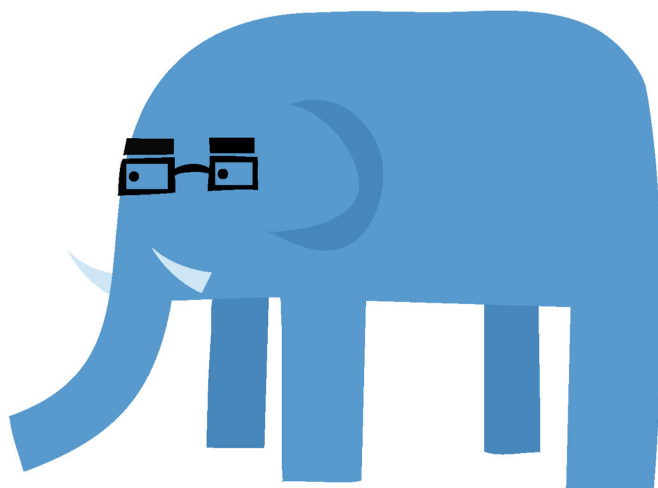
Изучай Haskell во имя добра!

Для начинающих



Миран Липовача

Learn You a Haskell for Great Good!

A Beginner's Guide

Miran Lipovača

San Francisco

Изучай Haskell во имя добра!

Миран Липовача

Москва, 2012

УДК 004.432.42Haskell
ББК 32.973.28-018.1
Л61

Л61 **Миран Липовача**

Изучай Haskell во имя добра! / Пер. с англ. Леушина Д., Синицына А., Арсанукаева Я.– М.: ДМК Пресс, 2012. – 490 с.: ил.

ISBN 978-5-94074-749-9

На взгляд автора, сущность программирования заключается в решении проблем. Программист всегда думает о проблеме и возможных решениях – либо пишет код для выражения этих решений.

Язык Haskell имеет множество впечатляющих возможностей, но главное его свойство в том, что меняется не только способ написания кода, но и сам способ размышления о проблемах и возможных решениях. Этим Haskell действительно отличается от большинства языков программирования. С его помощью мир можно представить и описать нестандартным образом. И поскольку Haskell предлагает совершенно новые способы размышления о проблемах, изучение этого языка может изменить и стиль программирования на всех прочих.

Ещё одно необычное свойство Haskell состоит в том, что в этом языке придаётся особое значение рассуждениям о типах данных. Как следствие, вы помещаете больше внимания и меньше кода в ваши программы.

Вне зависимости от того, в каком направлении вы намерены двигаться, путешествуя в мире программирования, небольшой заход в страну Haskell себя оправдает. А если вы решите там остаться, то наверняка найдёте чем заняться и чему поучиться!

Эта книга поможет многим читателям найти свой путь к Haskell.

УДК 004.432.42Haskell
ББК 32.973.28-018.1

Original English language edition published by No Starch Press, Inc. 38 Ringold Street, San Francisco, CA 94103. Copyright (c) 2011 by No Starch Press, Inc. Russian-language edition copyright (c) 2012 by DMK Press. All rights reserved.

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

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

ISBN 978-1-59327-283-8 (англ.)
ISBN 978-5-94074-749-9

© 2011 Miran Lipovaca, No Starch Press, Inc.
© Оформление, перевод на русский язык, ДМК Пресс, 2012

Оглавление

От издателя 12

Предисловие 13

Введение 16

1. На старт, внимание, марш! 21

Вызов функций 24
Функции: первые шаги 25
Списки 26
Конкатенация 29
Обращение к элементам списка 30
Списки списков 31
Сравнение списков 31
Другие операции над списками 32
Интервалы 35
Генераторы списков 37
Кортежи 41
Использование кортежей 42
Использование пар 43
В поисках прямоугольного треугольника 44

2. Типы и классы типов 46

Поверь в типы 46
Явное определение типов 47
Обычные типы в языке Haskell 48
Типовые переменные 50
Классы типов 51
Класс Eq 52
Класс Ord 53
Класс Show 54
Класс Read 54
Класс Enum 56
Класс Bounded 56
Класс Num 57

Класс Floating 58
Класс Integral 58
Несколько заключительных слов о классах типов 59

3. Синтаксис функций 60
Сопоставление с образцом 60
Сопоставление с парами 62
Сопоставление со списками и генераторы списков 63
Именованные образцы 66
Эй, стража! 67
Где же ты, where?! 69
Область видимости декларации where 71
Сопоставление с образцами в секции where 72
Функции в блоке where 72
Пусть будет let 73
Выражения let в генераторах списков 75
Выражения let в GHCi 75
Выражения для выбора из вариантов 76

4. Рекурсия 78
Привет, рекурсия! 78
Максимум удобства 79
Ещё немного рекурсивных функций 81
Функция replicate 81
Функция take 81
Функция reverse 82
Функция repeat 83
Функция zip 83
Функция elem 84
Сортируем, быстро!.. 84
Алгоритм 85
Определение 86
Думаем рекурсивно 87

5. Функции высшего порядка 89
Каррированные функции 89
Сечения 92
Печать функций 93
Немного о высоких материях 94
Реализация функции zipWith 95
Реализация функции flip 96
Инструментарий функционального программиста 98
Функция map 98
Функция filter 99
Ещё немного примеров использования map и filter 100
Функция map для функций нескольких переменных 103

Лямбда-выражения ... 104
Я вас сверну! ... 106
Левая свёртка foldl ... 107
Правая свёртка foldr ... 108
Функции foldl1 и foldr1 ... 110
Примеры свёрток ... 111
Иной взгляд на свёртки ... 112
Свёртка бесконечных списков ... 113
Сканирование ... 114
Применение функций с помощью оператора $ ... 115
Композиция функций ... 117
Композиция функций с несколькими параметрами ... 118
Бесточечная нотация ... 120

6. Модули ... 122
Импорт модулей ... 123
Решение задач средствами стандартных модулей ... 125
Подсчёт слов ... 125
Иголка в стоге сена ... 127
Салат из шифра Цезаря ... 129
О строгих левых свёртках ... 130
Поищем числа ... 132
Отображение ключей на значения ... 135
Почти хорошо: ассоциативные списки ... 135
Модуль Data.Map ... 137
Написание собственных модулей ... 142
Модуль Geometry ... 143
Иерархия модулей ... 145

7. Создание новых типов и классов типов ... 148
Введение в алгебраические типы данных ... 148
Отличная фигура за 15 минут ... 149
Верный способ улучшить фигуру ... 151
Фигуры на экспорт ... 153
Синтаксис записи с именованными полями ... 154
Параметры типа ... 157
Параметризовать ли машины? ... 160
Векторы судьбы ... 162
Производные экземпляры ... 163
Сравнение людей на равенство ... 164
Покажи мне, как читать ... 165
Порядок в суде! ... 167
Любой день недели ... 168
Синонимы типов ... 170
Улучшенная телефонная книга ... 170

Параметризация синонимов 172
Иди налево, потом направо 173
Рекурсивные структуры данных 176
Улучшение нашего списка 177
Вырастим-ка дерево 179
Классы типов, второй семестр 183
«Внутренности» класса Eq 183
Тип для представления светофора 184
Наследование классов 186
Создание экземпляров классов для параметризованных типов 187
Класс типов «да–нет» 190
Класс типов Functor 193
Экземпляр класса Functor для типа Maybe 195
Деревья тоже являются функторами 196
И тип Either является функтором 197
Сорта и немного тип-фу 198

8. Ввод-вывод 204
Разделение «чистого» и «нечистого» 204
Привет, мир! 205
Объединение действий ввода-вывода 207
Использование ключевого слова let внутри блока do 211
Обращение строк 212
Некоторые полезные функции для ввода-вывода 215
Функция putStr 215
Функция putChar 216
Функция print 217
Функция when 218
Функция sequence 218
Функция mapM 220
Функция forever 220
Функция forM 221
Обзор системы ввода-вывода 222

9. Больше ввода и вывода 223
Файлы и потоки 223
Перенаправление ввода 224
Получение строк из входного потока 225
Преобразование входного потока 228
Чтение и запись файлов 230
Использование функции withFile 233
Время заключать в скобки 234
Хватай дескрипторы! 235
Список дел 237
Удаление заданий 238

Уборка ... 240
Аргументы командной строки ... 241
Ещё больше шалостей со списком дел ... 243
Многозадачный список задач ... 244
Работаем с некорректным вводом ... 248
Случайность ... 249
Подбрасывание монет ... 252
Ещё немного функций, работающих со случайностью ... 254
Случайность и ввод-вывод ... 255
Bytestring: тот же String, но быстрее ... 259
Строгие и ленивые ... 261
Копирование файлов при помощи Bytestring ... 263
Исключения ... 265
Обработка исключений, возникших в чистом коде ... 267
Обработка исключений ввода-вывода ... 273
Вспомогательные функции для работы с исключениями ... 279

10. Решение задач в функциональном стиле ... 281
Вычисление выражений в обратной польской записи ... 281
Вычисление выражений в ОПЗ ... 282
Реализация функции вычисления выражений в ОПЗ ... 283
Добавление новых операторов ... 286
Из аэропорта в центр ... 287
Вычисление кратчайшего пути ... 289
Представление пути на языке Haskell ... 291
Реализация функции поиска оптимального пути ... 293
Получение описания дорожной системы из внешнего источника ... 296

11. Аппликативные функторы ... 299
Функторы возвращаются ... 300
Действия ввода-вывода в качестве функторов ... 301
Функции в качестве функторов ... 304
Законы функторов ... 308
Закон 1 ... 308
Закон 2 ... 309
Нарушение закона ... 310
Использование аппликативных функторов ... 313
Поприветствуйте аппликативные функторы ... 315
Аппликативный функтор Maybe ... 316
Аппликативный стиль ... 318
Списки ... 320
Тип IO – тоже аппликативный функтор ... 323
Функции в качестве аппликативных функторов ... 325
Застёгиваемые списки ... 327
Аппликативные законы ... 329
Полезные функции для работы с аппликативными функторами ... 329

12. Моноиды ... 336
Оборачивание существующего типа в новый тип ... 336
Использование ключевого слова newtype для создания экземпляров классов типов ... 339
О ленивости newtype ... 341
Ключевое слово type против newtype и data ... 344
В общих чертах о моноидах ... 346
Класс типов Monoid ... 348
Законы моноидов ... 349
Познакомьтесь с некоторыми моноидами ... 350
Списки являются моноидами ... 350
Типы Product и Sum ... 352
Типы Any и All ... 354
Моноид Ordering ... 355
Моноид Maybe ... 359
Свёртка на моноидах ... 361

13. Пригоршня монад ... 367
Совершенствуем наши аппликативные функторы ... 367
Приступаем к типу Maybe ... 369
Класс типов Monad ... 373
Прогулка по канату ... 376
Код, код, код ... 377
Я улечу ... 379
Банан на канате ... 382
Нотация do ... 385
Делай как я ... 387
Пьер возвращается ... 388
Сопоставление с образцом и неудача в вычислениях ... 390
Списковая монада ... 392
Нотация do и генераторы списков ... 396
Класс MonadPlus и функция guard ... 396
Ход конём ... 399
Законы монад ... 402
Левая единица ... 402
Правая единица ... 404
Ассоциативность ... 405

14. Ещё немного монад ... 408
Writer? Я о ней почти не знаю! ... 409
Моноиды приходят на помощь ... 412
Тип Writer ... 414
Использование нотации do с типом Writer ... 416
Добавление в программы функции журналирования ... 417
Добавление журналирования в программы ... 418

Неэффективное создание списков ... 420
Разностные списки ... 422
Сравнение производительности ... 424
Монада Reader?Тьфу, опять эти шуточки! ... 425
Функции в качестве монад ... 426
Монада Reader ... 427
Вкусные вычисления с состоянием ... 429
Вычисления с состоянием ... 430
Стеки и чебуреки ... 431
Монада State ... 433
Получение и установка состояния ... 437
Случайность и монада State ... 438
Свет мой, Error, скажи, да всю правду доложи ... 439
Некоторые полезные монадические функции ... 442
liftM и компания ... 442
Функция join ... 446
Функция filterM ... 449
Функция foldM ... 452
Создание безопасного калькулятора выражений в обратной польской записи ... 454
Композиция монадических функций ... 457
Создание монад ... 459

15. Застёжки ... 467
Прогулка ... 468
Тропинка из хлебных крошек ... 471
Движемся обратно вверх ... 473
Манипулируем деревьями в фокусе ... 476
Идем прямо на вершину, где воздух чист и свеж! ... 478
Фокусируемся на списках ... 478
Очень простая файловая система ... 480
Создаём застёжку для нашей файловой системы ... 482
Манипулируем файловой системой ... 485
Осторожнее – смотрите под ноги! ... 486
Благодарю за то, что прочитали! ... 489

От издателя

Эта книга начиналась как англоязычный онлайновый учебник по языку Haskell, написанный словенским студентом, изучающим информатику, Мираном Липовачей (Miran Lipovača) в 2008 году. В мае 2009 года на сайте translated.by пользователь Дмитрий Леушин предложил первые главы учебника для перевода на русский язык. Его инициативу подхватил Александр Синицын. В переводе также принимали участие Виталий Капустян, Иван Терёхин, Дмитрий Крылов, пользователи olegchir, artobstrel95, Julia и другие. Оригинальный учебник оказался настолько популярным, что в 2011 году он был издан в печатном виде. Текст онлайнового издания при подготовке печатного издания был серьёзно исправлен и улучшен. Последние пять глав учебника переводились уже по печатному изданию Ясиром Арсанукаевым. Готовый текст был отредактирован Романом Душкиным. На втором этапе редактированием занимался Виталий Брагилевский; он также привёл текст первых десяти глав книги в соответствие с англоязычным печатным изданием и переработал текст раздела «Исключения». Оформление, вёрстка и другие технические работы выполнялись сотрудниками издательства «ДМК Пресс».

Предисловие

Когда в начале 2006 года я садился за свою первую книгу по функциональному программированию [2], в которой намеревался проиллюстрировать все теоретические положения при помощи языка Haskell, у меня возникали некоторые сомнения на сей счёт. Да, за плечами уже был пятилетний опыт чтения потоковых лекций по функциональному программированию в Московском Инженерно-Физическом Институте (МИФИ), для которых я и ввёл в учебный процесс этот замечательный язык вместо использовавшегося прежде языка Lisp. Однако в качестве методической основы тогда ещё не было практически ничего, кроме формального описания языка и нескольких статей. Существовало, впрочем, несколько книг о Haskell на английском языке [3, 4, 5, 7], но в те времена достать их было несколько затруднительно. Тем не менее я выбрал именно этот язык, поскольку создавать очередной том о функциональном программировании на Lisp (на каком-либо из его многочисленных диалектов) было бы нецелесообразно – такие книги имелись в избытке.

Сегодня можно уверенно сказать, что тогда я не ошибся в своём выборе. Развитие языка шло темпами набирающего скорость локомотива. Появлялись компиляторы (в том числе и полноценная среда разработки Haskell Platform), разнообразные утилиты для помощи в разработке, обширнейший набор библиотек, а главное – сложилось сообщество программистов! За несколько лет язык приобрел огромное количество почитателей, в том числе русскоязычных. Притом возник так называемый эффект «петли положительной обратной связи»: стремительно растущее сообщество стало ещё активнее развивать язык и всё, что с ним связано. И вот уже количество библиотек для Haskell насчитывает не одну тысячу, охватывая всевозможные задачи, встречающиеся в повседневном процессе коммерческой разработки. Выходят но-

вые книги, одна из которых [6] буквально взрывает общественное мнение. Теперь Haskell уже не воспринимается в качестве языка «нёрдов», получая статус вполне респектабельного средства программирования. На русском языке начинают выходить многочисленные переводы статей по Haskell (в том числе и официальные), основывается первый журнал, посвящённый функциональному программированию – «Практика функционального программирования» (ISSN 2075-8456).

И вот сегодня вы, уважаемый читатель, держите в руках переводное издание новой интересной книги о языке Haskell и основах реального программирования на нём. Эта публикация опять же стала возможной благодаря деятельности профессионального сообщества. Группа инициативных любителей языка Haskell перевела значительную часть текста, после чего издательством «ДМК Пресс», которое уже становится флагманом в деле издания книг о функциональном программировании в России, был проведён весь комплекс предпечатных работ – научное редактирование, корректура, вёрстка.

Миран Липовача – автор из Словении, который написал свою книгу «Изучай Haskell во имя добра», с тем чтобы сделать процесс освоения Haskell легким и весёлым. Оригинал книги, опубликованный в сети Интернет, написан в весьма вольном стиле – автор позволяет себе многочисленные жаргонизмы и простое (даже, можно сказать, простецкое) обращение с читателем. Текст дополнен многочисленными авторскими рисунками, предназначенными исключительно для развлечения читателя и не несущими особой смысловой нагрузки. Поначалу всё это заставляет предположить, что книга «несерьёзная», однако это впечатление обманчиво. Здесь представлено очень хорошее описание как базовых принципов программирования на Haskell, так и серьёзных идиом языка, пришедших из теории категорий (функторы, аппликативные функторы, монады). Притом автор пользуется очень простым языком и приводит доступные для понимания примеры. Вообще, книга насыщена разнообразными примерами, и это её положительная черта.

При работе над русским изданием коллектив переводчиков постарался сохранить своеобразный стиль автора, чтобы передать своеобразие оригинала. Однако в процессе научного редактирования некоторые моменты были сглажены, терминология приведена к единообразию и согласована с уже устоявшимися терминами

на русском языке. Тем не менее манера изложения материала далека от сухого академического стиля, который характерен для многих публикаций о функциональном программировании.

Напоследок, впрочем, стоит отметить и некоторые недостатки. Автор сам признаётся, что написал свою книгу с целью структуризации и классификации собственных знаний о языке Haskell. Так что к ней надо относиться с определённой долей осторожности, хотя в процессе научного редактирования не было обнаружено фактологических ошибок. Ещё один минус – полное отсутствие каких-либо сведений об инструментарии языка: читателю предлагается лишь скачать и установить Haskell Platform, а затем приступать к работе. Можно именно так и поступить, но вдумчивому читателю будет интересно узнать о способах использования инструментария. Этот пробел можно восполнить книгой [1].

В целом книгу Мирана Липовачи можно рекомендовать в качестве дополнительного источника информации о практическом использовании языка Haskell. Она будет полезна всем, кто интересуется функциональным программированием, равно как и студентам, обучающимся по специальностям, связанным с программированием и вычислительной техникой.

ДУШКИН Роман Викторович,
автор первых книг о языке Haskell на русском языке,
Москва, 2011 г.

Ссылки на источники

1. Душкин Р. В. *Практика работы на языке Haskell.* – М.: ДМК-Пресс, 2010. – 288 стр., ил. – ISBN 978-5-94074-588-4.
2. Душкин Р. В. *Функциональное программирование на языке Haskell.* – М.: ДМК-Пресс, 2007. – 608 стр., ил. – ISBN 5-94074-335-8.
3. Davie A. J. T. *Introduction to Functional Programming Systems Using Haskell.* – Cambridge University Press, 1992. – 304 p. – ISBN 0-52127-724-8.
4. Doets K., Eijck J. v. *The Haskell Road To Logic, Maths And Programming.* – King's College Publications, 2004. – 444 p. – ISBN 0-95430-069-6.
5. Hudak P. *The Haskell School of Expression: Learning Functional Programming through Multimedia.* – Cambridge University Press, 2000. – 382 p. – ISBN 0-52164-408-9.
6. O'Sullivan B., Goerzen J., Stewart D. *Real World Haskell.* – O'Reilly, 2008. – 710 p. – ISBN 0-596-51498-0.
7. Thompson S. *Haskell: The Craft of Functional Programming.* – Addison Wesley, 1999. – 512 p. – ISBN 0-20134-275-8.

Введение

Перед вами книга «Изучай Haskell во имя добра!» И раз уж вы взялись за её чтение, есть шанс, что вы хотите изучить язык Haskell. В таком случае вы на правильном пути – но прежде чем продолжить его, давайте поговорим о самом учебнике.

Я решился написать это руководство потому, что захотел упорядочить свои собственные знания о Haskell, а также потому, что надеюсь помочь другим людям в освоении этого языка. В сети Интернет уже предостаточно литературы по данной теме, и когда я сам проходил период ученичества, то использовал самые разные ресурсы.

Чтобы поподробнее ознакомиться с Haskell, я читал многочисленные справочники и статьи, в которых описывались различные аспекты при помощи различных методов. Затем я собрал воедино все эти разрозненные сведения и положил их в основу собственной книги. Так что этот учебник представляет собой попытку создать ещё один полезный ресурс для изучения языка Haskell – и есть вероятность, что вы найдёте здесь именно то, что вам нужно!

Эта книга рассчитана на людей, которые уже имеют опыт работы с императивными языками программирования (C++, Java, Python...), а теперь хотели бы попробовать Haskell. Впрочем, бьюсь об заклад, что даже если вы не обладаете солидным опытом программирования, с вашей природной смекалкой вы легко освоите Haskell, пользуясь этим учебником!

Моей первой реакцией на Haskell было ощущение, что язык какой-то уж слишком чудной. Но после преодоления начального барьера всё пошло как по маслу. Даже если на первый взгляд Haskell кажется вам странным, не сдавайтесь! Освоение этого языка похоже на изучение программирования «с нуля» – и это очень занимательно, потому что вы начинаете мыслить совершенно иначе...

> **ПРИМЕЧАНИЕ.** *IRC-канал `#haskell` на Freenode Network – отличный ресурс для тех, кто испытывает затруднения в обучении и хочет задать вопросы по какой-либо теме. Люди там чрезвычайно приятные, вежливые и с радостью помогают новичкам.*

Так что же такое Haskell?

Язык Haskell – это *чисто функциональный* язык программирования. В *императивных* языках результат достигается при передаче компьютеру последовательности команд, которые он затем выполняет. При этом компьютер может изменять своё состояние. Например, мы устанавливаем переменную `a` равной 5, производим какое-либо действие, а затем меняем её значение... Кроме того, у нас есть управляющие инструкции, позволяющие повторять несколько раз определённые действия, такие как циклы `for` и `while`. В чисто функциональных языках вы не говорите компьютеру, *как* делать те или иные вещи, – скорее вы говорите, что представляет собой ваша проблема.

Факториал числа – это произведение целых чисел от 1 до данного числа; сумма списка чисел – это первое число плюс сумма всех остальных чисел, и так далее. Вы можете выразить обе эти операции в виде *функций*. В функциональной программе нельзя присвоить переменной сначала одно значение, а затем какое-то другое. Если вы решили, что `a` будет равняться 5, то потом уже не сможете просто передумать и заменить значение на что-либо ещё. В конце концов, вы же сами сказали, что `a` равно 5! Вы что, врун какой-нибудь?

В чисто функциональных языках у функций *отсутствуют побочные эффекты*. Функция может сделать только одно: рассчитать что-нибудь и возвратить это как результат. Поначалу такое ограничение смущает, но в действительности оно имеет приятные последствия: если функция вызывается дважды с одними и теми же параметрами, это гарантирует, что оба раза вернётся одинаковый результат. Это свойство называется *ссылочной прозрачностью*. Оно позволяет

программисту легко установить (и даже доказать), что функция корректна, а также строить более сложные функции, объединяя простые друг с другом.

Haskell – *ленивый* язык. Это означает, что он не будет выполнять функции и производить вычисления, пока это действительно вам не потребовалось для вывода результата (если иное не указано явно). Подобное поведение возможно как раз благодаря ссылочной прозрачности. Если вы знаете, что результат функции зависит только от переданных ей параметров, неважно, в какой именно момент вы её вызываете. Haskell, будучи ленивым языком, пользуется этой возможностью и откладывает вычисления на то время, на какое это вообще возможно. Как только вы захотите отобразить результаты, Haskell проделает минимум вычислений, достаточных для их отображения. Ленивость также позволяет создавать бесконечные структуры данных, потому что реально вычислять требуется только ту часть структуры данных, которую необходимо отобразить.

Предположим, что у нас есть неизменяемый список чисел `xs = [1,2,3,4,5,6,7]` и функция `doubleMe` («УдвойМеня»), которая умножает каждый элемент на 2 и затем возвращает новый список. Если мы захотим умножить наш список на 8 в императивных языках, то сделаем так:

```
doubleMe(doubleMe(doubleMe(xs)))
```

При вызове, вероятно, будет получен список, а затем создана и возвращена копия. Затем список будет получен ещё два раза – с возвращением результата. В ленивых языках программирования вызов `doubleMe` со списком без форсирования получения результата означает, что программа скажет вам что-то вроде: «Да-да, я сделаю это позже!». Но когда вы захотите увидеть результат, то первая функция `doubleMe` скажет второй, что ей требуется результат, и немедленно! Вторая функция передаст это третьей, и та неохотно вернёт удвоенную 1, то есть 2.

Вторая получит и вернёт первой функции результат – 4. Первая увидит результат и выдаст вам 8. Так что потребуется только один

проход по списку, и он будет выполнен только тогда, когда действительно окажется необходим.

Язык Haskell – *статически типизированный* язык. Когда вы компилируете вашу программу, то компилятор знает, какой кусок кода – число, какой – строка и т. д. Это означает, что множество возможных ошибок будет обнаружено во время компиляции. Если, скажем, вы захотите сложить вместе число и строку, то компилятор вам «пожалуется».

В Haskell есть очень хорошая система типов, которая умеет автоматически делать вывод типов. Это означает, что вам не нужно описывать тип в каждом куске кода, потому что система типов может вычислить это сама. Если, скажем, `a = 5 + 4`, то вам нет необходимости говорить, что `a` – число, так как это может быть выведено автоматически. Вывод типов делает ваш код более универсальным. Если функция принимает два параметра и складывает их, а тип параметров не задан явно, то функция будет работать с любыми двумя параметрами, которые ведут себя как числа.

Haskell – *ясный* и *выразительный* язык, потому что он использует множество высокоуровневых идей; программы обычно короче, чем их императивные эквиваленты, их легче сопровождать, в них меньше ошибок.

Язык Haskell был придуман несколькими по-настоящему умными ребятами (с диссертациями). Работа по его созданию началась в 1987 году, когда комитет исследователей задался целью изобрести язык, который станет настоящей сенсацией. В 1999 году было опубликовано описание языка (Haskell Report), ознаменовавшее появление первой официальной его версии.

Что понадобится для изучения языка

Если коротко, то для начала понадобятся текстовый редактор и компилятор Haskell. Вероятно, у вас уже установлен любимый

редактор, так что не будем заострять на этом внимание. На сегодняшний день самым популярным компилятором Haskell является GHC (Glasgow Haskell Compiler), который мы и будем использовать в примерах ниже. Проще всего обзавестись им, скачав Haskell Platform, которая включает, помимо прочего, ещё и массу полезных библиотек. Для получения Haskell Platform нужно пойти на сайт http://hackage.haskell.org/platform/ и далее следовать инструкциям по вашей операционной системе.

GHC умеет компилировать сценарии на языке Haskell (обычно это файлы с расширением *.hs*), а также имеет интерактивный режим работы, в котором можно загрузить функции из файлов сценариев, вызвать их и тут же получить результаты. Во время обучения такой подход намного проще и эффективнее, чем перекомпиляция сценария при каждом его изменении, а затем ещё и запуск исполняемого файла.

Как только вы установите Haskell Platform, откройте новое окно терминала – если, конечно, используете Linux или Mac OS X. Если же у вас установлена Windows, запустите интерпретатор командной строки (*cmd.exe*). Далее введите `ghci` и нажмите **Enter**. Если ваша система не найдёт программу GHCi, попробуйте перезагрузить компьютер.

Если вы определили несколько функций в сценарии, скажем, *myfunctions.hs*, то их можно загрузить в GHCi, напечатав команду `:l myfunctions`. Нужно только убедиться, что файл *myfunctions.hs* находится в том же каталоге, из которого вы запустили GHCi.

Если вы изменили *hs*-сценарий, введите в интерактивном режиме `:l myfunctions`, чтобы загрузить его заново. Можно также перегрузить загруженный ранее сценарий с помощью команды `:r`. Обычно я поступаю следующим образом: определяю несколько функций в *hs*-файле, загружаю его в GHCi, экспериментирую с функциями, изменяю файл, перезагружаю его и затем всё повторяю. Собственно, именно этим мы с вами и займёмся.

Благодарности

Благодарю всех, кто присылал мне свои замечания, предложения и слова поддержки. Также благодарю Кита, Сэма и Мэрилин, которые помогли мне отшлифовать мастерство писателя.

1

НА СТАРТ, ВНИМАНИЕ, МАРШ!

Отлично, давайте начнём! Если вы принципиально не читаете предисловий к книгам, в данном случае вам всё же придётся вернуться назад и заглянуть в заключительную часть введения: именно там рассказано, что вам потребуется для изучения данного руководства и для загрузки программ.

Первое, что мы сделаем, – запустим компилятор GHC в интерактивном режиме и вызовем несколько функций, чтобы «прочувствовать» язык Haskell – пока ещё в самых общих чертах. Откройте консоль и наберите `ghci`. Вы увидите примерно такое приветствие:

```
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

Поздравляю – вы в GHCi!

ПРИМЕЧАНИЕ. *Приглашение консоли ввода –* `Prelude>`*, но поскольку оно может меняться в процессе работы, мы будем использовать просто* `ghci>`*. Если вы захотите, чтобы у вас было такое же приглашение, выполните команду* `:set prompt "ghci> "`*.*

Немного школьной арифметики:

```
ghci> 2 + 15
17
ghci> 49 * 100
```

```
4900
ghci> 1892 - 1472
420
ghci> 5 / 2
2.5
```

Код говорит сам за себя. Также в одной строке мы можем использовать несколько операторов; при этом работает обычный порядок вычислений. Можно использовать и круглые скобки для облегчения читаемости кода или для изменения порядка вычислений:

```
ghci> (50 * 100) - 4999
1
ghci> 50 * 100 - 4999
1
ghci> 50 * (100 - 4999)
-244950
```

Здорово, правда? Чувствую, вы со мной не согласны, но немного терпения! Небольшая опасность кроется в использовании отрицательных чисел. Если нам захочется использовать отрицательные числа, то всегда лучше заключить их в скобки. Попытка выполнения `5 * -3` приведёт к ошибке, зато `5 * (-3)` сработает как надо.

Булева алгебра в Haskell столь же проста. Как и во многих других языках программирования, в Haskell имеется два логических значения `True` и `False`, для конъюнкции используется операция `&&` (логическое «И»), для дизъюнкции – операция `||` (логическое «ИЛИ»), для отрицания – операция `not`.

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True&&True)
False
```

Можно проверить два значения на равенство и неравенство с помощью операций `==` и `/=`, например:

```
ghci> 5 == 5
True
ghci> 1 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 4
True
ghci> "привет" == "привет"
True
```

А что насчёт `5 + лама` или `5 == True`? Если мы попробуем выполнить первый фрагмент, то получим большое и страшное сообщение об ошибке[1]!

```
No instance for (Num [Char])
arising from a use of `+' at <interactive>:1:0-9
Possible fix: add an instance declaration for (Num [Char])
In the expression: 5 + "лама"
In the definition of `it': it = 5 + "лама"
```

Та-ак! GHCi говорит нам, что `лама` не является числом, и непонятно, как это прибавить к 5. Даже если вместо `лама` подставить `четыре` или `4`, Haskell всё равно не будет считать это числом! Операция + ожидает, что аргументы слева и справа будут числовыми. Если же мы попытаемся посчитать `True == 5`, GHCi опять скажет нам, что типы не совпадают.

Несмотря на то что операция + производится только в отношении элементов, воспринимаемых как число, операция сравнения (==), напротив, применима к любой паре элементов, которые можно сравнить. Фокус заключается в том, что они должны быть одного типа. Вы не сможете сравнивать яблоки и апельсины. В подробностях мы это обсудим чуть позже.

ПРИМЕЧАНИЕ. *Запись* `5 + 4.0` *вполне допустима, потому что* `5` *может вести себя как целое число или как число с плавающей точкой.* `4.0` *не может выступать в роли целого числа, поэтому именно число* `5` *должно «подстроиться».*

---

[1] В современных версиях интерпретатора GHCi для печати результатов вычислений используется функция `show`, которая представляет кириллические символы соответствующими числовыми кодами Unicode. Поэтому в следующем листинге вместо строки "лама" будет фактически выведено "\1083\1072\1084\1072". В тексте книги для большей понятности кириллица в результатах оставлена без изменений. – *Прим. ред.*

Вызов функций

Возможно, вы этого пока не осознали, но все это время мы использовали функции. Например, операция `*` – это функция, которая принимает два числа и перемножает их. Как вы видели, мы вызываем её, вставляя символ `*` между числами. Это называется «*инфиксной* записью».

Обычно функции являются префиксными, поэтому в дальнейшем мы не будем явно указывать, что функция имеет префиксную форму – это будет подразумеваться. В большинстве императивных языков функции вызываются указанием имени функции, а затем её аргументов (как правило, разделенных запятыми) в скобках. В языке Haskell функции вызываются указанием имени функции и – через пробел – параметров, также разделенных пробелами. Для начала попробуем вызвать одну из самых скучных функций языка:

```
ghci> succ 8
9
```

Функция `succ` принимает на вход любое значение, которое может иметь последующее значение, после чего возвращает именно последующее значение. Как вы видите, мы отделяем имя функции от параметра пробелом. Вызывать функции с несколькими параметрами не менее просто.

Функции `min` и `max` принимают по два аргумента, которые можно сравнивать (как и числа!), и возвращают большее или меньшее из значений:

```
ghci> min 9 10
9
ghci> min 3.4 3.2
3.2
ghci> max 100 101
101
```

Операция применения функции (то есть вызов функции с указанием списка параметров через пробел) имеет наивысший приоритет. Для нас это значит, что следующие два выражения эквивалентны:

```
ghci> succ 9 + max 5 4 + 1
16
ghci> (succ 9) + (max 5 4) + 1
16
```

Однако если мы хотим получить значение, следующее за произведением чисел 9 и 10, мы не можем написать `succ 9 * 10`, потому что это даст значение, следующее за 9 (т. е. 10), умноженное на 10, т. е. 100. Следует написать `succ (9 * 10)`, чтобы получить 91.

Если функция принимает ровно два параметра, мы также можем вызвать её в инфиксной форме, заключив её имя в обратные апострофы. Например, функция `div` принимает два целых числа и выполняет их целочисленное деление:

```
ghci> div 92 10
9
```

Но если мы вызываем её таким образом, то может возникнуть неразбериха с тем, какое из чисел делимое, а какое делитель. Поэтому можно вызвать функцию в инфиксной форме, что, как оказывается, гораздо понятнее[2]:

```
ghci> 92 `div` 10
9
```

Многие люди, перешедшие на Haskell с императивных языков, придерживаются мнения, что применение функции должно обозначаться скобками. Например, в языке C используются скобки для вызова функций вроде `foo()`, `bar(1)` или `baz(3, ха-ха)`. Однако, как мы уже отмечали, для применения функций в Haskell предусмотрены пробелы. Поэтому вызов соответствующих функций производится следующим образом: `foo`, `bar 1` и `baz 3 ха-ха`. Так что если вы увидите выражение вроде `bar (bar 3)`, это не значит, что `bar` вызывается с параметрами `bar` и `3`. Это значит, что мы сначала вызываем функцию `bar` с параметром `3`, чтобы получить некоторое число, а затем опять вызываем `bar` с этим числом в качестве параметра. В языке C это выглядело бы так: “`bar(bar(3))`”.

[2] На самом деле любую функцию, число параметров которой больше одного, можно записать в инфиксной форме, заключив её имя в обратные апострофы и поместив её в таком виде ровно между первым и вторым аргументом. – *Прим. ред.*

Функции: первые шаги

Определяются функции точно так же, как и вызываются. За именем функции следуют параметры[3], разделенные пробелами. Но при определении функции есть ещё символ =, а за ним – описание того, что функция делает. В качестве примера напишем простую функцию, принимающую число и умножающую его на 2. Откройте свой любимый текстовый редактор и наберите в нём:

```
doubleMe x = x + x
```

Сохраните этот файл, например, под именем *baby.hs*. Затем перейдите в каталог, в котором вы его сохранили, и запустите оттуда GHCi. В GHCi выполните команду `:l baby`. Теперь наш сценарий загружен, и можно поупражняться с функцией, которую мы определили:

```
ghci> :l baby
[1 of 1] Compiling Main             ( baby.hs, interpreted )
Ok, modules loaded: Main.
ghci> doubleMe 9
18
ghci> doubleMe 8.3
16.6
```

Поскольку операция + применима как к целым числам, так и к числам с плавающей точкой (на самом деле – ко всему, что может быть воспринято как число), наша функция одинаково хорошо работает с любыми числами. А теперь давайте напишем функцию, которая принимает два числа, умножает каждое на два и складывает их друг с другом. Допишите следующий код в файл *baby.hs*:

```
doubleUs x y = x*2 + y*2
```

ПРИМЕЧАНИЕ. *Функции в языке Haskell могут быть определены в любом порядке. Поэтому совершенно неважно, в какой последовательности приведены функции в файле baby.hs.*

[3] На самом деле в определении функций они называются образцами, но об этом пойдёт речь далее. – *Прим. ред.*

Теперь сохраните файл и введите `:l baby` в GHCi, чтобы загрузить новую функцию. Результаты вполне предсказуемы:

```
ghci> doubleUs 4 9
26
ghci> doubleUs 2.3 34.2
73.0
ghci> doubleUs 28 88 + doubleMe 123
478
```

Вы можете вызывать свои собственные функции из других созданных вами же функций. Учитывая это, можно переопределить `doubleUs` следующим образом:

```
doubleUs x y = doubleMe x + doubleMe y
```

Это очень простой пример общего подхода, применяемого во всём языке – создание простых базовых функций, корректность которых очевидна, и построение более сложных конструкций на их основе.

Кроме прочего, подобный подход позволяет избежать дублирования кода. Например, представьте себе, что какие-то «математики» решили, будто 2 – это на самом деле 3, и вам нужно изменить свою программу. Тогда вы могли бы просто переопределить `doubleMe` как `x + x + x`, и поскольку `doubleUs` вызывает `doubleMe`, данная функция автоматически работала бы в странном мире, где 2 – это 3.

Теперь давайте напишем функцию, умножающую число на два, но только при условии, что это число меньше либо равно 100 (поскольку все прочие числа и так слишком большие!):

```
doubleSmallNumber x = if x > 100
                        then x
                        else x*2
```

Мы только что воспользовались условной конструкцией `if` в языке Haskell. Возможно, вы уже знакомы с условными операторами из других языков. Разница между условной конструкцией `if` в Haskell и операторами `if` из императивных языков заключается в том, что ветвь `else` в языке Haskell является обязательной. В императивных языках вы можете просто пропустить пару шагов, если

условие не выполняется, а в Haskell каждое выражение или функция должны что-то возвращать[4].

Можно было бы написать конструкцию `if` в одну строку, но я считаю, что это не так «читабельно». Ещё одна особенность условной конструкции в языке Haskell состоит в том, что она является выражением. *Выражение* – это код, возвращающий значение. `5` – это выражение, потому что возвращает 5; `4 + 8` – выражение, `x + y` – тоже выражение, потому что оно возвращает сумму *x* и *y*.

Поскольку ветвь `else` обязательна, конструкция `if` всегда что-нибудь вернёт, ибо является выражением. Если бы мы хотели добавить единицу к любому значению, получившемуся в результате выполнения нашей предыдущей функции, то могли бы написать её тело вот так:

```
doubleSmallNumber' x = (if x > 100 then x else x*2) + 1
```

Если опустить скобки, то единица будет добавляться только при условии, что `x` не больше 100. Обратите внимание на символ апострофа (`'`) в конце имени функции. Он не имеет специального значения в языке Haskell. Это допустимый символ для использования в имени функции.

Обычно мы используем символ прямого апострофа () для обозначения строгой (не ленивой) версии функции либо слегка модифицированной версии функции или переменной. Поскольку апостроф – допустимый символ в именах функций, мы можем определять такие функции:

```
conanO'Brien = "Это я, Конан О'Брайен!"
```

Здесь следует обратить внимание на две важные особенности. Во-первых, в названии функции мы не пишем имя `conan` с прописной буквы. Дело в том, что наименования функций не могут начинаться с прописной буквы – чуть позже мы разберёмся, почему. Во-вторых, данная функция не принимает никаких параметров.

Когда функция не принимает аргументов, говорят, что это *константная* функция. Поскольку мы не можем изменить содержание имён (и функций) после того, как их определили, идентификатор

---

[4] Вообще говоря, конструкцию с `if` можно определить в виде функции:

```
if' :: Bool -> a -> a -> a
if' True x _ = x
if' False _ y = y
```

Конструкция введена в язык Haskell на уровне ключевого слова для того, чтобы минимизировать количество скобок в условных выражениях. – *Прим. ред.*

conanO'Brien и строка "Это я, Конан О'Брайен!" могут использоваться взаимозаменяемо.

Списки

Как и списки покупок в реальном мире, списки в языке Haskell очень полезны. В данном разделе мы рассмотрим основы работы со списками, генераторами списков и строками (которые также являются списками).

Списки в языке Haskell являются *гомогенными* структурами данных; это означает, что в них можно хранить элементы только одного типа. Можно иметь список целых или список символов, но нельзя получить список с целыми числами и символами одновременно.

Списки заключаются в квадратные скобки, а элементы разделяются запятыми:

```
ghci> let lostNumbers = [4,8,15,16,23,42]
ghci> lostNumbers
[4,8,15,16,23,42]
```

ПРИМЕЧАНИЕ. *Можно использовать ключевое слово* `let`*, чтобы определить имя прямо в GHCi. Например, выполнение* `let a = 1` *из GHCi – эквивалент указания* `a = 1` *в скрипте с последующей загрузкой.*

Конкатенация

Объединение двух списков – стандартная задача. Она выполняется с помощью оператора ++[5].

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
ghci> "привет" ++ " " ++ "мир"
"привет мир"
```

[5] Следует отметить, что *операторами* называются двухместные инфиксные функции, имена которых состоят из служебных символов: `+`, `*`, `>>=` и т. д. – *Прим. ред.*

```
ghci> ['в','о'] ++ ['-'] ++  ['о','т']
"во-от"
```

ПРИМЕЧАНИЕ. *Строки в языке Haskell являются просто списками символов. Например, строка* `привет` *– это то же самое, что и список* `['п','р','и','в','е','т']`*. Благодаря этому для работы со строками можно использовать функции обработки символов, что очень удобно.*

Будьте осторожны при использовании оператора ++ с длинными строками. Если вы объединяете два списка (даже если в конец первого из них дописывается второй, состоящий из одного элемента, например `[1,2,3] ++ [4]`), то язык Haskell должен обойти весь список с левой стороны от ++. Это не проблема, когда обрабатываются небольшие списки, но добавление к списку из 50 000 000 элементов займет много времени. А вот если вы добавите что-нибудь в начало списка с помощью оператора `:` (также называемого «*cons*»), долго ждать не придётся.

```
ghci> 'В':"ОТ КОШКА"
"ВОТ КОШКА"
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

Обратите внимание, что оператор `:` принимает число и список чисел или символ и список символов, в то время как ++ принимает два списка. Даже если вы добавляете один элемент в конец списка с помощью оператора ++, следует заключить этот элемент в квадратные скобки, чтобы он стал списком:

```
ghci> [1,2,3,4] ++ [5]
[1,2,3,4,5]
```

Написать `[1,2,3,4] ++ 5` нельзя, потому что оба параметра оператора ++ должны быть списками, а `5` – это не список, а число.

Интересно, что `[1,2,3]` – это на самом деле синтаксический вариант `1:2:3:[]`. Список `[]` – пустой, и если мы добавим к его началу 3, получится `[3]`; если затем добавим в начало 2, получится `[2,3]` и т. д.

ПРИМЕЧАНИЕ. *Списки* [], [[]] *и* [[],[],[]] *совершенно разные. Первый – это пустой список; второй – список, содержащий пустой список; третий – список, содержащий три пустых списка.*

Обращение к элементам списка

Если вы хотите извлечь элемент из списка по индексу, используйте оператор `!!`. Индексы начинаются с нуля.

```
ghci> "Стив Бушеми" !! 5
'Б'
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

Но если вы попытаетесь получить шестой элемент списка, состоящего из четырёх элементов, то получите сообщение об ошибке, так что будьте осторожны!

Списки списков

Списки могут содержать другие списки. Также они могут содержать списки, которые содержат списки, которые содержат списки...

```
ghci> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
ghci> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b !! 2
[1,2,2,3,4]
```

Вложенные списки могут быть разной длины, но не могут быть разных типов. Подобно тому как нельзя создать список, содержащий несколько символов и несколько чисел, нельзя создать и список, содержащий несколько списков символов и несколько списков чисел.

Сравнение списков

Списки можно сравнивать, только если они содержат сравнимые элементы. При использовании операторов <, <=, >= и > сравнение происходит в лексикографическом порядке. Сначала сравниваются «головы» списков; если они равны, то сравниваюся вторые элементы. Если равны и вторые элементы, то сравниваются третьи – и т. д., пока не будут найдены различающиеся элементы. Результат сравнения списков определяется по результату сравнения первой пары различающихся элементов.

Сравним для примера `[3,4,2]<[3,4,3]`. Haskell видит, что `3` и `3` равны, поэтому переходит к сравнению `4` и `4`, но так как они тоже равны, сравнивает `2` и `3`. Число `2` меньше `3`, поэтому первый список меньше второго. Аналогично выполняется сравнение на <=, >= и >:

```
ghci> [3,2,1] > [2,1,0]
True
ghci> [3,2,1] > [2,10,100]
True
ghci> [3,4,2] < [3,4,3]
True
ghci> [3,4,2] > [2,4]
True
ghci> [3,4,2] == [3,4,2]
True
```

Непустой список всегда считается больше, чем пустой. Это позволяет сравнивать друг с другом любые два списка, даже если один из них точно совпадает с началом другого.

Другие операции над списками

Что ещё можно делать со списками? Вот несколько основных функций работы с ними.

Функция `head` принимает список и возвращает его головной элемент. Головной элемент списка – это, собственно, его первый элемент.

```
ghci> head [5,4,3,2,1]
5
```

Функция `tail` принимает список и возвращает его «хвост». Иными словами, эта функция отрезает «голову» списка и возвращает остаток.

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

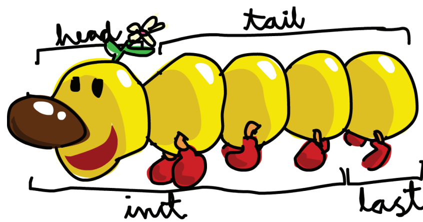
Функция `last` принимает список и возвращает его последний элемент.

```
ghci> last [5,4,3,2,1]
1
```

Функция `init` принимает список и возвращает всё, кроме его последнего элемента.

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

Если представить список в виде сороконожки, то с функциями получится примерно такая картина:



Но что будет, если мы попытаемся получить головной элемент пустого списка?

```
ghci> head []
*** Exception: Prelude.head: empty list
```

Ну и ну! Всё сломалось!.. Если нет сороконожки, нет и «головы». При использовании функций `head`, `tail`, `last` и `init` будьте осторожны – не применяйте их в отношении пустых списков. Эту ошибку нельзя отловить на этапе компиляции, так что всегда полезно предотвратить случайные попытки попросить язык Haskell выдать несколько элементов из пустого списка.

Функция `length`, очевидно, принимает список и возвращает его длину:

```
ghci> length [5,4,3,2,1]
5
```

Функция `null` проверяет, не пуст ли список. Если пуст, функция возвращает `True`, в противном случае – `False`. Используйте эту функцию вместо `xs == []` (если у вас есть список с именем `xs`).

```
ghci> null [1,2,3]
False
ghci> null []
True
```

Функция `reverse` обращает список (расставляет его элементы в обратном порядке).

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

Функция `take` принимает число и список. Она извлекает соответствующее числовому параметру количество элементов из начала списка:

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
```

Обратите внимание, что если попытаться получить больше элементов, чем есть в списке, функция возвращает весь список. Если мы пытаемся получить 0 элементов, функция возвращает пустой список.

Функция `drop` работает сходным образом, но отрезает указанное количество элементов с начала списка:

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

Функция `maximum` принимает список, состоящий из элементов, которые можно упорядочить, и возвращает наибольший элемент.

Функция `minimum` возвращает наименьший элемент.

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
```

Функция `sum` принимает список чисел и возвращает их сумму.

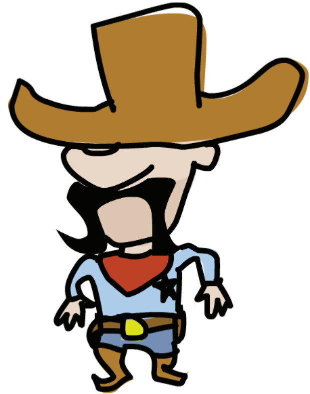
Функция `product` принимает список чисел и возвращает их произведение.

```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

Функция `elem` принимает элемент и список элементов и проверяет, входит ли элемент в список. Обычно эта функция вызывается как инфиксная, поскольку так её проще читать:

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

Интервалы



А что если нам нужен список всех чисел от 1 до 20? Конечно, мы могли бы просто набрать их подряд, но, очевидно, это не решение для джентльмена, требующего совершенства от языка программирования. Вместо этого мы будем использовать интервалы. *Интервалы* – это способ создания списков, являющихся арифметическими последовательностями элементов, которые можно перечислить по порядку: один, два, три, четыре и т. п. Символы тоже могут быть перечислены: например, алфавит – это перечень символов от A до Z. А вот имена перечислить нельзя. (Какое, например, имя будет идти после «Иван»? Лично я понятия не имею!)

Чтобы создать список, содержащий все натуральные числа от 1 до 20, достаточно написать `[1..20]`. Это эквивалентно полной записи `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]`, и единственная разница в том, что записывать каждый элемент списка, как показано во втором варианте, довольно глупо.

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

Интервалы замечательны ещё и тем, что они позволяют указать шаг. Что если мы хотим внести в список все чётные числа от 1 до 20? Или каждое третье число от 1 до 20?

```
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
```

Нужно всего лишь поставить запятую между первыми двумя элементами последовательности и указать верхний предел диапазона. Но, хотя интервалы достаточно «умны», на их сообразительность не всегда следует полагаться. Вы не можете написать `[1,2,4,8,16..100]` и после этого ожидать, что получите все степени двойки. Во-первых, потому, что при определении интервала можно указать только один шаг. А во-вторых, потому что некоторые последовательности, не являющиеся арифметическими, неоднозначны, если представлены только несколькими первыми элементами.

ПРИМЕЧАНИЕ. *Чтобы создать список со всеми числами от 20 до 1 по убыванию, вы не можете просто написать [20..1], а должны написать [20,19..1]. При попытке записать такой интервал без шага (т. е. [20..1]) Haskell начнёт с пустого списка, а затем будет увеличивать начальный элемент на единицу, пока не достигнет или не превзойдет элемент в конце интервала. Поскольку 20 уже превосходит 1, результат окажется просто пустым списком.*

Будьте осторожны при использовании чисел с плавающей точкой в интервалах! Из-за того что они не совсем точны (по определению), их использование в диапазонах может привести к весьма забавным результатам.

```
ghci> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

Мой совет: *не используйте такие числа в интервалах*!

Интервалы, кроме прочего, можно использовать для создания бесконечных списков, просто не указывая верхний предел. Позже мы рассмотрим этот вариант в подробностях. А сейчас давайте посмотрим, как можно получить список первых 24 чисел, кратных 13. Конечно, вы могли бы написать `[13,26..24*13]`. Но есть способ получше: `take 24 [13,26..]`. Поскольку язык Haskell ленив, он не будет пытаться немедленно вычислить бесконечный список, потому что процесс никогда не завершится. Он подождет, пока вы не захотите получить что-либо из такого списка. Тут-то обнаружится, что вы хотите получить только первые 24 элемента, что и будет исполнено.

Немного функций, производящих бесконечные списки:

- Функция `cycle` принимает список и зацикливает его в бесконечный. Если вы попробуете отобразить результат, на это уйдёт целая вечность, поэтому вам придётся где-то его обрезать.

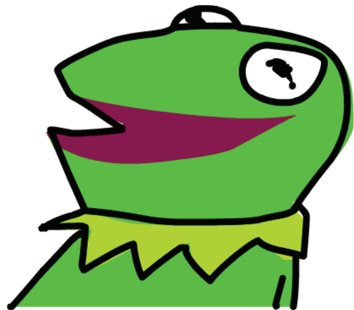
```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
```

- Функция `repeat` принимает элемент и возвращает бесконечный список, состоящий только из этого элемента. Это подобно тому, как если бы вы зациклили список из одного элемента.

```
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

Однако проще использовать функцию `replicate`, если вам нужен список из некоторого количества одинаковых элементов. `replicate 3 10` вернёт `[10,10,10]`.

Генераторы списков



Если вы изучали курс математики, то, возможно, сталкивались со способом задания множества путём описания характерных свойств, которыми должны обладать его элементы. Обычно этот метод используется для построения подмножеств из множеств.

Вот пример простого описания множества. Множество, состоящее из первых десяти чётных чисел, это $S = \{2 \cdot x \mid x \in N, x \leq 10\}$, где выражение перед символом | называется *производящей функцией* (out put function), x – переменная, N – входной набор, а $x \leq 10$ – условие выборки. Это означает, что множество содержит удвоенные натуральные числа, которые удовлетворяют условию выборки.

Если бы нам потребовалось написать то же самое на языке Haskell, можно было бы изобрести что-то вроде: `take 10 [2,4..]`. Но что если мы хотим не просто получить первые десять удвоенных нату-

ральных чисел, а применить к ним некую более сложную функцию? Для этого можно использовать *генератор списков*. Он очень похож на описание множеств:

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

В выражении `[x*2 | x <- [1..10]]` мы извлекаем элементы из списка `[1..10]`, т. е. `x` последовательно принимает все значения элементов списка. Иногда говорят, что `x` *связывается* с каждым элементом списка. Часть генератора, находящаяся левее вертикальной черты |, определяет значения элементов результирующего списка. В нашем примере значения `x`, извлеченные из списка `[1..10]`, умножаются на два.

Теперь давайте добавим к этому генератору условие выборки (*предикат*). Условия идут после задания источника данных и отделяются от него запятой. Предположим, что нам нужны только те элементы, которые, будучи удвоенными, больше либо равны 12.

```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
```

Это работает. Замечательно! А как насчёт ситуации, когда требуется получить все числа от 50 до 100, остаток от деления на 7 которых равен 3? Легко!

```
ghci> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
```

И снова получилось!

ПРИМЕЧАНИЕ. *Заметим, что прореживание списков с помощью условий выборки также называется фильтрацией.*

Мы взяли список чисел и отфильтровали их условиями. Теперь другой пример. Давайте предположим, что нам нужно выражение, которое заменяет каждое нечётное число больше 10 на `БАХ!"`, а каждое нечётное число меньше 10 – на `БУМ!"`. Если число чётное, мы выбрасываем его из нашего списка. Для удобства поместим выражение в функцию, чтобы потом легко использовать его повторно.

```
boomBangs xs = [if x < 10 then "БУМ!" else "БАХ!" | x <- xs, odd x]
```

ПРИМЕЧАНИЕ. *Помните, что если вы пытаетесь определить эту функцию в GHCi, то перед ее именем нужно написать* `let`. *Если же вы описываете ее в отдельном файле, а потом загружаете его в GHCi, то никакого* `let` *не требуется.*

Последняя часть описания – условие выборки. Функция `odd` возвращает значение `True` для нечётных чисел и `False` – для чётных. Элемент включается в список, только если все условия выборки возвращают значение `True`.

```
ghci> boomBangs [7..13]
["БУМ!","БУМ!","БАХ!","БАХ!"]
```

Мы можем использовать несколько условий выборки. Если бы потребовалось получить все числа от 10 до 20, кроме 13, 15 и 19, то мы бы написали:

```
ghci> [x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
```

Можно не только написать несколько условий выборки в генераторах списков (элемент должен удовлетворять всем условиям, чтобы быть включённым в результирующий список), но и выбирать элементы из нескольких списков. В таком случае выражения перебирают все комбинации из данных списков и затем объединяют их по производящей функции, которую мы указали:

```
ghci> [x+y | x <- [1,2,3], y <- [10,100,1000]]
[11,101,1001,12,102,1002,13,103,1003]
```

Здесь `x` берётся из списка `[1,2,3]`, а `y` – из списка `[10,100,1000]`. Эти два списка комбинируются следующим образом. Во-первых, `x` становится равным 1, а `y` последовательно принимает все значения из списка `[10,100,1000]`. Поскольку значения `x` и `y` складываются, в начало результирующего списка помещаются числа `11`, `101` и `1001` (`1` прибавляется к `10`, `100`, `1000`). После этого `x` становится равным `2` и всё повторяется, к списку добавляются числа `12`, `102` и `1002`. То же самое происходит для `x` равного `3`.

Таким образом, каждый элемент `x` из списка `[1,2,3]` всеми возможными способами комбинируется с каждым элементом `y` из списка `[10,100,1000]`, а `x+y` используется для построения из этих комбинаций результирующего списка.

Вот другой пример: если у нас есть два списка [2,5,10] и [8,10,11], и мы хотим получить произведения всех возможных комбинаций из элементов этих списков, то можно использовать следующее выражение:

```
ghci> [x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

Как и ожидалось, длина нового списка равна 9.

Допустим, нам потребовались все возможные произведения, которые больше 50:

```
ghci> [x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]
```

А как насчёт списка, объединяющего элементы списка прилагательных с элементами списка существительных... с довольно забавным результатом?

```
ghci> let nouns = ["бродяга","лягушатник","поп"]
ghci> let adjs = ["ленивый","ворчливый","хитрый"]
ghci> [adj ++ " " ++ noun | adj <- adjs, noun <- nouns]
["ленивый бродяга","ленивый лягушатник","ленивый поп",
"ворчливый бродяга","ворчливый лягушатник", "ворчливый поп",
"хитрый бродяга","хитрый лягушатник","хитрый поп"]
```

Генераторы списков можно применить даже для написания своей собственной функции `length`! Назовем ее `length'`: эта функция будет заменять каждый элемент списка на 1, а затем мы все эти единицы просуммируем функцией `sum`, получив длину списка:

```
length' xs = sum [1 | _ <- xs]
```

Символ _ означает, что нам неважно, что будет получено из списка, поэтому вместо того, чтобы писать имя образца, которое мы никогда не будем использовать, мы просто пишем _. Поскольку строки – это списки, генератор списков можно использовать для обработки и создания строк. Вот функция, которая принимает строку и удаляет из неё всё, кроме букв в верхнем регистре:

```
removeNonUppercase st = [c | c <- st, c `elem` ['А'..'Я']]
```

Всю работу здесь выполняет предикат: символ будет добавляться в новый список, только если он является элементом списка `['А'..'Я']`. Загрузим функцию в GHCi и проверим:

```
ghci> removeNonUppercase "Ха-ха-ха! А-ха-ха-ха!"
"ХА"
ghci> removeNonUppercase "ЯнеЕМЛЯГУШЕК"
"ЯЕМЛЯГУШЕК"
```

Вложенные генераторы списков также возможны, если вы работаете со списками, содержащими вложенные списки. Допустим, список содержит несколько списков чисел. Попробуем удалить все нечётные числа, не разворачивая список:

```
ghci> let xxs = [[1,3,5,2,3,1,2],[1,2,3,4,5,6,7],[1,2,4,2,1,6,3,1,3,2]]
ghci> [[x | x <- xs, even x ] | xs <- xxs]
[[2,2],[2,4,6],[2,4,2,6,2]]
```

ПРИМЕЧАНИЕ. *Вы можете писать генераторы списков в несколько строк. Поэтому, если вы не в GHCi, лучше разбить длинные генераторы списков, особенно вложенные, на несколько строк.*

Кортежи

Кортежи позволяют хранить несколько элементов разных типов как единое целое.

В некотором смысле кортежи похожи на списки, однако есть и фундаментальные отличия. Во-первых, кортежи гетерогенны, т. е. в одном кортеже можно хранить элементы нескольких различных типов. Во-вторых, кортежи имеют фиксированный размер: необходимо заранее знать, сколько именно элементов потребуется сохранить.

Кортежи обозначаются круглыми скобками, а их компоненты отделяются запятыми:

```
ghci> (1, 3)
(1,3)
ghci> (3, 'a', "привет")
```

```
(3,'a',"привет")
ghci> (50, 50.4, "привет", 'b')
(50,50.4,"привет",'b')
```

Использование кортежей

Подумайте о том, как бы мы представили двумерный вектор в языке Haskell. Один вариант – использовать список. Это могло бы сработать – ну а если нам нужно поместить несколько векторов в список для представления точек фигуры на двумерной плоскости?.. Мы могли бы, например, написать: `[[1,2],[8,11],[4,5]]`.

Проблема подобного подхода в том, что язык Haskell не запретит задать таким образом нечто вроде `[[1,2],[8,11,5],[4,5]]` – ведь это по-прежнему будет список списков с числами. Но по сути данная запись не имеет смысла. В то же время кортеж с двумя элементами (также называемый «парой») имеет свой собственный тип; это значит, что список не может содержать несколько пар, а потом «тройку» (кортеж размера 3). Давайте воспользуемся этим вариантом. Вместо того чтобы заключать векторы в квадратные скобки, применим круглые: `[(1,2),(8,11),(4,5)]`. А что произошло бы, если б мы попытались создать такую комбинацию: `[(1,2),(8,11,5),(4,5)]`? Получили бы ошибку:

```
Couldn't match expected type `(t, t1)'
against inferred type `(t2, t3, t4)'
In the expression: (8, 11, 5)
In the expression: [(1, 2), (8, 11, 5), (4, 5)]
In the definition of `it': it = [(1, 2), (8, 11, 5), (4, 5)]
```

Мы попытались использовать пару и тройку в одном списке, и нас предупреждают: такого не должно быть. Нельзя создать и список вроде `[(1,2),("Один",2)]`, потому что первый элемент списка – это пара чисел, а второй – пара, состоящая из строки и числа.

Кортежи также можно использовать для представления широкого диапазона данных. Например, если бы мы хотели представить чьё-либо полное имя и возраст в языке Haskell, то могли бы воспользоваться тройкой: `("Кристофер", "Уокен", 69)`. Как видно из этого примера, кортежи также могут содержать списки.

Используйте кортежи, когда вы знаете заранее, из скольких элементов будет состоять некоторая часть данных. Кортежи гораз-

до менее гибки, поскольку количество и тип элементов образуют тип кортежа, так что вы не можете написать общую функцию, чтобы добавить элемент в кортеж – понадобится написать функцию, чтобы добавить его к паре, функцию, чтобы добавить его к тройке, функцию, чтобы добавить его к четвёрке, и т. д.

Как и списки, кортежи можно сравнить друг с другом, если можно сравнивать их компоненты. Однако вам не удастся сравнить кортежи разных размеров (хотя списки разных размеров сравниваются, если можно сравнивать их элементы).

Несмотря на то что есть списки с одним элементом, не бывает кортежей с одним компонентом. Если вдуматься, это неудивительно. Кортеж с единственным элементом был бы просто значением, которое он содержит, и, таким образом, не давал бы нам никаких дополнительных возможностей[6].

Использование пар

Вот две полезные функции для работы с парами:

- `fst` – принимает пару и возвращает её первый компонент.

```
ghci> fst (8,11)
8
ghci> fst ("Bay", False)
"Bay"
```

- `snd` – принимает пару и возвращает её второй компонент. Неожиданно!

```
ghci> snd (8,11)
11
ghci> snd ("Bay", False)
False
```

ПРИМЕЧАНИЕ. *Эти функции работают только с парами. Они не будут работать с тройками, четвёрками, пятёрками и т. д. Выделение данных из кортежей мы рассмотрим чуть позже.*

Замечательная функция, производящая список пар, – `zip`. Она принимает два списка и сводит их в один, группируя соответствующие элементы в пары. Это очень простая, но крайне полезная функция. Особенно она полезна, когда вы хотите объединить два

[6] Однако есть нульместный кортеж, обозначаемый в языке Haskell как (). – *Прим. ред.*

списка или обойти два списка одновременно. Продемонстрируем работу `zip`:

```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> zip [1 .. 5] ["один", "два", "три", "четыре", "пять"]
[(1,"один"),(2,"два"),(3,"три"),(4,"четыре"),(5,"пять")]
```

Функция «спаривает» элементы и производит новый список. Первый элемент идёт с первым, второй – со вторым и т. д. Обратите на это внимание: поскольку пары могут содержать разные типы, функция `zip` может принять два списка, содержащих разные типы, и объединить их. А что произойдёт, если длина списков не совпадает?

```
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["я","не","черепаха"]
[(5,"я"),(3,"не"),(2,"черепаха")]
```

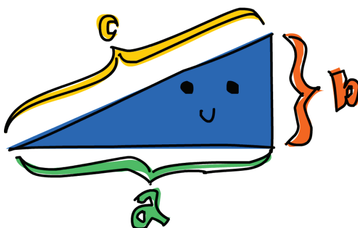
Более длинный список просто обрезается до длины более короткого! Поскольку язык Haskell ленив, мы можем объединить бесконечный список с конечным:

```
ghci> zip [1..] ["яблоко", "апельсин", "вишня", "манго"]
[(1,"яблоко"),(2,"апельсин"),(3,"вишня"),(4,"манго")]
```

В поисках прямоугольного треугольника

Давайте закончим главу задачей, в решении которой пригодятся и генераторы списков, и кортежи. Предположим, что требуется найти прямоугольный треугольник, удовлятворяющий всем следующим условиям:

- длины сторон являются целыми числами;
- длина каждой стороны меньше либо равна 10;
- периметр треугольника (то есть сумма длин сторон) равен 24.



Треугольник называется *прямоугольным*, если один из его углов является прямым (равен 90 градусам).

Прямоугольные треугольники обладают полезным свойством: если возвести в квадрат длины сторон, образующих прямой угол, то сумма этих квадратов окажется равной квадрату стороны, противоположной прямому углу. На рисунке стороны, образующие прямой угол, помечены буквами `a` и `b`; сторона, противоположная прямому углу, помечена буквой `c`. Эта сторона называется *гипотенузой*.

Первым делом построим все тройки, элементы которых меньше либо равны 10:

```
ghci> let triples = [(a,b,c) | c <- [1..10], b <- [1..10], a <- [1..10]]
```

Мы просто собираем вместе три списка, и наша производящая функция объединяет их в тройки. Если вы вызовете функцию `triples` в GHCi, то получите список из тысячи троек. Теперь добавим условие, позволяющее отфильтровать только те тройки, которые соответствуют длинам сторон *прямоугольных* треугольников. Мы также модифицируем эту функцию, приняв во внимание, что сторона `b` не больше гипотенузы, и сторона `a` не больше стороны `b`.

```
ghci> let rightTriangles = [ (a,b,c) | c <- [1..10], b <- [1..c], a <- [1..b],
a  2 + b  2 == c  2]
```

> **ПРИМЕЧАНИЕ.** *В консоли интерпретатора GHCi невозможно определять программные сущности в нескольких строках. Но в данной книге нам иногда приходится разбивать определения на несколько строк, чтобы код помещался на странице. В противном случае книга оказалась бы такой широкоформатной, что для неё вам пришлось бы купить гигантский книжный шкаф!*

Почти закончили. Теперь давайте модифицируем функцию, чтобы получить треугольники, периметр которых равен 24.

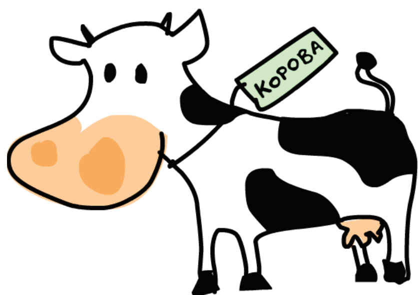
```
ghci> let rightTriangles' = [ (a,b,c) | c <- [1..10], b <- [1..c], a <- [1..b],
a  2 + b  2 == c  2, a+b+c == 24]
ghci> rightTriangles'
[(6,8,10)]
```

Вот и ответ! Это общий шаблон в функциональном программировании. Вы берёте начальный набор решений и затем применяете преобразования и фильтруете их, пока не получите результат.

2

ТИПЫ И КЛАССЫ ТИПОВ

Поверь в типы



Мы уже говорили о том, что Haskell является статически типизированным языком. Тип каждого выражения известен во время компиляции – это залог безопасного кода. Если вы напишете программу, которая попытается поделить булевский тип на число, то она даже не скомпилируется.

И хорошо, потому что уж лучше ловить такие ошибки на этапе компиляции, чем наблюдать, как ваша программа аварийно закрывается во время работы! Всему в языке Haskell назначен свой тип, так что компилятор может сделать довольно много выводов о программе перед её компиляцией.

В отличие от языков Java или Pascal, у Haskell есть механизм вывода типов. Если мы напишем число, то нет необходимости указывать, что это число. Язык Haskell может вывести это сам, так что нам не приходится явно обозначать типы функций и выражений.

Мы изучили некоторые основы языка, лишь вскользь упомянув о типах. Тем не менее понимание системы типов – очень важная часть обучения языку Haskell.

Тип – это нечто вроде ярлыка, который есть у каждого выражения. Он говорит нам, к какой категории относится данное выражение. Выражение `True` – булево, `"привет"` – это строка, и т. д.

Явное определение типов

А сейчас воспользуемся интерпретатором GHCi для определения типов нескольких выражений. Мы сделаем это с помощью команды `:t`, которая, если за ней следует любое правильное выражение, выдаст нам тип последнего. Итак…

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t "ПРИВЕТ!"
"ПРИВЕТ!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
ghci> :t 4 == 5
4 == 5 :: Bool
```

Мы видим, что `:t` печатает выражения, за которыми следуют `::` и их тип. Символы `::` означают: «имеет тип». У явно указанных типов первый символ всегда в верхнем регистре. Символ `'a'`, как вы заметили, имеет тип `Char`. Несложно сообразить, что это сокращение от «character» – символ. Константа `True` имеет тип `Bool`. Выглядит логично… Идём дальше.

Исследуя тип `"ПРИВЕТ!"`, получим `[Char]`. Квадратные скобки указывают на список – следовательно, перед нами «список символов». В отличие от списков, каждый кортеж любой длины имеет свой тип. Так выражение `(True, 'a')` имеет тип `(Bool, Char)`, тогда как выражение `('a','b','c')` будет иметь тип `(Char, Char, Char)`. Выражение `4==5` всегда вернёт `False`, поэтому его тип – `Bool`.

У функций тоже есть типы. Когда мы пишем свои собственные функции, то можем указывать их тип явно. Обычно это считается нормой, исключая случаи написания очень коротких функций. Здесь и далее мы будем декларировать типы для всех создаваемых нами функций.

