App Bool
checkFileType path = do
  mRequiredFileType <- asks configRequiredFileType
  case mRequiredFileType of
    Just requiredFileType -> do
      fileType <- liftIO $ determineFileType path
      return (requiredFileType == fileType)
    Nothing -> return True
```

Читает поле configRequiredFileType из Config

Проверяет, совпадает ли тип указанного файла с требуемым

Определять, нуждаются ли файлы в синхронизации, мы научились, теперь требуется написать действие, которое синхронизирует файлы. Нам нужно регистрировать в журнале успехи, неудачи и причины пропуска синхронизации, поэтому сначала напишем несколько вспомогательных функций для добавления записей разного типа в журнал:

```
logSingleEntry :: LogEntry -> App ()
logSingleEntry x = tell $ Log [x]

logFailure :: FailureEntry -> App ()
logFailure = logSingleEntry . Failure

logSuccess :: SuccessEntry -> App ()
logSuccess = logSingleEntry . Success

logSkipped :: SkipEntry -> App ()
logSkipped = logSingleEntry . Skipped
```

Наше действие синхронизации должно сначала проверить, сколько файлов уже было скопировано и каково максимально разрешенное количество копируемых файлов. После этого оно должно сравнить тип файла с разрешенным и поместить в журнал соответствующую запись. Далее оно копирует файл с помощью действия `safeCopyFileAndCreateDir`, протоколирует успех или неудачу и увеличивает количество скопированных файлов. Все это показано в листинге ниже.

Листинг 15.10 Действие синхронизации двух файлов

```
syncFile :: FilePath -> FilePath -> App ()
syncFile src dst = do
  curFiles <- gets statsTransferredFiles
  mMaxFiles <- asks configMaxTransferredFiles
  when (maybe True (curFiles <) mMaxFiles) $ do
    fileTypeCorrect <- checkFileType src
    if not fileTypeCorrect
      then
        logSkipped $
          SkipEntry dst "Required filetype did not match"
      else do
        filesMatch <- checkFileDiff src dst
        if filesMatch == Matching
          then logSkipped $ SkipEntry dst "Files matched"
          else do
            mDidCopy <-
              liftIO $
➥ Dir.safeCopyFileAndCreateDir src dst
            case mDidCopy of
              Left ex -> logFailure $
➥ FailureEntry dst ex
              Right () -> do
                modify (modifyTransferredFiles (+
                logSuccess $ SuccessEntry src dst
```

Читает текущее состояние и окружение

Проверяет, достигнут ли предел копируемых файлов

Проверяет тип копируемого файла

Проверяет, различаются ли два файла и нужна ли синхронизация

Безопасно копирует исходный файл

Протоколирует исключение в случае, если оно возникло при копировании

Увеличивает количество скопированных файлов

Протоколирует факт успешной синхронизации файлов

Здесь мы видим функцию `when` из модуля `Control.Monad`, которая позволяет выполнять действие, только если соблюдено некоторое условие. В нашем действии смысл ее в том, чтобы производить дальнейшую проверку и копирование, только если количество уже скопированных файлов меньше максимального или максимальное количество не было задано. Если продолжать можно, то мы вызываем действия `checkFileType` и `checkFileDiff` и протоколируем результаты. Если все проверки прошли успешно, то мы копируем файл, увеличиваем количество скопированных файлов, вызывая `modify (modifyTransferredFiles (+ 1))`, и протоколируем успех.

Остался последний кусочек: обход каталога. Он похож на `syncFile` тем, что сначала проверяется предикат (достигнута ли максимальная глубина рекурсии), а затем, если работу можно продолжать, протоколируется успех, модифицируется состояние и рекурсивно обходятся подкаталоги. Код показан в листинге ниже.

Листинг 15.11 Действие обхода каталога и синхронизации обнаруженных файлов

```
traverseAndCopy :: FilePath -> App ()
traverseAndCopy pathSuffix = do
  fromPath <- asks configFromPath
  toPath <- asks configToPath
  mMaxRecDepth <- asks configMaxRecursionDepth
  curRecDepth <- gets statsRecursionDepth
  let path = fromPath </> pathSuffix
  when (maybe True (curRecDepth <=) mMaxRecDepth) $ do
    mFilesAndDirs <-
      liftIO $ Dir.safeListFilesAndDirectories path
    case mFilesAndDirs of
      Left ex -> logFailure $ FailureEntry path ex
      Right (files, dirs) -> do
        forM_ files $ \fileName ->
          syncFile
            (fromPath </> pathSuffix </> fileName)
            (toPath </> pathSuffix </> fileName)
        modify (modifyRecursionDepth (+ 1))
        forM_ dirs $ traverseAndCopy . (pathSuffix </>)
        modify (modifyRecursionDepth (\x -> x - 1))
```

Читает конфигурацию и состояние

Проверяет, достигнута ли максимальная глубина рекурсии

Перечисляет все файлы и подкаталоги в текущем каталоге

Протоколирует ошибку в случае исключения

Синхронизирует файлы

Увеличивает текущую глубину рекурсии

Рекурсивно вызывает traverseAndCopy для всех подкаталогов

Уменьшает текущую глубину рекурсии

Действие получает суффикс текущего пути в качестве аргумента. Глубина рекурсии изменяется до и после рекурсивного вызова `traverseAndCopy`.

Итак, все действия на месте, и мы можем определить главное приложение, которое получает пути из окружения. Действие `traverseAndCopy` параметризовано только суффиксом пути, который дописывается в конец `configFromPath`. Поэтому в самом начале путь должен быть пустым. Это подводит нас к окончательному определению в модуле `App.Impl`.

Листинг 15.12 Определение главного приложения

```
mainApp :: App ()
mainApp = traverseAndCopy ""
```

Наша программа будет состоять из единственного обращения к `runApp` с помощью `mainApp`. Единственный аргумент `runApp` – это `Config`. Мы хотим получать конфигурацию из командной строки, поэтому должны написать парсер параметров, заданных пользователем. На этот раз мы воспользуемся уже готовым парсером из пакета `optparse-applicative`.

15.3 Организация интерфейса с командной строкой

Программа синхронизации файлов почти закончена. Осталось только реализовать способ задания конфигурации `App` пользователем. В предыдущих главах мы часто прибегали к написанию собственных функций разбора аргументов программы, но теперь обратимся к решению, предоставляемому пакетом `optparse-applicative`.

Как всегда, добавим его в секцию `dependencies` файла `package.yaml`:

```
dependencies:
- base >= 4.7 && < 5
- bytestring
- directory
- filepath
- transformers
- optparse-applicative
```

И посмотрим, что этот пакет умеет.

15.3.1 Разбор аргументов с помощью optparse-applicative

Пакет `optparse-applicative` предоставляет парсеры аргументов командной строки, написанные в *аппликативном* стиле. Также он автоматически генерирует справку для приложения, справляется с аргументами различных типов и проверяет, что все обязательные аргументы присутствуют.

Рассмотрим пример парсера аргументов. Будем предполагать, что для представления аргументов, заданных пользователем, используется следующий тип:

```
data Args = Args
  { arg1 :: String,
    arg2 :: Int,
    arg3 :: Bool,
    arg4 :: Maybe String
  }
  deriving (Show)
```

Мы хотим разбирать аргументы `arg1` и `arg2` как строку и целое число соответственно. `arg3` можно представить булевым флагом, а аргумент `arg4` факультативный. Посмотрим, как выглядит парсер для этого типа.

Листинг 15.13 Пример парсера аргументов

```
import Options.Applicative

...

argsParser :: Parser Args
argsParser = Args <$> strArg <*> intArg <*>
➥ boolArg <*> maybeArg
  where
    strArg =
      strOption $
        long "string"
          <> short 's'
          <> metavar "STRING"
          <> help "An example string"
    intArg =
      option auto $
        long "integer"
          <> short 'i'
          <> metavar "INTEGER"
          <> help "An example integer"
    boolArg =
      switch $
        long "boolean"
          <> short 'b'
          <> help "An example boolean switch"
    maybeArg =
      optional $
        strOption $
          long "optional"
            <> short 'o'
            <> metavar "OPTIONAL"
            <> help "An example optional string"
```

Конструирует тип результата в аппликативном стиле

Определяет парсер строкового аргумента

Определяет парсер целого аргумента

Определяет парсер булева переключателя

Определяет парсер факультативного строкового аргумента

Наш парсер пользуется типом `Parser` из пакета. Для каждого аргумента имеется свой парсер, который мы определяем отдельно. `strOption` обозначает строковый аргумент, а `option auto` можно использовать для определения парсера любого типа, для которого существует экземпляр `Read`.

> **ПРИМЕЧАНИЕ** Желая разобрать значение типа `String`, мы должны использовать парсер `strOption` (или `option str`), а не `option auto`, т. к. экземпляр `Read String` требует кавычек вокруг значения, поэтому `ABC` не удастся разобрать, а `"ABC"` удастся. `strOption` решает эту проблему.

Парсер для разбора факультативных аргументов создается функцией `optional`. В параметрах `long` и `short` мы передаем парсеру длинное и короткое имена аргумента. Параметры `metavar` и `help` служат для генерирования справки. Обратите внимание, как эти функции используются для построения описания каждого аргумента в *моноидном* стиле с использованием `(<>)`. В данном случае порядок параметров неважен и большая их часть факультативна. Обязательно только задать `short` или `long`, потому что иначе на аргумент невозможно будет сослаться.

Вышеупомянутый *аппликативный* стиль можно видеть в определении парсера, когда фактические значения типа `Args` строятся на основе результатов разбора. Мы познакомились с этим стилем в главе 12 при изучении того, как парсеры можно определять в терминах аппликативных функторов и монад.

Определив тип `Parser`, мы должны определить `ParserInfo`, который затем сможем выполнить. `ParserInfo` объединяет парсер с дополнительной информацией, которая, как и информация об аргументах, нужна для генерирования текста справки.

Листинг 15.14 Пример `ParserInfo`

```
argumentParser :: ParserInfo Args
argumentParser =
  info
  ( argsParser
      <**> helper
      <**> simpleVersioner "v1.0"
  )
  ( fullDesc
      <> progDesc "This is an example"
      <> header "OPTPARSE APPLICATIVE EXAMPLE"
      <> footer "Thanks for checking this example out!"
  )
```

Создает ParserInfo из Parser и дополнительной информации

Добавляет парсер, который разбирает аргумент --help и отображает текст справки

Добавляет парсер, который разбирает аргумент --version и отображает номер версии

Предоставляет дополнительную информацию для текста справки

Как и предыдущие, параметры `progDesc`, `header` и `footer` факультативны. Здесь мы объединяем наш `argsParser` с дополнительным парсером `helper`, который позволяет задать аргумент `--help` для вывода полного текста справки, и парсером `simpleVersioner`, который разбирает аргумент `--version` и отображает номер версии.

Сконструировав значение типа `ParserInfo`, мы можем выполнить его в монаде `IO`:

```
main :: IO ()
main = do
  args <- execParser argumentParser
  print args
```

Посмотрим, как реагирует программа:

```
shell $ stack run
Missing: (-s|--string STRING) (-i|--integer INTEGER)

Usage: optparse-exe (-s|--string STRING) (-i|--integer INTEGER) [-b|--boolean]
                    [-o|--optional OPTIONAL]

  This is an example
```

Как видим, парсер правильно определил, что два обязательных аргумента отсутствуют. Также печатается ранее определенная строка `progDesc` вместе с именем программы – в данном случае `optparse-exe` (это имя присвоила программа `stack`). Теперь зададим обязательные аргументы:

```
shell $ stack run -- -s "Foo" -i 10
Args {arg1 = "Foo", arg2 = 10, arg3 = False, arg4 = Nothing}
```

Как видим, аргумент `arg3` необязателен, потому что если он опущен, то равен `False`:

```
shell $ stack run -- -s "Foo" -i 10 -b -o "Bar"
Args {arg1 = "Foo", arg2 = 10, arg3 = True, arg4 = Just "Bar"}
```

В пакете `optparse-applicative` выбран *аппликативный* стиль разбора по двум причинам:

- текст справки автоматически генерируется парсером, какой бы сложной ни оказалась логика разбора;
- порядок задания аргументов неважен.

Посмотрим, как выглядит справка:

```
shell $ stack run -- --help
OPTPARSE APPLICATIVE EXAMPLE

Usage: optparse-exe (-s|--string STRING) (-i|--integer INTEGER) [-b|--boolean]
                    [-o|--optional OPTIONAL]

  This is an example

Available options:
  -s,--string STRING        An example string
  -i,--integer INTEGER      An example integer
  -b,--boolean              An example boolean switch
  -o,--optional OPTIONAL    An example optional string
  -h,--help                 Show this help text
  --version                 Show version information

Thanks for checking this example out!
```

И снова посмотрим на порядок аргументов:

```
shell $ stack run -- -o "Bar" -i 10 -b -s "Foo"
Args {arg1 = "Foo", arg2 = 10, arg3 = True, arg4 = Just "Bar"}
```

Пакет предлагает еще много способов построения интерфейсов командной строки, например парсеры команд с собственными аргументами. Но для краткости мы опустим обсуждение этих возможностей.

15.3.2 Перечисление пользовательских типов

Теперь, познакомившись с основами, мы можем написать парсер аргументов, который будет строить определенный ранее тип `Config`. Напомним, как этот тип выглядит:

```
data Config = Config
  { configFromPath :: FilePath,
    configToPath :: FilePath,
    configRequiredFileType :: Maybe FileType,
    configMaxRecursionDepth :: Maybe Int,
    configMaxTransferredFiles :: Maybe Int
  }
  deriving (Show, Eq)
```

Мы можем определить, что оба аргумента типа `FilePath` должны быть обязательными и разбираться парсером `strOption`. Остальные аргументы факультативны. Может возникнуть вопрос, как разбирать определенный нами `FileType`. Как было сказано выше, парсер, созданный с помощью `auto`, может разобрать любой тип данных, имеющий экземпляр класса типов `Read`. Мы кратко обсуждали его использование в главе 8, когда разбирали целые числа.

Итак, мы хотим разобрать значения типа `FileType`. Если мы согласны на то, чтобы они разбирались как слова, начинающиеся заглавной буквой (именно так мы называем конструкторы в исходном коде), то можно легко вывести экземпляр `Read` для этого типа:

```
data FileType
  = Other
  | Image
  | Compressed
  deriving (Show, Read, Eq)
```

Добавив `Read` во фразу `deriving`, мы легко сможем разобрать строковые значения:

```
ghci> read "Image" :: FileType
Image
ghci> read "Compressed" :: FileType
```

```
Compressed
ghci> read "Foobar" :: FileType
*** Exception: Prelude.read: no parse
```

Это прекрасно, но нужен какой-то способ сообщить пользователю, какие значения можно задавать и как их записывать. Экземпляры `Show` и `Read` двойственные в том смысле, что `Show` порождает строки, которые `Read` готов разбирать. Было бы полезно, если бы мы умели перечислять все конструкторы типа `FileType`, чтобы продемонстрировать пользователю все возможные значения. Предоставить простой список конструкторов можно так:

```
allFileTypes :: [FileType]
allFileTypes = [Other, Image, Compressed]
```

Хорошо, но что, если мы захотим расширить этот тип? Мы должны следить, чтобы эта новая константа была синхронизирована с типом, а это чревато ошибками. Нет ли какого-нибудь автоматического способа?

15.3.3 Класс Enum

Есть, потому что `FileType` – *перечислительный тип*. Такие типы имеют только *нульарные конструкторы*, или, проще говоря, конструкторы без полей. Поэтому все возможные значения типа можно перечислить путем выведения экземпляра `Enum`. Класс типов `Enum` предоставляет ряд функций для вычисления диапазона значений указанного типа и придания каждому конструктору соответствующего числового значения (начиная с `0`). Рассмотрим пример:

```
ghci> :i Enum
type Enum :: * -> Constraint
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]
  {-# MINIMAL toEnum, fromEnum #-}
ghci> data EnumType = A | B | C deriving (Show, Enum)
ghci> enumFromTo A C
[A,B,C]
ghci> toEnum 0 :: EnumType
A
ghci> toEnum 1 :: EnumType
B
```

```
ghci> map fromEnum $ enumFromTo A C
[0,1,2]
```

Обратите внимание на функцию `enumFromTo`, которая вычисляет диапазон, заданный начальным и конечным значениями. Это напоминает синтаксис диапазонов для списков, с которым мы встречались в предыдущих главах, и, как выясняется, тот синтаксис – всего лишь синтаксический сахар именно для этой функции!

```
ghci> [A .. B]
[A,B]
ghci> [B .. C]
[B,C]
ghci> [A .. C]
[A,B,C]
```

Мы можем вычислить `[1 ..]` и `['a' .. 'z']`, потому что типы `Int` и `Char` имеют экземпляры этого класса типов.

Это, конечно, помогает перечислить конструкторы, но мы по-прежнему должны вручную писать первый и последний конструкторы, так? А вот и нет, любезный читатель! Не должны, потому что есть еще один класс типов, экземпляр которого мы можем вывести, – `Bounded`, и он-то предоставит нам эти значения:

```
ghci> :i Bounded
type Bounded :: * -> Constraint
class Bounded a where
  minBound :: a
  maxBound :: a
  {-# MINIMAL minBound, maxBound #-}
ghci> data EnumType = A | B | C deriving (Show, Enum, Bounded)
ghci> minBound :: EnumType
A
ghci> maxBound :: EnumType
C
ghci> enumFromTo minBound maxBound :: [EnumType]
[A,B,C]
```

Для перечислительных типов принято выводить `Bounded` и `Enum`. Это позволяет автоматически определять все возможные конструкторы нашего типа. Изменения, которые следует внести в тип `FileType`, показаны в следующем листинге.

Листинг 15.15 Тип `FileType` и вспомогательные функции для перечисления всех его конструкторов

```
data FileType
  = Other
  | Image
  | Compressed
```

```
  deriving (Show, Read, Eq, Enum, Bounded)

allFileTypes :: [FileType]
allFileTypes = enumFromTo minBound maxBound
```

Выводит экземпляры для сериализации, разбора, сравнения и перечисления

Перечисляет все конструкторы типа `FileType` в виде списка

Теперь можно написать парсер для нашего типа `Config`.

15.3.4 Интерфейс с командной строкой для App

Поскольку наше приложение `App` порождает журнал `Log`, было бы хорошо иметь флаг, разрешающий печать журнала по завершении `App`. Мы можем определить новый тип, который содержит собственно `Config` и этот флаг `verbose`. Он показан в листинге ниже.

Листинг 15.16 Парсер аргументов командной строки

```
data CliArgs = CliArgs
  { cliArgsConfig :: Config,
    cliArgsVerbose :: Bool
  }

cliArgsParser :: Parser CliArgs
cliArgsParser = CliArgs <$> config <*> isVerbose
  where
    config =
      Config
        <$> fromPath
        <*> toPath
        <*> onlyFileType
        <*> maxRecDepth
        <*> maxFiles
      fromPath =
        strOption $
          long "from-path"
            <> metavar "PATH"
            <> help "Path to sync files from"
      toPath =
        strOption $
          long "to-path"
            <> metavar "PATH"
            <> help "Path to sync files to"
      fileTypeMetaVar = L.intercalate "|" $
➥ map show allFileTypes
      onlyFileType =
```

Тип для аргументов командной строки

Конструирует значение типа Parser CliArgs

Конструирует значение типа Parser Config

Разбирает обязательные аргументы – пути к каталогам

Создает текстовое представление различных значений типа FileType, задаваемых пользователем

```
    optional $
      option auto $
        long "only-filetype"
          <> help "Only consider files of a certain filetype"
          <> metavar fileTypeMetaVar
  maxRecDepth =
    optional $
      option auto $
        long "max-rec-depth"
          <> help "Limit maximum recursive depth to traverse"
          <> metavar "NUMBER"
  maxFiles =
    optional $
      option auto $
        long "max-files"
          <> help "Limit number of files to sync"
          <> metavar "NUMBER"
isVerbose =
  switch
    ( short 'v'
        <> long "verbose"
        <> help "Enable verbose output"
    )
```

Разбирает факультативные аргументы

Разбирает флаг --verbose

Здесь видно, как используется *аппликативный* стиль для построения парсера более сложного типа. На основе этого парсера мы можем сконструировать наш тип `ParserInfo`. Это показано в листинге ниже.

Листинг 15.17 `ParserInfo` для аргументов командной строки приложения