Помните генератор списка, который мы использовали ранее: он фильтровал строку так, что оставались только прописные буквы? Вот как это выглядит с объявлением типа:

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['А'..'Я']]
```

Функция `removeNonUppercase` имеет тип `[Char] -> [Char]`. Эта запись означает, что функция принимает одну строку в качестве параметра и возвращает другую в качестве результата.

А как записать тип функции, которая принимает несколько параметров? Вот, например, простая функция, принимающая три целых числа и складывающая их:

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Параметры разделены символами –>, и здесь нет никакого различия между параметрами и типом возвращаемого значения. Возвращаемый тип – это последний элемент в объявлении, а параметры – первые три.

Позже мы увидим, почему они просто разделяются с помощью символов `->`, вместо того чтобы тип возвращаемого значения както специально отделялся от типов параметров (например, `Int, Int, Int -> Int` или что-то в этом духе).

Если вы хотите объявить тип вашей функции, но не уверены, каким он должен быть, то всегда можно написать функцию без него, а затем проверить тип с помощью `:t`. Функции – тоже выражения, так что `:t` будет работать с ними без проблем.

Обычные типы в языке Haskell

А вот обзор некоторых часто используемых типов.

- Тип `Int` обозначает целое число. Число 7 может быть типа `Int`, но 7.2 – нет. Тип `Int` ограничен: у него есть минимальное

и максимальное значения. Обычно на 32-битных машинах максимально возможное значение типа `Int` – это 2 147 483 647, а минимально возможное – соответственно, –2 147 483 648.

ПРИМЕЧАНИЕ. *Мы используем компилятор GHC, в котором множество возможных значений типа `Int` определено размером машинного слова на используемом компьютере. Так что если у вас 64-битный процессор, вполне вероятно, что наименьшим значением типа `Int` будет -2^{63}, а наибольшим $2^{63}-1$.*

- Тип `Integer` обозначает... э-э-э... тоже целое число. Основная разница в том, что он не имеет ограничения, поэтому может представлять большие числа. Я имею в виду – *очень* большие. Между тем тип `Int` более эффективен. В качестве примера сохраните следующую функцию в файл:

```
factorial :: Integer -> Integer
factorial n = product [1..n]
```

Затем загрузите этот файл в GHCi с помощью команды `:l` и проверьте её:

```
ghci> factorial 50
30414093201713378043612608166064768844377641568960512000000000000
```

- Тип `Float` – это действительное число с плавающей точкой одинарной точности. Добавьте в файл ещё одну функцию:

```
circumference :: Float -> Float
circumference r = 2 * pi * r
```

Загрузите дополненный файл и запустите новую функцию:

```
ghci> circumference 4.0
25.132742
```

- Тип `Double` – это действительное число с плавающей точкой двойной точности. Двойная точность означает, что для представления чисел используется вдвое больше битов, поэтому дополнительная точность требует большего расхода памяти. Добавим в файл ещё одну функцию:

```
circumference' :: Double -> Double
```

```
circumference' r = 2 * pi * r
```

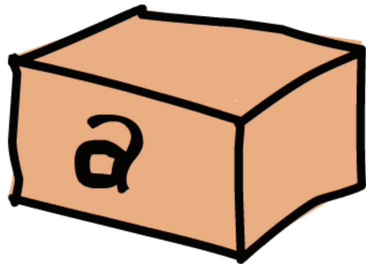
Загрузите дополненный файл и запустите новую функцию

```
ghci> circumference' 4.0
25.132741228718345
```

- Тип `Bool` – булевский. Он может принимать только два значения: `True` и `False`.
- Тип `Char` представляет символ Unicode. Его значения записываются в одинарных кавычках. Список символов является строкой.
- Кортежи – это типы, но тип кортежа зависит от его длины и от типа его компонентов. Так что теоретически количество типов кортежей бесконечно – а стало быть, перечислить их все в этой книге нет возможности. Заметьте, что пустой кортеж `()` – это тоже тип, который может содержать единственное значение: `()`.

Типовые переменные

Некоторые функции могут работать с данными разных типов. Например, функция `head` принимает список и возвращает его первый элемент. При этом неважно, что именно этот список содержит – числа, символы или вообще другие списки. Функция должна работать со списками, что бы они ни содержали.



Как вы думаете, каков тип функции `head`? Проверим, воспользовавшись командой `:t`.

```
ghci> :t head
head :: [a] -> a
```

Гм-м! Что такое `a`? Тип ли это? Мы уже отмечали, что все типы пишутся с большой буквы, так что это точно не может быть типом. В действительности это типовая переменная. Иначе говоря, `a` может быть любым типом.

Подобные элементы напоминают «дженерики» в других языках – но только в Haskell они гораздо более мощные, так как позволяют нам легко писать самые общие функции (конечно, если эти функции не используют какие-нибудь специальные свойства конкретных типов).

Функции, в объявлении которых встречаются переменные типа, называются *полиморфными*. Объявление типа функции `head` выше означает, что она принимает список любого типа и возвращает один элемент того же типа.

ПРИМЕЧАНИЕ. *Несмотря на то что переменные типа могут иметь имена, состоящие более чем из одной буквы, мы обычно называем их a, b, c, d…*

Помните функцию `fst`? Она возвращает первый компонент в паре. Проверим её тип:

```
ghci> :t fst
fst :: (a, b) -> a
```

Можно заметить, что функция `fst` принимает в качестве параметра кортеж, который состоит из двух компонентов, и возвращает значение того же типа, что и первый компонент пары. Поэтому мы можем применить функцию `fst` к паре, которая содержит значения любых двух типов.

Заметьте, что хотя `a` и `b` – различные переменные типа, они вовсе не обязаны быть *разного* типа. Сигнатура функции `fst` лишь означает, что тип первого компонента и тип возвращаемого значения одинаковы.

Классы типов

Класс типов – интерфейс, определяющий некоторое поведение. Если тип является *экземпляром* класса типов, то он поддерживает и реализует поведение, описанное классом типов. Более точно можно сказать, что класс типов определяет набор функций, и если мы

решаем сделать тип экземпляром класса типов, то должны явно указать, что эти функции означают применительно к нашему типу.

Хорошим примером будет класс типов, определяющий равенство. Значения многих типов можно сравнивать на равенство с помощью оператора ==. Посмотрим на его сигнатуру:

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

Заметьте: оператор равенства == – это функция. Функциями также являются операторы +, *, -, / и почти все остальные операторы. Если имя функции содержит только специальные символы, по умолчанию подразумевается, что это инфиксная функция. Если мы захотим проверить её тип, передать её другой функции или вызвать как префиксную функцию, мы должны поместить её в круглые скобки.

Интересно... мы видим здесь что-то новое, а именно символ =>. Всё, что находится перед символом =>, называется *ограничением класса*. Мы можем прочитать предыдущее объявление типа следующим образом: «функция сравнения на равенство принимает два значения одинакового типа и возвращает значение типа `Bool`. Тип этих двух значений должен быть экземпляром класса `Eq`» (это и есть ограничение класса).

Класс типа `Eq` предоставляет интерфейс для проверки на равенство. Каждый тип, для значений которого операция проверки на равенство имеет смысл, должен быть экземпляром класса `Eq`. Все стандартные типы языка Haskell (кроме типов для ввода-вывода и функций) являются экземплярами `Eq`.

> **ПРИМЕЧАНИЕ.** *Важно отметить, что классы типов в языке Haskell не являются тем же самым, что и классы в объектно-ориентированных языках программирования.*

У функции `elem` тип `(Eq a) => a -> [a] -> Bool`, потому что она применяет оператор == к элементам списка, чтобы проверить, есть ли в этом списке значение, которое мы ищем.

Далее приводятся описания нескольких базовых классов типов.

Класс Eq

Класс `Eq` используется для типов, которые поддерживают проверку равенства. Типы, являющиеся его экземплярами, должны реализовывать функции == и /=. Так что если у нас есть ограничение

класса `Eq` для переменной типа в функции, то она может использовать `==` или `/=` внутри своего определения. Все типы, которые мы упоминали выше, за исключением функций, входят в класс `Eq`, и, следовательно, могут быть проверены на равенство.

```
ghci> 5 == 5
True
ghci> 5 /= 5
False
ghci> 'a' == 'a'
True
ghci> "Хо Хо" == "Хо Хо"
True
ghci> 3.432 == 3.432
True
```

Класс Ord

Класс `Ord` предназначен для типов, которые поддерживают отношение порядка.

```
ghci> :t (>)
(>) :: (Ord a) => a -> a -> Bool
```

Все типы, упоминавшиеся ранее, за исключением функций, имеют экземпляры класса `Ord`. Класс `Ord` содержит все стандартные функции сравнения, такие как `>`, `<`, `>=` и `<=`. Функция `compare` принимает два значения одного и того же типа, являющегося экземпляром класса `Ord`, и возвращает значение типа `Ordering`. Тип `Ordering` может принимать значения `GT`, `LT` или `EQ`, означая, соответственно, «больше чем», «меньше чем» и «равно».

```
ghci> "Абракадабра" < "Зебра"
True
ghci> "Абракадабра" `compare` "Зебра"
LT
ghci> 5 >= 2
True
ghci> 5 `compare` 3
GT
```

Класс Show

Значения, типы которых являются экземплярами класса типов Show, могут быть представлены как строки. Все рассматривавшиеся до сих пор типы (кроме функций) являются экземплярами Show. Наиболее часто используемая функция в классе типов Show – это, собственно, функция show. Она берёт значение, для типа которого определён экземпляр класса Show, и представляет его в виде строки.

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"
```

Класс Read

Класс Read – это нечто противоположное классу типов Show. Функция read принимает строку и возвращает значение, тип которого является экземпляром класса Read.

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

Отлично. Но что случится, если попробовать вызвать read "4"?

```
ghci> read "4"
<interactive>:1:0:
    Ambiguous type variable `a' in the constraint:
      `Read a' arising from a use of `read' at <interactive>:1:0-7
    Probable fix: add a type signature that fixes these type variable(s)
```

Интерпретатор GHCi пытается нам сказать, что он не знает, что именно мы хотим получить в результате. Заметьте: во время предыдущих вызовов функции read мы что-то делали с результатом

функции. Таким образом, интерпретатор GHCi мог вычислить, какой тип ответа из функции read мы хотим получить.

Когда мы использовали результат как булево выражение, GHCi «понимал», что надо вернуть значение типа Bool. А в данном случае он знает, что нам нужен некий тип, входящий в класс Read, но не знает, какой именно. Давайте посмотрим на сигнатуру функции read.

```
ghci> :t read
read :: (Read a) => String -> a
```

ПРИМЕЧАНИЕ. *Идентификатор String – альтернативное наименование типа [Char]. Идентификаторы String и [Char] могут быть использованы взаимозаменяемо, но далее будет использоваться только String, поскольку это удобнее и писать, и читать.*

Видите? Функция возвращает тип, имеющий экземпляр класса Read, но если мы не воспользуемся им позже, то у компилятора не будет способа определить, какой именно это тип. Вот почему используются явные аннотации типа. Аннотации типа – способ явно указать, какого типа должно быть выражение. Делается это с помощью добавления символов :: в конец выражения и указания типа. Смотрите:

```
ghci> read "5" :: Int
5
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
ghci> read "(3, 'a')" :: (Int, Char)
(3, 'a')
```

Для большинства выражений компилятор может вывести тип самостоятельно. Но иногда он не знает, вернуть ли значение типа Int или Float для выражения вроде read "5". Чтобы узнать, какой у него тип, язык Haskell должен был бы фактически вычислить read "5".

Но так как Haskell – статически типизированный язык, он должен знать все типы до того, как скомпилируется код (или, в случае

GHCi, вычислится). Так что мы должны сказать языку: «Эй, это выражение должно иметь вот такой тип, если ты сам случайно не понял!»

Обычно компилятору достаточно минимума информации, чтобы определить, значение какого именно типа должна вернуть функция `read`. Скажем, если результат функции `read` помещается в список, то Haskell использует тип списка, полученный благодаря наличию других элементов списка:

```
ghci> [read "True" , False, True, False]
[True, False, True, False]
```

Так как `read "True"` используется как элемент списка булевых значений, Haskell самостоятельно определяет, что тип `read "True"` должен быть `Bool`.

Класс Enum

Экземплярами класса `Enum` являются последовательно упорядоченные типы; их значения можно перенумеровать. Основное преимущество класса типов `Enum` в том, что мы можем использовать его типы в интервалах списков. Кроме того, у них есть предыдущие и последующие элементы, которые можно получить с помощью функций `succ` и `pred`. Типы, входящие в этот класс: `()`, `Bool`, `Char`, `Ordering`, `Int`, `Integer`, `Float` и `Double`.

```
ghci> ['a'..'e']
"abcde"
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> [3 .. 5]
[3,4,5]
ghci>succ 'B'
'C'
```

Класс Bounded

Экземпляры класса типов `Bounded` имеют верхнюю и нижнюю границу.

```
ghci> minBound :: Int
-2147483648
```

```
ghci> maxBound :: Char
'\1114111'
ghci> maxBound :: Bool
True
ghci> minBound :: Bool
False
```

Функции `minBound` и `maxBound` интересны тем, что имеют тип `(Bounded a) => a`. В этом смысле они являются полиморфными константами.

Все кортежи также являются частью класса `Bounded`, если их компоненты принадлежат классу `Bounded`.

```
ghci> maxBound :: (Bool, Int, Char)
(True,2147483647,'\1114111')
```

Класс Num

Класс `Num` – это класс типов для чисел. Его экземпляры могут вести себя как числа. Давайте проверим тип некоторого числа:

```
ghci> :t 20
20 :: (Num t) => t
```

Похоже, что все числа также являются полиморфными константами. Они могут вести себя как любой тип, являющийся экземпляром класса `Num` (`Int`, `Integer`, `Float` или `Double`).

```
ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
20.0
```

Если проверить тип оператора `*`, можно увидеть, что он принимает любые числа.

```
ghci> :t (*)
(*) :: (Num a) => a -> a -> a
```

Он принимает два числа одинакового типа и возвращает число этого же типа. Именно поэтому `(5 :: Int) * (6 :: Integer)` приведёт к ошибке, а `5 * (6 :: Integer)` будет работать нормально и вернёт значение типа `Integer` потому, что 5 может вести себя и как `Integer`, и как `Int`.

Чтобы присоединиться к классу `Num`, тип должен «подружиться» с классами `Show` и `Eq`.

Класс Floating

Класс `Floating` включает в себя только числа с плавающей точкой, то есть типы `Float` и `Double`.

Функции, которые принимают и возвращают значения, являющиеся экземплярами класса `Floating`, требуют, чтобы эти значения могли быть представлены в виде числа с плавающей точкой для выполнения осмысленных вычислений. Некоторые примеры: функции `sin`, `cos` и `sqrt`.

Класс Integral

Класс `Integral` – тоже числовой класс типов. Если класс `Num` включает в себя все типы, в том числе действительные и целые числа, то в класс `Integral` входят только целые числа. Для типов `Int` и `Integer` определены экземпляры данного класса.

Очень полезной функцией для работы с числами является `fromIntegral`. Вот её объявление типа:

```
fromIntegral :: (Num b, Integral a) => a -> b
```

Из этой сигнатуры мы видим, что функция принимает целое число `(Integral)` и превращает его как более общее число `(Num)`.

ПРИМЕЧАНИЕ. *Необходимо отметить, что функция `fromIntegral` имеет несколько ограничений классов в своей сигнатуре. Такое вполне допустимо – несколько ограничений разделяются запятыми и заключаются в круглые скобки.*

Это окажется полезно, когда потребуется, чтобы целые числа и числа с плавающей точкой могли «сработаться» вместе. Например, функция вычисления длины `length` имеет объявление `length :: [a] -> Int`, вместо того чтобы использовать более общий тип `(Num b) => length :: [a] -> b`. (Наверное, так сложилось

исторически – хотя, по-моему, какова бы ни была причина, это довольно глупо.) В любом случае, если мы попробуем вычислить длину списка и добавить к ней `3.2`, то получим ошибку, потому что мы попытались сложить значения типа `Int` и число с плавающей точкой. В этом случае можно использовать функцию `fromIntegral`:

```
ghci> fromIntegral (length [1,2,3,4]) + 3.2
7.2
```

Несколько заключительных слов о классах типов

Поскольку класс типа определяет абстрактный интерфейс, один и тот же тип данных может иметь экземпляры для различных классов, а для одного и того же класса могут быть определены экземпляры различных типов. Например, тип `Char` имеет экземпляры для многих классов, два из которых – `Eq` и `Ord`, поскольку мы можем сравнивать символы на равенство и располагать их в алфавитном порядке.

Иногда для типа данных должен быть определён экземпляр некоторого класса для того, чтобы имелась возможность определить для него экземпляр другого класса. Например, для определения экземпляра класса `Ord` необходимо предварительно иметь экземпляр класса `Eq`. Другими словами, наличие экземпляра класса `Eq` является *предварительным* (*необходимым*) *условием* для определения экземпляра класса `Ord`. Если поразмыслить, это вполне логично: раз уж допускается расположение неких значений в определённом порядке, то должна быть предусмотрена и возможность проверить их на равенство.

3

СИНТАКСИС ФУНКЦИЙ

Сопоставление с образцом

В этой главе будет рассказано о некоторых весьма полезных синтаксических конструкциях языка Haskell, и начнём мы с сопоставления с образцом. Идея заключается в указании определённых шаблонов – *образцов*, которым должны соответствовать некоторые данные. Во время выполнения программы данные проверяются на соответствие образцу (сопоставляются). Если они подходят под образец, то будут разобраны в соответствии с ним.

Когда вы определяете функцию, её определение можно разбить на несколько частей (клозов), по одной части для каждого образца. Это позволяет создать очень стройный код, простой и легко читаемый. Вы можете задавать образцы для любого типа данных – чисел, символов, списков, кортежей и т. д. Давайте создадим простую функцию, которая проверяет, является ли её параметр числом семь.

```
lucky :: Int -> String
lucky 7 = "СЧАСТЛИВОЕ ЧИСЛО 7!"
lucky x = "Прости, друг, повезёт в другой раз!"
```

Когда вы вызываете функцию lucky, производится проверка параметра на совпадение с заданными образцами в том порядке, в каком они были заданы. Когда проверка даст положительный результат, используется соответствующее тело функции. Единственный случай, когда число, переданное функции, удовлетворяет первому образцу, – когда оно равно семи. В противном случае проводится проверка на совпадение со следующим образцом. Следующий образец может быть успешно сопоставлен с любым числом; также он привязывает переданное число к переменной x.

Если в образце вместо реального значения (например, 7) пишут идентификатор, начинающийся со строчной буквы (например, x, y или myNumber), то этот образец будет сопоставлен любому переданному значению. Обратиться к сопоставленному значению в теле функции можно будет посредством введённого идентификатора.

Эта функция может быть реализована с использованием ключевого слова if. Ну а если нам потребуется написать функцию, которая называет цифры от 1 до 5 и выводит "Это число не в пределах от 1 до 5" для других чисел? Без сопоставления с образцом нам бы пришлось создать очень запутанное дерево условных выражений if – then – else. А вот что получится, если использовать сопоставление:

```
sayMe :: Int -> String
sayMe 1 = "Один!"
sayMe 2 = "Два!"
sayMe 3 = "Три!"
sayMe 4 = "Четыре!"
sayMe 5 = "Пять!"
sayMe x = "Это число не в пределах от 1 до 5"
```

Заметьте, что если бы мы переместили последнюю строку определения функции (образец в которой соответствует любому вводу) вверх, то функция всегда выводила бы "Это число не в пределах от 1 до 5", потому что невозможно было бы пройти дальше и провести проверку на совпадение с другими образцами.

Помните реализованную нами функцию факториала? Мы определили факториал числа n как произведение чисел [1..n]. Мы можем определить данную функцию рекурсивно, точно так же, как факториал определяется в математике. Начнём с того, что объявим факториал нуля равным единице.

Затем определим факториал любого положительного числа как данное число, умноженное на факториал предыдущего числа. Вот как это будет выглядеть в терминах языка Haskell.

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Мы в первый раз задали функцию рекурсивно. Рекурсия очень важна в языке Haskell, и подробнее она будет рассмотрена позже.

Сопоставление с образцом может завершиться неудачей, если мы зададим функцию следующим образом:

```
charName :: Char -> String
charName 'а' = "Артём"
charName 'б' = "Борис"
charName 'в' = "Виктор"
```

а затем попытаемся вызвать её с параметром, которого не ожидали. Произойдёт следующее:

```
ghci> charName 'а'
"Артём"
ghci> charName 'в'
"Виктор"
ghci> charName 'м'
"*** Exception: Non-exhaustive patterns in function charName
```

Это жалоба на то, что наши образцы не покрывают всех возможных случаев (недоопределены) – и, воистину, так оно и есть! Когда мы определяем функцию, мы должны всегда включать образец, который можно сопоставить с любым входным значением, для того чтобы наша программа не закрывалась с сообщением об ошибке, если функция получит какие-то непредвиденные входные данные.

Сопоставление с парами

Сопоставление с образцом может быть использовано и для кортежей. Что если мы хотим создать функцию, которая принимает два двумерных вектора (представленных в форме пары) и складывает их? Чтобы сложить два вектора, нужно сложить их соответствующие координаты. Вот как мы написали бы такую функцию, если б не знали о сопоставлении с образцом:

```
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors a b = (fst a + fst b, snd a + snd b)
```

Это, конечно, сработает, но есть способ лучше. Давайте исправим функцию, чтобы она использовала сопоставление с образцом:

```
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

Так гораздо лучше. Теперь ясно, что параметры функции являются кортежами; к тому же компонентам кортежа сразу даны имена – это повышает читабельность. Заметьте, что мы сразу написали образец, соответствующий любым значениям. Тип функции `addVectors` в обоих случаях совпадает, так что мы гарантированно получим на входе две пары:

```
ghci> :t addVectors
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
```

Функции `fst` и `snd` извлекают компоненты пары. Но как быть с тройками? Увы, стандартных функций для этой цели не существует, однако мы можем создать свои:

```
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

Символ _ имеет то же значение, что и в генераторах списков. Он означает, что нам не интересно значение на этом месте, так что мы просто пишем _.

Сопоставление со списками и генераторы списков

В генераторах списков тоже можно использовать сопоставление с образцом, например:

```
ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
ghci> [a+b | (a,b) <- xs]
[4,7,6,8,11,4]
```

Если сопоставление с образцом закончится неудачей для одного элемента списка, просто произойдёт переход к следующему элементу.

Списки сами по себе (то есть заданные прямо в тексте образца списковые литералы) могут быть использованы при сопоставлении с образцом. Вы можете проводить сравнение с пустым списком или с любым образцом, который включает оператор `:` и пустой список. Так как выражение `[1,2,3]` – это просто упрощённая запись выражения `1:2:3:[]`, можно использовать `[1,2,3]` как образец.

Образец вида `(x:xs)` связывает «голову» списка с `x`, а оставшуюся часть – с `xs`, даже если в списке всего один элемент; в этом случае `xs` – пустой список.

ПРИМЕЧАНИЕ. *Образец `(x:xs)` используется очень часто, особенно с рекурсивными функциями. Образцы, в определении которых присутствует `:`, могут быть использованы только для списков длиной не менее единицы.*

Если вы, скажем, хотите связать первые три элемента с переменными, а оставшиеся элементы списка – с другой переменной, то можете использовать что-то наподобие `(x:y:z:zs)`. Образец сработает только для списков, содержащих не менее трёх элементов.

Теперь, когда мы знаем, как использовать сопоставление с образцом для списков, давайте создадим собственную реализацию функции `head`:

```
head' :: [a] -> a
head' [] = error "Нельзя вызывать head на пустом списке, тупица!"
head' (x:_) = x
```

Проверим, работает ли это...

```
ghci> head' [4,5,6]
4
ghci> head' "Привет"
H'
```

Отлично! Заметьте, что если вы хотите выполнить привязку к нескольким переменным (даже если одна из них обозначена всего лишь символом _ и на самом деле ни с чем не связывается), вам необходимо заключить их в круглые скобки. Также обратите внимание на

использование функции `error`. Она принимает строковый параметр и генерирует ошибку времени исполнения, используя этот параметр для сообщения о причине ошибки.

Вызов функции `error` приводит к аварийному завершению программы, так что не стоит использовать её слишком часто. Но вызов функции `head` на пустом списке не имеет смысла.

Давайте напишем простую функцию, которая сообщает нам о нескольких первых элементах списка – в довольно неудобной, чересчур многословной форме.

```
tell :: (Show a) => [a] -> String
tell [] = "Список пуст"
tell (x:[]) = "В списке один элемент: " ++ show x
tell (x:y:[]) = "В списке два элемента: " ++ show x ++ " и " ++ show y
tell (x:y:_) = "Список длинный. Первые два элемента: " ++ show x
               ++ " и " ++ show y
```

Обратите внимание, что образцы `(x:[])` и `(x:y:[])` можно записать как `[x]` и `[x,y]`. Но мы не можем записать `(x:y:_)` с помощью квадратных скобок, потому что такая запись соответствует любому списку длиной два или более.

Вот несколько примеров использования этой функции:

```
ghci>  tell [1]
"В списке один элемент: 1"
ghci> tell [True, False]
"В списке два элемента: True и False"
ghci> tell [1, 2, 3, 4]
"Список длинный. Первые два элемента: 1 и 2"
ghci> tell []
"Список пуст"
```

Функцию `tell` можно вызывать совершенно безопасно, потому что её параметр можно сопоставлять пустому списку, одноэлементному списку, списку с двумя и более элементами. Она умеет работать со списками любой длины и всегда знает, что нужно возвратить.

А что если определить функцию, которая умеет обрабатывать только списки с тремя элементами? Вот один такой пример:

```
badAdd :: (Num a) => [a] -> a
badAdd (x:y:z:[]) = x + y + z
```

А вот что случится, если подать ей не то, что она ждёт:

```
ghci> badAdd [100, 20]
*** Exception: Non-exhaustive patterns in function badAdd
```

Это не так уж и хорошо. Если подобное случится в скомпилированной программе, то она просто вылетит.

И последнее замечание относительно сопоставления с образцами для списков: в образцах нельзя использовать операцию `++` (напомню, что это объединение двух списков). К примеру, если вы попытаетесь написать в образце `(xs++ys)`, то Haskell не сможет определить, что должно попасть в `xs`, а что в `ys`. Хотя и могут показаться логичными сопоставления типа `(xs++[x,y,z])` или даже `(xs ++ [x])`, работать это не будет – такова природа списков[1].

Именованные образцы

Ещё одна конструкция называется *именованным образцом*. Это удобный способ разбить что-либо в соответствии с образцом и связать результат разбиения с переменными, но в то же время сохранить ссылку на исходные данные. Такую задачу можно выполнить, поместив некий идентификатор образца и символ `@` перед образцом, описывающим структуру данных. Например, так выглядит образец `xs@(x:y:ys)`.

Подобный образец работает так же, как `(x:y:ys)`, но вы легко можете получить исходный список по имени `xs`, вместо того чтобы раз за разом печатать `x:y:ys` в теле функции. Приведу пример:

```
firstLetter :: String -> String
firstLetter "" = "Упс, пустая строка!"
firstLetter all@(x:xs) = "Первая буква строки " ++ all ++ " это " ++ [x]
```

Загрузим эту функцию и посмотрим, как она работает:

```
ghci> firstLetter "Дракула"
"Первая буква строки Дракула это Д"
```

[1] На деле в образцах нельзя использовать операторы, представляющие собой двухместные функции (например, +, / и ++), поскольку при сопоставлении с образцами производится, по сути, обратная операция. Как сопоставить заданное число 5 с образцом `(x + y)`? Это можно сделать несколькими способами, то есть ситуация неопределённа. Между тем оператор `:` является конструктором данных (все бинарные операторы, начинающиеся с символа `:`, могут использоваться как конструкторы данных), поэтому для него можно произвести однозначное сопоставление. – *Прим. ред.*

Эй, стража!

В то время как образцы – это способ убедиться, что значение соответствует некоторой форме, и разобрать его на части, *сторожевые условия* (охрана, охранные выражения) – это способ проверить истинность некоторого свойства значения или нескольких значений, переданных функции. Тут можно провести аналогию с условным выражением if: оно работает схожим образом. Однако охранные выражения гораздо легче читать, если у вас имеется несколько условий; к тому же они отлично работают с образцами.

Вместо того чтобы объяснять их синтаксис, давайте просто напишем функцию с использованием охранных условий. Эта простая функция будет оценивать вас на основе ИМТ (индекса массы тела). Ваш ИМТ равен вашему весу, разделенному на квадрат вашего роста.

Если ваш ИМТ меньше 18.5, можно считать вас тощим. Если ИМТ составляет от 18.5 до 25, ваш вес в пределах нормы. От 25 до 30 – вы полненький; более 30 – тучный. Запишем эту функцию (мы не будем рассчитывать ИМТ, функция принимает его как параметр и ругнёт вас соответственно).

```
bmiTell :: Double -> String
bmiTell bmi
   | bmi <= 18.5 = "Слышь, эмо, ты дистрофик!"
   | bmi <= 25.0 = "По части веса ты в норме. Зато, небось, уродец!"
   | bmi <= 30.0 = "Ты толстый! Сбрось хоть немного веса!"
   | otherwise = "Мои поздравления, ты жирный боров!"
```

Охранные выражения обозначаются вертикальными чёрточками после имени и параметров функции. Обычно они печатаются с отступом вправо и начинаются с одной позиции. Охранное выражение должно иметь тип Bool. Если после вычисления условие имеет значение True, используется соответствующее тело функции. Если вычисленное условие ложно, проверка продолжается со следующего условия, и т. д.

Если мы вызовем эту функцию с параметром 24.3, она вначале проверит, не является ли это значение меньшим или равным 18.5. Так как охранное выражение на данном значении равно `False`, функция перейдёт к следующему варианту. Проверяется следующее условие, и так как 24.3 меньше, чем 25.0, будет возвращена вторая строка.

Это очень напоминает большие деревья условий `if – else` в императивных языках программирования – только такой способ записи значительно лучше и легче для чтения. Несмотря на то что большие деревья условий `if – else` обычно не рекомендуется использовать, иногда задача представлена в настолько разрозненном виде, что просто невозможно обойтись без них. Охранные выражения – прекрасная альтернатива для таких задач.

Во многих случаях последним охранным выражением является `otherwise` («иначе»). Значение `otherwise` определяется просто: `otherwise = True`; такое условие всегда истинно. Работа условий очень похожа на то, как работают образцы, но образцы проверяют входные данные, а охранные выражения могут производить любые проверки.

Если все охранные выражения ложны (и при этом мы не записали `otherwise` как последнее условие), вычисление продолжается со следующей строки определения функции. Вот почему сопоставление с образцом и охранные выражения так хорошо работают вместе. Если нет ни подходящих условий, ни клозов, будет сгенерирована ошибка времени исполнения.

Конечно же, мы можем использовать охранные выражения с функциями, которые имеют столько входных параметров, сколько нам нужно. Вместо того чтобы заставлять пользователя вычислять свой ИМТ перед вызовом функции, давайте модифицируем её так, чтобы она принимала рост и вес и вычисляла ИМТ:

```
bmiTell :: Double -> Double -> String
bmiTell weight height
   | weight / height ^ 2 <= 18.5 = "Слышь, эмо, ты дистрофик!"
   | weight / height ^ 2 <= 25.0 = "По части веса ты в норме.
                                    Зато, небось, уродец!"
   | weight / height ^ 2 <= 30.0 = "Ты толстый!
                                    Сбрось хоть немного веса!"
   | otherwise = "Мои поздравления, ты жирный боров!"
```

Ну-ка проверим, не толстый ли я...

```
ghci> bmiTell 85 1.90
"По части веса ты в норме. Зато, небось, уродец!"
```

Ура! По крайней мере, я не толстый! Правда, Haskell обозвал меня уродцем. Ну, это не в счёт.

> **ПРИМЕЧАНИЕ.** *Обратите внимание, что после имени функции и ее параметров нет знака равенства до первого охранного выражения. Многие новички ставят этот знак, что приводит к ошибке.*

Ещё один очень простой пример: давайте напишем нашу собственную функцию max. Если вы помните, она принимает два значения, которые можно сравнить, и возвращает большее из них.

```
max' :: (Ord a) => a -> a -> a
max' a b
   | a <= b = b
   | otherwise = a
```

Продолжим: напишем нашу собственную функцию сравнения, используя охранные выражения.

```
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
   | a == b = EQ
   | a <= b = LT
   | otherwise = GT
```

```
ghci> 3 `myCompare` 2
GT
```

> **ПРИМЕЧАНИЕ.** *Можно не только вызывать функции с помощью обратных апострофов, но и определять их так же. Иногда такую запись легче читать.*

Где же ты, where?!

Программисты обычно стараются избегать многократного вычисления одних и тех же значений. Гораздо проще один раз вычислить что-то, а потом сохранить его значение. В императивных языках программирования эта проблема решается сохранением результата вычислений в переменной. В данном разделе вы научитесь использовать ключевое слово where для сохранения результатов

промежуточных вычислений примерно с той же функциональностью.

В прошлом разделе мы определили вычислитель ИМТ и «ругалочку» на его основе таким образом:

```
bmiTell :: Double -> Double -> String
bmiTell weight height
   | weight / height ^ 2 <= 18.5 = "Слышь, эмо, ты дистрофик!"
   | weight / height ^ 2 <= 25.0 = "По части веса ты в норме.
                                    Зато, небось, уродец!"
   | weight / height ^ 2 <= 30.0 = "Ты толстый!
                                    Сбрось хоть немного веса!"
   | otherwise = "Мои поздравления, ты жирный боров!"
```

Заметили – мы повторили вычисление три раза? Операции копирования и вставки, да ещё повторенные трижды, – сущее наказание для программиста. Раз уж у нас вычисление повторяется три раза, было бы очень удобно, если бы мы могли вычислить его единожды, присвоить результату имя и использовать его, вместо того чтобы повторять вычисление. Можно переписать нашу функцию так:

```
bmiTell :: Double -> Double -> String
bmiTell weight height
   | bmi <= 18.5 = "Слышь, эмо, ты дистрофик!"
   | bmi <= 25.0 = "По части веса ты в норме.
                    Зато, небось, уродец!"
   | bmi <= 30.0 = "Ты толстый!
                    Сбрось хоть немного веса!"
   | otherwise = "Мои поздравления, ты жирный боров!"
  where bmi = weight / height ^ 2
```

Мы помещаем ключевое слово `where` после охранных выражений (обычно его печатают с тем же отступом, что и сами охранные выражения), а затем определяем несколько имён или функций. Эти имена видимы внутри объявления функции и позволяют нам не повторять код. Если вдруг нам вздумается вычислять ИМТ другим методом, мы должны исправить способ его вычисления только один раз.

Использование ключевого слова `where` улучшает читаемость, так как даёт имена понятиям и может сделать программы быстрее за счёт того, что переменные вроде `bmi` вычисляются лишь однаж-

ды. Попробуем зайти ещё дальше и представить нашу функцию так:

```
bmiTell :: Double -> Double -> String
bmiTell weight height
   | bmi <= skinny = "Слышь, эмо, ты дистрофик!"
   | bmi <= normal = "По части веса ты в норме.
                      Зато, небось, уродец!"
   | bmi <= fat = "Ты толстый!
                   Сбрось хоть немного веса!"
   | otherwise = "Мои поздравления, ты жирный боров!"
   where bmi = weight / height ^ 2
         skinny = 18.5
         normal = 25.0
         fat = 30.0
```

ПРИМЕЧАНИЕ. *Заметьте, что все идентификаторы расположены в одном столбце. Если не отформатировать исходный код подобным образом, язык Haskell не поймёт, что все они – часть одного блока определений.*

Область видимости декларации where

Переменные, которые мы определили в секции `where` нашей функции, видимы только ей самой, так что можно не беспокоиться о том, что мы засоряем пространство имён других функций. Если же нам нужны переменные, доступные в нескольких различных функциях, их следует определить глобально. Привязки в секции `where` не являются общими для различных образцов данной функции. Предположим, что мы хотим написать функцию, которая принимает на вход имя человека и, если это имя ей знакомо, вежливо его приветствует, а если нет – тоже приветствует, но несколько грубее. Первая попытка может выглядеть примерно так:

```
greet :: String -> String
greet "Хуан" = niceGreeting ++ " Хуан!"
greet "Фернандо" = niceGreeting ++ " Фернандо!"
greet name = badGreeting ++ " " ++ name
   where niceGreeting = "Привет! Так приятно тебя увидеть,"
         badGreeting = "О, чёрт, это ты,"
```

Однако эта функция работать не будет, так как имена, введённые в блоке `where`, видимы только в последнем варианте определе-

ния функции. Исправить положение может только глобальное определение функций niceGreeting и badGreeting, например:

```
badGreeting :: String
badGreeting = "О, чёрт, это ты,"

niceGreeting :: String
niceGreeting = "Привет! Так приятно тебя увидеть,"

greet :: String -> String
greet "Хуан" = niceGreeting ++ " Хуан!"
greet "Фернандо" = niceGreeting ++ " Фернандо!"
greet name = badGreeting ++ " " ++ name
```

Сопоставление с образцами в секции where

Можно использовать привязки в секции where и для сопоставления с образцом. Перепишем секцию where в нашей функции так:

```
  ...
  where bmi = weight / height   2
        (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

Давайте создадим ещё одну простую функцию, которая принимает два аргумента: имя и фамилию, и возвращает инициалы.

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
  where (f:_) = firstname
        (l:_) = lastname
```

Можно было бы выполнять сопоставление с образцом прямо в параметрах функции (это проще и понятнее), но мы хотим показать, что это допускается сделать и в определениях после ключевого слова where.

Функции в блоке where

Точно так же, как мы определяли константы в секции where, можно определять и функции. Придерживаясь нашей темы «здорового» программирования, создадим функцию, которая принимает список из пар «вес–рост» и возвращает список из ИМТ.

```
calcBmis :: [(Double, Double)] -> [Double]
calcBmis xs = [bmi w h | (w, h) <- xs]
  where bmi weight height = weight / height   2
```

Видите, что происходит? Причина, по которой нам пришлось представить `bmi` в виде функции в данном примере, заключается в том, что мы не можем просто вычислить один ИМТ для параметров, переданных в функцию. Нам необходимо пройтись по всему списку и для каждой пары вычислить ИМТ.

Пусть будет let

Определения, заданные с помощью ключевого слова `let`, очень похожи на определения в секциях `where`. Ключевое слово `where` – это синтаксическая конструкция, которая позволяет вам связывать выражения с переменными в конце функции; объявленные переменные видны во всём теле функции, включая сторожевые условия. Ключевое же слово `let` позволяет связывать выражения с именами в любом месте функции; конструкции `let` сами по себе являются выражениями, но их область видимости ограничена локальным контекстом. Таким образом, определение `let`, сделанное в охранном выражении, видно только в нём самом.

Как и любые другие конструкции языка Haskell, которые используются для привязывания имён к значениям, определения `let` могут быть использованы в сопоставлении с образцом. Посмотрим на них в действии! Вот как мы могли бы определить функцию, которая вычисляет площадь поверхности цилиндра по высоте и радиусу:

```
cylinder :: Double -> Double -> Double
cylinder r h =
   let sideArea = 2 * pi * r * h
```

```
      topArea = pi * r   2
  in  sideArea + 2 * topArea
```

Общее выражение выглядит так: `let <определения> in <выражение>`. Имена, которые вы определили в части `let`, видимы в выражении после ключевого слова `in`. Как видите, мы могли бы воспользоваться ключевым словом `where` для той же цели. Обратите внимание, что имена также выровнены по одной вертикальной позиции. Ну и какая разница между определениями в секциях `where` и `let`? Просто, похоже, в секции `let` сначала следуют определения, а затем выражение, а в секции `where` – наоборот.

На самом деле различие в том, что определения `let` сами по себе являются выражениями. Определения в секциях `where` – просто синтаксические конструкции. Если нечто является выражением, то у него есть значение. `"Фуу!"` – это выражение, и `3+5` – выражение, и даже `head [1,2,3]`. Это означает, что определение `let` можно использовать практически где угодно, например:

```
ghci> 4 * (let a = 9 in a + 1) + 2
42
```

Ключевое слово `let` подойдёт для определения локальных функций:

```
ghci> [let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]
```

Если нам надо привязать значения к нескольким переменным в одной строке, мы не можем записать их в столбик. Поэтому мы разделяем их точкой с запятой.

```
ghci> (let a = 10; b = 20 in a*b, let foo="Эй, "; bar = "там!" in foo ++ bar)
(200,"Эй, там!")
```

Как мы уже говорили ранее, определения в секции `let` могут использоваться при сопоставлении с образцом. Они очень полезны, к примеру, для того, чтобы быстро разобрать кортеж на элементы и привязать значения элементов к переменным, а также в других подобных случаях.

```
ghci> (let (a,b,c) = (1,2,3) in a+b+c) * 100
600
```

Если определения `let` настолько хороши, то почему бы только их всё время и не использовать? Ну, так как это всего лишь выражения, причём с локальной областью видимости, то их нельзя использовать в разных охранных выражениях. К тому же некоторые предпочитают, чтобы их переменные вычислялись после использования в теле функции, а не до того. Это позволяет сблизить тело функции с её именем и типом, что способствует большей читабельности.

Выражения let в генераторах списков

Давайте перепишем наш предыдущий пример, который обрабатывал списки пар вида (вес, рост), чтобы он использовал секцию `let` в выражении вместо того, чтобы определять вспомогательную функцию в секции `where`.

```
calcBmis :: [(Double, Double)] -> [Double]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h    2]
```

Мы поместили выражение `let` в генератор списка так, словно это предикат, но он не фильтрует список, а просто определяет имя. Имена, определённые в секции `let` внутри генератора списка, видны в функции вывода (часть до символа `|`) и для всех предикатов и секций, которые следуют после ключевого слова `let`. Так что мы можем написать функцию, которая выводит только толстяков:

```
calcBmis :: [(Double, Double)] -> [Double]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi > 25.0]
```

Использовать имя `bmi` в части `(w, h) <- xs` нельзя, потому что она расположена до ключевого слова `let`.

Выражения let в GHCi

Часть `in` также может быть пропущена при определении функций и констант напрямую в GHCi. В этом случае имена будут видимы во время одного сеанса работы GHCi.

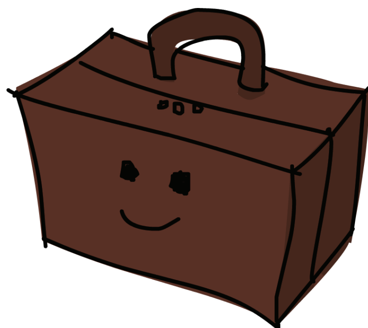
```
ghci> let zoot x y z = x * y + z
ghci> zoot 3 9 2
29
ghci> let boot x y z = x * y + z in boot 3 4 2
```

```
14
ghci> boot
<interactive>:1:0: Not in scope: `boot'
```

Поскольку в первой строке мы опустили часть `in`, GHCi знает, что в этой строке `zoot` не используется, поэтому запомнит его до конца сеанса. Однако во втором выражении `let` часть `in` присутствует, и определённая в нём функция `boot` тут же вызывается. Выражение `let`, в котором сохранена часть `in`, является выражением и представляет некоторое значение, так что GHCi именно это значение и печатает.

Выражения для выбора из вариантов

Во многих императивных языках (C, C++, Java, и т. д.) имеется оператор `case`, и если вам доводилось программировать на них, вы знаете, что это такое. Вы берёте переменную и выполняете некую часть кода для каждого значения этой переменной – ну и, возможно, используете финальное условие, которое срабатывает, если не отработали другие.



Язык Haskell позаимствовал эту концепцию и усовершенствовал её. Само имя «выражения для выбора» указывает на то, что они являются... э-э-э... *выражениями*, так же как `if – then – else` и `let`. Мы не только можем вычислять выражения, основываясь на возможных значениях переменной, но и производить сопоставление с образцом.

Итак, берём переменную, выполняем сопоставление с образцом, выполняем участок кода в зависимости от полученного значения... где-то мы это уже слышали!.. Ах да, сопоставление с образцом по параметрам при объявлении функции! На самом деле это всего лишь навсего облегчённая запись для выражений выбора. Эти два фрагмента кода делают одно и то же – они взаимозаменяемы:

```
head' :: [a] -> a
head' [] = error "Никаких head для пустых списков!"
head' (x:_) = x
```

```
head' :: [a] -> a
head' xs =
   case xs of
    [] -> error "Никаких head для пустых списков!"
    (x:_) -> x
```

Как видите, синтаксис для выражений отбора довольно прост:

```
case expression of
  pattern -> result
  pattern -> result
  ...
```

Выражения проверяются на соответствие образцам. Сопоставление с образцом работает как обычно: используется первый образец, который подошёл. Если были опробованы все образцы и ни один не подошёл, генерируется ошибка времени выполнения.

Сопоставление с образцом по параметрам функции может быть сделано только при объявлении функции; выражения отбора могут использоваться практически везде. Например:

```
describeList :: [a] -> String
describeList xs = "Список " ++
   case xs of
    [] -> "пуст."
    [x] -> "одноэлементный."
    xs -> "длинный."
```

Они удобны для сопоставления с каким-нибудь образцом в середине выражения. Поскольку сопоставление с образцом при объявлении функции – это всего лишь упрощённая запись выражений отбора, мы могли бы определить функцию таким образом:

```
describeList :: [a] -> String
describeList xs = "Список " ++ what xs
   where
     what [] = "пуст."
     what [x] = "одноэлементный."
     what xs = "длинный."
```

4

РЕКУРСИЯ

Привет, рекурсия!

В предыдущей главе мы кратко затронули рекурсию. Теперь мы изучим её более подробно, узнаем, почему она так важна для языка Haskell и как мы можем создавать лаконичные и элегантные решения, думая *рекурсивно.*

Если вы всё ещё не знаете, что такое рекурсия, прочтите это предложение ещё раз. Шучу!.. На самом деле *рекурсия* – это способ определять функции таким образом, что функция применяется в собственном определении. Стратегия решения при написании рекурсивно определяемых функций заключается в разбиении задачи на более мелкие подзадачи того же вида и в попытке их решения путём разбиения при необходимости на ещё более мелкие. Рано или поздно мы достигаем базовый случай (или базовые случаи) задачи, разбить который на подзадачи не удаётся и который требует написания явного (нерекурсивного) решения.

Многие понятия в математике даются рекурсивно. Например, последовательность чисел Фибоначчи. Мы определяем первые два числа Фибоначчи не рекурсивно. Допустим, $F(0) = 0$ и $F(1) = 1$; это означает, что нулевое и первое число из ряда Фибоначчи – это ноль и единица. Затем мы определим, что для любого натурального числа число Фибоначчи представляет собой сумму двух предыдущих чисел Фибоначчи. Таким образом, $F(n) = F(n - 1) + F(n - 2)$. Получается, что $F(3)$ – это $F(2) + F(1)$, что в свою очередь даёт $(F(1) + F(0)) + F(1)$. Так как мы достигли чисел Фибоначчи, заданных не рекурсивно, то можем точно сказать, что $F(3)$ равно двум.

Рекурсия исключительно важна для языка Haskell, потому что, в отличие от императивных языков, вы выполняете вычисления в Haskell, описывая некоторое понятие, а не указывая, как его получить. Вот почему в этом языке нет циклов типа `while` и `for` – вместо этого мы зачастую должны использовать рекурсию, чтобы описать, что представляет собой та или иная сущность.

Максимум удобства

Функция `maximum` принимает список упорядочиваемых элементов (то есть экземпляров класса `Ord`) и возвращает максимальный элемент. Подумайте, как бы вы реализовали эту функцию в императивном стиле. Вероятно, завели бы переменную для хранения текущего значения максимального элемента – и затем в цикле проверяли бы элементы списка. Если элемент больше, чем текущее максимальное значение, вы бы замещали его новым значением. То, что осталось в переменной после завершения цикла, – и есть максимальный элемент. Ух!.. Довольно много слов потребовалось, чтобы описать такой простой алгоритм!

Ну а теперь посмотрим, как можно сформулировать этот алгоритм рекурсивно. Для начала мы бы определили базовые случаи. В пустом списке невозможно найти максимальный элемент. Если список состоит из одного элемента, то максимум равен этому элементу. Затем мы бы сказали, что максимум списка из более чем двух элементов – это большее из двух чисел: первого элемента («головы») или максимального элемента оставшейся части списка («хвоста»). Теперь запишем это на языке Haskell.

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "максимум в пустом списке"
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)
```

Как вы видите, сопоставление с образцом отлично дополняет рекурсию! Возможность сопоставлять с образцом и разбивать сопоставляемое значение на компоненты облегчает запись подзадач в задаче поиска максимального элемента. Первый образец говорит, что если список пуст – это ошибка! В самом деле, какой максимум у пустого списка? Я не знаю. Второй образец также описывает базовый случай. Он говорит, что если в списке всего один элемент, надо его вернуть в качестве максимального.

В третьем образце происходит самое интересное. Мы используем сопоставление с образцом для того, чтобы разбить список на «голову» и «хвост». Это очень распространённый приём при работе со списками, так что привыкайте. Затем мы вызываем уже знакомую функцию `max`, которая принимает два параметра и возвращает больший из них. Если `x` больше наибольшего элемента `xs`, то вернётся `x`; в противном случае вернётся наибольший элемент `xs`. Но как функция `maximum'` найдёт наибольший элемент `xs`? Очень просто — вызвав себя рекурсивно.

Давайте возьмём конкретный пример и посмотрим, как всё это работает. Итак, у нас есть список `[2,5,1]`. Если мы вызовем функцию `maximum'` с этим значением, первые два образца не подойдут. Третий подойдёт – список разобьётся на `2` и `[5,1]`. Теперь мы заново вызываем функцию с параметром `[5,1]`. Снова подходит третий образец, список разбивается на `5` и `[1]`. Вызываем функцию для `[1]`. На сей раз подходит второй образец – возвращается `1`. Наконец-то! Отходим на один шаг назад, вычисляем максимум `5` и наибольшего

элемента [1] (он равен 1), получаем 5. Теперь мы знаем, что максимум [5,1] равен 5. Отступаем ещё на один шаг назад – там, где у нас было 2 и [5,1]. Находим максимум 2 и 5, получаем 5. Таким образом, наибольший элемент [2,5,1] равен 5.

Ещё немного рекурсивных функций

Теперь, когда мы знаем основы рекурсивного мышления, давайте напишем несколько функций, применяя рекурсию. Как и maximum, эти функции в Haskell уже есть, но мы собираемся создать свои собственные версии, чтобы, так сказать, прокачать рекурсивные группы мышц.

Функция replicate

Для начала реализуем функцию replicate. Функция replicate берёт целое число (типа Int) и некоторый элемент и возвращает список, который содержит несколько повторений заданного элемента. Например, replicate 3 5 вернёт список [5,5,5]. Давайте обдумаем базовые случаи. Сразу ясно, что возвращать, если число повторений равно нулю или вообще отрицательное – пустой список. Для отрицательных чисел функция вовсе не имеет смысла.

В общем случае список, состоящий из n повторений элемента x, – это список, имеющий «голову» x и «хвост», состоящий из (n-1)-кратного повторения x. Получаем следующий код:

```
replicate' :: Int -> a -> [a]
replicate' n x
  | n <= 0 = []
  | otherwise = x : replicate' (n-1) x
```

Мы использовали сторожевые условия вместо образцов потому, что мы проверяем булевы выражения.

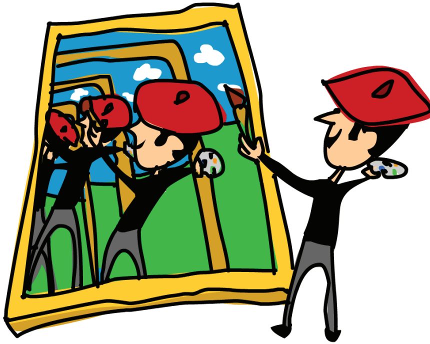
Функция take

Теперь реализуем функцию take. Эта функция берёт определённое количество первых элементов из заданного списка. Например, take 3 [5,4,3,2,1] вернёт список [5,4,3]. Если мы попытаемся получить ноль или менее элементов из списка, результатом будет пустой список. Если попытаться получить какую-либо часть пустого

списка, функция тоже возвратит пустой список. Заметили два базовых случая? Ну, давайте это запишем:

```
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

Заметьте, что в первом образце, который соответствует случаю, когда мы хотим взять нуль или меньше элементов, мы используем маску. Маска _ используется для сопоставления со списком, потому что сам список нас в данном случае не интересует. Также обратите внимание, что мы применяем охранное выражение, но без части otherwise. Это означает, что если значение n будет больше нуля, сравнение продолжится со следующего образца. Второй образец обрабатывает случай, когда мы пытаемся получить часть пустого списка, – возвращается пустой список. Третий образец разбивает список на «голову» и «хвост». Затем мы объявляем, что получить n элементов от списка – это то же самое, что взять «голову» списка и добавить (n-1) элемент из «хвоста».



Функция reverse

Функция reverse обращает список, выстраивая элементы в обратном порядке. И снова пустой список оказывается базовым случаем, потому что если обратить пустой список, получим тот же пустой список. Хорошо... А что насчёт всего остального? Ну, можно сказать, что если разбить список на «голову» и «хвост», то обращённый список – это обращённый «хвост» плюс «голова» списка в конце.

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

Готово!

Функция repeat

Функция `repeat` принимает на вход некоторый элемент и возвращает бесконечный список, содержащий этот элемент. Рекурсивное определение такой функции довольно просто – судите сами:

```
repeat' :: a -> [a]
repeat' x = x:repeat' x
```

Вызов `repeat 3` даст нам список, который начинается с тройки и содержит бесконечное количество троек в хвостовой части. Вызов будет вычислен как `3:repeat 3`, затем как `3:(3:repeat 3)`, `3:(3:(3:repeat 3))` и т. д. Вычисление `repeat 3` не закончится никогда, а вот `take 5 (repeat 3)` выдаст нам список из пяти троек. Это то же самое, что вызвать `replicate 5 3`.

Функция `repeat` наглядно показывает, что рекурсия может вообще не иметь базового случая, если она создаёт бесконечные списки – нам нужно только вовремя их где-нибудь обрезать.

Функция zip

Функция `zip` берёт два списка и стыкует их, образуя список пар (по аналогии с тем, как застёгивается замок-молния). Так, например, `zip [1,2,3] ['a','b']` вернёт список `[(1,'a'),(2,'b')]`. При этом более длинный список, как видите, обрезается до длины короткого. Ну а если мы состыкуем что-либо с пустым списком? Получим пустой список! Это базовый случай. Но так как функция принимает на вход два списка, то на самом деле это два базовых случая.

```
zip' :: [a] -> [b] -> [(a,b)]
zip' _ [] = []
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y):zip' xs ys
```

Первые два образца соответствуют базовым случаям: если первый или второй список пустые, возвращается пустой список. В третьем образце говорится, что склеивание двух списков эквивалентно созданию пары из их «голов» и присоединению этой пары к результату склеивания «хвостов».

Например, если мы вызовем `zip'` со списками `[1,2,3]` и `['a','b']`, то первым элементом результирующего списка станет пара `(1, 'a')`, и останется склеить списки `[2,3]` и `['b']`. После

ещё одного рекурсивного вызова функция попытается склеить [3] и [], что будет сопоставлено с первым образцом. Окончательным результатом теперь будет список (1,'a'):((2,'b'):[]), то есть, по сути, [(1,'a'),(2,'b')].

Функция elem

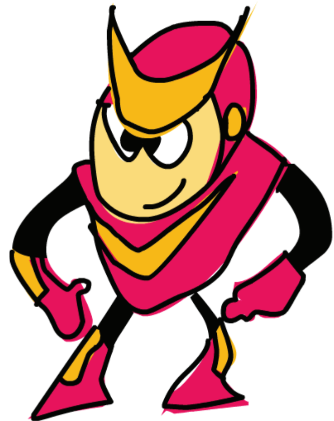
Давайте реализуем ещё одну функцию из стандартной библиотеки – elem. Она принимает элемент и список и проверяет, есть ли заданный элемент в этом списке. Как обычно, базовый случай – это пустой список. Мы знаем, что в пустом списке нет элементов, так что в нём определённо нет ничего, что мы могли бы искать.

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
    | a == x = True
    | otherwise = a `elem'` xs
```

Довольно просто и ожидаемо. Если «голова» не является искомым элементом, мы проверяем «хвост». Если мы достигли пустого списка, то результат – False.

Сортируем, быстро!..

Итак, у нас есть список элементов, которые могут быть отсортированы. Их тип – экземпляр класса Ord. А теперь требуется их отсортировать! Для этого предусмотрен очень классный алгоритм, называемый *быстрой сортировкой* (quicksort). Это довольно-таки хитроумный способ. В то время как его реализация на императивных языках занимает многим более 10 строк, на языке Haskell он намного короче и элегантнее. Настолько, что быстрая сортировка на Haskell стала притчей во языцех. Только ленивый не приводил пример



определения функции `quicksort`, чтобы наглядно продемонстрировать изящество языка. Давайте и мы напишем её, несмотря на то что подобный пример уже считается дурным тоном.

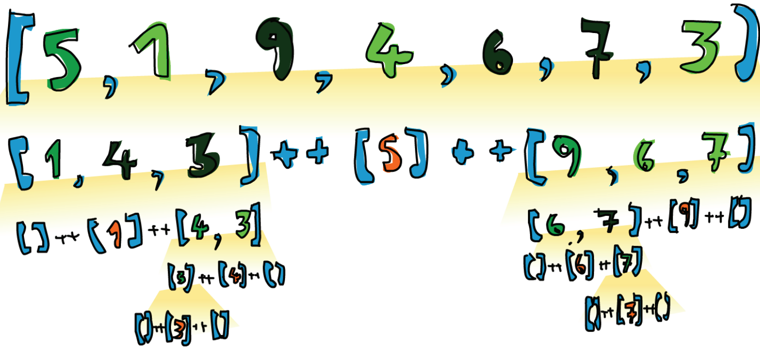
Алгоритм

Итак, сигнатура функции будет следующей:

```
quicksort :: (Ord a) => [a] -> [a]
```

Ничего удивительного тут нет. Базовый случай? Пустой список, как и следовало ожидать. Отсортированный пустой список – это пустой список. Затем следует основной алгоритм: отсортированный список – это список, в котором все элементы, меньшие либо равные «голове» списка, идут впереди (в отсортированном порядке), посередине следует «голова» списка, а потом – все элементы, большие «головы» списка (также отсортированные). Заметьте, в определении мы упомянули сортировку дважды, так что нам, возможно, придётся делать два рекурсивных вызова в теле функции. Также обратите внимание на то, что мы описали алгоритм, просто дав определение отсортированному списку. Мы не указывали явно: «делай это, затем делай то...» В этом красота функционального программирования! Как нам отфильтровать список, чтобы получить только те элементы, которые больше «головы» списка, и те, которые меньше? С помощью генераторов списков.

Если у нас, скажем, есть список `[5,1,9,4,6,7,3]` и мы хотим отсортировать его, этот алгоритм сначала возьмёт «голову», которая равна `5`, и затем поместит её в середину двух списков, где хранятся элементы меньшие и большие «головы» списка. То есть в нашем примере получается следующее: `[1,4,3] ++ [5] ++ [9,6,7]`. Мы знаем, что когда список будет отсортирован, число `5` будет находиться на четвёртой позиции, потому что есть три числа меньше и три числа больше 5. Теперь, если мы отсортируем списки `[1,4,3]` и `[9,6,7]`, то получится отсортированный список! Мы сортируем эти два списка той же самой функцией. Рано или поздно мы достигнем пустого списка, который уже отсортирован – в силу своей пустоты. Проиллюстрируем (цветной вариант рисунка приведён на форзаце книги):



Элемент, который расположен на своём месте и больше не будет перемещаться, выделен оранжевым цветом. Если вы просмотрите элементы слева направо, то обнаружите, что они отсортированы. Хотя мы решили сравнивать все элементы с «головами», можно использовать и другие элементы для сравнения. В алгоритме быстрой сортировки элемент, с которым производится сравнение, называется *опорным*. На нашей картинке такие отмечены зелёным цветом. Мы выбрали головной элемент в качестве опорного, потому что его легко получить при сопоставлении с образцом. Элементы, которые меньше опорного, обозначены светло-зелёным цветом; элементы, которые больше, – темно-зелёным. Желтоватый градиент демонстрирует применение быстрой сортировки.

Определение

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
   let smallerSorted = quicksort [a | a <- xs, a <= x]
       biggerSorted = quicksort [a | a <- xs, a > x]
   in smallerSorted ++ [x] ++ biggerSorted
```

Давайте немного «погоняем» функцию – так сказать, испытаем её в действии:

```
ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
[1,2,2,3,3,4,4,5,6,7,8,9,10]
```

```
ghci> quicksort "съешь ещё этих мягких французских булок, да выпей чаю"
"          ,aaaбвгдеееёзиииййкклмнопрсстууфхххцчшщъыьэюя"
```

Ура! Это именно то, чего я хотел!

Думаем рекурсивно

Мы уже много раз использовали рекурсию, и, как вы, возможно, заметили, тут есть определённый шаблон. Обычно вы определяете базовые случаи, а затем задаёте функцию, которая что-либо делает с рядом элементов, и функцию, применяемую к оставшимся элементам. Неважно, список ли это, дерево либо другая структура данных. Сумма – это первый элемент списка плюс сумма оставшейся его части. Произведение списка – это первый его элемент, умноженный на произведение оставшейся части. Длина списка – это единица плюс длина «хвоста» списка. И так далее, и тому подобное...

Само собой разумеется, у всех упомянутых функций есть базовые случаи. Обычно они представляют собой некоторые сценарии выполнения, при которых применение рекурсивного вызова не имеет смысла. Когда имеешь дело со списками, это, как правило, пустой список. Когда имеешь дело с деревьями, это в большинстве случаев узел, не имеющий потомков.

Похожим образом обстоит дело, если вы рекурсивно обрабатываете числа. Обычно мы работаем с неким числом, и функция применяется к тому же числу, но модифицированному некоторым образом. Ранее мы написали функцию для вычисления факториала – он равен произведению числа и факториала от того же числа, уменьшенного на единицу. Такой рекурсивный вызов не имеет смысла для нуля, потому что факториал не определён для отрицательных чисел. Часто базовым значением становится нейтральный элемент. Нейтральный элемент для умножения – 1, так как, умножая нечто на 1, вы получаете это самое нечто. Таким же образом при суммировании списка мы полагаем, что сумма пустого списка равна нулю,

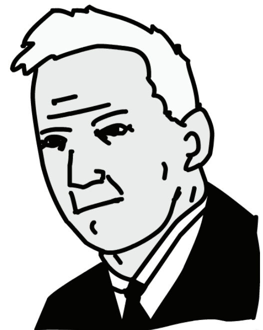
нуль – нейтральный элемент для сложения. В быстрой сортировке базовый случай – это пустой список; он же является нейтральным элементом, поскольку если присоединить пустой список к некоторому списку, мы снова получим исходный список.

Итак, пытаясь мыслить рекурсивным образом при решении задачи, попробуйте придумать, в какой ситуации рекурсивное решение не подойдёт, и понять, можно ли использовать этот вариант как базовый случай. Подумайте, что является нейтральным элементом, как вы будете разбивать параметры функции (например, списки обычно разбивают на «голову» и «хвост» путём сопоставления с образцом) и для какой части примените рекурсивный вызов.

5
ФУНКЦИИ ВЫСШЕГО ПОРЯДКА

Функции в языке Haskell могут принимать другие функции как параметры и возвращать функции в качестве результата. Если некая функция делает что-либо из вышеперечисленного, её называют *функцией высшего порядка* (ФВП). ФВП – не просто одна из значительных особенностей характера программирования, присущего языку Haskell, – она по большей части и определяет этот характер. Как выясняется, ФВП незаменимы, если вы хотите программировать исходя из того, *что* вы хотите получить, вместо того чтобы продумывать последовательность шагов, описывающую, *как* это получить. Это очень мощный способ решения задач и разработки программ.

Каррированные функции



Каждая функция в языке Haskell официально может иметь только один параметр. Но мы определяли и использовали функции, которые принимали несколько параметров. Как же такое может быть? Да, это хитрый трюк! Все функции, которые принимали несколько параметров, были *каррированы*. Функция называется каррированной, если она всегда принимает только один параметр вместо нескольких. Если потом её вызвать,

передав этот параметр, то результатом вызова будет новая функция, принимающая уже следующий параметр.

Легче всего объяснить на примере. Возьмём нашего старого друга – функцию `max`. Если помните, она принимает два параметра и возвращает максимальный из них. Если сделать вызов `max 4 5`, то вначале будет создана функция, которая принимает один параметр и возвращает `4` или поданный на вход параметр – смотря что больше. Затем значение `5` передаётся в эту новую функцию, и мы получаем желаемый результат. В итоге оказывается, что следующие два вызова эквивалентны:

```
ghci> max 4 5
5
ghci> (max 4) 5
5
```

Чтобы понять, как это работает, давайте посмотрим на тип функции `max`:

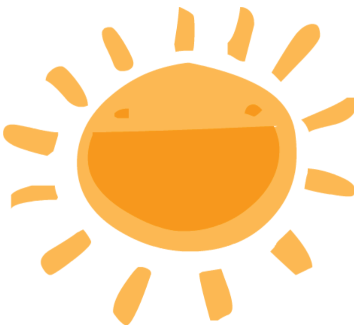
```
ghci> :t max
max :: (Ord a) => a -> a -> a
```

То же самое можно записать иначе:

```
max :: (Ord a) => a -> (a -> a)
```

Прочитать запись можно так: функция `max` принимает параметр типа `a` и возвращает (`->`) функцию, которая принимает параметр типа `a` и возвращает значение типа `a`. Вот почему возвращаемый функцией тип и параметры функции просто разделяются стрелками.

Ну и чем это выгодно для нас? Проще говоря, если мы вызываем функцию и передаём ей не все параметры, то в результате получаем новую функцию, а именно – результат частичного применения исходной функции. Новая функция принимает столько параметров, сколько мы не использовали при вызове оригинальной функции. Частичное применение (или, если угодно, вызов функции не со всеми параметрами) – это изящный способ создания



новых функций «на лету»: мы можем передать их другой функции или передать им ещё какие-нибудь параметры.

Посмотрим на эту простую функцию:

```
multThree :: Int -> Int -> Int -> Int
multThree x y z = x * y * z
```

Что происходит, если мы вызываем `multThree 3 5 9` или `((multThree 3) 5) 9`? Сначала значение `3` применяется к `multThree`, так как они разделены пробелом. Это создаёт функцию, которая принимает один параметр и возвращает новую функцию, умножающую на `3`. Затем значение `5` применяется к новой функции, что даёт функцию, которая примет параметр и умножит его уже на `15`. Значение `9` применяется к этой функции, и получается результат `135`. Вы можете думать о функциях как о маленьких фабриках, которые берут какие-то материалы и что-то производят. Пользуясь такой аналогией, мы даём фабрике `multThree` число `3`, и, вместо того чтобы выдать число, она возвращает нам фабрику немного поменьше. Эта новая фабрика получает число `5` и тоже выдаёт фабрику. Третья фабрика при получении числа `9` производит, наконец, результат — число `135`. Вспомним, что тип этой функции может быть записан так:

```
multThree :: Int -> (Int -> (Int -> Int))
```

Перед символом `->` пишется тип параметра функции; после записывается тип значения, которое функция вернёт. Таким образом, наша функция принимает параметр типа `Int` и возвращает функцию типа `Int -> (Int -> Int)`. Аналогичным образом эта новая функция принимает параметр типа `Int` и возвращает функцию типа `Int -> Int`. Наконец, функция принимает параметр типа `Int` и возвращает значение того же типа `Int`.

Рассмотрим пример создания новой функции путём вызова функции с недостаточным числом параметров:

```
ghci> let multTwoWithNine = multThree 9
ghci> multTwoWithNine 2 3
54
```

В этом примере выражение `multThree 9` возвращает функцию, принимающую два параметра. Мы называем эту функцию `multTwoWithNine`. Если при её вызове предоставить оба необходимых

параметра, то она перемножит их между собой, а затем умножит произведение на 9.

Вызывая функции не со всеми параметрами, мы создаём новые функции «на лету». Допустим, нужно создать функцию, которая принимает число и сравнивает его с константой 100. Можно сделать это так:

```
compareWithHundred :: Int -> Ordering
compareWithHundred x = compare 100 x
```

Если мы вызовем функцию с 99, она вернёт значение GT. Довольно просто. Обратите внимание, что параметр x находится с правой стороны в обеих частях определения. Теперь подумаем, что вернёт выражение compare 100. Этот вызов вернёт функцию, которая принимает параметр и сравнивает его с константой 100. Ага-а! Не этого ли мы хотели? Можно переписать функцию следующим образом:

```
compareWithHundred :: Int -> Ordering
compareWithHundred = compare 100
```

Объявление типа не изменилось, так как выражение compare 100 возвращает функцию. Функция compare имеет тип (Ord a) => a -> (a -> Ordering). Когда мы применим её к 100, то получим функцию, принимающую целое число и возвращающую значение типа Ordering.

Сечения

Инфиксные функции могут быть частично применены при помощи так называемых *сечений*. Для построения сечения инфиксной функции достаточно поместить её в круглые скобки и предоставить параметр только с одной стороны. Это создаст функцию, которая принимает один параметр и применяет его к стороне с пропущенным операндом. Вот донельзя простой пример:

```
divideByTen :: (Floating a) => a -> a
divideByTen = (/10)
```

Вызов, скажем, divideByTen 200 эквивалентен вызову 200 / 10, равно как и (/10) 200:

```
ghci> divideByTen 200
20.0
```

```
ghci> 200 / 10
20.0
ghci> (/10) 200
20.0
```

А вот функция, которая проверяет, находится ли переданный символ в верхнем регистре:

```
isUpperAlphanum :: Char -> Bool
isUpperAlphanum = (`elem` ['А'..'Я'])
```

Единственная особенность при использовании сечений – применение знака «минус». По определению сечений, `(-4)` – это функция, которая вычитает четыре из переданного числа. В то же время для удобства (–4) означает «минус четыре». Если вы хотите создать функцию, которая вычитает четыре из своего аргумента, выполняйте частичное применение таким образом: `(subtract 4)`.

Печать функций

До сих пор мы давали частично применённым функциям имена, после чего добавляли недостающие параметры, чтобы всё-таки посмотреть на результаты. Однако мы ни разу не попробовали напечатать сами функции. Попробуем? Что произойдёт, если мы попробуем выполнить `multThree 3 4` в GHCi вместо привязки к имени с помощью ключевого слова `let` либо передачи другой функции?

```
ghci> multThree 3 4
<interactive>:1:0:
   No instance for (Show (a -> a))
     arising from a use of `print' at <interactive>:1:0-12
   Possible fix: add an instance declaration for (Show (a -> a))
   In the expression: print it
   In a 'do' expression: print it
```

GHCi сообщает нам, что выражение порождает функцию типа `a -> a`, но он не знает, как вывести её на экран. Функции не имеют экземпляра класса `Show`, так что мы не можем получить точное строковое представление функций. Когда мы вводим, скажем, `1 + 1` в терминале GHCi, он сначала вычисляет результат (`2`), а затем вызывает функцию `show` для `2`, чтобы получить текстовое представление

этого числа. Текстовое представление 2 – это строка "2", которая и выводится на экран.

> **ПРИМЕЧАНИЕ.** *Удостоверьтесь в том, что вы поняли, как работает каррирование и частичное применение функций, поскольку эти понятия очень важны.*

Немного о высоких материях

Функции могут принимать функции в качестве параметров и возвращать функции в качестве значений. Чтобы проиллюстрировать это, мы собираемся создать функцию, которая принимает функцию, а затем дважды применяет её к чему-нибудь!

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

Прежде всего, обратите внимание на объявление типа. Раньше мы не нуждались в скобках, потому что символ -> обладает правой ассоциативностью. Однако здесь скобки обязательны. Они показывают, что первый параметр – это функция, которая принимает параметр некоторого типа и возвращает результат того же типа. Второй параметр имеет тот же тип, что и аргумент функции – как и возвращаемый результат. Мы можем прочитать данное объявление в каррированном стиле, но, чтобы избежать головной боли, просто скажем, что функция принимает два параметра и возвращает результат. Первый параметр – это функция (она имеет тип `a -> a`), второй параметр имеет тот же тип `a`. Заметьте, что совершенно неважно, какому типу будет соответствовать типовая переменная `a` – `Int`, `String` или вообще чему угодно – но при этом все значения должны быть одного типа.

> **ПРИМЕЧАНИЕ.** *Отныне мы будем говорить, что функция принимает несколько параметров, вопреки тому что в действительности каждая функция принимает только один параметр и возвращает*

частично применённую функцию. Для простоты будем говорить, что `a -> a -> a` *принимает два параметра, хоть мы и знаем, что происходит «за кулисами».*

Тело функции `applyTwice` достаточно простое. Мы используем параметр `f` как функцию, применяя её к параметру `x` (для этого разделяем их пробелом), после чего передаём результат снова в функцию `f`. Давайте поэкспериментируем с функцией:

```
ghci> applyTwice (+3) 10
16
ghci> applyTwice (++ " ХА-ХА") "ЭЙ"
"ЭЙ ХА-ХА ХА-ХА"
ghci> applyTwice ("ХА-ХА " ++) "ЭЙ"
"ХА-ХА ХА-ХА ЭЙ"
ghci> applyTwice (multThree 2 2) 9
144
ghci> applyTwice (3:) [1]
[3,3,1]
```

Красота и полезность частичного применения очевидны. Если наша функция требует передать ей функцию одного аргумента, мы можем частично применить функцию-параметр таким образом, чтобы оставался неопределённым всего один параметр, и затем передать её нашей функции. Например, функция + принимает два параметра; с помощью сечений мы можем частично применить её так, чтобы остался только один.

Реализация функции zipWith

Теперь попробуем применить ФВП для реализации очень полезной функции из стандартной библиотеки. Она называется `zipWith`. Эта функция принимает функцию и два списка, а затем соединяет списки, применяя переданную функцию для соответствующих элементов. Вот как мы её реализуем:

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

Посмотрите на объявление типа. Первый параметр – это функция, которая принимает два значения и возвращает одно. Пара-

метры этой функции не обязательно должны быть одинакового типа, но могут. Второй и третий параметры – списки. Результат тоже является списком. Первым идёт список элементов типа a, потому что функция сцепления принимает значение типа a в качестве первого параметра. Второй должен быть списком из элементов типа b, потому что второй параметр у связывающей функции имеет тип b. Результат – список элементов типа c. Если объявление функции говорит, что она принимает функцию типа a -> b -> c как параметр, это означает, что она также примет и функцию a -> a -> a, но не наоборот.

ПРИМЕЧАНИЕ. *Запомните: когда вы создаёте функции, особенно высших порядков, и не уверены, каким должен быть тип, вы можете попробовать опустить объявление типа, а затем проверить, какой тип выведет язык Haskell, используя команду :t в GHCi.*

Устройство данной функции очень похоже на обычную функцию zip. Базовые случаи одинаковы. Единственный дополнительный аргумент – соединяющая функция, но он не влияет на базовые случаи; мы просто используем для него маску подстановки _. Тело функции в последнем образце также очень похоже на функцию zip – разница в том, что она не создаёт пару (x, y), а возвращает f x y. Одна функция высшего порядка может использоваться для решения множества задач, если она достаточно общая. Покажем на небольшом примере, что умеет наша функция zipWith':

```
ghci> zipWith' (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]
ghci> zipWith' max [6,3,2,1] [7,3,1,5]
[7,3,2,5]
ghci> zipWith' (++) ["шелдон ", "леонард "] ["купер", "хофстадтер"]
["шелдон купер","леонард хофстадтер"]
ghci> zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]
ghci> zipWith' (zipWith' (*)) [[1,2,3],[3,5,6],[2,3,4]] [[3,2,2],[3,4,5],[5,4,3]]
[[3,4,6],[9,20,30],[10,12,12]]
```

Как видите, одна-единственная функция высшего порядка может применяться самыми разными способами.

Реализация функции flip

Теперь реализуем ещё одну функцию из стандартной библиотеки, flip. Функция flip принимает функцию и возвращает функцию.

Единственное отличие результирующей функции от исходной – первые два параметра переставлены местами. Мы можем реализовать flip следующим образом:

```
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = g
  where g x y = f y x
```

Читая декларацию типа, мы видим, что функция принимает на вход функцию с параметрами типов a и b и возвращает функцию с параметрами b и a. Так как все функции на самом деле каррированы, вторая пара скобок не нужна, поскольку символ -> правоассоциативен. Тип (a -> b -> c) -> (b -> a -> c) – то же самое, что и тип (a -> b -> c) -> (b -> (a -> c)), а он, в свою очередь, представляет то же самое, что и тип (a -> b -> c) -> b -> a -> c. Мы записали, что g x y = f y x. Если это верно, то верно и следующее: f y x = g x y. Держите это в уме – мы можем реализовать функцию ещё проще.

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f y x = f x y
```

Здесь мы воспользовались тем, что функции каррированы. Когда мы вызываем функцию flip' f без параметров y и x, то получаем функцию, которая принимает два параметра, но переставляет их при вызове. Даже несмотря на то, что такие «перевёрнутые» функции обычно передаются в другие функции, мы можем воспользоваться преимуществами каррирования при создании ФВП, если подумаем наперёд и запишем, каков будет конечный результат при вызове полностью определённых функций.

```
ghci> zip [1,2,3,4,5,6] "привет"
[(1,'п'),(2,'р'),(3,'и'),(4,'в'),(5,'е'),(6,'т')]
ghci> flip' zip [1,2,3,4,5] "привет"
[('п',1),('р',2),('и',3),('в',4),('е',5),('т',6)]
ghci> zipWith div [2,2..] [10,8,6,4,2]
[0,0,0,0,1]
ghci> zipWith (flip' div) [2,2..] [10,8,6,4,2]
[5,4,3,2,1]
```

Если применить функцию flip' к zip, то мы получим функцию, похожую на zip, за исключением того что элементы первого списка будут оказываться вторыми элементами пар результирующего списка, и наоборот. Функция flip' div делит свой второй параметр на

первый, так что если мы передадим ей числа 2 и 10, то результат будет такой же, что и в случае div 10 2.

Инструментарий функционального программиста

Как функциональные программисты мы редко будем обрабатывать одно значение. Обычно нам хочется сразу взять набор чисел, букв или значений каких-либо иных типов, а затем преобразовать всё это множество для получения результата. В данном разделе будет рассмотрен ряд полезных функций, которые позволяют нам работать с множествами значений.

Функция map

Функция map берёт функцию и список и применяет функцию к каждому элементу списка, формируя новый список. Давайте изучим сигнатуру этой функции и посмотрим, как она определена.

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Сигнатура функции говорит нам, что функция map принимает на вход функцию, которая вычисляет значение типа b по параметру типа a, список элементов типа a и возвращает список элементов типа b. Интересно, что глядя на сигнатуру функции вы уже можете сказать, что она делает. Функция map – одна из самых универсальных ФВП, и она может использоваться миллионом разных способов. Рассмотрим её в действии:

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!") ["БУХ", "БАХ", "ПАФ"]
["БУХ!","БАХ!","ПАФ!"]
ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

Возможно, вы заметили, что нечто аналогичное можно сделать с помощью генератора списков. Вызов `map (+3) [1,5,3,1,6]` – это то же самое, что и `[x+3 | x <- [1,5,3,1,6]]`. Тем не менее использование функции `map` обеспечивает более читаемый код в случаях, когда вы просто применяете некоторую функцию к списку. Особенно когда применяются отображения к отображениям (`map` – к результатам выполнения функции `map`): тогда всё выражение с кучей скобок может стать нечитаемым.

Функция filter

Функция `filter` принимает предикат и список, а затем возвращает список элементов, удовлетворяющих предикату. *Предикат* – это функция, которая говорит, является ли что-то истиной или ложью, – то есть функция, возвращающая булевское значение. Сигнатура функции и её реализация:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
  | p x       = x : filter p xs
  | otherwise = filter p xs
```

Довольно просто. Если выражение `p x` истинно, то элемент добавляется к результирующему списку. Если нет – элемент пропускается.

Несколько примеров:

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> let notNull x = not (null x) in filter notNull [[1],[],[3,4],[]]
[[1],[3,4]]
ghci> filter (`elem` ['а'..'я']) "тЫ СМЕЕШЬСя, ВЕДЬ я ДрУГой"
"тяярой"
ghci> filter (`elem` ['А'..'Я']) "я Смеюсь, Ведь ты такОЙ же"
"СВОЙ"
```

Того же самого результата можно достичь, используя генера-

торы списков и предикаты. Нет какого-либо правила, диктующего вам, когда использовать функции map и filter, а когда – генераторы списков. Вы должны решить, что будет более читаемым, основываясь на коде и контексте. В генераторах списков можно применять несколько предикатов; при использовании функции filter придётся проводить фильтрацию несколько раз или объединять предикаты с помощью логической функции &&. Вот пример:

```
ghci> filter (<15) (filter even [1..20])
[2,4,6,8,10,12,14]
```

Здесь мы берём список [1..20] и фильтруем его так, чтобы остались только чётные числа. Затем список передаётся функции filter (<15), которая избавляет нас от чисел 15 и больше. Вот версия с генератором списка:

```
ghci> [ x | x <- [1..20], x < 15, even x]
[2,4,6,8,10,12,14]
```

Мы используем генератор для извлечения элементов из списка [1..20], а затем указываем условия, которым должны удовлетворять элементы результирующего списка.

Помните нашу функцию быстрой сортировки (см. предыдущую главу, раздел «Сортируем, быстро!»)? Мы использовали генераторы списков для фильтрации элементов меньших (или равных) и больших, чем опорный элемент. Той же функциональности можно добиться и более понятным способом, используя функцию filter:

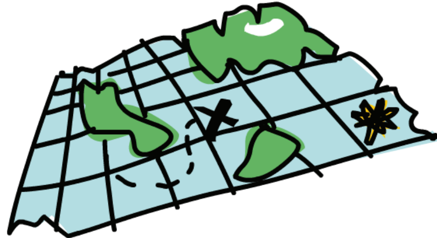
```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
   let smallerSorted = quicksort (filter (<= x) xs)
       biggerSorted = quicksort (filter (> x) xs)
   in  smallerSorted ++ [x] ++ biggerSorted
```

Ещё немного примеров использования map и filter

Давайте найдём наибольшее число меньше 100 000, которое делится на число 3829 без остатка. Для этого отфильтруем множество возможных вариантов, в которых, как мы знаем, есть решение.

```
largestDivisible :: Integer
largestDivisible = head (filter p [100000,99999..])
  where p x = x `mod` 3829 == 0
```

Для начала мы создали список всех чисел меньших 100 000 в порядке убывания. Затем отфильтровали список с помощью предиката. Поскольку числа отсортированы в убывающем порядке, наибольшее из них, удовлетворяющее предикату, будет первым элементом отфильтрованного списка. Нам даже не нужно использовать конечный список для нашего базового множества. Снова «лень в действии»! Поскольку мы используем только «голову» списка, нам неважно, конечен полученный список или бесконечен. Вычисления прекращаются, как только находится первое подходящее решение.



Теперь мы собираемся найти сумму всех нечётных квадратов меньших 10 000. Но для начала познакомимся с функцией `takeWhile`: она пригодится в нашем решении. Она принимает предикат и список, а затем начинает обход списка с его «головы», возвращая те его элементы, которые удовлетворяют предикату. Как только найден элемент, не удовлетворяющий предикату, обход останавливается. Если бы мы хотели получить первое слово строки `"слоны умеют веселиться"`, мы могли бы сделать такой вызов: `takeWhile (/=' ') "слоны умеют веселиться"`, и функция вернула бы `"слоны"`.

Итак, в первую очередь начнём применять функцию `(   2)` к бесконечному списку `[1..]`. Затем отфильтруем список, чтобы в нём были только нечётные элементы. Далее возьмём из него значения, меньшие `10000`. И, наконец, получим сумму элементов этого списка. Нам даже не нужно задавать для этого функцию – достаточно будет одной строки в GHCi:

```
ghci> sum (takeWhile (<10000) (filter odd (map (  2) [1..])))
166650
```

Потрясающе! Мы начали с некоторых начальных данных (бесконечный список натуральных чисел) и затем применяли к ним функцию, фильтровали, прореживали до тех пор, пока список не

удовлетворил нашим запросам, а затем просуммировали его. Можно было бы воспользоваться генераторами списков для той же цели:

```
ghci> sum (takeWhile (<10000) [m | m <- [n  2 | n <- [1..]], odd m])
166650
```

В следующей задаче мы будем иметь дело с рядами Коллатца. Берём натуральное число. Если это число чётное, делим его на два. Если нечётное – умножаем его на 3 и прибавляем единицу. Берём получившееся значение и снова повторяем всю процедуру, получаем новое число, и т. д. В сущности, у нас получается цепочка чисел. С какого бы значения мы ни начали, цепочка заканчивается на единице. Если бы начальным значением было 13, мы бы получили такую последовательность: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. Всё по вышеприведенной схеме: 13 × 3 + 1 равняется 40; 40, разделённое на 2, равно 20, и т. д. Как мы видим, цепочка имеет 10 элементов.

Теперь требуется выяснить: если взять все стартовые числа от 1 до 100, как много цепочек имеют длину больше 15? Для начала напишем функцию, которая создаёт цепочку:

```
chain :: Integer -> [Integer]
chain 1 = [1]
chain n
     | even n = n:chain (n `div` 2)
     | odd n  = n:chain (n*3 + 1)
```

Так как цепочка заканчивается на единице, это базовый случай. Получилась довольно-таки стандартная рекурсивная функция.

```
ghci> chain 10
[10,5,16,8,4,2,1]
ghci> chain 1
[1]
ghci> chain 30
[30,15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1]
```

Так! Вроде бы работает правильно. Ну а теперь функция, которая ответит на наш вопрос:

```
numLongChains :: Int
numLongChains = length (filter isLong (map chain [1..100]))
   where isLong xs = length xs > 15
```

Мы применяем функцию `chain` к списку `[1..100]`, чтобы получить список цепочек; цепочки также являются списками. Затем фильтруем их с помощью предиката, который проверяет длину цепочки. После фильтрации смотрим, как много цепочек осталось в результирующем списке.

ПРИМЕЧАНИЕ. *Эта функция имеет тип `numLongChains :: Int`, потому что `length` возвращает значение типа `Int` вместо экземпляра класса `Num` – так уж сложилось исторически. Если мы хотим вернуть более общий тип, имеющий экземпляр класса `Num`, нам надо применить функцию `fromIntegral` к результату, возвращённому функцией `length`.*

Функция map для функций нескольких переменных

Используя функцию `map`, можно проделывать, например, такие штуки: `map (*) [0..]` – если не для какой-либо практической цели, то хотя бы для того, чтобы продемонстрировать, как работает каррирование, и показать, что функции (частично применённые) – это настоящие значения, которые можно передавать в другие функции или помещать в списки (но нельзя представлять в виде строк). До сих пор мы применяли к спискам только функции с одним параметром, вроде `map (*2) [0..]`, чтобы получить список типа `(Num a) => [a]`, но с тем же успехом можем использовать `(*) [0..]` безо всяких проблем. При этом числа в списке будут применены к функции `*`, тип которой `(Num a) => a -> a -> a`. Применение только одного параметра к функции двух параметров возвращает функцию одного параметра. Применив оператор `*` к списку `[0..]`, мы получаем список функций, которые принимают только один параметр, а именно `(Num a) => [a -> a]`. Список, возвращаемый выражением `map (*) [0..]`, также можно получить, записав `[(0*),(1*),(2*),(3*),(4*),(5*)`...

```
ghci> let listOfFuns = map (*) [0..]
ghci> (listOfFuns !! 4) 5
20
```

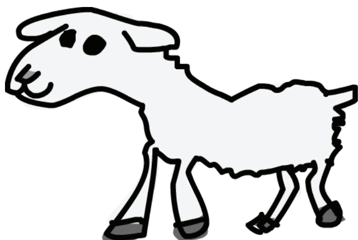
Элемент с номером четыре из списка содержит функцию, которая выполняет умножение на четыре – `(4*)`. Затем мы применяем значение `5` к этой функции. Это то же самое, что записать `(4*) 5` или просто `4 * 5`.

Лямбда-выражения

Лямбда-выражения – это анонимные функции, которые используются, если некоторая функция нужна нам только однажды. Как правило, мы создаём анонимные функции с единственной целью: передать их функции высшего порядка в качестве параметра. Чтобы записать лямбда-выражение, пишем символ \ (напоминающий, если хорошенько напрячь воображение, греческую букву лямбда – λ), затем записываем параметры, разделяя их пробелами. Далее пишем знак `->` и тело функции. Обычно мы заключаем лямбду в круглые скобки, иначе она продолжится до конца строки вправо.

Если вы обратитесь к примеру, приведенному в предыдущем разделе, то увидите, что мы создали функцию `isLong` в секции `where` функции `numLongChains` только для того, чтобы передать её в фильтр. Вместо этого можно использовать анонимную функцию:

```
numLongChains :: Int
numLongChains = length (filter (\xs -> length xs > 15) (map chain [1..100]))
```



Анонимные функции являются выражениями, поэтому мы можем использовать их таким способом, как в примере. Выражение `(\xs -> length xs > 15)` возвращает функцию, которая говорит нам, больше ли `15` длина переданного списка.

Те, кто не очень хорошо понимает, как работает каррирование и частичное применение функций, часто используют анонимные функции там, где не следует. Например, выражения `map (+3) [1,6,3,2]` и `map (\x -> x + 3) [1,6,3,2]` эквивалентны, так как `(+3)` и `(\x -> x +3)` – это функции, которые добавляют тройку к аргументу. Излишне говорить, что использование анонимной функции в этом случае неоправданно, так как частичное применение значительно легче читается.

Как и обычные функции, лямбда-выражения могут принимать произвольное количество параметров:

```
ghci> zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
[153.0,61.5,31.0,15.75,6.6]
```

По аналогии с обычными функциями, можно выполнять сопоставление с образцом в лямбда-выражениях. Единственное отличие в том, что нельзя определить несколько образцов для одного параметра – например, записать для одного параметра образцы `[]` и `(x:xs)` и рассчитывать, что выполнение перейдёт к образцу `(x:xs)` в случае неудачи с `[]`. Если сопоставление с образцом в анонимной функции заканчивается неудачей, происходит ошибка времени выполнения, так что поосторожнее с этим!

```
ghci> map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
[3,8,9,8,7]
```

Обычно анонимные функции заключаются в круглые скобки, если только мы не хотим, чтобы лямбда-выражение заняло всю строку. Интересная деталь: поскольку все функции каррированы по умолчанию, допустимы две эквивалентные записи.

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z

addThree' :: Int -> Int -> Int -> Int
addThree' = \x -> \y -> \z -> x + y + z
```

Если мы объявим функцию подобным образом, то станет понятно, почему декларация типа функции представлена именно в таком виде. И в декларации типа, и в теле функции имеются три символа `->`. Конечно же, первый способ объявления функций значительно легче читается; второй – это всего лишь очередная возможность продемонстрировать каррирование.

ПРИМЕЧАНИЕ. *Обратите внимание на то, что во втором примере анонимные функции не заключены в скобки. Когда вы пишете анонимную функцию без скобок, предполагается, что вся часть после символов `->` относится к этой функции. Так что пропуск скобок экономит на записи. Конечно, ничто не мешает использовать скобки, если это вам больше нравится.*

Тем не менее есть случаи, когда использование такой нотации оправдано. Я думаю, что функция `flip` будет лучше читаться, если мы объявим её так:

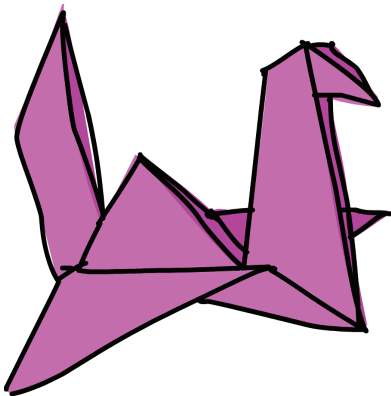
```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f = \x y -> f y x
```

Несмотря на то что эта запись равнозначна `flip' f x y = f y x`, мы даём понять, что данная функция чаще всего используется для создания новых функций. Самый распространённый сценарий использования `flip` – вызов её с некоторой функцией и передача результирующей функции в `map` или `zipWith`:

```
ghci> zipWith (flip (++)) ["люблю тебя", "любишь меня"] ["я ", "ты "]
["я люблю тебя","ты любишь меня"]
ghci> map (flip subtract 20) [1,2,3,4]
[19,18,17,16]
```

Итак, используйте лямбда-выражения таким образом, когда хотите явно показать, что ваша функция должна быть частично применена и передана далее как параметр.

Я вас сверну!



Когда мы разбирались с рекурсией, то во всех функциях для работы со списками наблюдали одну и ту же картину. Базовым случаем, как правило, был пустой список. Мы пользовались образцом `(x:xs)` и затем делали что-либо с «головой» и «хвостом» списка. Как выясняется, это очень распространённый шаблон. Были придуманы несколько полезных функций для его инкапсуляции. Такие функции называются *свёртками* (folds). Свёртки позволяют свести структуру данных (например, список) к одному значению.

Функция свёртки принимает бинарную функцию, начальное значение (мне нравится называть его «аккумулятором») и список.

Бинарная функция принимает два параметра. Она вызывается с аккумулятором и первым (или последним) элементом из списка и вычисляет новое значение аккумулятора. Затем функция вызывается снова, с новым значением аккумулятора и следующим элементом из списка, и т. д. То, что остаётся в качестве значения аккумулятора после прохода по всему списку, и есть результат свёртки.

Левая свёртка foldl

Для начала рассмотрим функцию `foldl` – свёртка слева. Она сворачивает список, начиная с левой стороны. Бинарная функция применяется для начального значения и первого элемента списка, затем для вновь вычисленного аккумулятора и второго элемента списка и т. д.

Снова реализуем функцию `sum`, но на этот раз будем пользоваться свёрткой вместо явной рекурсии.

0+3 [3,5,2,1]

3+5 [5,2,1]

8+2 [2,1]

10+1 [1]

11

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

Проверка – раз, два, три!

```
ghci> sum' [3,5,2,1]
11
```

Давайте посмотрим более внимательно, как работает функция `foldl`. Бинарная функция – это лямбда-выражение `(\acc x -> acc + x)`, нуль – стартовое значение, и `xs` – список. В самом начале нуль используется как значение аккумулятора, а `3` – как значение образца `x` (текущий элемент). Выражение `(0+3)` в результате даёт `3`; это становится новым значением аккумулятора. Далее, `3` используется как значение аккумулятора и `5` – как текущий элемент; новым значением аккумулятора становится `8`. На следующем шаге `8` – значение аккумулятора, `2` – текущий элемент, новое значение аккумулятора становится равным `10`. На последнем шаге `10` из аккумулятора и `1` как текущий элемент дают `11`. Поздравляю – вы только что выполнили свёртку списка!

Диаграмма на предыдущей странице иллюстрирует работу свёртки шаг за шагом, день за днем. Цифры слева от знака + представляют собой значения аккумулятора. Как вы можете видеть, аккумулятор будто бы «поедает» список, начиная с левой стороны. Нямням-ням! Если мы примем во внимание, что функции каррированы, то можем записать определение функции ещё более лаконично:

```
sum' :: (Num a) => [a] -> a
sum' = foldl (+) 0
```

Анонимная функция `(\acc x -> acc + x)` – это то же самое, что и оператор `(+)`. Мы можем пропустить `xs` в параметрах, потому что вызов `foldl (+) 0` вернёт функцию, которая принимает список. В общем, если у вас есть функция вида `foo a = bar b a`, вы всегда можете переписать её как `foo = bar b`, так как происходит каррирование.

Ну что ж, давайте реализуем ещё одну функцию с левой свёрткой перед тем, как перейти к правой. Уверен, все вы знаете, что функция `elem` проверяет, является ли некоторое значение частью списка, так что я не буду этого повторять (тьфу ты – не хотел, а повторил!). Итак:

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldl (\acc x -> if x == y then True else acc) False ys
```

Что мы имеем? Стартовое значение и аккумулятор – булевские значения. Тип аккумулятора и стартового значения в свёртках всегда совпадают. Запомните это правило: оно может подсказать вам, что следует использовать в качестве стартового значения, если вы затрудняетесь. В данном случае мы начинаем со значения `False`. В этом есть смысл: предполагается, что в списке нет искомого элемента. Если мы вызовем функцию свёртки с пустым списком, то результатом будет стартовое значение. Затем мы проверяем текущий элемент на равенство искомому. Если это он – устанавливаем в `True`. Если нет – не изменяем аккумулятор. Если он прежде был равен значению `False`, то остаётся равным `False`, так как текущий элемент – не искомый. Если же был равен `True`, мы опять-таки оставляем его неизменным.

Правая свёртка foldr

Правая свёртка, `foldr`, работает аналогично левой, только аккумулятор поглощает значения, начиная справа. Бинарная функция левой

свёртки принимает аккумулятор как первый параметр, а текущее значение – как второй (\acc x -> ...); бинарная функция правой свёртки принимает текущее значение как первый параметр и аккумулятор – как второй (\x acc -> ...). То, что аккумулятор находится с правой стороны, в некотором смысле логично, поскольку он поглощает значения из списка справа.

Значение аккумулятора (и, следовательно, результат) функции foldr могут быть любого типа. Это может быть число, булевское значение или даже список. Мы реализуем функцию map с помощью правой свёртки. Аккумулятор будет списком; будем накапливать пересчитанные элементы один за другим. Очевидно, что начальным элементом является пустой список:

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs
```

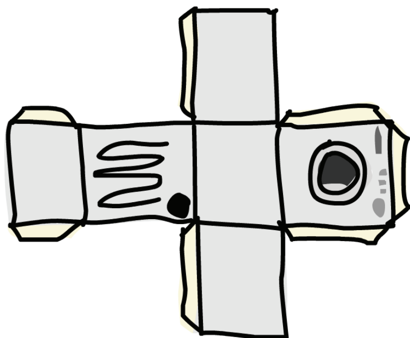
Если мы применяем функцию (+3) к списку [1,2,3], то обрабатываем список справа. Мы берём последний элемент, тройку, применяем к нему функцию, и результат оказывается равен 6. Затем добавляем это число к аккумулятору, который был равен []. 6:[] – то же, что и [6]; это новое значение аккумулятора. Мы применяем функцию (+3) к значению 2, получаем 5 и при помощи конструктора списка : добавляем его к аккумулятору, который становится равен [5,6]. Применяем функцию (+3) к значению 1, добавляем результат к аккумулятору и получаем финальное значение [4,5,6].

Конечно, можно было бы реализовать эту функцию и при помощи левой свёртки:

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldl (\acc x -> acc ++ [f x]) [] xs
```

Но операция конкатенации ++ значительно дороже, чем конструктор списка :, так что мы обычно используем правую свёртку, когда строим списки из списков.

Если вы обратите список задом наперед, то сможете выполнять правую свёртку с тем же результатом, что даёт левая свёртка, и наоборот. В некоторых случаях обращать список не требуется. Функцию sum можно реализовать как с помощью левой, так и с помощью правой свёртки. Единственное серьёзное отличие: правые свёртки работают на бесконечных списках, а левые – нет! Оно и понятно:



если вы берёте бесконечный список в некоторой точке и затем сворачиваете его справа, рано или поздно вы достигаете начала списка. Если же вы берёте бесконечный список в некоторой точке и пытаетесь свернуть его слева, вы никогда не достигнете конца!

Свёртки могут быть использованы для реализации любой функции, где вы вычисляете что-либо за один обход списка[1]. Если вам нужно обойти список для того, чтобы что-либо вычислить, скорее всего, вам нужна свёртка. Вот почему свёртки, наряду с функциями `map` и `filter`, – одни из наиболее часто используемых функций в функциональном программировании.

Функции foldl1 и foldr1

Функции `foldl1` и `foldr1` работают примерно так же, как и функции `foldl` и `foldr`, только нет необходимости явно задавать стартовое значение. Они предполагают, что первый (или последний) элемент списка является стартовым элементом, и затем начинают свёртку со следующим элементом. Принимая это во внимание, функцию `maximum` можно реализовать следующим образом:

```
maximum' :: (Ord a) => [a] -> a
maximum' = foldl1 max
```

Мы реализовали функцию `maximum`, используя `foldl1`. Вместо использования начального значения функция `foldl1` предполагает, что таковым является первый элемент списка, после чего перемещается к следующему. Поэтому всё, что ей необходимо, – это бинарная функция и сворачиваемый лист! Мы начинаем с «головы» списка и сравниваем каждый элемент с аккумулятором. Если элемент больше аккумулятора, мы сохраняем его в качестве нового значения аккумулятора; в противном случае сохраняем старое. Мы

[1] Это так. В качестве упражнения повышенной сложности читателю рекомендуется реализовать при помощи свёртки функции `drop` и `dropWhile` из стандартной библиотеки. – *Прим. ред.*

передаём функцию `max` в качестве параметра `foldl1`, поскольку она ровно это и делает: берёт два значения и возвращает большее. К моменту завершения свёртки останется самый большой элемент.

Поскольку эти функции требуют, чтобы сворачиваемые списки имели хотя бы один элемент, то, если вызвать их с пустым списком, произойдёт ошибка времени выполнения.

С другой стороны, функции `foldl` и `foldr` хорошо работают с пустыми списками. Подумайте, имеет ли смысл свёртка для пустых списков в вашем контексте. Если функция не имеет смысла для пустого списка, то, возможно, вы захотите использовать функции `foldl1` или `foldr1` для её реализации.

Примеры свёрток

Для того чтобы показать, насколько мощны свёртки, мы собираемся реализовать с их помощью несколько стандартных библиотечных функций. Во-первых, реализуем свою версию функции `reverse`:

```
reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []
```

Здесь мы обращаем список, пользуясь пустым списком как начальным значением аккумулятора, и, обходя затем исходный список слева, добавляем текущий элемент в начало аккумулятора.

Функция `\acc x -> x : acc` – почти то же, что и операция `:`, за исключением порядка следования параметров. Поэтому функцию `reverse'` можно переписать и так:

```
reverse' :: [a] -> [a]
reverse' = foldl (flip (:)) []
```

Теперь реализуем функцию `product`:

```
product' :: (Num a) => [a] -> a
product' = foldl (*) 1
```

Чтобы вычислить произведение всех элементов списка, следует начать с аккумулятора равного `1`. Затем мы выполняем свёртку функцией `(*)`, которая перемножает каждый элемент списка на аккумулятор.

Вот реализация функции `filter`:

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc) []
```

Здесь начальное значение аккумулятора является пустым списком. Мы сворачиваем список справа налево и проверяем каждый элемент, пользуясь предикатом p. Если p x возвращает истину, элемент x помещается в начало аккумулятора. В противном случае аккумулятор остаётся без изменения.

Напоследок реализуем функцию last:

```
last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

Для получения последнего элемента списка мы применяем foldr1. Начинаем с «головы» списка, а затем применяем бинарную функцию, которая игнорирует аккумулятор и устанавливает текущий элемент списка как новое значение аккумулятора. Как только мы достигаем конца списка, аккумулятор – то есть последний элемент – возвращается в качестве результата свёртки.

Иной взгляд на свёртки

Есть ещё один способ представить работу правой и левой свёртки. Скажем, мы выполняем правую свёртку с бинарной функцией f и стартовым значением z. Если мы применяем правую свёртку к списку [3,4,5,6], то на самом деле вычисляем вот что:

```
f 3 (f 4 (f 5 (f 6 z)))
```

Функция f вызывается с последним элементом в списке и аккумулятором; получившееся значение передаётся в качестве аккумулятора при вызове функции с предыдущим значением, и т. д. Если мы примем функцию f за операцию сложения и начальное значение за нуль, наш пример преобразуется так:

```
3 + (4 + (5 + (6 + 0)))
```

Или, если записать оператор + как префиксную функцию, получится:

```
(+) 3 ((+) 4 ((+) 5 ((+) 6 0)))
```

Аналогичным образом левая свёртка с бинарной функцией g и аккумулятором z является эквивалентом выражения

```
g (g (g (g z 3) 4) 5) 6
```

Если заменить бинарную функцию на flip (:) и использовать [] как аккумулятор (выполняем обращение списка), подобная запись эквивалентна следующей:

```
flip (:) (flip (:) (flip (:) (flip (:) [] 3) 4) 5) 6
```

Если вычислить это выражение, мы получим [6,5,4,3].

Свёртка бесконечных списков

Взгляд на свёртки как на последовательное применение функции к элементам списка помогает понять, почему правая свёртка иногда отлично работает с бесконечными списками. Давайте реализуем функцию and с помощью foldr, а потом выпишем последовательность применений, как мы это делали в предыдущих примерах. Тогда мы увидим, как ленивость языка Haskell позволяет правой свёртке обрабатывать бесконечные списки.

Функция and принимает список значений типа Bool и возвращает False, если хотя бы один из элементов равен False; в противном случае она возвращает True. Мы будем обходить список справа, используя True как начальное значение. В качестве бинарной функции будем использовать операцию &&, потому что должны вернуть True только в том случае, когда все элементы списка истинны. Функция && возвращает False, если хотя бы один из параметров равен False, поэтому если мы встретим в списке False, то аккумулятор будет установлен в значение False и окончательный результат также будет False, даже если среди оставшихся элементов списка обнаружатся истинные значения.

```
and' :: [Bool] -> Bool
and' xs = foldr (&&) True xs
```

Зная, как работает foldr, мы видим, что выражение and' [True,False,True] будет вычисляться следующим образом:

```
True && (False && (True && True))
```

Последнее True здесь – это начальное значение аккумулятора, тогда как первые три логических значения взяты из списка [True,False,True]. Если мы попробуем вычислить результат этого выражения, получится False.

А что если попробовать то же самое с бесконечным списком, скажем, repeat False? Если мы выпишем соответствуюшие применения, то получится вот что:

```
False && (False && (False && (False …
```

Ленивость Haskell позволит вычислить только то, что действительно необходимо. Функция && устроена таким образом, что если её первый параметр False, то второй просто игнорируется, поскольку и так ясно, что результат должен быть False.

Функция foldr будет работать с бесконечными списками, если бинарная функция, которую мы ей передаём, не требует обязательного вычисления второго параметра, если значения первого ей достаточно для вычисления результата. Такова функция && – ей неважно, каков второй параметр, при условии, что первый — False.

Сканирование

Функции scanl и scanr похожи на foldl и foldr, только они сохраняют все промежуточные значения аккумулятора в список. Также существуют функции scanl1 и scanr1, которые являются аналогами foldl1 и foldr1.

```
ghci> scanl (+) 0 [3,5,2,1]
[0,3,8,10,11]
ghci> scanr (+) 0 [3,5,2,1]
[11,8,3,1,0]
ghci> scanl1 (\acc x -> if x > acc then x else acc) [3,4,5,3,7,9,2,1]
[3,4,5,5,7,9,9,9]
ghci> scanl (flip (:)) [] [3,2,1]
[[],[3],[2,3],[1,2,3]]
```

При использовании функции scanl финальный результат окажется в последнем элементе итогового списка, тогда как функция scanr поместит результат в первый элемент.

Функции сканирования используются для того, чтобы увидеть, как работают функции, которые можно реализовать как свёртки.

Давайте ответим на вопрос: как много корней натуральных чисел нам потребуется, чтобы их сумма превысила 1000? Чтобы получить сумму квадратов натуральных чисел, воспользуемся `map sqrt [1..]`. Теперь, чтобы получить сумму, прибегнем к помощи свёртки, но поскольку нам интересно знать, как увеличивается сумма, будем вызывать функцию `scanl1`. После вызова `scanl1` посмотрим, сколько элементов не превышают 1000. Первый элемент в результате работы функции `scanl1` должен быть равен единице. Второй будет равен 1 плюс квадратный корень двух. Третий элемент – это корень трёх плюс второй элемент. Если у нас x сумм меньших 1000, то нам потребовалось (x+1) элементов, чтобы превзойти 1000.

```
sqrtSums :: Int
sqrtSums = length (takeWhile (< 1000) (scanl1 (+) (map sqrt [1..]))) + 1
```

```
ghci> sqrtSums
131
ghci> sum (map sqrt [1..131])
1005.0942035344083
ghci> sum (map sqrt [1..130])
993.6486803921487
```

Мы задействовали функцию `takeWhile` вместо `filter`, потому что последняя не работает на бесконечных списках. В отличие от нас, функция `filter` не знает, что список возрастает, поэтому мы используем `takeWhile`, чтобы отсечь список, как только сумма превысит 1000.

Применение функций с помощью оператора $

Пойдем дальше. Теперь объектом нашего внимания станет оператор `$`, также называемый *аппликатором функций*. Прежде всего посмотрим, как он определяется:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Зачем? Что это за бессмысленный оператор? Это просто применение функции! Верно, *почти*, но не совсем!.. В то время как обычное применение функции (с пробелом) имеет высший приоритет, оператор `$` имеет самый низкий приоритет. Применение функции с пробелом левоассоциативно (то есть `f a b c i` – это то же самое, что `(((f a) b) c))`, в то время как применение функции при помощи оператора `$` правоассоциативно.

Всё это прекрасно, но нам-то с того какая польза? Прежде всего оператор `$` удобен тем, что с ним не приходится записывать много вложенных скобок. Рассмотрим выражение `sum (map sqrt [1..130])`. Поскольку оператор `$` имеет самый низкий приоритет, мы можем переписать это выражение как `sum $ map sqrt [1..130]`, сэкономив драгоценные нажатия на клавиши. Когда в функции встречается знак `$`, выражение справа от него используется как параметр для функции слева от него. Как насчёт `sqrt 3 + 4 + 9`? Здесь складываются `9`, `4` и корень из `3`. Если мы хотим получить квадратный корень суммы, нам надо написать `sqrt (3 + 4 + 9)` – или же (в случае использования оператора `$`) `sqrt $ 3 + 4 + 9`, потому что у оператора `$` низший приоритет среди всех операторов. Вот почему вы можете представить символ `$` как эквивалент записи открывающей скобки с добавлением закрывающей скобки в крайней правой позиции выражения.

Посмотрим ещё на один пример:

```
ghci> sum (filter (> 10) (map (*2) [2..10]))
80
```

Очень много скобок, даже как-то уродливо. Поскольку оператор `$` правоассоциативен, выражение `f (g (z x))` эквивалентно записи `f $ g $ z x`. Поэтому пример можно переписать:

```
sum $ filter (> 10) $ map (*2) [2..10]
```

Но кроме избавления от скобок оператор `$` означает, что само применение функции может использоваться как и любая другая функция. Таким образом, мы можем, например, применить функцию к списку функций:

```
ghci> map ($ 3) [(4+), (10*), (  2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

Функция `($ 3)` применяется к каждому элементу списка. Если задуматься о том, что она делает, то окажется, что она берёт функцию и применяет её к числу `3`. Поэтому в данном примере каждая функция из списка применится к тройке, что, впрочем, и так очевидно.

Композиция функций

В математике композиция функций определяется следующим образом:

$(f \circ g)(x) = f(g(x))$

Это значит, что композиция двух функций создаёт новую функцию, которая, когда её вызывают, скажем, с параметром x, эквивалентна вызову g с параметром x, а затем вызову f с результатом первого вызова в качестве своего параметра.

В языке Haskell композиция функций понимается точно так же. Мы создаём её при помощи оператора `(.)`, который определён следующим образом:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

По декларации типа функция `f` должна принимать параметр того же типа, что и результат функции `g`. Таким образом, результирующая функция принимает параметр того же типа, что и функция `g`, и возвращает значение того же типа, что и функция `f`. Выражение `negate . (* 3)` возвращает функцию, которая принимает число, умножает его на три и меняет его знак на противоположный.

Одно из применений композиции функций – это создание функций «на лету» для передачи их другим функциям в качестве параметров. Конечно, мы можем использовать для этого анонимные функции, но зачастую композиция функций понятнее и лаконичнее. Допустим, что у нас есть список чисел и мы хотим сделать их отрицательными. Один из способов сделать это – получить абсо-

лютное значение числа (модуль), а затем перевести его в отрицательное, вот так:

```
ghci> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Обратите внимание на анонимную функцию и на то, как она похожа на результирующую композицию функций. А вот что выйдет, если мы воспользуемся композицией:

```
ghci> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Невероятно! Композиция функций правоассоциативна, поэтому у нас есть возможность включать в неё много функций за один раз. Выражение `f (g (z x))` эквивалентно `(f . g . z) x`. Учитывая это, мы можем превратить

```
ghci> map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

в

```
ghci> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

Функция `negate . sum . tail` принимает список, применяет к нему функцию `tail`, суммирует результат и умножает полученное число на `-1`. Получаем точный эквивалент анонимной функции из предыдущего примера.

Композиция функций с несколькими параметрами

Ну а как насчёт функций, которые принимают несколько параметров? Если мы хотим использовать их в композиции, обычно мы частично применяем их до тех пор, пока не получим функцию, принимающую только один параметр. Запись

```
sum (replicate 5 (max 6.7 8.9))
```

может быть преобразована так:

```
(sum . replicate 5) (max 6.7 8.9)
```

или так :

```
sum . replicate 5 $ max 6.7 8.9
```

Функция replicate 5 применяется к результату вычисления max 6.7 8.9, после чего элементы полученного списка суммируются. Обратите внимание, что функция replicate частично применена так, чтобы у неё остался только один параметр, так что теперь результат max 6.7 8.9 передаётся на вход replicate 5; новым результатом оказывается список чисел, который потом передаётся функции sum.

Если вы хотите переписать выражение с кучей скобок, используя функциональную композицию, можно сначала записать самую внутреннюю функцию с её параметрами, затем поставить перед ней знак $, а после этого пристраивать вызовы всех других функций, записывая их без последнего параметра и разделяя точками. Например, выражение

```
replicate 2 (product (map (*3) (zipWith max [1,2] [4,5])))
```

можно переписать так:

```
replicate 2 . product . map (*3) $ zipWith max [1,2] [4,5]
```

Как из одного выражения получилось другое? Ну, во-первых, мы посмотрели на самую правую функцию и её параметры как раз перед группой закрывающихся скобок. Это функция zipWith max [1,2] [4,5]. Так её и запишем:

```
zipWith max [1,2] [4,5]
```

Затем смотрим на функцию, которая применяется к zipWith max [1,2] [4,5], это map (*3). Поэтому мы ставим между ней и тем, что было раньше, знак $:

```
map (*3) $ zipWith max [1,2] [4,5]
```

Теперь начинаются композиции. Проверяем, какая функция применяется ко всему этому, и присоединяем её к map (*3):

```
product . map (*3) $ zipWith max [1,2] [4,5]
```

Наконец, дописываем функцию replicate 2 и получаем окончательное выражение:

```
replicate 2 . product . map (*3) $ zipWith max [1,2] [4,5]
```

Если выражение заканчивалось на три закрывающие скобки, велики шансы, что у вас получится два оператора композиции.

Бесточечная нотация

Композиция функций часто используется и для так называемого бесточечного стиля записи функций. Возьмём, для примера, функцию, которую мы написали ранее:

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (+) 0 xs
```

Образец `xs` представлен дважды с правой стороны. Из–за каррирования мы можем пропустить образец `xs` с обеих сторон, так как `foldl (+) 0` создаёт функцию, которая принимает на вход список. Если мы запишем эту функцию как `sum' = foldl (+) 0`, такая запись будет называться *бесточечной*. А как записать следующее выражение в бесточечном стиле?

```
fn x = ceiling (negate (tan (cos (max 50 x))))
```

Мы не можем просто избавиться от образца `x` с обеих правых сторон выражения. Образец `x` в теле функции заключен в скобки. Выражение `cos (max 50)` не будет иметь никакого смысла. Вы не можете взять косинус от функции! Всё, что мы можем сделать, – это выразить функцию `fn` в виде композиции функций.

```
fn = ceiling . negate . tan . cos . max 50
```

Отлично! Во многих случаях бесточечная запись легче читается и более лаконична; она заставляет думать о функциях, о том, как их соединение порождает результат, а не о данных и способе их передачи. Можно взять простые функции и использовать композицию как «клей» для создания более сложных. Однако во многих случаях написание функций в бесточечном стиле может делать код менее «читабельным», особенно если функция слишком сложна. Вот почему я не рекомендую создавать длинные цепочки функций, хотя меня частенько обвиняли в пристрастии к композиции. Предпочитаемый стиль – использование выражения `let` для присвоения меток промежуточным результатам или разбиение проблемы на подпроблемы и их совмещение таким образом, чтобы функции

имели смысл для того, кто будет их читать, а не представляли собой огромную цепочку композиций.

Ранее в этой главе мы решали задачу, в которой требовалось найти сумму всех нечётных квадратов меньших 10 000. Вот как будет выглядеть решение, если мы поместим его в функцию:

```
oddSquareSum :: Integer
oddSquareSum = sum (takeWhile (<10000) (filter odd (map (  2) [1..])))
```

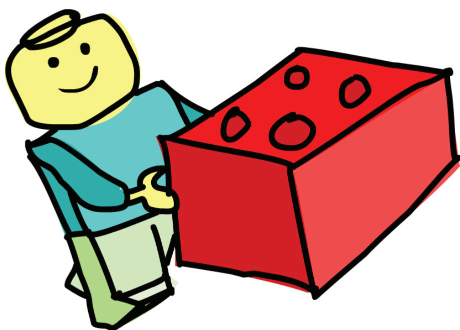
Со знанием композиции функций этот код можно переписать так:

```
oddSquareSum :: Integer
oddSquareSum = sum . takeWhile (<10000) . filter odd $ map (  2) [1..]
```

Всё это на первый взгляд может показаться странным, но вы быстро привыкнете. В подобных записях меньше визуального «шума», поскольку мы убрали все скобки. При чтении такого кода можно сразу сказать, что `filter odd` применяется к результату `map (  2) [1..]`, что затем применяется `takeWhile (<10000)`, а функция `sum` суммирует всё, что получилось в результате.

6

МОДУЛИ



В языке Haskell модуль – это набор взаимосвязанных функций, типов и классов типов. Программа на Haskell – это набор модулей; главный модуль подгружает все остальные и использует функции, определённые в них, чтобы что-либо сделать. Разбиение кода на несколько модулей удобно по многим причинам. Если модуль достаточно общий, экспортируемые им функции могут быть использованы во множестве программ. Если ваш код разделён на несколько самостоятельных модулей, не очень зависящих один от другого (мы говорим, что они слабо связаны), модули могут многократно использоваться в разных проектах. Это отчасти облегчает непростую задачу написания кода, разбивая его на несколько частей, каждая из которых имеет некоторое назначение.

Стандартная библиотека языка Haskell разбита на модули, каждый из которых содержит взаимосвязанные функции и типы, служащие некоторой общей цели. Есть модуль для работы со списками, модуль для параллельного программирования, модуль для работы с комплексными числами и т. д. Все функции, типы и классы типов, с которыми мы имели дело до сих пор, были частью стандартного

модуля Prelude – он импортируется по умолчанию. В этой главе мы познакомимся с несколькими полезными модулями и их функциями. Но для начала посмотрим, как импортировать модули.

Импорт модулей

Синтаксис для импорта модулей в программах на языке Haskell – import ModuleName. Импортировать модули надо прежде, чем вы приступите к определению функций, поэтому обычно импорт делается в начале файла. Конечно же, одна программа может импортировать несколько модулей. Для этого вынесите каждый оператор import в отдельную строку.

Давайте импортируем модуль Data.List, который содержит массу функций для работы со списками, и используем экспортируемую им функцию для того, чтобы написать свою – вычисляющую, сколько уникальных элементов содержит список.

```
import Data.List

numUniques :: (Eq a) => [a] -> Int
numUniques = length . nub
```

Когда выполняется инструкция import Data.List, все функции, экспортируемые модулем Data.List, становятся доступными в глобальном пространстве имён. Это означает, что вы можете вызывать их из любого места программы. Функция nub определена в модуле Data.List; она принимает список и возвращает список, из которого удалены дубликаты элементов исходного списка. Композиция функций length и nub создаёт функцию, которая эквивалентна \xs -> length (nub xs).

ПРИМЕЧАНИЕ. *Чтобы найти нужные функции и уточнить, где они определены, воспользуйтесь сервисом Hoogle, который доступен по адресу* http://www.haskell.org/hoogle/. *Это поистине удивительный поисковый механизм для языка Haskell, который позволяет вести поиск по имени функции, по имени модуля и даже по сигнатуре.*

В интерпретаторе GHCi вы также можете подключить функции из модулей к глобальному пространству имён. Если вы работаете в GHCi и хотите вызывать функции, экспортируемые модулем Data.List, напишите следующее:

```
ghci> :m + Data.List
```

Если требуется подгрузить программные сущности из нескольких модулей, не надо вызывать команду `:m +` несколько раз, так как можно загрузить ряд модулей одновременно:

```
ghci> :m + Data.List Data.Map Data.Set
```

Кроме того, если вы загрузили скрипт, который импортирует модули, то не нужно использовать команду `:m +`, чтобы получить к ним доступ.

Если вам необходимо всего несколько функций из модуля, вы можете выборочно импортировать только эти функции. Если бы вам были нужны только функции `nub` и `sort` из модуля `Data.List`, импорт выглядел бы так:

```
import Data.List (nub, sort)
```

Также вы можете осуществить импорт всех функций из модуля за исключением некоторых. Это бывает полезно, когда несколько модулей экспортируют функции с одинаковыми именами, и вы хотите избавиться от ненужных повторов. Предположим, у вас уже есть функция с именем `nub` и вы хотите импортировать все функции из модуля `Data.List`, кроме `nub`, определённой в нём:

```
import Data.List hiding (nub)
```

Другой способ разрешения конфликтов имён – квалифицированный импорт. Модуль `Data.Map`, который содержит структуру данных для поиска значения по ключу, экспортирует несколько функций с теми же именами, что и модуль `Prelude`, например `filter` и `null`. Если мы импортируем модуль `Data.Map` и вызовем функцию `filter`, язык Haskell не будет знать, какую функцию использовать. Вот как можно обойти такую ситуацию:

```
import qualified Data.Map
```

Если после такого импорта нам понадобится функция `filter` из модуля `Data.Map`; мы должны вызывать её как `Data.Map.filter` – просто идентификатор `filter` ссылается на обычную функцию из модуля `Prelude`, которую мы все знаем и любим. Но печатать строку `Data.Map` перед именем каждой функции может и поднадоесть! Вот

почему желательно переименовать модуль при импорте во что-нибудь более короткое:

```
import qualified Data.Map as M
```

Теперь, чтобы сослаться на функцию из `Data.Map`, мы вызываем её как `M.filter`.

Как вы видите, символ `.` используется для обращения к функциям, импортированным из модулей с указанием квалификатора, например: `M.filter`. Мы также помним, что он используется для обозначения композиции функций. Как Haskell узнаёт, что мы имеем в виду? Если мы помещаем символ `.` между квалифицированным именем модуля и функцией без пробелов – это обращение к функции из модуля; во всех остальных случаях – композиция функций.

> **ПРИМЕЧАНИЕ.** *Отличный способ узнать Haskell изнутри – просмотреть документацию к стандартной библиотеке и исследовать все стандартные модули и их функции. Также можно изучить исходные тексты всех модулей. Чтение исходных текстов некоторых модулей – отличный способ освоить язык и прочувствовать его особенности[1].*

Решение задач средствами стандартных модулей

Модули стандартной библиотеки содержат массу функций, способных облегчить программирование на языке Haskell. Познакомимся с некоторыми из них, решая конкретные задачи.

Подсчёт слов

Предположим, что у нас имеется строка, содержащая много слов. Мы хотим выяснить, сколько раз в этой строке встречается каждое слово. Первой функцией, которую мы применим, будет функция `words` из модуля `Data.List`. Эта функция преобразует строку в список строк, в котором каждая строка представляет одно слово из исходной строки. Небольшой пример:

[1] В тех же целях издательством «ДМК Пресс» выпущена книга: Душкин Р. В. Справочник по языку Haskell. – М.: ДМК Пресс, 2008. – 544 стр., ил. ISBN 5-94074-410-9.

```
ghci> words "всё это слова в этом предложении"
["всё","это","слова","в","этом","предложении"]
ghci> words "всё         это слова в      этом      предложении"
["всё","это","слова","в","этом","предложении"]
```

Затем воспользуемся функцией `group`, которая тоже «живёт» в `Data.List`, чтобы сгруппировать одинаковые слова. Эта функция принимает список и собирает одинаковые подряд идущие элементы в подсписки:

```
ghci> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
```

Но что если одинаковые элементы идут в списке не подряд?

```
ghci> group ["бум","бип","бип","бум","бум"]
[["бум"],["бип","бип"],["бум","бум"]]
```

Получаем два списка, содержащих `"бум"`, тогда как нам бы хотелось, чтобы все вхождения одного и того же слова попали в один список. Что делать? Мы можем предварительно отсортировать список! Для этого применим функцию `sort` из `Data.List`. Она принимает список элементов, которые могут быть упорядочены, и возвращает новый список, содержащий те же элементы, но упорядоченные от наименьшего к наибольшему:

```
ghci> sort [5,4,3,7,2,1]
[1,2,3,4,5,7]
ghci> sort ["бум","бип","бип","бум","бум"]
["бип","бип","бум","бум","бум"]
```

Заметим, что строки упорядочены по алфавиту.

Теперь всё необходимое у нас есть, осталось только записать решение. Берём строку, разбиваем её на список слов, сортируем слова и группируем одинаковые. Затем применяем `map` и получаем список вроде `("boom", 3)`; это означает, что слово `"boom"` встретилось в исходной строке трижды.

```
import Data.List

wordNums :: String -> [(String, Int)]
wordNums = map (\ws -> (head ws, length ws)) . group . sort . words
```

Для написания этой функции мы применили композицию функций. Предположим, что мы вызвали функцию `wordNums` для строки `"ya ya yи ya"`. К этой строке применяется функция `words`, результатом которой будет список `["ya","ya","yи","ya"]`. После его сортировки функцией `sort` получим новый список `["ya","ya","ya","yи"]`. Группировка одинаковых подряд идущих слов функцией `group` даст нам список `[["ya","ya","ya"],["yи"]]`. Затем с помощью функции `map` к каждому элементу такого списка (то есть к подсписку) будет применена анонимная функция, которая превращает список в пару – «голова» списка, длина списка. В конечном счёте получаем `[("ya",3),("yи",1)]`.

Вот как можно написать ту же функцию, не пользуясь операцией композиции:

```
wordNums xs = map (\ws -> (head ws, length ws)) (group (sort (words xs)))
```

Кажется, здесь избыток скобок! Думаю, нетрудно заметить, насколько более читаемой делает функцию операция композиции.

Иголка в стоге сена

Следующая наша миссия – написание функции, которая принимает на вход два списка и сообщает, содержится ли первый список целиком где-нибудь внутри второго. Например, список `[3,4]` содержится внутри `[1,2,3,4,5]`, а вот `[2,5]` – уже нет. Список, в котором мы ищем, назовём стогом, а список, который хотим обнаружить, – иголкой.

Чтобы выбраться из этой передряги, воспользуемся функцией `tails` из того же модуля `Data.List`. Она принимает список и последовательно применяет к нему функцию `tail`. Вот пример:

```
ghci> tails "победа"
["победа","обеда","беда","еда","да","а",""]
ghci> tails [1,2,3]
[[1,2,3],[2,3],[3],[]]
```

Возможно, пока неясно, зачем она нам вообще понадобилась. Сейчас увидим.

Предположим, мы хотим найти строку `"обед"` внутри строки `"победа"`. Для начала вызовем `tails` и получим все «хвосты» списка. Затем присмотримся к каждому «хвосту», и если хоть какой-нибудь

из них начинается со строки "обед", значит, иголка найдена! Вот если бы мы искали "фуу" внутри "победа" – тогда да, ни один из «хвостов» с "фуу" не начинается.

Чтобы узнать, не начинается ли одна строка с другой, мы применим функцию isPrefixOf, снова из модуля Data.List. Ей на вход подаются два списка, а она отвечает, начинается второй список с первого или нет.

```
ghci> "гавайи" `isPrefixOf` "гавайи джо"
True
ghci> "ха-ха" `isPrefixOf` "ха"
False
ghci> "ха" `isPrefixOf` "ха"
True
```

Теперь нужно лишь проверить, начинается ли какой-нибудь из хвостов нашего стога с иголки. Тут поможет функция any из Data.List. Она принимает предикат и список и сообщает, удовлетворяет ли этому предикату хотя бы один элемент списка. Внимание:

```
ghci> any (>4) [1,2,3]
False
ghci> any (=='Х') "Грегори Хаус"
True
ghci> any (\x -> x > 5 && x < 10) [1,4,11]
False
```

Соберём все эти функции вместе:

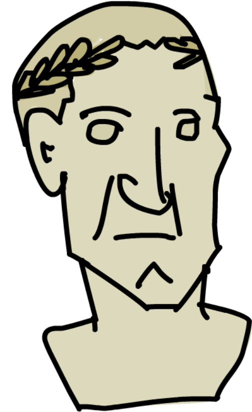
```
import Data.List

isIn :: (Eq a) => [a] -> [a] -> Bool
needle `isIn` haystack = any (needle `isPrefixOf`) (tails haystack)
```

Вот и всё! Функция tails создаёт список «хвостов» нашего стога, а затем мы смотрим, начинается ли что-нибудь из них с иголки. Проверим:

```
ghci> "обед" `isIn` "победа"
True
ghci> [1,2] `isIn` [1,3,5]
False
```

Ой, подождите-ка! Кажется, только что написанная функция уже есть в `Data.List`... Ба-а, и правда! Она называется `isInfixOf` и делает в точности то же, что наша `isIn`.

Салат из шифра Цезаря



Гай Юлий Цезарь возложил на нас ответственное поручение. Необходимо доставить совершенно секретное сообщение в Галлию Марку Антонию. На случай, если нас схватят, мы закодируем сообщение шифром Цезаря, воспользовавшись с этой целью некоторыми функциями из модуля `Data.Char`.

Шифр Цезаря – это простой метод кодирования сообщения путём сдвига каждого символа на фиксированное количество позиций алфавита. На самом деле мы реализуем некую вариацию шифра Цезаря, поскольку не будем ограничиваться только алфавитом, а возьмём весь диапазон символов Unicode.

Для сдвига символов вперёд и назад по кодовой таблице будем применять функции `ord` и `chr`, находящиеся в модуле `Data.Char`. Эти функции преобразуют символы к соответствующим кодам и наоборот:

```
ghci> ord 'a'
97
ghci> chr 97
'a'
ghci> map ord "abcdefgh"
[97,98,99,100,101,102,103,104]
```

Функция `ord 'a'` возвращает 97, поскольку символ `'a'` является девяносто седьмым символом в таблице символов Unicode.

Разность между значениями функции `ord` для двух символов равна расстоянию между ними в кодовой таблице.

Напишем функцию, которая принимает количество позиций сдвига и строку и возвращает новую строку, где каждый символ сдвинут вперёд по кодовой таблице на указанное число позиций.

```
import Data.Char

encode :: Int -> String -> String
encode offset msg = map (\c -> chr $ ord c + offset) msg
```

Кодирование строки тривиально настолько, насколько легко взять сообщение и пройтись по каждому его символу функцией, преобразующей его в соответствующий код, прибавляющей смещение и конвертирующей результат обратно в символ. Любитель композиции записал бы такую функцию как `(chr . (+offset) . ord)`.

```
ghci> encode 3 "привет марк"
"тулеих#пгун"
ghci> encode 5 "прикажи своим людям"
"фхнпелн%цзунс%рйєс"
ghci> encode 1 "веселиться вволю"
"гжтжмйуэт!ггпмя"
```

И вправду закодировано!

Декодирование сообщения – это всего навсего сдвиг обратно на то же количество позиций, на которое ранее проводился сдвиг вперёд.

```
decode :: Int -> String -> String
decode shift msg = encode (negate shift) msg
```

Теперь проверим, декодируется ли сообщение Цезаря:

```
ghci> decode 3 "тулеих#пгун"
"привет марк"
ghci> decode 5 "фхнпелн%цзунс%рйєс"
"прикажи своим людям"
ghci> decode 1 "гжтжмйуэт!ггпмя"
"веселиться вволю"
```

О строгих левых свёртках

В предыдущей главе мы видели, как работает функция `foldl` и как с её помощью реализовывать всякие крутые функции. Правда, мы пока не исследовали одну связанную с `foldl` ловушку: её использование иногда может приводить к так называемым ошибкам переполнения

стека, которые случаются, если программе требуется слишком много места в одном специальном разделе памяти (в сегменте стека). Проиллюстрируем проблему, воспользовавшись свёрткой с функцией + для суммирования списка из сотни единиц:

```
ghci> foldl (+) 0 (replicate 100 1)
100
```

Пока всё работает. Но что если сделать то же самое для списка, содержащего, спасибо доктору Зло, один миллион единиц?

```
ghci> foldl (+) 0 (replicate 1000000 1)
*** Exception: stack overflow
```

Ого, жестоко! Что же случилось? Haskell ленив, поэтому он откладывает реальные вычисления настолько, насколько возможно. Когда мы используем `foldl`, Haskell не вычисляет аккумулятор на каждом шаге. Вместо этого он откладывает вычисление. На каждом следующем шаге он снова ничего не считает, опять откладывая на потом. Ему, правда, приходится сохранять старое отложенное вычисление в памяти, потому что новому может потребоваться его результат. Таким образом, пока свёртка `foldl` радостно торопится по списку, в памяти образуется куча отложенных вычислений, каждое из которых занимает некоторый объём памяти. Рано или поздно это может привести к ошибке переполнения стека.

Вот как Haskell вычисляет выражение `foldl (+) 0 [1,2,3]`:

```
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
```

Здесь видно, что сначала строится большой стек из отложенных вычислений. Затем, по достижении конца списка, начинаются реальные вычисления. Для маленьких списков никакой проблемы нет, а вот если список громадный, с миллионом элементов или даже больше, вы и получите переполнение стека. Дело в том, что все эти отложенные вычисления выполняются рекурсивно. Было бы неплохо, если бы существовала функция, которая вычисления не откладывает, правда же? Она бы работала как-то так:

```
foldl' (+) 0 [1,2,3] =
foldl' (+) 1 [2,3] =
foldl' (+) 3 [3] =
foldl (+) 6 [] =
6
```

Вычисления между шагами свёртки не откладываются – они тут же выполняются. Ну что ж, нам повезло: строгая версия функции `foldl` в модуле `Data.List` есть, и называется она именно `foldl'`. Попробуем-ка с её помощью вычислить сумму миллиона единиц:

```
ghci> foldl' (+) 0 (replicate 1000000 1)
1000000
```

Потрясающий успех! Так что, если, используя `foldl`, получите ошибку переполнения стека, попробуйте переключиться на `foldl'`. Кстати, у `foldl1` тоже есть строгая версия, она называется `foldl1'`.

Поищем числа

Вы прогуливаетесь по улице, и тут к вам подходит старушка и спрашивает: «Простите, а каково первое натуральное число, сумма цифр которого равна 40?»

Ну что, сдулись? Давайте применим Haskell-магию и найдём это число. Если мы, к примеру, просуммируем цифры числа 123, то получим 6. У какого же числа тогда сумма цифр равна 40?

Первым делом напишем функцию, которая считает сумму цифр заданного числа. Внимание, хитрый трюк! Воспользуемся

функцией `show` и преобразуем наше число в строку. Когда у нас будет строка из цифр, мы переведём каждый её символ в число и просуммируем получившийся числовой список. Превращать символ в число будем с помощью функции `digitToInt` из модуля `Data.Char`. Она принимает значение типа `Char` и возвращает `Int`:

```
ghci> digitToInt '2'
2
ghci> digitToInt 'F'
15
ghci> digitToInt 'z'
*** Exception: Char.digitToInt: not a digit 'z'
```

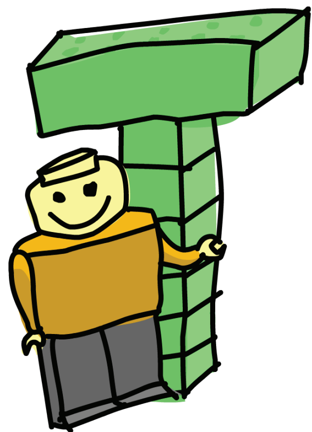
Функция `digitToInt` работает с символами из диапазона от `'0'` до `'9'` и от `'A'` до `'F'` (также и строчными).

Вот функция, принимающая число и возвращающая сумму его цифр:

```
import Data.Char
import Data.List

digitSum :: Int -> Int
digitSum = sum . map digitToInt . show
```

Преобразуем заданное число в строку, пройдёмся по строке функцией `digitToInt`, суммируем получившийся числовой список.

Теперь нужно найти первое натуральное число, применив к которому функцию `digitSum` мы получим в качестве результата число `40`. Для этого воспользуемся функцией `find` из модуля `Data.List`. Она принимает предикат и список и возвращает первый элемент списка, удовлетворяющий предикату. Правда, тип у неё несколько необычный:



```
ghci> :t find
find :: (a -> Bool) -> [a] -> Maybe a
```

Первый параметр – предикат, второй – список, с этим всё ясно. Но что с возвращаемым значением? Что это за

Maybe a? Это тип, который нам до сих пор не встречался. Значение с типом Maybe a немного похоже на список типа [a]. Если список может иметь ноль, один или много элементов, то значение типа Maybe a может иметь либо ноль элементов, либо в точности один. Эту штуку можно использовать, если мы хотим предусмотреть возможность провала. Значение, которое ничего не содержит, – Nothing. Оно аналогично пустому списку. Для конструирования значения, которое что-то содержит, скажем, строку "эй", будем писать Just "эй". Вот как всё это выглядит:

```
ghci> Nothing
Nothing
ghci> Just "эй"
Just "эй"
ghci> Just 3
Just 3
ghci> :t Just "эй"
Just "эй" :: Maybe [Char]
ghci> :t Just True
Just True :: Maybe Bool
```

Видите, значение Just True имеет тип Maybe Bool. Похоже на то, что список, содержащий значения типа Bool, имеет тип [Bool].

Если функция find находит элемент, удовлетворяющий предикату, она возвращает этот элемент, обёрнутый в Just. Если не находит, возвращает Nothing:

```
ghci> find (>4) [3,4,5,6,7]
Just 5
ghci> find odd [2,4,6,8,9]
Just 9
ghci> find (=='x') "меч-кладенец"
Nothing
```

Вернёмся теперь к нашей задаче. Мы уже написали функцию digitSum и знаем, как она работает, так что пришла пора собрать всё вместе. Напомню, что мы хотим найти число, сумма цифр которого равна 40.

```
firstTo40 :: Maybe Int
firstTo40 = find (\x -> digitSum == 40) [1..]
```

Мы просто взяли бесконечный список `[1..]` и начали искать первое число, значение `digitSum` для которого равно 40.

```
ghci> firstTo40
Just 49999
```

А вот и ответ! Можно сделать более общую функцию, которой нужно передавать искомую сумму в качестве параметра:

```
firstTo :: Int -> Maybe Int
firstTo n = find (\x -> digitSum x == n) [1..]
```

И небольшая проверка:

```
ghci> firstTo 27
Just 999
ghci> firstTo 1
Just 1
ghci> firstTo 13
Just 49
```

Отображение ключей на значения

Зачастую, работая с данными из некоторого набора, мы совершенно не заботимся, в каком порядке они расположены. Мы просто хотим получить к ним доступ по некоторому ключу. Например, желая узнать, кто живёт по известному адресу, мы ищем имена тех, кто по этому адресу проживает. В общем случае мы говорим, что ищем значение (чьё-либо имя) по ключу (адрес этого человека).

Почти хорошо: ассоциативные списки

Существует много способов построить отображение «ключ–значение». Один из них – ассоциативные списки. *Ассоциативные списки* (также называемые *словарями* или *отображениями*) – это списки, которые хранят неупорядоченные пары «ключ–значение». Например, мы можем применять ассоциативные списки для хранения телефонных номеров, используя телефонный номер как значение и имя человека как ключ. Нам неважно, в каком порядке они сохранены: всё, что нам требуется, – получить телефонный номер по имени. Наиболее простой способ представить ассоциативный список в языке

Haskell – использовать список пар. Первый компонент пары будет ключом, второй – значением. Вот пример ассоциативного списка с номерами телефонов:

```
phoneBook =
   [("оля","555-29-38")
   ,("женя","452-29-28")
   ,("катя","493-29-28")
   ,("маша","205-29-28")
   ,("надя","939-82-82")
   ,("юля","853-24-92")
   ]
```

За исключением странного выравнивания, это просто список, состоящий из пар строк. Самая частая задача при использовании ассоциативных списков – поиск некоторого значения по ключу. Давайте напишем функцию для этой задачи.

```
findKey :: (Eq k) => k -> [(k,v)] -> v
findKey key xs = snd . head $ filter (\(k,v) -> key == k) xs
```

Всё довольно просто. Функция принимает ключ и список, фильтрует список так, что остаются только совпадающие ключи, получает первую пару «ключ–значение», возвращает значение. Но что произойдёт, если искомого ключа нет в списке? В этом случае мы будем пытаться получить «голову» пустого списка, что вызовет ошибку времени выполнения. Однако следует стремиться к тому, чтобы наши программы были более устойчивыми к «падениям», поэтому давайте используем тип `Maybe`. Если мы не найдём ключа, то вернём значение `Nothing`. Если найдём, будем возвращать `Just` <*то, что нашли*>.

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key [] = Nothing
findKey key ((k,v):xs)
   | key == k = Just v
   | otherwise = findKey key xs
```

Посмотрите на декларацию типа. Функция принимает ключ, который можно проверить на равенство (`Eq`), и ассоциативный список, а затем, возможно, возвращает значение. Выглядит правдоподобно.

Это классическая рекурсивная функция, обрабатывающая список. Базовый случай, разбиение списка на «голову» и «хвост», рекурсивный вызов – всё на месте. Также это классический шаблон для применения свёртки. Посмотрим, как то же самое можно реализовать с помощью свёртки.

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key = foldr (\(k,v) acc -> if key == k then Just v else acc) Nothing
```

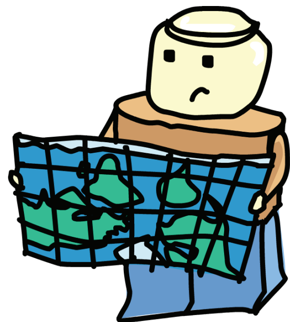
ПРИМЕЧАНИЕ. *Как правило, лучше использовать свёртки для подобных стандартных рекурсивных обходов списка вместо явного описания рекурсивной функции, потому что свёртки легче читаются и понимаются. Любой человек догадается, что это свёртка, как только увидит вызов функции `foldr` – однако потребуется больше интеллектуальных усилий для того, чтобы распознать явно написанную рекурсию.*

```
ghci> findKey "юля" phoneBook
Just "853-24-92"
ghci> findKey "оля" phoneBook
Just "555-29-38"
ghci> findKey "аня" phoneBook
Nothing
```

Отлично, работает! Если у нас есть телефонный номер девушки, мы просто (`Just`) получим номер; в противном случае не получим ничего (`Nothing`).

Модуль Data.Map

Мы только что реализовали функцию `lookup` из модуля `Data.List`. Если нам нужно значение, соответствующее ключу, понадобится обойти все элементы списка, пока мы его не найдём.



Модуль `Data.Map` предлагает ассоциативные списки, которые работают намного быстрее (поскольку они реализованы с помощью деревьев), а также множество дополнительных функций. Начиная с этого момента мы будем говорить, что работаем с отображениями вместо ассоциативных списков.

Так как модуль Data.Map экспортирует функции, конфликтующие с модулями Prelude и Data.List, мы будем импортировать их с помощью квалифицированного импорта.

```
import qualified Data.Map as Map
```

Поместите этот оператор в исходный код и загрузите его в GHCi. Мы будем преобразовывать ассоциативный список в отображение с помощью функции fromList из модуля Data.Map. Функция fromList принимает ассоциативный список (в форме списка) и возвращает отображение с теми же ассоциациями. Немного поиграем:

```
ghci> Map.fromList [(3, "туфли"),(4,"деревья"),(9,"пчёлы")]
fromList [(3, "туфли"),(4,"деревья"),(9,"пчёлы")]
ghci> Map.fromList [("эрик","форман"),("роберт","чейз"),("крис", "тауб")]
fromList [("крис","тауб"),("роберт","чейз"),("эрик","форман")]
```

Когда отображение из модуля Data.Map показывается в консоли, сначала выводится fromList, а затем ассоциативный список, представляющий отображение.

Если в исходном списке есть дубликаты ключей, они отбрасываются:

```
ghci> Map.fromList [("MS",1),("MS",2),("MS",3)]
fromList [("MS",3)]
```

Вот сигнатура функции fromList:

```
Map.fromList :: (Ord k) => [(k, v)] -> Map.Map k v
```

Она говорит, что функция принимает список пар со значениями типа k и v и возвращает отображение, которое отображает ключи типа k в значения типа v. Обратите внимание, что если мы реализуем ассоциативный список с помощью обычного списка, то значения ключей должны лишь уметь сравниваться (иметь экземпляр класса типов Eq); теперь же должна быть возможность их упорядочить (класс типов Ord). Это существенное ограничение модуля Data.Map. Упорядочиваемые ключи нужны ему для того, чтобы размещать данные более эффективно.

Теперь мы можем преобразовать наш исходный ассоциативный список phoneBook в отображение. Заодно добавим сигнатуру:

```
import qualified Data.Map as Map

phoneBook :: Map.Map String String
phoneBook = Map.fromList $
   [("оля","555-29-38")
   ,("женя","452-29-28")
   ,("катя","493-29-28")
   ,("маша","205-29-28")
   ,("надя","939-82-82")
   ,("юля","853-24-92")
   ]
```

Отлично. Загрузим этот сценарий в GHCi и немного поиграем с телефонной книжкой. Во-первых, воспользуемся функцией `lookup` и поищем какие-нибудь номера. Функция `lookup` принимает ключ и отображение и пытается найти соответствующее ключу значение. Если всё прошло удачно, возвращается обёрнутое в `Just` значение; в противном случае – `Nothing`:

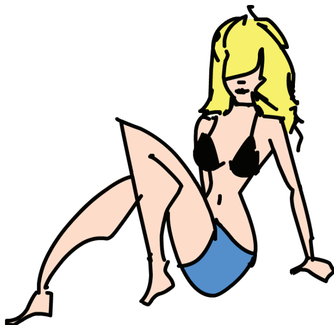
```
ghci> :t Map.lookup
Map.lookup :: (Ord k) => k -> Map.Map k a -> Maybe a
ghci> Map.lookup "оля" phoneBook
Just "555-29-38"
ghci> Map.lookup "надя" phoneBook
Just "939-82-82"
ghci> Map.lookup "таня" phoneBook
Nothing
```

Следующий трюк: создадим новое отображение, добавив в исходное новый номер. Функция `insert` принимает ключ, значение и отображение и возвращает новое отображение – почти такое же, что и исходное, но с добавленными ключом и значением:

```
ghci> :t Map.insert
Map.insert :: (Ord k) => k -> a -> Map.Map k a -> Map.Map k a
ghci> Map.lookup "таня" phoneBook
Nothing
ghci> let newBook = Map.insert "таня" "341-90-21" phoneBook
ghci> Map.lookup "таня" newBook
Just "341-90-21"
```

Давайте посчитаем, сколько у нас телефонных номеров. Для этого нам понадобится функция size из модуля Data.Map. Она принимает отображение и возвращает его размер. Тут всё ясно:

```
ghci> :t Map.size
Map.size :: Map.Map k a -> Int
ghci> Map.size phoneBook
6
ghci> Map.size newBook
7
```



Номера в нашей телефонной книжке представлены строками. Допустим, мы хотим вместо них использовать списки цифр: то есть вместо номера "939-82-82" – список [9,3,9,8,2,8,2]. Сначала напишем функцию, конвертирующую телефонный номер в строке в список целых. Можно попытаться применить функцию digitToInt из модуля Data.Char к каждому символу в строке, но она не знает, что делать с дефисом! Поэтому нужно избавиться от всех не-цифр. Попросим помощи у функции isDigit из модуля Data.Char, которая принимает символ и сообщает нам, является ли он цифрой. Как только строка будет отфильтрована, пройдёмся по ней функцией digitToInt.