```
argumentParser :: ParserInfo CliArgs
argumentParser =
  info
    (cliArgsParser <**> helper)
    ( fullDesc
        <> progDesc "A simple file synchronization tool"
    )
```

Создает ParserInfo из Parser

Объединяет обычный парсер аргументов командной строки с дополнительным парсером флага --help

Предоставляет рудиментарное описание программы

И в заключение напишем действие ввода-вывода для печати типа `Log`. В следующем листинге показано это действие, а заодно действие `main`, которое выполняет парсер аргументов. Далее мы видим рассмотренное ранее действие `mainApp` и условную распечатку журнала.

Листинг 15.18 Действие, печатающее журнал работы приложения

```
printLog :: Log -> IO ()
printLog (Log xs) = forM_ xs (putStrLn . prettyEntry)
  where
    prettyEntry (App.Success (SuccessEntry _ toPath)) =
      "[*] " ++ takeFileName toPath
    prettyEntry (App.Failure (FailureEntry path exc)) =
      "[!] " ++ takeFileName path ++ ": " ++ show exc
    prettyEntry (App.Skipped (SkipEntry path reason)) =
      "[s] " ++ takeFileName path ++ ": " ++ reason

main :: IO ()
main = do
  args <- execParser argumentParser
  result <- runApp mainApp $ cliArgsConfig args
  when (cliArgsVerbose args) $ printLog result
```

Печатает каждую запись журнала

Показывает имя успешно скопированного файла

Показывает имя файла и исключение в случае ошибки

Показывает имя файла и причину, в случае когда файл пропущен

Выполняет парсер аргументов

Выполняет mainApp с конфигурацией, полученной от argumentParser

Печатает журнал, если был задан флаг --verbose

Здесь мы включили в имена конструкторов префиксы `Success`, `Failure` и `Skipped`, чтобы не возникало конфликтов имен с `Options.Applicative`.

Упражнение. Удаление лишних файлов

Наше приложение только копирует файлы из начального каталога в конечный. Однако оно игнорирует файлы, которые имеются в конечном каталоге, но *отсутствуют* в начальном. Модифицируйте приложение, так чтобы оно удаляло такие файлы из конечного каталога. Кроме того, определите новый аргумент командной строки, который будет отключать эту функциональность.

Теперь протестируем приложение. Предположим, что в каталоге проекта имеется следующий каталог *foo*:

```
shell $ tree foo
foo
├── bar
│   ├── fileB.txt
│   └── fileC.txt
├── baz
│   ├── fileD.txt
│   └── fileE.txt
└── fileA.txt
```

Мы можем синхронизировать его с другим каталогом, воспользовавшись написанной программой:

```
shell $ stack run -- --from-path "foo" --to-path "/tmp/bar" --verbose
[*] fileA.txt
[*] fileD.txt
[*] fileE.txt
[*] fileB.txt
[*] fileC.txt
shell $ tree /tmp/bar
/tmp/bar
├── bar
│   ├── fileB.txt
│   └── fileC.txt
├── baz
│   ├── fileD.txt
│   └── fileE.txt
└── fileA.txt
```

И на этом мы завершаем эксперименты с синхронизацией каталогов.

Резюме

- Монадные трансформеры используются для расширения функциональности монад.
- Трансформер `ReaderT` наделяет монаду способностью читать из окружения.
- Трансформер `StateT` наделяет монаду способностью читать и модифицировать состояние во время ее выполнения.
- Трансформер `WriterT` наделяет монаду способностью порождать «побочный» вывод в процессе вычисления других, возвращаемых, значений.
- Трансформер `RWST` является комбинацией `ReaderT`, `StateT` и `WriterT`, не требующей организации стека трансформеров.
- В пакете `optparse-applicative` определены парсеры и комбинаторы для разбора сложных аргументов командной строки.

16 JSON и SQL

Краткое содержание этой главы:

- разбор и сериализация JSON;
- вывод классов типов с помощью `Generic`;
- использование баз данных SQLite в программах.

В предыдущих главах мы познакомились со многими концепциями и приемами работы в Haskell и применили их для написания нескольких программ. Пришло время использовать все полученные знания для реализации более крупного проекта. Эта и следующая главы станут демонстрацией написания на Haskell приложения, в котором используются база данных и сеть. В итоге мы получим типобезопасный API на основе HTTP и JSON. Кроме того, мы автоматически выведем клиентское приложение, которое будет безопасно взаимодействовать с этим API. Попутно мы познакомимся с несколькими новыми пакетами.

Мы начнем с обсуждения работы с данными в формате JSON с помощью библиотеки `aeson` и поговорим о том, как обобщенным образом выводить классы типов. Затем рассмотрим использование SQLite в программах, а также передачу данных в базу и их получение

из базы. Этими знаниями мы воспользуемся в следующей главе для построения приложения.

16.1 Кодирование значений в формате JSON

В последнем проекте мы хотим создать приложение для управления списком задач, применяя для этой цели API. Это означает, что нужно *серверное приложение*, сообщающее информацию о состоянии этого списка в виде JSON-кодированных данных по *протоколу передачи гипертекста* (HTTP). Кроме того, мы хотели бы иметь парное *клиентское приложение*, которое может взаимодействовать с сервером, как с командной строкой. Также серверное приложение должно генерировать HTML-страницу, отображающую наш список задач в удобном для восприятия виде.

То, что мы собираемся сделать в этом проекте, называется API *создания, чтения, обновления, удаления* (create, read, update, delete – CRUD). Его можно было бы использовать для реализации следующих приложений:

- службы анкетирования;
- планировщики деловых встреч;
- платформы бронирования;
- хранение данных на сервере.

Мы начнем реализацию с определения структуры данных, пересылаемых по сети.

16.1.1 Aeson и разбор JSON

Создадим новый проект командой `stack new todolist`.

Поскольку мы имеем дело с JSON API, необходим способ сериализовать наш тип данных в формате JSON и разбирать этот формат. Как сделать это на Haskell? Самый известный и широко используемый пакет для этой цели называется `aeson`. После добавления в секцию зависимостей yml-файла мы можем протестировать этот пакет в GHCi с помощью команды `stack repl`:

```
ghci> import Data.Aeson as J
ghci> :set -XOverloadedStrings
ghci> :i J.Value
type Value :: *
data Value
  = Object !Object
  | Array !Array
  | String !Text
  | Number !Scientific
```

```
  | Bool !Bool
  | Null
ghci> J.decode "{\"a\":\"Hello\", \"b\": [1,2,3]}" :: Maybe J.Value
Just (Object (fromList [("a",String "Hello"),("b",Array
➥ [Number 1.0,Number 2.0,Number 3.0])]))
ghci> :t J.decode
J.decode :: FromJSON a => ByteString -> Maybe a
```

Здесь мы видим тип `Value` – Haskell-представление возможных JSON-значений. Видно, что функция `decode` принимает ленивую строку `ByteString` (именно поэтому нам необходимо расширение языка `OverloadedStrings`) и разбирает ее в значение, удовлетворяющее ограничению `FromJSON`. Это позволяет получить простое JSON-представление строки. С помощью функции `encode` мы можем впоследствии снова получить из этого значения строку в формате JSON:

```
ghci> J.decode "[1,2,3]" :: Maybe [Int]
Just [1,2,3]
ghci> J.encode ([1,2,3] :: [Int])
"[1,2,3]"
ghci> :t J.encode
J.encode :: ToJSON a => a -> ByteString
```

Как видим, на `encode` наложено ограничение `ToJSON`; это класс типов, позволяющий сериализовывать значения. Если мы хотим поддержать преобразование типа в формат JSON, то должны создать для него экземпляры `ToJSON` и `FromJSON`.

Для начала займемся созданием типа, который поместим в модуль `Todos.Types`. Нам нужно представить запись списка задач. Какие поля должны быть в такой записи? Поскольку мы хотим ссылаться на задачи в API, нужен идентификатор. Кроме того, должно быть текстовое описание задачи и признак ее пометки как выполненной. Так как нам понадобится отображать список, нужно подумать о его сортировке. Для этого добавим необязательный ранг. После того как мы добавим зависимость `text` в файл `package.yml`, тип будет выглядеть следующим образом:

```
import qualified Data.Text as T

data TodoEntry = TodoEntry
  { entryId :: Maybe Int,
    text :: Text,
    checked :: Bool,
    rank :: Maybe Int
  }
  deriving (Eq, Show)
```

Теперь подумаем об экземплярах классов типов, относящихся к JSON. В экземпляре `FromJSON` должна быть реализована функция

`parseJSON` типа `Value -> Parser a`. Построить такой `Parser` можно с помощью вспомогательных функций, предоставляемых модулем `Data.Aeson`. Вот некоторые из них:

- `withObject :: String -> (Object -> Parser a) -> Value -> Parser a`;
- `withText :: String -> (Text -> Parser a) -> Value -> Parser a`;
- `withArray :: String -> (Array -> Parser a) -> Value -> Parser a`;
- `withScientific :: String -> (Scientific -> Parser a) -> Value -> Parser a`;
- `withBool :: String -> (Bool -> Parser a) -> Value -> Parser a`.

Эти функции получают имя в первом аргументе (оно нужно для формирования понятных сообщений об ошибках), функцию, а также значение `Value` в формате JSON `Value`, к которому эта функция применяется. Результирующие парсеры возвращают ошибку, если `Value` не является JSON-значением, в частности `withObject` завершается неудачно, если `Value` – не объект.

```
import qualified Data.Aeson as J
import qualified Data.Aeson.Types as J
import qualified Data.Text as T
import qualified Data.ByteString.Lazy as BSL

myParser :: BSL.ByteString -> J.Parser T.Text
myParser string =
  case J.decode string of
    Nothing -> fail "failed to decode value as JSON"
    Just json -> J.withText "a simple string" pure json
```

Здесь мы определяем простой парсер, который декодирует ленивую `ByteString`, содержащую JSON, и разбирает простой строковый литерал. Если декодировать строку не удалось, то парсер вызывает функцию `fail`. Это возможно, потому что существует экземпляр `MonadFail Parser`:

```
ghci> import qualified Data.Aeson.Types as J
ghci> J.parse myParser "\"Hello\""
Success "Hello"
ghci> J.parse myParser "1"
Error "parsing a simple string failed, expected String,
➥ but encountered Number"
ghci> J.parse myParser ""
Error "failed to decode value as JSON"
```

Используя `withText`, мы получаем сообщение об ошибке, которое точно говорит, чего мы ожидали и что получили на самом деле. Далее рассмотрим более интересную часть: разбор JSON-объекта. Объекты состоят из пар ключ–значение, в которых строковые ключи отображаются в произвольные JSON-значения. `Data.Aeson` предлагает для разбора несколько операторов:

- `(.:) :: FromJSON a => Object -> Key -> Parser a` – требует, чтобы в объекте присутствовал некоторый ключ, и возвращает связанное с ним значение;
- `(.:?) :: FromJSON a => Object -> Key -> Parser (Maybe a)` – ищет в объекте ключ и возвращает значение, если он присутствует и не равен `Null`;
- `(.:!) :: FromJSON a => Object -> Key -> Parser (Maybe a)` – то же, что `(.:?)`, но пытается разобрать `Null` как значение;
- `(.!=) :: Parser (Maybe a) -> a -> Parser a` – может использоваться в сочетании с `(.:?)` и `(.:!)` для предоставления значения по умолчанию.

Имея эти операторы, мы можем определить парсер, который принимает JSON-значение `Value` и пытается построить из него запись типа `TodoEntry`. Разумно потребовать, чтобы JSON-представление `TodoEntry` было объектом, ключи которого называются так же, как в типе Haskell. Типы `Text`, `Int` и `Bool` не имеют прямых аналогов в JSON, поэтому о значениях мы знаем только из соответствующих ключей. Поля, разбор которых может дать `Maybe`, факультативны. Все это можно лаконично записать в виде простого парсера, показанного в листинге ниже.

Листинг 16.1 Парсер JSON для `TodoEntry`

```
{-# LANGUAGE RecordWildCards #-}

...

todoEntryParser :: J.Value -> J.Parser TodoEntry
todoEntryParser = J.withObject "TodoEntry" $
➥ \object -> do
  entryId <- object J..:? "entryId"
  text <- object J..: "text"
  checked <- object J..: "checked"
  rank <- object J..:? "rank"
  return TodoEntry {..}
```

Ожидает JSON-объект, чтобы парсер мог продолжить работу

Разбирает факультативные поля JSON-объекта

Разбирает обязательные поля JSON-объекта

Разбирает факультативные поля JSON-объекта

Строит полную запись TodoEntry (используется расширение языка RecordWildCards)

Теперь мы можем написать новую функцию разбора, которая принимает `ByteString` и порождает парсер нашего типа `TodoEntry`:

```
parseTodoEntry :: BSL.ByteString -> J.Parser TodoEntry
parseTodoEntry string =
  case J.decode string of
    Nothing -> J.parseFail "failed to decode value as JSON"
    Just json -> todoEntryParser json
```

С помощью этой функции можно разбирать строки в формате JSON, соответствующие типу `TodoEntry`:

```
ghci> J.parse parseTodoEntry "{\"text\":\"Do something\",\"checked\":false}"
Success (TodoEntry {entryId = Nothing, text = "Do something",
➥ checked = False, rank = Nothing})
ghci> J.parse parseTodoEntry "{\"text\":\"Do something\",
➥ \"checked\":false,\"rank\":100}"
Success (TodoEntry {entryId = Nothing, text = "Do something",
➥ checked = False, rank = Just 100})
ghci> J.parse parseTodoEntry "{\"text\":\"Do something\"}"
Error "key \"checked\" not found"
```

Однако функция `parseTodoEntry` необязательна, потому что `decode` умеет автоматически разбирать значения типа `ByteString`, если для типа существует экземпляр `FromJSON`. Этот экземпляр можно определить с помощью нашей функции `todoEntryParser`, как показано в листинге ниже.

Листинг 16.2 Экземпляр класса `FromJSON` для `TodoEntry`

```
instance J.FromJSON TodoEntry where
  parseJSON = todoEntryParser
```

Определяет функцию parseJSON для экземпляра FromJSON

Имея этот экземпляр, мы можем непосредственно разбирать значения типа `TodoEntry`, не прибегая к написанной выше вспомогательной функции:

```
ghci> J.decode "{\"text\":\"Do something\",\"checked\":false}"
➥ :: Maybe TodoEntry
Just (TodoEntry {entryId = Nothing, text = "Do something",
➥ checked = False, rank = Nothing})
```

Научившись разбирать JSON-строки как значения типа `TodoEntry`, мы должны решить еще одну задачу: сериализовать значения в формат JSON. Это можно сделать, реализовав экземпляр `ToJSON`.

16.1.2 Сериализация в формат JSON

Для преобразования значений из Haskell в формат JSON мы можем воспользоваться функцией `toJson`, которая создает значение типа `Value`, и функцией `encode`, которая преобразует его в ленивую строку байтов. Преобразовать строки, числа и булевы значения легко:

```
ghci> J.toJSON ("Hello" :: String)
String "Hello"
```

```
ghci> J.toJSON (100 :: Int)
Number 100.0
ghci> J.toJSON (True :: Bool)
Bool True
```

Сложнее сконструировать объект с ключами и значениями:

- `object :: [Pair] -> Value` – конструирует `Value` из списка значений типа `Pair`;
- `(.=) :: (KeyValue kv, ToJSON v) => Key -> v -> kv` – конструирует из ключа и значения пару `KeyValue`, которую можно сериализовать в формат JSON.

С помощью этих двух функций мы можем написать сериализатор, преобразующий `TodoEntry` в JSON.

Листинг 16.3 Экземпляр класса `ToJSON` для `TodoEntry`

```
todoEntrySerializer :: TodoEntry -> J.Value
todoEntrySerializer TodoEntry {..} = do
  J.object
    [ "entryId" J..= entryId,
      "text" J..= text,
      "checked" J..= checked,
      "rank" J..= rank
    ]

instance J.ToJSON TodoEntry where
  toJSON = todoEntrySerializer
```

Конструирует объект Value из пар ключ–значение

Строит пары ключ–значение из полей в аргументе

Определяет функцию toJSON для экземпляра ToJSON

Определив этот экземпляр, мы можем протестировать функции `encode` и `decode` и убедиться, что разбор закодированного значения дает исходное значение:

```
ghci> e = TodoEntry Nothing "Do Something" False (Just 1)
ghci> J.encode e
"{\"checked\":false,\"entryId\":null,\"rank\":1,\"text\":\"Do Something\"}"
ghci> J.decode $ J.encode e :: Maybe TodoEntry
Just (TodoEntry {entryId = Nothing, text = "Do Something",
➥ checked = False, rank = Just 1})
```

Чудненько! Мы написали функции для преобразования значений в формат JSON и обратно. Но возникает вопрос: не является ли этот код трафаретным и скучноватым? Глядя на наши экземпляры, мы видим, что должны вручную добавлять запросы и пары ключ–значение для каждого поля записи. Допустим, что понадобится добавить еще одно поле. Тогда его нужно будет добавить в функции разбора и сериализации, хотя компилятор мог бы определить это статически, поскольку все имеющиеся в типе поля известны.

16.2 Вывод классов типов с помощью Generic

Чтобы избежать ручной работы, мы можем воспользоваться модулем `GHC.Generics`. В Haskell встроен механизм вывода, позволяющий разработчикам библиотек обращаться к обобщенному представлению типа данных, которое можно использовать для определения экземпляров. Мы не станем вдаваться в детали этих представлений и в способы их применения для реализации типов классов, а сосредоточимся на том, как этот механизм позволяет избежать написания трафаретного кода.

Основной интерфейс к этому механизму – класс типов `Generic`, в котором имеется функция для преобразования значения в его *представление* и функция для преобразования представления обратно в значение:

```
ghci> import GHC.Generics
ghci> :i Generic
type Generic :: * -> Constraint
class Generic a where
  type Rep :: * -> * -> *
  type family Rep a
  from :: a -> Rep a x
  to :: Rep a x -> a
  {-# MINIMAL from, to #-}
```

Библиотека `aeson` может воспользоваться этим, чтобы автоматически предоставить нам JSON-представления наших значений. С этой целью она определяет функции `genericToJSON` и `genericParseJSON`, являющиеся реализациями по умолчанию функций `toJSON` и `parseJSON` из классов типов `ToJSON` и `FromJSON`. Таким образом, нам нужно только предоставить экземпляр `Generic` для своего типа, а экземпляры `ToJSON` и `FromJSON` мы получим даром!

По счастью, компилятор может сделать это автоматически. Для этого нужно активировать расширение языка `DeriveGeneric`, а затем вывести `Generic` для нашего типа данных. После этого мы сможем создать экземпляры `ToJSON` и `FromJSON` с пустым определением. Все это будет находиться в новом модуле, показанном в листинге ниже.

Листинг 16.4 Экземпляр класса `ToJSON` для `TodoEntry`

```
{-# LANGUAGE DeriveGeneric #-}

module Todos.Types where

import qualified Data.Aeson as J
import qualified Data.Text as T
import GHC.Generics (Generic)
```

Активирует расширение языка DeriveGeneric, чтобы можно было выводить класс типов Generic

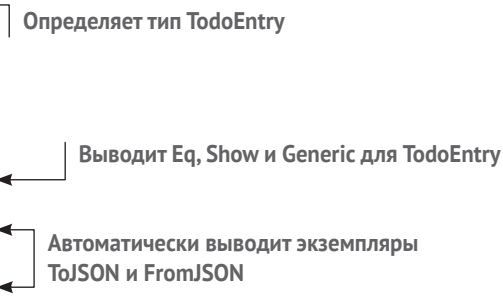
Импортирует Data.Aeson и Data.Text с квалифицированными именами

Импортирует класс типов Generic из модуля GHC.Generics

```
data TodoEntry = TodoEntry
  { entryId :: Maybe Int,
    text :: T.Text,
    checked :: Bool,
    rank :: Maybe Int
  }
  deriving (Eq, Show, Generic)

instance J.ToJSON TodoEntry

instance J.FromJSON TodoEntry
```



Пользуясь этим определением, мы можем протестировать, работает ли все еще преобразование туда-обратно (кодирование с последующим декодированием должно восстанавливать исходное значение):