```
string2digits :: String -> [Int]
string2digits = map digitToInt . filter isDigit
```

Да, не забудьте импортировать модуль Data.Char. Пробуем:

```
ghci> string2digits "948-92-82"
[9,4,8,9,2,8,2]
```

Замечательно! Теперь применим функцию map из модуля Data.Map, чтобы пропустить функцию string2digits по элементам отображения phoneBook:

```
ghci> let intBook = Map.map string2digits phoneBook
ghci> :t intBook
intBook :: Map.Map String [Int]
```

```
ghci> Map.lookup "оля" intBook
Just [5,5,5,2,9,3,8]
```

Функция map из модуля Data.Map принимает функцию и отображение и применяет эту функцию к каждому значению в отображении.

Расширим телефонную книжку. Предположим, что у кого-нибудь есть несколько телефонных номеров, и наш ассоциативный список выглядит как-то так:

```
phoneBook =
   [("оля","555-29-38")
   ,("оля","342-24-92")
   ,("женя","452-29-28")
   ,("катя","493-29-28")
   ,("катя","943-29-29")
   ,("катя","827-91-62")
   ,("маша","205-29-28")
   ,("надя","939-82-82")
   ,("юля","853-24-92")
   ,("юля","555-21-11")
   ]
```

Если мы просто вызовем fromList, чтобы поместить всё это в отображение, то потеряем массу номеров! Вместо этого воспользуемся другой функцией из модуля Data.Map, а именно функцией fromListWith. Эта функция действует почти как fromList, но вместо отбрасывания повторяющихся ключей вызывает переданную ей функцию, которая и решает, что делать.

```
phoneBookToMap :: (Ord k) => [(k, String)] -> Map.Map k String
phoneBookToMap xs = Map.fromListWith add xs
   where add number1 number2 = number1 ++ ", " ++ number2
```

Если функция fromListWith обнаруживает, что ключ уже существует, она вызывает переданную ей функцию, которая соединяет оба значения в одно, а затем заменяет старое значение на новое, полученное от соединяющей функции:

```
ghci> Map.lookup "катя" $ phoneBookToMap phoneBook
"827-91-62, 943-29-29, 493-29-28"
ghci> Map.lookup "надя" $ phoneBookToMap phoneBook
```

```
"939-82-82"
ghci> Map.lookup "оля" $ phoneBookToMap phoneBook
"342-24-92, 555-29-38"
```

А ещё можно было бы сделать все значения в ассоциативном списке одноэлементными списками, а потом скомбинировать их операцией ++, например:

```
phoneBookToMap :: (Ord k) => [(k, a)] -> Map.Map k [a]
phoneBookToMap xs = Map.fromListWith (++) $ map (\(k,v) -> (k, [v])) xs
```

Проверим в GHCi:

```
ghci> Map.lookup "катя" $ phoneBookToMap phoneBook
["827-91-62","943-29-29","493-29-28"]
```

Превосходно!

Ещё примеры. Допустим, мы делаем отображение из ассоциативного списка чисел и при обнаружении повторяющегося ключа хотим, чтобы сохранилось наибольшее значение. Это можно сделать так:

```
ghci> Map.fromListWith max [(2,3),(2,100),(3,29),(3,11),(4,22),(4,15)]
fromList [(2,100),(3,29),(4,22)]
```

Или хотим, чтобы значения с повторяющимися ключами складывались:

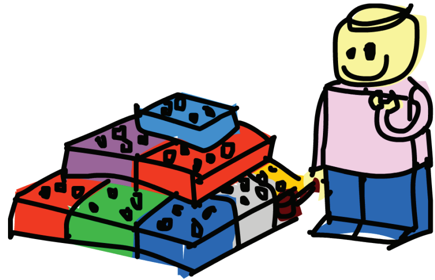
```
ghci> Map.fromListWith (+) [(2,3),(2,100),(3,29),(3,11),(4,22),(4,15)]
fromList [(2,103),(3,40),(4,37)]
```

Ну что ж, модуль `Data.Map`, да и другие модули из стандартной библиотеки языка Haskell довольно неплохи. Далее посмотрим, как написать свой собственный модуль.

Написание собственных модулей

Практически все языки программирования позволяют разделять код на несколько файлов, и Haskell – не исключение. При написании программ очень удобно помещать функции и типы, служащие схожим целям, в отдельный модуль. Таким образом, можно будет повторно использовать эти функции в других программах, просто импортировав нужный модуль.

Мы говорим, что модуль экспортирует функции. Это значит, что когда мы его импортируем, то можем использовать экспортируемые им функции. Модуль может определить функции для внутреннего использования, но извне модуля мы видим только те, которые он экспортирует.



Модуль Geometry

Давайте разберём процесс создания модулей на простом примере. Создадим модуль, который содержит функции для вычисления объёма и площади поверхности нескольких геометрических фигур. И начнём с создания файла *Geometry.hs.*

В начале модуля указывается его имя. Если мы назвали файл *Geometry.hs*, то имя нашего модуля должно быть `Geometry`. Затем следует перечислить экспортируемые функции, после чего мы можем писать сами функции:

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where
```

Как видите, мы будем вычислять площади и объёмы для сфер (`sphere`), кубов (`cube`) и прямоугольных параллелепипедов (`cuboid`). Сфера – это круглая штука наподобие грейпфрута, куб – квадратная штука, похожая на кубик Рубика, а прямоугольный параллелепипед – точь-в-точь пачка сигарет. (Дети, курить вредно!)

Продолжим и определим наши функции:

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
```

```
, cubeArea
, cuboidArea
, cuboidVolume
) where

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side

cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectArea a b * c

cuboidArea :: Float -> Float -> Float -> Float
cuboidArea a b c = rectArea a b * 2 + rectArea a c * 2 + rectArea c b * 2

rectArea :: Float -> Float -> Float
rectArea a b = a * b
```

Довольно стандартная геометрия, но есть несколько вещей, на которые стоит обратить внимание. Так как куб – это разновидность параллелепипеда, мы определили его площадь и объём, трактуя куб как параллелепипед с равными сторонами. Также мы определили вспомогательную функцию `rectArea`, которая вычисляет площадь прямоугольника по его сторонам. Функция очень проста – она просто перемножает стороны. Заметьте, мы используем функцию `rectArea` в функциях модуля (а именно в функциях `cuboidArea` и `cuboidVolume`), но не экспортируем её, так как хотим создать модуль для работы только с трёхмерными объектами.

При создании модуля мы обычно экспортируем только те функции, которые служат интерфейсом нашего модуля, и скрываем реализацию. Использующий наш модуль человек ничего не должен знать о тех функциях, которые мы не экспортируем. Мы можем полностью их поменять или удалить в следующей версии (скажем,

удалить определение функции `rectArea` и просто использовать умножение), и никто не будет против – в первую очередь потому, что эти функции не экспортируются.

Чтобы использовать наш модуль, запишем:

```
import Geometry
```

Файл *Geometry.hs* должен находиться в той же папке, что и импортирующая его программа.

Иерархия модулей

Модулям можно придать иерархическую структуру. Каждый модуль может иметь несколько подмодулей, которые в свою очередь также могут содержать подмодули. Давайте разделим наш модуль `Geometry` таким образом, чтобы в него входили три подмодуля, по одному на каждый тип объекта.

Сначала создадим папку с именем *Geometry*. В этой папке мы разместим три файла: *Sphere.hs*, *Cuboid.hs* и *Cube.hs*. Посмотрим, что должно находиться в каждом файле.

Вот содержимое файла *Sphere.hs*:

```
module Geometry.Sphere
( volume
, area
) where

volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)

area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

Файл *Cuboid.hs* выглядит так:

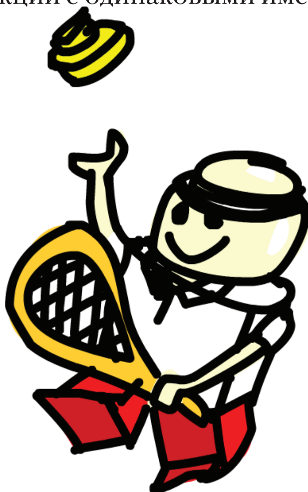
```
module Geometry.Cuboid
( volume
, area
) where

volume :: Float -> Float -> Float -> Float
volume a b c = rectArea a b * c
```

```
area :: Float -> Float -> Float -> Float
area a b c = rectArea a b * 2 + rectArea a c * 2 + rectArea c b * 2

rectArea :: Float -> Float -> Float
rectArea a b = a * b
```

А вот и содержимое файла *Cube.hs*:

```
module Geometry.Cube
( volume
, area
) where

import qualified Geometry.Cuboid as Cuboid

volume :: Float -> Float
volume side = Cuboid.volume side side side

area :: Float -> Float
area side = Cuboid.area side side side
```

Обратите внимание, что мы поместили файл *Sphere.hs* в папку с именем *Geometry* и определили имя модуля как `Geometry.Sphere`. То же самое мы сделали для куба и параллелепипеда. Также отметьте, что во всех трёх модулях определены функции с одинаковыми именами. Мы вправе так поступать, потому что функции находятся в разных модулях.



Итак, если мы редактируем файл, который находится на одном уровне с папкой *Geometry*, то запишем:

```
import Geometry.Sphere
```

после чего сможем вызывать функции `area` и `volume`, которые вычислят площадь и объём сферы. Если нам потребуется использовать несколько наших модулей, мы должны выполнить квалифицированный импорт, потому что они экспортируют функции с одинаковыми

именами. Делаем так:

```
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```

Затем мы сможем вызывать функции `Sphere.area`, `Sphere.volume`, `Cuboid.area` и т. д., и каждая функция вычислит площадь или объём соответствующего объекта.

В следующий раз, когда вы поймаете себя за написанием огромного файла с кучей функций, попытайтесь выяснить, какие функции служат некоей общей цели, и можно ли включить их в отдельный модуль.

Позднее при написании программы со схожей функциональностью вы сможете просто импортировать свой модуль.

7

СОЗДАНИЕ НОВЫХ ТИПОВ И КЛАССОВ ТИПОВ

В предыдущих главах мы изучили некоторые типы и классы типов в языке Haskell. Из этой главы вы узнаете, как создать и заставить работать свои собственные!

Введение в алгебраические типы данных

До сих пор мы сталкивались со многими типами данных – `Bool`, `Int`, `Char`, `Maybe` и др. Но как создать свой собственный тип? Один из способов – использовать ключевое слово `data`. Давайте посмотрим, как в стандартной библиотеке определён тип `Bool`:

```
data Bool = False | True
```

Ключевое слово `data` объявляет новый тип данных. Часть до знака равенства вводит идентификатор типа, в данном случае `Bool`. Часть после знака равенства – это конструкторы данных, которые также называют конструкторами значений. Они определяют, какие значения может принимать тип. Символ `|` означает «или». Объявление можно прочесть так: тип

Bool может принимать значения True или False. И имя типа, и конструкторы данных должны начинаться с прописной буквы.

Рассуждая подобным образом, мы можем думать, что тип Int объявлен так:

```
data Int = -2147483648 | -2147483647 | ... | -1 | 0 | 1 | 2 | ... | 2147483647
```

Первое и последнее значения – минимальное и максимальное для Int. На самом деле тип Int объявлен иначе – видите, я пропустил уйму чисел – такая запись полезна лишь в иллюстративных целях.

Отличная фигура за 15 минут

Теперь подумаем, как бы мы представили некую геометрическую фигуру в языке Haskell. Один из способов – использовать кортежи. Круг может быть представлен как (43.1, 55.0, 10.4), где первое и второе поле – координаты центра, а третье – радиус. Вроде бы подходит, но такой же кортеж может представлять вектор в трёхмерном пространстве или что-нибудь ещё. Лучше было бы определить свой собственный тип для фигуры. Скажем, наша фигура может быть кругом или прямоугольником.

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

Ну и что это? Размышляйте следующим образом. Конструктор для значения Circle содержит три поля типа Float. Когда мы записываем конструктор значения типа, опционально мы можем добавлять типы после имени конструктора; эти типы определяют, какие значения будет содержать тип с данным конструктором. В нашем случае первые два числа – это координаты центра, третье число – радиус. Конструктор для значения Rectangle имеет четыре поля, которые также являются числами с плавающей точкой. Первые два числа – это координаты верхнего левого угла, вторые два числа – координаты нижнего правого угла.

Когда я говорю «поля», то подразумеваю «параметры». Конструкторы данных на самом деле являются функциями, только эти функции возвращают значения типа данных. Давайте посмотрим на сигнатуры для наших двух конструкторов:

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
```

```
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

Классно, конструкторы значений – такие же функции, как любые другие! Кто бы мог подумать!..

Давайте напишем функцию, которая принимает фигуру и возвращает площадь её поверхности:

```
area :: Shape -> Float
area (Circle _ _ r) = pi * r ^ 2
area (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

Первая примечательная вещь в объявлении – это декларация типа. Она говорит, что функция принимает фигуру и возвращает значение типа `Float`. Мы не смогли бы записать функцию типа `Circle -> Float`, потому что идентификатор `Circle` не является типом; типом является идентификатор `Shape`. По той же самой причине мы не смогли бы написать функцию с типом `True -> Int`. Вторая примечательная вещь – мы можем выполнять сопоставление с образцом по конструкторам. Мы уже записывали подобные сопоставления раньше (притом очень часто), когда сопоставляли со значениями `[]`, `False`, `5`, только эти значения не имели полей. Только что мы записали конструктор и связали его поля с именами. Так как для вычисления площади нам нужен только радиус, мы не заботимся о двух первых полях, которые говорят нам, где располагается круг.

```
ghci> area $ Circle 10 20 10
314.15927
ghci> area $ Rectangle 0 0 100 100
10000.0
```

Ура, работает! Но если попытаться напечатать `Circle 10 20 5` в командной строке интерпретатора, то мы получим ошибку. Пока Haskell не знает, как отобразить наш тип данных в виде строки. Вспомним, что когда мы пытаемся напечатать значение в командной строке, интерпретатор языка Haskell вызывает функцию `show`, для того чтобы получить строковое представление значения, и затем печатает результат в терминале. Чтобы определить для нашего типа `Shape` экземпляр класса `Show`, модифицируем его таким образом:

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
     deriving (Show)
```

Не будем пока концентрировать внимание на конструкции `deriving (Show)`. Просто скажем, что если мы добавим её в конец объявления типа данных, Haskell автоматически определит экземпляр класса `Show` для этого типа. Теперь можно делать так:

```
ghci> Circle 10 20 5
Circle 10.0 20.0 5.0
ghci> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0
```

Конструкторы значений – это функции, а значит, мы можем их отображать, частично применять и т. д. Если нам нужен список концентрических кругов с различными радиусами, напишем следующий код:

```
ghci> map (Circle 10 20) [4,5,6,6]
[Circle 10.0 20.0 4.0,Circle 10.0 20.0 5.0,Circle 10.0 20.0 6.0,Circle 10.0
20.0 6.0]
```

Верный способ улучшить фигуру

Наш тип данных хорош, но может быть и ещё лучше. Давайте создадим вспомогательный тип данных, который определяет точку в двумерном пространстве. Затем используем его для того, чтобы сделать наши фигуры более понятными:

```
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float  | Rectangle Point Point deriving (Show)
```

Обратите внимание, что при определении точки мы использовали одинаковые имена для конструктора типа и для конструктора данных. В этом нет какого-то особого смысла, но если у типа данных только один конструктор, как правило, он носит то же имя, что и тип. Итак, теперь у конструктора `Circle` два поля: первое имеет тип `Point`, второе – `Float`. Так легче разобраться, что есть что. То же верно и для прямоугольника. Теперь, после всех изменений, мы должны исправить функцию `area`:

```
area :: Shape -> Float
area (Circle _ r) = pi * r    2
area (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

Единственное, что мы должны поменять, – это образцы. Мы игнорируем точку у образца для круга. В образце для прямоугольника используем вложенные образцы при сопоставлении для того, чтобы получить все поля точек. Если бы нам нужны были точки целиком, мы бы использовали именованные образцы. Проверим улучшенную версию:

```
ghci> area (Rectangle (Point 0 0) (Point 100 100))
10000.0
ghci> area (Circle (Point 0 0) 24)
1809.5574
```

Как насчёт функции, которая двигает фигуру? Она принимает фигуру, приращение координаты по оси абсцисс, приращение координаты по оси ординат – и возвращает новую фигуру, которая имеет те же размеры, но располагается в другом месте.

```
nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b
     = Rectangle (Point (x1+a) (y1+b)) (Point (x2+a) (y2+b))
```

Всё довольно очевидно. Мы добавляем смещение к точкам, определяющим положение фигуры:

```
ghci> nudge (Circle (Point 34 34) 10) 5 10
Circle (Point 39.0 44.0) 10.0
```

Если мы не хотим иметь дело напрямую с точками, то можем сделать вспомогательные функции, которые создают фигуры некоторого размера с нулевыми координатами, а затем их подвигать.

Во-первых, напишем функцию, принимающую радиус и создающую круг с указанным радиусом, расположенный в начале координат:

```
baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r
```

Добавим функцию, которая по заданным ширине и высоте создаёт прямоугольник соответствующего размера. При этом левый нижний угол прямоугольника находится в начале координат:

```
baseRect :: Float -> Float -> Shape
baseRect width height = Rectangle (Point 0 0) (Point width height)
```

Теперь создавать формы гораздо легче: достаточно создать форму в начале координат, а затем сдвинуть её в нужное место:

```
ghci> nudge (baseRect 40 100) 60 23
Rectangle (Point 60.0 23.0) (Point 100.0 123.0)
```

Фигуры на экспорт

Конечно же, вы можете экспортировать типы данных из модулей. Чтобы сделать это, запишите имена ваших типов вместе с именами экспортируемых функций. В отдельных скобках, через запятую, укажите, какие конструкторы значений вы хотели бы экспортировать. Если хотите экспортировать все конструкторы значений, просто напишите две точки (`..`).

Если бы мы хотели поместить функции и типы, определённые выше, в модуль, то могли бы начать как-то так:

```
module Shapes
( Point(..)
, Shape(..)
, area
, nudge
, baseCircle
, baseRect
) where
```

Запись `Shape(..)` обозначает, что мы экспортируем все конструкторы данных для типа `Shape`. Тот, кто импортирует наш модуль, сможет создавать фигуры, используя конструкторы `Rectangle` и `Circle`. Это то же самое, что и `Shape (Rectangle, Circle)`, но короче.

К тому же, если мы позже решим дописать несколько конструкторов данных, перечень экспортируемых объектов исправлять не придётся. Всё потому, что конструкция `..` автоматически экспортирует все конструкторы соответствующего типа.

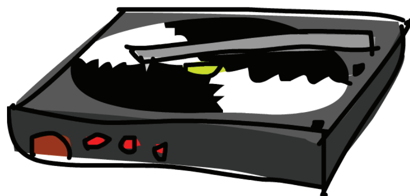
Мы могли бы не указывать ни одного конструктора для типа `Shape`, просто записав `Shape` в операторе экспорта. В таком случае тот, кто импортирует модуль, сможет создавать фигуры только с помощью функций `baseCircle` и `baseRect`.

Помните, конструкторы данных – это простые функции, принимающие поля как параметры и возвращающие значение некоторого типа (например, `Shape`) как результат. Если мы их не экспортируем, то вне модуля они будут недоступны. Отказ от экспорта конструкторов данных делает наши типы данных более абстрактными, поскольку мы скрываем их реализацию. К тому же, пользователи нашего модуля не смогут выполнять сопоставление с образцом для этих конструкторов данных. Это полезно, если мы хотим, чтобы программисты, импортирующие наш тип, работали только со вспомогательными функциями, которые мы специально для этого написали. Таким образом, у них нет необходимости знать о деталях реализации модуля, и мы можем изменить эти детали, когда захотим – лишь бы экспортируемые функции работали как прежде.

Модуль `Data.Map` использует такой подход. Вы не можете создать отображение напрямую при помощи соответствующего конструктора данных, потому что такой конструктор не экспортирован. Однако можно создавать отображения, вызвав одну из вспомогательных функций, например `Map.fromList`. Разработчики, ответственные за `Data.Map`, в любой момент могут поменять внутреннее представление отображений, и при этом ни одна существующая программа не сломается.

Разумеется, экспорт конструкторов данных для типов попроще вполне допустим.

Синтаксис записи с именованными полями



Есть ещё один способ определить тип данных. Предположим, что перед нами поставлена задача создать тип данных для описания человека. Данные, которые мы намереваемся хра-

нить, – имя, фамилия, возраст, рост, телефон и любимый сорт мороженого. (Не знаю, как насчёт вас, но это всё, что я хотел бы знать о человеке!) Давайте опишем такой тип:

```
data Person = Person String String Int Float String String deriving (Show)
```

Первое поле – это имя, второе – фамилия, третье – возраст и т. д. И вот наш персонаж:

```
gghci> let guy = Person "Фредди" "Крюгер" 43 184.2 "526-2928" "Эскимо"
ghci> guy
Person "Фредди" "Крюгер" 43 184.2 "526-2928" "Эскимо"
```

Ну, в целом приемлемо, хоть и не очень «читабельно». Что если нам нужна функция для получения какого-либо поля? Функция, которая возвращает имя, функция для фамилии и т. д.? Мы можем определить их таким образом:

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

age :: Person -> Int
age (Person _ _ age _ _ _) = age

height :: Person -> Float
height (Person _ _ _ height _ _) = height

phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number

flavor :: Person -> String
flavor (Person _ _ _ _ _ flavor) = flavor
```

Фу-ух! Мало радости писать такие функции!.. Этот метод очень громоздкий и скучный, но он работает.

```
ghci> let guy = Person "Фредди" "Крюгер" 43 184.2 "526-2928" "Эскимо"
ghci> firstName guy
"Фредди"
ghci> height guy
```

```
184.2
ghci> flavor guy
"Эскимо"
```

Вы скажете – должен быть лучший способ! Ан нет, извиняйте, нету... Шучу, конечно же. Такой метод есть! «Ха-ха» два раза. Создатели языка Haskell предусмотрели подобную возможность – предоставили ещё один способ для записи типов данных. Вот как мы можем достигнуть той же функциональности с помощью синтаксиса записей с именованными полями:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String } deriving (Show)
```

Вместо того чтобы просто перечислять типы полей через запятую, мы используем фигурные скобки. Вначале пишем имя поля, например `firstName`, затем ставим два двоеточия `::` и, наконец, указываем тип. Результирующий тип данных в точности такой же. Главная выгода – такой синтаксис генерирует функции для извлечения полей. Язык Haskell автоматически создаст функции `firstName`, `lastName`, `age`, `height`, `phoneNumber` и `flavor`.

```
ghci> :t flavor
flavor :: Person -> String
ghci> :t firstName
firstName :: Person -> String
```

Есть ещё одно преимущество в использовании синтаксиса записей. Когда мы автоматически генерируем экземпляр класса `Show` для типа, он отображает тип не так, как если бы мы использовали синтаксис записей с именованными полями для объявления и инстанцирования типа. Например, у нас есть тип, представляющий автомобиль. Мы хотим хранить следующую информацию: компания-производитель, название модели и год производства.

```
data Car = Car String String Int deriving (Show)
```

Автомобиль отображается так:

```
ghci> Car "Форд" "Мустанг" 1967
Car "Форд" "Мустанг" 1967
```

Используя синтаксис записей с именованными полями, мы можем описать новый автомобиль так:

```
data Car = Car { company :: String
               , model :: String
               , year :: Int
               } deriving (Show)
```

Автомобиль теперь создаётся и отображается следующим образом:

```
ghci> Car {company="Форд", model="Мустанг", year=1967}
Car {company = "Форд", model = "Мустанг", year = 1967}
```

При создании нового автомобиля мы, разумеется, обязаны перечислить все поля, но указывать их можно в любом порядке. Но если мы не используем синтаксис записей с именованными полями, то должны указывать их по порядку.

Используйте синтаксис записей с именованными полями, если конструктор имеет несколько полей и не очевидно, какое поле для чего используется. Если, скажем, мы создаём трёхмерный вектор: `data Vector = Vector Int Int Int`, то вполне понятно, что поля конструктора данных – это компоненты вектора. Но в типах `Person` и `Car` назначение полей совсем не так очевидно, и мы значительно выиграем, используя синтаксис записей с именованными полями.

Параметры типа

Конструктор данных может принимать несколько параметров-значений и возвращать новое значение. Например, конструктор `Car` принимает три значения и возвращает одно – экземпляр типа `Car`. Таким же образом конструкторы типа могут принимать типы-параметры и создавать новые типы. На первый взгляд это несколько абстрактно, но на самом деле не так уж сложно. Если вы знакомы с шаблонами в языке C++, то увидите некоторые параллели. Чтобы получить более ясное представление о том, как работают типы-параметры, давайте посмотрим, как реализованы типы, с которыми мы уже встречались.

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

В данном примере идентификатор a – тип-параметр (переменная типа, типовая переменная). Так как в выражении присутствует тип-параметр, мы называем идентификатор Maybe конструктором типов. В зависимости от того, какой тип данных мы хотим сохранять в типе Maybe, когда он не Nothing, конструктор типа может производить такие типы, как Maybe Int, Maybe Car, Maybe String и т. д. Ни одно значение не может иметь тип «просто Maybe», потому что это не тип как таковой – это конструктор типов. Для того чтобы он стал настоящим типом, значения которого можно создать, мы должны указать все типы-параметры в конструкторе типа.

Итак, если мы передадим тип Char как параметр в тип Maybe, то получим тип Maybe Char. Для примера: значение Just 'a' имеет тип Maybe Char.

Обычно нам не приходится явно передавать параметры конструкторам типов, поскольку в языке Haskell есть вывод типов. Поэтому когда мы создаём значение Just 'a', Haskell тут же определяет его тип – Maybe Char.

Если мы всё же хотим явно указать тип как параметр, это нужно делать в типовой части выражений, то есть после символа ::. Явное указание типа может понадобиться, если мы, к примеру, хотим, чтобы значение Just 3 имело тип Maybe Int. По умолчанию Haskell выведет тип (Num a) => Maybe a. Воспользуемся явным аннотированием типа:

```
ghci> Just 3 :: Maybe Int
Just 3
```

Может, вы и не знали, но мы использовали тип, у которого были типы-параметры ещё до типа Maybe. Этот тип – список. Несмотря на

то что дело несколько скрывается синтаксическим сахаром, конструктор списка принимает параметр для того, чтобы создать конкретный тип. Значения могут иметь тип [Int], [Char], [[String]], но вы не можете создать значение с типом [].

ПРИМЕЧАНИЕ. *Мы называем тип конкретным, если он вообще не принимает никаких параметров (например, Int или Bool) либо если параметры в типе заполнены (например, Maybe Char). Если у вас есть какое-то значение, у него всегда конкретный тип.*

Давайте поиграем с типом Maybe:

```
ghci> Just "Ха-ха"
Just "Ха-ха"
ghci> Just 84
Just 84
ghci> :t Just "Ха-ха"
Just "Ха-ха" :: Maybe [Char]
ghci> :t Just 84
Just 84 :: (Num t) => Maybe t
ghci> :t Nothing
Nothing :: Maybe a
ghci> Just 10 :: Maybe Double
Just 10.0
```

Типы-параметры полезны потому, что мы можем с их помощью создавать различные типы, в зависимости от того, какой тип нам надо хранить в нашем типе данных. К примеру, можно объявить отдельные Maybe-подобные типы данных для любых типов:

```
data IntMaybe = INothing | IJust Int

data StringMaybe = SNothing | SJust String

data ShapeMaybe = ShNothing | ShJust Shape
```

Более того, мы можем использовать типы-параметры для определения самого обобщённого Maybe, который может содержать данные вообще любых типов!

Обратите внимание: тип значения Nothing – Maybe a. Это полиморфный тип: в его имени присутствует типовая переменная – конкретнее, переменная a в типе Maybe a. Если некоторая функция принимает параметр типа Maybe Int, мы можем передать ей значение

Nothing, так как оно не содержит значения, которое могло бы этому препятствовать. Тип Maybe a может вести себя как Maybe Int, точно так же как значение 5 может рассматриваться как значение типа Int или Double. Аналогичным образом тип пустого списка – это [a]. Пустой список может вести себя как список чего угодно. Вот почему можно производить такие операции, как [1,2,3] ++ [] и ["ха","ха","ха"] ++ [].

Параметризовать ли машины?

Когда имеет смысл применять типовые параметры? Обычно мы используем их, когда наш тип данных должен уметь сохранять внутри себя любой другой тип, как это делает Maybe a. Если ваш тип – это некоторая «обёртка», использование типов-параметров оправданно. Мы могли бы изменить наш тип данных Car с такого:

```
data Car = Car { company :: String
               , model :: String
               , year :: Int
               } deriving (Show)
```

на такой:

```
data Car a b c = Car { company :: a
                     , model :: b
                     , year :: c
                     } deriving (Show)
```

Но выиграем ли мы в чём-нибудь? Ответ – вероятно, нет, потому что впоследствии мы всё равно определим функции, которые работают с типом Car String String Int. Например, используя первое определение Car, мы могли бы создать функцию, которая отображает свойства автомобиля в виде понятного текста:

```
tellCar :: Car -> String
tellCar (Car {company = c, model = m, year = y}) =
   "Автомобиль " ++ c ++ " " ++ m ++ ", год: " ++ show y
```

```
ghci> let stang = Car {company="Форд", model="Мустанг", year=1967}
ghci> tellCar stang
"Автомобиль Форд Мустанг, год: 1967"
```

Приятная маленькая функция. Декларация типа функции красива и понятна. А что если Car – это Car a b c?

```
tellCar :: (Show a) => Car String String a -> String
tellCar (Car {company = c, model = m, year = y}) =
   "Автомобиль " ++ c ++ " " ++ m ++ ", год: " ++ show y
```

Мы вынуждены заставить функцию принимать параметр `Car` типа `(Show a) => Car String String a`. Как видите, декларация типа функции более сложна; единственное преимущество, которое здесь имеется, – мы можем использовать любой тип, имеющий экземпляр класса `Show`, как тип для типовой переменной `c`.

```
ghci> tellCar (Car "Форд" "Мустанг" 1967)
"Автомобиль Форд Мустанг, год: 1967"
ghci> tellCar (Car "Форд" "Мустанг" "тысяча девятьсот шестьдесят седьмой")
"Автомобиль Форд Мустанг, год: \"тысяча девятьсот шестьдесят седьмой\""
ghci> :t Car "Форд" "Мустанг" 1967
Car "Форд" "Мустанг" 1967 :: (Num t) => Car [Char] [Char] t
ghci> :t Car "Форд" "Мустанг" "тысяча девятьсот шестьдесят седьмой"
Car "Форд" "Мустанг" "тысяча девятьсот шестьдесят седьмой"
           :: Car [Char] [Char] [Char]
```

На практике мы всё равно в большинстве случаев использовали бы `Car String String Int`, так что в параметризации типа `Car` большого смысла нет. Обычно мы параметризируем типы, когда для работы нашего типа неважно, что в нём хранится. Список элементов – это просто список элементов, и неважно, какого они типа: список работает вне зависимости от этого. Если мы хотим суммировать список чисел, то в суммирующей функции можем уточнить, что нам нужен именно список чисел. То же самое верно и для типа `Maybe`. Он предоставляет возможность не иметь никакого значения или иметь какое-то одно значение. Тип хранимого значения не важен.

Ещё один известный нам пример параметризованного типа – отображения `Map k v` из модуля `Data.Map`. Параметр `k` – это тип ключей в отображении, параметр `v` – тип значений. Это отличный пример правильного использования параметризации типов. Параметризация отображений позволяет нам использовать любые типы, требуя лишь, чтобы тип ключа имел экземпляр класса `Ord`. Если бы мы определяли тип для отображений, то могли бы добавить ограничение на класс типа в объявлении:

```
data (Ord k) => Map k v = ...
```

Тем не менее в языке Haskell принято соглашение никогда не использовать ограничения класса типов при объявлении типов данных. Почему? Потому что серьёзных преимуществ мы не получим, но в конце концов будем использовать всё больше ограничений, даже если они не нужны. Поместим ли мы ограничение `(Ord k)` в декларацию типа или не поместим – всё равно придётся указывать его при объявлении функций, предполагающих, что ключ может быть упорядочен. Но если мы не поместим ограничение в объявлении типа, нам не придётся писать его в тех функциях, которым неважно, может ключ быть упорядочен или нет. Пример такой функции – `toList :: Map k a -> [(k, a)]`. Если бы `Map k a` имел ограничение типа в объявлении, тип для функции `toList` был бы таким: `toList :: (Ord k) => Map k a -> [(k, a)]`, даже несмотря на то что функция не сравнивает элементы друг с другом.

Так что не помещайте ограничения типов в декларации типов данных, даже если это имело бы смысл, потому что вам всё равно придётся помещать ограничения в декларации типов функций.

Векторы судьбы

Давайте реализуем трёхмерный вектор и несколько операций для него. Мы будем использовать параметризованный тип, потому что хоть вектор и содержит только числовые параметры, он должен поддерживать разные типы чисел.

```
data Vector a = Vector a a a deriving (Show)

vplus :: (Num a) => Vector a -> Vector a -> Vector a
(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)

scalarProd :: (Num a) => Vector a -> Vector a -> a
(Vector i j k) `scalarProd` (Vector l m n) = i*l + j*m + k*n

vmult :: (Num a) => Vector a -> a -> Vector a
(Vector i j k) `vmult` m = Vector (i*m) (j*m) (k*m)
```

Функция `vplus` складывает два вектора путём сложения соответствующих координат. Функция `scalarProd` используется для вычисления скалярного произведения двух векторов, функция `vmult` – для умножения вектора на константу.

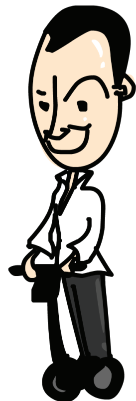
Эти функции могут работать с типами `Vector Int`, `Vector Integer`, `Vector Float` и другими, до тех пор пока тип-параметр `a` из опреде-

ления `Vector a` принадлежит классу типов `Num`. По типам функций можно заметить, что они работают только с векторами одного типа, и все координаты вектора также должны иметь одинаковый тип. Обратите внимание на то, что мы не поместили ограничение класса `Num` в декларацию типа данных, так как нам всё равно бы пришлось повторять его в функциях.

Ещё раз повторю: очень важно понимать разницу между конструкторами типов и данных. При декларации типа данных часть объявления до знака `=` представляет собой конструктор типа, а часть объявления после этого знака – конструктор данных (возможны несколько конструкторов, разделенных символом `|`). Попытка дать функции тип `Vector a a a -> Vector a a a -> a` будет неудачной, потому что мы должны помещать типы в декларацию типа, и конструктор типа для вектора принимает только один параметр, в то время как конструктор данных принимает три. Давайте поупражняемся с нашими векторами:

```
ghci> Vector 3 5 8 `vplus` Vector 9 2 8
Vector 12 7 16
ghci> Vector 3 5 8 `vplus` Vector 9 2 8 `vplus` Vector 0 2 3
Vector 12 9 19
ghci> Vector 3 9 7 `vmult` 10
Vector 30 90 70
ghci> Vector 4 9 5 `scalarProd` Vector 9.0 2.0 4.0
74.0
ghci> Vector 2 9 3 `vmult` (Vector 4 9 5 `scalarProd` Vector 9 2 4)
Vector 148 666 222
```

Производные экземпляры



В разделе «Классы типов» главы 2 приводились базовые сведения о классах типов. Мы упомянули, что класс типов – это нечто вроде интерфейса, который определяет некоторое поведение. Тип может быть сделан экземпляром класса, если поддерживает это поведение. Пример: тип `Int` есть экземпляр класса типов `Eq`, потому что класс `Eq` определяет поведение для сущностей, которые могут быть проверены на равенство. Так как целые числа можно проверить на равенство,

тип Int имеет экземпляр для класса Eq. Реальная польза от этого видна при использовании функций, которые служат интерфейсом класса Eq, – операторов == и /=. Если тип имеет определённый экземпляр класса Eq, мы можем применять оператор == к значениям этого типа. Вот почему выражения 4 == 4 и "раз" /= "два" проходят проверку типов.

Классы типов часто путают с классами в языках вроде Java, Python, C++ и им подобных, что сбивает с толку множество людей. В вышеперечисленных языках классы – это нечто вроде чертежей, по которым потом создаются объекты, хранящие некое состояние и способные производить некие действия. Мы не создаём типы из классов типов – вместо этого мы сначала создаём свои типы данных, а затем думаем о том, как они могут себя вести. Если то, что мы создали, можно проверить на равенство, – определяем для него экземпляр класса Eq. Если наш тип может вести себя как нечто, что можно упорядочить, – создаём для него экземпляр класса Ord.

Давайте посмотрим, как язык Haskell умеет автоматически делать наши типы экземплярами таких классов типов, как Eq, Ord, Enum, Bounded, Show и Read. Haskell умеет порождать поведение для наших типов в этих контекстах, если мы используем ключевое слово deriving при создании типа данных.

Сравнение людей на равенство

Рассмотрим такой тип данных:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     }
```

Тип описывает человека. Предположим, что среди людей не встречаются тёзки одного возраста. Если у нас есть два описания, можем ли мы выяснить, относятся ли они к одному и тому же человеку? Есть ли в такой операции смысл? Конечно, есть. Мы можем сравнить записи и проверить, равны они или нет. Вот почему имело бы смысл определить для нашего типа экземпляр класса Eq. Порождаем экземпляр:

```
data Person = Person { firstName :: String
                     , lastName :: String
```

```
                      , age :: Int
                      } deriving (Eq)
```

Когда мы определяем экземпляр класса Eq для типа и пытаемся сравнить два значения с помощью операторов == или /=, язык Haskell проверяет, совпадают ли конструкторы значений (хотя в нашем типе только один конструктор), а затем проверяет все данные внутри конструктора на равенство, сравнивая каждую пару полей с помощью оператора ==. Таким образом, типы всех полей также должны иметь определённый экземпляр класса Eq. Так как типы полей нашего типа, String и Int, имеют экземпляры класса Eq, всё в порядке.

Запишем в файл несколько людей:

```
mikeD = Person {firstName = "Майкл", lastName = "Даймонд", age = 45}
adRock = Person {firstName = "Адам", lastName = "Горовиц", age = 45}
mca = Person {firstName = "Адам", lastName = "Яух", age = 47}
```

И проверим экземпляр класса Eq:

```
ghci> mca == adRock
False
ghci> mikeD == adRock
False
ghci> mikeD == mikeD
True
ghci> mca == Person {firstName = "Адам", lastName = "Яух", age = 47}
True
```

Конечно же, так как теперь тип Person имеет экземпляр класса Eq, мы можем передавать его любым функциям, которые содержат ограничение на класс типа Eq в декларации, например функции elem.

```
ghci> let beastieBoys = [mca, adRock, mikeD]
ghci> mikeD `elem` beastieBoys
True
```

Покажи мне, как читать

Классы типов Show и Read предназначены для сущностей, которые могут быть преобразованы в строки и из строк соответственно. Как

и для класса Eq, все типы в конструкторе типов также должны иметь экземпляры для классов Show и/или Read, если мы хотим получить такое поведение. Давайте сделаем наш тип данных Person частью классов Show и Read:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq, Show, Read)
```

Теперь мы можем распечатать запись на экране:

```
ghci> mikeD
Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> "mikeD is: " ++ show mikeD
"mikeD is: Person {firstName = \"Michael\", lastName = \"Diamond\", age = 43}"
```

Если бы мы попытались распечатать запись до того, как предусмотрели для типа Person экземпляры класса Show, язык Haskell пожаловался бы на то, что он не знает, как представить запись в виде строки. Но после того как мы определили экземпляр класса Show, всё проясняется.

Класс Read в чём-то является обратным классом типов для класса Show. Класс Show служит для преобразования значений нашего типа в строку, класс Read нужен для преобразования строк в значения типа. Запомните, что при использовании функции чтения мы должны явно аннотировать тип возвращаемого значения. Если не указать тип результата явно, язык Haskell не сможет угадать, какой тип мы желали бы получить. Чтобы это проиллюстрировать, поместим в файл строку, представляющую некоторого человека, а затем загрузим файл в GHCi:

```
mysteryDude = "Person { firstName =\"Майкл\"" ++
                     ", lastName =\"Даймонд\"" ++
                     ", age = 45}"
```

Для большей «читабельности» мы разбили строку на несколько фрагментов. Если теперь необходимо вызвать с этой строкой фунцию read, то потребуется указать тип, который мы ожидаем получить:

```
ghci> read mysteryDude :: Person
```

```
Person {firstName = "Майкл", lastName = "Даймонд", age = 45}
```

Если далее в программе мы используем результат чтения таким образом, что язык Haskell сможет вывести его тип, мы не обязаны использовать аннотацию типа.

```
ghci> read mysteryDude == mikeD
True
```

Так же можно считывать и параметризованные типы, но при этом следует явно указывать все типы-параметры.

Если мы попробуем сделать так:

```
ghci> read "Just 3" :: Maybe a
```

то получим сообщение об ошибке: Haskell не в состоянии определить конкретный тип, который следует подставить на место типовой переменной `a`. Если же мы точно укажем, что хотим получить `Int`, то всё будет прекрасно:

```
ghci> read "Just 3" :: Maybe Int
Just 3
```

Порядок в суде!

Класс типов `Ord`, предназначенный для типов, значения которых могут быть упорядочены, также допускает автоматическое порождение экземпляров. Если сравниваются два значения одного типа, сконструированные с помощью различных конструкторов данных, то меньшим считается значение, конструктор которого определён раньше. Рассмотрим, к примеру, тип `Bool`, значениями которого могут быть `False` или `True`. Для наших целей удобно предположить, что он определён следующим образом:

```
data Bool = False | True deriving (Ord)
```

Поскольку конструктор `False` указан первым, а конструктор `True` – после него, мы можем считать, что `True` больше, чем `False`.

```
ghci> True `compare` False
GT
ghci> True > False
```

```
True
ghci> True < False
False
```

Если два значения имеют одинаковый конструктор, то при отсутствии полей они считаются равными. Если поля есть, то выполняется их сравнение. Заметьте, что в этом случае типы полей должны быть частью класса типов `Ord`.

В типе данных `Maybe a` конструктор значений `Nothing` указан раньше `Just` – это значит, что значение `Nothing` всегда меньше, чем `Just` <*нечто*>, даже если это «нечто» равно минус одному миллиону триллионов. Но если мы сравниваем два значения `Just`, после сравнения конструкторов начинают сравниваться поля внутри них.

```
ghci> Nothing < Just 100
True
ghci> Nothing > Just (-49999)
False
ghci> Just 3 `compare` Just 2
GT
ghci>Just 100 > Just 50
True
```

Но сделать что-нибудь вроде `Just (*3) > Just (*2)` не получится, потому что `(*3)` и `(*2)` – это функции, а они не имеют экземпляров для класса `Ord`.

Любой день недели

Мы легко можем использовать алгебраические типы данных для того, чтобы создавать перечисления, и классы типов `Enum` и `Bounded` помогают нам в этом. Рассмотрим следующий тип:

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

Так как все конструкторы значений нульарные (не принимают параметров, то есть не имеют полей), допустимо сделать для нашего типа экземпляр класса `Enum`. Класс типов `Enum` предназначен для типов, для значений которых можно определить предшествующие и последующие элементы. Также мы можем определить для него экземпляр класса `Bounded` – он предназначен для типов, у которых есть минимальное и максимальное значения. Ну и уж заодно давай-

те сделаем для него экземпляры всех остальных классов типов, которые можно сгенерировать автоматически, и посмотрим, что это даст.

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
          deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

Так как для нашего типа автоматически сгенерированы экземпляры классов `Show` и `Read`, можно конвертировать значения типа в строки и из строк:

```
ghci> Wednesday
Wednesday
ghci> show Wednesday
"Wednesday"
ghci> read "Saturday" :: Day
Saturday
```

Поскольку он имеет экземпляры классов `Eq` и `Ord`, допускаются сравнение и проверка на равенство:

```
ghci> Saturday == Sunday
False
ghci> Saturday == Saturday
True
ghci> Saturday > Friday
True
ghci> Monday `compare` Wednesday
LT
```

Наш тип также имеет экземпляр класса `Bounded`, так что мы можем найти минимальный и максимальный день.

```
ghci> minBound :: Day
Monday
ghci> maxBound :: Day
Sunday
```

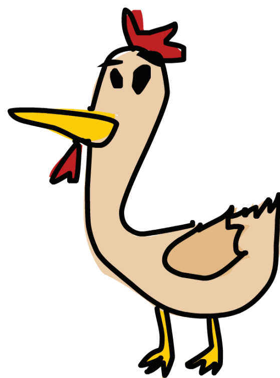
Благодаря тому что тип имеет экземпляр класса `Enum`, можно получать предшествующие и следующие дни, а также задавать диапазоны дней.

```
ghci> succ Monday
Tuesday
ghci> pred Saturday
```

```
Friday
ghci> [Thursday .. Sunday]
[Thursday,Friday,Saturday,Sunday]
ghci> [minBound .. maxBound] :: [Day]
[Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday]
```

Замечательно!

Синонимы типов



Ранее мы упоминали, что типы `[Char]` и `String` являются эквивалентами и могут взаимно заменяться. Это осуществляется с помощью синонимов типов. Синоним типа сам по себе ничего не делает – он просто даёт другое имя существующему типу, облегчая понимание нашего кода и документации. Вот так стандартная библиотека определяет тип `String` как синоним для `[Char]`:

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

Ключевое слово `type` может ввести в заблуждение, потому что на самом деле мы не создаём ничего нового (создаём мы с помощью ключевого слова `data`), а просто определяем синоним для уже существующего типа.

Если мы создадим функцию, которая преобразует строку в верхний регистр, и назовём её `toUpperString`, то можем дать ей сигнатуру типа `toUpperString :: [Char] -> [Char]` или `toUpperString :: String -> String`. Обе сигнатуры обозначают одно и то же, но вторая легче читается.

Улучшенная телефонная книга

Когда мы работали с модулем `Data.Map`, то вначале представляли записную книжку в виде ассоциативного списка, а потом преобразовывали его в отображение. Как мы уже знаем, ассоциативный список – это список пар «ключ–значение». Давайте взглянем на этот вариант записной книжки:

```
phoneBook :: [(String,String)]
```

```
phoneBook =
   [("оля","555-29-38")
   ,("женя","452-29-28")
   ,("катя","493-29-28")
   ,("маша","205-29-28")
   ,("надя","939-82-82")
   ,("юля","853-24-92")
   ]
```

Мы видим, что функция phoneBook имеет тип [(String,String)]. Это говорит о том, что перед нами ассоциативный список, который отображает строки в строки, – но не более. Давайте зададим синоним типа, и мы сможем узнать немного больше по декларации типа:

```
type PhoneBook = [(String,String)]
```

Теперь декларация типа для нашей записной книжки может быть такой: phoneBook :: PhoneBook. Зададим также синоним для String.

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name,PhoneNumber)]
```

Те, кто программирует на языке Haskell, дают синонимы типу String, если хотят сделать объявления более «говорящими» – пояснить, чем являются строки и как они должны использоваться.

Итак, реализуя функцию, которая принимает имя и номер телефона и проверяет, есть ли такая комбинация в нашей записной книжке, мы можем дать ей красивую и понятную декларацию типа:

```
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnumber pbook = (name,pnumber) `elem` pbook
```

Если бы мы не использовали синонимы типов, тип нашей функции был бы String -> String -> [(String,String)] -> Bool. В этом случае декларацию функции легче понять при помощи синонимов типов. Однако не надо перегибать палку. Мы применяем синонимы типов для того, чтобы описать, как используются существующие типы в наших функциях (таким образом декларации типов лучше документированы), или когда мы имеем дело с длинной декларацией типа, которую приходится часто повторять (вроде

[(String,String)]), причем эта декларация обозначает что-то более специфичное в контексте наших функций.

Параметризация синонимов

Синонимы типов также могут быть параметризованы. Если мы хотим задать синоним для ассоциативного списка и при этом нам нужно, чтобы он мог принимать любые типы для ключей и значений, мы можем сделать так:

```
type AssocList k v = [(k,v)]
```

Функция, которая получает значение по ключу в ассоциативном списке, может иметь тип (Eq k) => k -> AssocList k v -> Maybe v. Тип AssocList – это конструктор типов, который принимает два типа и производит конкретный тип, например AssocList Int String.

Мы можем частично применять функции, чтобы получить новые функции; аналогичным образом можно частично применять типы-параметры и получать новые конструкторы типов. Так же, как мы вызываем функцию, не передавая всех параметров для того, чтобы получить новую функцию, мы будем вызывать и конструктор типа, не указывая всех параметров, и получать частично применённый конструктор типа. Если мы хотим получить тип для отображений (из модуля Data.Map) с целочисленными ключами, можно сделать так:

```
type IntMap v = Map Int v
```

или так:

```
type IntMap = Map Int
```

В любом случае конструктор типов IntMap принимает один параметр – это и будет типом, в который мы будем отображать Int.

И вот ещё что. Если вы попытаетесь реализовать этот пример, вам потребуется произвести квалифицированный импорт модуля Data.Map. При квалифицированном импорте перед конструкторами типов также надо ставить имя модуля. Таким образом, мы бы записали: IntMap = Map.Map Int.

Убедитесь, что вы понимаете различие между конструкторами типов и конструкторами данных. Если мы создали синоним типа IntMap или AssocList, это ещё не означает, что можно делать такие

вещи, как AssocList [(1,2),(4,5),(7,9)]. Это означает только то, что мы можем ссылаться на тип, используя другое имя. Можно написать: [(1,2),(3,5),(8,9)] :: AssocList Int Int, в результате чего числа в списке будут трактоваться как целые – но мы также сможем работать с этим списком как с обычным списком пар целых чисел. Синонимы типов (и вообще типы) могут использоваться в языке Haskell только при объявлении типов. Часть языка, относящаяся к объявлению типов, – собственно объявление типов (то есть при определении данных и типов) или часть объявления после символа :: (два двоеточия). Символ :: используется при декларировании или аннотировании типов.

Иди налево, потом направо

Ещё один чудесный тип, принимающий два других в качестве параметров, – это тип Either. Он определён приблизительно так:

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

У него два конструктора данных. Если используется конструктор Left, его содержимое имеет тип a; если Right – содержимое имеет тип b. Таким образом, мы можем использовать данный тип для инкапсуляции значения одного из двух типов. Когда мы работаем с типом Either a b, то обычно используем сопоставление с образцом по Left и Right и выполняем действия в зависимости от того, какой вариант совпал.

```
ghci> Right 20
Right 20
ghci> Left "вOOт"
Left "вOOт"
ghci> :t Right 'a'
Right 'a' :: Either a Char
ghci> :t Left True
Left True :: Either Bool b
```

Из приведённого примера следует, что типом значения Left True является Either Bool b. Первый параметр типа Bool, поскольку значение создано конструктором Left; второй же параметр остался полиморфным. Ситуация подобна тому как значение Nothing имеет тип Maybe a.

Мы видели, что тип Maybe главным образом используется для того, чтобы представить результат вычисления, которое может завершиться неудачей. Но иногда тип Maybe не так удобен, поскольку значение Nothing не несёт никакой информации, кроме того что что-то пошло не так. Это нормально для функций, которые могут выдавать ошибку только в одном случае – или если нам просто не интересно, как и почему функция «упала». Поиск в отображении типа Data.Map может завершиться неудачей, только если искомый ключ не найден, так что мы знаем, что случилось. Но если нам нужно знать, почему не сработала некоторая функция, обычно мы возвращаем результат типа Either a b, где a – это некоторый тип, который может нам что-нибудь рассказать о причине ошибки, и b – результат удачного вычисления. Следовательно, ошибки используют конструктор данных Left, правильные результаты используют конструктор Right.

Например, в школе есть шкафчики для того, чтобы ученикам было куда клеить постеры Guns'n'Roses. Каждый шкафчик открывается кодовой комбинацией. Если школьнику понадобился шкафчик, он говорит администратору, шкафчик под каким номером ему нравится, и администратор выдаёт ему код. Если этот шкафчик уже кем-либо используется, администратор не сообщает код – они вместе с учеником должны будут выбрать другой вариант. Будем использовать модуль Data.Map для того, чтобы хранить информацию о шкафчиках. Это будет отображение из номера шкафчика в пару, где первый компонент указывает, используется шкафчик или нет, а второй компонент – код шкафчика.

```
import qualified Data.Map as Map

data LockerState = Taken | Free deriving (Show, Eq)

type Code = String

type LockerMap = Map.Map Int (LockerState, Code)
```

Довольно просто. Мы объявляем новый тип данных для хранения информации о том, был шкафчик занят или нет. Также мы создаём синоним для кода шкафчика и для типа, который отображает целые числа в пары из статуса шкафчика и кода. Теперь создадим функцию для поиска кода по номеру. Мы будем использовать тип Either String Code для представления результата, так как поиск мо-

жет не удаться по двум причинам – шкафчик уже занят, в этом случае нельзя сообщать код, или номер шкафчика не найден вообще. Если поиск не удался, возвращаем значение типа `String` с пояснениями.

```
lockerLookup :: Int -> LockerMap -> Either String Code
lockerLookup lockerNumber map =
   case Map.lookup lockerNumber map of
    Nothing -> Left $ "Шкафчик № " ++ show lockerNumber ++
                      " не существует!"
    Just (state, code) ->
       if state /= Taken
         then Right code
         else Left $ "Шкафчик № " ++ show lockerNumber ++ " уже занят!"
```

Мы делаем обычный поиск по отображению. Если мы получили значение `Nothing`, то вернём значение типа `Left String`, говорящее, что такой номер не существует. Если мы нашли номер, делаем дополнительную проверку, занят ли шкафчик. Если он занят, возвращаем значение `Left`, говорящее, что шкафчик занят. Если он не занят, возвращаем значение типа `Right Code`, в котором даём студенту код шкафчика. На самом деле это `Right String`, но мы создали синоним типа, чтобы сделать наши объявления более понятными. Вот пример отображения:

```
lockers :: LockerMap
lockers = Map.fromList
   [(100,(Taken,"ZD39I"))
   ,(101,(Free,"JAH3I"))
   ,(103,(Free,"IQSA9"))
   ,(105,(Free,"QOTSA"))
   ,(109,(Taken,"893JJ"))
   ,(110,(Taken,"99292"))
   ]
```

Давайте попытаемся узнать несколько кодов.

```
ghci> lockerLookup 101 lockers
Right "JAH3I"
ghci> lockerLookup 100 lockers
Left "Шкафчик № 100 уже занят!"
ghci> lockerLookup 102 lockers
Left "Шкафчик № 102 не существует!"
ghci> lockerLookup 110 lockers
```

```
Left "Шкафчик № 110 уже занят!"
ghci> lockerLookup 105 lockers
Right "QOTSA"
```

Мы могли бы использовать тип `Maybe` для представления результата, но тогда лишились бы возможности узнать, почему нельзя получить код. А в нашей функции причина ошибки выводится из результирующего типа.

Рекурсивные структуры данных

Как мы уже видели, конструкторы алгебраических типов данных могут иметь несколько полей (или не иметь вовсе), и у каждого поля должен быть конкретный тип. Принимая это во внимание, мы можем создать тип, конструктор которого имеет поля того же самого типа! Таким образом мы можем создавать рекурсивные типы данных, где одно значение некоторого типа содержит другие значения этого типа, а они, в свою очередь, содержат ещё значения того же типа, и т. д.

Посмотрите на этот список: `[5]`. Это упрощённая запись выражения `5:[]`. С левой стороны от оператора `:` ставится значение, с правой стороны – список (в нашем случае пустой). Как насчёт списка `[4,5]`? Его можно переписать так: `4:(5:[])`. Смотря на первый оператор `:`, мы видим, что слева от него – всё так же значение, а справа – список `(5:[])`. То же можно сказать и в отношении списка `3:(4:(5:6:[]))`; это выражение можно переписать и как `3:4:5:6:[]` (поскольку оператор `:` правоассоциативен), и как `[3,4,5,6]`.

Мы можем сказать, что список может быть пустым или это может быть элемент, присоединенный с помощью оператора `:` к другому списку (который в свою очередь может быть пустым или нет).

Ну что ж, давайте используем алгебраические типы данных, чтобы создать наш собственный список.

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

Это можно прочитать почти как наше определение списка в од-

ном из предыдущих разделов. Это либо пустой список, либо комбинация некоторого значения («головы») и собственно списка («хвоста»). Если такая формулировка трудна для понимания, то с использованием синтаксиса записей она будет восприниматься легче.

```
data List a = Empty | Cons { listHead :: a, listTail :: List a}
    deriving (Show, Read, Eq, Ord)
```

Конструктор Cons может вызвать недоумение. Идентификатор Cons – всего лишь альтернативное обозначение :. Как вы видите, в списках оператор : – это просто конструктор, который принимает значение и список и возвращает список. Мы можем использовать и наш новый тип для задания списка! Другими словами, он имеет два поля: первое типа a и второе типа [a].

```
ghci> Empty
Empty
ghci> 5 `Cons` Empty
Cons 5 Empty
ghci> 4 `Cons` (5 `Cons` Empty)
Cons 4 (Cons 5 Empty)
ghci> 3 `Cons` (4 `Cons` (5 `Cons` Empty))
Cons 3 (Cons 4 (Cons 5 Empty))
```

Мы вызываем конструктор Cons как инфиксный оператор, чтобы наглядно показать, что мы используем его вместо оператора :. Конструктор Empty играет роль пустого списка [], и выражение 4 `Cons` (5 `Cons` Empty) подобно выражению 4:(5:[]).

Улучшение нашего списка

Мы можем определить функцию как инфиксную по умолчанию, если её имя состоит только из специальных символов. То же самое можно сделать и с конструкторами, поскольку это просто функции, возвращающие тип данных. Смотрите:

```
infixr 5 :-:
data List a = Empty | a :-: (List a) deriving (Show, Read, Eq, Ord)
```

Первое: мы использовали новую синтаксическую конструкцию, декларацию ассоциативности функции. Если мы определяем функции как операторы, то можем присвоить им значение

ассоциативности, но не обязаны этого делать. Ассоциативность показывает, какова приоритетность оператора и является ли он лево- или правоассоциативным. Например, ассоциативность умножения – `infixl 7 *`, ассоциативность сложения – `infixl 6`. Это значит, что оба оператора левоассоциативны, выражение `4 * 3 * 2` означает `((4 * 3) * 2)`, умножение имеет более высокий приоритет, чем сложение, поэтому выражение `5 * 4 + 3` означает `(5 * 4) + 3`.

Следовательно, ничто не мешает записать `a :-: (List a)` вместо `Cons a (List a)`. Теперь мы можем представлять списки нашего нового спискового типа таким образом:

```
ghci> 3 :-: 4 :-: 5 :-: Empty
3 :-: (4 :-: (5 :-: Empty))
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> 100 :-: a
100 :-: (3 :-: (4 :-: (5 :-: Empty)))
```

Напишем функцию для сложения двух списков. Вот как оператор ++ определён для обычных списков:

```
infixr 5 ++
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Давайте просто передерём это объявление для нашего списка! Назовём нашу функцию ^++:

```
infixr 5   ++
(^++) :: List a -> List a -> List a
Empty ^++ ys = ys
(x :-: xs)   ++ ys = x :-: (xs   ++ ys)
```

И посмотрим, как это работает...

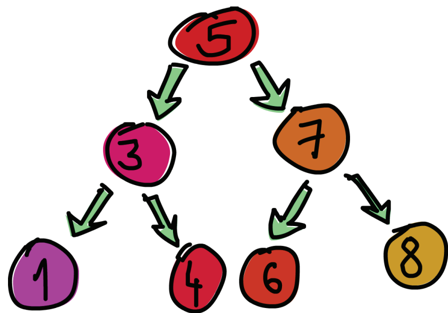
```
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> let b = 6 :-: 7 :-: Empty
ghci> a   ++ b
3 :-: (4 :-: (5 :-: (6 :-: (7 :-: Empty))))
```

Очень хорошо. Если бы мы хотели, мы могли бы реализовать все функции для работы со списками и для нашего спискового типа.

Обратите внимание, как мы выполняли сопоставление с образцом по `(x :-: xs)`. Это работает, потому что на самом деле данная операция сопоставляет конструкторы. Мы можем сопоставлять по конструктору `:-:` потому, что это конструктор для нашего собственного спискового типа, так же как можем сопоставлять и по конструктору `:`, поскольку это конструктор встроенного спискового типа. Так как сопоставление производится только по конструкторам, можно искать соответствие по образцам, подобным `(x :-: xs)`, или константам, таким как `8` или `'a'`, поскольку на самом деле они являются конструкторами для числового и символьного типов[1].

Вырастим-ка дерево

Теперь мы собираемся реализовать бинарное поисковое дерево. Если вам не знакомы поисковые деревья из языков наподобие C, вот что они представляют собой: элемент указывает на два других элемента, один из которых правый, другой – левый. Элемент слева – меньше, чем текущий, элемент справа – больше. Каждый из этих двух элементов также может ссылаться на два других элемента (или на один, или не ссылаться вообще). Получается, что каждый элемент может иметь до двух поддеревьев. Бинарные поисковые деревья удобны тем, что мы знаем, что все элементы в левом поддереве элемента со значением, скажем, пять, будут меньше пяти. Элементы в правом поддереве будут больше пяти. Таким образом, если нам надо найти 8 в нашем дереве, мы начнём с пятёрки, и так как 8 больше 5, будем проверять правое поддерево. Теперь проверим узел со значением 7, и так как 8 больше 7, снова выберем правое поддерево. В результате элемент найдётся всего за три операции сравнения! Если мы бы искали в обычном списке (или в сильно разбалансированном дереве), потребовалось бы до семи сравнений вместо трёх для поиска того же элемента.



[1] На самом деле в синтаксисе языка Haskell имеются ещё так называемые (n + k)-образцы. Впрочем, большая часть сообщества языка их отвергает. – *Прим. ред.*

ПРИМЕЧАНИЕ. *Множества и отображения из модулей* `Data.Set` *и* `Data.Map` *реализованы с помощью деревьев, но вместо обычных бинарных поисковых деревьев они используют сбалансированные поисковые деревья. Дерево называется сбалансированным, если высоты его левого и правого поддеревьев примерно равны. Это условие ускоряет поиск по дереву. В наших примерах мы реализуем обычные поисковые деревья.*

Вот что мы собираемся сказать: дерево – это или пустое дерево, или элемент, который содержит некоторое значение и два поддерева. Такая формулировка идеально соответствует алгебраическому типу данных.

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show)
```

Что ж, отлично. Вместо того чтобы вручную создавать дерево, мы напишем функцию, которая принимает дерево и элемент и добавляет элемент к дереву. Мы будем делать это, сравнивая вставляемый элемент с корневым. Если вставляемый элемент меньше корневого – идём налево, если больше – направо. Эту же операцию продолжаем для каждого последующего узла дерева, пока не достигнем пустого дерева. После этого мы добавляем новый элемент вместо пустого дерева.

В языках, подобных C, мы бы делали это, изменяя указатели и значения внутри дерева. В Haskell мы на самом деле не можем изменять наше дерево – придётся создавать новое поддерево каждый раз, когда мы переходим к левому или правому поддереву. Таким образом, в конце функции добавления мы вернём полностью новое дерево, потому что в языке Haskell нет концепции указателей, есть только значения. Следовательно, тип функции для добавления элемента будет примерно следующим: `a -> Tree a - > Tree a`. Она принимает элемент и дерево и возвращает новое дерево с уже добавленным элементом. Это может показаться неэффективным, но язык Haskell умеет организовывать совместное владение большей частью поддеревьев старым и новым деревьями.

Итак, напишем две функции. Первая будет вспомогательной функцией для создания дерева, состоящего из одного элемента; вторая будет вставлять элемент в дерево.

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree
```

```
treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
    | x == a = Node x left right
    | x < a = Node a (treeInsert x left) right
    | x > a = Node a left (treeInsert x right)
```

Функция singleton служит для создания узла, который хранит некоторое значение и два пустых поддерева. В функции для добавления нового элемента в дерево мы вначале обрабатываем граничное условие. Если мы достигли пустого поддерева, это значит, что мы в нужном месте нашего дерева, и вместо пустого дерева помещаем одноэлементное дерево, созданное из нашего значения. Если мы вставляем не в пустое дерево, следует кое-что проверить. Первое: если вставляемый элемент равен корневому элементу – просто возвращаем дерево текущего элемента. Если он меньше, возвращаем дерево, которое имеет то же корневое значение и то же правое поддерево, но вместо левого поддерева помещаем дерево с добавленным элементом. Так же (но с соответствующими поправками) обстоит дело, если значение больше, чем корневой элемент.

Следующей мы напишем функцию для проверки, входит ли некоторый элемент в наше дерево или нет. Для начала определим базовые случаи. Если мы ищем элемент в пустом дереве, его там определённо нет. Заметили – такой же базовый случай мы использовали для поиска элемента в списке? Если мы ищем в пустом списке, то ничего не найдём. Если ищем не в пустом дереве, надо проверить несколько условий. Если элемент в текущем корне равен тому, что мы ищем, – отлично. Ну а если нет, тогда как быть?.. Мы можем извлечь пользу из того, что все элементы в левом поддереве меньше корневого элемента. Поэтому, если искомый элемент меньше корневого, начинаем искать в левом поддереве. Если он больше – ищем в правом поддереве.

```
treeElem :: (Ord a) => a -> Tree a -> Bool
treeElem x EmptyTree = False
treeElem x (Node a left right)
    | x == a = True
    | x < a = treeElem x left
    | x > a = treeElem x right
```

Всё, что нам нужно было сделать, – переписать предыдущий параграф в коде. Давайте немного «погоняем» наши деревья. Вместо того чтобы вручную задавать деревья (а мы можем!), будем использовать свёртку для того, чтобы создать дерево из списка. Запомните: всё, что обходит список элемент за элементом и возвращает некоторое значение, может быть представлено свёрткой. Мы начнём с пустого дерева и затем будем проходить список справа налево и вставлять элемент за элементом в дерево-аккумулятор.

```
ghci> let nums = [8,6,4,1,7,3,5]
ghci> let numsTree = foldr treeInsert EmptyTree nums
ghci> numsTree
Node 5
     (Node 3
         (Node 1 EmptyTree EmptyTree)
         (Node 4 EmptyTree EmptyTree)
      )
      (Node 7
         (Node 6 EmptyTree EmptyTree)
         (Node 8 EmptyTree EmptyTree)
      )
```

ПРИМЕЧАНИЕ. *Если вы вызовете этот код в интерпретаторе GHCi, то в качестве вывода будет одна длинная строка. Здесь она разбита на несколько строк, иначе она бы вышла за пределы страницы.*

В этом вызове функции `foldr` функция `treeInsert` играет роль функции свёртки (принимает дерево и элемент списка и создаёт новое дерево); `EmptyTree` – стартовое значение аккумулятора. Параметр `nums` – это, конечно же, список, который мы сворачиваем.

Если напечатать дерево на консоли, мы получим не очень-то легко читаемое выражение, но если постараться, можно уловить структуру. Мы видим, что корневое значение – `5`; оно имеет два поддерева, в одном из которых корневым элементом является `3`, а в другом – `7`, и т. д.

```
ghci> 8 `treeElem` numsTree
True
ghci> 100 `treeElem` numsTree
False
ghci> 1 `treeElem` numsTree
```

```
True
ghci> 10 `treeElem` numsTree
False
```

Проверка на вхождение также работает отлично. Классно!

Как вы можете видеть, алгебраические типы данных в языке Haskell нереально круты. Мы можем использовать их для создания чего угодно – от булевских значений и перечислимого типа для дней недели до бинарных поисковых деревьев и даже большего!

Классы типов, второй семестр

Мы уже изучили несколько стандартных классов типов языка Haskell и некоторые типы, имеющие для них экземпляры. Также мы знаем, как автоматически сделать для наших типов экземпляры стандартных классов, стоит только попросить Haskell автоматически сгенерировать нужное нам поведение. В этой главе будет рассказано о том, как писать свои собственные классы типов и как создавать экземпляры класса вручную.

Вспомним, что классы типов по сути своей подобны интерфейсам. Они определяют некоторое поведение (проверку на равенство, проверку на «больше-меньше», перечисление элементов). Типы, обладающие таким поведением, можно сделать экземпляром класса типов. Поведение класса типов определяется функциями, входящими в класс, или просто декларацией класса; элементы класса мы потом должны будем реализовать. Таким образом, если мы говорим, что для типа имеется экземпляр класса, то подразумеваем, что можем использовать все функции, определённые в классе типов в нашем типе.

> **ПРИМЕЧАНИЕ.** *Классы типов практически не имеют ничего общего с классами в таких языках, как Java или Python. Это сбивает с толку, поэтому советую вам забыть всё, что вы знаете о классах в императивных языках!*

«Внутренности» класса Eq

Возьмём для примера класс типов `Eq`: он используется в отношении неких значений, которые можно проверить на равенство. Он определяет операторы `==` и `/=`. Если у нас есть тип, скажем, `Car` (*автомобиль*),

и сравнение двух автомобилей с помощью функции == имеет смысл, то имеет смысл и определить для типа Car экземпляр класса Eq.

Вот как класс Eq определён в стандартном модуле:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

О-хо-хо!.. Новый синтаксис и новые ключевые слова. Не беспокойтесь, скоро мы это поясним. Прежде всего, мы записали декларацию class Eq a where – это означает, что мы определяем новый класс, имя которого Eq. Идентификатор a – это переменная типа; иными словами, идентификатор играет роль типа, который в дальнейшем будет экземпляром нашего класса. Эту переменную необязательно называть именно a; пусть даже имя не состоит из одной буквы, но оно непременно должно начинаться с символа в нижнем регистре. Затем мы определяем несколько функций. Нет необходимости писать реализацию функций – достаточно только декларации типа.

Некоторым будет проще понять эту декларацию, если мы запишем class Eq equatable where, а затем декларации функций, например (==) :: equatable -> equatable -> Bool.

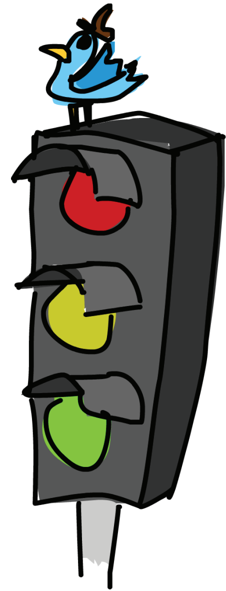
Мы определили тела функций для функций в классе Eq, притом определили их *взаимно рекурсивно*. Мы записали, что два экземпляра класса Eq равны, если они не отличаются, и что они отличаются, если не равны. Необязательно было поступать так, и всё же скоро мы увидим, чем это может быть полезно.

Если записать декларацию class Eq a where, описать в ней функцию таким образом: (==) :: a -> a -> Bool, а затем посмотреть объявление этой функции, мы увидим следующий тип: (Eq a) => a -> a -> Bool.

Тип для представления светофора

Итак, что мы можем сделать с классом после того, как объявили его? Весьма немногое. Но как только мы начнём создавать экземпляры этого класса, то станем получать интересные результаты. Посмотрим на этот тип:

```
data TrafficLight = Red | Yellow | Green
```



Он определяет состояние светофора. Обратите внимание, что мы не порождаем автоматическую реализацию классов для него. Мы собираемся реализовать их поддержку вручную, даже несмотря на то, что многое можно было бы сгенерировать автоматически, например экземпляры для классов Eq и Show. Вот как мы создадим экземпляр для класса Eq.

```
instance Eq TrafficLight where
   Red == Red = True
   Green == Green = True
   Yellow == Yellow = True
   _ == _ = False
```

Экземпляр создан с помощью ключевого слова instance. Таким образом, ключевое слово class служит для определения новых классов типов, а ключевое слово instance – для того, чтобы сделать для нашего типа экземпляр некоторого класса. Когда мы определяли класс Eq, то записали декларацию class Eq a where и сказали, что идентификатор a играет роль типа, который мы позднее будем делать экземпляром класса. Теперь мы это ясно видим, потому что когда мы создаём экземпляр, то пишем: instance Eq TrafficLight where. Мы заменили идентификатор на название нашего типа.

Так как операция == была определена в объявлении класса через вызов операции /= и наоборот, следует переопределить только одну функцию в объявлении экземпляра класса. Это называется *минимальным полным определением класса типов* – имеется в виду минимум функций, которые надо реализовать, чтобы наш тип мог вести себя так, как предписано классом. Для того чтобы создать минимально полное определение для класса Eq, нам нужно реализовать или оператор ==, или оператор /=. Если бы класс Eq был определён таким образом:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

то нам бы потребовалось реализовывать обе функции при создании экземпляра, потому что язык Haskell не знал бы, как эти функции

взаимосвязаны. В этом случае минимально полным определением были бы обе функции, == и /=.

Мы реализовали оператор == с помощью сопоставления с образцом. Так как комбинаций двух неравных цветов значительно больше, чем комбинаций равных, мы перечислили все равные цвета и затем использовали маску подстановки, которая говорит, что если ни один из предыдущих образцов не подошёл, то два цвета не равны.

Давайте сделаем для нашего типа экземпляр класса Show. Чтобы удовлетворить минимально полному определению для класса Show, мы должны реализовать функцию show, которая принимает значение и возвращает строку:

```
instance Show TrafficLight where
   show Red = "Красный свет"
   show Yellow = "Жёлтый свет"
   show Green = "Зелёный свет"
```

Мы снова использовали сопоставление с образцом, чтобы достичь нашей цели. Давайте посмотрим, как это всё работает:

```
ghci> Red == Red
True
ghci> Red == Yellow
False
ghci> Red `elem` [Red, Yellow, Green]
True
ghci> [Red, Yellow, Green]
[Красный свет,Жёлтый свет,Зелёный свет]
```

Можно было бы просто автоматически сгенерировать экземпляр для класса Eq с абсолютно тем же результатом (мы этого не сделали в образовательных целях). Кроме того, автоматическая генерация для класса Show просто напрямую переводила бы конструкторы значений в строки. Если нам требуется печатать что-то дополнительно, то придётся создавать экземпляр класса Show вручную.

Наследование классов

Также можно создавать классы типов, которые являются подклассами других классов типов. Декларация класса Num довольно длинна, но вот её начало:

```
class (Eq a) => Num a where
   ...
```

Как уже говорилось ранее, есть множество мест, куда мы можем втиснуть ограничения на класс. Наша запись равнозначна записи `class Num a where`, но мы требуем, чтобы тип `a` имел экземпляр класса `Eq`. Это означает, что мы должны определить для нашего типа экземпляр класса `Eq` до того, как сможем сделать для него экземпляр класса `Num`. Прежде чем некоторый тип сможет рассматриваться как число, мы должны иметь возможность проверять значения этого типа на равенство.

Ну вот и всё, что надо знать про наследование, – это просто ограничения на класс типа-параметра при объявлении класса. При написании тел функций в декларации класса или при их определении в экземпляре класса мы можем полагать, что тип `a` имеет экземпляр для класса `Eq` и, следовательно, допускается использование операторов `==` и `/=` со значениями этого типа.

Создание экземпляров классов для параметризованных типов

Но как тип `Maybe` и списковый тип сделаны экземплярами классов? Тип `Maybe` отличается, скажем, от типа `TrafficLight` тем, что `Maybe` сам по себе не является конкретным типом – это конструктор типов, который принимает один тип-параметр (например, `Char`), чтобы создать конкретный тип (как `Maybe Char`). Давайте посмотрим на класс `Eq` ещё раз:

```
class Eq a where
   (==) :: a -> a -> Bool
   (/=) :: a -> a -> Bool
   x == y = not (x /= y)
   x /= y = not (x == y)
```

Из декларации типа мы видим, что `a` используется как конкретный тип, потому что все типы в функциях должны быть конкретными (помните, мы обсуждали, что не можем иметь функцию типа `a -> Maybe`, но можем – функцию типа: `a -> Maybe a` или `Maybe Int -> Maybe String`). Вот почему недопустимо делать что-нибудь в таком роде:

```
instance Eq Maybe where
    ...
```

Ведь, как мы видели, идентификатор `a` должен принимать значение в виде конкретного типа, а тип `Maybe` не является таковым. Это конструктор типа, который принимает один параметр и производит конкретный тип.

Было бы скучно прописывать `instance Eq (Maybe Int) where`, `instance Eq (Maybe Char) where` и т. д. для всех существующих типов. Вот почему мы можем записать это так:

```
instance Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

Это всё равно что сказать, что мы хотим сделать для всех типов формата `Maybe` <*нечто*> экземпляр класса `Eq`. Мы даже могли бы записать `(Maybe something)`, но обычно программисты используют одиночные буквы, чтобы придерживаться стиля языка Haskell. Выражение `(Maybe m)` выступает в качестве типа `a` в декларации `class Eq a where`. Тип `Maybe` не является конкретным типом, а `Maybe m` – является. Указание типа-параметра (`m` в нижнем регистре) свидетельствует о том, что мы хотим, чтобы все типы вида `Maybe m`, где `m` – любой тип, имели экземпляры класса `Eq`.

Однако здесь есть одна проблема. Заметили? Мы используем оператор `==` для содержимого типа `Maybe`, но у нас нет уверенности, что то, что содержит тип `Maybe`, может быть использовано с методами класса `Eq`. Вот почему необходимо поменять декларацию экземпляра на следующую:

```
instance (Eq m) => Eq (Maybe m) where
   Just x == Just y = x == y
   Nothing == Nothing = True
   _ == _ = False
```

Нам пришлось добавить ограничение на класс. Таким объявлением экземпляра класса мы утверждаем: необходимо, чтобы все типы вида `Maybe m` имели экземпляр для класса `Eq`, но при этом тип `m` (тот, что хранится в `Maybe`) также должен иметь экземпляр класса `Eq`. Такой же экземпляр породил бы сам язык Haskell, если бы мы воспользовались директивой `deriving`.

В большинстве случаев ограничения на класс в декларации класса используются для того, чтобы сделать класс подклассом другого класса. Ограничения на класс в определении экземпляра используются для того, чтобы выразить требования к содержимому некоторого типа. Например, в данном случае мы требуем, чтобы содержимое типа `Maybe` также имело экземпляр для класса `Eq`.

При создании экземпляров, если вы видите, что тип использовался как конкретный при декларации (например, `a -> a -> Bool`), а вы реализуете экземпляр для конструктора типов, следует предоставить тип-параметр и добавить скобки, чтобы получить конкретный тип.

Примите во внимание, что тип, экземпляр для которого вы пытаетесь создать, заменит параметр в декларации класса. Параметр `a` из декларации `class Eq a where` будет заменён конкретным типом при создании экземпляра; попытайтесь в уме заменить тип также и в декларациях функций. Сигнатура `(==) :: Maybe -> Maybe -> Bool` не имеет никакого смысла, но сигнатура `(==) :: (Eq m) => Maybe m -> Maybe m -> Bool` имеет. Впрочем, это нужно только для упражнения, потому что оператор == всегда будет иметь тип `(==) :: (Eq a) => a -> a -> Bool` независимо от того, какие экземпляры мы порождаем.

О, и ещё одна классная фишка! Если хотите узнать, какие экземпляры существуют для класса типов, вызовите команду `:info` в GHCi. Например, выполнив команду `:info Num`, вы увидите, какие функции определены в этом классе типов, и выведете список принадлежащих классу типов. Команда `:info` также работает с типами и конструкторами типов. Если выполнить `:info Maybe`, мы увидим все классы типов, к которым относится тип `Maybe`. Вот пример:

```
ghci> :info Maybe
data Maybe a = Nothing | Just a  -- Defined in Data.Maybe
instance Eq a => Eq (Maybe a) -- Defined in Data.Maybe
instance Monad Maybe -- Defined in Data.Maybe
instance Functor Maybe -- Defined in Data.Maybe
instance Ord a => Ord (Maybe a) -- Defined in Data.Maybe
instance Read a => Read (Maybe a) -- Defined in GHC.Read
instance Show a => Show (Maybe a) -- Defined in GHC.Show
```

Класс типов «да–нет»

В языке JavaScript и в некоторых других слабо типизированных языках вы можете поместить в оператор `if` практически любые выражения. Например, все следующие выражения правильные:

```
if (0) alert("ДА!") else alert("НЕТ!")
```

```
if ("") alert ("ДА!") else alert("НЕТ!")
```

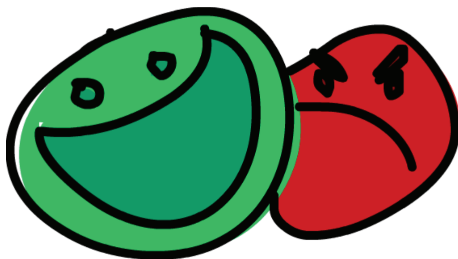
```
if (false) alert("ДА!") else alert("НЕТ!)
```

и все они покажут `НЕТ!"`.

Если вызвать

```
if ("ЧТО") alert ("ДА!") else alert("НЕТ!")
```

мы увидим `"ДА!"`, так как язык JavaScript рассматривает непустые строки как вариант истинного значения.



Несмотря на то что строгое использование типа `Bool` для булевских выражений является преимуществом языка Haskell, давайте реализуем подобное поведение. Просто для забавы. Начнём с декларации класса:

```
class YesNo a where
   yesno :: a -> Bool
```

Довольно просто. Класс типов `YesNo` определяет один метод. Эта функция принимает одно значение некоторого типа, который может рассматриваться как хранитель некоей концепции истинности; функция говорит нам, истинно значение или нет. Обратите внимание: из того, как мы использовали параметр `a` в функции, следует, что он должен быть конкретным типом.

Теперь определим несколько экземпляров. Для чисел, так же как и в языке JavaScript, предположим, что любое ненулевое значение истинно, а нулевое – ложно.

```
instance YesNo Int where
   yesno 0 = False
```

```
   yesno _ = True
```

Пустые списки (и, соответственно, строки) считаются имеющими ложное значение; не пустые списки истинны.

```
instance YesNo [a] where
   yesno [] = False
   yesno _ = True
```

Обратите внимание, как мы записали тип-параметр для того, чтобы сделать список конкретным типом, но не делали никаких предположений о типе, хранимом в списке. Что ещё? Гм-м... Я знаю, что тип `Bool` также содержит информацию об истинности или ложности, и сообщает об этом довольно недвусмысленно:

```
instance YesNo Bool where
   yesno = id
```

Что? Какое `id`?.. Это стандартная библиотечная функция, которая принимает параметр и его же и возвращает. Мы всё равно записали бы то же самое. Сделаем экземпляр для типа `Maybe`:

```
instance YesNo (Maybe a) where
   yesno (Just _) = True
   yesno Nothing = False
```

Нам не нужно ограничение на класс параметра, потому что мы не делаем никаких предположений о содержимом типа `Maybe`. Мы говорим, что он истинен для всех значений `Just` и ложен для значения `Nothing`. Нам приходится писать `(Maybe a)` вместо просто `Maybe`, потому что, если подумать, не может существовать функции `Maybe -> Bool`, так как `Maybe` – не конкретный тип; зато может существовать функция `Maybe a -> Bool`. Круто – любой тип вида `Maybe <нечто>` является частью `YesNo` независимо от того, что представляет собой это «нечто»!

Ранее мы определили тип `Tree` для представления бинарного поискового дерева. Мы можем сказать, что пустое дерево должно быть аналогом ложного значения, а не пустое – истинного.

```
instance YesNo (Tree a) where
   yesno EmptyTree = False
   yesno _ = True
```

Есть ли аналоги истинности и ложности у цветов светофора? Конечно. Если цвет красный, вы останавливаетесь. Если зелёный – идёте. Ну а если жёлтый? Ну, я обычно бегу на жёлтый: жить не могу без адреналина!

```
instance YesNo TrafficLight where
   yesno Red = False
   yesno _ = True
```

Ну что ж, мы определили несколько экземпляров, а теперь давайте поиграем с ними:

```
ghci> yesno $ length []
False
ghci> yesno "ха-ха"
True
ghci> yesno ""
False
ghci> yesno $ Just 0
True
ghci> yesno True
True
ghci> yesno EmptyTree
False
ghci> yesno []
False
ghci> yesno [0,0,0]
True
ghci> :t yesno
yesno :: (YesNo a) => a -> Bool
```

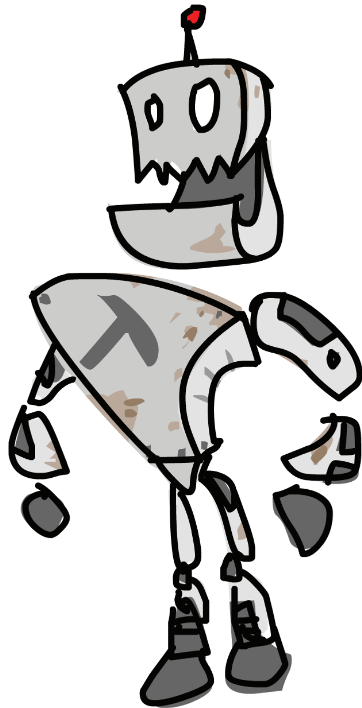
Та-ак, работает. Теперь сделаем функцию, которая работает, как оператор `if`, но со значениями типов, для которых есть экземпляр класса `YesNo`:

```
yesnoIf :: (YesNo y) => y -> a -> a -> a
yesnoIf yesnoVal yesResult noResult =
    if yesno yesnoVal
        then yesResult
        else noResult
```

Всё довольно очевидно. Функция принимает значение для определения истинности и два других параметра. Если значение истинно, возвращается первый параметр; если нет – второй.

```
ghci> yesnoIf [] "ДА!" "НЕТ!"
"НЕТ!"
ghci> yesnoIf [2,3,4] "ДА!" "НЕТ!"
"ДА!"
ghci> yesnoIf True "ДА!" "НЕТ!"
"ДА!"
ghci> yesnoIf (Just 500) "ДА!" "НЕТ!"
"ДА!"
ghci> yesnoIf Nothing "ДА!" НЕТ!"
НЕТ!"
```

Класс типов Functor



Мы уже встречали множество классов типов из стандартной библиотеки. Ознакомились с классом `Ord`, предусмотренным для сущностей, которые можно упорядочить. Вдоволь набаловались с классом `Eq`, предназначенным для сравнения на равенство. Изучили класс `Show`, предоставляющий интерфейс для типов, которые можно представить в виде строк. Наш добрый друг класс `Read` помогает, когда нам надо преобразовать строку в значение некоторого типа. Ну а теперь приступим к рассмотрению класса типов `Functor`, предназначенного для типов, которые могут быть отображены друг в друга.

Возможно, в этот момент вы подумали о списках: ведь отображение списков – это очень распространённая идиома в языке Haskell. И вы правы: списковый тип имеет экземпляр для класса `Functor`.

Нет лучшего способа изучить класс типов `Functor`, чем посмотреть, как он реализован. Вот и посмотрим:

```
fmap :: (a -> b) -> f a -> f b
```

Итак, что у нас имеется? Класс определяет одну функцию fmap и не предоставляет для неё реализации по умолчанию. Тип функции fmap весьма интересен. Во всех вышеприведённых определениях классов типов тип-параметр, игравший роль типа в классе, был некоторого конкретного типа, как переменная a в сигнатуре (==) :: (Eq a) => a -> a -> Bool. Но теперь тип-параметр f не имеет конкретного типа (нет конкретного типа, который может принимать переменная, например Int, Bool или Maybe String); в этом случае переменная – конструктор типов, принимающий один параметр. (Напомню: выражение Maybe Int является конкретным типом, а идентификатор Maybe – конструктор типов с одним параметром.) Мы видим, что функция fmap принимает функцию из одного типа в другой и функтор, применённый к одному типу, и возвращает функтор, применённый к другому типу.

Если это звучит немного непонятно, не беспокойтесь. Всё прояснится, когда мы рассмотрим несколько примеров.

Гм-м... что-то мне напоминает объявление функции fmap! Если вы не знаете сигнатуру функции map, вот она:

```
map :: (a -> b) -> [a] -> [b]
```

О, как интересно! Функция map берёт функцию из a в b и список элементов типа a и возвращает список элементов типа b. Друзья, мы только что обнаружили функтор! Фактически функция map – это функция fmap, которая работает только на списках. Вот как список сделан экземпляром класса Functor:

```
instance Functor [] where
   fmap = map
```

И всё! Заметьте, мы не пишем instance Functor [a] where, потому что из определения функции

```
fmap :: (a -> b) -> f a -> f b
```

мы видим, что параметр f должен быть конструктором типов, принимающим один тип. Выражение [a] – это уже конкретный тип (список элементов типа a), а вот [] – это конструктор типов, который принимает один тип; он может производить такие конкретные типы, как [Int], [String] или даже [[String]].

Так как для списков функция fmap – это просто map, то мы получим одинаковые результаты при их использовании на списках:

```
map :: (a -> b) -> [a] -> [b]
ghci>fmap (*2) [1..3]
[2,4,6]
ghci> map (*2) [1..3]
[2,4,6]
```

Что случится, если применить функцию `map` или `fmap` к пустому списку? Мы получим опять же пустой список. Но функция `fmap` преобразует пустой список типа `[a]` в пустой список типа `[b]`.

Экземпляр класса Functor для типа Maybe

Типы, которые могут вести себя как контейнеры по отношению к другим типам, могут быть функторами. Можно представить, что списки – это коробки с бесконечным числом отсеков; все они могут быть пустыми, или же один отсек заполнен, а остальные пустые, или несколько из них заполнены. А что ещё умеет быть контейнером для других типов? Например, тип `Maybe`. Он может быть «пустой коробкой», и в этом случае имеет значение `Nothing`, или же в нём хранится какое-то одно значение, например `"ХА-ХА"`, и тогда он равен `Just "ХА-ХА"`.

Вот как тип `Maybe` сделан функтором:

```
instance Functor Maybe where
   fmap f (Just x) = Just (f x)
   fmap f Nothing = Nothing
```

Ещё раз обратите внимание на то, как мы записали декларацию `instance Functor Maybe where` вместо `instance Functor (Maybe m) where` – подобно тому как мы делали для класса `YesNo`. Функтор принимает конструктор типа с одним параметром, не конкретный тип. Если вы мысленно замените параметр `f` на `Maybe`, функция `fmap` работает как `(a -> b) -> Maybe a -> Maybe b`, только для типа `Maybe`, что вполне себя оправдывает. Но если заменить `f` на `(Maybe m)`, то получится `(a -> b) -> Maybe m a -> Maybe m b`, что не имеет никакого смысла, так как тип `Maybe` принимает только один тип-параметр.

Как бы то ни было, реализация функции `fmap` довольно проста. Если значение типа `Maybe` – это `Nothing`, возвращается `Nothing`. Если мы отображаем «пустую коробку», мы получим «пустую коробку», что логично. Точно так же функция `map` для пустого списка возвращает пустой список. Если это не пустое значение, а некоторое зна-

чение, упакованное в конструктор `Just`, то мы применяем функцию к содержимому `Just`:

```
ghci> fmap (++ " ПРИВЕТ, Я ВНУТРИ JUST") (Just "Серьёзная штука.")
Just "Серьёзная штука. ПРИВЕТ, Я ВНУТРИ JUST"
ghci> fmap (++ " ПРИВЕТ, Я ВНУТРИ JUST") Nothing
Nothing
ghci> fmap (*2) (Just 200)
Just 400
ghci> fmap (*2) Nothing
Nothing
```

Деревья тоже являются функторами

Ещё один тип, который можно отображать и сделать для него экземпляр класса `Functor`, – это наш тип `Tree`. Дерево может хранить ноль или более других элементов, и конструктор типа `Tree` принимает один тип-параметр. Если бы мы хотели записать функцию `fmap` только для типа `Tree`, её сигнатура выглядела бы так: `(a -> b) -> Tree a -> Tree b`.

Для этой функции нам потребуется рекурсия. Отображение пустого дерева возвращает пустое дерево. Отображение непустого дерева – это дерево, состоящее из результата применения функции к корневому элементу и из правого и левого поддеревьев, к которым также было применено отображение.

```
instance Functor Tree where
   fmap f EmptyTree = EmptyTree
   fmap f (Node x left right) = Node (f x) (fmap f left) (fmap f right)
```

Проверим:

```
ghci> fmap (*2) EmptyTree
EmptyTree
ghci> fmap (*4) (foldr treeInsert EmptyTree [5,7,3])
Node 20 (Node 12 EmptyTree EmptyTree) (Node 28 EmptyTree EmptyTree)
```

Впрочем, тут следует быть внимательным! Если тип `Tree` используется для представления бинарного дерева поиска, то нет никакой гарантии, что дерево останется таковым после применения к каждому его узлу некоторой фукции. Проход по дереву функцией, скажем, `negate` превратит дерево поиска в обычное дерево.

И тип Either является функтором

Отлично! Ну а теперь как насчёт `Either a b`? Можно ли сделать его функтором? Класс типов `Functor` требует конструктор типов с одним параметром, а у типа `Either` их два. Гм-м... Придумал – мы частично применим конструктор `Either`, «скормив» ему один параметр, и таким образом он получит один свободный параметр. Вот как для типа `Either` определён экземпляр класса `Functor` в стандартных библиотеках:

```
instance Functor (Either a) where
   fmap f (Right x) = Right (f x)
   fmap f (Left x) = Left x
```

Что же здесь происходит? Как видно из записи, мы сделали экземпляр класса не для типа `Either`, а для `Either a`. Это потому, что `Either` – конструктор типа, который принимает два параметра, а `Either a` – только один. Если бы функция `fmap` была только для `Either a`, сигнатура типа выглядела бы следующим образом:

```
(b -> c) -> Either a b -> Either a c
```

поскольку это то же самое, что

```
(b -> c) -> (Either a) b -> (Either a) c
```

В реализации мы выполняем отображение в конструкторе данных `Right`, но не делаем этого в `Left`. Почему? Вспомним, как определён тип `Either a b`:

```
data Either a b = Left a | Right b
```

Если мы хотим применять некую функцию к обеим альтернативам, параметры `a` и `b` должны конкретизироваться одним и тем же типом. Если попытаться применить функцию, которая принимает строку и возвращает строку, то `b` у нас – строка, а `a` – число; это не сработает. Также, когда мы смотрели на тип функции `fmap` для типа `Either a`, то видели, что первый параметр не изменяется, а второй может быть изменён; первый параметр актуализируется конструктором данных `Left`.

Здесь можно продолжить нашу аналогию с коробками, представив часть `Left` как пустую коробку, на которой сбоку записано сообщение об ошибке, поясняющее, почему внутри пусто.

Отображения из модуля `Data.Map` также можно сделать функтором, потому что они хранят (или не хранят) значения. Для типа `Map k v` функция `fmap` будет применять функцию `v -> v'` на отображении типа `Map k v` и возвращать отображение типа `Map k v'`.

> **ПРИМЕЧАНИЕ.** *Обратите внимание: апостроф не имеет специального значения в типах (как не имеет его и в именовании значений). Этот символ используется для обозначения схожих понятий, незначительно отличающихся друг от друга.*

Попытайтесь самостоятельно догадаться, как для типа `Map k` определён экземпляр класса `Functor`!

На примере класса типов `Functor` мы увидели, что классы типов могут представлять довольно мощные концепции высокого порядка. Также немного попрактиковались в частичном применении типов и создании экземпляров. В одной из следующих глав мы познакомимся с законами, которые должны выполняться для функторов.

Сорта и немного тип-фу

Конструкторы типов принимают другие типы в качестве параметров для того, чтобы рано или поздно вернуть конкретный тип. Это в некотором смысле напоминает мне функции, которые принимают значения в качестве параметров для того, чтобы вернуть значение. Мы видели, что конструкторы типов могут быть частично применены, так же как и функции (`Either String` – это тип, который принимает ещё один тип и возвращает конкретный тип, например `Either String Int`). Это очень интересно. В данном разделе мы рассмотрим формальное определение того, как типы применяются к конструкторам типов. Точно так же мы выясняли, как формально определяется применение значе-

ний к функциям по декларациям типов. Вам не обязательно читать этот раздел для того, чтобы продолжить своё волшебное путешествие в страну языка Haskell, и если вы не поймёте, что здесь изложено, – не стоит сильно волноваться. Тем не менее, если вы усвоили содержание данного раздела, это даст вам чёткое понимание системы типов.

Итак, значения, такие как 3, "ДА" или takeWhile (функции тоже являются значениями, поскольку мы можем передать их как параметр и т. д.), имеют свой собственный тип. Типы – это нечто вроде маленьких меток, привязанных к значениям, чтобы мы могли строить предположения относительно них. Но и типы имеют свои собственные маленькие меточки, называемые *сортами*. Сорт – это нечто вроде «типа типов». Звучит немного странно, но на самом деле это очень мощная концепция.

Что такое сорта и для чего они полезны? Давайте посмотрим сорт типа, используя команду :k в интерпретаторе GHCi.

```
ghci> :k Int
Int :: *
```

Звёздочка? Как затейливо! Что это значит? Звёздочка обозначает, что тип является конкретным. *Конкретный тип* – это такой тип, у которого нет типов-параметров; значения могут быть только конкретных типов. Если бы мне надо было прочитать символ * вслух (до этого не приходилось), я бы сказал «звёздочка» или просто «тип».

О'кей, теперь посмотрим, каков сорт у типа Maybe:

```
ghci> :k Maybe
Maybe :: * -> *
```

Конструктор типов Maybe принимает один конкретный тип (например, Int) и возвращает конкретный тип (например, Maybe Int). Вот о чём говорит нам сорт. Точно так же тип Int -> Int означает, что функция принимает и возвращает значение типа Int; сорт * -> * означает, что конструктор типов принимает конкретный тип и возвращает конкретный тип. Давайте применим параметр к типу Maybe и посмотрим, какого он станет сорта.

```
ghci> :k Maybe Int
Maybe Int :: *
```

Так я и думал! Мы применили тип-параметр к типу Maybe и получили конкретный тип. Можно провести параллель (но не отождествление: типы – это не то же самое, что и сорта) с тем, как если бы мы сделали :t isUpper и :t isUpper 'A'. У функции isUpper тип Char -> Bool; выражение isUpper 'A' имеет тип Bool, потому что его значение – просто False. Сорт обоих типов, тем не менее, *.

Мы используем команду :k для типов, чтобы получить их сорт, так же как используем команду :t для значений, чтобы получить их тип. Выше уже было сказано, что типы – это метки значений, а сорта – это метки типов; и в этом они схожи.

Посмотрим на другие сорта.

```
ghci> :k Either
Either :: * -> * -> *
```

Это говорит о том, что тип Either принимает два конкретных типа для того, чтобы вернуть конкретный тип. Выглядит как декларация функции, которая принимает два значения и что-то возвращает. Конструкторы типов являются каррированными (так же, как и функции), поэтому мы можем частично применять их.

```
ghci> :k Either String
Either String :: * -> *
ghci> :k Either String Int
Either String Int :: *
```

Когда нам нужно было сделать для типа Either экземпляр класса Functor, пришлось частично применить его, потому что класс Functor принимает типы только с одним параметром, в то время как у типа Either их два. Другими словами, класс Functor принимает типы сорта * -> *, и нам пришлось частично применить тип Either для того, чтобы получить сорт * -> * из исходного сорта * -> * -> *. Если мы посмотрим на определение класса Functor ещё раз:

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

то увидим, что переменная типа f используется как тип, принимающий один конкретный тип для того, чтобы создать другой. Мы знаем, что возвращается конкретный тип, поскольку он используется как тип значения в функции. Из этого можно заключить, что

типы, которые могут «подружиться» с классом Functor, должны иметь сорт * -> *.

Ну а теперь займёмся тип-фу. Посмотрим на определение такого класса типов:

```
class Tofu t where
  tofu :: j a -> t a j
```

Объявление выглядит странно. Как мы могли бы создать тип, который будет иметь экземпляр такого класса? Посмотрим, каким должен быть сорт типа. Так как тип j a используется как тип значения, который функция tofu принимает как параметр, у типа j a должен быть сорт *. Мы предполагаем сорт * для типа a и, таким образом, можем вывести, что тип j должен быть сорта * -> *. Мы видим, что тип t также должен производить конкретный тип, и что он принимает два типа. Принимая во внимание, что у типа a сорт * и у типа j сорт * -> *, мы выводим, что тип t должен быть сорта * -> (* -> *) -> *. Итак, он принимает конкретный тип (a) и конструктор типа, который принимает один конкретный тип (j), и производит конкретный тип. Вау!

Хорошо, давайте создадим тип такого сорта: * -> (* -> *) -> *. Вот один из вариантов:

```
data Frank a b = Frank {frankField :: b a} deriving (Show)
```

Откуда мы знаем, что этот тип имеет сорт * -> (* -> *) - > *? Именованные поля в алгебраических типах данных сделаны для того, чтобы хранить значения, так что они по определению должны иметь сорт *. Мы предполагаем сорт * для типа a; это означает, что тип b принимает один тип как параметр. Таким образом, его сорт – * -> *. Теперь мы знаем сорта типов a и b; так как они являются параметрами для типа Frank, можно показать, что тип Frank имеет сорт * -> (* -> *) - > *. Первая * обозначает сорт типа a; (* -> *) обозначает сорт типа b. Давайте создадим несколько значений типа Frank и проверим их типы.

```
ghci> :t Frank {frankField = Just "ХА-ХА"}
Frank {frankField = Just "ХА-ХА"} :: Frank [Char] Maybe
ghci> :t Frank {frankField = Node 'a' EmptyTree EmptyTree}
Frank {frankField = Node 'a' EmptyTree EmptyTree} :: Frank Char Tree
```

```
ghci> :t Frank {frankField = "ДА"}
Frank {frankField = "ДА"} :: Frank Char []
```

Гм-м-м… Так как поле `frankField` имеет тип вида `a b`, его значения должны иметь типы похожего вида. Например, это может быть `Just "ХА-ХА"`, тип в этом примере – `Maybe [Char]`, или `['Д','А']` (тип `[Char]`; если бы мы использовали наш собственный тип для списка, это был бы `List Char`). Мы видим, что значения типа `Frank` соответствуют сорту типа `Frank`. Сорт `[Char]` – это `*`, тип `Maybe` имеет сорт `* -> *`. Так как мы можем создать значение только конкретного типа и тип значения должен быть полностью определён, каждое значение типа `Frank` имеет сорт `*`.

Сделать для типа `Frank` экземпляр класса `Tofu` довольно просто. Мы видим, что функция `tofu` принимает значение типа `a j` (примером для типа такой формы может быть `Maybe Int`) и возвращает значение типа `t a j`. Если мы заменим тип `Frank` на `t`, результирующий тип будет `Frank Int Maybe`.

```
instance Tofu Frank where
   tofu x = Frank x
```

Проверяем типы:

```
ghci> tofu (Just 'a') :: Frank Char Maybe
Frank {frankField = Just 'a'}
ghci> tofu ["ПРИВЕТ"] :: Frank [Char] []
Frank {frankField = ["ПРИВЕТ"]}
```

Пусть и без особой практической пользы, но мы потренировали наше понимание типов. Давайте сделаем ещё несколько упражнений из тип-фу. У нас есть такой тип данных:

```
data Barry t k p = Barry { yabba :: p, dabba :: t k }
```

Ну а теперь определим для него экземпляр класса `Functor`. Класс `Functor` принимает типы сорта `* -> *`, но непохоже, что у типа `Barry` такой сорт. Каков же сорт у типа `Barry`? Мы видим, что он принимает три типа-параметра, так что его сорт будет похож на `(`*нечто* `->` *нечто* `->` *нечто* `-> *)`. Наверняка тип `p` – конкретный; он имеет сорт `*`. Для типа `k` мы предполагаем сорт `*`; следовательно, тип `t` имеет сорт `* -> *`. Теперь соединим всё в одну цепочку и получим, что тип `Barry` имеет сорт `(* -> *) -> * -> * -> *`. Давайте проверим это в интерпретаторе GHCi:

```
ghci> :k Barry
Barry :: (* -> *) -> * -> * -> *
```

Ага, мы были правы. Как приятно! Чтобы сделать для типа `Barry` экземпляр класса `Functor`, мы должны частично применить первые два параметра, после чего у нас останется сорт `* -> *`. Следовательно, начало декларации экземпляра будет таким:

```
instance Functor (Barry a b) where
```

Если бы функция `fmap` была написана специально для типа `Barry`, она бы имела тип

```
fmap :: (a -> b) -> Barry c d a -> Barry c d b
```

Здесь тип-параметр `f` просто заменён частично применённым типом `Barry c d`. Третий параметр типа `Barry` должен измениться, и мы видим, что это удобно сделать таким образом:

```
instance Functor (Barry a b) where
   fmap f (Barry {yabba = x, dabba = y}) = Barry {yabba = f x, dabba = y}
```

Готово! Мы просто отобразили тип `f` по первому полю.

В данной главе мы хорошенько изучили, как работают параметры типов, и как они формализуются с помощью сортов по аналогии с тем, как формализуются параметры функций с помощью декларации типов. Мы провели любопытные параллели между функциями и конструкторами типов, хотя на первый взгляд они и не имеют ничего общего. При реальной работе с языком Haskell обычно не приходится возиться с сортами и делать вывод сортов вручную, как мы делали в этой главе. Обычно вы просто частично применяете свой тип к сорту `* -> *` или `*` при создании экземпляра от одного из стандартных классов типов, но полезно знать, как это работает на самом деле. Также интересно, что у типов есть свои собственные маленькие типы.

Ещё раз повторю: вы не должны понимать всё, что мы сделали, в деталях, но если вы по крайней мере понимаете, как работают сорта, есть надежда на то, что вы постигли суть системы типов языка Haskell.

8
ВВОД-ВЫВОД

Разделение «чистого» и «нечистого»

В этой главе вы узнаете, как вводить данные с клавиатуры и печатать их на экран. Начнём мы с основ ввода-вывода:

- Что такое действия?
- Как действия позволяют выполнять ввод-вывод?
- Когда фактически исполняются действия?

Вводу-выводу приходится иметь дело с некоторыми ограничениями функций языка Haskell, поэтому первым делом мы обсудим, что с этим можно сделать.

Мы уже упоминали, что Haskell – чисто функциональный язык. В то время как в императивных языках вы указываете компьютеру серию шагов для достижения некой цели, в функциональном программировании мы описываем, чем является то или иное понятие. В языке Haskell функция не может изменить некоторое состояние, например поменять значение переменной (если функция изменяет состояние, мы говорим, что она имеет побочные эффекты). Единственное, что могут сделать функции в языке Haskell, – это вернуть нам некоторый результат, основываясь на переданных им параметрах. Если вызвать функцию дважды с одинаковыми параметрами, она всегда вернёт одинаковый результат. Если вы знакомы с императивными языками, может показаться, что это ограничивает свободу наших действий, но мы видели, что на самом деле это даёт весьма мощные возможности. В императивном языке у вас нет

гарантии, что простая функция, которая всего-то навсего должна обсчитать пару чисел, не сожжёт ваш дом, не похитит собаку и не поцарапает машину во время вычислений! Например, когда мы создавали бинарное поисковое дерево, то вставляли элемент в дерево не путём модификации дерева в точке вставки. Наша функция добавления нового элемента в дерево возвращала новое дерево, так как не могла изменить старое.

Конечно, это хорошо, что функции не могут изменять состояние: это помогает нам строить умозаключения о наших программах. Но есть одна проблема. Если функция не может ничего изменить, как она сообщит нам о результатах вычислений? Для того чтобы вывести результат, она должна изменить состояние устройства вывода – обычно это экран, который излучает фотоны; они путешествуют к нашему мозгу и изменяют состояние нашего сознания… вот так-то, чувак!

Но не надо отчаиваться, не всё ещё потеряно. Оказывается, в языке Haskell есть весьма умная система для работы с функциями с побочными эффектами, которая чётко разделяет чисто функциональную и «грязную» части нашей программы. «Грязная» часть выполняет всю грязную работу, например отвечает за взаимодействие с клавиатурой и экраном. Разделив «чистую» и «грязную»части, мы можем так же свободно рассуждать о чисто функциональной части нашей программы, получать все преимущества функциональной чистоты, а именно – ленивость, гибкость, модульность, и при этом эффективно взаимодействовать с внешним миром.

Привет, мир!

До сих пор для того, чтобы протестировать наши функции, мы загружали их в интерпретатор GHCi. Там же мы изучали функции из стандартной библиотеки. Но теперь, спустя семь глав,

мы наконец-то собираемся написать первую программу на языке Haskell! Ура! И, конечно же, это будет старый добрый шедевр «Привет, мир».

Итак, для начинающих: наберите в вашем любимом текстовом редакторе строку

```
main = putStrLn "Привет, мир"
```

Мы только что определили имя `main`; в нём мы вызываем функцию `putStrLn` с параметром `"Привет, мир"`. На первый взгляд, ничего необычного, но это не так: мы убедимся в этом через несколько минут. Сохраните файл как *helloworld.hs*.

Сейчас мы собираемся сделать то, чего ещё не пробовали делать. Мы собираемся скомпилировать нашу программу! Я даже разволновался!.. Откройте ваш терминал, перейдите в папку с сохранённым файлом *helloworld.hs* и выполните следующую команду:

```
$ ghc helloworld
[1 of 1] Compiling Main    ( helloworld.hs, helloworld.o )
Linking helloworld ...
```

О'кей! При некотором везении вы получите нечто похожее и теперь можете запустить свою программу, вызвав `./helloworld`.

```
$ ./helloworld
Привет, мир
```

ПРИМЕЧАНИЕ. *Если вы используете Windows, то вместо выполнения команды* `./helloworld` *просто запустите файл helloworld.exe.*

Ну вот и наша первая программа, которая печатает что-то на терминале! Банально до невероятности!

Давайте изучим более подробно, что же мы написали. Сначала посмотрим на тип функции `putStrLn`:

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t putStrLn "Привет, мир"
putStrLn "Привет, мир" :: IO ()
```

Тип `putStrLn` можно прочесть таким образом: `putStrLn` принимает строку и возвращает действие ввода-вывода (*I/O action*) с результирующим типом `()` (это пустой кортеж). Действие ввода-вывода – это нечто вызывающее побочные эффекты при выполнении (обычно чтение входных данных или печать на экране); также действие может возвращать некоторые значения. Печать строки на экране не имеет какого-либо значимого результата, поэтому возвращается значение `()`.

ПРИМЕЧАНИЕ. *Пустой кортеж имеет значение* `()`*, его тип – также* `()`.

Когда будет выполнено действие ввода-вывода? Вот для чего нужна функция `main`. Операции ввода-вывода выполняются, если мы поместим их в функцию `main` и запустим нашу программу.

Объединение действий ввода-вывода

Возможность поместить в программу всего один оператор ввода-вывода не очень-то вдохновляет. Но мы можем использовать ключевое слово `do` для того, чтобы «склеить» несколько операторов ввода-вывода в один. Рассмотрим пример:

```
main = do
   putStrLn "Привет, как тебя зовут?"
   name <- getLine
   putStrLn ("Привет, " ++ name ++ ", ну ты и хипстота!")
```

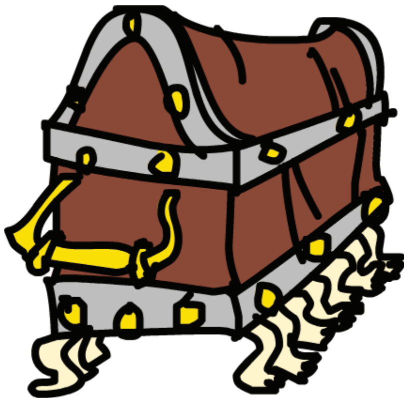
О, новый синтаксис!.. И он похож на синтаксис императивных языков. Если откомпилировать и запустить эту программу, она будет работать так, как вы и предполагаете. Обратите внимание: мы записали ключевое слово `do` и затем последовательность шагов, как сделали бы в императивном языке. Каждый из этих шагов – действие ввода-вывода. Расположив их рядом с помощью ключевого слова `do`, мы свели их в одно действие ввода-вывода. Получившееся действие имеет тип `IO()`; это тип последнего оператора в цепочке.

По этой причине функция `main` всегда имеет тип `main :: IO` <*нечто*>, где <*нечто*> – некоторый конкретный тип. По общепринятому соглашению обычно не пишут декларацию типа для функции `main`.

В третьей строке можно видеть ещё один не встречавшийся нам ранее элемент синтаксиса, `name <- getLine`. Создаётся впечатление, будто считанная со стандартного входа строка сохраняется в переменной с именем `name`. Так ли это на самом деле? Давайте посмотрим на тип `getLine`.

```
ghci> :t getLine
getLine :: IO String
```

Ага!.. Функция `getLine` – действие ввода-вывода, которое содержит результирующий тип – строку. Это понятно: действие ждёт, пока пользователь не введёт что-нибудь с терминала, и затем это нечто будет представлено как строка. Что тогда делает выражение



`name <- getLine`? Можно прочитать его так: «выполнить действие `getLine` и затем связать результат выполнения с именем `name`». Функция `getLine` имеет тип `IO String`, поэтому образец `name` будет иметь тип `String`. Можно представить действие ввода-вывода в виде ящика с ножками, который ходит в реальный мир, что-то в нём делает (рисует граффити на стене, например) и иногда приносит обратно какие-либо данные. Если ящик что-либо принёс, единственный способ открыть его и извлечь данные – использовать конструкцию с символом <–. Получить данные из действия ввода-вывода можно только внутри другого действия ввода-вывода. Таким образом, язык Haskell чётко разделяет чистую и «грязную» части кода. Функция `getLine` – не чистая функция, потому что её результат может быть неодинаковым при последовательных вызовах. Вот почему она как бы «запачкана» конструктором типов `IO`, и мы можем получить данные только внутри действий ввода-вывода,

имеющих в сигнатуре типа маркер IO. Так как код для ввода-вывода также «испачкан», любое вычисление, зависящее от «испачканных» IO-данных, также будет давать «грязный»результат.

Если я говорю «испачканы», это не значит, что мы не сможем использовать результат, содержащийся в типе IO в чистом коде. Мы временно «очищаем» данные внутри действия, когда связываем их с именем. В выражении name <- getLine образец name содержит обычную строку, представляющую содержимое ящика.

Мы можем написать сложную функцию, которая, скажем, принимает ваше имя как параметр (обычная строка) и предсказывает вашу удачливость или будущее всей вашей жизни, основываясь на имени:

```
main = do
   putStrLn "Привет, как тебя зовут?"
   name <- getLine
   putStrLn $ "Вот твоё будущее: " ++ tellFortune name
```

Функция tellFortune (или любая другая, которой мы передаём значение name) не должна знать ничего про IO – это обычная функция String -> String.

Посмотрите на этот образец кода. Корректен ли он?

```
nameTag = "Привет, меня зовут " ++ getLine
```

Если вы ответили «нет», возьмите с полки пирожок. Если ответили «да», убейте себя об стену... Шучу, не надо! Это выражение не сработает, потому что оператор ++ требует, чтобы оба параметра были списками одинакового типа. Левый параметр имеет тип String (или [Char], если вам угодно), в то время как функция getLine возвращает значение типа IO String. Вы не сможете конкатенировать строку и результат действия ввода-вывода. Для начала нам нужно извлечь результат из действия ввода-вывода, чтобы получить значение типа String, и единственный способ сделать это – выполнить что-то вроде name <- getLine внутри другого действия ввода-вывода. Если мы хотим работать с «нечистыми» данными, то должны делать это в «нечистом» окружении!... Итак, грязь от нечистоты распространяется как моровое поветрие, и в наших интересах делать часть для осуществления ввода-вывода настолько малой, насколько это возможно.

Каждое выполненное действие ввода-вывода заключает в себе результат. Вот почему наш предыдущий пример можно переписать так:

```
main = do
   foo <- putStrLn "Привет, как тебя зовут?"
   name <- getLine
   putStrLn ("Привет, " ++ name ++ ", ну ты и хипстота!")
```

Тем не менее образец `foo` всегда будет получать значение `()`, так что большого смысла в этом нет. Заметьте: мы не связываем последний вызов функции `putStrLn` с именем, потому что в блоке `do` последний оператор, в отличие от предыдущих, не может быть связан с именем. Мы узнаем причины такого поведения немного позднее, когда познакомимся с миром монад. До тех пор можно считать, что блок `do` автоматически получает результат последнего оператора и возвращает его в качестве собственного результата.

За исключением последней строчки, каждая строка в блоке `do` может быть использована для связывания. Например, `putStrLn "ЛЯ"` может быть записана как `_ <- putStrLn "ЛЯ"`. Но в этом нет никакого смысла, так что мы опускаем `<-` для действий ввода-вывода, не возвращающих значимого результата.

Иногда начинающие думают, что вызов

```
myLine = getLine
```

считает значение со стандартного входа и затем свяжет это значение с именем `myLine`. На самом деле это не так. Такая запись даст функции `getLine` другое синонимичное имя, в данном случае – `myLine`. Запомните: чтобы получить значение из действия ввода-вывода, вы должны выполнять его внутри другого действия ввода-вывода и связывать его с именем при помощи символа `<-`.

Действие ввода-вывода будет выполнено, только если его имя `main` или если оно помещено в составное действие с помощью блока `do`. Также мы можем использовать блок `do` для того, чтобы «склеить» несколько действий ввода-вывода в одно. Затем можно будет использовать его в другом блоке `do` и т. д. В любом случае действие будет выполнено, только если оно каким-либо образом вызывается из функции `main`.

Ах, да, есть ещё один способ выполнить действие ввода-вывода! Если напечатать его в интерпретаторе GHCi и нажать клавишу **Enter**, действие выполнится.

```
gchi> putStrLn "При-и-и-вет"
При-и-и-вет
```

Даже если мы просто наберём некоторое число или вызовем некоторую функцию в GHCi и нажмём **Enter**, интерпретатор GHCi вычислит значение, затем вызовет для него функцию `show`, чтобы получить строку, и напечатает строку на терминале, используя функцию `putStrLn`.

Использование ключевого слова let внутри блока do

Помните связывания при помощи ключевого слова `let`? Если уже подзабыли, освежите свои знания. Связывания должны быть такого вида: `let` *<определения>* `in` *<выражение>*, где *<определения>* – это имена, даваемые выражениям, а *<выражение>* использует имена из *<определений>*. Также мы говорили, что в списковых выражениях часть `in` не нужна. Так вот, в блоках `do` можно использовать выражение `let` таким же образом, как и в списковых выражениях. Смотрите:

```
import Data.Char

main = do
   putStrLn "Ваше имя?"
   firstName <- getLine
   putStrLn "Ваша фамилия?"
   lastName <- getLine
   let bigFirstName = map toUpper firstName
       bigLastName = map toUpper lastName
   putStrLn $ "Привет, "  ++ bigFirstName ++ " "
                          ++ bigLastName
                          ++ ", как дела?"
```

Видите, как выровнены операторы действий ввода-вывода в блоке `do`? Обратите внимание и на то, как выровнено выражение `let` по отношению к действиям ввода-вывода и как выровнены образцы внутри выражения `let`. Это хороший пример, потому что выравнивание текста очень важно в языке Haskell. Далее мы записали вызов `map toUpper firstName`, что превратит, например, `"Иван"` в намного более солидное `"ИВАН"`. Мы связали эту строку в верхнем регистре с именем, которое использовали в дальнейшем при выводе на терминал.

Вам может быть непонятно, когда использовать символ `<-`, а когда выражение `let`. Запомните: символ `<-` (в случае действий ввода-вывода) используется для выполнения действий ввода-вывода и связывания результатов с именами. Выражение `map toUpper firstName` не является действием ввода-вывода – это чистое выражение. Соответственно, используйте символ `<-` для связывания результатов действий ввода-вывода с именами, а выражение `let` – для связывания имён с чистыми значениями. Если бы мы выполнили что-то вроде `let firstName = getLine`, то просто создали бы синоним функции `getLine`, для которого значение всё равно должно получаться с помощью символа `<-`.

Обращение строк

Теперь напишем программу, которая будет считывать строки, переставлять в обратном порядке буквы в словах и распечатывать их. Выполнение программы прекращается при вводе пустой строки. Итак:

```
main = do
   line <- getLine
   if null line
       then return ()
       else do
          putStrLn $ reverseWords line
          main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

Чтобы лучше понять, как работает программа, сохраните её в файле *reverse.hs*, скомпилируйте и запустите:

```
$ ghc reverse.hs
[1 of 1] Compiling Main  ( reverse.hs, reverse.o )
Linking reverse ...
$ ./reverse
уберитесь в проходе номер 9
ьсетиребу в едохорп ремон 9
козёл ошибки осветит твою жизнь
лёзок икбишо титевсо юовт ьнзиж
но это всё мечты
он отэ ёсв ытчем
```

Для начала посмотрим на функцию `reverseWords`. Это обычная функция, которая принимает строку, например `"эй ты мужик"`, и вызывает функцию `words`, чтобы получить список слов `["эй", "ты", "мужик"]`. Затем мы применяем функцию `reverse` к каждому элементу списка, получаем `["йэ", "ыт", "кижум"]` и помещаем результат обратно в строку, используя функцию `unwords`. Конечным результатом будет `"йэ ыт кижум"`.

Теперь посмотрим на функцию `main`. Сначала мы получаем строку с терминала с помощью функции `getLine`. Далее у нас имеется условное выражение. Запомните, что в языке Haskell каждое ключевое слово `if` должно сопровождаться секцией `else`, так как каждое выражение должно иметь некоторое значение. Наш оператор записан так, что если условие истинно (в нашем случае – когда введут пустую строку), мы выполним одно действие ввода-вывода; если оно ложно – выполним действие ввода-вывода из секции `else`. По той же причине в блоке `do` условные операторы `if` должны иметь вид `if` *<условие>* `then` *<действие ввода-вывода>* `else` *<действие ввода-вывода>*.

Вначале посмотрим, что делается в секции `else`. Поскольку можно поместить только одно действие ввода-вывода после ключевого слова `else`, мы используем блок `do` для того, чтобы «склеить» несколько операторов в один. Эту часть можно было бы написать так:

```
else (do
    putStrLn $ reverseWords line
    main)
```

Подобная запись явно показывает, что блок `do` может рассматриваться как одно действие ввода-вывода, но и выглядит она не очень красиво. В любом случае внутри блока `do` мы можем вызвать функцию `reverseWords` со строкой – результатом действия `getLine` и распечатать результат. После этого мы выполняем функцию `main`. Получается, что функция `main` вызывается рекурсивно, и в этом нет ничего необычного, так как сама по себе функция `main` – тоже действие ввода-вывода. Таким образом, мы возвращаемся к началу программы в следующей рекурсивной итерации.

Ну а что случится, если мы получим на вход пустую строку? В этом случае выполнится часть после ключевого слова `then`. То есть выполнится выражение `return ()`. Если вам приходилось писать на импера-

тивных языках вроде C, Java или на Python, вы наверняка уверены, что знаете, как работает функция `return` – и, возможно, у вас возникнет искушение пропустить эту часть текста. Но не стоит спешить: функция `return` в языке Haskell работает совершенно не так, как в большинстве других языков! Её название сбивает с толку, но на самом деле она довольно сильно отличается от своих «тёзок». В императивных языках ключевое слово `return` обычно прекращает выполнение метода или процедуры и возвращает некоторое значение вызывающему коду. В языке Haskell (и особенно в действиях ввода-вывода) одноимённая функция создаёт действие ввода-вывода из чистого значения. Если продолжать аналогию с коробками, она берёт значение и помещает его в «коробочку». Получившееся в результате действие ввода-вывода на самом деле не выполняет никаких действий – оно просто инкапсулирует некоторое значение. Таким образом, в контексте системы ввода-вывода `return "ха-ха"` будет иметь тип `IO String`. Какой смысл преобразовывать чистое значение в действие ввода-вывода, которое ничего не делает? Зачем «пачкать» нашу программу больше необходимого? Нам нужно некоторое действие ввода-вывода для второй части условного оператора, чтобы обработать случай пустой строки. Вот для чего мы создали фиктивное действие ввода-вывода, которое ничего не делает, записав `return ()`.

Вызов функции `return` не прекращает выполнение блока `do` – ничего подобного! Например, следующая программа успешно выполнится вся до последней строчки:

```
main = do
   return ()
   return "ХА-ХА-ХА"
   line <- getLine
   return "ЛЯ-ЛЯ-ЛЯ"
   return 4
   putStrLn line
```

Всё, что делает функция `return`, – создаёт действия ввода-вывода, которые не делают ничего, кроме как содержат значения, и все они отбрасываются, поскольку не привязаны к образцам. Мы можем использовать функцию `return` вместе с символом `<-` для того, чтобы связывать значения с образцами.

```
main = do
   let a = "ад"
       b = "да!"
   putStrLn $ a ++ " " ++ b
```

Как вы можете видеть, функция `return` выполняет обратную операцию по отношению к операции `<-`. В то время как функция `return` принимает значение и помещает его в «коробку», операция `<-` принимает (и исполняет) «коробку», а затем привязывает полученное из неё значение к имени. Но всё это выглядит лишним, так как в блоках `do` можно использовать выражение `let` для привязки к именам, например так:

```
main = do
   let a = "hell"
       b = "yeah"
   putStrLn $ a ++ " " ++ b
```

При работе с блоками `do` мы чаще всего используем функцию `return` либо для создания действия ввода-вывода, которое ничего не делает, либо для того, чтобы блок `do` возвращал нужное нам значение, а не результат последнего действия ввода-вывода. Во втором случае мы используем функцию `return`, чтобы создать действие ввода-вывода, которое будет всегда возвращать нужное нам значение, и эта функция `return` должна находиться в самом конце блока `do`.

Некоторые полезные функции для ввода-вывода

В стандартной библиотеке языка Haskell имеется масса полезных функций и действий ввода-вывода. Давайте рассмотрим некоторые из них и увидим, как ими пользоваться.

Функция putStr

Функция `putStr` похожа на функцию `putStrLn` – она принимает строку как параметр и возвращает действие ввода-вывода, которое печатает строку на терминале. Единственное отличие: функция `putStr` не выполняет перевод на новую строку после печати, как это делает `putStrLn`.

```
main = do
   putStr "Привет, "
   putStr "я "
   putStrLn "Энди!"
```

Если мы скомпилируем эту программу, то при запуске получим:

```
Привет, я Энди!
```

Функция putChar

Функция `putChar` принимает символ и возвращает действие ввода-вывода, которое напечатает его на терминале.

```
main = do
   putChar 'А'
   putChar 'Б'
   putChar 'В'
```

Функция `putStr` определена рекурсивно с помощью функции `putChar`. Базовый случай для функции `putStr` – это пустая строка. Если печатаемая строка пуста, функция возвращает пустое действие ввода-вывода, то есть `return ()`. Если строка не пуста, функция выводит на терминал первый символ этой строки, вызывая функцию `putChar`, а затем выводит остальные символы, снова рекурсивно вызывая саму себя.

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do
    putChar x
    putStr xs
```

Как вы заметили, мы можем использовать рекурсию в системе ввода-вывода подобно тому, как делаем это в чистом коде. Точно так же образом мы определяем базовые случаи, а затем думаем, что будет результатом. В результате мы получим действие, которое выведет первый символ, а затем остаток строки.

Функция print

Функция `print` принимает значение любого типа – экземпляра класса `Show` (то есть мы знаем, как представить значение этого типа

в виде строки), вызывает функцию show, чтобы получить из данного значения строку, и затем выводит её на экран. По сути, это putStrLn.show. Это выражение сначала вызывает функцию show на переданном параметре, а затем «скармливает» результат функции putStrLn, которая возвращает действие ввода-вывода; оно, в свою очередь, печатает заданное значение.

```
main = do
   print True
   print 2
   print "ха-ха"
   print 3.2
   print [3,4,3]
```

После компиляции и запуска получаем:

```
True
2
"ха-ха"
3.2
[3,4,3]
```

Как вы могли заметить, это очень полезная функция. Помните, мы говорили о том, что действия ввода-вывода выполняются только из функции main или когда мы выполняем их в интерпретаторе GHCi? После того как мы напечатаем значение (например, 3 или [1, 2, 3]) и нажмём клавишу «Ввод», интерпретатор GHCi вызовет функцию print с введённым значением для вывода на терминал!

```
ghci> 3
3
ghci> print 3
3
ghci> map (++"!") ["хей","хо","ууу"]
["хей!","хо!","ууу!"]
ghci> print $ map (++"!") ["хей","хо","ууу"]
["хей!","хо!","ууу!"]
```

Как правило, мы хотим видеть строку на экране, не заключённую в кавычки, поэтому для печати строк обычно используется функция putStrLn. Но для печати значений других типов преимущественно используется функция print.

Функция when

Функция `when` находится в модуле `Control.Monad` (чтобы к ней обратиться, воспользуйтесь `import Control.Monad`). Она интересна, потому что выглядит как оператор управления ходом вычислений, но на самом деле это обычная функция. Она принимает булевское значение и действие ввода-вывода. Если булевское значение истинно, она возвращает второй параметр – действие ввода-вывода. Если первый параметр ложен, функция возвращает `return ()`, то есть пустое действие.

Напишем программу, которая запрашивает строку текста и, если строка равна «РЫБА-МЕЧ», печатает её:

```
import Control.Monad

main = do
   input <- getLine
   when (input == "РЫБА-МЕЧ") $ do
      putStrLn input
```

Без `when` нам понадобилось бы написать нечто такое:

```
main = do
   input <- getLine
   if (input == "РЫБА-МЕЧ")
      then putStrLn input
      else return ()
```

Как вы видите, функция `when` позволяет выполнить заданное действие в случае, если некоторое условие истинно, и ничего не делать в противном случае.

Функция sequence

Функция `sequence` принимает список действий ввода-вывода и возвращает одно действие ввода-вывода, последовательно выполняющее действия из списка. Результат выполнения этого действия – список результатов вложенных действий. Сигнатура типа функции: `sequence :: [IO a] -> IO [a]`. Выполним следующее:

```
main = do
   a <- getLine
```

```
   b <- getLine
   c <- getLine
   print [a,b,c]
```

То же самое, но с использованием функции `sequence`:

```
main = do
   rs <- sequence [getLine, getLine, getLine]
   print rs
```

Итак, выражение `sequence [getLine, getLine, getLine]` создаст действие ввода-вывода, которое выполнит функцию `getLine` три раза. Если мы свяжем это действие с именем, результат будет представлять собой список результатов действий из изначального списка, в нашем случае – то, что пользователь введёт с клавиатуры.

Функция `sequence` обычно используется, если мы хотим пройтись по списку функциями `print` или `putStrLn`. Вызов `map print [1,2,3,4]` не создаёт действия ввода-вывода – вместо этого создаётся список действий. Такой код на самом деле эквивалентен следующему:

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

Если мы хотим преобразовать список действий в действие, то необходимо воспользоваться функцией `sequence`:

```
ghci> sequence $ map print [1,2,3,4]
1
2
3
4
[(),(),(),()]
```

Но что это за `[(),(),(),()]` в конце вывода? При выполнении в GHCi действия ввода-вывода помимо самого действия выводится результат выполнения, но только если этот результат не есть `()`. Поэтому при выполнении в GHCi `putStrLn "ха-ха"` просто выводится строка – результатом является `()`. Если же попробовать ввести `getLine`, то помимо собственно ввода с клавиатуры будет выведено введённое значение – результатом является `IO String`.

Функция mapM

Поскольку применение функции, возвращающей действие ввода-вывода, к элементам списка и последующее выполнение всех полу-

ченных действий очень распространено, для этих целей были введены две вспомогательные функции – `mapM` и `mapM_`. Функция `mapM` принимает функцию и список, применяет функцию к элементам списка, сводит элементы в одно действие ввода-вывода и выполняет их. Функция `mapM_` работает так же, но отбрасывает результат действия ввода-вывода. Она используется, когда нам не важен результат комбинированного действия ввода-вывода.

```
ghci> mapM print [1,2,3]
1
2
3
[(),(),()]
ghci> mapM_ print [1,2,3]
1
2
3
```

Функция forever

Функция `forever` принимает действие ввода-вывода – параметр и возвращает действие ввода-вывода – результат. Действие-результат будет повторять действие-параметр вечно. Эта функция входит в модуль `Control.Monad`. Следующая программа будет бесконечно спрашивать у пользователя строку и возвращать её в верхнем регистре:

```
import Control.Monad
import Data.Char

main = forever $ do
   putStr "Введите что-нибудь: "
   l <- getLine
   putStrLn $ map toUpper l
```

Функция forM

Функция `forM` (определена в модуле `Control.Monad`) похожа на функцию `mapM`, но её параметры поменяны местами. Первый параметр – это список, второй – это функция, которую надо применить к списку и затем свести действия из списка в одно действие. Для чего

это придумано? Если творчески использовать лямбда-выражения и ключевое слово do, можно проделывать такие фокусы:

```
import Control.Monad

main = do
   colors <- forM [1,2,3,4] (\a -> do
      putStrLn $ "С каким цветом ассоциируется число "
                 ++ show a ++ "?"
      color <- getLine
      return color)
   putStrLn "Цвета, ассоциирующиеся с 1, 2, 3 и 4: "
   mapM putStrLn colors
```

Вот что мы получим при запуске:

```
С каким цветом ассоциируется число 1?
белый
С каким цветом ассоциируется число 2?
синий
С каким цветом ассоциируется число 3?
красный
С каким цветом ассоциируется число 4?
оранжевый
Цвета, ассоциирующиеся с 1, 2, 3 и 4:
белый
синий
красный
оранжевый
```

Анонимная функция (\a -> do ...) – это функция, которая принимает число и возвращает действие ввода-вывода. Нам пришлось поместить её в скобки, иначе анонимная функция решит, что следующие два действия ввода-вывода принадлежат ей. Обратите внимание, что мы производим вызов return color внутри блока do. Это делается для того, чтобы действие ввода-вывода, возвращаемое блоком do, содержало в себе цвет. На самом деле мы не обязаны этого делать, потому что функция getLine уже содержит цвет внутри себя. Выполняя color <- getLine и затем return color, мы распаковываем результат getLine и затем запаковываем его обратно, то есть это то же самое, что просто вызвать функцию getLine. Функция forM (вызываемая с двумя параметрами) создаёт действие ввода-вывода, результат

которого мы связываем с идентификатором `colors`. Этот идентификатор – обычный список, содержащий строки. В конце мы распечатываем все цвета, вызывая выражение `mapM putStrLn colors`.

Вы можете думать, что функция `forM` имеет следующий смысл: «Создай действие ввода-вывода для каждого элемента в списке. Каков будет результат каждого такого действия, может зависеть от элемента, из которого оно создаётся. После создания списка действий исполни их и привяжи их результаты к чему-либо». Однако мы не обязаны их связывать – результаты можно просто отбросить.

На самом деле мы могли бы сделать это без использования функции `forM`, но так легче читается. Обычно эта функция используется, когда нам нужно отобразить `(map)` и объединить `(sequence)` действия, которые мы тут же определяем в секции `do`. Таким образом, мы могли бы заменить последнюю строку на выражение `forM colors putStrLn`.

Обзор системы ввода-вывода

В этой главе мы изучили основы системы ввода-вывода языка Haskell. Также мы узнали, что такое действия ввода-вывода, как они позволяют выполнять ввод-вывод, в какой момент они выполняются. Итак, повторим пройденное: действия ввода-вывода – это значения, такие же, как любые другие в языке Haskell. Мы можем передать их в функции как параметры, функции могут возвращать действия ввода-вывода в качестве результата. Они отличаются тем, что если они попадут в функцию `main` (или их введут в интерпретаторе GHCi), то будут выполнены. В этот момент они могут выводить что-либо на экран или управлять звуковыводящим устройством. Каждое действие ввода-вывода может содержать результат общения с реальным миром.

Не думайте о функции, например о `putStrLn`, как о функции, которая принимает строку и печатает её на экране. Думайте о ней как о функции, которая принимает строку и возвращает действие ввода-вывода. Это действие при выполнении печатает нечто ценное на вашем терминале.

9

БОЛЬШЕ ВВОДА И ВЫВОДА

Теперь, когда вы понимаете идеи, лежащие в основе ввода-вывода в языке Haskell, можно приступать к интересным штукам. В этой главе мы будем обрабатывать файлы, генерировать случайные числа, читать аргументы командной строки и много чего ещё. Будьте готовы!

Файлы и потоки

Вооружившись знанием того, как работают действия ввода-вывода, можно перейти к чтению и записи файлов. Но прежде давайте посмотрим, как Haskell умеет работать с потоками данных. *Потоком* называется последовательность фрагментов данных, которые поступают на вход программы и выводятся в результате её работы. Например, когда вы вводите в программу символы, печатая их на клавиатуре, последовательность этих символов может рассматриваться как поток.

Перенаправление ввода

Многие интерактивные программы получают пользовательский ввод с клавиатуры. Однако зачастую гораздо удобнее «скормить» программе содержимое текстового файла. Такой способ подачи входных данных называется *перенаправлением ввода*.

Посмотрим, как перенаправление ввода работает с программой на языке Haskell. Для начала создадим текстовый файл, содержащий небольшое хайку, и сохраним его под именем *haiku.txt*:

```
Я маленький чайник
Ох уж этот обед в самолёте
Он столь мал и невкусен
```

Ну да, хайку, прямо скажем, не шедевр – и что? Если кто в курсе, где найти хороший учебник по хайку, дайте знать.

Теперь напишем маленькую программу, которая непрерывно читает строку ввода и выводит её в верхнем регистре:

```
import Control.Monad
import Data.Char

main = forever $ do
   l <- getLine
   putStrLn $ map toUpper l
```

Сохраните эту программу в файле *capslocker.hs* и скомпилируйте её.

Вместо того чтобы вводить строки с клавиатуры, мы перенаправим на вход программы содержимое файла *haiku.txt*. Чтобы сделать это, нужно добавить символ < после имени программы и затем указать имя файла, в котором хранятся исходные данные. Посмотрите:

```
$ ghc capslocker
[1 of 1] Compiling Main   ( capslocker.hs, capslocker.o )
Linking capslocker ...
$ ./capslocker < haiku.txt
Я МАЛЕНЬКИЙ ЧАЙНИК
ОХ УЖ ЭТОТ ОБЕД В САМОЛЁТЕ
ОН СТОЛЬ МАЛ И НЕВКУСЕН
capslocker: <stdin>: hGetLine: end of file
```

То, что мы проделали, практически эквивалентно запуску программы `capslocker`, вводу нашего хайку с клавиатуры и передаче символа конца файла (обычно это делается нажатием клавиш **Ctrl+D**). С тем же успехом можно было бы запустить `capslocker` и сказать: «Погоди, не читай ничего с клавиатуры, возьми содержимое этого файла!».

Получение строк из входного потока

Давайте посмотрим на действие ввода-вывода `getContents`, упрощающее обработку входного потока за счёт того, что оно позволяет рассматривать весь поток как обычную строку. Действие `getContents` читает всё содержимое стандартного потока ввода вплоть до обнаружения символа конца файла. Его тип: `getContents :: IO String`. Самое приятное в этом действии то, что ввод-вывод в его исполнении является ленивым. Это означает, что выполнение `foo <- getContents` не приводит к загрузке в память всего содержимого потока и связыванию его с именем `foo`. Нет, действие `getContents` для этого слишком лениво.Оно скажет: «Да, да, я прочту входные данные с терминала как-нибудь потом, когда это действительно понадобится!».

В примере *capslocker.hs* для чтения ввода строка за строкой и печати их в верхнем регистре использовалась функция `forever`. Если мы перейдём на `getContents`, то она возьмёт на себя все заботы о деталях ввода-вывода– о том, когда и какую часть входных данных нужно прочитать. Поскольку наша программа просто берёт входные данные, преобразует их и выводит результат, пользуясь `getContents`, её можно написать короче:

```
import Data.Char

main = do
   contents <- getContents
   putStr $ map toUpper contents
```

Мы выполняем действие `getContents` и даём имя `contents` строке, которую она прочтёт. Затем проходим функцией `toUpper` по всем символам этой строки и выводим результат на терминал. Имейте в виду: поскольку строки являются списками, а списки ленивы, как и действие `getContents`, программа не будет пытаться прочесть и сохранить в памяти всё содержимое входного потока. Вместо этого

она будет читать данные порциями, переводить каждую порцию в верхний регистр и печатать результат.

Давайте проверим:

```
$ ./capslocker < haiku.txt
Я МАЛЕНЬКИЙ ЧАЙНИК
ОХ УЖ ЭТОТ ОБЕД В САМОЛЁТЕ
ОН СТОЛЬ МАЛ И НЕВКУСЕН
```

Работает. А что если мы просто запустим *capslocker* и будем печатать строки вручную (для выхода из программы нужно нажать **Ctrl+D**)?

```
$ ./capslocker
хей хо
ХЕЙ ХО
идём
ИДЁМ
```

Чудесно! Как видите, программа печатает строки в верхнем регистре по мере ввода строк. Когда результат действия `getContents` связывается с идентификатором `contents`, он представляется в памяти не в виде настоящей строки, но в виде обещания, что рано или поздно он вернёт строку. Также есть обещание применить функцию `toUpper` ко всем символам строки `contents`. Когда выполняется функция `putStr`, она говорит предыдущему обещанию: «Эй, мне нужна строка в верхнем регистре!». Поскольку никакой строки ещё нет, она говорит идентификатору `contents`: «Аллё, а не считать ли строку с терминала?». Вот тогда функция `getContents` в самом деле считывает с терминала и передаёт строку коду, который её запрашивал, чтобы сделать что-нибудь осязаемое. Затем этот код применяет функцию `toUpper` к символам строки и отдаёт результат в функцию `putStr`, которая его печатает. После чего функция `putStr` говорит, «Ау, мне нужна следующая строка, шевелись!» – и так продолжается до тех пор, пока не закончатся строки на входе, что мы обозначаем символом конца файла.

Теперь давайте напишем программу, которая будет принимать некоторый вход и печатать только те строки, длина которых меньше 15 символов. Смотрим:

```
main = do
   contents <- getContents
   putStr $ shortLinesOnly contents

shortLinesOnly :: String -> String
shortLinesOnly = unlines . filter (\line -> length line < 15) . lines
```

Фрагмент программы, ответственный за ввод-вывод, сделан настолько малым, насколько это вообще возможно. Так как предполагается, что наша программа печатает результат, основываясь на входных данных, её можно реализовать согласно следующей логике: читаем содержимое входного потока, запускаем на этом содержимом некоторую функцию, печатаем результат работы этой функции.

Функция `shortLinesOnly` принимает строку – например, такую: `"коротко\nдлииииииииииииинно\nкоротко"`. В этом примере в строке на самом деле три строки входных данных: две короткие и одна (посередине) длинная. В результате применения функции `lines` получаем список `["коротко", "длииииииииииииинно", "коротко"]`. Затем список строк фильтруется, и остаются только строки, длина которых меньше 15 символов: `["коротко", "коротко"]`. Наконец, функция `unlines` соединяет элементы списка в одну строку, разделяя их символом перевода строки: `"коротко\nкоротко"`.

Попробуем проверить, что получилось. Сохраните этот текст в файле *shortlines.txt*:

```
Я короткая
И я
А я длииииииинная!!!
А уж я-то какая длиннющая!!!!!!!
Коротенькая
Длииииииииииииииииииииииинная
Короткая
```

Сохраните программу в файле *shortlinesonly.hs* и скомпилируйте её:

```
$ ghc shortlinesonly.hs
[1 of 1] Compiling Main   ( shortlinesonly.hs, shortlinesonly.o )
Linking shortlinesonly ...
```

Чтобы её протестировать, перенаправим содержимое файла *shortlines.txt* на её поток ввода:

```
$ ./shortlinesonly < shortlines.txt
Я короткая
И я
Коротенькая
Короткая
```

Видно, что на терминал выведены только короткие строки.