```
ghci> e = TodoEntry Nothing "Do Something" False (Just 1)
ghci> J.encode e
"{\"checked\":false,\"entryId\":null,\"rank\":1,
➥ \"text\":\"Do Something\"}"
ghci> J.decode $ J.encode e :: Maybe TodoEntry
Just (TodoEntry {entryId = Nothing, text = "Do Something",
➥ checked = False, rank = Just 1})
```

Но как именно `aeson` выводит экземпляры? Есть несколько общих правил:

- именованные поля ADT представляются парами ключ–значение в объекте;
- множественные конструкторы представляются ключом `tag`, содержащим имя конструктора;
- конструкторы с одним непомеченным полем преобразуются в это поле;
- конструкторы с несколькими непомеченными полями преобразуются в JSON-массивы;
- тип, содержащий только конструкторы без полей, сериализуется в строку, содержащую имя конструктора;
- значения `Maybe` кодируются как факультативные: в случае отсутствия они будут сериализованы в `null`;
- типы, содержащие другие типы, которые могут быть сериализованы в JSON, оказываются рекурсивно вложенными.

В общем случае можно предполагать, что экземпляры `ToJSON` генерируют JSON-значения, которые может разобрать `FromJSON`. Логика взаимоотношений между сериализацией и разбором не меняется.

ПРИМЕЧАНИЕ Использование обобщенных экземпляров `FromJSON` и `ToJSON` в большинстве случаев не вызывает осложнений, но нужно иметь в виду, что изменение типа данных

влечет за собой изменение этих классов типов. Если наша программа сохраняет JSON-код надолго, то может быть нарушена обратная совместимость. Если вас интересует сериализация данных с поддержкой обратной совместимости, то присмотритесь к пакетам `safecopy` или `proto-lens`.

А мы на этом заканчиваем знакомство с JSON-сериализацией. И переходим к сохранению значений в базе данных.

16.3 Использование базы данных SQLite

Мы хотим сохранять записи из списка задач в базе данных, чтобы список не исчезал после перезагрузки сервера. Различных систем управления базами данных, которые можно было бы включить в проект, много, например Postgresql, MySQL или MariaDB. Но мы остановились на *SQLite*, потому что ее проще настроить. SQLite – низкоуровневая библиотека, которая хранит базу данных в обычном файле, но при этом позволяет выполнять SQL-запросы для чтения и модификации данных. У нее есть интерфейсы к разным языкам, в т. ч. к Haskell. Причем для работы с SQLite имеется не одна, а много библиотек, каждая со своими уникальными особенностями.

В этой главе мы будем пользоваться пакетом `sqlite-simple`, потому что с ним (как явствует из названия) легко работать. Если другие библиотеки предлагают предметно-ориентированные языки или типобезопасное моделирование, то `sqlite-simple` демонстрирует более прямолинейный подход к интерфейсу с базой данных.

16.3.1 Основы sqlite-simple

Для использования пакета нужно добавить строку `sqlite-simple` в секцию `dependencies` файла `package.yml`. И затем поэкспериментировать, выполнив команду `stack repl`:

```
ghci> :set -XOverloadedStrings
ghci> :set -XScopedTypeVariables
ghci> import Database.SQLite.Simple
ghci> dbConn <- open ":memory:"
ghci> execute_ dbConn "CREATE TABLE myTable (id INTEGER PRIMARY KEY,
➥ data TEXT)"
ghci> execute dbConn "INSERT INTO myTable VALUES (?,?)" (1 :: Int, "hello" ::
     String)
ghci> res :: [(Int, String)] <- query_ dbConn "SELECT * FROM myTable"
ghci> res
[(1,"hello")]
```

Здесь мы видим подключение к базе данных, создание таблицы, запись в нее данных и выборку данных. Для подключения мы воспользовались действием `open :: String -> IO Connection`, которое создает базу данных, если той еще не существует. Первым аргументом обычно является путь к локальному файлу, но можно использовать и `":memory:"`, чтобы создать временную базу данных в памяти, которая будет автоматически удалена после использования.

Самые простые функции для взаимодействия с базой данных – `query` и `execute`:

- `query :: (ToRow q, FromRow r) => Connection -> Query -> q -> IO [r]`;
- `query_ :: FromRow r => Connection -> Query -> IO [r]`;
- `execute :: ToRow q => Connection -> Query -> q -> IO ()`;
- `execute_ :: Connection -> Query -> IO ()`;
- `executeMany :: ToRow q => Connection -> Query -> [q] -> IO ()`;
- `executeNamed :: Connection -> Query -> [NamedParam] -> IO ()`.

Обе функции имеют по два варианта: с данными `q`, включаемыми в запрос, и без данных. Функция `query` служит для выполнения запросов, возвращающих данные, например `SELECT`, а функция `execute` – для запросов, не возвращающих данные, например `INSERT`, `UPDATE` или `DELETE`. Функция `changes` позволяет узнать, сколько строк было затронуто запросом:

```
ghci> rows = [(i :: Int, "hello" :: String) | i <- [2..10]]
ghci> executeMany dbConn "INSERT INTO myTable VALUES (?,?)" rows
ghci> execute_ dbConn "UPDATE myTable SET data = 'updated' WHERE id > 5"
ghci> changes dbConn
5
ghci> executeNamed dbConn "UPDATE myTable SET data = :data WHERE id =
➥ :id" [":data" := "HELLO" :: String, ":id" := 1 :: Int]
ghci> res :: [(Int, String)] <- query_ dbConn "SELECT * FROM myTable"
ghci> res
[(1,"hello"),(2,"hello"),(3,"hello"),(4,"hello"),(5,"hello"),(6,"updated"),
➥ (7,"updated"),(8,"updated"),(9,"updated"),(10,"updated")]
```

В `executeNamed` мы видим специальный синтаксис для получения списка именованных параметров. Смысл `(:=)` мы объясним позже. А пока посмотрим, как поместить данные из программы в базу.

16.3.2 Функции ToRow и FromRow

Знак `?` обозначает места в запросе, куда должны быть вставлены данные. Чтобы превратить значения Haskell в нечто, чем может воспользоваться библиотека, имеется класс типов `ToRow`. Наоборот, класс `FromRow` преобразует данные в значения Haskell. Приведем примеры работы с этими классами:

```
ghci> :i ToRow
type ToRow :: * -> Constraint
class ToRow a where
  toRow :: a -> [SQLData]
  default toRow :: (Generic a, GToRow (Rep a)) => a -> [SQLData]
ghci> :i FromRow
type FromRow :: * -> Constraint
class FromRow a where
  fromRow :: RowParser a
  default fromRow :: (Generic a, GFromRow (Rep a)) => RowParser a
ghci> :i SQLData
type SQLData :: *
data SQLData
  = SQLInteger Int64
  | SQLFloat Double
  | SQLText Text
  | SQLBlob ByteString
  | SQLNull
```

Здесь мы видим (сильно упрощенный) вывод GHCi. Функция `toRow` из класса `ToRow` преобразует значение в список `SQLData`, представляющий собой просто помеченное объединение. Функция `fromRow` из класса `FromRow` определяет парсер таких данных. Тип `RowParser` чуть более сложен, но в общем случае мы можем предполагать, что это парсер, преобразующий `SQLData` в какое-то значение. Экземпляры `ToRow` определены для кортежей (размера не более 10), равно как и для списков значений, но только если их элементы удовлетворяют ограничению `ToField`:

```
ghci> import Database.SQLite.Simple.ToField
ghci> :i ToField
type ToField :: * -> Constraint
class ToField a where
  toField :: a -> SQLData
ghci> toField ("hello" :: String)
SQLText "hello"
ghci> toField (1 :: Int)
SQLInteger 1
ghci> toField (Nothing :: Maybe Int)
SQLNull
ghci> toField (Just 1 :: Maybe Int)
SQLInteger 1
```

Собственно преобразованием значения Haskell в тип `SQLData` занимается функция `toField`. Ограничение `ToField` необходимо, чтобы оператор `(:=)` мог преобразовать данные в именованные параметры:

```
ghci> :t (:=)
(:=) :: ToField v => Text -> v -> NamedParam
```

```
ghci> "param" := 1
("param",SQLInteger 1)
```

По счастью, экземпляры класса типов `ToField` уже определены для наиболее употребительных типов:

```
ghci> toRow (1 :: Int, "hello" :: String)
[SQLInteger 1,SQLText "hello"]
ghci> toRow [1,2,3,4,5 :: Int]
[SQLInteger 1,SQLInteger 2,SQLInteger 3,SQLInteger 4,SQLInteger 5]
ghci> toRow (Only "hello" :: Only String)
[SQLText "hello"]
```

Значение `Only` нужно, когда мы хотим использовать в запросе единственный параметр, потому что кортежей с одним элементом не бывает:

```
ghci> res :: [(Int, String)] <- query dbConn "SELECT * FROM myTable
➥ WHERE id = ?" (Only 1 :: Only Int)
ghci> res
[(1,"hello")]
```

Теперь, когда мы знаем, как `sqlite-simple` обращается с данными Haskell, можно написать экземпляры для нашего типа `TodoEntry`. Начнем с экземпляра `ToRow TodoEntry`, показанного в листинге ниже.

Листинг 16.5 Экземпляр `ToRow` для `TodoEntry`

```
import Database.SQLite.Simple.ToRow
import Database.SQLite.Simple.ToField
...

instance ToRow where
  toRow (TodoEntry entryId text checked rank) =
    [ toField entryId,
      toField text,
      toField checked,
      toField rank
    ]
```

Импортирует класс типов ToRow

Импортирует класс типов ToField

Определяет функцию toRow

Определяет строку SqlData, соответствующую TodoEntry

Это простое определение позволяет преобразовать запись `TodoEntry` в список `SQLData`:

```
ghci> toRow (TodoEntry Nothing "Hello" False (Just 1))
[SQLNull,SQLText "Hello",SQLInteger 0,SQLInteger 1]
ghci> toRow (TodoEntry (Just 10) "Hello" True Nothing)
[SQLInteger 10,SQLText "Hello",SQLInteger 1,SQLNull]
```

Отметим, что `Bool` преобразуется в целое, поскольку в SQLite нет специального булева типа.

Научившись преобразовывать значение `TodoEntry` в строку базы данных, мы должны научиться обратной операции: чтению данных из строки. Для определения парсера нам сначала нужно написать парсер одного поля. За это отвечает класс типов `FromField`:

```
ghci> import Database.SQLite.Simple.FromField
ghci> import Database.SQLite.Simple.FromRow
ghci> :i FromField
type FromField :: * -> Constraint
class FromField a where
  fromField :: FieldParser a
ghci> :i FieldParser
type FieldParser :: * -> *
type FieldParser a = Field -> Ok a
```

Функция `fromField` определяет парсер `FieldParser`, который принимает поле и возвращает тип, похожий на `Maybe` с результатом разбора. Мы не будем заниматься определением парсеров `FieldParser`, потому что для тех типов, которые нам нужны, они уже определены. А как определить `RowParser` из нескольких `FieldParser`? Тут нам поможет функция `field` типа `FromField a => RowParser a`, которая определяет `RowParser` для значений, имеющих `FieldParser`. После этого мы можем применить *аппликативный* стиль из глав 12 и 14 для объединения нескольких `RowParser` в один парсер. Это показано в листинге ниже.

Листинг 16.6 Экземпляр `FromRow` для `TodoEntry`

```
import Database.SQLite.Simple.FromRow
...

instance FromRow where
  fromRow =
    TodoEntry
      <$> field
      <*> field
      <*> field
      <*> field
```

Импортирует класс типов FromRow и функцию field

Определяет функцию fromRow

Определяет RowParser, который разбирает все поля типа TodoEntry, получив строку из базы данных

Определив эти экземпляры, мы можем сначала создать таблицу, в которой будут храниться значения нашего типа данных, а затем вставить значение как строку и выбрать ее обратно:

```
ghci> execute_ dbConn "CREATE TABLE IF NOT EXISTS entries (
➥ id INTEGER PRIMARY KEY, text TEXT, checked INTEGER, rank INTEGER)"
ghci> execute dbConn "INSERT INTO entries (id, text, checked, rank)
➥ VALUES (?,?,?,?)" (TodoEntry Nothing "Hello" False (Just 1))
ghci> res :: [TodoEntry] <- query_ dbConn "SELECT * FROM entries"
ghci> res
[TodoEntry {entryId = Just 1, text = "Hello", checked = False, rank = Just 1}]
```

Однако не кажется ли вам, что повторяющееся использование `toField` и `field` выглядит как трафаретный код? Нельзя ли как-нибудь вывести его обобщенно? Да, экземпляры `ToRow` и `FromRow` можно вывести автоматически, если для нашего типа имеется экземпляр `Generic`. Чтобы это заработало, для полей типа данных должны быть определены экземпляры классов `ToField` и `FromField`, но, как мы уже знаем, для нашего типа `TodoEntry` это условие выполнено. Поэтому мы можем вывести необходимые экземпляры точно так же, как сделали для `FromJSON` и `ToJSON`. Это показано в листинге ниже.

Листинг 16.7 Обобщенные экземпляры `FromRow` и `ToRow` для `TodoEntry`

```
import qualified Database.SQLite.Simple.FromRow as SQL
import qualified Database.SQLite.Simple.ToRow as SQL

...

instance SQL.FromRow TodoEntry

instance SQL.ToRow TodoEntry
```

Импортирует необходимые модули с квалифицированным именем SQL

Определяет экземпляры классов FromRow и ToRow для TodoEntry

Наш тип, равно как и экземпляры, будет находиться в модуле `Todos.Types`. Теперь, когда мы умеем преобразовывать записи в SQL и обратно, можно написать несколько функций для взаимодействия с базой данных.

16.3.3 Определение действий для доступа к базе данных

Наш API будет содержать следующую функциональность:

- вывод списка всех записей;
- добавление записей;
- обновление одной записи.

Теперь мы хотим реализовать функции взаимодействия с базой данных и поместить весь код в один модуль. Для этого мы создадим новый модуль `Todos.Database`. Поскольку наш сервер сначала создает подключение `Connection` и все дальнейшее взаимодействие с базой ведется через него, имеет смысл завести функции, которые принимают подключение в качестве аргумента и модифицируют базу данных.

Когда база данных только создана, она не содержит никаких данных и в том числе таблицы, куда мы собираемся записывать. Поэтому нужна функция, инициализирующая базу. Она выполнит команду `CREATE TABLE IF NOT EXISTS`, которая гарантирует, что таблица будет создана, но не изменяет уже существующую таблицу. Эта функция показана в следующем листинге.

Листинг 16.8 Функция инициализации базы данных

```
module Todos.Database where

import Database.SQLite.Simple

setupDb :: Connection -> IO ()
setupDb dbConn =
  execute_
    dbConn
    "CREATE TABLE IF NOT EXISTS entries \
    \ (id INTEGER PRIMARY KEY AUTOINCREMENT, \
    \ text TEXT, checked INTEGER, rank INTEGER)"
```

Выполняет запрос к базе данных

Задает запрос, который создает таблицу для хранения записей списка задач, если она не существует

Мы определяем поле `id` как `INTEGER PRIMARY KEY AUTOINCREMENT`, чтобы каждая новая запись автоматически получала уникальный целый идентификатор. Кроме того, при попытке вставить строку с уже существующим значением `id` будет возбуждаться исключение.

Разобравшись с инициализацией, мы можем написать функцию выборки записей. В терминах SQL это будет простой запрос `SELECT * FROM entries`. На наше счастье, больше ничего делать не нужно, потому что за разбор строк, прочитанных из базы данных, отвечает тип `FromRow TodoEntry`. Полное определение показано в листинге ниже.

Листинг 16.9 Функция выборки всех записей из таблицы базы данных

```
import Todos.Types

...

getEntries :: Connection -> IO [TodoEntry]
getEntries dbConn = query_ dbConn
➥ "SELECT * FROM entries"
```

Запрос к базе данных для выборки всех записей из таблицы

Теперь пора написать функцию, которая добавляет в базу данных запись `TodoEntry`. В принципе, нужно воспользоваться функцией `execute` с командой `INSERT INTO`, но в данном случае требуется осторожность, потому что при определении таблицы мы добавили фразу `PRIMARY KEY`. Вставка двух записей с одинаковым `id` приведет к исключению `SQLError`, поэтому мы должны учитывать возможность ошибки при выполнении запроса. Для этого можно использовать функцию `try`, с которой мы познакомились в главе 14. Но нельзя ли как-нибудь сделать саму базу более отказоустойчивой? Для этой цели модуль `Database.SQLite.Simple` предоставляет функцию `withTransaction :: Connection -> IO a -> IO a`, которая начинает транзакцию в базе данных (командой `BEGIN TRANSACTION`), затем выполняет указанное действие, после чего фиксирует транзакцию (командой `COMMIT TRANSACTION`). Если действие возбуждает исключение, то транз-

акция откатывается (командой `ROLLBACK TRANSACTION`), и база данных остается в исходном состоянии. Пользуясь этими двумя функциями, мы можем написать функцию, добавляющую `TodoEntry` в базу данных. Этот код показан в листинге ниже.

Листинг 16.10 Функция добавления записи `TodoEntry` в базу данных

```
import Control.Exception (try)

...

addEntry :: Connection -> TodoEntry -> IO (Either SQLError Int)
addEntry dbConn entry = try $
  withTransaction dbConn $ do
    execute
      dbConn
      "INSERT INTO entries (id, text, checked, rank) \
      \ VALUES (?,?,?,?)"
      entry
    newIdx <- lastInsertRowId dbConn
    return $ fromIntegral newIdx
```

Перехватывает исключения типа SQLError

Выполняет команду INSERT в базе данных

Извлекает идентификатор последней вставленной строки

Преобразует идентификатор строки в Int и возвращает его

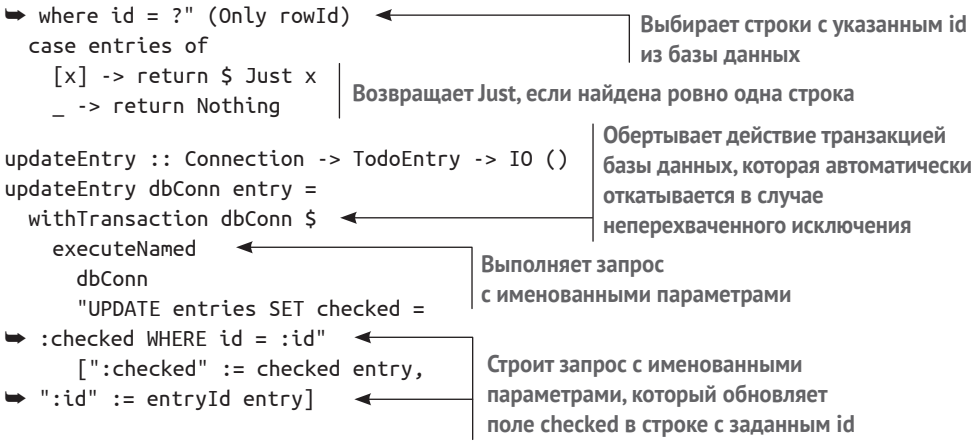
Обертывает действие транзакцией базы данных, которая автоматически откатывается в случае неперехваченного исключения

Мы возвращаем исключение `SQLError`, если оно возникло в процессе выполнения, чтобы у вызывающей стороны было больше информации для показа пользователю. Если же исключения не было, то мы возвращаем идентификатор `id` последней вставленной строки. Поскольку в команде `CREATE TABLE` поле `id` было определено как `INTEGER PRIMARY KEY`, то `lastInsertRowId` будет возвращать именно это поле последней вставленной строки. Поэтому мы знаем, какой идентификатор был присвоен записи, если он не был явно указан в значении `TodoEntry`. Функция `lastInsertRowId` имеет тип `Connection -> Int64`, поэтому мы должны преобразовать его в `Int`.

И напоследок нам нужен способ сбросить поле `checked` в одной записи. Для этого мы определим функцию, которая выбирает запись по `id`, и функцию, которая обновляет поле `checked` в записи с `id`, заданным в значении типа `TodoEntry`. Тогда наш API сможет сначала проверить, что запись, соответствующая `TodoEntry`, существует, а затем обновит ее. Обе функции показаны в листинге ниже.

Листинг 16.11 Функции выборки записи `TodoEntry` из базы данных и ее обновления