Преобразование входного потока

Подобная последовательность действий – считывание строки из потока ввода, преобразование её функцией и вывод результата – настолько часто встречается, что существует функция, которая делает эту задачу ещё легче; она называется `interact`. Функция `interact` принимает функцию типа `String -> String` как параметр и возвращает действие ввода-вывода, которое примет некоторый вход, запустит заданную функцию и распечатает результат. Давайте изменим нашу программу так, чтобы воспользоваться этой функцией:

```
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly = unlines . filter (\line -> length line < 15) . lines
```

Этой программой можно пользоваться, либо перенаправляя файл в поток ввода, либо вводя данные непосредственно с клавиатуры, строка за строкой. Результат будет одинаковым, однако при вводе с клавиатуры входные данные будут чередоваться с выходными.

Давайте напишем программу, которая постоянно считывает строку и затем говорит нам, является ли введённая строка палиндромом. Можно было бы использовать функцию `getLine`, чтобы она считывала строку, затем говорить пользователю, является ли она палиндромом, и снова запускать функцию `main`. Но легче делать это с помощью функции `interact`. Когда вы её используете, всегда думайте, как преобразовать некий вход в желаемый выход. В нашем случае мы хотим заменить строку на входе на `"палиндром"` или `"не палиндром"`.

```
respondPalindromes :: String -> String
respondPalindromes =
   unlines .
   map (\xs -> if isPal xs then "палиндром" else "не палиндром") .
   lines

isPal xs = xs == reverse xs
```

Всё вполне очевидно. Вначале преобразуем строку, например

```
"слон\nпотоп\nчто-нибудь"
```

в список строк

```
["слон", "потоп", "что-нибудь"]
```

Затем применяем анонимную функцию к элементам списка и получаем:

```
["не палиндром", "палиндром", "не палиндром"]
```

Соединяем список обратно в строку функцией `unlines`. Теперь мы можем определить главное действие ввода-вывода:

```
main = interact respondPalindromes
```

Протестируем:

```
$ ./palindromes
ха-ха
не палиндром
арозаупаланалапуазора
палиндром
печенька
не палиндром
```

Хоть мы и написали программу, которая преобразует одну большую составную строку в другую составную строку, она работает так, как будто мы обрабатываем строку за строкой. Это потому что язык Haskell ленив – он хочет распечатать первую строку результата, но не может, поскольку пока не имеет первой строки ввода. Как только мы введем первую строку на вход, он напечатает первую строку на выходе. Мы выходим из программы по символу конца файла.

Также можно запустить нашу программу, перенаправив в неё содержимое файла. Например, у нас есть файл *words.txt*:

```
кенгуру
радар
ротор
мадам
```

Вот что мы получим, если перенаправим его на вход нашей программы:

```
$ ./palindromes < words.txt
не палиндром
палиндром
палиндром
палиндром
```

Ещё раз: результат аналогичен тому, как если бы мы запускали программу и вводили слова вручную. Здесь мы не видим входных строк, потому что вход берётся из файла, а не со стандартного ввода.

К этому моменту, вероятно, вы уже усвоили, как работает ленивый ввод-вывод и как его можно использовать с пользой для себя. Вы можете рассуждать о том, каким должен быть выход для данного входа, и писать функцию для преобразования входа в выход. В ленивом вводе-выводе ничего не считывается со входа до тех пор, пока это не станет абсолютно необходимым для того, что мы собираемся напечатать.

Чтение и запись файлов

До сих пор мы работали с вводом-выводом, печатая на терминале и считывая с него. Ну а как читать и записывать файлы? В некотором смысле мы уже работали с файлами. Чтение с терминала можно представить как чтение из специального файла. То же верно и для печати на терминале – это почти что запись в файл. Два файла – `stdin` и `stdout` – обозначают, соответственно, стандартный ввод и вывод. Принимая это во внимание, мы увидим, что запись и чтение из файлов очень похожи на запись в стандартный вывод и чтение со стандартного входа.

Для начала напишем очень простую программу, которая открывает файл с именем *girlfriend.txt* и печатает его на терминале. В этом

файле записаны слова лучшего хита Аврил Лавин, «Girlfriend». Вот содержимое *girlfriend.txt*:

```
Эй! Ты! Эй! Ты!
Мне не нравится твоя подружка!
Однозначно! Однозначно!
Думаю, тебе нужна другая!
```

Программа:

```
import System.IO

main = do
   handle <- openFile "girlfriend.txt" ReadMode
   contents <- hGetContents handle
   putStr contents
   hClose handle
```

Скомпилировав и запустив её, получаем ожидаемый результат:

```
Эй! Ты! Эй! Ты!
Мне не нравится твоя подружка!
Однозначно! Однозначно!
Думаю, тебе нужна другая!
```

Посмотрим, что у нас тут? Первая строка – это просто четыре восклицания: они привлекают наше внимание. Во второй строке Аврил сообщает вам, что ей не нравится ваша подружка. Третья строка подчёркивает, что неприятие это категорическое. Ну а четвёртая предписывает подружиться с кем-нибудь получше.

А теперь пройдемся по каждой строке кода. Наша программа – это несколько действий ввода-вывода, «склеенных» с помощью блока `do`. В первой строке блока `do` мы использовали новую функцию, `openFile`. Вот её сигнатура: `openFile :: FilePath -> IOMode -> IO Handle`. Если попробовать это прочитать, получится следующее: «Функция `openFile` принимает путь к файлу и режим открытия файла `(IOMode)` и возвращает действие ввода-вывода, которое откроет файл, получит дескриптор файла и заключит его в результат».

Тип `FilePath` – это просто синоним для типа `String`; он определён так:

```
type FilePath = String
```

Тип IOMode определён так:

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Этот тип содержит перечисление режимов открытия файла, так же как наш тип содержал перечисление дней недели. Очень просто! Обратите внимание, что этот тип – IOMode; не путайте его с IO Mode. Тип IO Mode может быть типом действия ввода-вывода, которое возвращает результат типа Mode, но тип IOMode – это просто перечисление.

В конце концов функция вернёт действие ввода-вывода, которое откроет указанный файл в указанном режиме. Если мы привяжем это действие к имени, то получим дескриптор файла (Handle). Значение типа Handle описывает, где находится наш файл. Мы будем использовать дескриптор для того, чтобы знать, из какого файла читать. Было бы глупо открыть файл и не связать дескриптор файла с именем, потому что с ним потом ничего нельзя будет сделать! В нашем случае мы связали дескриптор с идентификатором handle.

На следующей строке мы видим функцию hGetContents. Она принимает значение типа Handle; таким образом, она знает, с каким файлом работать, и возвращает значение типа IO String – действие ввода-вывода, которое вернёт содержимое файла в результате. Функция похожа на функцию getContents. Единственное отличие – функция getContents читает со стандартного входа (то есть с терминала), в то время как функция hGetContents принимает дескриптор файла, из которого будет происходить чтение. Во всех остальных смыслах они работают одинаково. Так же как и getContents, наша функция hGetContents не пытается прочитать весь файл целиком и сохранить его в памяти, но читает его по мере необходимости. Это очень удобно, поскольку мы можем считать, что идентификатор contents хранит всё содержимое файла, но на самом деле содержимого фай-

ла в памяти нет. Так что даже чтение из очень больших файлов не отожрёт всю память, но будет считывать только то, что нужно, и тогда, когда нужно.

Обратите внимание на разницу между дескриптором, который используется для идентификации файла, и его содержимым. В нашей программе они привязываются к именам `handle` и `contents`. Дескриптор – это нечто, с помощью чего мы знаем, что есть наш файл. Если представить всю файловую систему в виде очень большой книги, а каждый файл в виде главы, то дескриптор будет чем-то вроде закладки, которая показывает нам, где мы в данный момент читаем (или пишем), в то время как идентификатор `contents` будет содержать саму главу.

С помощью вызова `putStr contents` мы распечатываем содержимое на стандартном выводе, а затем выполняем функцию `hClose`, которая принимает дескриптор и возвращает действие ввода-вывода, закрывающее файл. После открытия файла с помощью функции `openFile` вы должны закрывать файлы самостоятельно!

Использование функции withFile

То, что мы только что сделали, можно сделать и по-другому – с использованием функции `withFile`. Сигнатура этой функции:

```
withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
```

Она принимает путь к файлу, режим открытия файла и некоторую функцию, принимающую дескриптор и возвращающую некое действие ввода-вывода. Функция `withFile` вернёт действие ввода-вывода, которое откроет файл, сделает с ним то, что нам нужно, и закроет его. Результат, помещённый в заключительном действии ввода-вывода, будет взят из результата переданной нами функции. С виду это может показаться сложным, но на самом деле всё просто, особенно если использовать анонимные функции. Вот как можно переписать предыдущий пример с использованием функции `withFile`:

```
import System.IO

main = do
   withFile "girlfriend.txt" ReadMode (\handle -> do
       contents <- hGetContents handle
       putStr contents)
```

Функция `(\handle -> ...)` принимает дескриптор файла и возвращает действие ввода-вывода. Обычно пишут именно так, пользуясь анонимной функцией. Нам действительно нужна функция, возвращающая действие ввода-вывода, а не просто выполнение некоторого действия и последующее закрытие файла, поскольку действие, переданное функции `withFile`, не знало бы, с каким файлом ему необходимо работать. Сейчас же функция `withFile` открывает файл, а затем передаёт его дескриптор функции, которую мы ей передали. Функция возвращает действие ввода-вывода, на основе которого `withFile` создаёт новое действие, работающее почти так же, как и исходное, но с добавлением гарантированного закрытия файла даже в тех случаях, когда что-то пошло не так.

Время заключать в скобки

Обычно, если какой-нибудь фрагмент кода вызывает функцию `error` (например, когда мы пытаемся вызвать функцию `head` для пустого списка) или случается что-то плохое при вводе-выводе, наша программа завершается с сообщением об ошибке. В таких обстоятельствах говорят, что произошло *исключение*. Функция `withFile` гарантирует, что независимо от того, возникнет исключение или нет, файл будет закрыт.

Подобные сценарии встречаются довольно часто. Мы получаем в распоряжение некоторый ресурс (например, файловый дескриптор), хотим с ним что-нибудь сделать, но кроме того хотим, чтобы он был освобождён (файл закрыт). Как раз для таких случаев в модуле `Control.Exception` имеется функция `bracket`. Вот её сигнатура:

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

Первым параметром является действие, получающее ресурс (дескриптор файла). Второй параметр – функция, освобождающая ресурс. Эта функция будет вызвана даже в случае возникновения исключения. Третий параметр – это функция, которая также принимает на вход ресурс и что-то с ним делает. Именно в третьем параметре и происходит всё самое важное, а именно: чтение файла или его запись.

Поскольку функция `bracket` – это и есть всё необходимое для получения ресурса, работы с ним и гарантированного освобождения, с её помощью можно получить простую реализацию функции `withFile`:

```
withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile name mode f = bracket (openFile name mode)
   (\handle -> hClose handle)
   (\handle -> f handle)
```

Первый параметр, который мы передали функции `bracket`, открывает файл; результатом является дескриптор. Второй параметр принимает дескриптор и закрывает его. Функция `bracket` даёт гарантию, что это произойдёт, даже если возникнет исключение. Наконец, третий параметр функции `bracket` принимает дескриптор и применяет к нему функцию `f`, которая по заданному дескриптору делает с файлом всё необходимое, будь то его чтение или запись.

Хватай дескрипторы!

Подобно тому как функция `hGetContents` работает по аналогии с функцией `getContents`, но с указанным файлом, существуют функции `hGetLine`, `hPutStr`, `hPutStrLn`, `hGetChar` и т. д., ведущие себя так же, как их варианты без буквы `h`, но принимающие дескриптор как параметр и работающие с файлом, а не со стандартным вводом-выводом. Пример: `putStrLn` – это функция, принимающая строку и возвращающая действие ввода-вывода, которое напечатает строку на терминале, а затем выполнит перевод на новую строку. Функция `hPutStrLn` принимает дескриптор файла и строку и возвращает действие, которое запишет строку в файл и затем поместит в файл символ(ы) перехода на новую строку. Функция `hGetLine` принимает дескриптор и возвращает действие, которое считывает строку из файла.

Загрузка файлов и обработка их содержимого в виде строк настолько распространена, что есть три маленькие удобные функции, которые делают эту задачу ещё легче.

Сигнатура функции `readFile` такова:

```
readFile :: FilePath -> IO String
```

Мы помним, что тип `FilePath` – это просто удобное обозначение для `String`. Функция `readFile` принимает путь к файлу и возвращает действие ввода-вывода, которое прочитает файл (лениво, конечно же) и свяжет содержимое файла в виде строки с некоторым именем. Обычно это более удобно, чем вызывать функцию `openFile` и связывать дескриптор с именем, а затем вызывать функцию `hGetContents`. Вот как мы могли бы переписать предыдущий пример с использованием `readFile`:

```
import System.IO

main = do
   contents <- readFile "girlfriend.txt"
   putStr contents
```

Так как мы не получаем дескриптор файла в качестве результата, то не можем закрыть его сами. Если мы используем функцию `readFile`, за нас это сделает язык Haskell.

Функция `writeFile` имеет тип

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

Она принимает путь к файлу и строку для записи в файл и возвращает действие ввода-вывода, которое выполнит запись. Если такой файл уже существует, перед записью он будет обрезан до нулевой длины. Вот как получить версию файла *girlfriend.txt* в верхнем регистре и записать её в файл *girlfriendcaps.txt*:

```
import System.IO
import Data.Char

main = do
   contents <- readFile "girlfriend.txt"
   writeFile "girlfriendcaps.txt" (map toUpper contents)
```

Функция `appendFile` имеет ту же сигнатуру, что и `writeFile`, и действует почти так же. Она только не обрезает уже существующий файл до нулевой длины перед записью, а добавляет новое содержимое в конец файла.

Список дел

Воспользуемся функцией `appendFile` на примере написания программы, которая добавляет в текстовый файл, содержащий список наших дел, новое задание. Допустим, у нас уже есть такой файл с названием *todo.txt*, и каждая его строка соответствует одному заданию.

Наша программа будет читать из стандартного потока ввода одну строку и добавлять её в конец файла *todo.txt*:

```
import System.IO

main = do
   todoItem <- getLine
   appendFile "todo.txt" (todoItem ++ "\n")
```

Обратите внимание на добавление символа конца строки вручную, функция `getLine` возвращает строку без него.

Сохраните этот файл с именем *appendtodo.hs*, скомпилируйте его и несколько раз запустите.

```
$ ./appendtodo
Погладить посуду
$ ./appendtodo
Помыть собаку
$ ./appendtodo
Вынуть салат из печи
$ cat todo.txt
Погладить посуду
Помыть собаку
Вынуть салат из печи
```

> **ПРИМЕЧАНИЕ.** *Программа* cat *в Unix-подобных системах используется для вывода содержимого текстового файла на терминал. В Windows можно воспользоваться командой* type *или посмотреть содержимое файла в любом текстовом редакторе.*

Удаление заданий

Мы уже написали программу, которая добавляет новый элемент к списку заданий в файл *todo.txt*; теперь напишем программу для удаления элемента. Мы применим несколько новых функций из модуля System.Directory и одну новую функцию из модуля System.IO; их работа будет объяснена позднее.

```
import System.IO
import System.Directory
import Data.List

main = do
   contents <- readFile "todo.txt"
   let todoTasks = lines contents
       numberedTasks = zipWith (\n line -> show n ++ " - " ++ line)
                               [0..] todoTasks
   putStrLn "Ваши задания:"
   mapM_ putStrLn numberedTasks
   putStrLn "Что вы хотите удалить?"
   numberString <- getLine
   let number = read numberString
       newTodoItems = unlines $ delete (todoTasks !! number) todoTasks
   (tempName, tempHandle) <- openTempFile "." "temp"
   hPutStr tempHandle newTodoItems
   hClose tempHandle
   removeFile "todo.txt"
   renameFile tempName "todo.txt"
```

Сначала мы читаем содержимое файла *todo.txt* и связываем его с именем contents. Затем разбиваем всё содержимое на список строк. Список todoTasks выглядит примерно так:

```
["Погладить посуду", "Помыть собаку", "Вынуть салат из печи"]
```

Далее соединяем числа, начиная с 0, и элементы списка дел с помощью функции, которая берёт число (скажем, 3) и строку (например, "привет") и возвращает новую строку ("3 – привет"). Вот примерный вид списка numberedTasks:

```
["0 - Погладить посуду", "1 - Помыть собаку", "2 - Вынуть салат из печи"]
```

Затем с помощью вызова `mapM_ putStrLn numberedTasks` мы печатаем каждое задание на отдельной строке, после чего спрашиваем пользователя, что он хочет удалить, и ждём его ответа. Например, он хочет удалить задание 1 (Помыть собаку), так что мы получим число `1`. Значением переменной `numberString` будет `"1"`, и, поскольку вместо строки нам необходимо число, мы применяем функцию `read` и связываем результат с именем `number`.

Помните функции `delete` и `!!` из модуля `Data.List`? Оператор `!!` возвращает элемент из списка по индексу, функция `delete` удаляет первое вхождение элемента в список, возвращая новый список без удалённого элемента. Выражение `(todoTasks !! number)`, где `number` – это `1`, возвращает строку `"Помыть собаку"`. Мы удаляем первое вхождение этой строки из списка `todoTasks`, собираем всё оставшееся в одну строку функцией `unlines` и даём результату имя `newTodoItems`.

Далее используем новую функцию из модуля `System.IO` – `openTempFile`. Имя функции говорит само за себя: `open temp file` – «открыть временный файл». Она принимает путь к временному каталогу и шаблон имени файла и открывает временный файл. Мы использовали символ `.` в качестве каталога для временных файлов, так как `.` обозначает текущий каталог практически во всех операционных системах. Строку `"temp"` мы указали в качестве шаблона имени для временного файла; это означает, что временный файл будет назван `temp` плюс несколько случайных символов. Функция возвращает действие ввода-вывода, которое создаст временный файл; результат действия – пара значений, имя временного файла и дескриптор. Мы могли бы открыть обычный файл, например с именем *todo2.txt*, но использовать `openTempFile` – хорошая практика: в этом случае не приходится опасаться, что вы случайно что-нибудь перезапишете.

Теперь, когда временный файл открыт, запишем туда строку `newTodoItems`. В этот момент исходный файл не изменён, а временный содержит все строки из исходного, за исключением удалённой.

Затем мы закрываем временный файл и удаляем исходный с помощью функции `removeFile`, которая принимает путь к файлу и удаляет его. После удаления старого файла *todo.txt* мы используем функцию `renameFile`, чтобы переименовать временный файл в *todo.txt*. Обратите внимание: функции `removeFile` и `renameFile` (обе они определены в модуле `System.Directory`) принимают в качестве параметров не дескрипторы, а пути к файлам.

Сохраните программу в файле с именем *deletetodo.hs*, скомпилируйте её и проверьте:

```
$ ./deletetodo
Ваши задания:
0 - Погладить посуду
1 - Помыть собаку
2 - Вынуть салат из печи
Что вы хотите удалить?
1
```

Смотрим, что осталось:

```
$ cat todo.txt
Погладить посуду
Вынуть салат из печи
```

Круто! Удалим ещё что-нибудь:

```
$ ./deletetodo
Ваши задания:
0 - Погладить посуду
1 - Вынуть салат из печи
Что вы хотите удалить?
0
```

Проверяя файл с заданиями, убеждаемся, что осталось только одно:

```
$ cat todo.txt
Вынуть салат из печи
```

Итак, всё работает. Осталась только одна вещь, которую мы в этой программе не учли. Если после открытия временного файла что-то произойдёт и программа неожиданно завершится, то временный файл не будет удалён. Давайте это исправим.

Уборка

Чтобы гарантировать удаление временного файла, воспользуемся функцией `bracketOnError` из модуля `Control.Exception`. Она очень похожа на `bracket`, но если последняя получает ресурс и гарантирует, что освобождение ресурса будет выполнено всегда, то функция `bracketOnError` выполнит завершающие действия только в случае возникновения исключения. Вот исправленный код:

```
import System.IO
import System.Directory
import Data.List
import Control.Exception

main = do
   contents <- readFile "todo.txt"
   let todoTasks = lines contents
       numberedTasks = zipWith (\n line -> show n ++ " - " ++ line)
                               [0..] todoTasks
   putStrLn "Ваши задания:"
   mapM_ putStrLn numberedTasks
   putStrLn "Что вы хотите удалить?"
   numberString <- getLine
   let number = read numberString
       newTodoItems = unlines $ delete (todoTasks !! number) todoTasks
   bracketOnError (openTempFile "." "temp")
      (\(tempName, tempHandle) -> do
            hClose tempHandle
            removeFile tempName)
      (\(tempName, tempHandle) -> do
            hPutStr tempHandle newTodoItems
            hClose tempHandle
            removeFile "todo.txt"
            renameFile tempName "todo.txt")
```

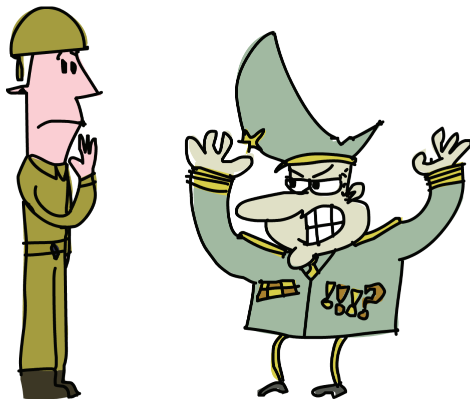
Вместо обычного использования функции `openTempFile` мы заключаем её в `bracketOnError`. Затем пишем, что должно произойти при возникновении исключения: мы хотим закрыть и удалить временный файл. Если же всё нормально, пишем новый список заданий во временный файл; все эти строки остались без изменения. Мы выводим новые задания, удаляем исходный файл и переименовываем временный.

Аргументы командной строки

Если вы пишете консольный скрипт или приложение, то вам наверняка понадобится работать с аргументами командной строки. К счастью, в стандартную библиотеку языка Haskell входят удобные функции для работы с ними.

В предыдущей главе мы написали программы для добавления и удаления элемента в список заданий. Но у нашего подхода есть две

проблемы. Во-первых, мы жёстко задали имя файла со списком заданий в тексте программы. Мы решили, что файл будет называться *todo.txt*, и что пользователь никогда не захочет вести несколько списков.



Эту проблему можно решить, спрашивая пользователя каждый раз, какой файл он хочет использовать как файл со списком заданий. Мы использовали такой подход, когда спрашивали пользователя, какой элемент он хочет удалить. Это, конечно, работает, но не идеально, поскольку пользователь должен запустить программу, подождать, пока она спросит что-нибудь, и затем дать ответ. Такая программа называется *интерактивной*, и сложность здесь заключается вот в чём: вдруг вам понадобится автоматизировать выполнение этой программы, например, с помощью скрипта? Гораздо сложнее написать скрипт, который будет взаимодействовать с программой, чем обычный скрипт, который просто вызовет её один или несколько раз!

Вот почему иногда лучше сделать так, чтобы пользователь сообщал, чего он хочет, при запуске программы, вместо того чтобы она сама спрашивала его после запуска. И что может послужить этой цели лучше командной строки!..

В модуле `System.Environment` есть два полезных действия ввода-вывода. Первое – это функция `getArgs`; её тип – `getArgs :: IO [String]`. Она получает аргументы, с которыми была вызвана программа, и возвращает их в виде списка. Второе – функция `getProgName`, тип которой – `getProgName :: IO String`. Это действие ввода-вывода, возвращающее имя программы.

Вот простенькая программа, которая показывает, как работают эти два действия:

```
import System.Environment
import Data.List

main = do
```

```
   args <- getArgs
   progName <- getProgName
   putStrLn "Аргументы командной строки:"
   mapM putStrLn args
   putStrLn "Имя программы:"
   putStrLn progName
```

Мы связываем значения, возвращаемые функциями `getArgs` **и** `progName`**, с именами** `args` **и** `progName`**. Выводим строку** `"Аргументы командной строки:"` **и затем для каждого аргумента из списка** `args` **выполняем функцию** `putStrLn`**. После этого печатаем имя программы. Скомпилируем программу с именем** `arg-test` **и проверим, как она работает:**

```
$ ./arg-test first second w00t "multi word arg"
Аргументы командной строки:
first
second
w00t
multi word arg
Имя программы:
arg-test
```

Ещё больше шалостей со списком дел

В предыдущих примерах мы писали отдельные программы для добавления и удаления заданий в списке дел. Теперь мы собираемся объединить их в новое приложение, а что ему делать, будем указывать в командной строке. Кроме того, позаботимся о том, чтобы программа смогла работать с разными файлами – не только *todo.txt*.

Назовём программу просто `todo`, она сможет делать три разные вещи:

- просматривать задания;
- добавлять задания;
- удалять задания.

Для добавления нового задания в список дел в файле *todo.txt* мы будем писать:

```
$ ./todo add todo.txt "Найти магический меч силы"
```

Просмотреть текущие задания можно будет командой `view`:

```
$ ./todo view todo.txt
```

Для удаления задания потребуется дополнительно указать его индекс:

```
$ ./todo remove todo.txt 2
```

Многозадачный список задач

Начнём с реализации функции, которая принимает команду в виде строки (например, `"add"` или `"view"`) и возвращает функцию, которая в свою очередь принимает список аргументов и возвращает действие ввода-вывода, выполняющее в точности то, что необходимо:

```
import System.Environment
import System.Directory
import System.IO
import Data.List
import Control.Exception

dispatch :: String -> [String] -> IO ()
dispatch "add" = add
dispatch "view" = view
dispatch "remove" = remove
```

Функция `main` будет выглядеть так:

```
main = do
   (command:argList) <- getArgs
   dispatch command argList
```

Первым делом мы получаем аргументы и связываем их со списком `(command:argsList)`. Таким образом, первый аргумент будет связан с именем `command`, а все остальные – со списком `argList`. В следующей строке к переменной `commands` применяется функция `dispatch`, результатом которой может быть одна из функций `add`, `view` или `remove`. Затем результирующая функция применяется к списку аргументов `argList`.

Предположим, программа запущена со следующими параметрами:

```
$ ./todo add todo.txt "Найти магический меч силы"
```

Тогда значением command будет "add", а значением argList – список ["todo.txt", "Найти магический меч силы"]. Поэтому сработает первый вариант определения функции dispatch и будет возвращена функция add. Применяем её к argList, результатом оказывается действие ввода-вывода, добавляющее новое задание в список.

Теперь давайте реализуем функции add, view и remove. Начнём с первой из них:

```
add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")
```

При вызове

```
$ ./todo add todo.txt "Найти магический меч силы"
```

функции add будет передан список ["todo.txt", "Найти магический меч силы"]. Поскольку пока мы не обрабатываем некорректный ввод, достаточно будет сопоставить аргумент функции add с двухэлементным списком. Результатом функции будет действие ввода-вывода, добавляющее строку вместе с символом конца строки в конец файла.

Далее реализуем функциональность просмотра списка. Если мы хотим просмотреть элементы списка, то вызываем программу так: todo view todo.txt. В первом сопоставлении с образцом идентификатор command будет связан со строкой view, а идентификатор argList будет равен ["todo.txt"].

Вот код функции view:

```
view :: [String] -> IO ()
view [fileName] = do
   contents <- readFile fileName
   let todoTasks = lines contents
       numberedTasks = zipWith (\n line -> show n ++ " - " ++ line)
                       [0..] todoTasks
   putStr $ unlines numberedTasks
```

Программа, которая удаляла задачу из списка, производила практически те же самые действия: мы отображали список задач, чтобы пользователь мог выбрать, какую из них удалить. Но в этой функции мы просто отображаем список.

Ну и наконец реализуем функцию `remove`. Функция будет очень похожа на программу для удаления элемента, так что если вы не понимаете, как работает функция удаления, прочитайте пояснения к её определению. Основное отличие – мы не задаём жёстко имя файла, а получаем его как аргумент. Также мы не спрашиваем у пользователя номер задачи для удаления – его мы также получаем в виде аргумента.

```
remove :: [String] -> IO ()
remove [fileName, numberString] = do
   contents <- readFile fileName
   let todoTasks = lines contents
       number = read numberString
       newTodoItems = unlines $ delete (todoTasks !! number) todoTasks
   bracketOnError (openTempFile "." "temp")
      (\(tempName, tempHandle) -> do
            hClose tempHandle
            removeFile tempName)
      (\(tempName, tempHandle) -> do
            hPutStr tempHandle newTodoItems
            hClose tempHandle
            removeFile fileName
            renameFile tempName fileName)
```

Мы открываем файл, полное имя которого задаётся в идентификаторе `fileName`, открываем временный файл, удаляем строку по индексу, записываем во временный файл, удаляем исходный файл и переименовываем временный в `fileName`. Приведем полный листинг программы во всей её красе:

```
import System.Environment
import System.Directory
import System.IO
import Control.Exception
import Data.List

dispatch :: String -> [String] -> IO ()
dispatch "add" = add
dispatch "view" = view
dispatch "remove" = remove

main = do
```

```
   (command:argList) <- getArgs
   dispatch command argList

add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")

view :: [String] -> IO ()
view [fileName] = do
   contents <- readFile fileName
   let todoTasks = lines contents
       numberedTasks = zipWith (\n line -> show n ++ " - " ++ line)
                       [0..] todoTasks
   putStr $ unlines numberedTasks

remove :: [String] -> IO ()
remove [fileName, numberString] = do
   contents <- readFile fileName
   let todoTasks = lines contents
       number = read numberString
       newTodoItems = unlines $ delete (todoTasks !! number) todoTasks
   bracketOnError (openTempFile "." "temp")
      (\(tempName, tempHandle) -> do
            hClose tempHandle
            removeFile tempName)
      (\(tempName, tempHandle) -> do
            hPutStr tempHandle newTodoItems
            hClose tempHandle
            removeFile fileName
            renameFile tempName fileName)
```

Резюмируем наше решение. Мы написали функцию dispatch, отображающую команды на функции, которые принимают аргументы командной строки в виде списка и возвращают соответствующее действие ввода-вывода. Основываясь на значении первого аргумента, функция dispatch даёт нам необходимую функцию. В результате вызова этой функции мы получаем требуемое действие и выполняем его.

Давайте проверим, как наша программа работает:

```
$ ./todo view todo.txt
0 - Погладить посуду
1 - Помыть собаку
```

```
2 - Вынуть салат из печи

$ ./todo add todo.txt "Забрать детей из химчистки"

$ ./todo view todo.txt
0 - Погладить посуду
1 - Помыть собаку
2 - Вынуть салат из печи
3 - Забрать детей из химчистки

$ ./todo remove todo.txt 2

$ ./todo view todo.txt
0 - Погладить посуду
1 - Помыть собаку
2 - Забрать детей из химчистки
```

Большой плюс такого подхода – легко добавлять новую функциональность. Добавить вариант определения функции `dispatch`, реализовать соответствующую функцию – и готово! В качестве упражнения можете реализовать функцию `bump`, которая примет файл и номер задачи и вернёт действие ввода-вывода, которое поднимет указанную задачу на вершину списка задач.

Работаем с некорректным вводом

Можно было бы дописать эту программу, улучшив сообщения об ошибках, возникающих при некорректных исходных данных. Начать можно с добавления варианта функции `dispatch`, который срабатывает при любой несуществующей команде:

```
dispatch :: String -> [String] -> IO ()
dispatch "add" = add
dispatch "view" = view
dispatch "remove" = remove
dispatch command = doesntExist command

doesntExist :: String -> [String] -> IO ()
doesntExist command _ =
   putStrLn $ "Команда " ++ command ++ " не определена"
```

Также можно добавить варианты определения функций `add`, `view` и `remove` для случаев, когда программе передано неправильное количество аргументов. Например:

```
add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")
add _ = putStrLn "Команда add принимает в точности два аргумента"
```

Если функция add будет применена к списку, содержащему не два элемента, первый образец не сработает, поэтому пользователю будет выведено сообщение об ошибке. Аналогично дописываются функции view и remove.

Заметьте, что мы не обрабатываем все возможные случаи некорректного ввода. К примеру, программа «упадёт», если мы запустим её так:

```
./todo
```

Мы также не проверяем, существует ли файл, с которым идёт работа. Добавить обработку всех этих событий несложно, хотя и несколько утомительно, поэтому оставляем реализацию «защиты от дурака» в качестве упражнения для читателя.

Случайность

Зачастую при программировании бывает необходимо получить некоторые случайные данные. Возможно, вы создаёте игру, где нужно бросать игральные кости, или генерируете тестовые данные, чтобы проверить вашу программу. Существует много применений случайным данным. На самом деле они, конечно, псевдослучайны – ведь мы-то с вами знаем, что настоящим примером случайности можно считать разве что пьяную обезьяну на одноколесном велосипеде, которая одной лапой хватается за собственный зад, а в другой держит сыр. В этой главе мы узнаем, как заставить язык Haskell генерировать *вроде бы* случайные данные (без сыра и велосипеда).

В большинстве языков программирования есть функции, которые возвращают некоторое случайное число. Каждый раз, когда вы вызываете такую функцию, вы (надеюсь) получаете новое случайное число. Ну а как в языке Haskell? Как мы помним, Haskell – чистый функциональный язык. Это означает, что он обладает свойством *детерминированности*. Выражается оно в том, что если функции дважды передать один и тот же аргумент, она должна дважды вернуть один и тот же результат. На самом деле это удобно,

поскольку облегчает наши размышления о программах, а также позволяет отложить вычисление до тех пор, пока оно на самом деле не пригодится. Если я вызываю функцию, то могу быть уверен, что она не делает каких-либо темных делишек на стороне, прежде чем вернуть мне результат. Однако из-за этого получать случайные числа не так-то просто. Допустим, у меня есть такая функция:

```
randomNumber :: Int
randomNumber = 4
```

Она не очень-то полезна в качестве источника случайных чисел, потому что всегда возвращает 4, даже если я поклянусь, что эта четвёрка абсолютно случайная, так как я использовал игральную кость для определения этого числа!

Как другие языки вычисляют псевдослучайные числа? Они получают некую информацию от компьютера, например: текущее время, как часто и в каком направлении вы перемещаете мышь, какие звуки вы издаёте, когда сидите за компьютером, и, основываясь на этом, выдают число, которое на самом деле выглядит случайным. Комбинации этих факторов (их случайность), вероятно, различаются в каждый конкретный момент времени; таким образом, вы и получаете разные случайные числа.

Ага!.. Так же вы можете создавать случайные числа и в языке Haskell, если напишете функцию, которая принимает случайные величины как параметры и, основываясь на них, возвращает некоторое число (или другой тип данных).

Посмотрим на модуль `System.Random`. В нём содержатся функции, которые удовлетворят все наши нужды в отношении случайностей! Давайте посмотрим на одну из экспортируемых функций, а именно `random`. Вот её тип:

```
random :: (RandomGen g, Random a) => g -> (a, g)
```

Так! В декларации мы видим несколько новых классов типов. Класс типов RandomGen предназначен для типов, которые могут служить источниками случайности. Класс типов Random предназначен для типов, которые могут принимать случайные значения. Булевские значения могут быть случайными; это может быть True или False. Число может принимать огромное количество случайных значений. Может ли функция принимать случайное значение? Не думаю – скорее всего, нет! Если мы попытаемся перевести объявление функции random на русский язык, получится что-то вроде «функция принимает генератор случайности (источник случайности), возвращает случайное значение и новый генератор случайности». Зачем она возвращает новый генератор вместе со случайным значением?.. Увидим через минуту.

Чтобы воспользоваться функцией random, нам нужно получить один из генераторов случайности. Модуль System.Random экспортирует полезный тип StdGen, который имеет экземпляр класса RandomGen. Мы можем создать значение типа StdGen вручную или попросить систему выдать нам генератор, основывающийся на нескольких вроде бы случайных вещах.

Для того чтобы создать генератор вручную, используйте функцию mkStdGen. Её тип – mkStdGen :: Int -> StdGen. Он принимает целое число и основывается на нём, возвращая нам генератор. Давайте попробуем использовать функции random и mkStdGen, чтобы получить... сомнительно, что случайное число.

```
ghci> random (mkStdGen 100)
<interactive>:1:0:
   Ambiguous type variable `a' in the constraint:
     `Random a' arising from a use of `random' at <interactive>:1:0-20
   Probable fix: add a type signature that fixes these type variable(s)
```

Что это?... Ах, да, функция random может возвращать значения любого типа, который входит в класс типов Random, так что мы должны указать языку Haskell, какой тип мы желаем получить в результате. Также не будем забывать, что функция возвращает случайное значение и генератор в паре.

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

Ну наконец-то! Число выглядит довольно-таки случайным. Первый компонент кортежа – это случайное число, второй элемент – текстовое представление нового генератора. Что случится, если мы вызовем функцию random с тем же генератором снова?

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

Как и следовало ожидать! Тот же результат для тех же параметров. Так что давайте-ка передадим другой генератор в параметре.

```
ghci> random (mkStdGen 949494) :: (Int, StdGen)
(539963926,466647808 1655838864)
```

Отлично, получили другое число. Мы можем использовать аннотацию типа для того, чтобы получать случайные значения разных типов.

```
ghci> random (mkStdGen 949488) :: (Float, StdGen)
(0.8938442,1597344447 1655838864)
ghci> random (mkStdGen 949488) :: (Bool, StdGen)
(False,1485632275 40692)
ghci> random (mkStdGen 949488) :: (Integer, StdGen)
(1691547873,1597344447 1655838864)
```

Подбрасывание монет

Давайте напишем функцию, которая эмулирует трёхкратное подбрасывание монеты. Если бы функция random не возвращала новый генератор вместе со случайным значением, нам пришлось бы передавать в функцию три случайных генератора в качестве параметров и затем возвращать результат подбрасывания монеты для каждого из них. Но это выглядит не очень разумным, потому что если один генератор может создавать случайные значения типа Int (а он может принимать довольно много разных значений), его должно хватить и на троекратное подбрасывание монеты (что даёт нам в точности восемь комбинаций). В таких случаях оказывается очень полезно, что функция random возвращает новый генератор вместе со значением.

Будем представлять монету с помощью Bool. True – это «орёл», а False –«решка».

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
   let (firstCoin, newGen) = random gen
       (secondCoin, newGen') = random newGen
       (thirdCoin, newGen'') = random newGen'
   in  (firstCoin, secondCoin, thirdCoin)
```

Мы вызываем функцию `random` с генератором, который нам передали в параметре, и получаем монету и новый генератор. Затем снова вызываем функцию `random`, но на этот раз с новым генератором, чтобы получить вторую монету. Делаем то же самое с третьей монетой. Если бы мы вызывали функцию `random` с одним генератором, все монеты имели бы одинаковое значение, и в результате мы могли бы получать только `(False, False, False)` или `(True, True, True)`.

```
ghci> threeCoins (mkStdGen 21)
(True,True,True)
ghci> threeCoins (mkStdGen 22)
(True,False,True)
ghci> threeCoins (mkStdGen 943)
(True,False,True)
ghci> threeCoins (mkStdGen 944)
(True,True,True)
```

Обратите внимание, что нам не надо писать `random gen :: (Bool, StdGen)`: ведь мы уже указали, что мы желаем получить булевское значение, в декларации типа функции. По декларации язык Haskell может вычислить, что нам в данном случае нужно получить булевское значение.

Ещё немного функций, работающих со случайностью

А что если бы мы захотели подкинуть четыре монеты? Или пять? На этот случай есть функция `randoms`, которая принимает генератор и возвращает бесконечную последовательность значений, основываясь на переданном генераторе.

```
ghci> take 5 $ randoms (mkStdGen 11) :: [Int]
[-1807975507,545074951,-1015194702,-1622477312,-502893664]
ghci> take 5 $ randoms (mkStdGen 11) :: [Bool]
```

```
[True,True,True,True,False]
ghci> take 5 $ randoms (mkStdGen 11) :: [Float]
[7.904789e-2,0.62691015,0.26363158,0.12223756,0.38291094]
```

Почему функция `randoms` не возвращает новый генератор вместе со списком? Мы легко могли бы реализовать функцию `randoms` вот так:

```
randoms' :: (RandomGen g, Random a) => g -> [a]
randoms' gen = let (value, newGen) = random gen in value:randoms' newGen
```

Рекурсивное определение. Мы получаем случайное значение и новый генератор из текущего генератора, а затем создаём список, который помещает сгенерированное значение в «голову» списка, а значения, сгенерированные по новому генератору, – в «хвост». Так как теоретически мы можем генерировать бесконечное количество чисел, вернуть новый генератор нельзя.

Мы могли бы создать функцию, которая генерирует конечный поток чисел и новый генератор таким образом:

```
finiteRandoms :: (RandomGen g, Random a, Num n) => n -> g -> ([a], g)
finiteRandoms 0 gen = ([], gen)
finiteRandoms n gen =
   let (value, newGen) = random gen
       (restOfList, finalGen) = finiteRandoms (n-1) newGen
   in  (value:restOfList, finalGen)
```

Опять рекурсивное определение. Мы полагаем, что если нам нужно 0 чисел, мы возвращаем пустой список и исходный генератор. Для любого другого количества требуемых случайных значений вначале мы получаем одно случайное число и новый генератор. Это будет «голова» списка. Затем мы говорим, что «хвост» будет состоять из $(n - 1)$ чисел, сгенерированных новым генератором. Далее возвращаем объединенные «голову» и остаток списка и финальный генератор, который мы получили после вычисления $(n - 1)$ случайных чисел.

Ну а если мы захотим получить случайное число в некотором диапазоне? Все случайные числа до сих пор были чрезмерно большими или маленькими. Что если нам нужно подбросить игральную кость?.. Для этих целей используем функцию `randomR`. Она имеет следующий тип:

```
randomR :: (RandomGen g, Random a) :: (a, a) -> g -> (a, g)
```

Это значит, что функция похожа на функцию `random`, но получает в первом параметре пару значений, определяющих верхнюю и нижнюю границы диапазона, и возвращаемое значение будет в границах этого диапазона.

```
ghci> randomR (1,6) (mkStdGen 359353)
(6,1494289578 40692)
ghci> randomR (1,6) (mkStdGen 35935335)
(3,1250031057 40692)
```

Также существует функция `randomRs`, которая возвращает поток случайных значений в заданном нами диапазоне. Смотрим:

```
ghci> take 10 $ randomRs ('a','z') (mkStdGen 3) :: [Char]
"ndkxbvmomg"
```

Неплохо, выглядит как сверхсекретный пароль или что-то в этом духе!

Случайность и ввод-вывод

Вы, должно быть, спрашиваете себя: а какое отношение имеет эта часть главы к системе ввода-вывода? Пока ещё мы не сделали ничего, что имело бы отношение к вводу-выводу! До сих пор мы создавали генераторы случайных чисел вручную, основывая их на некотором целочисленном значении. Проблема в том, что если делать так в реальных программах, они всегда будут возвращать одинаковые последовательности случайных чисел, а это нас не вполне устраивает. Вот почему модуль `System.Random` содержит действие ввода-вывода `getStdGen`, тип которого – `IO StdGen`. При запуске программа запрашивает у системы хороший генератор случайных чисел и сохраняет его в так называемом глобальном генераторе. Функция `getStdGen` передаёт этот глобальный генератор вам, когда вы связываете её с чем-либо.

Вот простая программа, генерирующая случайную строку.

```
import System.Random

main = do
   gen <- getStdGen
   putStrLn $ take 20 (randomRs ('a','z') gen)
```

Теперь проверим:

```
$ ./random_string
pybphhzzhuepknbykxhe

$ ./random_string
eiqgcxykivpudlsvvjpg

$ ./random_string
nzdceoconysdgcyqjruo

$ ./random_string
bakzhnnuzrkgvesqplrx
```

Но будьте осторожны: если дважды вызвать функцию `getStdGen`, система два раза вернёт один и тот же генератор. Если сделать так:

```
import System.Random

main = do
   gen <- getStdGen
   putStrLn $ take 20 (randomRs ('a','z') gen)
   gen2 <- getStdGen
   putStr $ take 20 (randomRs ('a','z') gen2)
```

вы получите дважды напечатанную одинаковую строку.

Лучший способ получить две различные строки – использовать действие ввода-вывода `newStdGen`, которое разбивает текущий глобальный генератор на два генератора. Действие замещает глобальный генератор одним из результирующих генераторов и возвращает второй генератор в качестве результата.

```
import System.Random

main = do
  gen <- getStdGen
  putStrLn $ take 20 (randomRs ('a','z') gen)
  gen' <- newStdGen
  putStr $ take 20 (randomRs ('a','z') gen')
```

Мы не только получаем новый генератор, когда связываем с чем-либо значение, возвращённое функцией `newStdGen`, но и заменяем глобальный генератор; так что если мы воспользуемся функ-

цией getStdGen ещё раз и свяжем его с чем-нибудь, мы получим генератор, отличный от gen.

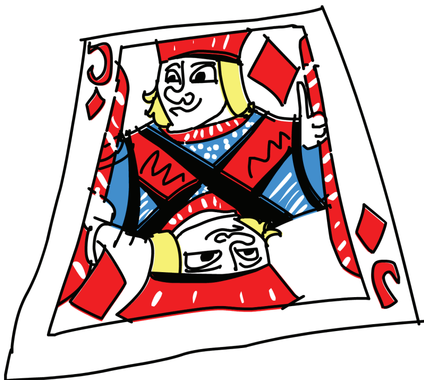
Вот маленькая программка, которая заставляет пользователя угадывать загаданное число.

```
import System.Random
import Control.Monad(when)

main = do
  gen <- getStdGen
  askForNumber gen

askForNumber :: StdGen -> IO ()
askForNumber gen = do
   let (randNumber, newGen) = randomR (1,10) gen :: (Int, StdGen)
   putStr "Я задумал число от 1 до 10. Какое? "
   numberString <- getLine
   when (not $ null numberString) $ do
      let number = read numberString
      if randNumber == number
        then putStrLn "Правильно!"
        else putStrLn $ "Извините, но правильный ответ "
                                  ++ show randNumber
          askForNumber newGen
```

Здесь мы создаём функцию askForNumber, принимающую генератор случайных чисел и возвращающую действие ввода-вывода, которое спросит число у пользователя и сообщит ему, угадал ли он. В этой функции мы сначала генерируем случайное число и новый генератор, основываясь на исходном генераторе; случайное число мы называем randNumber, а новый генератор – newGen. Допустим, что было сгенерировано число 7. Затем мы предлагаем пользователю угадать, какое число мы задумали. Вызываем функцию getLine и связываем её результат с идентификатором numberString. Если пользователь введёт 7, numberString будет равно 7. Далее мы используем функцию when для того, чтобы проверить, не ввёл ли пользователь пустую строку. Если ввёл, выполняется пустое действие ввода-вывода return(), которое закончит выполнение программы. Если пользователь ввёл не пустую строку, выполняется действие, состоящее из блока do. Мы вызываем функцию read со значением numberString в качестве параметра, чтобы преобразовать его в число; образец number становится равным 7.

> **ПРИМЕЧАНИЕ.** *На минуточку!.. Если пользователь введёт что-нибудь, чего функция `read` не сможет прочесть (например, `"ха-ха"`), наша программа «упадёт» с ужасным сообщением об ошибке. Если вы не хотите, чтобы программа «падала» на некорректном вводе, используйте функцию `reads`: она возвращает пустой список, если у функции не получилось считать строку. Если чтение прошло удачно, функция вернёт список из одного элемента, содержащий пару, один компонент которой содержит желаемый элемент; второй компонент хранит остаток строки после считывания первого.*

Мы проверяем, равняется ли `number` случайно сгенерированному числу, и выдаём пользователю соответствующее сообщение. Затем рекурсивно вызываем нашу функцию `askForNumber`, но на сей раз с вновь полученным генератором; это возвращает нам такое же действие ввода-вывода, как мы только что выполнили, но основанное на новом генераторе. Затем это действие выполняется.



Функция `main` состоит всего лишь из получения генератора случайных чисел от системы и вызова функции `askForNumber` с этим генератором для того, чтобы получить первое действие.

Посмотрим, как работает наша программа!

```
$ ./guess_the_number
Я задумал число от 1 до 10. Какое?
4
Извините, но правильный ответ 3
Я задумал число от 1 до 10. Какое?
10
Правильно!
Я задумал число от 1 до 10. Какое?
2
Извините, но правильный ответ 4
Я задумал число от 1 до 10. Какое?
5
```

```
Извините, но правильный ответ 10
Я задумал число от 1 до 10. Какое?
```

Можно написать эту же программу по-другому:

```
import System.Random
import Control.Monad (when)

main = do
   gen <- getStdGen
   let (randNumber, _) = randomR (1,10) gen :: (Int, StdGen)
   putStr "Я задумал число от 1 до 10. Какое? "
   numberString <- getLine
   when (not $ null numberString) $ do
      let number = read numberString
      if randNumber == number
        then putStrLn "Правильно!"
        else putStrLn $ "Извините, но правильный ответ "
                         ++ show randNumber
      newStdGen
      main
```

Эта версия очень похожа на предыдущую, но вместо создания функции, которая принимает генератор и вызывает сама себя рекурсивно с вновь полученным генератором, мы производим все действия внутри функции `main`. После того как пользователь получит ответ, угадал ли он число, мы обновим глобальный генератор и снова вызовем функцию `main`. Оба подхода хороши, но мне больше нравится первый способ, так как он предусматривает меньше действий в функции `main` и даёт нам функцию, которую мы можем легко использовать повторно.

Bytestring: тот же String, но быстрее

Список – полезная и удобная структура данных. Мы использовали списки почти что везде. Существует очень много функций, работающих со списками, и ленивость языка Haskell позволяет нам заменить циклы типа `for` и `while` из других языков программирования на фильтрацию и отображение списков, потому что вычисление произойдёт только тогда, когда оно действительно понадобится. Вот почему такие вещи, как бесконечные списки (и даже

бесконечные списки бесконечных списков!) для нас не проблема. По той же причине списки могут быть использованы в качестве потоков, читаем ли мы со стандартного ввода или из файла. Мы можем открыть файл и считать его как строку, но на самом деле обращение к файлу будет происходить только по мере необходимости.

Тем не менее обработка файлов как строк имеет один недостаток: она может оказаться медленной. Как вы знаете, тип `String` – это просто синоним для типа `[Char]`. У символов нет фиксированного размера, так как для представления, скажем, символа в кодировке Unicode может потребоваться несколько байтов. Более того, список – ленивая структура. Если у вас есть, например, список `[1,2,3,4]`, он будет вычислен только тогда, когда это необходимо. На самом деле список, в некотором смысле, – это обещание списка. Вспомним, что `[1,2,3,4]` – это всего лишь синтаксический сахар для записи `1:2:3:4:[]`. Когда мы принудительно выполняем вычисление первого элемента списка (например, выводим его на экран), остаток списка `2:3:4:[]` также представляет собой «обещание списка», и т. д. Список всего лишь обещает, что следующий элемент будет вычислен, как только он действительно понадобится, причём вместе с элементом будет создано обещание следующего элемента. Не нужно прилагать больших умственных усилий, чтобы понять, что обработка простого списка чисел как серии обещаний – не самая эффективная вещь на свете!

Все эти накладные расходы, связанные со списками, обычно нас не волнуют, но при чтении больших файлов и манипулировании ими это становится помехой. Вот почему в языке Haskell есть байтовые строки. Они похожи на списки, но каждый элемент имеет размер один байт. Также списки и байтовые строки по-разному реализуют ленивость.

Строгие и ленивые

Байтовые строки бывают двух видов: строгие и ленивые. Строгие байтовые строки объявлены в модуле `Data.ByteString`, и они полностью не ленивые. Не используется никаких «обещаний», строгая строка байтов представляет собой последовательность байтов в массиве. Подобная строка не может быть бесконечной. Если вы вычисляете первый байт из строгой строки, вы должны вычислить её целиком. Положительный момент – меньше накладных расходов, поскольку не используются «обещания». Отрицательный момент – такие строки заполнят память быстрее, так как они считываются целиком.

Второй вид байтовых строк определён в модуле `Data.ByteString.Lazy`. Они ленивы – но не настолько, как списки. Как мы говорили ранее, в списке столько же «обещаний», сколько элементов. Вот почему это может сделать его медленным для некоторых целей. Ленивые строки байтов применяют другой подход: они хранятся блоками размером 64 Кб. Если вы вычисляете байт в ленивой байтовой строке (печатая или другим способом), то будут вычислены первые 64 Кб. После этого будет возращено обещание вычислить остальные блоки. Ленивые байтовые строки похожи на список строгих байтовых строк размером 64 Кб. При обработке файла ленивыми байтовыми строками файл будет считываться блок за блоком. Это удобно, потому что не вызывает резкого увеличения потребления памяти, и 64 Кб, вероятно, влезет в L2 – кэш вашего процессора.

Если вы посмотрите документацию на модуль `Data.ByteString.Lazy`, то увидите множество функций с такими же именами, как и в модуле `Data.List`, только в сигнатурах функций будет указан тип `ByteString` вместо `[a]` и `Word8` вместо `a`. Функции в этом модуле работают со значениями типа `ByteString` так же, как одноимённые функции – со списками. Поскольку имена совпадают, нам придётся сделать уточнённый импорт в скрипте и затем загрузить этот скрипт в интерпретатор GHCi для того, чтобы поэкспериментировать с типом `ByteString`.

```
import qualified Data.ByteString.Lazy as B
import qualified Data.ByteString as S
```

Модуль *B* содержит ленивые строки байтов и функции, модуль *S* – строгие. Главным образом мы будем использовать ленивую версию.

Функция `pack` имеет сигнатуру `pack :: [Word8] -> ByteString`. Это означает, что она принимает список байтов типа Word8 и возвращает значение типа `ByteString`. Можно думать, будто функция принимает ленивый список и делает его менее ленивым, так что он ленив только блоками по 64 Кб.

Что за тип `Word8`? Он похож на `Int`, но имеет значительно меньший диапазон, а именно 0 – 255. Тип представляет собой восьмибитовое число. Так же как и `Int`, он имеет экземпляр класса `Num`. Например, мы знаем, что число 5 полиморфно, а значит, оно может вести себя как любой числовой тип. В том числе – принимать тип `Word8`.

```
ghci> B.pack [99,97,110]
Chunk "can" Empty
ghci> B.pack [98..120]
Chunk "bcdefghijklmnopqrstuvwx" Empty
```

Как можно видеть, `Word8` не доставляет много хлопот, поскольку система типов определяет, что числа должны быть преобразованы к нему. Если вы попытаетесь использовать большое число, например 336, в качестве значения типа `Word8`, число будет взято по модулю 256, то есть сохранится 80.

Мы упаковали всего несколько значений в тип `ByteString`; они уместились в один блок. Значение `Empty` – это нечто вроде `[]` для списков.

Если нужно просмотреть байтовую строку байт за байтом, её нужно распаковать. Функция `unpack` обратна функции `pack`. Она принимает строку байтов и возвращает список байтов. Вот пример:

```
ghci> let by = B.pack [98,111,114,116]
ghci> by
Chunk "bort" Empty
ghci> B.unpack by
[98,111,114,116]
```

Вы также можете преобразовывать байтовые строки из строгих в ленивые и наоборот. Функция `fromChunks` принимает список строгих строк и преобразует их в ленивую строку. Соответственно, функция `toChunks` принимает ленивую строку байтов и преобразует её в список строгих строк.

```
ghci> B.fromChunks [S.pack [40,41,42], S.pack [43,44,45], S.pack [46,47,48]]
Chunk "()*" (Chunk "+,-" (Chunk "./0" Empty))
```

Это полезно, если у вас есть множество маленьких строгих строк байтов и вы хотите эффективно обработать их, не объединяя их в памяти в одну большую строгую строку.

Аналог конструктора `:` для строк байтов называется `cons`. Он принимает байт и строку байтов и помещает байт в начало строки.

```
ghci> B.cons 85 $ B.pack [80,81,82,84]
Chunk "U" (Chunk "PQRT" Empty)
```

Модули для работы со строками байтов содержат большое количество функций, аналогичных функциям в модуле `Data.List`, включая следующие (но не ограничиваясь ими): `head`, `tail`, `init`, `null`, `length`, `map`, `reverse`, `foldl`, `foldr`, `concat`, `takeWhile`, `filter` и др.

Есть и функции, имя которых совпадает с именем функций из модуля `System.IO`, и работают они аналогично, только строки заменены значениями типа `ByteString`. Например, функция `readFile` в модуле `System.IO` имеет тип

```
readFile :: FilePath -> IO String
```

а функция `readFile` из модулей для строк байтов имеет тип

```
readFile :: FilePath -> IO ByteString
```

ПРИМЕЧАНИЕ. *Обратите внимание, что если вы используете строгие строки и выполняете чтение файла, он будет считан в память целиком! При использовании ленивых байтовых строк файл будет читаться аккуратными порциями.*

Копирование файлов при помощи Bytestring

Давайте напишем простую программу, которая принимает два имени файла в командной строке и копирует первый файл во второй. Обратите внимание, что модуль `System.Directory` уже содержит функцию `copyFile`, но мы собираемся создать нашу собственную реализацию.

```
import System.Environment
import qualified Data.ByteString.Lazy as B
```

```
main = do
   (fileName1:fileName2:_) <- getArgs
   copy fileName1 fileName2

copy :: FilePath -> FilePath -> IO ()
copy source dest = do
   contents <- B.readFile source
   bracketOnError
      (openTemplFile "." "temp")
      (\(tempName, tempHandle) -> do
          hClose templHandle
          removeFile tempName)
      (\(tempName, tempHandle) -> do
          B.hPutStr tempHandle contents
          hClose tempHandle
          renameFile tempName dest)
```

В функции `main` мы получаем аргументы командной строки и вызываем функцию `copy`, в которой всё волшебство и происходит. Вообще говоря, можно было бы просто прочитать содержимое одного файла и записать его в другой. Однако если бы что-то пошло не так (например, закончилось бы место на диске), у нас в каталоге остался бы файл с некорректным содержимым. Поэтому мы пишем во временный файл, который в случае возникновения ошибки просто удаляется.

Сначала для чтения содержимого входного файла мы используем функцию `B.readFile`. Затем с помощью `bracketOnError` организуем обработку ошибок. Мы получаем ресурс посредством вызова `openTemplFile "." "temp"`, который возвращает пару из имени временного файла и его дескриптора. После этого указываем, что должно произойти при возникновении исключения. В этом случае мы закроем дескриптор и удалим временный файл. Наконец, выполняется собственно копирование. Для записи содержимого во временный файл используется функция `B.hPutStr`. Временный файл закрывается, и ему даётся имя, которое он должен иметь в итоге.

Заметьте, что мы использовали `B.readFile` и `B.hPutStr` вместо их обычных версий. Для открытия, закрытия и переименования файлов специальные функции не требуются. Они нужны только для чтения и записи.

Проверим программу:.

```
$ ./bytestringcopy bart.txt bort.txt
```

Обратите внимание, что программа, не использующая строки байтов, могла бы выглядеть точно так же. Единственное отличие – то, что мы используем `B.readFile` и `B.hPutStr` вместо `readFile` и `hPutStr`. Во многих случаях вы можете «переориентировать» программу, использующую обычные строки, на использование строк байтов, просто импортировав нужные модули и проставив имя модуля перед некоторыми функциями. В ряде случаев вам придётся конвертировать свои собственные функции для использования строк байтов, но это несложно.

Если вы хотите улучшить производительность программы, которая считывает много данных в строки, попробуйте использовать строки байтов; скорее всего, вы добьётесь значительного улучшения производительности, затратив совсем немного усилий. Обычно я пишу программы, используя обычные строки, а затем переделываю их на использование строк байтов, если производительность меня не устраивает.

Исключения[1]

В любой программе может встретиться фрагмент, который может отработать неправильно. Разные языки предлагают различные способы обработки подобных ошибок. В языке C мы обычно используем некоторое заведомо неправильное возвращаемое значение (например, –1 или пустой указатель), чтобы указать, что результат функции не должен рассматриваться как правильное значение. Языки Java и C#, с другой стороны, предлагают использовать для обработки ошибок механизм исключений. Когда возникает исключительная ситуация, выполнение программы передаётся некоему определённому нами участку кода, который выполняет ряд действий по восстановлению и, возможно, снова вызывает исключение, чтобы другой код для обработки ошибок мог выполниться и позаботиться о каких-либо других вещах.

В языке Haskell очень хорошая система типов. Алгебраические типы данных позволяют объявить такие типы данных, как `Maybe` и `Either`; мы можем использовать значения этих типов для представления результатов, которые могут отсутствовать. В языке C выбор,

[1] Текст этого раздела переработан в соответствии с современным стилем обработки исключений. – *Прим. ред.*

скажем, –1 для сигнала об ошибке – это просто предварительная договоренность. Эта константа имеет значение только для человека. Если мы не очень аккуратны, то можем трактовать подобные специальные значения как допустимые, и затем они могут привести к упадку и разорению вашего кода. Система типов языка Haskell даёт нам столь желанную безопасность в этом аспекте. Функция `a -> Maybe b` явно указывает, что результатом может быть значение типа `b`, завёрнутое в конструктор `Just`, или значение `Nothing`. Тип функции отличается от простого `a -> b`, и если мы попытаемся использовать один тип вместо другого, компилятор будет «жаловаться» на нас.

Кроме алгебраических типов, хорошо представляющих вычисления, которые могут закончиться неудачей, язык Haskell имеет поддержку исключительных ситуаций, так как они приобретают особое значение в контексте ввода-вывода. Всё может пойти вкривь и вкось, если вы работаете с внешним миром, который столь ненадёжен! Например, при открытии файла может случиться всякое. Он может быть заблокирован, его может не оказаться по заданному пути, или не будет такого диска, или ещё что-нибудь...

При возникновении исключительной ситуации хорошо бы иметь возможность перейти на некоторый код обработки ошибки. Хорошо, код для ввода-вывода (то есть «грязный» код) может вызывать исключения. Имеет смысл. Ну а как насчёт чистого кода? Он тоже может вызывать исключения! Вспомним функции `div` и `head`. Их типы – `(Integral a) => a -> a -> a` и `[a] -> a` соответственно. Никаких значений типа `Maybe` или `Either` в возвращаемом типе, и тем не менее они могут вызвать ошибку! Функция `div` взорвётся у вас в руках, если вы попытаетесь разделить на нуль, а функция `head` выпадет в осадок, если передать ей пустой список.

```
ghci> 4 `div` 0
*** Exception: divide by zero
ghci> head []
*** Exception: Prelude.head: empty list
```

Чистый код может выбрасывать исключения, но они могут быть перехвачены только в части кода, работающей с системой ввода-вывода (когда мы внутри блока `do` в функции `main`). Причина в том, что вы не знаете, когда что-то будет (если вообще будет!) вычислено в чистом коде, так как он ленив и не имеет жёстко определённого по-

рядка выполнения, в то время как код для ввода-вывода такой порядок имеет.

Раньше мы говорили, что нам желательно проводить как можно меньше времени в части нашей программы, посвящённой вводу-выводу. Логика программы должна располагаться главным образом в чистых функциях, поскольку их результат зависит только от параметров, с которыми функции были вызваны. При работе с чистыми функциями вы должны думать только о том, что функции возвращают, так как они не могут сделать чего-либо другого. Это облегчит вам жизнь!.. Даже несмотря на то, что некоторая логика в коде для ввода-вывода необходима (например, открытие файлов и т. п.), она должна быть сведена к минимуму. Чистые функции по умолчанию ленивы; следовательно, мы не знаем, когда они будут вычислены – это не должно иметь значения. Но как только чистые функции начинают вызывать исключения, становится важным момент их выполнения. Вот почему мы можем перехватывать исключения из чистых функций в части кода, посвящённой вводу-выводу. И это плохо: ведь мы стремимся оставить такую часть настолько маленькой, насколько возможно!... Однако если мы не перехватываем исключения, наша программа «падает». Решение? Не надо мешать исключения и чистый код! Пользуйтесь преимуществами системы типов языка Haskell и используйте типы вроде `Either` и `Maybe` для представления результатов, при вычислении которых может произойти ошибка.

Обработка исключений, возникших в чистом коде

В стандарте языка Haskell 98 года присутствует механизм обработки исключений ввода-вывода, который в настоящее время считается устаревшим. Согласно современному подходу все исключения, возникшие как при выполнении чистого кода, так и при осуществлении ввода-вывода, должны обрабатываться единообразно. Этой цели служит единая иерархия типов исключений из модуля `Control.Exception`, в которую легко можно включать собственные типы исключений. Любой тип исключения должен реализовывать экземпляр класса типов `Exception`. В модуле `Control.Exception` объявлено несколько конкретных типов исключений, среди которых `IOException` (исключения ввода-вывода), `ArithException` (арифметические ошибки, например, деление на ноль), `ErrorCall` (вызов функции `error`), `PatternMatchFail`

(не удалось выбрать подходящий образец в определении функции) и другие.

Простейший способ выполнить действие, которое потенциально может вызвать исключение,– воспользоваться функцией `try`:

```
try :: Exception e => IO a -> IO (Either e a)
```

Функция `try` пытается выполнить переданное ей действие ввода-вывода и возвращает либо `Right` <*результат действия*> либо `Left` <*исключение*>, например:

```
ghci> try (print $ 5 `div` 2) :: IO (Either ArithException ())
2
Right ()
ghci> try (print $ 5 `div` 0) :: IO (Either ArithException ())
Left divide by zero
```

Обратите внимание, что в данном случае потребовалось явно указать тип выражения, поскольку для вывода типа информации недостаточно. Помимо прочего, указание типа исключения позволяет обрабатывать не все исключения, а только некоторые. В следующем примере исключение функцией `try` обнаружено не будет:

```
> try (print $ 5 `div` 0) :: IO (Either IOException ())
*** Exception: divide by zero
```

Указание типа `SomeException` позволяет обнаружить любое исключение:

```
ghci> try (print $ 5 `div` 0) :: IO (Either SomeException ())
Left divide by zero
```

Попробуем написать программу, которая принимает два числа в виде параметров командной строки, делит первое число на второе и наоборот и выводит результаты. Нашей первой целью будет корректная обработка ошибки деления на ноль.

```
import Control.Exception
import System.Environment

printQuotients :: Integer -> Integer -> IO ()
printQuotients a b = do
  print $ a `div` b
```

```
  print $ b `div` a

params :: [String] -> (Integer, Integer)
params [a,b] = (read a, read b)

main = do
  args <- getArgs
  let (a, b) = params args
  res <- try (printQuotients a b) :: IO (Either ArithException ())
  case res of
    Left e -> putStrLn "Деление на 0!"
    Right () -> putStrLn "OK"
  putStrLn "Конец программы"
```

Погоняем программу на различных значениях:

```
$ ./quotients 20 7
2
0
OK
Конец программы
$ ./quotients 0 7
0
Деление на 0!
Конец программы
$ ./quotients 7 0
Деление на 0!
Конец программы
```

Понятно, что пока эта программа неустойчива к другим видам ошибок. В частности, мы можем «забыть» передать параметры командной строки или передать их не в том количестве:

```
$ ./quotients
quotients: quotients.hs:10:1-31: Non-exhaustive patterns in function params
$ ./quotients 2 3 4
quotients: quotients.hs:10:1-31: Non-exhaustive patterns in function params
```

Это исключение генерируется при вызове функции `params`, если переданный ей список оказывается не двухэлементным. Можно также указать нечисловые параметры:

```
$ ./quotients a b
quotients: Prelude.read: no parse
```

Исключение здесь генерируется функцией `read`, которая не в состоянии преобразовать переданный ей параметр к числовому типу.

Чтобы справиться с любыми возможными исключениями, выделим тело программы в отдельную функцию, оставив в функции `main` получение параметров командной строки и обработку исключений:

```
mainAction :: [String] -> IO ()
mainAction args = do
  let (a, b) = params args
  printQuotients a b

main = do
  args <- getArgs
  res <- try (mainAction args) :: IO (Either SomeException ())
  case res of
    Left e -> putStrLn "Ошибка"
    Right () -> putStrLn "OK"
  putStrLn "Конец программы"
```

Мы были вынуждены заменить тип исключения на `SomeException` и сделать сообщение об ошибке менее информативным, поскольку теперь неизвестно, исключение какого вида в данном случае произошло.

```
$ ./quotients a b
Ошибка
Конец программы
$ ./quotients
Ошибка
Конец программы
```

Понятно, что в общем случае обработка исключения должна зависеть от её типа. Предположим, что у нас имеется несколько обработчиков для исключений разных типов:

```
handleArith :: ArithException -> IO ()
handleArith _ = putStrLn "Деление на 0!"

handleArgs :: PatternMatchFail -> IO ()
handleArgs _ = putStrLn "Неверное число параметров командной строки!"

handleOthers :: SomeException -> IO ()
handleOthers e = putStrLn $ "Неизвестное исключение: " ++ show e
```

К сожалению, чтобы увидеть исключение от функции `read`, нужно воспользоваться наиболее общим типом `SomeException`.

Вместо того чтобы вручную вызывать функцию обработчика при анализе результата `try`, можно применить функцию `catch`, вот её тип:

```
ghci> :t catch
catch :: Exception e => IO a -> (e -> IO a) -> IO a
```

ПРИМЕЧАНИЕ. *Модуль `Prelude` экспортирует старую версию функции `catch`, которая способна обрабатывать только исключения ввода-вывода. Чтобы использовать новый вариант её определения, необходимо использовать скрывающий импорт: `import Prelude hiding (catch)`.*

Функция `catch` принимает в качестве параметров действие и обработчик исключения: если при выполнении действия генерируется исключение, то вызывается его обработчик. Тип обработчика определяет, какие именно исключения будут обработаны. Рассмотрим примеры, в которых функция `mainAction` вызывается непосредственно в GHCi:

```
ghci> mainAction ["2","0"]
*** Exception: divide by zero
ghci> mainAction ["0","2"] `catch` handleArith
0
Деление на 0!
ghci> mainAction ["2","0"] `catch` handleArgs
*** Exception: divide by zero
ghci> mainAction ["2","0"] `catch` handleOthers
Неизвестное исключение: divide by zero
ghci> mainAction ["a", "b"] `catch` handleArgs
*** Exception: Prelude.read: no parse
ghci> mainAction ["a", "b"] `catch` handleOthers
Неизвестное исключение: Prelude.read: no parse
```

Если строка, выводимая GHCi, начинается с `***`, то соответствующее исключение не было обработано. Обратите внимание на обычный для функции `catch` инфиксный способ вызова. Заметьте также, что обработчик `handleOthers` способен обработать любое исключение.

Вернёмся к основной программе. Нам хочется, чтобы возникшее исключение было обработано наиболее подходящим образом: если произошло деление на ноль, то следует выполнить `handleArith`, при неверном числе параметров командной строки – `handleArgs`, в остальных случаях – `handleOthers`. В этом нам поможет функция `catches`, посмотрим на её тип:

```
> :t catches
catches :: IO a -> [Handler a] -> IO a
```

Функция `catches` принимает в качестве параметров действие и список обработчиков (функций, которые упакованы конструктором данных `Handler`) и возвращает результат действия. Если в процессе выполнения происходит исключение, то вызывается первый из подходящих по типу исключения обработчиков (поэтому, в частности, обработчик `handleOthers` должен быть последним). Перепишем функцию `main` так, чтобы корректно обрабатывались все возможные исключительные ситуации:

```
main = do
  args <- getArgs
  mainAction args `catches`
                  [Handler handleArith,
                   Handler handleArgs,
                   Handler handleOthers]
  putStrLn "Конец программы"
```

Посмотрим, как она теперь работает:

```
$ ./quotients 20 10
2
0
Конец программы
$ ./quotients
Неверное число параметров командной строки!
Конец программы
$ ./quotients 2 0
Деление на 0!
Конец программы
$ ./quotients a b
Неизвестное исключение: Prelude.read: no parse
Конец программы
```

В этом разделе мы разобрались с работой функций `try`, `catch` и `catches`, позволяющих обработать исключение, в том числе и возникшее в чистом коде. Заметьте ещё раз, что вся обработка выполнялась в рамках действий ввода-вывода. Посмотрим теперь, как работать с исключениями, которые возникают при выполнении операций ввода-вывода.

Обработка исключений ввода-вывода

Исключения ввода-вывода происходят, когда что-то пошло не так при взаимодействии с внешним миром в действии ввода-вывода, являющемся частью функции `main`. Например, мы пытаемся открыть файл, и тут оказывается, что он был удалён, или ещё что-нибудь в этом духе. Посмотрите на программу, открывающую файл, имя которого передаётся в командной строке, и говорящую нам, сколько строк содержится в файле:

```
import System.Environment
import System.IO

main = do
   (fileName:_) <- getArgs
   contents <- readFile fileName
   putStrLn $ "В этом файле " ++ show (length (lines contents)) ++
              " строк!"
```

Очень простая программа. Мы выполняем действие ввода- вывода `getArgs` и связываем первую строку в возвращённом списке с идентификатором `fileName`. Затем связываем имя `contents` с содержимым файла. Применяем функцию `lines` к `contents`, чтобы получить список строк, считаем их количество и передаём его функции `show`, чтобы получить строковое представление числа. Это работает – но что получится, если передать программе имя несуществующего файла?