```
getEntryById :: Connection -> Int -> IO (Maybe TodoEntry)
getEntryById dbConn rowId = do
  entries <- query dbConn "SELECT * from entries
```



Здесь мы видим, как функция `executeNamed` используется для построения запроса с параметрами, взятыми из значения типа `TodoEntry`. Параметры обозначаются префиксом `:` или `@`. Если параметр встречается в запросе, но отсутствует в данных, то функция возбуждает исключение `FormatError`.

И чтобы завершить написание модуля `Todos.Database`, экспортируем функции, а также модуль `Database.SQLite.Simple`, дабы иметь доступ к типу `Connection` и прочим в коде, который импортирует `Todos.Database`. Заголовок модуля должен выглядеть следующим образом:

```
{-# LANGUAGE OverloadedStrings #-}

module Todos.Database
  ( module Database.SQLite.Simple,
    setupDb,
    getEntries,
    addEntry,
    getEntryById,
    updateEntry,
  )
where

import Control.Exception (try)
import Database.SQLite.Simple
import Todos.Types
```

Позаботившись о базе данных, мы можем перейти к сердцу нашего API. Этим мы займемся в следующей главе, где воспользуемся пакетом `servant`.

Резюме

- Пакет `aeson` помогает определять парсеры и сериализаторы для данных в формате JSON.
- `ToJSON` и `FromJSON` – классы типов, которые используются соответственно для сериализации и разбора JSON-данных.
- Класс типов `Generic`, который можно вывести автоматически, помогает выводить другие классы типов (в частности, `ToJSON` и `FromJSON`).
- Вывод с применением `Generic` создает реализации классов типов, зависящие от определения типа данных.
- `ToField`, `ToRow` и `FromRow` – классы типов, обеспечивающие сериализацию и разбор данных в базе.
- `Generic` можно использовать, чтобы вывести классы типов для взаимодействия с базой данных (`ToField`, `ToRow` и `FromRow`).
- Вставка строк в базу данных может привести к исключению `SQLError`, если не удовлетворяется какое-то ограничение, например первичного ключа.

17 Построение API с применением Servant

Краткое содержание этой главы:

- разработка типобезопасных API с применением Servant;
- реализация WAI-приложения;
- автоматический вывод клиентов API.

В предыдущей главе мы обсудили, как преобразовать данные Haskell в формат JSON и обратно и как хранить их в базе данных. Мы также написали несколько типов, которые собираемся использовать в API списка задач.

В этой главе мы закончим работу над проектом, но сначала изучим экосистему Servant, состоящую из библиотек и пакетов, которая позволит нам определить типобезопасный API и сервер для него. Затем мы расширим сервер, так чтобы он мог отдавать клиенту HTML-код. И наконец, напишем обертку, которая автоматически выведет клиентское приложение для сервера API.

17.1 Определение типобезопасного API

Для своего сервера мы напишем API на основе HTTP и JSON, т. е. будем использовать HTTP в качестве протокола, а JSON – в качестве формата сериализации данных. С JSON мы уже разобрались, теперь

же приспело время заняться сетью. Есть много способов работы с сетью, но нас не интересует низкий уровень – мы хотим определять API, а не возиться с докучными деталями сетевого взаимодействия. Существует длинный список библиотек и пакетов, готовых нам помочь, но мы выберем хорошо известную и поддержанную индустрией библиотеку, знаменитую своей типобезопасностью: *Servant*.

Servant – не просто библиотека, а семейство пакетов для моделирования API, автоматического генерирования клиентов этого API и даже генерирования документации. Он позволяет сначала описать общую «сигнатуру» (какие есть конечные точки, какие используются методы и для каких данных), а затем предоставить реализацию, согласованную с этим описанием.

Прежде всего добавим Servant в проект. Поскольку зависимостей понадобится много, приведем сразу полный перечень:

```
dependencies:
- base >= 4.7 && < 5
- aeson
- sqlite-simple
- bytestring
- text
- wai
- warp
- http-types
- http-client
- servant
- servant-server
- servant-client
- blaze-markup
- blaze-html
- optparse-applicative
```

В список зависимостей включены также `aeson` и `sqlite-simple`, но сейчас нам гораздо важнее пакеты, относящиеся к сети. Мы рассмотрим их по ходу дела.

17.1.1 Servant и типизированные API

Сначала рассмотрим объявление API в Servant. Создадим новый модуль `Todos.Api`, в котором будут определены конечные точки API с указанием принимаемых ими глаголов HTTP и описанием возвращаемых данных. Этот API кодируется одним типом, который мы построим из нескольких *операторов типа*, позволяющих объединять несколько типов в один. Чтобы создать такое определение, нам понадобятся два расширения языка: `DataKinds` и `TypeOperators`. Мы не будем вдаваться в детали их работы, поскольку для использования библиотеки это не важно.

При создании API используются операторы (:>) и (:<|>). Первый объединяет информацию о пути и запросе в описание одной подпрограммы API. Второй объединяет эти описания подпрограмм в более крупное определение библиотеки. Начнем с простого примера:

```
type Api = "simple" :> "example" :> GetNoContent
```

Этот тип описывает API с единственной конечной точкой, которой соответствует маршрут /simple/example. Она обслуживает HTTP-глагол GET, и после успешной обработки запроса никакое содержимое не доставляется. Но предположим, что мы хотим доставить в качестве ответа JSON-данные. Тогда их нужно закодировать типом:

```
type Api = "simple" :> "example" :> Get '[JSON] SomeType
```

Теперь наш API должен будет вернуть тип SomeType, представленный в формате JSON. Хотя библиотека позволяет возвращать не только JSON-ответ, нам эта возможность не понадобится.

Если API состоит из нескольких конечных точек, то для их объединения используется оператор (:<|>):

```
type TodoApi =
  "simple" :> "example" :> Get '[JSON] SomeType
    :<|> "another" :> "example" :> Get '[JSON] AnotherType
```

Этот API обслуживает две конечные точки. GET-запросы к /simple/example возвращают тип SomeType в формате JSON, а GET-запросы к /another/example – тип AnotherType, тоже в формате JSON. Таким образом, мы можем объединять сколько угодно конечных точек в один большой API.

До сих пор мы видели только конечные точки для GET-запросов, но можно определять их и для других глаголов HTTP: POST, PATCH, DELETE и прочих. Для POST-запросов нужно указать ожидаемый вид тела клиентского запроса. Это делается с помощью модификатора ReqBody:

```
type Api = "send" :> ReqBody '[JSON] SomeType :> PostNoContent
```

Здесь требуется, чтобы JSON-представление SomeType было отправлено в конечную точку /send, а результатом будет пустой ответ. Если тело запроса не отвечает спецификации, то Servant автоматически отклонит запрос.

Иногда нам нужно выделить часть маршрута в описании точки входа. Ее можно *запомнить* с помощью модификатора Capture:

```
type Api = "select" :> Capture "id" String :> Get '[JSON] SomeType
```

Если в запросе был указан маршрут /select/this, то запомненной частью будет this. Эта часть будет передана нашей реализации, если маршрут запроса соответствует типу.

> **ПРИМЕЧАНИЕ** Servant допускает еще много модификаторов маршрута, например Summary и Description для документирования конечных точек. Мы в этой главе рассмотрим только те, что нам необходимы. Полный перечень см. в документации по адресу https://hackage.haskell.org/package/servant.

Вооруженные этими знаниями, мы можем приступить к определению API. Повторим, что нам нужны три конечные точки для выполнения следующих действий:

- получить список всех записей;
- добавить запись;
- обновить одну запись.

При получении списка записей сервер должен вернуть все имеющиеся в базе строки `TodoEntry`, так что маршруту соответствует тип `[TodoEntry]`. На наше счастье, существует экземпляр `FromJSON` для списков, элементы которых также реализуют `FromJSON`, поэтому делать лишнюю работу нам не придется. Мы посылаем `GET`-запросы конечной точке `/entries`. Описание этой точки выглядит так:

```
"entries" :> Get '[JSON] [TodoEntry]
```

Конечная точка, добавляющая новую запись в список, должна отвечать на `POST`-запросы, потому что на сервере создается новая сущность. Данные записи следует отправлять в теле запроса в виде значения `TodoEntry` в формате JSON. В случае успешного создания ресурса мы возвращаем в ответе `id` вновь созданной записи и код HTTP-состояния 201, означающий, что ресурс был успешно создан. Это можно закодировать в типе `PostCreated`. Вызов конечной точки `/add` описывается так:

```
"add" :> ReqBody '[JSON] TodoEntry :> PostCreated '[JSON] Int
```

Дошла очередь и до обновления записи. В этой главе мы ограничимся только изменением значения поля `checked`, т. е. разрешим помечать задачу как выполненную или невыполненную. Но изменять, к примеру, текстовое описание задачи мы не позволяем. Чтобы изменить поле `checked`, клиент отправляет `PATCH`-запрос конечной точке, которая выделяет `id` из URL-адреса. Для этого мы можем определить конечную точку `/toggle/<id>`, где `<id>` – маркер идентификатора подлежащей обновлению записи. Серверу не нужно ничего посылать в ответ, если запрос был успешен:

```
"toggle" :> Capture "id" Int :> PatchNoContent
```

Здесь мы говорим, что нужно запомнить часть `id` маршрута к конечной точке в виде `Int`. Если это невозможно, например потому что идентификатор содержит буквы, то Servant автоматически отклонит

запрос. Собирая все конечные точки вместе, приходим к следующему определению API.

Листинг 17.1 Тип для сервера API списка задач

```
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE DataKinds #-}
module Todos.Api where

import Servant.API
import Todos.Types

type TodoApi =
  "entries" :> Get '[JSON] [TodoEntry]
    :<|> "add" :> ReqBody '[JSON] TodoEntry :>
➥ PostCreated '[JSON] Int
    :<|> "toggle" :> Capture "id" Int :> PatchNoContent
```

Активирует необходимые расширения языка

Определяет конечную точку /entries, которая выбирает все записи TodoEntry

Определяет конечную точку /add, которая добавляет новые записи с помощью POST-запроса

Определяет конечную точку /toggle, которая помечает задачу как выполненную или невыполненную, получив в URL ее id

Впоследствии нам понадобится так называемое значение `Proxy` нашего API. Этот тип и его конструктор можно импортировать из модуля `Data.Proxy`, а поскольку он весьма интересен, мы посвятим ему отдельный раздел.

17.1.2 Фантомные типы

Грубо говоря, определение типа `Proxy` выглядит так:

```
data Proxy a = Proxy
```

Не особенно вдохновляет, правда? Этот тип имеет всего один конструктор. Но что делает переменная `a` в его определении? У конструктора `Proxy` нет никаких полей, поэтому `a` не может встречаться в правой части определения. Это называется *фантомным типом*. Переменная `a` будет известна системе типов Haskell, но не связана ни с какими значениями во время выполнения:

```
ghci> data Proxy a = Proxy
ghci> :t Proxy
Proxy :: Proxy a
ghci> :t Proxy :: Proxy Float
Proxy :: Proxy Float :: Proxy Float
ghci> :t Proxy :: Proxy [Int]
Proxy :: Proxy [Int] :: Proxy [Int]
```

И чем же полезен такой тип? Мы можем использовать его для передачи информации о типе функциям, предоставив значение `Proxy` с подходящим фантомным типом:

```
readProxy :: Read a => Proxy a -> String -> a
readProxy _ = read

intProxy :: Proxy Int
intProxy = Proxy

stringProxy :: Proxy String
stringProxy = Proxy
```

Обратите внимание, что мы полностью игнорируем значение типа `Proxy` в функции `readProxy`. При таких определениях мы можем разбирать различные данные в зависимости от типа `Proxy`:

```
ghci> :t readProxy stringProxy
readProxy stringProxy :: String -> String
ghci> :t readProxy intProxy
readProxy intProxy :: String -> Int
ghci> readProxy stringProxy "\"abcde\""
"abcde"
ghci> readProxy intProxy "\"abcde\""
*** Exception: Prelude.read: no parse
```

Разобравшись с этой деталью, мы можем определить `Proxy` для своего API, поскольку Servant требует этого. Полный код модуля, включающий наш тип и `Proxy`, приведен в листинге ниже.

Листинг 17.2 Модуль, определяющий тип нашего Servant API и значение Proxy

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
```

Активирует расширения языка, необходимые для определения типа TodoApi

```
module Todos.Api
( TodoApi,
  todoApi,
)
where

import Data.Proxy (Proxy (..))
import Servant.API
import Todos.Types (TodoEntry)
```

Импортирует необходимые типы и конструкторы

```
type TodoApi =
  "entries" :> Get '[JSON] [TodoEntry]
    :<|> "add" :> ReqBody '[JSON] TodoEntry :>
➥ PostCreated '[JSON] Int
    :<|> "toggle" :> Capture "id" Int :>
➥ PatchNoContent
```

Определяет тип для серверного API списка задач

```
todoApi :: Proxy TodoApi
todoApi = Proxy
```

Определяет значение Proxy для типа TodoApi

Определив тип нашего API, мы готовы его реализовать.

17.1.3 Реализация API

Для реализации мы можем определить список обработчиков, которые будут отвечать на запросы. Servant знает два типа, используемых при определении реализации:

- `Handler` – действие для обработки одной конечной точки в определении API;
- `Server` – объединение обработчиков для обслуживания полной спецификации API.

Мы поместим все определения в новый модуль `Todos.Handlers`. Ему необходимо импортировать довольно много модулей, поэтому выпишем их прямо сейчас, чтобы не возвращаться к этому позже.

```
import Control.Monad.IO.Class
import qualified Data.ByteString.Lazy as BSL
import Data.String (fromString)
import qualified Data.Text as T
import qualified Data.Text.Encoding as T
import Servant
import qualified Text.Blaze.Html5 as H
import Todos.Api
import Todos.Database
import Todos.Types
```

Еще нам понадобятся следующие расширения языка:

```
{-# LANGUAGE NamedFieldPuns #-}
{-# LANGUAGE OverloadedStrings #-}
```

Теперь подумаем об обработчике для конечной точки `/entries`. Ему нужно будет вернуть значение типа `[TodoEntry]`, потому что преобразование в формат JSON автоматически выполняет Servant. Выбрать список `[TodoEntry]` из базы данных просто – это результат действия `getEntries` из модуля `Todos.Database`. Нам нужно только передать `getEntries` значение типа `Connection` и встроить действие ввода-вывода нашей базы данных в действие `Handler`. Поскольку все обработчики должны будут взаимодействовать с базой данных, имеет смысл передавать им действия ввода-вывода, которые уже имеют доступ к подключению. Мы сконструируем их в одном определении позже.

Пока что предположим, что наш обработчик получает действие `IO [TodoEntry]`, так что его полная сигнатура имеет вид:

```
entriesHandler :: IO [TodoEntry] -> Handler [TodoEntry]
```

Как преобразовать `IO` в `Handler`? Мы видели похожие встраивания действий ввода-вывода в другие монады в главе 14, когда обсуждали

`liftIO` и `MonadIO`. Оказывается, что `MonadIO Handler` уже существует, поэтому простая `liftIO` решает проблему преобразования запроса к базе данных в обработчик для нашего API.

Листинг 17.3 Обработчик, возвращающий список значений типа `TodoEntry`

```
entriesHandler :: IO [TodoEntry] -> Handler [TodoEntry]
entriesHandler = liftIO
```

Поднимает действие ввода-вывода в монаду Handler

Чтобы превратить это в `Handler`, которым может воспользоваться Servant, все аргументы должны быть применены, так что фактическое определение этого обработчика больше похоже на такое:

```
entriesHandler (getEntries dbConn)
```

Поскольку `dbConn` имеет тип `Connection`, мы можем предоставить обработчику `entriesHandler` подходящее действие ввода-вывода. Остальные обработчики определяются аналогично. Таким образом, мы гарантируем, что обработчики не смогут непреднамеренно взаимодействовать с подключением к базе данных, а смогут использовать лишь переданные им действия. Дополнительное преимущество такого абстрагирования в том, что становится проще создавать объекты-подставки вместо реальных баз данных, так как действия ввода-вывода могут взаимодействовать с файлами, а не с настоящей базой.

Теперь определим обработчик для добавления новой записи `TodoEntry`. Поскольку мы потребовали, чтобы тело запроса содержало `TodoEntry` в формате JSON, обработчик должен принимать значение этого типа в качестве аргумента. Поэтому Servant ожидает от нас тип `TodoEntry → Handler Int`. Однако мы должны еще предоставить действие ввода-вывода для взаимодействия с базой данных. Подойдет действие `addEntry`, имеющее тип `Connection → TodoEntry → IO (Either SQLError Int)`. Частичное применение `Connection` и передача результирующего действия обработчику приводят к выводу, что окончательный тип обработчика имеет вид:

```
addHandler ::
  (TodoEntry -> IO (Either SQLError Int)) ->
  TodoEntry ->
  Handler Int
```

В случае если наше действие возвращает ошибку, мы должны сообщить об этом клиенту. Внутри монады `Handler` нам предоставляется действие `throwError`, которое позволяет прервать обработку с ошибкой. Кроме того, Servant предлагает тип `ServerError`, который можно возбудить в виде исключения, используя это действие:

```
ghci> import Servant
ghci> :i ServerError
type ServerError :: *
data ServerError
  = ServerError {errHTTPCode :: Int,
                 errReasonPhrase :: String,
                 errBody :: ByteString,
                 errHeaders :: [Header]}
```

Он дает возможность задать подходящий HTTP-код ответа, а также заголовки, тело и причину. В библиотеке имеется еще много готовых ошибок, где уже проставлены код и причина:

```
ghci> err403
ServerError {errHTTPCode = 403, errReasonPhrase = "Forbidden",
➥ errBody = "", errHeaders = []}
ghci> err404
ServerError {errHTTPCode = 404, errReasonPhrase = "Not Found",
➥ errBody = "", errHeaders = []}
ghci> err418
ServerError {errHTTPCode = 418, errReasonPhrase = "I'm a teapot",
➥ errBody = "", errHeaders = []}
ghci> err500
ServerError {errHTTPCode = 500, errReasonPhrase = "Internal Server Error",
➥ errBody = "", errHeaders = []}
```

Однако, как мы видим, тело и заголовки пусты. Это позволяет включить в тело дополнительную информацию, чтобы клиент знал, что случилось.

А как нам превратить `SQLError` в нечто, понятное человеку? Оказывается, что в этом типе уже имеется описание ошибки в поле `sqlErrorDetails`:

```
ghci> import Database.SQLite.Simple
ghci> :i SQLError
type SQLError :: *
data SQLError
  = SQLError {sqlError :: Error,
              sqlErrorDetails :: Text,
              sqlErrorContext :: Text}
```

Так что мы можем взять это описание и включить в него текст сообщения об ошибке. Полный код приведен в листинге ниже.

Листинг 17.4 Обработчик, который добавляет запись `TodoEntry` и возвращает содержательные сообщения об ошибках

```
addHandler ::
  (TodoEntry -> IO (Either SQLError Int)) ->
  TodoEntry ->
```

```
  Handler Int
addHandler addAct entry = do
  mSqlError <- liftIO $ addAct entry
  case mSqlError of
    Right idx -> pure idx
    Left exc ->
      let errBody =
            BSL.fromStrict
              (T.encodeUtf8 $ sqlErrorDetails exc)
       in throwError $ err422 {errBody}
```

Поднимает действие добавления записи в монаду Handler и вычисляет его для аргумента типа TodoEntry

Если исключения не возникло, то возвращает пустой ответ

Кодирует описание ошибки в виде ленивой строки байтов

Возбуждает исключение с кодом состояния 422 и описанием ошибки в теле

Как и раньше, мы используем `liftIO`, чтобы поднять действие ввода-вывода в монаду. Поскольку детальные сведения об ошибке `SQLError` заданы как `Text`, мы должны сначала преобразовать их в `ByteString`. Однако Servant требует ленивую `ByteString`, потому-то мы и используем для преобразования `fromStrict`. Мы остановились на коде состояния 422, который означает, что запрос был правильно понят, но не мог быть обработан. Это случается, когда идентификатор `id` записи `TodoEntry`, переданной в запросе, уже присутствует в базе данных.

Остался последний обработчик, который переключает атрибут `checked` на противоположный. Не существует одного запроса к базе данных, который выполнил бы это действие. Однако у нас есть действие, которое выбирает запись по `id`, и действие, которое обновляет эту запись.

Оба этих действия нужно передать обработчику в качестве аргументов. Кроме того, поскольку в маршруте к конечной точке присутствует часть `Capture "id" Int`, обработчику нужен еще аргумент типа `Int`. А возвращать он должен специальный тип `NoContent`, потому что мы сказали, что конечная точка не возвращает никаких данных в теле ответа. Тип должен выглядеть следующим образом:

```
toggleHandler ::
 (Int -> IO (Maybe TodoEntry)) ->
 (TodoEntry -> IO ()) ->
 Int ->
 Handler NoContent
```

Реализация прямолинейна. Если запись `TodoEntry` с заданным `id` не найдена, то мы возвращаем классический HTTP-код состояния 404, присовокупив в описании, что `id` не найден. В противном случае поле `checked` в записи `TodoEntry` меняется на противоположное и передается нашему действию обновления, как показано в следующем листинге.

Листинг 17.5 Обработчик, который изменяет поле checked в записи TodoEntry на противоположное

```
toggleHandler ::
  (Int -> IO (Maybe TodoEntry)) ->
  (TodoEntry -> IO ()) ->
  Int ->
  Handler NoContent
toggleHandler queryAct updateAct rowId = do
  mEntry <- liftIO $ queryAct rowId
  case mEntry of
    Nothing ->
      let errBody =
            fromString $
              "Couldn't find entry with id \""
                <> show rowId
                <> "\""
       in throwError $ err404 {errBody}
    Just entry -> do
    let newEntry = entry
➥ {checked = not $ checked entry}
    liftIO $ updateAct newEntry
    pure NoContent
```

Выбирает запись TodoEntry с заданным id

Возбуждает исключение с HTTP-кодом состояния 404, если запись не найдена

Меняет поле checked в найденной записи на противоположное

Выполняет обновление записи TodoEntry

Возвращает пустое содержимое

В этом определении мы используем функцию `fromString` для создания ленивой `ByteString` из обычной `String`. `NoContent` – единственное значение типа `NoContent`, в этом контексте оно более понятно, чем `()`.

Определив обработчики, мы можем определить `Server` для нашего типа `TodoApi`. Как уже отмечалось, конструктору этого сервера нужно передать подключение к базе данных `Connection`, чтобы определить действия, которые мы передаем обработчикам. Сделав это, мы должны объединить обработчики с сервером. Это делается с помощью уже знакомого нам оператора `(:<|>)`. Тот же оператор, который мы использовали на *уровне типа*, чтобы объединить описания конечных точек в полное описание API, используется и на *уровне термов*, чтобы присоединить обработчики к серверу.

Листинг 17.6 Функция, которая создает Server для TodoApi

```
mkTodoApiServer :: Connection -> Server TodoApi
mkTodoApiServer dbConn =
  entriesHandler getEntries'
    :<|> addHandler addEntry'
    :<|> toggleHandler getEntryById' updateEntry'
```

Объединяет обработчики в полную реализацию сервера

```
  where
    getEntries' = getEntries dbConn
    addEntry' = addEntry dbConn
    getEntryById' = getEntryById dbConn
    updateEntry' = updateEntry dbConn
```

Частично применяет действия базы данных, передавая обязательный аргумент Connection

Удивительно в этом определении то, что все определения типов Servant статически проверяются для предоставленной нами реализации, т. е. компилятор автоматически отвергнет реализацию, если мы забыли реализовать какую-то конечную точку, приняли некорректные параметры или возвращали неверные типы данных. Если эта функция компилируется, то мы можем быть уверены, что реализация отвечает спецификации, заданной в типе `TodoApi`. К тому же значительную часть трудной работы проделал за нас Servant. Отметим, что нам не пришлось задавать HTTP-коды состояния (кроме как в случае ошибок) и вручную преобразовывать данные в формат JSON и обратно. Все это поведение за нас уже выведено. А нам нужно просто определить тип API и реализовать бизнес-логику. Библиотека видит информацию о типе и автоматически выводит из него поведение API, уступая управление нашему коду только при необходимости (чтобы выполнить бизнес-логику).

17.2 Запуск приложения

Теперь, когда реализация API по существу закончена, мы хотим собрать и запустить исполняемый файл. Этот проект по-своему уникален, т. к. мы хотим собрать два независимых исполняемых файла с одной и той же библиотекой: один для сервера, другой для клиента. Поэтому изменим секцию `executables` в файле `package.yml` следующим образом:

```
executables:
  todolist-server:
    main:            Server.hs
    source-dirs:     app
    ghc-options:
    - -main-is Server
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    dependencies:
    - todolist
```

Это позволит нам использовать файл `Server.hs` в каталоге `app` для сборки сервера. Исполняемый файл будет называться `todolist-server`. Важно, что мы используем параметр GHC `-main-is`, который

говорит компилятору, что главный модуль называется `Server`. В противном случае в файле `Server.hs` нужно было бы создать модуль с именем `Main`, не совпадающим с именем файла.

Теперь можно удалить файл `Main.hs` и создать новый файл `Server.hs` в каталоге `app`. Секция импорта в нем будет выглядеть следующим образом:

```
module Server (main) where

import Network.Wai
import Network.Wai.Handler.Warp
import Servant
import Todos.Api
import Todos.Database
import Todos.Handlers
```

Далее можно определить функцию, которая преобразует наш `Server TodoApi` в интерфейс веб-приложений `Application`.

17.2.1 WAI-приложение

Интерфейс веб-приложений определяет типы и функции, необходимые веб-каркасам для взаимодействия с реализациями веб-серверов. Самая распространенная реализация сервера, предназначенная для использования совместно с WAI, – Warp, мы импортировали ее в свой файл. Скоро мы изучим возможности этого интерфейса, а пока нас интересует подключение нашего Servant `Server` к Warp, необходимое для запуска сервера. С этой целью Servant предоставляет функцию `serve`, которая принимает `Proxy` нашего API и `Server` и порождает приложение.

Листинг 17.7 Функция, которая создает WAI-приложение `Application` по реализации API

```
mkApiServer :: Connection -> Application
mkApiServer = serve todoApi . mkTodoApiServer
```

Создает Server, используя заданное подключение Connection, и преобразует его в Application

Теперь можно написать действие `main`, которое наконец-то запускает наш сервер:

```
main :: IO ()
main = do
  let port = 9001
  withConnection ":memory:" $ \dbConn -> do
    setupDb dbConn
    putStrLn $
```

```
    "Running todo list server on port " ++ show port
  run port (mkApiServer dbConn)
```

Поскольку мы тестируем приложение, используется фиксированный порт, прослушиваемый сервером, а база данных размещается в памяти, чтобы не нужно было ничего сохранять между перезапусками. Благодаря функции `withConnection` мы можем запомнить `Connection`, которое потом будет автоматически закрыто. Сначала мы настраиваем базу данных, затем начинаем исполнять наш API. Запустить приложение можно командой `stack run todolist-server`. Мы должны увидеть приветствие `Running todo list server on port 9001`.

Теперь можно воспользоваться каким-нибудь инструментом для взаимодействия с нашим API. Поклонники терминала могут взять для этой цели утилиту `curl`, в этом нет ничего плохого. Мы же будем использовать *Restman*, расширение для браузеров на базе движка Chromium, поскольку оно простое и кросс-платформенное. Оно позволяет отправлять запросы нашему API и красиво форматирует полученные от API JSON-ответы.

Можно отправить простой `GET`-запрос локальному узлу `localhost` в порт `9001` конечной точке `/entries`. Результатом будет пустой JSON-список – это ожидаемо и показано на рис. 17.1.

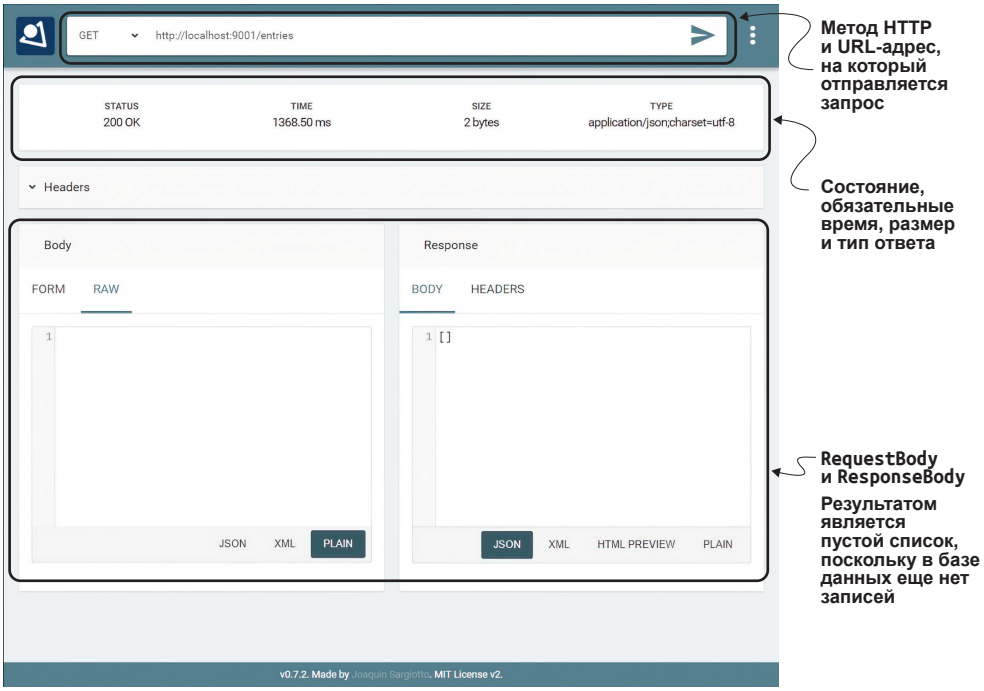


Рис. 17.1 Get-запрос к конечной точке /entries при пустой базе данных

Теперь добавим запись. Мы должны будем переключиться с `GET` на `POST` и отправить запрос конечной точке `/add`. Кроме того, понадобится присвоить заголовку `Content-Type` значение `application/json`. Потом можно отправлять записи, как показано на рис. 17.2.

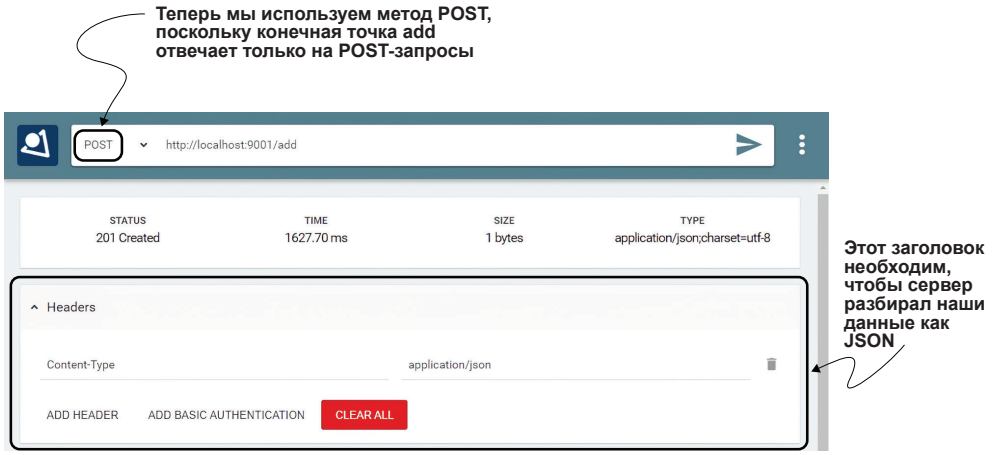


Рис. 17.2 Обязательные данные заголовка для POST-запроса к конечной точке /add

Как видим, ответ содержит 1, поскольку именно этот `id` использовался для сохранения записи в базе данных. Теперь добавим еще одну запись, но на этот раз зададим `id` и `rank`, как показано на рис. 17.3.

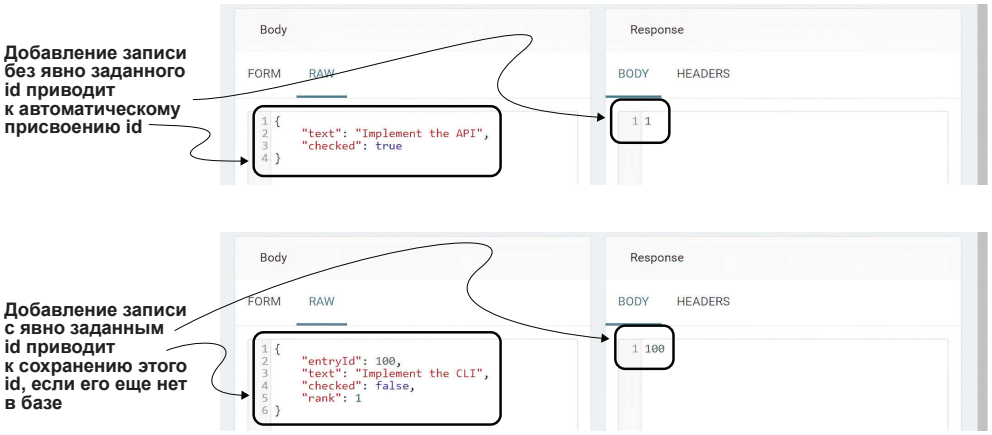


Рис. 17.3 POST-запросы к конечной точке /add без явно заданного id и с явно заданным id

При попытке добавить еще одну запись с таким же `id` мы получаем ответ с кодом 422 и сообщением об ошибке (`UNIQUE constraint failed: entries.id`). Отправка некорректно сформированного JSON-

запроса конечной точке также приводит к ошибке, она обрабатывается автоматически. Вот как выглядит полный ответ с ошибкой, если отсутствует ключ `checked`:

```
Error in $: parsing Todos.Types.TodoEntry(TodoEntry) failed,
➥ key "checked" not found
```

Теперь можно снова извлечь сохраненные записи. Повторный `GET`-запрос к конечной точке `/entries` даст следующий результат:

```
[
  {
    "checked": true,
    "entryId": 1,
    "rank": null,
    "text": "Implement the API"
  },
  {
    "checked": true,
    "entryId": 100,
    "rank": 1,
    "text": "Implement the API"
  }
]
```

Теперь мы можем менять состояние задачи, как показано на рис. 17.4.

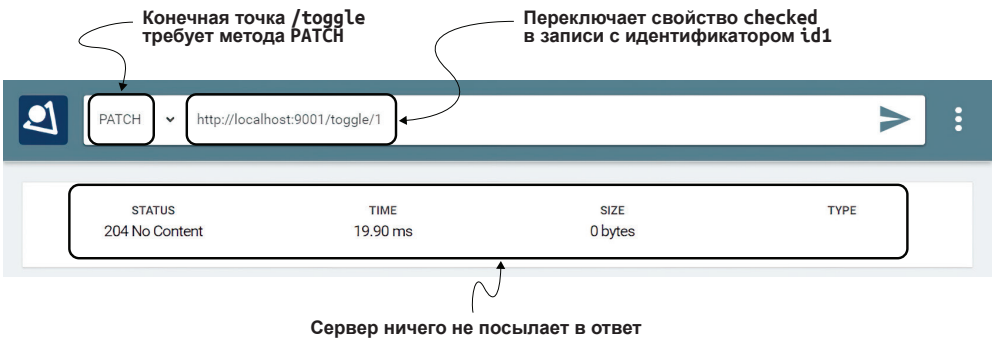


Рис. 17.4 `PATCH`**-запрос к конечной точке** `/toggle` **для переключения свойства** `checked` **в записи с идентификатором** `id1`

Опросив записи, легко убедиться, что поле было изменено:

```
[
  {
    "checked": true,
    "entryId": 1,
    "rank": null,
```

```
    "text": "Implement the API"
  },
  {
    "checked": true,
    "entryId": 100,
    "rank": 1,
    "text": "Implement the API"
  }
]
```

Отлично! Мы успешно создали сервер API, который можем опрашивать по протоколу HTTP/JSON. Но ведь мы работаем с веб-сервером, так, может быть, можно передать пользователю еще и HTML-страницу?

> **Упражнение. Удаление записей**
>
> Мы реализовали конечные точки для создания, чтения и обновления записей. Чтобы CRUD API был полным, нужно еще научиться удалять записи. Реализуйте такую конечную точку самостоятельно! Имеет смысл смоделировать ее так же, как конечную точку `/toggle`. URL-адрес должен иметь вид `/delete/<id>`.

Теперь мы хотим реализовать еще одну конечную точку, не входящую в API; она будет выводить список задач.

17.2.2 Средний уровень приложения

Наш сервер располагает пригодным к использованию JSON API, но ему не хватает интерфейса, ориентированного непосредственно на человека. Мы хотим исправить это упущение и для этого будем возвращать список задач в виде HTML-страницы в ответ на обращение к корневой конечной точке (/). Но нужно принять проектное решение о том, как смоделировать эту конечную точку.

Вообще говоря, вариантов два. Можно было бы смоделировать ее в Servant API, указав тип `HTML` вместо `JSON`. Но поскольку впоследствии мы собираемся вывести клиента для этого API, неразумно заставлять клиента запрашивать HTML-страницу. Поэтому мы предпочитаем оставить отрисовку HTML-страницы за пределами нашего API.

Но как вставить дополнительный маршрут в веб-сервер, если трогать API мы себе запретили? Поскольку Servant порождает WAI-приложение `Application`, нам нужен какой-то способ модифицировать его. Что вообще такое WAI `Application`? Тип довольно простой:

```
type Application = Request -> (Response -> IO ResponseReceived) ->
➥ IO ResponseReceived
```

`Application` – простая функция, которая принимает функции обратного вызова для обработки запроса и ответа. Желая реализовать эту функцию, мы можем принять `Request`, вычислить `Response` и вызвать функцию обратного вызова для отправки ответа. Желая модифицировать `Application`, мы должны обернуть одну функцию другой. Отображение `Application` в `Application` называется ПО среднего уровня – `Middleware`:

```
type Middleware = Application -> Application
```

При определении `Middleware` стоит начать с трех аргументов:

```
myMiddleware :: Middleware
myMiddleware app req responseAct = ...
```

Первый – обертываемое `Application`, второй и третий – аргументы, принимаемые `Application`. Так мы получаем тип `Application -> Application`.

Наше `Middleware` должно проверять, является ли `Request` `GET`-запросом к корню приложения (`/`). Если да, то мы возвращаем свой HTML-код. В противном случае мы просто используем обернутое значение `app` (т. е. наш сервер API). Извлечь интересующие нас атрибуты запроса позволяют функции `requestMethod` и `pathInfo`:

```
import Network.HTTP.Types (methodGet)

...

mkAppRoot :: Connection -> Middleware
mkAppRoot dbConn app req responseAct
  | isRootRequest = ...
  | otherwise = app req responseAct
  where
    isRootRequest =
      requestMethod req == methodGet
        && null (pathInfo req)
```

Теперь нужно сгенерировать ответную HTML-страницу. Сначала решим, в каком порядке отображать записи. Ведь у нас есть поле `rank` специально для этой цели! По нему и будем сортировать. Чем меньше `rank`, тем выше запись расположена в списке, при этом ранжированные записи располагаются перед всеми неранжированными. Увы, экземпляр `Ord` для `Maybe Int` располагает значения `Nothing` перед всеми значениями, обернутыми `Just`, поэтому придется написать функцию, которая разделяет записи по признаку наличия ранга и сортирует только ранжированные. Ее код, который мы добавим в модуль `Todos.Types`, показан в следующем листинге.

Листинг 17.8 Функция сортировки записей списка задач

```
import qualified Data.List as L
import qualified Data.Maybe as M

...

sortTodoEntries :: [TodoEntry] -> [TodoEntry]
sortTodoEntries entries = L.sortOn rank ranked
➥ ++ unranked
  where
    (ranked, unranked) = L.partition
➥ (M.isJust . rank) entries
```

Сортирует ранжированные записи и дописывает в конец неранжированные

Разделяет записи на ранжированные и неранжированные

Нам также необходимо импортировать модули `Data.List` и `Data.Maybe` под квалифицированными именами `L` и `M` соответственно. И теперь мы можем определить действие ввода-вывода, которое принимает `Connection` и возвращает HTML-код.

17.2.3 Порождение HTML-кода по данным

Чтобы построить HTML-код по значениям Haskell, мы прибегнем к библиотеке blaze-html. Она предлагает комбинаторы для конструирования HTML-документа с применением `do`-нотации.