```
$ ./linecount dont_exist.txt
linecount: dont_exist.txt: openFile: does not exist (No such file or directory)
```

Ага, получили ошибку от GHC с сообщением, что файла не существует! Наша программа «упала». Но лучше бы она печатала красивое сообщение, если файл не найден. Как этого добиться? Можно про-

верять существование файла, прежде чем попытаться его открыть, используя функцию doesFileExist из модуля System.Directory.

```
import System.Environment
import System.IO
import System.Directory

main = do
   (fileName:_) <- getArgs
   fileExists <- doesFileExist fileName
   if fileExists
      then do
         contents <- readFile fileName
         putStrLn $ "В этом файле " ++
                    show (length (lines contents)) ++
                    " строк!"
      else putStrLn "Файл не существует!"
```

Мы делаем вызов fileExists <- doesFileExist fileName, потому что функция doesFileExist имеет тип doesFileExist :: FilePath -> IO Bool; это означает, что она возвращает действие ввода-вывода, содержащее булевское значение, которое говорит нам, существует ли файл. Мы не можем напрямую использовать функцию doesFileExist в условном выражении.

Другим решением было бы использовать исключения. В этом контексте они совершенно уместны. Ошибка при отсутствии файла происходит в момент выполнения действия ввода-вывода, так что его перехват в секции ввода-вывода лёгок и приятен. К тому же, обработка исключений позволяет сделать этот код менее громоздким:

```
import Prelude hiding (catch)
import Control.Exception
import System.Environment

countLines :: String -> IO ()
countLines fileName = do
  contents <- readFile fileName
  putStrLn $ "В этом файле " ++ show (length (lines contents)) ++ "
            строк!"

handler :: IOException -> IO ()
```

```
handler e = putStrLn "У нас проблемы!"

main = do
   (fileName:_) <- getArgs
   countLines fileName `catch` handler
```

Здесь мы определяем обработчик `handler` для всех исключений ввода-вывода и пользуемся функцией `catch` для перехвата исключения, возникающего в функции `countLines`.

Попробуем:

```
$ ./linecount linecount.hs
В этом файле 17 строк!
$ ./linecount dont_exist.txt
У нас проблемы!
```

Исключение ввода-вывода может быть вызвано целым рядом причин, среди которых, помимо отсутствия файла, может быть также отсутствие права на чтение файла или вообще отказ жёсткого диска. В обработчике мы не проверяли, какой вид исключения `IOException` получили. Мы просто возвращаем строку `"У нас проблемы"`, что бы ни произошло.

Простой перехват всех типов исключений в одном обработчике – плохая практика в языке Haskell, так же как и в большинстве других языков. Что если произошло какое-либо другое исключение, которое мы не хотели бы перехватывать, например прерывание программы? Вот почему мы будем делать то же, что делается в других языках: проверять, какой вид исключения произошел. Если это тот вид, который мы ожидали перехватить, вызовем обработчик. Если это нечто другое, мы не мешаем исключению распространяться далее. Давайте изменим нашу программу так, чтобы она перехватывала только исключение, вызываемое отсутствием файла:

```
import Prelude hiding (catch)
import Control.Exception
import System.Environment
import System.IO.Error (isDoesNotExistError)

countLines :: String -> IO ()
countLines fileName = do
   contents <- readFile fileName
   putStrLn $ "В этом файле " ++ show (length (lines contents)) ++
```

```
            " строк!"

handler :: IOException -> IO ()
handler e
   | isDoesNotExistError e = putStrLn "Файл не существует!"
   | otherwise = ioError e

main = do
   (fileName:_) <- getArgs
   countLines fileName `catch` handler
```

Программа осталась той же самой, но поменялся обработчик, который мы изменили таким образом, что он реагирует только на одну группу исключений ввода-вывода. С этой целью мы воспользовались предикатом `isDoesNotExistError` из модуля `System.IO.Error`. Мы применяем его к исключению, переданному в обработчик, чтобы определить, было ли исключение вызвано отсутствием файла. В данном случае мы используем охранные выражения, но могли бы использовать и условное выражение `if-then-else`. Если исключение вызвано другими причинами, перевызываем исключение с помощью функции `ioError`.

> **ПРИМЕЧАНИЕ.** *Функции `try`, `catch`, `ioError` и некоторые другие объявлены одновременно в модулях `System.IO.Error` (устаревший вариант) и `Control.Exception` (современный вариант), поэтому подключение обоих модулей (например, для использования предикатов исключений ввода-вывода) требует скрывающего или квалифицированного импорта либо же, как в предыдущем примере, явного указания импортируемых функций.*

Итак, исключение, произошедшее в действии ввода-вывода `countLines`, но не по причине отсутствия файла, будет перехвачено и перевызвано в обработчике:

```
$ ./linecount dont_exist.txt
Файл не существует!
$ ./linecount norights.txt
linecount: noaccess.txt: openFile: permission denied (Permission denied)
```

Существует несколько предикатов, предназначенных для определения вида исключения ввода-вывода:

- `isAlreadyExistsError` (файл уже существует);
- `isDoesNotExistError` (файл не существует);

- isAlreadyInUseError (файл уже используется);
- isFullError (не хватает места на диске);
- isEOFError (достигнут конец файла);
- isIllegalOperation (выполнена недопустимая операция);
- isPermissionError (недостаточно прав доступа).

Пользуясь этими предикатами, можно написать примерно такой обработчик:

```
handler :: IOException -> IO ()
handler e
   | isDoesNotExistError e = putStrLn "Файл не существует!"
   | isPermissionError e = putStrLn "Не хватает прав доступа!"
   | isFullError e = putStrLn "Освободите место на диске!"
   | isIllegalOperation e = putStrLn "Караул! Спасите!"
   | otherwise = ioError e
```

Убедитесь, что вы перевызываете исключение, если оно не подходит под ваши критерии; в противном случае ваша программа иногда будет «падать» молча, что крайне нежелательно.

Модуль System.IO.Error также экспортирует функции, которые позволяют нам получать атрибуты исключения, например дескриптор файла, вызвавшего исключение, или имя файла. Все эти функции начинаются с префикса ioe; их полный список вы можете найти в документации. Скажем, мы хотим напечатать имя файла в сообщении об ошибке. Значение fileName, полученное при помощи функции getArgs, напечатать нельзя, потому что в обработчик передаётся только значение типа IOException и он не знает ни о чём другом. Функция зависит только от своих параметров. Но мы можем вызвать функцию ioeGetFileName, которая по переданному ей исключению возвращает Maybe FilePath. Функция пытается получить из значения исключения имя файла, если такое возможно. Давайте изменим обработчик так, чтобы он печатал полное имя файла, из-за которого возникло исключение (не забудьте включить функцию ioeGetFileName в список импорта для модуля System.IO.Error):

```
handler :: IOException -> IO ()
handler e
   | isDoesNotExistError e =
      case ioeGetFileName e of
         Just fileName -> putStrLn $ "Файл " ++ fileName ++
                                     " не существует!"
```

```
        Nothing -> putStrLn "Файл не существует!"
    | otherwise = ioError e
  where fileName = ioeGetFileName e
```

В охранном выражении, если предикат isDoesNotExistError вернёт значение True, мы использовали выражение case, чтобы вызвать функцию ioeGetFileName с параметром e; затем сделали сопоставление с образцом по возвращённому значению с типом Maybe. Выражение case часто используется в случаях, когда вам надо сделать сопоставление с образцом, не создавая новую функцию. Посмотрим, как это сработает:

```
$ ./linecount dont_exist.txt
Файл dont_exists.txt не существует!
```

Вы не обязаны использовать один обработчик для перехвата всех исключений в части кода, работающей с системой ввода-вывода. Вы можете перекрыть только отдельные части кода с помощью функции catch или перекрывать разные участки кода разными обработчиками, например так:

```
main = do
   action1 `catch` handler1
   action2 `catch` handler2
   launchRockets
```

Функция action1 использует функцию handler1 в качестве обработчика, а функция action2 использует handler2. Функция launchRockets не является параметром функции catch, так что любое сгенерированное в ней исключение обрушит нашу программу, если только эта функция не использует try или catch внутри себя для обработки собственных ошибок. Конечно же, action1, action2 и launchRockets – это действия ввода-вывода, которые «склеены» друг с другом блоком do и, вероятно, определены где-то в другом месте. Это похоже на блоки try-catch в других языках: вы можете поместить всю вашу программу в один блок try-catch или защищать отдельные участки программы и перехватывать различные исключения для разных участков.

Вспомогательные функции для работы с исключениями

Ранее в этой главе мы уже познакомились с функциями `bracket` и `bracketOnError`, которые реализуют наиболее часто используемый сценарий обработки исключений, когда работа с ресурсом состоит из трёх стадий:

- получение ресурса;
- использование ресурса;
- освобождение ресурса.

В наших примерах на первой стадии открывался файл, на второй шла работа с его содержимым, а на третьей файл закрывался. Функция `bracket` гарантировала выполнение всех трёх действий, даже если в процессе генерировалось исключение, а функция `bracketOnError` запускала третье действие только в случае возникновения исключения.

Обратите внимание, что программист, использующий такого рода функции, не работает непосредственно с исключениями – ему лишь достаточно понимать логику и порядок вызова конкретных действий.

Модуль `Control.Exception` содержит ещё несколько подобных функций. Функция `finally` обеспечивает гарантированное выполнение некоторого действия по завершении другого действия. Это всего навсего упрощённый вариант функции `bracket`. Вот её сигнатура:

```
finally :: IO a -> IO b -> IO a
```

В следующем примере текст `"Готово!"` печатается в каждом из двух случаев, несмотря на возникновение исключения во втором:

```
ghci> print (20 `div` 10) `finally` putStrLn "Готово!"
2
Готово!
ghci> print (2 `div` 0) `finally` putStrLn "Готово!"
Готово!
*** Exception: divide by zero
```

Функция `onException` позволяет выполнить заключительное действие только в случае возникновения исключения:

```
ghci> print (20 `div` 10) `onException` putStrLn "Ошибка!"
2
ghci> print (2 `div` 0) `finally` putStrLn "Ошибка!"
Ошибка!
*** Exception: divide by zero
```

Заметьте, что обе эти функции, в отличие от `try` или `catch`, не обрабатывают исключения – они лишь гарантируют выполнение указанных действий. Все эти функции нетрудно реализовать вручную, пользуясь лишь `try` или `catch`. Фактически они устанавливают свой обработчик, перехватывают исключение, выполняют заданные действия, а после этого повторно генерируют то же самое исключение. Тем не менее, если ваша задача соответствует одному из приведённых сценариев, стоит воспользоваться уже существующей функцией.

10
РЕШЕНИЕ ЗАДАЧ В ФУНКЦИОНАЛЬНОМ СТИЛЕ

В этой главе мы рассмотрим пару интересных задач и узнаем, как мыслить функционально для того, чтобы решить их по возможности элегантно. Скорее всего, мы не будем вводить новых концепций, а просто используем вновь приобретённые навыки работы с языком Haskell и попрактикуем методы программирования. Каждый раздел представляет отдельную задачу. Мы будем давать её описание и предложим поиск лучшего (или не самого худшего) решения.

Вычисление выражений в обратной польской записи

Обычно мы записываем математические выражения в инфиксной нотации, например: `10 - (4 + 3) * 2`. Здесь `+`, `*` и `-` представляют собой инфиксные операторы, такие же, как инфиксные функции Haskell (`+`, `` `elem` `` и т. д.). Так нам удобнее, потому что мы можем легко разобрать подобную формулу в уме. Но у такой записи есть и негативное свойство: приходится использовать скобки для обозначения приоритета операций.

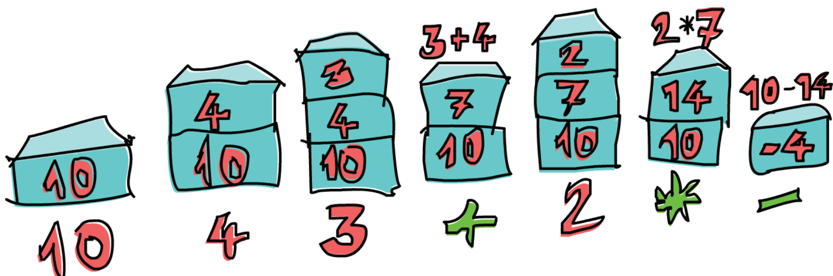
Обратная польская запись (ОПЗ) является одним из способов записи математических выражений. В ОПЗ операторы записываются не между числами, а после них. Так, вместо 4 + 3 нужно писать 4 3 +. Но как тогда записать выражения, содержащие несколько

операторов? Например, как бы мы записали выражение, складывающее 4 и 3, а потом умножающее сумму на 10? Легко: 4 3 + 10 *. Поскольку 4 3 + равно 7, то всё выражение равно 7 10 *, т. е. 70. Поначалу такая запись воспринимается с трудом, но её довольно просто понять и использовать, так как необходимости в скобках нет и произвести вычисление очень легко. Хотя большинство современных калькуляторов используют инфиксную нотацию, некоторые люди до сих пор являются приверженцами калькуляторов, использующих ОПЗ.

Вычисление выражений в ОПЗ

Как мы можем вычислить результат? Представьте себе стек. Вы проходите по выражению слева направо. Если текущий элемент – число, его надо поместить (*push* – «втолкнуть») в стек. Если мы рассматриваем оператор, необходимо взять (*pop* – «вытолкнуть») два числа с вершины стека, применить к ним оператор и втолкнуть результат обратно в стек. Когда вы достигнете конца выражения, у вас должно остаться одно число, если, конечно, выражение было записано правильно. Это число и будет результатом.

Давайте разберём выражение 10 4 3 + 2 * –. Сначала мы помещаем 10 в стек; в стеке теперь содержится одно число. Следующий элемент – число 4, которое мы также помещаем в стек. То же проделываем со следующей тройкой – стек теперь содержит 10, 4, 3. И наконец-то нам встречается оператор, а именно «плюс». Мы выталкиваем предыдущие два числа из стека (в стеке остаётся 10), складываем их, помещаем результат в стек. Теперь в стеке 10, 7. Заталкиваем 2 в стек, теперь там 10, 7, 2. Мы снова дошли до оператора; вытолкнем 7 и 2 из стека, перемножим их, положим результат в стек. Умножение 7 на 2 даст 14; в стеке будет 10, 14. Получаем последний



оператор – «минус». Выталкиваем 10 и 14 из стека, вычитаем 10 из 14, получаем –4, помещаем число в стек, и так как у нас больше нет чисел и операторов для разбора, мы получили конечный результат!

Теперь, когда мы знаем, как вычислять выражения на ОПЗ вручную, давайте подумаем, как бы нам написать функцию на языке Haskell, которая делает то же самое.

Реализация функции вычисления выражений в ОПЗ

Наша функция будет принимать строку, содержащую выражение в обратной польской записи, например, `"10 4 3 + 2 * -"`, и возвращать нам результат вычисления этого выражения.

Каков может быть тип такой функции? Мы хотим, чтобы она принимала строку и возвращала число. Давайте договоримся, что результат должен быть вещественным числом, потому что среди других операторов хочется иметь и деление. Тип может быть приблизительно таким:

```
solveRPN :: String -> Double
```

ПРИМЕЧАНИЕ. *В процессе работы очень полезно сначала подумать о том, какой будет декларация типа функции, и записать её, прежде чем приступать к её реализации. В языке Haskell декларация типа функции говорит нам очень многое о функции благодаря строгой системе типов.*

Отлично. При реализации решения проблемы на языке Haskell хорошо припомнить, как вы делали это вручную, и попытаться выделить какую-то идею. В нашем случае мы видим, что каждое число и оператор рассматривались как отдельные элементы. Так что будет полезно разбить строку вида `"10 4 3 + 2 * -"` на список элементов:

```
["10","4","3","+","2","*","-"]
```

Идём дальше. Что мы мысленно делали со списком элементов? Мы проходили по нему слева направо и работали со стеком по мере

прохождения списка. Последнее предложение ничего не напоминает? Помните, в главе, посвящённой свёрткам, мы говорили, что практически любая функция, которая проходит список слева направо или справа налево, один элемент за другим, и накапливает (аккумулирует) некоторый результат – неважно, число, список или стек, – может быть реализована в виде свёртки?

В нашем случае будем использовать левую свёртку, поскольку мы проходим список слева направо. Аккумулятором будет стек, и, следовательно, результатом свёртки также будет стек, но, как мы видели, он будет содержать единственный элемент.

Ещё одна вещь, о которой стоит подумать: а как мы будем реализовывать стек? Я предлагаю использовать список. Также рекомендую в качестве вершины стека использовать «голову» списка – потому что добавление элемента к «голове» (началу) списка работает гораздо быстрее, чем добавление элемента к концу списка. В таком случае, если у нас, например, есть стек `10, 4, 3`, мы представим его списком `[3,4,10]`.

Теперь мы знаем достаточно для того, чтобы написать черновик функции. Она будет принимать строку, например `"10 4 3 + 2 * -"`, разбивать её на элементы и формировать из них список, используя функцию `words`. Получится `["10","4","3","+","2","*","-"]`. Далее мы выполним левую свёртку и в конце получим стек, содержащий единственный элемент, `[-4]`. Мы получим этот элемент из списка; он и будет окончательным результатом.

Вот черновик нашей функции:

```
solveRPN :: String -> Double
solveRPN expr = head (foldl foldingFunction [] (words expr))
   where foldingFunction stack item = ...
```

Мы принимаем выражение и превращаем его в список элементов. Затем выполняем свёртку, используя некоторую функцию. Обратите внимание на `[]`: это начальное значение аккумулятора. Аккумулятором будет стек – следовательно, `[]` представляет пустой стек, каковым он и должен быть в самом начале. После получения результирующего списка с единственным элементом мы вызываем функцию `head` для получения первого элемента.

Всё, что осталось, – реализовать функцию для свёртки, которая будет принимать стек, например `[4,10]`, элемент, например `"3"`, и возвращать новый стек, `[3,4,10]`. Если стек содержит `[4,10]`,

а элемент равен *, то функция должна вернуть [40]. Но прежде всего давайте перепишем функцию в бесточечном стиле, так как она содержит множество скобок: лично меня они бесят!

```
solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words
   where foldingFunction stack item = ...
```

То-то! Намного лучше. Итак, функция для свёртки принимает стек и элемент и возвращает новый стек. Мы будем использовать сопоставление с образцом для того, чтобы получать первые элементы стека, и для сопоставления с операторами, например * и -.

```
solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words
   where
      foldingFunction (x:y:ys) "*" = (x * y):ys
      foldingFunction (x:y:ys) "+" = (x + y):ys
      foldingFunction (x:y:ys) "-" = (y - x):ys
      foldingFunction xs numberString = read numberString:xs
```

Мы уложились в четыре образца. Образцы будут сопоставляться транслятором в порядке записи. Вначале функция свёртки проверит, равен ли текущий элемент "*". Если да, то функция возьмёт список, например [3,4,9,3], и присвоит двум первым элементам имена x и y соответственно. В нашем случае x будет соответствовать тройке, а y – четвёрке; ys будет равно [9,3]. В результате будет возвращён список, состоящий из [9,3], и в качестве первого элемента будет добавлено произведение тройки и четвёрки. Таким образом, мы выталкиваем два первых числа из стека, перемножаем их и помещаем результат обратно в стек. Если элемент не равен "*", сопоставление с образцом продолжается со следующего элемента, проверяя "+", и т. д.

Если элемент не совпадёт ни с одним оператором, то мы предполагаем, что это строка, содержащая число. Если это так, то мы вызываем функцию read с этой строкой, чтобы получить число, добавляем его в вершину предыдущего стека и возвращаем получившийся стек.

Для списка ["2","3","+"] наша функция начнёт свёртку с самого левого элемента. Стек в начале пуст, то есть представляет собой []. Функция свёртки будет вызвана с пустым списком в качестве стека (аккумулятора) и "2" в качестве элемента. Так как этот элемент не

является оператором, он будет просто добавлен в начало стека `[]`. Новый стек будет равен `[2]`, функция свёртки будет вызвана со значением `[2]` в качестве стека и `"3"` в качестве элемента; функция вернёт новый стек, `[3,2]`. Затем функция свёртки вызывается в третий раз, со стеком равным `[3,2]` и элементом `"+"`. Это приводит к тому, что оба числа будут вытолкнуты из стека, сложены, а результат будет помещён обратно в стек. Результирующий стек равен `[5]` – это число мы вернём.

Погоняем нашу функцию:

```
ghci> solveRPN "10 4 3 + 2 * -"
-4.0
ghci> solveRPN "2 3.5 +"
5.5
ghci> solveRPN "90 34 12 33 55 66 + * - +"
-3947.0
ghci> solveRPN "90 34 12 33 55 66 + * - + -"
4037.0
ghci> solveRPN "90 3.8 -"
86.2
```

Отлично, работает!

Добавление новых операторов

Чем ещё хороша наша функция – её можно легко модифицировать для поддержки других операторов. Операторы не обязательно должны быть бинарными. Например, мы можем создать оператор `log`, который выталкивает из стека одно число и заталкивает обратно его логарифм. Также можно создать тернарный оператор, который будет извлекать из стека три числа и помещать обратно результат. Или, к примеру, реализовать оператор `sum`, который будет поднимать все числа из стека и суммировать их.

Давайте изменим нашу функцию так, чтобы она понимала ещё несколько операторов.

```
solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words
   where
      foldingFunction (x:y:ys) "*" = (x * y):ys
      foldingFunction (x:y:ys) "+" = (x + y):ys
      foldingFunction (x:y:ys) "-" = (y - x):ys
      foldingFunction (x:y:ys) "/" = (y / x):ys
```

```
foldingFunction (x:y:ys) "^" = (y ** x):ys
foldingFunction (x:xs) "ln" = log x:xs
foldingFunction xs "sum" = [sum xs]
foldingFunction xs numberString = read numberString:xs
```

Прекрасно. Здесь / – это, конечно же, деление, и ** – возведение в степень для действительных чисел. Для логарифма мы осуществляем сравнение с образцом для одного элемента и «хвоста» стека, потому что нам нужен только один элемент для вычисления натурального логарифма. Для оператора суммы возвращаем стек из одного элемента, который равен сумме элементов, находившихся в стеке до этого.

```
ghci> solveRPN "2.7 ln"
0.9932517730102834
ghci> solveRPN "10 10 10 10 sum 4 /"
10.0
ghci> solveRPN "10 10 10 10 10 sum 4 /"
12.5
ghci> solveRPN "10 2 ^"
100.0
```

На мой взгляд, это делает функцию, способную вычислять произвольное выражение в обратной польской записи с дробными числами, которое может быть расширено 10 строчками кода, просто-таки расчудесной.

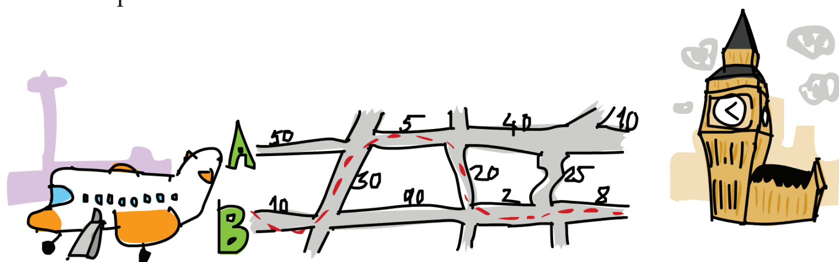
> **ПРИМЕЧАНИЕ.** *Как можно заметить, функция не устойчива к ошибкам. Если передать ей бессмысленный вход, она вывалится с ошибкой. Мы сделаем её устойчивой к ошибкам, определив её тип как `solveRPN :: String -> Maybe Double`, как только разберёмся с монадами (они не страшные, честно!). Можно было бы написать безопасную версию функции прямо сейчас, но довольно-таки скучным будет сравнение с `Nothing` на каждом шаге. Впрочем, если у вас есть желание, попробуйте! Подсказка: можете использовать функцию `reads`, чтобы проверить, было ли чтение успешным.*

Из аэропорта в центр

Рассмотрим такую ситуацию. Ваш самолёт только что приземлился в Англии, и у вас арендована машина. В скором времени запланировано совещание, и вам надо добраться из аэропорта Хитроу в Лондон настолько быстро, насколько это возможно (но без риска!).

Существуют две главные дороги из Хитроу в Лондон, а также некоторое количество более мелких дорог, пересекающих главные. Путь от одного перекрёстка до другого занимает чётко определённое время. Выбор оптимального пути возложен на вас: ваша задача – добраться до Лондона самым быстрым способом! Вы начинаете с левой стороны и можете переехать на соседнюю главную дорогу либо ехать прямо.

Как видно по рисунку, самый короткий путь – начать движение по главной дороге B, свернуть на A, проехав немного, вернуться на B и снова ехать прямо. В этом случае дорога занимает 75 минут. Если бы мы выбрали любой другой путь, нам потребовалось бы больше времени.



Наша задача – создать программу, которая примет на вход некоторое представление системы дорог и напечатает кратчайший путь. Вот как может выглядеть входная информация в нашем случае:

```
50
10
30
5
90
20
40
2
25
10
8
0
```

Чтобы разобрать входной файл в уме, представьте его в виде дерева и разбейте систему дорог на секции. Каждая секция состоит из дороги A, дороги B и пересекающей дороги. Чтобы предста-

вить это в виде дерева, мы предполагаем, что есть последняя замыкающая секция, которую можно проехать за 0 секунд, так как нам неважно, откуда именно мы въедем в город: важно только, что мы в городе.

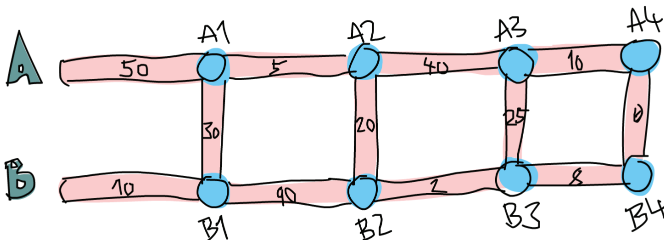
Будем решать проблему за три шага – так же мы поступали при создании вычислителя выражений в ОПЗ:

1. На минуту забудьте о языке Haskell и подумайте, как бы вы решали эту задачу в уме. При решении предыдущей задачи мы выясняли, что для вычисления в уме нам нужно держать в памяти некоторое подобие стека и проходить выражение по одному элементу за раз.
2. Подумайте, как вы будете представлять данные в языке Haskell. В вычислителе ОПЗ мы решили представлять выражение в виде списка строк.
3. Выясните, как манипулировать данными в языке Haskell так, чтобы получить результат. В прошлом разделе мы воспользовались левой свёрткой списка строк, используя стек в качестве аккумулятора свёртки.

Вычисление кратчайшего пути

Итак, как мы будем искать кратчайший путь от Хитроу до Лондона, не используя программных средств? Мы можем посмотреть на картинку, прикинуть, какой путь может быть оптимальным – и, вероятно, сделаем правильное предположение… *Вероятно*, если дорога небольшая; ну а если у неё насчитывается 10 000 секций? Ого! К тому же мы не будем знать наверняка, что наше решение оптимально: можно лишь сказать, что мы более или менее в этом уверены. Следовательно, это плохое решение.

Посмотрим на упрощённую карту дорожной системы. Можем ли мы найти кратчайший путь до первого перекрёстка (первая точ-



ка на A, помеченная A1)? Это довольно просто. Легко увидеть, что будет быстрее – проехать по A или проехать по B и повернуть на A. Очевидно, что выгоднее ехать по B и поворачивать: это займет 40 минут, в то время как езда напрямую по дороге A займет 50 минут. Как насчёт пересечения B1? То же самое! Значительно выгоднее ехать по B (включая 10 минут), так как путь по A вместе с поворотом займет целых 80 минут.

Теперь мы знаем, что кратчайший путь до A1 – это движение по дороге B и переезд на дорогу A по отрезку, который мы назовём C (общее время 40 минут), а также знаем кратчайший путь до B1 – проезд по дороге B (10 минут). Поможет ли нам это, если нужно узнать кратчайший путь до следующего перекрёстка? Представьте себе, да!

Найдём кратчайший путь до пункта A2. Мы можем проехать до A2 из A1 напрямую или ехать через B1 (далее – до B2 либо повернуть на перпендикулярную дорогу). Поскольку мы знаем время пути до A1 и B1, можно легко определить кратчайший путь до A2. Наименьшее время пути до A1 – 40 минут, и ещё за 5 минут мы доберёмся до A2; в результате минимальное время пути на отрезке B-C-A составит 45 минут. Время пути до B1 – всего 10 минут, но затем потребуется ещё целых 110, чтобы добраться до B2 и проехать поворот. Очевидно, кратчайший путь до A2 – это B-C-A. Аналогично кратчайший путь до B2 – проезд до A1 и поворот на другую дорогу.

ПРИМЕЧАНИЕ. *Возможно, вы задались вопросом: а что если добраться до A2, переехав на B1 и затем двигаясь прямо? Но мы уже рассмотрели переезд из B1 в A1, когда искали лучший путь до A1, так что нам больше не нужно анализировать этот вариант.*

Итак, мы вычислили кратчайшие пути до A2 и B2. Продолжать в том же духе можно до бесконечности, пока мы не достигнем последней точки. Как только мы выясним, как быстрее всего попасть в пункты A4 и B4, можно будет определить самый короткий путь – он и будет оптимальным.

В общем-то для второй секции мы повторяли те же шаги, что и для первой, но уже принимая во внимание предыдущие кратчайшие пути до A и B. Мы можем сказать, что на первом шаге наилучшие пути были пустыми, с «нулевой стоимостью».

Подведём итог. Чтобы вычислить наилучший путь от Хитроу до Лондона, для начала следует найти кратчайший путь до перекрёстка на дороге A. Есть два варианта: сразу ехать по A или двигаться по

параллельной дороге и затем сворачивать на дорогу A. Мы запоминаем время и маршрут. Затем используем тот же метод для нахождения кратчайшего пути до следующего перекрёстка дороги B и запоминаем его. Наконец, смотрим, как выгоднее ехать до следующего перекрёстка на дороге A: сразу по A или по дороге B с поворотом на A. Запоминаем кратчайший путь и производим те же расчёты для параллельной дороги. Так мы анализируем все секции, пока не достигнем конца. Когда все секции пройдены, самый короткий из двух путей можно считать оптимальным.

Вкратце: мы определяем один кратчайший путь по дороге A и один кратчайший путь по дороге B; когда мы достигаем точки назначения, кратчайший из двух путей и будет искомым. Теперь мы знаем, как решать эту задачу в уме. Если у вас достаточно бумаги, карандашей и свободного времени, вы можете вычислить кратчайший путь в дорожной сети с любым количеством секций.

Представление пути на языке Haskell

Следующий наш шаг: как представить дорожную систему в системе типов языка Haskell? Один из способов – считать начальные точки и перекрёстки узлами графа, которые ведут к другим перекрёсткам. Если мы представим, что начальные точки связаны друг с другом дорогой единичной длины, мы увидим, что каждый перекрёсток (или узел графа) указывает на узел на противоположной стороне, а также на следующий узел с той же стороны. За исключением последних узлов они просто показывают на противоположную сторону.

```
data Node = Node Road Road | EndNode Road

data Road = Road Int Node
```

Узел – это либо обычный узел, указывающий путь до противоположной дороги и путь до следующего узла по той же дороге, либо конечный узел, который содержит информацию *только* о противоположной дороге. Дорога хранит информацию о длине отрезка и об узле, к которому она ведёт. Например, первая часть дороги A будет представлена как `Road 50 a1`, где `a1` равно `Node x y`; при этом `x` и `y` – дороги, которые ведут к `B1` и `A2`.

Мы могли бы использовать тип `Maybe` для определения данных `Road`, которые указывают путь по той же дороге. Все узлы содержат

путь до параллельной дороги, но только те узлы, которые не являются конечными, содержат пути, ведущие вперёд.

```
data Node = Node Road (Maybe Road)

data Road = Road Int Node
```

Можно решить задачу, пользуясь таким способом представления дорожной системы; но нельзя ли придумать что-нибудь попроще? Если вспомнить решение задачи в уме, мы всегда проверяли длины трёх отрезков дороги: отрезок по дороге `A`, отрезок по дороге `B` и отрезок `C`, который их соединяет. Когда мы искали кратчайший путь к пунктам `A1` и `B1`, то рассматривали длины первых трёх частей, которые были равны 50, 10 и 30. Этот участок сети дорог назовём секцией. Таким образом, дорожная система в нашем примере легко может быть представлена в виде четырёх секций: `(50, 10, 30)`, `(5, 90, 20)`, `(40, 2, 25)` и `(10, 8, 0)`.

Всегда лучше делать типы данных настолько простыми, насколько это возможно – но не проще!

```
data Section = Section { getA :: Int, getB :: Int, getC :: Int }
   deriving (Show)

type RoadSystem = [Section]
```

Так гораздо ближе к идеалу! Записывается довольно просто, и у меня есть предчувствие, что для решения нашей задачи такое описание подойдёт отлично. Секция представлена обычным алгебраическим типом данных, который содержит три целых числа для представления длин трёх отрезков пути. Также мы определили синоним типа, который говорит, что `RoadSystem` представляет собой список секций.

ПРИМЕЧАНИЕ. *Для представления секции дороги мы могли бы использовать кортеж из трёх целых чисел:* `(Int, Int, Int)`. *Кортежи вместо алгебраических типов данных лучше применять для решения маленьких локальных задач, но в таких случаях, как наш, лучше создать новый тип. Это даёт системе типов больше информации о том, что есть что. Мы можем использовать* `(Int, Int, Int)` *для представления секции дороги или вектора в трёхмерном пространстве и оперировать таким представлением, но тут не исключена путаница. А вот если использовать типы данных* `Section` *и* `Vector`*, мы не сможем случайно сложить вектор с секцией дорожной системы.*

Теперь дорожная система между Хитроу и Лондоном может быть представлена так:

```
heathrowToLondon :: RoadSystem
heathrowToLondon = [ Section 50 10 30
                   , Section 5 90 20
                   , Section 40 2 25
                   , Section 10 8 0
                   ]
```

Всё, что нам осталось сделать, – разработать решение на языке Haskell.

Реализация функции поиска оптимального пути

Какой может быть декларация типа для функции, вычисляющей кратчайший путь для дорожной системы? Она должна принимать дорожную систему как параметр и возвращать путь. Мы будем представлять путь в виде списка. Давайте определим тип `Label`, который может принимать три фиксированных значения: `A`, `B` или `C`. Также создадим синоним типа – `Path`.

```
data Label = A | B | C deriving (Show)
type Path = [(Label, Int)]
```

Наша функция, назовём её `optimalPath`, будет иметь такую декларацию типа:

```
optimalPath :: RoadSystem -> Path
```

Если вызвать её с дорожной системой `heathrowToLondon`, она должна вернуть следующий путь:

```
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8)]
```

Мы собираемся пройти по списку секций слева направо и сохранять оптимальные пути по `A` и `B` по мере обхода списка. Будем накапливать лучшие пути по мере обхода списка – слева направо... На что это похоже? Тук-тук-тук! Правильно, *левая свёртка*!

При решении задачи вручную был один шаг, который мы повторяли раз за разом. Мы проверяли оптимальные пути по `A` и `B` на текущий момент и текущую секцию, чтобы найти новый оптимальный путь по `A` и `B`. Например, вначале оптимальные пути по `A` и `B` равны, соответственно, `[]` и `[]`. Мы проверяем секцию `Section 50 10 30` и решаем, что новый оптимальный путь до `A1` – это

[(B,10),(C,30)]; оптимальный путь до B1 – это [(B,10)]. Если посмотреть на этот шаг как на функцию, она принимает пару путей и секцию и возвращает новую пару путей. Тип функции такой: (Path, Path) -> Section -> (Path, Path). Давайте её реализуем – похоже, она нам пригодится.

Подсказка: функция будет нам полезна, потому что её можно использовать в качестве бинарной функции в левой свёртке; тип любой такой функции должен быть a -> b -> a.

```
roadStep :: (Path, Path) -> Section -> (Path, Path)
roadStep (pathA, pathB) (Section a b c) =
   let timeA = sum $ map snd pathA
       timeB = sum $ map snd pathB
       forwardTimeToA = timeA + a
       crossTimeToA = timeB + b + c
       forwardTimeToB = timeB + b
       crossTimeToB = timeA + a + c
       newPathToA = if forwardTimeToA <= crossTimeToA
                       then (A,a):pathA
                       else (C,c):(B,b):pathB
       newPathToB = if forwardTimeToB <= crossTimeToB
                       then (B,b):pathB
                       else (C,c):(A,a):pathA
   in (newPathToA, newPathToB)
```

Как это работает? Для начала вычисляем оптимальное время по дороге A, основываясь на текущем лучшем маршруте; то же самое для B. Мы выполняем sum $ map snd pathA, так что если pathA – это что-то вроде [(A,100),(C,20)], timeA станет равным 120.

forwardTimeToA – это время, которое мы потратили бы, если бы ехали до следующего перекрёстка по A от предыдущего перекрёстка на A напрямую. Оно равно лучшему времени по дороге A плюс длительность по A текущей секции.

crossTimeToA – это время, которое мы потратили бы, если бы ехали до следующего перекрёстка на A по B, а затем повернули бы на A. Оно равно лучшему времени по B плюс длительность B в текущей секции плюс длительность секции C.

Таким же образом вычисляем forwardTimeToB и crossTimeToB.

Теперь, когда мы знаем лучший путь до A и B, нам нужно создать новые пути до A и B с учетом этой информации. Если выгоднее ехать до A просто напрямую, мы устанавливаем newPathToA равным (A,a):pathA. Подставляем метку A и длину секции a к началу текущего оп-

тимального пути. Мы полагаем, что лучший путь до следующего перекрёстка по `A` – это путь до предыдущего перекрёстка по `A` плюс ещё одна секция по `A`. Запомните, `A` – это просто метка, в то время как `a` имеет тип `Int`. Для чего мы подставляем их к началу, вместо того чтобы написать `pathA ++ [(A,a)]`? Добавление элемента к началу списка (также называемое *конструированием списка*) работает значительно быстрее, чем добавление к концу. Это означает, что получающийся путь будет накапливаться в обратном порядке, по мере выполнения свёртки с нашей функцией, но нам легче будет обратить список впоследствии, чем переделать формирование списка. Если выгоднее ехать до следующего перекрёстка по `A`, двигаясь по `B` и поворачивая на `A`, то `newPathToA` будет старым путём до `B` плюс секция до перекрёстка по `B` и переезд на `A`. Далее мы делаем то же самое для `newPathToB`, но в зеркальном отражении.

Рано или поздно мы получим пару из `newPathToA` и `newPathToB`.

Запустим функцию на первой секции `heathrowToLondon`. Поскольку эта секция первая, лучшие пути по `A` и `B` будут пустыми списками.

```
ghci> roadStep ([], []) (head heathrowToLondon)
([(C,30),(B,10)],[(B,10)])
```

Помните, что пути записаны в обратном порядке, так что читайте их справа налево. Из результата видно, что лучший путь до следующего перекрёстка по `A` – это начать движение по `B` и затем переехать на `A`; ну а лучший путь до следующего перекрёстка по `B` – ехать прямо по `B`.

> **ПРИМЕЧАНИЕ.** *Подсказка для оптимизации: когда мы выполняем* `timeA = sum $ map snd pathA`*, мы заново вычисляем время пути на каждом шаге. Нам не пришлось бы делать этого, если бы мы реализовали функцию* `roadStep` *так, чтобы она принимала и возвращала лучшее время по* `A` *и по* `B` *вместе с соответствующими путями.*

Теперь у нас есть функция, которая принимает пару путей и секцию, а также вычисляет новый оптимальный путь, так что мы легко

можем выполнить левую свёртку по списку секций. Функция roadStep вызывается со значением в качестве аккумулятора ([],[]) и первой секцией, а возвращает пару оптимальных путей до этой секции. Затем она вызывается с этой парой путей и следующей секцией и т. д. Когда мы прошли по всем секциям, у нас остаётся пара оптимальных путей; кратчайший из них и будет являться решением задачи. Принимая это во внимание, мы можем реализовать функцию optimalPath.

```
optimalPath :: RoadSystem -> Path
optimalPath roadSystem =
   let (bestAPath, bestBPath) = foldl roadStep ([],[]) roadSystem
   in if sum (map snd bestAPath) <= sum (map snd bestBPath)
         then reverse bestAPath
         else reverse bestBPath
```

Мы выполняем левую свёртку по roadSystem (это список секций), указывая в качестве начального значения аккумулятора пару пустых путей. Результат свёртки – пара путей, так что нам потребуется сопоставление с образцом, чтобы добраться до самих путей. Затем мы проверяем, который из двух путей короче, и возвращаем его. Прежде чем вернуть путь, мы его обращаем, так как мы накапливали оптимальный путь, добавляя элементы в начало.

Проведём тест:

```
ghci> optimalPath heathrowToLondon
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8),(C,0)]
```

Это практически тот результат, который мы ожидали получить. Чудесно. Он слегка отличается от ожидаемого, так как в конце пути есть шаг (C,0), который означает, что мы переехали на другую дорогу, как только попали в Лондон; но поскольку этот переезд ничего не стоит, результат остаётся верным.

Получение описания дорожной системы из внешнего источника

Итак, у нас есть функция, которая находит оптимальный путь по заданной системе дорог. Теперь нам надо считать текстовое представление дорожной системы со стандартного ввода, преобразовать его в тип RoadSystem, пропустить его через функцию optimalPath, после чего напечатать путь.

Для начала напишем функцию, которая принимает список и разбивает его на группы одинакового размера. Назовём её `groupsOf`. Если передать в качестве параметра `[1..10]`, то `groupsOf 3` должна вернуть `[[1,2,3],[4,5,6],[7,8,9],[10]]`.

```
groupsOf :: Int -> [a] -> [[a]]
groupsOf 0 _ = undefined
groupsOf _ [] = []
groupsOf n xs = take n xs : groupsOf n (drop n xs)
```

Обычная рекурсивная функция. Для `xs` равного `[1..10]` и `n = 3`, получаем `[1,2,3] :groupsOf 3 [4,5,6,7,8,9,10]`. После завершения рекурсии мы получаем наш список, сгруппированный по три элемента. Теперь напишем главную функцию, которая считывает данные со стандартного входа, создаёт `RoadSystem` из считанных данных и печатает кратчайший путь:

```
import Data.List

main = do
   contents <- getContents
   let threes = groupsOf 3 (map read $ lines contents)
       roadSystem = map (\[a,b,c] -> Section a b c) threes
       path = optimalPath roadSystem
       pathString = concat $ map (show . fst) path
       pathTime = sum $ map snd path
   putStrLn $ "Лучший путь: " ++ pathString
   putStrLn $ "Время: " ++ show pathTime
```

Вначале получаем данные со стандартного входа. Затем вызываем функцию `lines` с полученными данными, чтобы преобразовать строку вида `"50\n10\n30\n`... в список `["50","10","30"`..., и функцию `map read`, чтобы преобразовать строки из списка в числа. Вызываем функцию `groupsOf 3`, чтобы получить список списков длиной 3. Применяем анонимную функцию `(\[a,b,c] -> Section a b c)` к полученному списку списков. Как мы видим, данная анонимная функция принимает список из трёх элементов и превращает его в секцию. В итоге `roadSystem` содержит систему дорог и имеет правильный тип, а именно `RoadSystem` (или `[Section]`). Далее мы вызываем функцию `optimalPath`, получаем путь и общее время в удобной текстовой форме, и распечатываем их.

Сохраним следующий текст:

```
50
10
30
5
90
20
40
2
25
10
8
0
```

в файле *paths.txt* и затем «скормим» его нашей программе.

```
$ ./heathrow < paths.txt
Лучший путь: BCACBBC
Время: 75
```

Отлично работает!

Можете использовать модуль `Data.Random`, чтобы сгенерировать более длинные системы дорог и «скормить» их только что написанной программе. Если вы получите переполнение стека, попытайтесь использовать функцию `foldl'` вместо `foldl` и `foldl' (+) 0` вместо `sum`. Можно также скомпилировать программу следующим образом:

```
$ ghc -O heathrow.hs
```

Указание флага `O` включает оптимизацию, которая предотвращает переполнение стека в таких функциях, как `foldl` и `sum`.

11
АППЛИКАТИВНЫЕ ФУНКТОРЫ

Сочетание чистоты, функций высшего порядка, параметризованных алгебраических типов данных и классов типов в языке Haskell делает реализацию полиморфизма более простой, чем в других языках. Нам не нужно думать о типах, принадлежащих к большой иерархии. Вместо этого мы изучаем, как могут действовать типы, а затем связываем их с помощью подходящих классов типов. Тип `Int` может вести себя как множество сущностей – сравниваемая сущность, упорядочиваемая сущность, перечислимая сущность и т. д.

Классы типов открыты – это означает, что мы можем определить собственный тип данных, обдумать, как он может действовать, и связать его с классами типов, которые определяют его поведение. Также можно ввести новый класс типов, а затем сделать уже существующие типы его экземплярами. По этой причине и благодаря прекрасной системе типов языка Haskell, которая позволяет нам знать многое о функции только по её объявлению типа, мы можем определять классы типов, которые описывают очень общее, абстрактное поведение.

Мы говорили о классах типов, которые определяют операции для проверки двух элементов на равенство и для сравнения двух элементов по размещению их в каком-либо порядке. Это очень абстрактное и элегантное поведение, хотя мы не воспринимаем его как нечто особенное, поскольку нам доводилось наблюдать его большую часть нашей жизни. В главе 7 были введены функто-

ры – они являются типами, значения которых можно отобразить. Это пример полезного и всё ещё довольно абстрактного свойства, которое могут описать классы типов. В этой главе мы ближе познакомимся с функторами, а также с немного более сильными и более полезными их версиями, которые называются *аппликативными функторами*.

Функторы возвращаются

Как вы узнали из главы 7, функторы – это сущности, которые можно отобразить, как, например, списки, значения типа `Maybe` и деревья. В языке Haskell они описываются классом типов `Functor`, содержащим только один метод `fmap`. Функция `fmap` имеет тип `fmap :: (a -> b) -> f a -> f b`, который говорит: «Дайте мне функцию, которая принимает `a` и возвращает `b` и коробку, где содержится `a` (или несколько a), и я верну коробку с `b` (или несколькими `b`) внутри». Она применяет функцию к элементу внутри коробки.

Мы также можем воспринимать значения функторов как значения с добавочным контекстом. Например, значения типа `Maybe` обладают дополнительным контекстом того, что вычисления могли окончиться неуспешно. По отношению к спискам контекстом является то, что значение может быть множественным либо отсутствовать. Функция `fmap` применяет функцию к значению, сохраняя его контекст.

Если мы хотим сделать конструктор типа экземпляром класса `Functor`, он должен иметь сорт `* -> *`; это значит, что он принимает ровно один конкретный тип в качестве параметра типа. Например, конструктор `Maybe` может быть сделан экземпляром, так как он получает один параметр типа для произведения конкретного типа, как, например, `Maybe Int` или `Maybe String`. Если конструктор типа принимает два параметра, как, например, конструктор `Either`, мы должны частично применять конструктор типа до тех пор, пока он не будет принимать только один параметр. Поэтому мы не можем написать определение `Functor Either where`, зато можем написать определение `Functor (Either a) where`. Затем, если бы мы вообразили, что функция `fmap` предназначена только для работы со значениями типа `Either a`, она имела бы следующее описание типа:

```
fmap :: (b -> c) -> Either a b -> Either a c
```

Как видите, часть Either a – фиксированная, потому что частично применённый конструктор типа Either a принимает только один параметр типа.

Действия ввода-вывода в качестве функторов

К настоящему моменту вы изучили, каким образом многие типы (если быть точным, конструкторы типов) являются экземплярами класса Functor: [] и Maybe, Either a, равно как и тип Tree, который мы создали в главе 7. Вы видели, как можно отображать их с помощью функций на всеобщее благо. Теперь давайте взглянем на экземпляр типа IO.

Если какое-то значение обладает, скажем, типом IO String, это означает, что перед нами действие ввода-вывода, которое выйдет в реальный мир и получит для нас некую строку, которую затем вернёт в качестве результата. Мы можем использовать запись <- в синтаксисе do для привязывания этого результата к имени. В главе 8 мы говорили о том, что действия ввода-вывода похожи на ящики с маленькими ножками, которые выходят наружу и приносят нам какое-то значение из внешнего мира. Мы можем посмотреть, что они принесли, но после просмотра нам необходимо снова обернуть значение в тип IO. Рассматривая эту аналогию с ящиками на ножках, вы можете понять, каким образом тип IO действует как функтор.

Давайте посмотрим, как же это тип IO является экземпляром класса Functor… Когда мы используем функцию fmap для отображения действия ввода-вывода с помощью функции, мы хотим получить обратно действие ввода-вывода, которое делает то же самое, но к его результирующему значению применяется наша функция. Вот код:

```
instance Functor IO where
   fmap f action = do
      result <- action
      return (f result)
```

Результатом отображения действия ввода-вывода с помощью чего-либо будет действие ввода-вывода, так что мы сразу же используем синтаксис do для склеивания двух действий и создания одного нового. В реализации для метода fmap мы создаём новое действие ввода-вывода, которое сначала выполняет первоначальное действие ввода-вывода, давая результату имя result. Затем мы выполняем

return (f result). Вспомните, что return – это функция, создающая действие ввода-вывода, которое ничего не делает, а только возвращает что-либо в качестве своего результата.

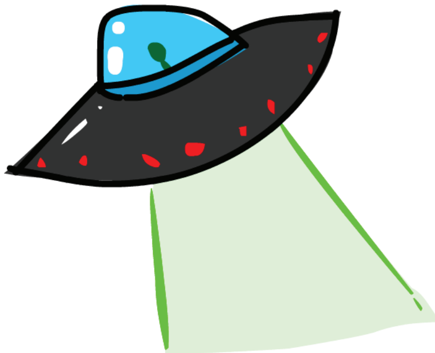
Действие, которое производит блок do, будет всегда возвращать результирующее значение своего последнего действия. Вот почему мы используем функцию return, чтобы создать действие ввода-вывода, которое в действительности ничего не делает, а просто возвращает применение f result в качестве результата нового действия ввода-вывода. Взгляните на этот кусок кода:

```
main = do
   line <- getLine
   let line' = reverse line
   putStrLn $ "Вы сказали " ++ line' ++ " наоборот!"
   putStrLn $ "Да, вы точно сказали " ++ line' ++ " наоборот!"
```

У пользователя запрашивается строка, и мы отдаём её обратно пользователю, но в перевёрнутом виде. А вот как можно переписать это с использованием функции fmap:

```
main = do
   line <- fmap reverse getLine
   putStrLn $ "Вы сказали " ++ line ++ " наоборот!"
   putStrLn $ "Да, вы точно сказали " ++ line ++ " наоборот!"
```

Так же как можно отобразить Just "уфф" с помощью отображения fmap reverse, получая Just "ффу", мы можем отобразить и функцию getLine с помощью отображения fmap reverse. Функция getLine – это действие ввода-вывода, которое имеет тип IO String, и отображение его с помощью функции reverse даёт нам действие ввода-вывода, которое выйдет в реальный мир и получит строку, а затем применит функцию reverse к своему результату. Таким же образом, как мы можем применить функцию к тому, что находится внутри коробки Maybe, можно применить функцию и к тому, что находится внутри коробки IO,



но она должна выйти в реальный мир, чтобы получить что-либо. Затем, когда мы привязываем результат к имени, используя запись `<-`, имя будет отражать результат, к которому уже применена функция `reverse`.

Действие ввода-вывода `fmap (++"!") getLine` ведёт себя в точности как функция `getLine`, за исключением того, что к её результату всегда добавляется строка `"!"` в конец!

Если бы функция `fmap` работала только с типом `IO`, она имела бы тип `fmap :: (a -> b) -> IO a -> IO b`. Функция `fmap` принимает функцию и действие ввода-вывода и возвращает новое действие ввода-вывода, похожее на старое, за исключением того, что к результату, содержащемуся в нём, применяется функция.

Предположим, вы связываете результат действия ввода-вывода с именем лишь для того, чтобы применить к нему функцию, а затем даёте очередному результату какое-то другое имя, – в таком случае подумайте над использованием функции `fmap`. Если вы хотите применить несколько функций к некоторым данным внутри функтора, то можете объявить свою функцию на верхнем уровне, создать анонимную функцию или, в идеале, использовать композицию функций:

```
import Data.Char
import Data.List

main = do
   line <- fmap (intersperse '-' . reverse . map toUpper) getLine
   putStrLn line
```

Вот что произойдёт, если мы сохраним этот код в файле *fmapping_io.hs*, скомпилируем, запустим и введём `"Эй, привет"`:

```
$ ./fmapping_io
Эй, привет
Т-Е-В-И-Р-П- -,-Й-Э
```

Выражение `intersperse '-' . reverse . map toUpper` берёт строку, отображает её с помощью функции `toUpper`, применяет функцию `reverse` к этому результату, а затем применяет к нему выражение `intersperse '-'`. Это более красивый способ записи следующего кода:

```
(\xs -> intersperse '-' (reverse (map toUpper xs)))
```

Функции в качестве функторов

Другим экземпляром класса Functor, с которым мы всё время имели дело, является (->) r. Стойте!.. Что, чёрт возьми, означает (->) r? Тип функции r -> a может быть переписан в виде (->) r a, так же как мы можем записать 2 + 3 в виде (+) 2 3. Когда мы воспринимаем его как (->) r a, то (->) представляется немного в другом свете. Это просто конструктор типа, который принимает два параметра типа, как это делает конструктор Either.

Но вспомните, что конструктор типа должен принимать в точности один параметр типа, чтобы его можно было сделать экземпляром класса Functor. Вот почему нельзя сделать конструктор (->) экземпляром класса Functor; однако, если частично применить его до (->) r, это не составит никаких проблем. Если бы синтаксис позволял частично применять конструкторы типов с помощью сечений – подобно тому как можно частично применить оператор +, выполнив (2+), что равнозначно (+) 2, – вы могли бы записать (->) r как (r ->).

Каким же образом функции выступают в качестве функторов? Давайте взглянем на реализацию, которая находится в модуле Control.Monad.Instances.

```
instance Functor ((->) r) where
   fmap f g = (\x -> f (g x))
```

Сначала подумаем над типом метода fmap:

```
fmap :: (a -> b) -> f a -> f b
```

Далее мысленно заменим каждое вхождение идентификатора f, являющегося ролью, которую играет наш экземпляр функтора, выражением (->) r. Это позволит нам понять, как функция fmap должна вести себя в отношении данного конкретного экземпляра. Вот результат:

```
fmap :: (a -> b) -> ((->) r a) -> ((->) r b)
```

Теперь можно записать типы (->) r a и (->) r b в инфиксном виде, то есть r -> a и r -> b, как мы обычно поступаем с функциями:

```
fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

Хорошо. Отображение одной функции с помощью другой должно произвести функцию, так же как отображение типа Maybe

с помощью функции должно произвести тип `Maybe`, а отображение списка с помощью функции – список. О чём говорит нам предыдущий тип? Мы видим, что он берёт функцию из `a` в `b` и функцию из `r` в `a` и возвращает функцию из `r` в `b`. Напоминает ли это вам что-нибудь? Да, композицию функций!.. Мы присоединяем выход `r -> a` ко входу `a -> b`, чтобы получить функцию `r -> b`, чем в точности и является композиция функций. Вот ещё один способ записи этого экземпляра:

```
instance Functor ((->) r) where
   fmap = (.)
```

Код наглядно показывает, что применение функции `fmap` к функциям – это просто композиция функций.

В исходном коде импортируйте модуль `Control.Monad.Instances`, поскольку это модуль, где определён данный экземпляр, а затем загрузите исходный код и попробуйте поиграть с отображением функций:

```
ghci> :t fmap (*3) (+100)
fmap (*3) (+100) :: (Num a) => a -> a
ghci> fmap (*3) (+100) 1
303
ghci> (*3) `fmap` (+100) $ 1
303
ghci> (*3) . (+100) $ 1
303
ghci> fmap (show . (*3)) (*100) 1
"300"
```

Мы можем вызывать `fmap` как инфиксную функцию, чтобы сходство с оператором `.` было явным. Во второй строке ввода мы отображаем `(+100)` с помощью `(*3)`, что даёт функцию, которая примет ввод, применит к нему `(+100)`, а затем применит к этому результату `(*3)`. Затем мы применяем эту функцию к значению `1`.

Как и все функторы, функции могут восприниматься как значения с контекстами. Когда у нас есть функция вроде `(+3)`, мы можем рассматривать значение как окончательный результат функции, а контекстом является то, что мы должны применить эту функцию к чему-либо, чтобы получить результат. Применение `fmap (*3)` к `(+100)` создаст ещё одну функцию, которая действует так же,

как (+100), но перед возвратом результата к этому результату будет применена функция (*3).

Тот факт, что функция fmap является композицией функций при применении к функциям, на данный момент не слишком нам полезен, но, по крайней мере, он вызывает интерес. Это несколько меняет наше сознание и позволяет нам увидеть, как сущности, которые действуют скорее как вычисления, чем как коробки (IO и (->) r), могут быть функторами. Отображение вычисления с помощью функции возвращает тот же самый тип вычисления, но результат этого вычисления изменён функцией.

Перед тем как перейти к законам, которым должна следовать fmap, давайте ещё раз задумаемся о типе fmap:

```
fmap :: (a -> b) -> f a -> f b
```

Если помните, введение в каррированные функции в главе 5 началось с утверждения, что все функции в языке Haskell на самом деле принимают один параметр. Функция a -> b -> c в действительности

берёт только один параметр типа a, после чего возвращает функцию b -> c, которая принимает один параметр типа b и возвращает значение типа c. Вот почему вызов функции с недостаточным количеством параметров (её частичное применение) возвращает нам обратно функцию, принимающую несколько параметров, которые мы пропустили (если мы опять воспринимаем функции так, как если бы они принимали несколько параметров). Поэтому a -> b -> c можно записать в виде a -> (b -> c), чтобы сделать каррирование более очевидным.

Аналогичным образом, записав fmap :: (a -> b) -> (f a -> f b), мы можем воспринимать fmap не как функцию, которая принимает одну функцию и значение функтора и возвращает значение функтора, но как функцию, которая принимает функцию и возвращает

новую функцию, которая такая же, как и прежняя, за исключением того, что она принимает значение функтора в качестве параметра и возвращает значение функтора в качестве результата. Она принимает функцию типа `a -> b` и возвращает функцию типа `f a -> f b`. Это называется *«втягивание функции»*. Давайте реализуем эту идею, используя команду `:t` в GHCi:

```
ghci> :t fmap (*2)
fmap (*2) :: (Num a, Functor f) => f a -> f a
ghci> :t fmap (replicate 3)
fmap (replicate 3) :: (Functor f) => f a -> f [a]
```

Выражение `fmap (*2)` – это функция, которая получает функтор `f` над числами и возвращает функтор над числами. Таким функтором могут быть список, значение `Maybe`, `Either String` или что-то другое. Выражение `fmap (replicate 3)` получит функтор над любым типом и вернёт функтор над списком элементов данного типа. Это становится ещё очевиднее, если мы частично применим, скажем, `fmap (++"!")`, а затем привяжем её к имени в GHCi.

Вы можете рассматривать `fmap` двояко:

- как функцию, которая принимает функцию и значение функтора, а затем отображает это значение функтора с помощью данной функции;
- как функцию, которая принимает функцию и втягивает её в функтор, так чтобы она оперировала значениями функторов.

Обе точки зрения верны.

Тип `fmap (replicate 3) :: (Functor f) => f a -> f [a]` означает, что функция будет работать с любым функтором. Что именно она будет делать, зависит от функтора. Если мы применим `fmap (replicate 3)` к списку, будет выбрана реализация `fmap` для списка, то есть просто `map`. Если мы применим её к `Maybe a`, она применит `replicate 3` к значению внутри `Just`. Если это значение равно `Nothing`, то оно останется равным `Nothing`. Вот несколько примеров:

```
ghci> fmap (replicate 3) [1,2,3,4]
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]
ghci> fmap (replicate 3) (Just 4)
Just [4,4,4]
ghci> fmap (replicate 3) (Right "ля")
```

```
Right ["ля","ля","ля"]
ghci> fmap (replicate 3) Nothing
Nothing
ghci> fmap (replicate 3) (Left "фуу")
Left "фуу"
```

Законы функторов

Предполагается, что все функторы проявляют определённые свойства и поведение. Они должны надёжно вести себя как сущности, которые можно отобразить. Применение функции `fmap` к функтору должно только отобразить функтор с помощью функции – ничего более. Это поведение описано в законах функторов. Все экземпляры класса `Functor` должны следовать этим двум законам. Язык Haskell не принуждает, чтобы эти законы выполнялись автоматически, поэтому вы должны проверять их сами, когда создаёте функтор. Все экземпляры класса `Functor` в стандартной библиотеке выполняют эти законы.

Закон 1

Первый закон функторов гласит, что если мы применяем функцию `id` к значению функтора, то значение функтора, которое мы получим, должно быть таким же, как первоначальное значение функтора. В формализованной записи это выглядит так: `fmap id = id`. Иными словами, если мы применим `fmap id` к значению функтора, это должно быть то же самое, что и просто применение функции `id` к значению. Вспомните, что `id` – это функция тождества, которая просто возвращает свой параметр неизменным. Она также может быть записана в виде `\x -> x`. Если воспринимать значение функтора как нечто, что может быть отображено, то закон `fmap id = id` представляется довольно очевидным.

Давайте посмотрим, выполняется ли он для некоторых значений функторов:

```
ghci> fmap id (Just 3)
Just 3
ghci> id (Just 3)
Just 3
ghci> fmap id [1..5]
[1,2,3,4,5]
ghci> id [1..5]
[1,2,3,4,5]
ghci> fmap id []
[]
ghci> fmap id Nothing
Nothing
```

Если посмотреть на реализацию функцию `fmap`, например, для типа `Maybe`, мы можем понять, почему выполняется первый закон функторов:

```
instance Functor Maybe where
   fmap f (Just x) = Just (f x)
   fmap f Nothing= Nothing
```

Мы представляем, что функция `id` играет роль параметра `f` в этой реализации. Нам видно, что если мы применяем `fmap id` к значению `Just x`, то результатом будет `Just (id x)`, и поскольку `id` просто возвращает свой параметр, мы можем сделать вывод, что `Just (id x)` равно `Just x`. Теперь нам известно, что если мы применим функцию `id` к значению типа `Maybe`, созданному с помощью конструктора данных `Just`, обратно мы получим то же самое значение.

Видно, что применение функции `id` к значению `Nothing` возвращает то же самое значение `Nothing`. Поэтому из этих двух равенств в реализации функции `fmap` нам видно, что закон `fmap id = id` соблюдается.

Закон 2

Второй закон гласит, что композиция двух функций и последующее применение результирующей функции к функтору должны давать тот же результат, что и применение первой функции к функтору, а затем применение другой. В формальной записи это выглядит так: `fmap (f . g) = fmap f . fmap g`. Или если записать по-другому, то для

любого значения функтора `x` должно выполняться следующее: `fmap (f . g) x = fmap f (fmap g x)`.

Если мы выявили, что некоторый тип подчиняется двум законам функторов, надо надеяться, что он обладает такими же фундаментальными поведениями, как и другие функторы, когда дело доходит до отображения. Мы можем быть уверены, что когда мы применяем к нему функцию `fmap`, за кулисами ничего не произойдёт, кроме отображения, и он будет действовать как сущность, которая может быть отображена – то есть функтор.

Можно выяснить, как второй закон выполняется по отношению к некоторому типу, посмотрев на реализацию функции `fmap` для этого типа, а затем использовав метод, который мы применяли, чтобы проверить, подчиняется ли тип `Maybe` первому закону. Итак, чтобы проверить, как второй закон функторов выполняется для типа `Maybe`, если мы применим выражение `fmap (f . g)` к значению `Nothing`, мы получаем то же самое значение `Nothing`, потому что применение любой функции к `Nothing` даёт `Nothing`. Если мы выполним выражение `fmap f (fmap g Nothing)`, то получим результат `Nothing` по тем же причинам.

Довольно просто увидеть, как второй закон выполняется для типа `Maybe`, когда значение равно `Nothing`. Но что если это значение `Just`? Ладно – если мы выполним `fmap (f . g) (Just x)`, из реализации нам будет видно, что это реализовано как `Just ((f . g) x)`; аналогичной записью было бы `Just (f (g x))`. Если же мы выполним `fmap f (fmap g (Just x))`, то из реализации увидим, что `fmap g (Just x)` – это `Just (g x)`. Следовательно, `fmap f (fmap g (Just x))` равно `fmap f (Just (g x))`, а из реализации нам видно, что это равнозначно `Just (f (g x))`.

Если вы немного смущены этим доказательством, не волнуйтесь. Убедитесь, что вы понимаете, как устроена композиция функций. Часто вы можете интуитивно понимать, как выполняются эти законы, поскольку типы действуют как контейнеры или функции. Вы также можете просто проверить их на нескольких разных значениях типа – и сумеете с определённой долей уверенности сказать, что тип действительно подчиняется этим законам.

Нарушение закона

Давайте посмотрим на «патологический» пример конструктора типов, который является экземпляром класса типов `Functor`, но не

является функтором, потому что он не выполняет законы. Скажем, у нас есть следующий тип:

```
data CMaybe a = CNothing | CJust Int a deriving (Show)
```

Буква C здесь обозначает счётчик. Это тип данных, который во многом похож на тип Maybe a, только часть Just содержит два поля вместо одного. Первое поле в конструкторе данных CJust всегда имеет тип Int; оно будет своего рода счётчиком. Второе поле имеет тип a, который берётся из параметра типа, и его тип будет зависеть от конкретного типа, который мы выберем для CMaybe a. Давайте поэкспериментируем с нашим новым типом:

```
ghci> CNothing
CNothing
ghci> CJust 0 "ха-ха"
CJust 0 "ха-ха"
ghci> :t CNothing
CNothing :: CMaybe a
ghci> :t CJust 0 "ха-ха"
CJust 0 "ха-ха" :: CMaybe [Char]
ghci> CJust 100 [1,2,3]
CJust 100 [1,2,3]
```

Если мы используем конструктор данных CNothing, в нём нет полей. Если мы используем конструктор данных CJust, первое поле является целым числом, а второе может быть любого типа. Давайте сделаем этот тип экземпляром класса Functor, так чтобы каждый раз, когда мы используем функцию fmap, функция применялась ко второму полю, а первое поле увеличивалось на 1:

```
instance Functor CMaybe where
   fmap f CNothing= CNothing
   fmap f (CJust counter x) = CJust (counter+1) (f x)
```

Это отчасти похоже на реализацию экземпляра для типа Maybe, только когда функция fmap применяется к значению, которое не представляет пустую коробку (значение CJust), мы не просто применяем функцию к содержимому, но и увеличиваем счётчик на 1. Пока вроде бы всё круто! Мы даже можем немного поиграть с этим:

```
ghci> fmap (++"-ха") (CJust 0 "хо")
CJust 1 "хо-ха"
```

```
ghci> fmap (++"-хе") (fmap (++"-ха") (CJust 0 "хо"))
CJust 2 "хо-ха-хе"
ghci> fmap (++"ля") CNothing
CNothing
```

Подчиняется ли этот тип законам функторов? Для того чтобы увидеть, что что-то не подчиняется закону, достаточно найти всего одно исключение.

```
ghci> fmap id (CJust 0 "ха-ха")
CJust 1 "ха-ха"
ghci> id (CJust 0 "ха-ха")
CJust 0 "ха-ха"
```

Как гласит первый закон функторов, если мы отобразим значение функтора с помощью функции `id`, это должно быть то же самое, что и просто вызов функции `id` с тем же значением функтора. Наш пример показывает, что это не относится к нашему функтору `CMaybe`. Хотя он и имеет экземпляр класса `Functor`, он не подчиняется данному закону функторов и, следовательно, не является функтором.

Поскольку тип `CMaybe` не является функтором, хотя он и притворяется таковым, использование его в качестве функтора может привести к неисправному коду. Когда мы используем функтор, не должно иметь значения, производим ли мы сначала композицию нескольких функций, а затем с её помощью отображаем значение функтора, или же просто отображаем значение функтора последовательно с помощью каждой функции. Но при использовании типа `CMaybe` это имеет значение, так как он следит, сколько раз его отобразили. Проблема!.. Если мы хотим, чтобы тип `CMaybe` подчинялся законам функторов, мы должны сделать так, чтобы поле типа `Int` не изменялось, когда используется функция `fmap`.

Вначале законы функторов могут показаться немного запутанными и ненужными. Но если мы знаем, что тип подчиняется обоим законам, мы можем строить определённые предположения о том, как он будет действовать. Если тип подчиняется законам функторов, мы знаем, что вызов функции `fmap` со значением этого типа только применит к нему функцию – ничего более. В результате наш код становится более абстрактным и расширяемым, потому что мы можем использовать законы, чтобы судить о поведении, кото-

рым должен обладать любой функтор, а также создавать функции, надёжно работающие с любым функтором.

В следующий раз, когда вы будете делать тип экземпляром класса `Functor`, найдите минутку, чтобы убедиться, что он удовлетворяет законам функторов. Вы всегда можете пройти по реализации строка за строкой и посмотреть, выполняются ли законы, либо попробовать найти исключение. Изучив функторы в достаточном количестве, вы станете узнавать общие для них свойства и поведение и интуитивно понимать, следует ли тот или иной тип законам функторов.

Использование аппликативных функторов

В этом разделе мы рассмотрим аппликативные функторы, которые являются расширенными функторами.

До настоящего времени мы были сосредоточены на отображении функторов с помощью функций, принимающих только один параметр. Но что происходит, когда мы отображаем функтор с помощью функции, которая принимает два параметра? Давайте рассмотрим пару конкретных примеров.

Если у нас есть `Just 3`, и мы выполняем выражение `fmap (*) (Just 3)`, что мы получим? Из реализации экземпляра типа `Maybe` для класса `Functor` мы знаем, что если это значение `Just`, то функция будет применена к значению внутри `Just`. Следовательно, выполнение выражения `fmap (*) (Just 3)` вернёт `Just ((*) 3)`, что может быть также записано в виде `Just (3 *)`, если мы используем сечения. Интересно! Мы получаем функцию, обёрнутую в конструктор `Just`!

Вот ещё несколько функций внутри значений функторов:

```
ghci> :t fmap (++) (Just "эй")
fmap (++) (Just "эй") :: Maybe ([Char] -> [Char])
```

```
ghci> :t fmap compare (Just 'a')
fmap compare (Just 'a') :: Maybe (Char -> Ordering)
ghci> :t fmap compare "A LIST OF CHARS"
fmap compare "A LIST OF CHARS" :: [Char -> Ordering]
ghci> :t fmap (\x y z -> x + y / z) [3,4,5,6]
fmap (\x y z -> x + y / z) [3,4,5,6] :: (Fractional a) => [a -> a -> a]
```

Если мы отображаем список символов с помощью функции `compare`, которая имеет тип `(Ord a) => a -> a -> Ordering`, то получаем список функций типа `Char -> Ordering`, потому что функция `compare` частично применяется с помощью символов в списке. Это не список функций типа `(Ord a) => a -> Ordering`, так как первый идентификатор переменной типа `a` имел тип `Char`, а потому и второе вхождение `a` обязано принять то же самое значение – тип `Char`.

Мы видим, как, отображая значения функторов с помощью «многопараметрических» функций, мы получаем значения функторов, которые содержат внутри себя функции. А что теперь с ними делать?.. Мы можем, например, отображать их с помощью функций, которые принимают эти функции в качестве параметров – поскольку, что бы ни находилось в значении функтора, оно будет передано функции, с помощью которой мы его отображаем, в качестве параметра.

```
ghci> let a = fmap (*) [1,2,3,4]
ghci> :t a
a :: [Integer -> Integer]
ghci> fmap (\f -> f 9) a
[9,18,27,36]
```

Но что если у нас есть значение функтора `Just (3 *)` и значение функтора `Just 5`, и мы хотим извлечь функцию из `Just (3 *)` и отобразить с её помощью `Just 5`? С обычными функторами у нас этого не получится, потому что они поддерживают только отображение имеющихся функторов с помощью обычных функций. Даже когда мы отображали функтор, содержащий функции, с помощью анонимной функции `\f -> f 9`, мы делали именно это и только это. Но используя то, что предлагает нам функция `fmap`, мы не можем с помощью функции, которая находится внутри значения функтора, отобразить другое значение функтора. Мы могли бы произвести сопоставление конструктора `Just` по образцу для извлечения из него

функции, а затем отобразить с её помощью Just 5, но мы ищем более общий и абстрактный подход, работающий с функторами.