Мы можем импортировать `Text.Blaze.Html5` с квалифицированным именем `H` в модуль `Todos.Handlers`, чтобы определить обработчик корневого URL. Работать с библиотекой нетрудно. Комбинаторы называются так же, как элементы HTML. С помощью `do`-нотации мы можем вкладывать комбинаторы, что будет отражено в результирующем HTML-коде:

```
H.body $ do
  H.h1 "Title"
  H.p "Foobar"
```

Это определение транслируется в такую разметку:

```
<body>
    <h1>Title</h1>
    <p>Foobar</p>
</body>
```

Также мы можем использовать функции, которые обычно использовали бы с монадами. Например, функцию `mapM_` для порождения списка элементов HTML из списка строк:

```
mkHtmlList xs = H.ul $ mapM_ (H.li . H.string) xs
```

Результатом вызова `mkHtmlList ["Foo", "Bar", "Baz"]` будет

```
<ul>
    <li>Foo</li>
    <li>Bar</li>
    <li>Baz</li>
</ul>
```

Имея эти комбинаторы, мы можем построить веб-страницу. Я рекомендую показанный ниже обработчик, но вы можете придумать свой собственный дизайн.

Листинг 17.9 Действие для получения всех записей списка задач и представления их в виде HTML-кода

```
rootHandler :: Connection -> IO H.Html
rootHandler dbConn = do
  entries <- sortTodoEntries <$> getEntries dbConn
  let listHtml
        | null entries = H.p $ H.text "No entry found"
        | otherwise = H.ul $ mapM_
➥ (H.li . mkTodoEntryHtml) entries
      fullHtml = H.docTypeHtml $ do
        H.head $ do
          H.title "TODOs"
          H.style $ H.text style
        H.body $ do
          H.h1 "TODOs"
          listHtml
  return fullHtml
  where
    style :: T.Text
    style =
      "body {font-family: sans-serif} \
      \ ul {list-style: none} \
      \ li {margin-bottom: 5px}"

    mkTodoEntryHtml :: TodoEntry -> H.Html
    mkTodoEntryHtml (TodoEntry _ text checked _) =
      let mark =
            if checked
              then "\x2713"
              else "\x2718"
       in H.text (mark <> " " <> text)
```

Извлекает и сортирует все записи

На случай, если не найдено ни одной записи

Отображает записи в виде маркированного списка

Определяет тип документа

Определяет заголовок HTML-документа, включающий название и стиль

Определяет тело HTML-документа, состоящее из заголовка списка и самого списка

Определяет стиль документа

Рисует либо галочку (U+2713), либо крестик (U+2718) в кодировке UTF-8

Присоединяет галочку или крестик к тексту записи

Заметим, что здесь используются литеры в кодировке UTF-8. Когда мы будем преобразовывать определение в данные, отправляемые в ответе, это важно иметь в виду.

Теперь мы можем закончить `Middleware`. Функция `rootHandler` вычисляет значение `HTML`, которое затем нужно преобразовать в `ByteString`. В модуле `Server` мы можем импортировать модуль `Text.Blaze.Html.Renderer.Utf8`, чтобы получить доступ к функции `renderHTML`, которая вычисляет байтовую строку в кодировке UTF-8. Чтобы включить ее в ответ, можно вызвать функцию `responseLBS` из модуля `Network.WAI`. И чтобы определение заработало, нужно импортировать еще несколько модулей.

```
import Network.HTTP.Types.Header
import Network.HTTP.Types.Method
import Network.HTTP.Types.Status
import qualified Text.Blaze.Html.Renderer.Utf8 as H
```

Окончательный код `Middleware` показан в следующем листинге.

Листинг 17.10 `Middleware` для представления списка задач в виде HTML-страницы в ответ на обращение к корню приложения

```
mkAppRoot :: Connection -> Middleware
mkAppRoot dbConn app req responseAct
  | isRootRequest = do
      html <- rootHandler dbConn
      responseAct $ responseLBS ok200 headers
➥ (H.renderHtml html)
  | otherwise = app req responseAct
  where
    headers = [(hContentType,
➥ "text/html; charset=utf-8")]
    isRootRequest =
      requestMethod req == methodGet
        && null (pathInfo req)
```

Вычисляет HTML для списка задач, прочитанного из базы данных

Генерирует HTML-код в виде ByteString в кодировке UTF-8, которая используется для построения ответа с HTTP-кодом состояния 200

Если был получен не GET-запрос, то мы передаем управление обернутому Application

Определяет заголовок, устанавливая тип содержимого text/html и кодировку utf-8

Проверяет, что в запросе к корню приложения указан метод GET

Важно отметить, что, модифицируя `Application` таким способом, мы теряем некоторые статические гарантии. Если наш Servant API впоследствии реализует конечную точку для `GET`-запросов к `/`, то мы не сможем гарантировать, что `Middleware` не вмешается. Однако поскольку наше приложение довольно простое, это не страшно, и, воспользовавшись `Middleware`, мы можем изменить прежнее действие `main`:

```
main :: IO ()
main = do
  let port = 9001
  withConnection ":memory:" $ \dbConn -> do
```

```
  setupDb dbConn
  putStrLn $
    "Running todo list server on port " ++ show port
  run port (mkAppRoot dbConn $ mkApiServer dbConn)
```

Допустим, что в базе данных имеются следующие записи, к которым мы можем получить доступ через конечную точку `/entries`:

```
[
  {
    "checked": true,
    "entryId": 1,
    "rank": null,
    "text": "Specify JSON and Database serialization"
  },
  {
    "checked": true,
    "entryId": 2,
    "rank": null,
    "text": "Specify API type"
  },
  {
    "checked": true,
    "entryId": 3,
    "rank": null,
    "text": "Implement API"
  },
  {
    "checked": true,
    "entryId": 4,
    "rank": null,
    "text": "HTML rendering for list"
  },
  {
    "checked": false,
    "entryId": 5,
    "rank": 2,
    "text": "CLI arguments"
  },
  {
    "checked": false,
    "entryId": 6,
    "rank": 1,
    "text": "Implement client application"
  }
]
```

На рис. 17.5 показана результирующая HTML-страница:

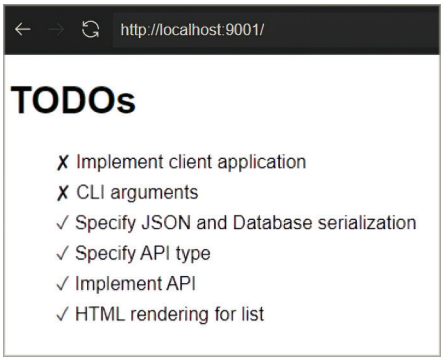


Рис. 17.5 Пример отрисовки корневой конечной точки, показывающей созданные записи списка задач

Заметим, что галочки и крестики на вашем компьютере могут выглядеть иначе – в зависимости от операционной системы и браузера. Добавив этот код в сервер, мы, наконец, можем закончить его, задав аргументы командной строки. Поскольку мы уже рассматривали пакет `optparse-applicative` в главе 14, не станем подробно обсуждать, как добавляются эти аргументы. Мы позволим пользователю указывать, к какому порту подключаться и какой файл базы данных использовать (по умолчанию используется база данных в памяти). Код приведен в листинге ниже.

Листинг 17.11 Аргументы командной строки для исполняемого файла сервера списка задач

```
import Options.Applicative

...

data CliArgs = CliArgs
  { cliArgsPort :: Int,
    cliArgsDatabaseFile :: String
  }

cliArgsParser :: Parser CliArgs
cliArgsParser = CliArgs <$> port <*> databaseFile
  where
    port =
      option auto $
        long "port"
          <> value 9001
          <> metavar "PORT"
          <> help "Port to connect to"
          <> showDefault
    databaseFile =
      strOption $
        long "database"
          <> value ":memory:"
```

Тип для аргументов командной строки сервера

Парсер номера порта, по умолчанию 9001

Парсер файла базы данных, по умолчанию :memory:

```
          <> metavar "PATH"
          <> help "Database file to use"
          <> showDefaultWith
            ( const
                "none, using in-memory database"
            )

argumentParser :: ParserInfo CliArgs
argumentParser =
  info
    (cliArgsParser <**> helper)
    ( fullDesc
        <> progDesc "A todo list API server"
    )
```

Информация для парсера аргументов командной строки

Листинг 17.12 Действие `main` для сервера списка задач

```
main :: IO ()
main = do
  args <- execParser argumentParser
  let port = cliArgsPort args
  withConnection (cliArgsDatabaseFile args) $
➥ \dbConn -> do
    setupDb dbConn
    putStrLn $
      "Running todo list server on port " ++ show port
    run port (mkAppRoot dbConn $ mkApiServer dbConn)
```

Разбирает аргументы командной строки

Создает подключение к базе данных

Инициализирует базу данных

Генерирует и запускает Application, вызывая mkApiServer со средним уровнем mkAppRoot

Теперь протестируем сервер, используя локальную базу данных и порт по умолчанию:

```
shell $ stack run todolist-server -- -h
Usage: todolist-server [--port PORT] [--database PATH]

  A todo list API server

Available options:
  --port PORT              Port to connect to (default: 9001)
  --database PATH          Database file to use
                           (default: none, using in-memory database)
  -h,--help                Show this help text
shell $ stack run todolist-server -- --database todolist.db
Running todo list server on port 9001
```

Далее для тестирования сервера можно было бы снова воспользоваться Restman или `curl`. Но мы хотим реализовать клиент сервера API, который предоставит нам простую командную утилиту для взаимодействия с этим сервером.

17.3 Вывод клиента

Позаботившись о сервере, мы теперь создадим новый файл `Client.hs` в каталог `app`. И еще добавим исполняемый файл в секцию `executables` файла `package.yml`:

```
todolist-client:
  main:             Client.hs
  source-dirs:      app
  ghc-options:
  - -main-is Client
  - -threaded
  - -rtsopts
  - -with-rtsopts=-N
  dependencies:
  - todolist
```

Затем включим в новый файл следующий заголовок:

```
module Client (main) where

import qualified Data.Text as T
import Network.HTTP.Client
  ( defaultManagerSettings,
    newManager,
  )
import Options.Applicative
import Servant
import Servant.Client
import Todos.Api
import Todos.Types
```

17.3.1 Использование servant-client

Самый важный из импортируемых здесь модулей – `Servant.Client`, который находится в пакете `servant-client`. Он предоставляет тип `ClientM`, моделирующий действия клиентов, а также две функции. Первая из них – `client` – имеет тип `HasClient ClientM api => Proxy api -> Client ClientM api`. Получив `Proxy` для API, она *автоматически выводит* функции клиента этого API. После этого можно использовать функцию `runClientM` типа `ClientM a -> ClientEnv -> IO (Either ClientError a)`, которая выполняет монаду `ClientM` в монаде `IO`.

А как использовать `client`? Рассмотрим полный вывод клиента нашего API, показанный в листинге ниже, чтобы разобраться, как он устроен.

Листинг 17.13 Вывод функциональности клиента по Proxy API

```
getEntries :: ClientM [TodoEntry]
addEntry :: TodoEntry -> ClientM Int
toggleEntry :: Int -> ClientM NoContent
getEntries :<|> addEntry :<|> toggleEntry =
➥ client todoApi
```

Определяет типы действий клиента

Выводит определение действий клиента из Proxy API

Эти четыре строчки кода составляют полное определение нашего клиента! Сначала определяются типы действий. Обратите внимание на сходство с типом `TodoApi`. `"entries" :> Get '[JSON] [TodoEntry]` соответствует `ClientM [TodoEntry]`, поскольку результирующий тип один и тот же. `"add" :> ReqBody '[JSON] TodoEntry :> PostCreated '[JSON] Int` соответствует `TodoEntry -> ClientM Int`, т. к. запросу нужно значение `TodoEntry` для включения в тело, а значит, и действие требует этот тип в качестве аргумента. Снова отметим, что преобразование в формат JSON к определению клиента не имеет никакого отношения. Это Servant делает автоматически.

17.3.2 Определение команд

Теперь можно определить интерфейс командной строки для клиента. В этом приложении мы хотим ввести новое понятие – *команды*. До сих пор наши командные интерфейсы поддерживали параметры, но не команды, о которых мы знаем по таким программам, как git, где идентификатор после имени исполняемого файла открывает доступ к определенной части функциональности:

```
git pull --rebase
git commit -m "Cool work"
git push --force-with-lease
```

Параметры, которые здесь тоже присутствуют, допустимы и обретают смысл (возможно, различный), только если следуют за идентификатором команды. Наша программа будет поддерживать три команды:

- `list` – вывести список записей;
- `add` – добавить новые записи;
- `toggle` – изменить состояние записи (задача выполнена или не выполнена).

Дополнительно нужно добавить информацию о сервере и номере порта, к которому должен подключиться клиент. Сначала определим типы аргументов:

```
data CliArgs = CliArgs
  { cliArgsHost :: String,
    cliArgsPort :: Int,
    cliArgsCommand :: CliCommand
  }

data CliCommand
  = GetEntries
  | AddEntry TodoEntry
  | ToggleEntry Int
```

Далее определим тип `Parser` и соответствующий `ParserInfo` для `CliArgs`. Мы сознательно опускаем определения парсеров команд.

Листинг 17.14 Парсер аргументов командной строки клиента API

```
cliArgsParser :: Parser CliArgs
cliArgsParser =
  CliArgs
    <$> host
    <*> port
    <*> commandParser
  where
    host =
      strOption $
        long "host"
          <> value "localhost"
          <> metavar "HOST"
          <> help "host to connect to"
          <> showDefault
    port =
      option auto $
        long "port"
          <> value 9001
          <> metavar "PORT"
          <> help "port to connect to"
          <> showDefault
    commandParser =
      subparser
        ( command "list" getEntriesCommandParser
            <> command "add" addEntryCommandParser
            <> command "toggle" toggleEntryCommandParser
        )

argumentParser :: ParserInfo CliArgs
argumentParser =
  info
    (cliArgsParser <**> helper)
    ( fullDesc
        <> progDesc "A todo list API client"
    )
```

Определяет парсер имени хоста, по умолчанию localhost

Определяет парсер порта, по умолчанию 9001

Определяет парсер команды

Определяет команды: идентификатор команды и парсер для разбора последующих параметров

Определяет ParserInfo для интерфейса командной строки клиента

В листинге 17.14 мы видим, в частности, код `subparser`, который служит для создания вложенных парсеров команды, конструируемых с помощью функции `command`. Эта функция требует идентификатора (имени команды) и значения `ParserInfo` для разбора ее аргументов. Мы можем определить их так же, как и прежде, ничего особенного в них нет.

Реализации `getEntriesCommandParser` и `toggleEntryCommandParser` сравнительно просты и показаны в листинге 17.15. Обычно мы разделяем определения `Parser` и `ParserInfo`, но здесь объединили, потому что они очень короткие. Так как `getEntriesCommandParser` не имеет параметров, мы возвращаем сам конструктор `GetEntries`. Но `toggleEntryCommandParser` нуждается в параметре типа `Int` для идентификатора `id`.

Листинг 17.15 Парсер аргументов командной строки клиента API

```
getEntriesCommandParser :: ParserInfo CliCommand
getEntriesCommandParser =
  info
    (pure GetEntries <**> helper)
    ( fullDesc
        <> progDesc "Retrieves and lists all todo list entries"
    )

toggleEntryCommandParser :: ParserInfo CliCommand
toggleEntryCommandParser =
  info
    ((ToggleEntry <$> idParser) <**> helper)
    ( fullDesc
        <> progDesc "(Un)checks the entry with the given id"
    )
  where
    idParser =
      option auto $
        long "id"
          <> metavar "INT"
          <> help "Row id to toggle"
```

Определяет ParserInfo для команды list

Определяет ParserInfo для команды toggle

Определяет ParserInfo для одного Id без значения по умолчанию

Поскольку команда `add` требует в качестве данных значения типа `TodoEntry`, мы должны реализовать `Parser TodoEntry`, как показано в листинге 17.16. Важно отметить, что почти все определения вращаются вокруг определения `TodoEntry`. Поскольку типы проверяются статически, даже через границу между сервером и клиентом, мы можем быть уверены, что клиент всегда будет совместим с сервером, при условии что они собраны в одной и той же версии проекта. Конечно, предполагается, что логика клиента и сервера реализована правильно.

Листинг 17.16 Парсер аргументов командной строки клиента API

```
addEntryCommandParser :: ParserInfo CliCommand
addEntryCommandParser =
  info
    ((AddEntry <$> todoEntryParser) <**> helper)
    ( fullDesc
        <> progDesc "Adds a new entry to the todo list"
    )

todoEntryParser :: Parser TodoEntry
todoEntryParser =
  TodoEntry
    <$> teEntryId
    <*> (T.pack <$> teText)
    <*> teChecked
    <*> teRank
  where
    teEntryId =
      optional $
        option auto $
          long "id"
            <> metavar "INT"
            <> help "Row id to use (optional)"
    teText =
      strOption $
        long "text"
          <> metavar "TEXT"
          <> help "What is there to do?"
    teChecked =
      option auto $
        long "checked"
          <> metavar "BOOL"
          <> help "Is it done?"
    teRank =
      optional $
        option auto $
          long "rank"
            <> metavar "INT"
            <> help "Sorting rank (optional)"
```

Определяет ParserInfo для команды add

Определяет парсер факультативного id

Определяет парсер текста записи

Определяет парсер свойства checked записи

Определяет парсер факультативного ранга

17.3.3 Реализация клиента

Покончив с разбором аргументов, мы можем завершить определение клиента написанием действия `main`. Мы имеем действия из типа `ClientM` и должны научиться выполнять их с помощью `runClientM`. Для этого нам нужно значение типа `ClientEnv`, которое мы можем построить функцией `mkClientEnv` типа `Manager -> BaseUrl -> ClientEnv`. Тип `Manager` нужен для управления заголовками keep-alive,

которые нам не важны. Для создания `Manager` с разумными значениями по умолчанию мы можем использовать `newManager defaultManagerSettings`. Но `BaseUrl` нам важен, потому что он определяет базовый URL-адрес, к которому обращается наш клиент:

```
ghci> import Servant.Client
ghci> :i BaseUrl
type BaseUrl :: *
data BaseUrl
  = BaseUrl {baseUrlScheme :: Scheme,
             baseUrlHost :: String,
             baseUrlPort :: Int,
             baseUrlPath :: String}
```

Он определяет схему, хост, порт и путь по умолчанию для всех запросов. Для нашего клиента схема `Scheme` равна `Http`, путь пустой, а информация о хосте и порте берется из разобранных аргументов.

Листинг 17.17 Действие main для клиента списка задач

```
main :: IO ()
main = do
  args <- execParser argumentParser
  clientManager <- newManager defaultManagerSettings
  let url = BaseUrl Http (cliArgsHost args)
➥ (cliArgsPort args) ""
      clientEnv = mkClientEnv clientManager url
      runClientM' :: ClientM a -> (a -> IO ()) -> IO ()
      runClientM' clientAct resAct = do
        res <- runClientM clientAct clientEnv
        case res of
          Left err -> putStrLn $ "Error: " ++ show err
          Right x -> resAct x
  case cliArgsCommand args of
    GetEntries ->
      runClientM' getEntries $ mapM_ print .
➥ sortTodoEntries
    AddEntry entry ->
      runClientM' (addEntry entry) print
    ToggleEntry rowId ->
     runClientM' (toggleEntry rowId) (const $ pure ())
```

Разбирает аргументы и команду

Определяет Manager по умолчанию для клиента

Определяет BaseUrl сервера API

Создает ClientEnv для клиента

Выбирает все записи с помощью getEntries, затем сортирует и печатает их

Добавляет новую запись с помощью addEntry и печатает результирующий id

Переключает состояние записи и игнорирует результат

Определяет действие, которое выполняет runClientM над ClientM, а затем второе действие над результатом или печатает сообщение об ошибке, если клиент завершился неудачно

Собрав все вместе, получаем определение, показанное в листинге 17.17. Мы определяем функцию `runClientM'`, для того чтобы `run-`

`ClientM` всегда использовала наше `clientEnv` и обрабатывала ошибки единообразно. Итак, исполняемый файл готов, и мы можем его протестировать!

Предположим, что наш сервер прослушивает порт `9001`. Тогда клиент запускается следующим образом:

```
shell $ stack run todolist-client -- list
TodoEntry {entryId = Just 6, text = "Implement client application",
➥ checked = False, rank = Just 1}
TodoEntry {entryId = Just 5, text = "CLI arguments", checked = False,
➥ rank = Just 2}
TodoEntry {entryId = Just 1, text = "Specify JSON and Database
➥ serialization", checked = True, rank = Nothing}
TodoEntry {entryId = Just 2, text = "Specify API type", checked = True,
➥ rank = Nothing}
TodoEntry {entryId = Just 3, text = "Implement API", checked = True,
➥ rank = Nothing}
TodoEntry {entryId = Just 4, text = "HTML rendering for list",
➥ checked = True, rank = Nothing}
```

Стоит проверить записи с идентификаторами 5 и 6:

```
shell $ stack run todolist-client -- toggle --id 5
shell $ stack run todolist-client -- toggle --id 6
shell $ stack run todolist-client -- list
TodoEntry {entryId = Just 6, text = "Implement client application",
➥ checked = True, rank = Just 1}
TodoEntry {entryId = Just 5, text = "CLI arguments", checked = True,
➥ rank = Just 2}
TodoEntry {entryId = Just 1, text = "Specify JSON and Database
➥ serialization", checked = True, rank = Nothing}
TodoEntry {entryId = Just 2, text = "Specify API type", checked = True,
➥ rank = Nothing}
TodoEntry {entryId = Just 3, text = "Implement API", checked = True,
➥ rank = Nothing}
TodoEntry {entryId = Just 4, text = "HTML rendering for list",
➥ checked = True, rank = Nothing}
Все отлично! Напоследок добавим еще одну задачу:
shell $ stack run todolist-client -- add --text "Celebrate learning
➥ Haskell!" --checked False --rank 0
7
```

Ну а выполнение этой задачи оставим читателю в качестве упражнения!

Резюме

- Servant позволяет определять типобезопасные API, используя операторы типа `(:>)` и `(:<|>)`, чтобы присоединить информацию о пути к конечной точке и объединить несколько конечных точек в единый API соответственно.
- Фантомным называется тип, который содержит по меньшей мере одну переменную типа, которая не встречается в правой части определения и используется для передачи дополнительной информации о типе функциям без необходимости конструировать явные значения типа.
- Тип `Proxy` из модуля `Data.Proxy` часто используется в качестве фантомного.
- Внутри обработчика `Handler` мы можем возбудить исключение типа `ServerError`, чтобы вернуть клиенту ответ с описанием ошибки.
- WAI-приложение `Application` можно модифицировать с помощью функции `Middleware`, которая принимает `Application` и возвращает модифицированное `Application`.
- С помощью функции `client` из модуля `Servant.Client` мы можем автоматически вывести функции, взаимодействующие с API `servant`.
- Для определения команд в интерфейсе командной строки используется функция `subparser`.

Приложение А

Комплект инструментов Haskell

Как любой современный язык, Haskell поставляется в комплекте с разнообразными инструментами. Ниже приведен краткий перечень основных.

- *GHC – Glasgow Haskell Compiler*, стандартный компилятор языка Haskell.
- *GHCi* – интерактивная среда GHC. Ее можно использовать для возни с функциями, тестирования фрагментов кода и получения информации о типах и определениях.
- *Cabal* – инструмент для сборки Haskell-приложений. Применяется для скачивания зависимостей, компиляции программ и даже сборки пакетов кода. Он использует Hackage – репозиторий пакетов Haskell – в качестве источника зависимостей и GHC для компиляции.
- *Stack* – средство организации проектов и разрешения зависимостей с гарантией успешной сборки. При нормальном применении заменяет взаимодействие с cabal. Использует Hackage, Cabal, GHC и набор пакетов Stackage, который состоит из тщательно отобранных из Hackage пакетов, протестированных на предмет совместимости.

- *HLS* – языковой сервер Haskell. Применяется в сочетании с редакторами и IDE для предоставления информации о типах, ошибках и предупреждениях компилятора в реальном времени.

Полезны также другие инструменты:

- *GHCup* – средство установки основного инструментария. Умеет устанавливать различные версии GHC, Cabal, Stack и HLS.
- *HLint* – рекомендует, как улучшить код на Haskell. Может подсказать способы упрощения кода (например, посоветовать эта-редукции).
- *Ormolu* – форматер кода на Haskell. Отформатированный по умолчанию код очень удобен для восприятия, именно такой код приведен в этой книге.
- *Hasktags* – генерирует файлы тегов для кода на Haskell. В этих файлах хранятся указатели на местоположение определений, чтобы можно было быстро находить их в редакторе или IDE.
- *Shake* – библиотека для конфигурирования систем сборки проектов на Haskell.

A.1 Выбор инструментов и редактора

В этой книге мы используем GHCi и Stack, так что эти инструменты необходимы. GHCup очень удобен для установки других инструментов, поэтому мы его настоятельно рекомендуем. Если вы работаете с Visual Studio Code, то HLS может быть неплохим дополнением к технологическому процессу, однако у HLS возникают трудности, когда кодовая база велика. Для таких проектов hasktags является более быстрой и, пожалуй, более разумной альтернативой.

Конечно, выбор инструментария – дело ваше. Но я рекомендую установить HLint, особенно начинающим, поскольку он дает хорошие советы по улучшению кода. Ormolu (или любой другой форматер) также может сильно помочь, потому что вам не придется думать о количестве пробелов и отступах.

A.2 Установка инструментов Haskell

Ссылки на конкретные инструменты приведены в файле README в репозитории кода! Обязательные инструменты можно установить с помощью GHCup в различные операционные системы. Из командной строки можно вызвать интерактивное меню:

```
ghcup tui
```

Оно позволит установить GHC, Stack, Cabal и HLS. Отметим, что вместе с GHC устанавливается и GHCi, так что вы можете вызвать его из оболочки:

```
shell $ ghci
```

A.2.1 Запуск Stack и GHCi

При первом выполнении Stack необходимо обновить индекс пакетов:

```
stack update
```

Если вы хотите убедиться, что установлена последняя версия Stack, можете сделать это с помощью GHCup. Запустить `ghci` можно двумя способами: непосредственно из Stack, выполнив одну из двух команд:

```
stack repl
stack ghci # Просто псевдоним stack repl
```

Но в некоторых системах двоичные файлы могут быть установлены в каталог, не прописанный в переменной окружения PATH. Чтобы узнать, в какой каталог Stack устанавливает программы, можете выполнить команду `stack path` и поискать в выданной ей распечатке путь *programs*. Именно туда Stack копирует установленные двоичные файлы. Добавив `ghc` и `ghci` в PATH, вы сможете вызывать эти программы без указания полного пути!

A.2.2 Установка других инструментов

Многие инструменты Stack позволяет установить глобально. При таком способе они устанавливаются в каталоги, прописанные в PATH:

```
stack install <tool name>
```

A.3 Все в одном Docker-файле

Если вы не хотите устанавливать никакие инструменты Haskell на свой компьютер локально, тоже нет причин для паники. В репозитории кода для этой книги имеется Docker-файл, с помощью которого можно создать образ, который будет содержать весь комплект инструментов!

Находясь в репозитории, выполните следующую команду, которая построит образ:

```
shell $ docker build -t haskell-bookcamp .
```

После построения образа можете выполнить контейнер, который смонтирует репозиторий на каталог /root/repo, откуда можно будет запускать код. Можете также поместить в контейнер свои собственные проекты на Haskell, чтобы было проще работать.

```
shell $ docker run \
  -ti \
  --mount type=bind,source="$(pwd)",target=/root/repo \
  haskell-bookcamp \
  bash
```

Приложение B
Ленивое вычисление

Haskell вычисляет значения не строго, а *лениво*, т. е. именно в тот момент, когда в них возникает необходимость; это называется *вызовом по необходимости*. Однако на этом история не заканчивается.

Прежде всего *почему ленивое вычисление полезно*? Взгляните на функцию `const`:

```
const :: a -> b -> a
const x _ = x
```

Как видим, второй аргумент всегда отбрасывается. А что, если вычислять значение второго аргумента трудно и долго? Ответ – *наплевать*, потому что оно и не нужно! Это относится также к различению ветвей (например, в конструкции if-then-else или при сопоставлении с образцом). Рассмотрим такой код:

```
expensive1 :: Int
expensive1 = ... -- something terribly expensive

expensive2 :: Int
expensive2 = ... -- something terribly expensive

f :: Bool -> Int
f b = if b then expensive1 else expensive2
```

При вычислении `f` нам нужно вычислить только одно из двух значений. Другое попросту игнорируется!

B.1 Ленивые структуры данных

В главе 5 ленивые вычисления нашли применение в нашем алгоритме поиска. Мы строим граф лениво, и поиск в нем тоже ленивый (поскольку в Haskell по умолчанию все вычисления ленивые). Это позволяет вычислять только ту часть графа, в которой производится поиск.

Канонический пример ленивой структуры данных – *бесконечный список*. Да-да, вы не ослышались:

```
nat :: Int
nat = [1..]
```

Так выглядит список всех натуральных чисел. Почему эта конструкция работает? Рассмотрим альтернативную реализацию:

```
natRec :: [Int]
natRec = go 1
  where
    go n = n : go (n+1)
```

Вызов `go n` производит вычисления до `n : go (n+1)`, но *только если необходимо*! Поэтому когда нам нужно вычислить первые 15 элементов списка, мы можем сделать это, не вычисляя весь список.

```
ghci> take 15 natRec
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
ghci> take 15 (drop 15 natRec)
[16,17,18,19,20,21,22,23,24,25,26,27,28,29,30]
```

B.2 Как это работает

Ленивое вычисление реализуется с помощью *переходников* (thunk). Переходник представляет *отложенное вычисление*. Мы можем в GHCi посмотреть, какие части структуры данных вычисляются:

```
ghci> x = [1..10] :: [Int]
ghci> y = map (+1) x
ghci> :print y
y = (_t1::[Int])
ghci> take 5 y
[2,3,4,5,6]
ghci> :print y
y = 2 : 3 : 4 : 5 : 6 : (_t2::[Int])
```

Команда `:print` печатает значение, *не вызывая* его принудительного вычисления. Мы видим, что сначала `y` вообще не вычисляется,

но после того как были принудительно вычислены первые пять элементов (вследствие печати их на экран), список оказался вычисленным до пятого элемента:

```
ghci> x = [1,1,1,1,1] :: [Int]
ghci> y = map (+1) x
ghci> :print y
y = (_t1::[Int])
ghci> length y
5
ghci> :print y
y = [(_t2::Int),(_t3::Int),(_t4::Int),(_t5::Int),(_t6::Int)]
```

Важно различать вычисление конструкторов и фактических значений. Поскольку функции `length` есть дело только до конструкторов списка, но не до значений, она полностью вычисляет конструкторы, но не переходники, ведущие к значениям!

B.3 Утечки памяти и времени

Ленивые вычисления порождают несколько проблем. Первая – непредсказуемая производительность, потому что мы никогда не знаем, когда именно выражение будет полностью вычислено. То есть не знаем, в какой момент работа будет выполнена.

Но куда более серьезная проблема связана с тем, что переходники – это дополнительный шаг между выражением и его значением. Рассмотрим простое применение функции `foldl`:

```
foldl (+) 0 [1,2,3]
-> foldl (+) (0 + 1) [2,3]
-> foldl (+) ((0 + 1) + 2) [3]
-> foldl (+) (((0 + 1) + 2) + 3) []
-> (((0 + 1) + 2) + 3)
-> ((1 + 2) + 3)
-> (3 + 3)
-> 6
```

Знак → представляет один шаг вычисления. `foldl` сначала строит постепенно увеличивающийся терм, состоящий из переходников, который редуцируется только после завершения рекурсии. Лучше вместо этого использовать строгую версию той же функции: `foldl'`.

Ленивое вычисление может приводить и к гораздо более зловещему явлению: *утечкам памяти*. Рассмотрим очень простую функцию, которая вычисляет среднее списка чисел:

```
avg :: (Fractional a) => [a] -> a
avg xs = sum xs / fromIntegral (length xs)
```

Выглядит просто. Функция `avg` должна потреблять память постоянного размера, поскольку мы всего лишь вычисляем сумму всех элементов списка и заодно длину списка. Однако при попытке вычислить эту функцию мы увидим, что потребление памяти во много раз выше ожидаемого. Даже для тех списков, для которых ни `sum`, ни `length` не нуждаются в большом объеме памяти, функция `avg` «пожирает» много гигабайтов! В чем же дело? В том, что `xs` разделяется между двумя вызовами функции. Под *разделением* мы понимаем тот факт, что если два выражения указывают на один и тот же переходник, то переходник вычисляется только один раз, потому что новое вычисление не изменит результата. Но в данном случае это означает, что после того как сумма вычислена, весь гигантский список будет храниться в памяти до тех пор, пока не закончится вычисление `length`. Напомним, что функции `sum` и `length` ленивые! Они создают огромную коллекцию переходников, вычисление которых начинается только при вычислении `(/)`, что приводит к гигантской утечке памяти в этой простой функции.

Чтобы избежать этой проблемы, мы должны вычислять сумму и длину одновременно. Это не очень сложно – нужно только воспользоваться функцией `foldl'`:

```
avg :: (Fractional a) => [a] -> a
avg xs =
  let (sum, count) = L.foldl' f (0,0) xs
   in sum / count
  where
   f (sum, count) x = (sum + x, count + 1)
```

Однако это не решает проблему, потому что `foldl'` будет вычислять *слабую головную нормальную форму*, т. е. термы будут вычисляться только до их первого конструктора. В нашем случае это конструктор кортежа `(,)`, а не сами значения внутри кортежей, поэтому переходники все равно накапливаются в процессе свертки. Чтобы исправить ситуацию, мы должны также сделать значения аккумулятора строгими. Это возможно с помощью расширения языка `BangPatterns` и добавления восклицательных знаков перед аргументами аккумулятора:

```
avg :: (Fractional a) => [a] -> a
avg xs =
  let (sum, count) = L.foldl' f (0,0) xs
   in sum / count
  where
   f (!sum, !count) x = (sum + x, count + 1)
```

Тогда `sum` и `count` будут вычисляться до слабой головной нормальной формы перед вызовом `f`. Для числовых типов это означает, что

они будут вычисляться полностью. Это решает проблему избыточного потребления памяти, теперь функции нужна память постоянного размера, поскольку она потребляет список как ленивый поток.

B.4 undefined и newtype

undefined – это состояние ошибки, поскольку что-то из того, что мы хотели вычислить, не определено. Однако у undefined есть тип:

```
ghci> x = undefined :: Int
ghci> :t x
x :: Int
ghci> :t undefined
undefined :: a
```

undefined имеет не один какой-то тип, а может быть *любого* типа! Следовательно, для каждого типа может существовать значение undefined! Это значение часто называют *дном*, оно обозначает «неопределенность».

Вообще говоря, столкнувшись с undefined, программа аварийно завершается, но ленивые вычисления иногда изменяют это правило:

```
ghci> case undefined of _ -> print "I did not crash!"
"I did not crash!"
```

Почему программа не «упала»? Потому что образец _ не вынуждает вычислять неопределенный переходник!

```
ghci> :print undefined
undefined = (_t1::GHC.Stack.Types.HasCallStack => a)
```

Поскольку undefined не вычислялось, причин для краха нет.

```
ghci> data X = X
ghci> case undefined of X -> print "I did not crash!"
*** Exception: Prelude.undefined
```

Однако все изменилось, стоило нам ввести образец, который вынуждает вычислить переходник до его первого конструктора. Теперь переходник undefined вычисляется, и программа «падает», но это верно не *всегда*:

```
ghci> newtype Newtype = Newtype Int
ghci> data Data = Data Int
ghci> case undefined of Newtype _ -> print "I did not crash!"
"I did not crash!"
ghci> case undefined of Data _ -> print "I did not crash!"
*** Exception: Prelude.undefined
```

Ключевое слово `newtype`, похоже, нарушает правила! Почему? В главе 5 мы узнали, что `newtype` позволяет компилятору убирать конструктор, оптимизируя программу, поскольку существует взаимно однозначное соответствие между полем `newtype` и его конструктором. Поэтому `newtype` более ленивый! Конструктор никогда не проверяется, а остается только образец _, и поведение оказывается тем же, что и раньше!

Предметный указатель

Символы

++ оператор, 181
(++) конструктор, 361
(++) функция, 361
</> оператор, 410
:<|> оператор, 468
:kind команда, 204
:type команда, 53
<$> (инфиксная версия fmap), 218
(<$) оператор, 351
==> оператор, 253
(==) функция, 126
:> оператор, 468
(.|.) :: a -> a -> a оператор, 356
|| оператор, 55
(.|) оператор, 312
$ оператор, 180
(!) оператор, 388
(*) оператор, 49
&& оператор, 55
-main-is параметр, 477

A

AbsoluteSeek режим, 405
add команда, 491
ADSR-огибающая
  параметры, 281
  подъем, спад, поддержание, затухание, 273
aeson библиотека, 454
  разбор JSON, 448
all функция, 97, 233
Alphabet тип, 54
alphabetRot функция, 237, 242
alter функция, 131, 136
and функция, 306
App действие, 423
AppendMode, 403
Applicative экземпляр, 351
applyAtomicGen функция, 227
Arbitrary класс типов, 220, 243, 247
arbitrary функция, 243, 247
ask функция, 424
async пакет, 394
AtomicGenM тип, 227
AtomicGenM StdGen тип, 227
Attoparsec библиотека, 348
Audio значение, 270
avg функция, 504

B

BangPatterns расширение языка, 505
BaseUrl, 495
Bits класс типов, 356, 358
BlockBuffering режим, 404
Bool тип, 451
bracket функция, 408
BufferMode тип, 404
bytestring пакет, 166, 348, 406

ByteString тип, 348

C

c2w функция, 348
caesar функция, 64, 67
case ключевое слово, 103
catchJust функция, 417
catMaybes функция, 148
ceiling функция, 266
changes функция, 457
checkCoverage функция, 254
choice функция, 347
choose функция, 243
chooseAny функция, 243
client функция, 490
ClientM тип, 490, 494
Codec.Picture модуль, 367, 387
collect функция, 252
command функция, 493
concat функция, 149, 272
concatMap функция, 149, 272
Connection тип, 472, 476
const функция, 99, 167, 502
Content-Type заголовок, 480
Control.Concurrent модуль, 394
Control.Exception модуль, 408
Control.Monad модуль, 410
Control.Monad.Trans.Reader модуль, 421
Control.Parallel.Strategies модуль, 394, 396, 398
Conversion модуль, 192
copyFile действие, 415
copyFileWithMetadata действие, 415
cover функция, 253
CRUD (создание, чтение, обновление, удаление), 448
Csv модуль, 195
Csv тип, 175, 191
CSV (формат значений через запятую), 169
  классы типов, 181
  моделирование CSV-данных, 170
  парсеры командной строки, 212
  печать, 207
  разбор данных, 195
  свертка структур данных, 198
  умные конструкторы, 176
Csv.Operations модуль, 211
CsvParseOptions тип, 196, 202
Csv.Parsing модуль, 195
Csv.Type модуль, 173
curry функция, 216

D

data ключевое слово, 82, 102
Data.AssocMap модуль, 165
Data.Attoparsec.ByteString модуль, 348
Data.Attoparsec.ByteString.Char8 модуль, 350
Data.Attoparsec.Text, 348
Database.SQLite.Simple, 464
Data.Bits модуль, 356
Data.ByteString модуль, 406
Data.ByteString.Internal модуль, 348
Data.Char модуль, 78, 96
DataField тип, 175
Data.Hashable модуль, 165
Data.HashMap.Lazy модуль, 165
DataKinds расширение языка, 467
Data.List модуль, 60, 84, 129, 140, 142, 145, 172, 177, 201, 205
Data.Maybe модуль, 83, 112, 142, 178
Data.Sequence модуль, 428
Data.Sliceable модуль, 188
Data.String модуль, 91, 187
Data.Text модуль, 172, 201
Data.Text.IO модуль, 207, 406
Data.Vector.Strategies модуль, 398
Data.Word модуль, 350
decimal парсер, 351, 355
decode функция, 449, 452
delete функция, 147
deleteNodes функция, 155
denominator функция, 293
DeriveAnyClass расширение языка, 413, 428
DeriveGeneric расширение языка, 454
deriving ключевое слово, 112, 137
DerivingVia расширение языка, 428
determineFiletype функция, 408
DiGraph функция, 124, 154
div функция, 105
do блок, 73
do-нотация, 72, 94, 244
dropExtension функция, 410
DRY (Не повторяйся), 30
DynamicImage тип, 367, 387
DynamicPnmImage тип, 375

E

Either тип, 170, 175, 368, 376, 414
elem функция, 54, 200

elemIndex функция, 60, 84
encode функция, 449, 452
Enum класс, 265
EOF (конец файла), 407
Eq классов типов, 127, 137
error функция, 178
Eval монада, 394
evalList функция, 396
evalTraversable функция, 397
Exception класс типов, 413
execute функция, 457
executeNamed функция, 464

F

Fail конструктор, 349
FilePath тип, 54, 81, 90
filter функция, 145, 210
filterM функция, 410
flip функция, 129
Float тип, 127
floor функция, 266
fmap функция, 203
Foldable класс типов, 200
forall ключевое слово, 298
FormatError исключение, 464
Fractional класс типов, 265
FromField класс типов, 460
fromIntegral функция, 356
FromJSON ограничение, 449
fromList функция, 372
fromMaybe функция, 84
FromRow класс типов, 457
fromStrict функция, 475
fromString функция, 476
Functor, 212
  класс типов, 203, 233
  экземпляр, 247, 351

G

GeneralizedNewtypeDeriving расширение языка, 428
generateImage функция, 387
Generic класс типов, 454
GenericRGB тип, 382
GET-запросы, 468
getLine функция, 73
getProgName функция, 80
GHC (Glasgow Haskell Compiler), 40
GHCi, 500
globalStdGen тип, 228
Graph модуль, 145
Graphics.PNM модуль, 376
Graphics.PNM.Conversion модуль, 382
Graphics.PNM.Types модуль, 372
Graphics.PNM.Validation модуль, 366, 376

H

Handler монада, 473
hClose функция, 404, 407
HCodecs библиотека, 270
HeapOverflow конструктор, 416
Hedgehog каркас, 258
hFlush функция, 404
hPutStr функция, 405
hPutStrLn функция, 404
hSeek функция, 405
hSetBuffering функция, 404
hTell функция, 405
HTML-код, порождение по данным, 484
HTTP (протокол передачи гипертекста), 448, 495
HUnit каркас, 258

I

id функция, 178
if-then-else предложение, 62
Image тип, 368
ImageRGB8 конструктор, 387
ImageRGB16 конструктор, 387
ImageY8 конструктор, 387
ImageY16 конструктор, 387
imap функция, 385
import предложение, 42
in ключевое слово, 94
Int тип, 127, 451
Integral класс типов, 265, 377
IO монада, 412, 490
IOException исключение, 415, 427
IOMode тип, 403
isAscii функция, 233
isControl функция, 233
isDigit предикат, 238
IsList класс типов, 187
isLower предикат, 238
isNotEmpty функция, 101
isPrefixOf функция, 407
isPrint функция, 96
isSeparator функция, 96

IsString класс типов, 186
isUpper предикат, 238

J

JSON (JavaScript Object Notation)
  вывод классов типов с помощью Generic, 454
  кодирование значений, 448
Just конструктор, 82, 108, 131

K

KISS (Не усложняй, глупышка), 30

L

label функция, 252
Ladder модуль, 145, 162
ladderSolve функция, 162
Left конструктор, 379, 412, 414
length функция, 200, 504
let-привязки, 86, 94, 98, 103, 388
liftIO функция, 423
LineBuffering режим, 404
lines функция, 90, 96
Lisp язык программирования, 47
list команда, 491
listArray функция, 271
listDirectory функция, 410
log функция, 266
lookup функция, 129, 138, 164

M

main действие, 42, 72, 113, 116, 215, 494
Main модуль, 42, 114, 162, 478
map функция, 37, 67, 147, 149, 181, 233, 271, 410
mapM функция, 109, 379, 410
mapMaybe функция, 112
max функция, 271
maximum функция, 107, 200
Maybe монада, 379
Maybe тип, 82, 93, 95, 102, 130, 178, 190
mconcat функция, 186
Megaparsec библиотека, 348
member функция, 128, 133
mempty, 185
min функция, 271
MINIMAL прагма, 127
minimum функция, 200
mix функция, 305
module ключевое слово, 193
Monad ограничение, 380
MonadFail класс, 412
MonadIO действие, 473
MonadTrans класс типов, 423

N

NamedFieldPuns расширение языка, 182
Natural тип, 286
newtype определение, 134
NoBuffering режим, 404
NoContent тип, 475
NoFieldSelectors расширение языка, 280
not оператор, 55
Notelength, 310
NoteStructure, 300, 308
Nothing функция, 131
nub функция, 140, 145
nubBy функция, 177
null функция, 75, 200
Num класс типов, 288
numerator функция, 293

O

of ключевое слово, 103
oneof функция, 243
Only значение, 459
openFile функция, 403
optparse-applicative пакет, 419
or функция, 306
Oscillator, 281, 303, 307
overlaps функция, 305
OverloadedLists расширения языка, 187
OverloadedStrings расширения языка, 449

P

pack функция, 172
pad функция, 104
PadCenter режим, 104
PadLeft режим, 111
Pair тип, 453
Pandoc, 32
parallel пакет, 394
parListChunk функция, 398
parse функция, 349
Parsec библиотека, 347

parseJSON функция, 450
parseOnly функция, 349
Parser тип, 348, 351, 450
parVectorChunk функция, 399
PATCH-запросы, 468
PATH переменная окружения, 218, 500
Performance тип, 300, 306
Pitch тип, 299
Pixel8 тип, 367, 376
Pixel16 тип, 367, 376
PixelMapping тип, 390, 393, 397
PixelRGB8 тип, 367
PixelRGB16 тип, 367
playEvents функция, 308
PNG (Portable Network Graphics), 387
PnmImagex тип, 374, 380, 381
POST-запросы, 468
pretty префикс функций, 109
print действие, 163
Product тип, 186
Property тип, 240
propertyTest функция, 234
proto-lens пакет, 456
putStrLn функция, 73, 163, 403

Q

qualified ключевое слово, 139
QuickCheck, генерирование случайных значений, 242
  генератор Gen, 243
  сокращение тестов, 248
  AssocMap, 246
QuickCheck каркас тестирования , 239
quickCheck функция, 240, 251
quickCheckAll функция, 255

R

Random класс типов, 230
random функция, 224
RandomIO тип, 231
randomListIO функция, 230
Ratio тип, 288
RawPixel тип, 377
RawPnm тип, 365
Read класс типов, 137, 197
readDictionary действие, 162
readFile функция, 90, 207
ReadMode, 403
readProxy функция, 471
ReadWriteMode, 403
Real класс, 265
RealFloat класс типов, 267
RealFrac класс типов, 266
RecordWildCards расширение языка, 182, 208, 421
RelativeSeek режим, 405
renderHTML функция, 486
REPL (цикл чтения–выполнения–печати), 40
replicate функция, 102
replicateIO функция, 231
resize функция, 245
Restman, 479
Result тип, 349
return функция, 76
reverse функция, 112
Right конструктор, 414
Right случай, 178
ROT13, 39
rot13 функция, 241
rot135 функция, 67
rotChar функция, 62
round функция, 264, 266
RowParser тип, 458
runClientM функция, 490, 494
runEval функция, 396
RWST, 429

S

safecopy пакет, 456
sample функция, 245
Sampler тип, 314
samplesPerPeriod функция, 267
samplesPerSecond функция, 267
ScopedTypeVariables расширение языка, 417
SeekFromEnd режим, 405
SeekMode параметр, 405
Semigroup класс типов, 184, 217
Sequence конструктор, 311
Servant, 466
  вывод клиента, 490
  запуск приложения, 477
  типобезопасный API, 466
servant-client пакет, 490
serve функция, 478
Server модуль, 478, 486
ServerError тип, 473
shift метод, 356

show функция, 75, 137, 163
ShowData тип, 298
shrink функция, 243, 248
shuffle функция, 244
sin функция, 266
skipMany функция, 347
slice метод, 188
slice функция, 216
sliceDelete функция, 192
sliceMap функция, 192
slicePartition метод, 188
SmallCheck библиотека, 258
SomeException тип, 413
sort функция, 142, 222
split функция, 196
SQL, 447
  база данных SQLite, 456
  вывод классов типов с помощью Generic, 454
sqlErrorDetails поле, 474
sqlite-simple пакет, 456
sqrt функция, 266
stack
  запуск программ, 87
  команда new, 254, 406
  команда repl, 91
  команда test, 255
StackOverflow конструктор, 416
StandaloneDeriving расширение языка, 299
StateT, 423
Stats тип, 424
stderr описатель, 403
stdin описатель, 73, 403
stdout описатель, 403, 405
stm пакет, 394
stock стратегия, 428
Strategy тип, 394
StrictData расширение языка, 168
String тип, 51, 90, 166, 171, 370
string функция, 348
subparser, 493
suchThat функция, 243
Sum тип, 186
sum функция, 200
System.Directory модуль, 409, 415
System.Directory.Safe модуль, 415
System.Environment модуль, 80
System.FilePath модуль, 409
System.FileType модуль, 406, 418
System.IO модуль, 91, 403, 406, 409
System.Random модуль, 224, 230
System.Random.Stateful модуль, 224

T

take функция, 189
takeDirectory функция, 416
takeExtension функция, 410
takeWhile функция, 347, 352
Tasty библиотека, 258
tell функция, 426
TemplateHaskell расширение языка, 256
TempoInfo тип, 292
test_ префикс, 236
Testable класс типов, 240
testBit функция, 358
Test.SimpleCheck модуль, 235
text пакет, 166
Text тип, 166, 172, 202, 207, 451
Text.Read модуль, 197
ThreadKilled конструктор, 416
throw функция, 412
throwError действие, 473
throwIO функция, 412
TodoApi тип, 476
TodoEntry тип, 451, 459, 472
todoEntryParser функция, 452
todolist-server исполняемый файл, 477
Todos.Api модуль, 467
Todos.Database модуль, 461
Todos.Handlers модуль, 472, 484
Todos.Types модуль, 449
toField функция, 458
toFrequency функция, 283, 299
toggle команда, 491
toggleEntryCommandParser, 493
toJSON функция, 454
toPerformance функция, 303
toRational функция, 289
toRow функция, 458
toUpper функция, 78
transform функция, 67
transformers пакет, 422, 429
transpose функция, 205
Traversable тип, 379
traverse функция, 380
try функция, 414
tryJust функция, 417
type ключевое слово, 51, 102
TypeOperators расширение языка, 467

TypeSynonymInstances расширение языка, 383

U

uncurry функция, 216
undefined, 506
unliftio пакет, 416
unlines функция, 90, 96
unordered-containers, 165
unpack функция, 172, 407
unwords функция, 90, 96
unzip функция, 106
UserInterrupt исключение, 417
UserInterrupt конструктор, 416
using функция, 396

V

validatePnm функция, 368, 386
vector пакет, 371
Vector тип данных, 371, 387
vector функция, 244
vectorOf функция, 244
verboseCheck функция, 254

W

w2c функция, 348
WAI (интерфес веб-приложений), 478
where привязка, 388
withStrategy функция, 396
withText функция, 450
withTransform функция, 381, 386
Word8 тип, 348
words функция, 90, 96
writeDynamicPng, 388
writeFile функция, 91, 207
WriteMode, 403
WriterT, 423
WSL (Windows Subsystem for Linux), 34

X

x:xs образец, 112

Z

zip функция, 105
zipWith функция, 107

А

Автономное тестирование, 220
Алгебраические структуры данных как кодирование возможностей, 101
  не повторяйся, 104
  отсутствующие значения, 108
  печать списка значений с помощью mapM, 109
  помеченное объединение, 102
  функция zip, 105
Алгебраические типы данных (ADT), 82
Алфавиты, 51
Анонимные функции, 48
Аргументы командной строки
  парсеры, 212
  разбор и печать, 84
  чтение из окружения, 80
Асинхронные исключения, 416
Ассоциативность, 184
Ассоциативные списки, 92
Атомарные типы, 45

Б

Байты, чтение из файла, 406
Бесконечные списки, 273, 503
Бинарная функция, 48, 54
Буферизация, 404

В

Виртуальные потоки, 393
Внешнее соединение, 184
Внутреннее соединение, 184
Возведение в степень, 291
Возвращаемого значения тип, 49
Вывод типов, 44
Вызов по необходимости, 502

Г

Граф предшествования, 153
Графы, построение, 121

Д

Действия ввода-вывода, 72
  выход из рекурсивного действия, 75
  моделирование цикла, 74
  простые, 72

Декларативное программирование, 28

Е

Единичный тип, 72

З

Задача поиска, 121
Запоминание маршрута, 468
Захват работы, 395
Захват ресурсов и функция bracket, 408
Зеленые потоки, 393
Значения, кодирование в формате JSON, 448

И

Игра в цепочки слов, 151
Идентификатор, 44
Изображений разбор, 350
Императивное программирование, 74
Индекс элемента, 57
Интерфейс
  веб-приложений, 478
  командной строки, определение команд, 491
Инфиксная нотация, 55
Инфиксный оператор, 180, 309
Информация о типе, 365
Исключения
  возбуждение и перехват, 413
  обработка ошибок, 411
Искры, 395, 399

К

Каррирование, 216
Карты перестановок, 141
Каталоги, перечисление, 410
Квалифицированный импорт, 139
Классы
  типов, 75, 181
    вывод с помощью Generic, 454
    создание, 188
    числовых, 263
    IsString, 187
    Semigroup и Monoid, 183
  числовых типов, 263
Клиентское приложение, 448
Комбинаторы парсеров, 347
Комплексное тестирование, 220
Комплекты тестов, 254
Композиция функций, 208
Конструкторы, 58, 82
  данных, 173
  использование списков экспорта для сокрытия, 133
Кортежи, 47, 92

Л

Левая свертка, 199
Лексическая область видимости, 159
Ленивое вычисление, 167
  как работает, 503
  ленивые структуры данных, 503
  утечки памяти, 504
Линейный конгруэнтный генератор, 223
Логические комбинации, 55
Лямбда-абстракция, 48

М

Магические числа (сигнатуры), 406
Модули, 42
Монадическое программирование, 29
Монадные трансформеры, 419
  ReaderT, 420
  RWST, 429
  StateT, 423
  WriterT, 425
Монады, 73
Мономорфный тип, 204
Музыка, интерпретация структур, 300
  микширование сигналов, 303
  полифонические группы, 306
Музыкальные композиции
  полиморфные структуры данных с несколькими типами, 296
  реализация предметно-ориентированного языка, 309

Н

Навешивание квантора
  всеобщности, 157
  существования, 297, 374
Неизменяемые данные, 57
Неоднородные структуры данных, 297
Непрерывная доставка, 259
Непрерывная интеграция, 259

Нечистый код, 119
Низкоуровневые операции, 356
Нормальная форма, 394
Ноты с точкой, 290

О

Область видимости переменной типа, 157
Обобщенные алгебраические типы данных (GADT), 369
Обработка ошибок, 415
Объявление приоритета операторов, 312
Ограничения типа, 127
Однородные структуры данных, 297
Операторы типа, 467
Описатели, допускающие поиск, 405
Описатель файла, 403
Отношение сигнал–шум, 263
Отсутствующие значения, 108

П

Параллелизм, 393
Параллельная обработка изображений, 364
  обобщенный алгоритм преобразования изображения, 382
  предоставление информации о типе вызывающей стороне, 365
  преобразование данных, 389
  проверка разобранных данных, 375
Параметризация типов, 65
Параметрический полиморфизм, 66
Парсер командной строки, 212
Переходники, 394, 503
Периодические функции, 267
Побочные эффекты, 28, 412
Поднятие функций, 423
Подстановочные шифры, 50
Поиск
  в глубину, 152, 161
  в файле, 404
  в ширину, 152, 161
Полиморфизм по типу возвращаемого значения, 230, 365
Полиморфные типы, 122
Полифонические группы, 306
Пользовательские операторы, 310
Помеченное объединение, 102
Постусловия, определение для случайных значений, 230
Поток управления, 62, 113
Правая свертка, 200
Правоассоциативность, 180
Предметно-ориентированные языки (DSL), реализация, 309
Предусловие, 253
Преобразование изображения, обобщенный алгоритм, 382
Привязка образца, 211
Профилирование, 163
Пул искр, 395

Р

Рандомизированное тестирование, 235
Расширения языка, 159
Рекурсивная функция, 59
Рекурсия, 57

С

Селекторы
  полей, 173, 279
  частичных полей, 278
Семантические выражения, 44
Синтаксис записей, 171, 173
Синхронные исключения, 416
Система времени выполнения, 393
Системное тестирование, 220
Слабая головная нормальная форма, 394
Случайные значения
  генерирование, 222
  использование глобального генератора, 226
Сокращение тестов, 248
Сопоставление с образцом, 57, 83
Списки, 46
  экспорта, 42, 43
    использование для сокрытия конструкторов, 133
Списковые включения, 147
Список
  импорта, 142
  смежности, 124
Средний уровень приложения, 482
Ссылочная прозрачность, 238
Статически типизированный язык, 44
Стражи, 306
Стратегия вывода, 427
Строгое вычисление, 167
Строки, преобразование, 64

Т

Тестирование свойств, 221
  модификация параметров теста, 254
  построение комплектов тестов, 254
  простой тест свойств, 228
  уровень детализации и отчеты о покрытии, 251
  эффективность тестирования, 258
Тест спецификации, 236
Тип, выбор по умолчанию, 251
Трансформеры
  монадные, 420
  стек, 429

У

Умные конструкторы, 176
  оператор $, 180
  предоставление небезопасной альтернативы, 178
  проверка свойства на этапе конструирования, 176
Унарная функция, 48
Уровень
  детализации, 251
  термов, 476
Утечки памяти, 504

Ф

Файловая система, 409
  перечисление файлов и каталогов, 410
  System.Directory модуль, 410
  System.FilePath модуль, 410
Файлы, 401
  исключения, 411
  открытие и чтение, 402
  перечисление, 410
  чтение байтов, 406
Фантомные типы, 470
Формальная верификация, 259
Функции, 41, 44
  высшего порядка, 65
Функциональное программирование, 208

Х

Хеш-таблицы, 162

Ц

Цезаря шифр, 38
Центры затрат, 164
Цепочка слов, игра, 120

Ч

Частичное применение функции, 98
Частота дискретизации, 262
Числовые значения, разбор, 197
Чистое функциональное программирование, 24, 27
Чистый код, 119

Э

Эта-редукция, 99
Эффект размытия, 391

Я

Ядра, 384

Книги издательства «ДМК Пресс»
можно купить оптом и в розницу на складе издательства по адресу:
Москва, ул. Электродная, д. 2, стр. 12, офис 7, тел. **+7 (499) 322-19-38**,
а также заказать на сайте **www.dmkpress.com**
с доставкой в любой регион РФ

Филипп Хагенлохер

Изучаем Haskell на примерах

Главный редактор *Мовчан Д. А.*
Зам. главного редактора *Яценков В. С.*
editor@dmkpress.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 42,09. Тираж 100 экз.

Веб-сайт издательства: **www.dmkpress.com**

Haskell – это мощный и красивый, но непростой функциональный язык программирования. Он запрещает опасные действия, которые обычно приводят к дефектам и краху программ и побуждает программистов сосредоточиться на простых функциях, четко определенных поведениях и правильных высокоуровневых абстракциях, избавляя их от утомительной отладки и рефакторинга кода.

Эта книга научит вас искусству создания Haskell-приложений на примерах разработки полезных и увлекательных проектов. Вас ждут язык для сочинения музыки, библиотека обработки изображений и многое другое! Особое внимание уделено таким вопросам, как проектирование функций, композиция и неизменяемость данных. Познакомившись с Haskell поближе, вы поймете, почему этот язык приобрел преданных поклонников.

«Замечательное введение и ясный подход к решению задач на Haskell».

Тэд Мейер, LI-COR Environmental

«Haskell для тех, кто любопытен и обладает практическим складом ума».

Марк Мойу, NVIDIA

«Современные и всеобъемлющие знания, которых достаточно для создания поисковых систем мирового класса».

Магнус Тэрнинг, CarbonCloud AB

«Отличная работа об использовании этого суперского языка в небольших проектах. Горячо рекомендую!»

Виниций Венц, JPMorgan Chase & Co.

Среди рассматриваемых тем:

- применение Haskell для решения повседневных задач программирования;
- чистое функциональное программирование;
- как не попасть в ловушки, расставленные Haskell.

Издание адресовано начинающим и опытным программистам, желающим освоить функциональное программирование. Ожидается, что читатель знает какой-либо объектно-ориентированный язык.

Филипп Хагенлохер занимается разработкой на Haskell и является создателем многим полюбившегося курса Haskell for Imperative Programmers на YouTube.

ISBN 978-5-93700-193-1

ДМК ПРЕСС издательство
www.дмк.рф