Поприветствуйте аппликативные функторы

Итак, встречайте класс типов Applicative, находящийся в модуле Control.Applicative!.. Он определяет две функции: pure и <*>. Он не предоставляет реализации по умолчанию для какой-либо из этих функций, поэтому нам придётся определить их обе, если мы хотим, чтобы что-либо стало аппликативным функтором. Этот класс определён вот так:

```
class (Functor f) => Applicative f where
   pure :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

Простое определение класса из трёх строк говорит нам о многом!.. Первая строка начинается с определения класса Applicative; также она вводит ограничение класса. Ограничение говорит, что если мы хотим определить для типа экземпляр класса Applicative, он, прежде всего, уже должен иметь экземпляр класса Functor. Вот почему, когда нам известно, что конструктор типа принадлежит классу Applicative, можно смело утверждать, что он также принадлежит классу Functor, так что мы можем применять к нему функцию fmap.

Первый метод, который он определяет, называется pure. Его сигнатура выглядит так: pure :: a -> f a. Идентификатор f играет здесь роль нашего экземпляра аппликативного функтора. Поскольку язык Haskell обладает очень хорошей системой типов и притом всё, что может делать функция, – это получать некоторые параметры и возвращать некоторое значение, мы можем многое сказать по объявлению типа, и данный тип – не исключение.

Функция pure должна принимать значение любого типа и возвращать аппликативное значение с этим значением внутри него. Словосочетание «внутри него» опять вызывает в памяти нашу аналогию с коробкой, хотя мы и видели, что она не всегда выдерживает проверку. Но тип a -> f a всё равно довольно нагляден. Мы берём значение и оборачиваем его в аппликативное значение, которое содержит в себе это значение в качестве результата. Лучший способ представить себе функцию pure – это сказать, что она берёт

значение и помещает его в некий контекст по умолчанию (или чистый контекст) – минимальный контекст, который по-прежнему возвращает это значение.

Оператор <*> действительно интересен. У него вот такое определение типа:

```
f (a -> b) -> f a -> f b
```

Напоминает ли оно вам что-нибудь? Оно похоже на сигнатуру `fmap :: (a -> b) -> f a -> f b`. Вы можете воспринимать оператор <*> как разновидность расширенной функции `fmap`. Тогда как функция `fmap` принимает функцию и значение функтора и применяет функцию внутри значения функтора, оператор <*> принимает значение функтора, который содержит в себе функцию, и другой функтор – и извлекает эту функцию из первого функтора, затем отображая с её помощью второй.

Аппликативный функтор Maybe

Давайте взглянем на реализацию экземпляра класса `Applicative` для типа `Maybe`:

```
instance Applicative Maybe where
   pure = Just
   Nothing <*> _ = Nothing
   (Just f) <*> something = fmap f something
```

Опять же из определения класса мы видим, что идентификатор `f`, который играет роль аппликативного функтора, должен принимать один конкретный тип в качестве параметра. Поэтому мы пишем `instance Applicative Maybe where` вместо `instance Applicative (Maybe a) where`.

Далее, у нас есть функция `pure`. Вспомните, что функция должна что-то принять и обернуть в аппликативное значение. Мы написали `pure = Just`, потому что конструкторы данных вроде `Just` являются обычными функциями. Также можно было бы написать `pure x = Just x`.

Наконец, у нас есть определение оператора <*>. Извлечь функцию из значения `Nothing` нельзя, поскольку внутри него нет функции. Поэтому мы говорим, что если мы пробуем извлечь функцию из значения `Nothing`, результатом будет то же самое значение `Nothing`.

В определении класса Applicative есть ограничение класса Functor – значит, мы можем считать, что оба параметра оператора <*> являются значениями функтора. Если первым аргументом выступает не значение Nothing, а Just с некоторой функцией внутри, то мы говорим, что с помощью данной функции хотим отобразить второй параметр. Этот код также заботится о случае, когда вторым аргументом является значение Nothing, потому что его отображение с помощью любой функции при использовании метода fmap вернёт всё то же Nothing. Итак, в случае с типом Maybe оператор <*> извлекает функцию из значения слева, если это Just, и отображает с её помощью значение справа. Если какой-либо из параметров является значением Nothing, то и результатом будет Nothing.

Теперь давайте это опробуем:

```
ghci> Just (+3) <*> Just 9
Just 12
ghci> pure (+3) <*> Just 10
Just 13
ghci> pure (+3) <*> Just 9
Just 12
ghci> Just (++"ха-ха") <*> Nothing
Nothing
ghci> Nothing <*> Just "во-от"
Nothing
```

Вы видите, что выполнение выражений pure (+3) и Just (+3) в данном случае – одно и то же. Используйте функцию pure, если имеете дело со значениями типа Maybe в аппликативном контексте (если вы используете их с оператором <*>); в противном случае предпочитайте конструктор Just.

Первые четыре введённых строки демонстрируют, как функция извлекается, а затем используется для отображения; но в данном случае этого можно было добиться, просто применив не обёрнутые функции к функторам. Последняя строка любопытна тем, что мы пытаемся извлечь функцию из значения Nothing, а затем отображаем с её помощью нечто, что в результате даёт Nothing.

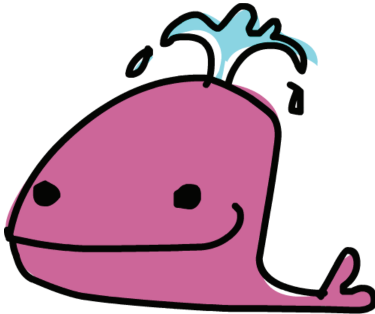
Когда вы отображаете функтор с помощью функции при использовании обычных функторов, вы не можете извлечь результат каким-либо общим способом, даже если результатом является частично применённая функция. Аппликативные функторы, с другой

стороны, позволяют вам работать с несколькими функторами, используя одну функцию.

Аппликативный стиль

При использовании класса типов `Applicative` мы можем последовательно задействовать несколько операторов `<*>` в виде цепочки вызовов, что позволяет легко работать сразу с несколькими аппликативными значениями, а не только с одним. Взгляните, например, на это:

```
ghci> pure (+) <*> Just 3 <*> Just 5
Just 8
ghci> pure (+) <*> Just 3 <*> Nothing
Nothing
ghci> pure (+) <*> Nothing <*> Just 5
Nothing
```



Мы обернули оператор + в аппликативное значение, а затем использовали оператор `<*>`, чтобы вызвать его с двумя параметрами, оба из которых являются аппликативными значениями.

Давайте посмотрим, как это происходит, шаг за шагом. Оператор `<*>` левоассоциативен; это значит, что

```
pure (+) <*> Just 3 <*> Just 5
```

то же самое, что и вот это:

```
(pure (+) <*> Just 3) <*> Just 5
```

Сначала оператор + помещается в аппликативное значение – в данном случае значение типа `Maybe`, которое содержит функцию. Итак, у нас есть `pure (+)`, что, по сути, равно `Just (+)`. Далее происходит вызов `Just (+) <*> Just 3`. Его результатом является `Just (3+)`. Это из-за частичного применения. Применение только значения `3` к оператору + возвращает в результате функцию, которая принимает один параметр и добавляет к нему `3`. Наконец, выполняется `Just (3+) <*> Just 5`, что в результате возвращает `Just 8`.

Ну разве не здорово?! Аппликативные функторы и аппликативный стиль вычисления `pure f <*> x <*> y <*>` ... позволяют взять функцию, которая ожидает параметры, не являющиеся аппликативными значениями, и использовать эту функцию для работы с несколькими аппликативными значениями. Функция может принимать столько параметров, сколько мы захотим, потому что она всегда частично применяется шаг за шагом между вхождениями оператора `<*>`.

Это становится ещё более удобным и очевидным, если мы примем во внимание тот факт, что выражение `pure f <*> x` равно `fmap f x`. Это один из законов аппликативных функторов, которые мы более подробно рассмотрим чуть позже; но давайте подумаем, как он применяется здесь. Функция `pure` помещает значение в контекст по умолчанию. Если мы просто поместим функцию в контекст по умолчанию, а затем извлечем её и применим к значению внутри другого аппликативного функтора, это будет то же самое, что просто отобразить этот аппликативный функтор с помощью данной функции. Вместо записи `pure f <*> x <*> y <*>` ..., мы можем написать `fmap f x <*> y <*>` ... Вот почему модуль `Control.Applicative` экспортирует оператор, названный `<$>`, который является просто синонимом функции `fmap` в виде инфиксного оператора. Вот как он определён:

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

ПРИМЕЧАНИЕ. *Вспомните, что переменные типов не зависят от имён параметров или имён других значений. Здесь идентификатор `f` в сигнатуре функции является переменной типа с ограничением класса, которое говорит, что любой конструктор типа, который заменяет `f`, должен иметь экземпляр класса `Functor`. Идентификатор `f` в теле функции обозначает функцию, с помощью которой мы отображаем значение `x`. Тот факт, что мы использовали `f` для представления обеих вещей, не означает, что они представляют одну и ту же вещь.*

При использовании оператора `<$>` аппликативный стиль проявляет себя во всей красе, потому что теперь, если мы хотим применить функцию `f` к трем аппликативным значениям, можно просто написать `f <$> x <*> y <*> z`. Если бы параметры были обычными значениями, мы бы написали `f x y z`.

Давайте подробнее рассмотрим, как это работает. Предположим, что мы хотим соединить значения Just "johntra" и Just "volta" в одну строку, находящуюся внутри функтора Maybe. Сделать это вполне в наших силах!

```
ghci> (++) <$> Just "johntra" <*> Just "volta"
Just "johntravolta"
```

Прежде чем мы увидим, что происходит, сравните предыдущую строку со следующей:

```
ghci> (++) "johntra" "volta"
"johntravolta"
```

Чтобы использовать обычную функцию с аппликативным функтором, просто разбросайте вокруг несколько <$> и <*>, и функция будет работать с аппликативными значениями и возвращать аппликативное значение. Ну не здорово ли?

Возвратимся к нашему выражению (++) <$> Just "джонтра" <*> Just "волта": сначала оператор (++), который имеет тип (++) :: [a] -> [a] -> [a], отображает значение Just "джонтра". Это даёт в результате такое же значение, как Just ("джонтра"++), имеющее тип Maybe ([Char] -> [Char]). Заметьте, как первый параметр оператора (++) был «съеден» и идентификатор a превратился в тип [Char]! А теперь выполняется выражение Just ("джонтра"++) <*> Just "волта", которое извлекает функцию из Just и отображает с её помощью значение Just "волта", что в результате даёт новое значение – Just "джонтраволта". Если бы одним из двух значений было значение Nothing, результатом также было бы Nothing.

Списки

Списки (на самом деле конструктор типа списка, []) являются аппликативными функторами. Вот так сюрприз! Вот как [] является экземпляром класса Applicative:

```
instance Applicative [] where
   pure x = [x]
   fs <*> xs = [f x | f <- fs, x <- xs]
```

Вспомните, что функция pure принимает значение и помещает его в контекст по умолчанию. Другими словами, она помещает его

в минимальный контекст, который всё ещё возвращает это значение. Минимальным контекстом для списков был бы пустой список, но пустой список означает отсутствие значения, поэтому он не может содержать в себе значение, к которому мы применили функцию `pure`. Вот почему эта функция принимает значение и помещает его в одноэлементный список. Подобным образом минимальным контекстом для аппликативного функтора `Maybe` было бы значение `Nothing` – но оно означает отсутствие значения вместо самого значения, поэтому функция `pure` в реализации экземпляра для типа `Maybe` реализована как вызов конструктора данных `Just`.

Вот функция `pure` в действии:

```
ghci> pure "Эй" :: [String]
["Эй"]
ghci> pure "Эй" :: Maybe String
Just "Эй"
```

Что насчёт оператора `<*>`? Если бы тип оператора `<*>` ограничивался только списками, мы получили бы `(<*>) :: [a -> b] -> [a] -> [b]`. Этот оператор реализован через генератор списков. Он должен каким-то образом извлечь функцию из своего левого параметра, а затем с её помощью отобразить правый. Но левый список может не содержать в себе функций или содержать одну либо несколько функций, а правый список также может содержать несколько значений. Вот почему мы используем генератор списков для извлечения из обоих списков. Мы применяем каждую возможную функцию из левого списка к каждому возможному значению из правого. Результирующий список содержит все возможные комбинации применения функции из левого списка к значению из правого.

Мы можем использовать оператор `<*>` со списками вот так:

```
ghci> [(*0),(+100),(  2)] <*> [1,2,3]
[0,0,0,101,102,103,1,4,9]
```

Левый список содержит три функции, а правый – три значения, поэтому в результирующем списке будет девять элементов. Каждая функция из левого списка применяется к каждому элементу из правого. Если у нас имеется список функций, принимающих два параметра, то мы можем применить эти функции между двумя списками.

В следующем примере применяются две функции между двумя списками:

```
ghci> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
```

Оператор `<*>` левоассоциативен, поэтому сначала выполняется `[(+),(*)] <*> [1,2]`, результатом чего является такой же список, как `[(1+),(2+),(1*),(2*)]`, потому что каждая функция слева применяется к каждому значению справа. Затем выполняется `[(1+),(2+),(1*),(2*)] <*> [3,4]`, что возвращает окончательный результат.

Как здорово использовать аппликативный стиль со списками!

```
ghci> (++) <$> ["ха","хех","хм"] <*> ["?","!","."]
["ха?","ха!","ха.","хех?","хех!","хех.","хм?","хм!","хм."]
```

Ещё раз: мы использовали обычную функцию, принимающую две строки, между двумя списками строк, просто вставляя соответствующие аппликативные операторы.

Вы можете воспринимать списки как недетерминированные вычисления. Значение вроде `100` или `"что"` можно рассматривать как детерминированное вычисление, которое имеет только один результат. В то же время список вроде `[1,2,3]` можно рассматривать как вычисление, которое не в состоянии определиться, какой результат оно желает иметь, поэтому возвращает нам все возможные результаты. Поэтому когда вы пишете что-то наподобие `(+) <$> [1,2,3] <*> [4,5,6]`, то можете рассматривать это как объединение двух недетерминированных вычислений с помощью оператора `+` только для того, чтобы создать ещё одно недетерминированное вычисление, которое ещё меньше уверено в своём результате.

Использование аппликативного стиля со списками часто является хорошей заменой генераторам списков. В главе 1 мы хотели вывести все возможные комбинации произведений `[2,5,10]` и `[8,10,11]` и с этой целью предприняли следующее:

```
ghci> [x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

Мы просто извлекаем значения из обоих списков и применяем функцию между каждой комбинацией элементов. То же самое можно сделать и в аппликативном стиле:

```
ghci> (*) <$> [2,5,10] <*> [8,10,11]
[16,20,22,40,50,55,80,100,110]
```

Для меня такой подход более понятен, поскольку проще понять, что мы просто вызываем оператор * между двумя недетерминированными вычислениями. Если бы мы захотели получить все возможные произведения элементов, больших 50, мы бы использовали следующее:

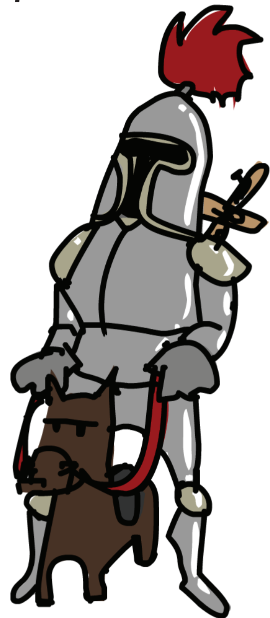
```
ghci> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

Легко увидеть, что вызов выражения `pure f <*> xs` при использовании списков эквивалентен выражению `fmap f xs`. Результат вычисления `pure f` – это просто `[f]`, а выражение `[f] <*> xs` применит каждую функцию в левом списке к каждому значению в правом; но в левом списке только одна функция, и, следовательно, это похоже на отображение.

Тип IO – тоже аппликативный функтор

Другой экземпляр класса `Applicative`, с которым мы уже встречались, – экземпляр для типа `IO`. Вот как он реализован:

```
instance Applicative IO where
   pure = return
   a <*> b = do
      f <- a
      x <- b
      return (f x)
```



Поскольку суть функции `pure` состоит в помещении значения в минимальный контекст, который всё ещё содержит значение как результат, логично, что в случае с типом `IO` функция `pure` – это просто вызов `return`. Функция `return` создаёт действие ввода-вывода, которое ничего не делает. Оно просто возвращает некое значение в качестве своего результата, не производя

никаких операций ввода-вывода вроде печати на терминал или чтения из файла.

Если бы оператор `<*>` ограничивался работой с типом `IO`, он бы имел тип `(<*>) :: IO (a -> b) -> IO a -> IO b`. В случае с типом `IO` он принимает действие ввода-вывода `a`, которое возвращает функцию, выполняет действие ввода-вывода и связывает эту функцию с идентификатором `f`. Затем он выполняет действие ввода-вывода `b` и связывает его результат с идентификатором `x`. Наконец, он применяет функцию `f` к значению `x` и возвращает результат этого применения в качестве результата. Чтобы это реализовать, мы использовали здесь синтаксис `do`. (Вспомните, что суть синтаксиса `do` заключается в том, чтобы взять несколько действий ввода-вывода и «склеить» их в одно.)

При использовании типов `Maybe` и `[]` мы могли бы воспринимать применение функции `<*>` просто как извлечение функции из её левого параметра, а затем применение её к правому параметру. В отношении типа `IO` извлечение остаётся в силе, но теперь у нас появляется понятие помещения в последовательность, поскольку мы берём два действия ввода-вывода и «склеиваем» их в одно. Мы должны извлечь функцию из первого действия ввода-вывода, но для того, чтобы можно было извлечь результат из действия ввода-вывода, последнее должно быть выполнено. Рассмотрите вот это:

```
myAction :: IO String
myAction = do
   a <- getLine
   b <- getLine
   return $ a ++ b
```

Это действие ввода-вывода, которое запросит у пользователя две строки и вернёт в качестве своего результата их конкатенацию. Мы достигли этого благодаря «склеиванию» двух действий ввода-вывода `getLine` и `return`, поскольку мы хотели, чтобы наше новое «склеенное» действие ввода-вывода содержало результат выполнения `a ++ b`. Ещё один способ записать это состоит в использовании аппликативного стиля:

```
myAction :: IO String
myAction = (++) <$> getLine <*> getLine
```

Это то же, что мы делали ранее, когда создавали действие ввода-вывода, которое применяло функцию между результатами двух

других действий ввода-вывода. Вспомните, что функция `getLine` – это действие ввода-вывода, которое имеет тип `getLine :: IO String`. Когда мы применяем оператор `<*>` между двумя аппликативными значениями, результатом является аппликативное значение, так что всё это имеет смысл.

Если мы вернёмся к аналогии с коробками, то можем представить себе функцию `getLine` как коробку, которая выйдет в реальный мир и принесёт нам строку. Выполнение выражения `(++) <$> getLine <*> getLine` создаёт другую, бóльшую коробку, которая посылает эти две коробки наружу для получения строк с терминала, а потом возвращает конкатенацию этих двух строк в качестве своего результата.

Выражение `(++) <$> getLine <*> getLine` имеет тип `IO String`. Это означает, что данное выражение является совершенно обычным действием ввода-вывода, как и любое другое, тоже возвращая результирующее значение, подобно другим действиям ввода-вывода. Вот почему мы можем выполнять следующие вещи:

```
main = do
   a <- (++) <$> getLine <*> getLine
   putStrLn $ "Две строки, соединённые вместе: " ++ a
```

Функции в качестве аппликативных функторов

Ещё одним экземпляром класса `Applicative` является тип `(->) r`, или функции. Мы нечасто используем функции в аппликативном стиле, но концепция, тем не менее, действительно интересна, поэтому давайте взглянем, как реализован экземпляр функции[1].

```
instance Applicative ((->) r) where
   pure x = (\_ -> x)
   f <*> g = \x -> f x (g x)
```

Когда мы оборачиваем значение в аппликативное значение с помощью функции `pure`, результат, который оно возвращает, должен быть этим значением. Минимальный контекст по умолчанию по-прежнему возвращает это значение в качестве результата. Вот почему в реализации экземпляра функция `pure` принимает значение

[1] Читателей, знакомых с комбинаторной логикой, такое определение экземпляра класса `Applicative` для функционального типа смутить не должно – методы определяют комбинаторы **K** и **S** соответственно. – *Прим. ред.*

и создаёт функцию, которая игнорирует передаваемый ей параметр и всегда возвращает это значение. Тип функции `pure` для экземпляра типа `(->) r` выглядит как `pure :: a -> (r -> a)`.

```
ghci> (pure 3) "ля"
3
```

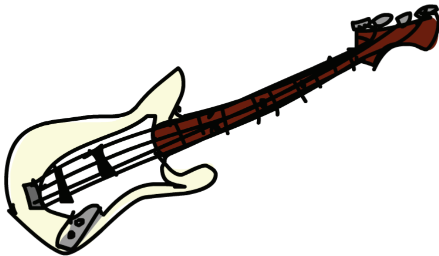
Из-за каррирования применение функции левоассоциативно, так что мы можем опустить скобки:

```
ghci> pure 3 "ля"
3
```

Реализация экземпляра `<*>` немного загадочна, поэтому давайте посмотрим, как использовать функции в качестве аппликативных функторов в аппликативном стиле:

```
ghci> :t (+) <$> (+3) <*> (*100)
(+) <$> (+3) <*> (*100) :: (Num a) => a -> a
ghci> (+) <$> (+3) <*> (*100) $ 5
508
```

Вызов оператора `<*>` с двумя аппликативными значениями возвращает аппликативное значение, поэтому если мы вызываем его с двумя функциями, то получаем функцию. Что же здесь происходит? Когда мы выполняем `(+) <$> (+3) <*> (*100)`, мы создаём функцию, которая применит оператор `+` к результатам выполнения функций `(+3)` и `(*100)` и вернёт это значение. При вызове выражения `(+) <$> (+3) <*> (*100) $ 5` функции `(+3)` и `(*100)` сначала применяются к значению 5, что в результате даёт 8 и 500; затем оператор `+` вызывается со значениями 8 и 500, что в результате даёт 508.



Следующий код аналогичен:

```
ghci> (\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
[8.0,10.0,2.5]
```

Мы создаём функцию, которая вызовет функцию `\x y z -> [x, y, z]` с окончательными результатами выполнения, возвращёнными функциями `(+3)`, `(*2)` и `(/2)`. Значение `5` передаётся каждой из трёх функций, а затем с этими результатами вызывается анонимная функция `\x y z -> [x, y, z]`.

ПРИМЕЧАНИЕ. *Не так уж важно, поняли ли вы, как работает экземпляр типа `(->) r` для класса `Applicative`, так что не отчаивайтесь, если вам это пока не ясно. Поработайте с аппликативным стилем и функциями, чтобы получить некоторое представление о том, как использовать функции в качестве аппликативных функторов.*

Застёгиваемые списки

Оказывается, есть и другие способы для списков быть аппликативными функторами. Один способ мы уже рассмотрели: вызов оператора `<*>` со списком функций и списком значений, который возвращает список всех возможных комбинаций применения функций из левого списка к значениям в списке справа.

Например, если мы выполним `[(+3),(*2)] <*> [1,2]`, то функция `(+3)` будет применена и к `1`, и к `2`; функция `(*2)` также будет применена и к `1`, и к `2`, а результатом станет список из четырёх элементов: `[4,5,2,4]`. Однако `[(+3),(*2)] <*> [1,2]` могла бы работать и таким образом, чтобы первая функция в списке слева была применена к первому значению в списке справа, вторая была бы применена ко второму значению и т. д. Это вернуло бы список с двумя значениями: `[4,4]`. Вы могли бы представить его как `[1 + 3, 2 * 2]`.

Экземпляром класса `Applicative`, с которым мы ещё не встречались, является тип `ZipList`, и находится он в модуле `Control.Applicative`.

Поскольку один тип не может иметь два экземпляра для одного и того же класса типов, был введён тип `ZipList a`, в котором имеется один конструктор (`ZipList`) с единственным полем (список). Вот так определяется его экземпляр:

```
instance Applicative ZipList where
   pure x = ZipList (repeat x)
   ZipList fs <*> ZipList xs = ZipList (zipWith (\f x -> f x) fs xs)
```

Оператор <*> применяет первую функцию к первому значению, вторую функцию – ко второму значению, и т. д. Это делается с помощью выражения zipWith (\f x -> f x) fs xs. Ввиду особенностей работы функции zipWith окончательный список будет той же длины, что и более короткий список из двух.

Функция pure здесь также интересна. Она берёт значение и помещает его в список, в котором это значение просто повторяется бесконечно. Выражение pure "ха-ха" вернёт ZipList (["ха-ха", "ха-ха", "ха-ха"... Это могло бы сбить с толку, поскольку вы узнали, что функция pure должна помещать значение в минимальный контекст, который по-прежнему возвращает данное значение. И вы могли бы подумать, что бесконечный список чего-либо едва ли является минимальным. Но это имеет смысл при использовании застёгиваемых списков, так как значение должно производиться в каждой позиции. Это также удовлетворяет закону о том, что выражение pure f <*> xs должно быть эквивалентно выражению fmap f xs. Если бы вызов выражения pure 3 просто вернул ZipList [3], вызов pure (*2) <*> ZipList [1,5,10] дал бы в результате ZipList [2], потому что длина результирующего списка из двух застёгнутых списков равна длине более короткого списка из двух. Если мы застегнем конечный список с бесконечным, длина результирующего списка всегда будет равна длине конечного списка.

Так как же застёгиваемые списки работают в аппликативном стиле? Давайте посмотрим.

Ладно, тип ZipList a не имеет экземпляра класса Show, поэтому мы должны использовать функцию getZipList для извлечения обычного списка из застёгиваемого:

```
ghci> getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100,100]
[101,102,103]
ghci> getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100..]
[101,102,103]
ghci> getZipList $ max <$> ZipList [1,2,3,4,5,3] <*> ZipList [5,3,1,2]
[5,3,3,4]
ghci> getZipList $ (,,) <$> ZipList "пар" <*> ZipList "ток" <*> ZipList "вид"
[('п','т','в'),('а','о','и'),('р',кt','д')]
```

ПРИМЕЧАНИЕ. *Функция (,,) – это то же самое, что и анонимная функция \x y z -> (x,y,z). В свою очередь, функция (,) – то же самое, что и \x y -> (x,y).*

Помимо функции `zipWith` в стандартной библиотеке есть такие функции, как `zipWith3`, `zipWith4`, вплоть до 7. Функция `zipWith` берёт функцию, которая принимает два параметра, и застёгивает с её помощью два списка. Функция `zipWith3` берёт функцию, которая принимает три параметра, и застёгивает с её помощью три списка, и т. д. При использовании застёгиваемых списков в аппликативном стиле нам не нужно иметь отдельную функцию застёгивания для каждого числа списков, которые мы хотим застегнуть друг с другом. Мы просто используем аппликативный стиль для застёгивания произвольного количества списков при помощи функции, и это очень удобно.

Аппликативные законы

Как и в отношении обычных функторов, применительно к аппликативным функторам действует несколько законов. Самый главный состоит в том, чтобы выполнялось тождество `pure f <*> x = fmap f x`. В качестве упражнения можете доказать выполнение этого закона для некоторых аппликативных функторов из этой главы. Ниже перечислены другие аппликативные законы:

- `pure id <*> v = v`
- `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- `pure f <*> pure x = pure (f x)`
- `u <*> pure y = pure ($ y) <*> u`

Мы не будем рассматривать их подробно, потому что это заняло бы много страниц и было бы несколько скучно. Если вам интересно, вы можете познакомиться с этими законами поближе и посмотреть, выполняются ли они для некоторых экземпляров.

Полезные функции для работы с аппликативными функторами

Модуль `Control.Applicative` определяет функцию, которая называется `liftA2` и имеет следующий тип:

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
```

Она определена вот так:

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

Она просто применяет функцию между двумя аппликативными значениями, скрывая при этом аппликативный стиль, который мы обсуждали. Однако она ясно демонстрирует, почему аппликативные функторы более мощны по сравнению с обычными.

При использовании обычных функторов мы можем просто отображать одно значение функтора с помощью функций. При использовании аппликативных функторов мы можем применять функцию между несколькими значениями функторов. Интересно также рассматривать тип этой функции в виде `(a -> b -> c) -> (f a -> f b -> f c)`. Когда мы его воспринимаем подобным образом, мы можем сказать, что функция `liftA2` берёт обычную бинарную функцию и преобразует её в функцию, которая работает с двумя аппликативными значениями.

Есть интересная концепция: мы можем взять два аппликативных значения и свести их в одно, которое содержит в себе результаты этих двух аппликативных значений в списке. Например, у нас есть значения `Just 3` и `Just 4`. Предположим, что второй функтор содержит одноэлементный список, так как этого очень легко достичь:

```
ghci> fmap (\x -> [x]) (Just 4)
Just [4]
```

Хорошо, скажем, у нас есть значения `Just 3` и `Just [4]`. Как нам получить `Just [3,4]`? Это просто!

```
ghci> liftA2 (:) (Just 3) (Just [4])
Just [3,4]
ghci> (:) <$> Just 3 <*> Just [4]
Just [3,4]
```

Вспомните, что оператор `:` – это функция, которая принимает элемент и список и возвращает новый список с этим элементом в начале. Теперь, когда у нас есть значение `Just [3,4]`, могли бы ли мы объединить это со значением `Just 2`, чтобы произвести результат `Just [2,3,4]`? Да, могли бы. Похоже, мы можем сводить любое количество аппликативных значений в одно, которое содержит список результатов этих аппликативных значений.

Давайте попробуем реализовать функцию, которая принимает список аппликативных значений и возвращает аппликативное

значение, которое содержит список в качестве своего результирующего значения. Назовём её sequenceA:

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```

А-а-а, рекурсия! Прежде всего смотрим на тип. Он трансформирует список аппликативных значений в аппликативное значение со списком. После этого мы можем заложить некоторую основу для базового случая. Если мы хотим превратить пустой список в аппликативное значение со списком результатов, то просто помещаем пустой список в контекст по умолчанию. Теперь в дело вступает рекурсия. Если у нас есть список с «головой» и «хвостом» (вспомните, x – это аппликативное значение, а xs – это список, состоящий из них), мы вызываем функцию sequenceA с «хвостом», что возвращает аппликативное значение со списком внутри него. Затем мы просто предваряем значением, содержащимся внутри аппликативного значения x, список, находящийся внутри этого аппликативного значения, – вот именно!

Предположим, мы выполняем:

```
sequenceA [Just 1, Just 2]
```

По определению такая запись эквивалентна следующей:

```
(:) <$> Just 1 <*> sequenceA [Just 2]
```

Разбивая это далее, мы получаем:

```
(:) <$> Just 1 <*> ((:) <$> Just 2 <*> sequenceA [])
```

Мы знаем, что вызов выражения sequenceA [] оканчивается в виде Just [], поэтому данное выражение теперь выглядит следующим образом:

```
(:) <$> Just 1 <*> ((:) <$> Just 2 <*> Just [])
```

что аналогично этому:

```
(:) <$> Just 1 <*> Just [2]
```

...что равно Just [1,2]!

Другой способ реализации функции sequenceA – использование свёртки. Вспомните, что почти любая функция, где мы проходим

по списку элемент за элементом и попутно накапливаем результат, может быть реализована с помощью свёртки:

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
```

Мы проходим список с конца, начиная со значения аккумулятора равного `pure []`. Мы применяем функцию `liftA2 (:)` между аккумулятором и последним элементом списка, что даёт в результате аппликативное значение, содержащее одноэлементный список. Затем мы вызываем функцию `liftA2 (:)` с текущим в данный момент последним элементом и текущим аккумулятором и т. д., до тех пор пока у нас не останется только аккумулятор, который содержит список результатов всех аппликативных значений.

Давайте попробуем применить нашу функцию к каким-нибудь аппликативным значениям:

```
ghci> sequenceA [Just 3, Just 2, Just 1]
Just [3,2,1]
ghci> sequenceA [Just 3, Nothing, Just 1]
Nothing
ghci> sequenceA [(+3),(+2),(+1)] 3
[6,5,4]
ghci> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> sequenceA [[1,2,3],[4,5,6],[3,4,4],[]]
[]
```

При использовании со значениями типа `Maybe` функция `sequenceA` создаёт значение типа `Maybe`, содержащее все результаты в виде списка. Если одно из значений равно `Nothing`, результатом тоже является `Nothing`. Это просто расчудесно, когда у вас есть список значений типа `Maybe` и вы заинтересованы в значениях, только когда ни одно из них не равно `Nothing`!

В применении к функциям `sequenceA` принимает список функций и возвращает функцию, которая возвращает список. В нашем примере мы создали функцию, которая приняла число в качестве параметра и применила его к каждой функции в списке, а затем вернула список результатов. Функция `sequenceA [(+3),(+2),(+1)] 3` вызовет функцию `(+3)` с параметром `3`, `(+2)` – с параметром `3` и `(+1)` – с параметром `3` и вернёт все эти результаты в виде списка.

Выполнение выражения (+) <$> (+3) <*> (*2) создаст функцию, которая принимает параметр, передаёт его и функции (+3) и (*2), а затем вызывает оператор + с этими двумя результатами. Соответственно, есть смысл в том, что выражение sequenceA [(+3),(*2)] создаёт функцию, которая принимает параметр и передаёт его всем функциям в списке. Вместо вызова оператора + с результатами функций используется сочетание : и pure [] для накопления этих результатов в список, который является результатом этой функции.

Использование функции sequenceA полезно, когда у нас есть список функций и мы хотим передать им всем один и тот же ввод, а затем просмотреть список результатов. Например, у нас есть число и нам интересно, удовлетворяет ли оно всем предикатам в списке. Вот один из способов сделать это:

```
ghci> map (\f -> f 7) [(>4),(<10),odd]
[True,True,True]
ghci> and $ map (\f -> f 7) [(>4),(<10),odd]
True
```

Вспомните, что функция and принимает список значений типа Bool и возвращает значение True, если все они равны True. Ещё один способ достичь такого же результата – применение функции sequenceA:

```
ghci> sequenceA [(>4),(<10),odd] 7
[True,True,True]
ghci> and $ sequenceA [(>4),(<10),odd] 7
True
```

Выражение sequenceA [(>4),(<10),odd] создаёт функцию, которая примет число, передаст его всем предикатам в списке [(>4),(<10),odd] и вернёт список булевых значений. Она превращает список с типом (Num a) => [a -> Bool] в функцию с типом (Num a) => a -> [Bool]. Правда, клёво, а?

Поскольку списки однородны, все функции в списке должны быть одного и того же типа, конечно же. Вы не можете получить список вроде [ord, (+3)], потому что функция ord принимает символ и возвращает число, тогда как функция (+3) принимает число и возвращает число.

При использовании со значением [] функция sequenceA принимает список списков и возвращает список списков. На самом деле

она создаёт списки, которые содержат все комбинации находящихся в них элементов. Проиллюстрируем это предыдущим примером, который выполнен с применением функции sequenceA, а затем с помощью генератора списков:

```
ghci> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> [[x,y] | x <- [1,2,3], y <- [4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> sequenceA [[1,2],[3,4]]
[[1,3],[1,4],[2,3],[2,4]]
ghci> [[x,y] | x <- [1,2], y <- [3,4]]
[[1,3],[1,4],[2,3],[2,4]]
ghci> sequenceA [[1,2],[3,4],[5,6]]
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]
ghci> [[x,y,z] | x <- [1,2], y <- [3,4], z <- [5,6]]
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]
```

Выражение `(+) <$> [1,2] <*> [4,5,6]` возвращает в результате недетерминированное вычисление `x + y`, где образец `x` принимает каждое значение из `[1,2]`, а `y` принимает каждое значение из `[4,5,6]`. Мы представляем это в виде списка, который содержит все возможные результаты. Аналогичным образом, когда мы выполняем выражение `sequenceA [[1,2],[3,4],[5,6]]`, результатом является недетерминированное вычисление `[x,y,z]`, где образец `x` принимает каждое значение из `[1,2]`, а `y` – каждое значение из `[3,4]` и т. д. Для представления результата этого недетерминированного вычисления мы используем список, где каждый элемент в списке является одним возможным списком. Вот почему результатом является список списков.

При использовании с действиями ввода-вывода функция sequenceA представляет собой то же самое, что и функция sequence! Она принимает список действий ввода-вывода и возвращает действие ввода-вывода, которое выполнит каждое из этих действий и в качестве своего результата будет содержать список результатов этих действий ввода-вывода. Так происходит, потому что чтобы превратить значение `[IO a]` в значение `IO [a]`, чтобы создать действие ввода-вывода, возвращающее список результатов при выполнении, все эти действия ввода-вывода должны быть помещены в последовательность, а затем быть выполненными одно за другим,

когда потребуется результат выполнения. Вы не можете получить результат действия ввода-вывода, не выполнив его!

Давайте поместим три действия ввода-вывода `getLine` в последовательность:

```
ghci> sequenceA [getLine, getLine, getLine]
эй
хо
ух
["эй","хо","ух"]
```

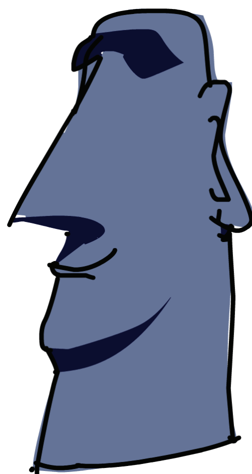
В заключение отмечу, что аппликативные функторы не просто интересны, но и полезны. Они позволяют нам объединять разные вычисления – как, например, вычисления с использованием ввода-вывода, недетерминированные вычисления, вычисления, которые могли окончиться неуспешно, и т. д., – используя аппликативный стиль. Просто с помощью операторов `<$>` и `<*>` мы можем применять обычные функции, чтобы единообразно работать с любым количеством аппликативных функторов и использовать преимущества семантики каждого из них.

12
МОНОИДЫ

В этой главе представлен ещё один полезный и интересный класс типов `Monoid`. Он существует для типов, значения которых могут быть объединены при помощи бинарной операции. Мы рассмотрим, что именно представляют собой моноиды и что утверждают их законы. Затем рассмотрим некоторые моноиды в языке Haskell и обсудим, как они могут нам пригодиться.

И прежде всего давайте взглянем на ключевое слово `newtype`: мы будем часто его использовать, когда углубимся в удивительный мир моноидов.

Оборачивание существующего типа в новый тип



Пока что вы научились создавать свои алгебраические типы данных, используя ключевое слово `data`. Вы также увидели, как можно давать синонимы имеющимся типам с применением ключевого слова `type`. В этом разделе мы рассмотрим, как создаются новые типы на основе имеющихся типов данных с использованием ключевого слова `newtype`. И в первую очередь, конечно, поговорим о том, чем всё это может быть нам полезно.

В главе 11 мы обсудили пару способов, при помощи которых списковый тип может быть аппликативным функтором. Один из этих способов состоит в том, чтобы заставить оператор `<*>` брать каждую функцию из списка, являющегося его левым параметром, и применять её к каждому значению в списке, который находится справа, что в результате возвращает все возможные комбинации применения функции из левого списка к значению в правом:

```
ghci> [(+1),(*100),(*5)] <*> [1,2,3]
[2,3,4,100,200,300,5,10,15]
```

Второй способ заключается в том, чтобы взять первую функцию из списка слева от оператора `<*>` и применить её к первому значению справа, затем взять вторую функцию из списка слева и применить её ко второму значению справа, и т. д. В конечном счёте получается нечто вроде застёгивания двух списков.

Но списки уже имеют экземпляр класса `Applicative`, поэтому как нам определить для списков второй экземпляр класса `Applicative`? Как вы узнали, для этой цели был введён тип `ZipList a`. Он имеет один конструктор данных `ZipList`, у которого только одно поле. Мы помещаем оборачиваемый нами список в это поле. Далее для типа `ZipList` определяется экземпляр класса `Applicative`, чтобы, когда нам понадобится использовать списки в качестве аппликативных функторов для застёгивания, мы могли просто обернуть их с помощью конструктора `ZipList`. Как только мы закончили, разворачиваем их с помощью `getZipList`:

```
ghci> getZipList $ ZipList [(+1),(*100),(*5)] <*> ZipList [1,2,3]
[2,200,15]
```

Итак, какое отношение это имеет к ключевому слову `newtype`? Хорошо, подумайте, как бы мы могли написать объявление `data` для нашего типа `ZipList a`! Вот один из способов:

```
data ZipList a = ZipList [a]
```

Это тип, который обладает лишь одним конструктором данных, и этот конструктор данных имеет только одно поле, которое является списком сущностей. Мы также могли бы использовать синтаксис записей с именованными полями, чтобы автоматически получать функцию, извлекающую список из типа `ZipList`:

```
data ZipList a = ZipList { getZipList :: [a] }
```

Это прекрасно смотрится и на самом деле работает очень хорошо. У нас было два способа сделать существующий тип экземпляром класса типов, поэтому мы использовали ключевое слово `data`, чтобы просто обернуть этот тип в другой, и сделали другой тип экземпляром вторым способом.

Ключевое слово `newtype` в языке Haskell создано специально для тех случаев, когда мы хотим просто взять один тип и обернуть его во что-либо, чтобы представить его как другой тип. В существующих сейчас библиотеках тип `ZipList a` определён вот так:

```
newtype ZipList a = ZipList { getZipList :: [a] }
```

Вместо ключевого слова `data` используется `newtype`. Теперь разберёмся, почему. Ну, к примеру, декларация `newtype` быстрее. Если вы используете ключевое слово `data` для оборачивания типа, появляются «накладные расходы» на все эти оборачивания и разворачивания, когда ваша программа выполняется. Но если вы воспользовались ключевым словом `newtype`, язык Haskell знает, что вы просто применяете его для оборачивания существующего типа в новый тип (отсюда название), поскольку хотите, чтобы внутренне он остался тем же, но имел иной тип. По этой причине язык Haskell может избавиться от оборачивания и разворачивания, как только решит, какое значение какого типа.

Так почему бы всегда не использовать `newtype` вместо `data`? Когда вы создаёте новый тип из имеющегося типа, используя ключевое слово `newtype`, у вас может быть только один конструктор значения, который имеет только одно поле. Но с помощью ключевого слова `data` вы можете создавать типы данных, которые имеют несколько конструкторов значения, и каждый конструктор может иметь ноль или более полей:

```
data Profession = Fighter | Archer | Accountant

data Race = Human | Elf | Orc | Goblin

data PlayerCharacter = PlayerCharacter Race Profession
```

При использовании ключевого слова `newtype` мы можем использовать ключевое слово `deriving` – точно так же, как мы бы делали это с декларацией `data`. Мы можем автоматически порождать

экземпляры для классов Eq, Ord, Enum, Bounded, Show и Read. Если мы породим экземпляр для класса типа, то оборачиваемый нами тип уже должен иметь экземпляр для данного класса типов. Это логично, поскольку ключевое слово newtype всего лишь оборачивает существующий тип. Поэтому теперь мы сможем печатать и сравнивать значения нашего нового типа, если сделаем следующее:

```
newtype CharList = CharList { getCharList :: [Char] } deriving (Eq, Show)
```

Давайте попробуем:

```
ghci> CharList "Вот что мы покажем!"
CharList {getCharList = "Вот что мы покажем!"}
ghci> CharList "бенни" == CharList "бенни"
True
ghci> CharList "бенни" == CharList "устрицы"
False
```

В данном конкретном случае использования ключевого слова newtype конструктор данных имеет следующий тип:

```
CharList :: [Char] -> CharList
```

Он берёт значение типа [Char] и возвращает значение типа CharList. Из предыдущих примеров, где мы использовали конструктор данных CharList, видно, что действительно так оно и есть. И наоборот, функция getCharList, которая была автоматически сгенерирована за нас (потому как мы использовали синтаксис записей с именованными полями в нашей декларации newtype), имеет следующий тип:

```
getCharList :: CharList -> [Char]
```

Она берёт значение типа CharList и преобразует его в значение типа [Char]. Вы можете воспринимать это как оборачивание и разворачивание, но также можете рассматривать это как преобразование значений из одного типа в другой.

Использование ключевого слова newtype для создания экземпляров классов типов

Часто мы хотим сделать наши типы экземплярами определённых классов типов, но параметры типа просто не соответствуют тому,

что нам требуется. Сделать для типа Maybe экземпляр класса Functor легко, потому что класс типов Functor определён вот так:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Поэтому мы просто начинаем с этого:

```
instance Functor Maybe where
```

А потом реализуем функцию fmap.

Все параметры типа согласуются, потому что тип Maybe занимает место идентификатора f в определении класса типов Functor. Если взглянуть на функцию fmap, как если бы она работала только с типом Maybe, в итоге она ведёт себя вот так:

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

Разве это не замечательно? Ну а что если мы бы захотели определить экземпляр класса Functor для кортежей так, чтобы при отображении кортежа с помощью функции fmap входная функция применялась к первому элементу кортежа? Таким образом, выполнение fmap (+3) (1,1) вернуло бы (4,1). Оказывается, что написание экземпляра для этого отчасти затруднительно. При использовании типа Maybe мы просто могли бы написать: instance Functor Maybe where, так как только для конструкторов типа, принимающих ровно один параметр, могут быть определены экземпляры класса Functor. Но, похоже, нет способа сделать что-либо подобное при использовании типа (a,b) так, чтобы в итоге изменялся только параметр типа a, когда мы используем функцию fmap. Чтобы обойти эту проблему, мы можем сделать новый тип из нашего кортежа с помощью ключевого слова newtype так, чтобы второй параметр типа представлял тип первого компонента в кортеже:

```
newtype Pair b a = Pair { getPair :: (a, b) }
```

А теперь мы можем определить для него экземпляр класса Functor так, чтобы функция отображала первый компонент:

```
instance Functor (Pair c) where
   fmap f (Pair (x, y)) = Pair (f x, y)
```

Как видите, мы можем производить сопоставление типов, объявленных через декларацию newtype, с образцом. Мы производим сопоставление, чтобы получить лежащий в основе кортеж, применяем функцию f к первому компоненту в кортеже, а потом используем конструктор значения Pair, чтобы преобразовать кортеж обратно в значение типа Pair b a. Если мы представим, какого типа была бы функция fmap, если бы она работала только с нашими новыми парами, получится следующее:

```
fmap :: (a -> b) -> Pair c a -> Pair c b
```

Опять-таки, мы написали instance Functor (Pair c) where, и поэтому конструктор Pair c занял место идентификатора f в определении класса типов для Functor:

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

Теперь, если мы преобразуем кортеж в тип Pair b a, можно будет использовать с ним функцию fmap, и функция будет отображать первый компонент:

```
ghci> getPair $ fmap (*100) (Pair (2, 3))
(200,3)
ghci> getPair $ fmap reverse (Pair ("вызываю лондон", 3))
("ноднол юавызыв",3)
```

О ленивости newtype

Единственное, что можно сделать с помощью ключевого слова newtype, – это превратить имеющийся тип в новый тип, поэтому внутренне язык Haskell может представлять значения типов, определённых с помощью декларации newtype, точно так же, как и первоначальные, зная в то же время, что их типы теперь различаются. Это означает, что декларация newtype не только зачастую быстрее,

чем data, – её механизм сопоставления с образцом ленивее. Давайте посмотрим, что это значит.

Как вы знаете, язык Haskell по умолчанию ленив, что означает, что какие-либо вычисления будут иметь место только тогда, когда мы пытаемся фактически напечатать результаты выполнения наших функций. Более того, будут произведены только те вычисления, которые необходимы, чтобы наша функция вернула нам результаты. Значение undefined в языке Haskell представляет собой ошибочное вычисление. Если мы попытаемся его вычислить (то есть заставить Haskell на самом деле произвести вычисление), напечатав его на экране, то в ответ последует настоящий припадок гнева – в технической терминологии он называется исключением:

```
ghci> undefined
*** Exception: Prelude.undefined
```

А вот если мы создадим список, содержащий в себе несколько значений undefined, но запросим только «голову» списка, которая не равна undefined, всё пройдёт гладко! Причина в том, что языку Haskell не нужно вычислять какие-либо из остальных элементов в списке, если мы хотим посмотреть только первый элемент. Вот пример:

```
ghci> head [3,4,5,undefined,2,undefined]
3
```

Теперь рассмотрите следующий тип:

```
data CoolBool = CoolBool { getCoolBool :: Bool }
```

Это ваш обыкновенный алгебраический тип данных, который был объявлен с использованием ключевого слова data. Он имеет один конструктор данных, который содержит одно поле с типом Bool. Давайте создадим функцию, которая сопоставляет с образцом значение CoolBool и возвращает значение "привет" вне зависимости от того, было ли значение Bool в CoolBool равно True или False:

```
helloMe :: CoolBool -> String
helloMe (CoolBool _) = "привет"
```

Вместо того чтобы применять эту функцию к обычному значению типа CoolBool, давайте сделаем ей обманный бросок – применим её к значению undefined!

```
ghci> helloMe undefined
*** Exception: Prelude.undefined
```

Тьфу ты! Исключение! Почему оно возникло? Типы, определённые с помощью ключевого слова `data`, могут иметь много конструкторов данных(хотя `CoolBool` имеет только один конструктор). Поэтому для того чтобы понять, согласуется ли значение, переданное нашей функции, с образцом `(CoolBool _)`, язык Haskell должен вычислить значение ровно настолько, чтобы понять, какой конструктор данных был использован, когда мы создавали значение. И когда мы пытаемся вычислить значение `undefined`, будь оно даже небольшим, возникает исключение.

Вместо ключевого слова `data` для `CoolBool` давайте попробуем использовать `newtype`:

```
newtype CoolBool = CoolBool { getCoolBool :: Bool }
```

Нам не нужно изменять нашу функцию `helloMe`, поскольку синтаксис сопоставления с образцом одинаков независимо от того, использовалось ли ключевое слово `newtype` или `data` для объявления вашего типа. Давайте сделаем здесь то же самое и применим `helloMe` к значению `undefined`:

```
ghci> helloMe undefined
"привет"
```

Сработало! Хм-м-м, почему? Ну, как вы уже узнали, когда вы используете ключевое слово `newtype`, язык Haskell внутренне может представлять значения нового типа таким же образом, как и первоначальные значения. Ему не нужно помещать их ещё в одну коробку; он просто должен быть в курсе, что значения имеют разные типы. И поскольку язык Haskell знает, что типы, созданные с помощью ключевого слова `newtype`, могут иметь лишь один конструктор данных и одно поле, ему не нужно вычислять значение,

переданное функции, чтобы убедиться, что значение соответствует образцу (CoolBool _).

Это различие в поведении может казаться незначительным, но на самом деле оно очень важно. Оно показывает, что хотя типы, определённые с помощью деклараций data и newtype, ведут себя одинаково с точки зрения программиста (так как оба имеют конструкторы данных и поля), это фактически два различных механизма. Тогда как ключевое слово data может использоваться для создания ваших новых типов с нуля, ключевое слово newtype предназначено для создания совершенно нового типа из существующего. Сравнение значений деклараций newtype с образцом не похоже на вынимание содержимого коробки (что характерно для деклараций data); это скорее представляет собой прямое преобразование из одного типа в другой.

Ключевое слово type против newtype и data

К этому моменту, возможно, вы с трудом улавливаете различия между ключевыми словами type, data и newtype. Поэтому давайте немного повторим пройденное.

Ключевое слово type предназначено для создания синонимов типов. Мы просто даём другое имя уже существующему типу, чтобы на этот тип было проще сослаться. Скажем, мы написали следующее:

```
type IntList = [Int]
```

Всё, что это нам даёт, – возможность сослаться на тип [Int] как IntList. Их можно использовать взаимозаменяемо. Мы не получаем конструктор данных IntList или что-либо в этом роде. Поскольку идентификаторы [Int] и IntList являются лишь двумя способами сослаться на один и тот же тип, неважно, какое имя мы используем в наших аннотациях типов:

```
ghci> ([1,2,3] :: IntList) ++ ([1,2,3] :: [Int])
[1,2,3,1,2,3]
```

Мы используем синонимы типов, когда хотим сделать наши сигнатуры типов более наглядными. Мы даём типам имена, которые говорят нам что-либо об их предназначении в контексте функций,

где они используются. Например, когда мы использовали ассоциативный список типа [(String, String)] для представления телефонной книги в главе 7, то дали ему синоним типа PhoneBook, чтобы сигнатуры типов наших функций легко читались.

Ключевое слово newtype предназначено для оборачивания существующих типов в новые типы – в основном чтобы для них можно было проще определить экземпляры некоторых классов типов. Когда мы используем ключевое слово newtype для оборачивания существующего типа, получаемый нами тип отделён от исходного. Предположим, мы определяем следующий тип при помощи декларации newtype:

```
newtype CharList = CharList { getCharList :: [Char] }
```

Нельзя использовать оператор ++, чтобы соединить значение типа CharList и список типа [Char]. Нельзя даже использовать оператор ++, чтобы соединить два значения типа CharList, потому что оператор ++ работает только со списками, а тип CharList не является списком, хотя можно сказать, что CharList содержит список. Мы можем, однако, преобразовать два значения типа CharList в списки, соединить их с помощью оператора ++, а затем преобразовать получившееся обратно в CharList.

Когда в наших объявлениях типа newtype мы используем синтаксис записей с именованными полями, то получаем функции для преобразования между новым типом и изначальным типом – а именно конструктор данных нашего типа newtype и функцию для извлечения значения из его поля. Для нового типа также автоматически не определяются экземпляры классов типов, для которых есть экземпляры исходного типа, поэтому нам необходимо их сгенерировать (ключевое слово deriving) либо определить вручную.

На деле вы можете воспринимать декларации newtype как декларации data, только с одним конструктором данных и одним полем. Если вы поймаете себя на написании такого объявления, рассмотрите использование newtype.

Ключевое слово data предназначено для создания ваших собственных типов данных. Ими вы можете увлечься не на шутку. Они могут иметь столько конструкторов и полей, сколько вы пожелаете, и использоваться для реализации любого алгебраического типа

данных – всего, начиная со списков и Maybe-подобных типов и заканчивая деревьями.

Подведём итог вышесказанному. Используйте ключевые слова следующим образом:

- если вы просто хотите, чтобы ваши сигнатуры типов выглядели понятнее и были более наглядными, вам, вероятно, нужны синонимы типов;
- если вы хотите взять существующий тип и обернуть его в новый, чтобы определить для него экземпляр класса типов, скорее всего, вам пригодится newtype;
- если вы хотите создать что-то совершенно новое, есть шанс, что вам поможет ключевое слово data.

В общих чертах о моноидах

Классы типов в языке Haskell используются для представления интерфейса к типам, которые обладают неким схожим поведением. Мы начали с простых классов типов вроде класса Eq, предназначенного для типов, значения которых можно сравнить, и класса Ord – для сущностей, которые можно упорядочить. Затем перешли к более интересным классам типов, таким как классы Functor и Applicative.

Создавая тип, мы думаем о том, какие поведения он поддерживает (как он может действовать), а затем решаем, экземпляры каких классов типов для него определить, основываясь на необходимом нам поведении. Если разумно, чтобы значения нашего типа были сравниваемыми, мы определяем для нашего типа экземпляр класса Eq. Если мы видим, что наш тип является чем-то вроде функтора – определяем для него экземпляр класса Functor, и т. д.

Теперь рассмотрим следующее: оператор * – это функция, которая принимает два числа и перемножает их. Если мы умножим какое-нибудь число на 1, результат всегда равен этому числу.

Неважно, выполним ли мы `1 * x` или `x * 1` – результат всегда равен `x`. Аналогичным образом оператор `++` – это функция, которая принимает две сущности и возвращает третью. Но вместо того, чтобы перемножать числа, она принимает два списка и конкатенирует их. И так же, как оператор `*`, она имеет определённое значение, которое не изменяет другое значение при использовании с оператором `++`. Этим значением является пустой список: `[]`.

```
ghci> 4 * 1
4
ghci> 1 * 9
9
ghci> [1,2,3] ++ []
[1,2,3]
ghci> [] ++ [0.5, 2.5]
[0.5,2.5]
```

Похоже, что оператор `*` вместе с `1` и оператор `++` наряду с `[]` разделяют некоторые общие свойства:

- функция принимает два параметра;
- параметры и возвращаемое значение имеют одинаковый тип;
- существует такое значение, которое не изменяет другие значения, когда используется с бинарной функцией.

Есть и ещё нечто общее между двумя этими операциями, хотя это может быть не столь очевидно, как наши предыдущие наблюдения. Когда у нас есть три и более значения и нам необходимо использовать бинарную функцию для превращения их в один результат, то порядок, в котором мы применяем бинарную функцию к значениям, неважен. Например, независимо от того, выполним ли мы `(3 * 4) * 5` или `3 * (4 * 5)`, результат будет равен `60`. То же справедливо и для оператора `++`:

```
ghci> (3 * 2) * (8 * 5)
240
ghci> 3 * (2 * (8 * 5))
240
ghci> "ой" ++ ("лю" ++ "ли")
"ойлюли"
ghci> ("ой" ++ "лю") ++ "ли"
"ойлюли"
```

Мы называем это свойство ассоциативностью. Оператор * ассоциативен, оператор ++ тоже. Однако оператор -, например, не ассоциативен, поскольку выражения (5 - 3) - 4 и 5 - (3 - 4) возвращают различные результаты.

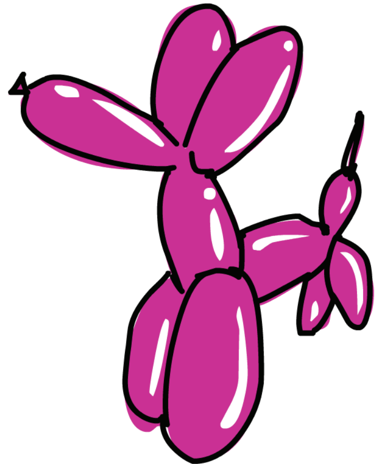
Зная об этих свойствах, мы наконец-то наткнулись на моноиды!

Класс типов Monoid

Моноид состоит из ассоциативной бинарной функции и значения, которое действует как единица (*единичное* или *нейтральное значение*) по отношению к этой функции. Когда что-то действует как единица по отношению к функции, это означает, что при вызове с данной функцией и каким-то другим значением результат всегда равен этому другому значению. Значение 1 является единицей по отношению к оператору *, а значение [] является единицей по отношению к оператору ++. В мире языка Haskell есть множество других моноидов, поэтому существует целый класс типов Monoid. Он предназначен для типов, которые могут действовать как моноиды. Давайте посмотрим, как определён этот класс типов:

```
class Monoid m where
   mempty :: m
   mappend :: m -> m -> m
   mconcat :: [m] -> m
   mconcat = foldr mappend mempty
```

Класс типов Monoid определён в модуле Data.Monoid. Давайте потратим некоторое время, чтобы как следует с ним познакомиться.



Прежде всего, нам видно, что экземпляры класса Monoid могут быть определены только для конкретных типов, потому что идентификатор m в определении класса типов не принимает никаких параметров типа. В этом состоит отличие от классов Functor и Applicative, которые требуют, чтобы их экземплярами были конструкторы типа, принимающие один параметр.

Первой функцией является `mempty`. На самом деле это не функция, поскольку она не принимает параметров. Это полиморфная константа вроде `minBound` из класса `Bounded`. Значение `mempty` представляет единицу для конкретного моноида.

Далее, у нас есть функция `mappend`, которая, как вы уже, наверное, догадались, является бинарной. Она принимает два значения одного типа и возвращает ещё одно значение того же самого типа. Решение назвать так функцию `mappend` было отчасти неудачным, поскольку это подразумевает, что мы в некотором роде присоединяем два значения. Тогда как оператор ++ действительно принимает два списка и присоединяет один в конец другого, оператор * на самом деле не делает какого-либо присоединения – два числа просто перемножаются. Когда вы встретите другие экземпляры класса `Monoid`, вы поймёте, что большинство из них тоже не присоединяют значения. Поэтому избегайте мыслить в терминах присоединения; просто рассматривайте `mappend` как бинарную функцию, которая принимает два моноидных значения и возвращает третье.

Последней функцией в определении этого класса типов является `mconcat`. Она принимает список моноидных значений и сокращает их до одного значения, применяя функцию `mappend` между элементами списка. Она имеет реализацию по умолчанию, которая просто принимает значение `mempty` в качестве начального и сворачивает список справа с помощью функции `mappend`. Поскольку реализация по умолчанию хорошо подходит для большинства экземпляров, мы не будем сильно переживать по поводу функции `mconcat`. Когда для какого-либо типа определяют экземпляр класса `Monoid`, достаточно реализовать всего лишь методы `mempty` и `mappend`. Хотя для некоторых экземпляров функцию `mconcat` можно реализовать более эффективно, в большинстве случаев реализация по умолчанию подходит идеально.

Законы моноидов

Прежде чем перейти к более конкретным экземплярам класса `Monoid`, давайте кратко рассмотрим законы моноидов.

Вы узнали, что должно иметься значение, которое действует как тождество по отношению к бинарной функции, и что бинарная функция должна быть ассоциативна. Можно создать экземпляры

класса `Monoid`, которые не следуют этим правилам, но такие экземпляры никому не нужны, поскольку, когда мы используем класс типов `Monoid`, мы полагаемся на то, что его экземпляры ведут себя как моноиды. Иначе какой в этом смысл? Именно поэтому при создании экземпляров класса `Monoid` мы должны убедиться, что они следуют нижеприведённым законам:

- `` mempty `mappend` x = x ``
- `` x `mappend` mempty = x ``
- `` (x `mappend` y) `mappend` z = x `mappend` (y `mappend` z) ``

Первые два закона утверждают, что значение `mempty` должно вести себя как единица по отношению к функции `mappend`, а третий говорит, что функция `mappend` должна быть ассоциативна (порядок, в котором мы используем функцию `mappend` для сведения нескольких моноидных значений в одно, не имеет значения). Язык Haskell не проверяет определяемые экземпляры на соответствие этим законам, поэтому мы должны быть внимательными, чтобы наши экземпляры действительно выполняли их.

Познакомьтесь с некоторыми моноидами

Теперь, когда вы знаете, что такое моноиды, давайте изучим некоторые типы в языке Haskell, которые являются моноидами, посмотрим, как выглядят экземпляры класса `Monoid` для них, и поговорим об их использовании.

Списки являются моноидами

Да, списки являются моноидами! Как вы уже видели, функция ++ с пустым списком `[]` образуют моноид. Экземпляр очень прост:

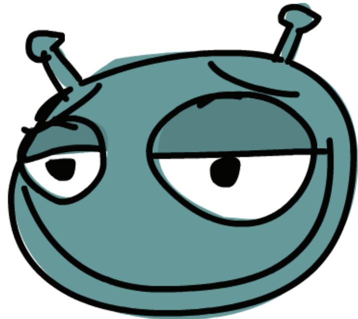
```
instance Monoid [a] where
   mempty = []
   mappend = (++)
```

Для списков имеется экземпляр класса `Monoid` независимо от типа элементов, которые они содержат. Обратите внимание, что мы написали `instance Monoid [a]`, а не `instance Monoid []`, поскольку класс `Monoid` требует конкретный тип для экземпляра.

При тестировании мы не встречаем сюрпризов:

```
ghci> [1,2,3] `mappend` [4,5,6]
[1,2,3,4,5,6]
ghci> ("один" `mappend` "два") `mappend` "три"
"одиндватри"
ghci> "один" `mappend` ("два" `mappend` "три")
"одиндватри"
ghci> "один" `mappend` "два" `mappend` "три"
"одиндватри"
ghci> "бах" `mappend` mempty
"бах"
ghci> mconcat [[1,2],[3,6],[9]]
[1,2,3,6,9]
ghci> mempty :: [a]
[]
```

Обратите внимание, что в последней строке мы написали явную аннотацию типа. Если бы было написано просто `mempty`, то интерпретатор GHCi не знал бы, какой экземпляр использовать, поэтому мы должны были сказать, что нам нужен списковый экземпляр. Мы могли использовать общий тип `[a]` (в отличие от указания `[Int]` или `[String]`), потому что пустой список может действовать так, будто он содержит любой тип.



Поскольку функция `mconcat` имеет реализацию по умолчанию, мы получаем её просто так, когда определяем экземпляр класса `Monoid` для какого-либо типа. В случае со списком функция `mconcat` соответствует просто функции `concat`. Она принимает список списков и «разглаживает» его, потому что это равнозначно вызову оператора ++ между всеми смежными списками, содержащимися в списке.

Законы моноидов действительно выполняются для экземпляра списка. Когда у нас есть несколько списков и мы объединяем их с помощью функции `mappend` (или ++), не имеет значения, какие списки мы соединяем первыми, поскольку так или иначе они соединяются на концах. Кроме того, пустой список действует как единица, поэтому всё хорошо.

Обратите внимание, что моноиды не требуют, чтобы результат выражения a `mappend` b был равен результату выражения b `mappend` a. В случае со списками они очевидно не равны:

```
ghci> "один" `mappend` "два"
"одиндва"
ghci> "два" `mappend` "один"
"дваодин"
```

И это нормально. Тот факт, что при умножении выражения 3 * 5 и 5 * 3 дают один и тот же результат, – это просто свойство умножения, но оно не выполняется для большинства моноидов.

Типы Product и Sum

Мы уже изучили один из способов рассматривать числа как моноиды: просто позволить бинарной функции быть оператором *, а единичному значению – быть 1. Ещё один способ для чисел быть моноидами состоит в том, чтобы в качестве бинарной функции выступал оператор +, а в качестве единичного значения – значение 0:

```
ghci> 0 + 4
4
ghci> 5 + 0
5
ghci> (1 + 3) + 5
9
ghci> 1 + (3 + 5)
9
```

Законы моноидов выполняются, потому что если вы прибавите 0 к любому числу, результатом будет то же самое число. Сложение также ассоциативно, поэтому здесь у нас нет никаких проблем.

Итак, в нашем распоряжении два одинаково правомерных способа для чисел быть моноидами. Какой же способ выбрать?.. Ладно, мы не обязаны выбирать! Вспомните, что когда имеется несколько способов определения для какого-то типа экземпляра одного и того же класса типов, мы можем обернуть этот тип в декларацию newtype, а затем сделать для нового типа экземпляр класса типов по-другому. Можно совместить несовместимое.

Модуль Data.Monoid экспортирует для этого два типа: Product и Sum.

Product определён вот так:

```
newtype Product a = Product { getProduct :: a }
   deriving (Eq, Ord, Read, Show, Bounded)
```

Это всего лишь обёртка newtype с одним параметром типа наряду с некоторыми порождёнными экземплярами. Его экземпляр для класса Monoid выглядит примерно так:

```
instance Num a => Monoid (Product a) where
   mempty = Product 1
   Product x `mappend` Product y = Product (x * y)
```

Значение mempty – это просто 1, обёрнутая в конструктор Product. Функция mappend производит сопоставление конструктора Product с образцом, перемножает два числа, а затем оборачивает результирующее число. Как вы можете видеть, имеется ограничение класса Num a. Это значит, что Product a является экземпляром Monoid для всех значений типа a, для которых уже имеется экземпляр класса Num. Для того чтобы использовать тип Product a в качестве моноида, мы должны произвести некоторое оборачивание и разворачивание newtype:

```
ghci> getProduct $ Product 3 `mappend` Product 9
27
ghci> getProduct $ Product 3 `mappend` mempty
3
ghci> getProduct $ Product 3 `mappend` Product 4 `mappend` Product 2
24
ghci> getProduct . mconcat . map Product $ [3,4,2]
24
```

Тип Sum определён в том же духе, что и тип Product, и экземпляр тоже похож. Мы используем его точно так же:

```
ghci> getSum $ Sum 2 `mappend` Sum 9
11
ghci> getSum $ mempty `mappend` Sum 3
3
ghci> getSum . mconcat . map Sum $ [1,2,3]
6
```

Типы Any и All

Ещё одним типом, который может действовать как моноид двумя разными, но одинаково допустимыми способами, является Bool. Первый способ состоит в том, чтобы заставить функцию ||, которая представляет собой логическое ИЛИ, действовать как бинарная функция, используя False в качестве единичного значения. Если при использовании логического ИЛИ какой-либо из параметров равен True, функция возвращает True; в противном случае она возвращает False. Поэтому если мы используем False в качестве единичного значения, операция ИЛИ вернёт False при использовании с False – и True при использовании с True. Конструктор newtype Any аналогичным образом имеет экземпляр класса Monoid. Он определён вот так:

```
newtype Any = Any { getAny :: Bool }
   deriving (Eq, Ord, Read, Show, Bounded)
```

А его экземпляр выглядит так:

```
instance Monoid Any where
   mempty = Any False
   Any x `mappend` Any y = Any (x || y)
```

Он называется Any, потому что x `mappend` y будет равно True, если *любое* из этих двух значений равно True. Даже когда три или более значений Bool, обёрнутых в Any, объединяются с помощью функции mappend, результат будет содержать True, если любое из них равно True.

```
ghci> getAny $ Any True `mappend` Any False
True
ghci> getAny $ mempty `mappend` Any True
True
ghci> getAny . mconcat . map Any $ [False, False, False, True]
True
ghci> getAny $ mempty `mappend` mempty
False
```

Другой возможный вариант экземпляра класса Monoid для типа Bool – всё как бы наоборот: заставить оператор && быть бинарной функцией, а затем сделать значение True единичным значением.

Логическое И вернёт True, только если оба его параметра равны True.

Это объявление newtype:

```
newtype All = All { getAll :: Bool }
   deriving (Eq, Ord, Read, Show, Bounded)
```

А это экземпляр:

```
instance Monoid All where
   mempty = All True
   All x `mappend` All y = All (x && y)
```

Когда мы объединяем значения типа All с помощью функции mappend, результатом будет True только в случае, если все значения, использованные в функции mappend, равны True:

```
ghci> getAll $ mempty `mappend` All True
True
ghci> getAll $ mempty `mappend` All False
False
ghci> getAll . mconcat . map All $ [True, True, True]
True
ghci> getAll . mconcat . map All $ [True, True, False]
False
```

Так же, как при использовании умножения и сложения, мы обычно явно указываем бинарные функции вместо оборачивания их в значения newtype и последующего использования функций mappend и mempty. Функция mconcat кажется полезной для типов Any и All, но обычно проще использовать функции or и and. Функция or принимает списки значений типа Bool и возвращает True, если какое-либо из них равно True. Функция and принимает те же значения и возвращает значение True, если все из них равны True.

Моноид Ordering

Помните тип Ordering? Он используется в качестве результата при сравнении сущностей и может иметь три значения: LT, EQ и GT, которые соответственно означают «меньше, чем», «равно» и «больше, чем».

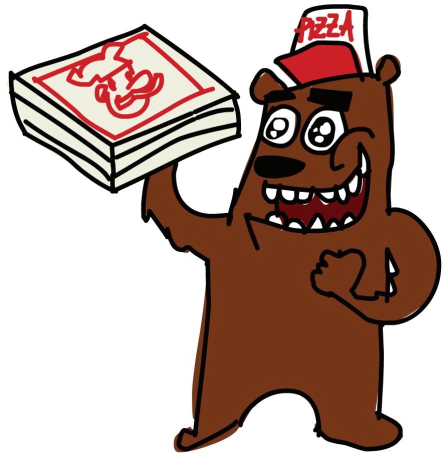
```
ghci> 1 `compare` 2
LT
```

```
ghci> 2 `compare` 2
EQ
ghci> 3 `compare` 2
GT
```

При использовании чисел и значений типа `Bool` поиск моноидов сводился к просмотру уже существующих широко применяемых функций и их проверке на предмет того, проявляют ли они какое-либо поведение, присущее моноидам. При использовании типа `Ordering` нам придётся приложить больше старания, чтобы распознать моноид. Оказывается, его экземпляр класса `Monoid` настолько же интуитивен, насколько и предыдущие, которые мы уже встречали, и кроме того, весьма полезен:

```
instance Monoid Ordering where
   mempty = EQ
   LT `mappend` _ = LT
   EQ `mappend` y = y
   GT `mappend` _ = GT
```

Экземпляр определяется следующим образом: когда мы объединяем два значения типа `Ordering` с помощью функции `mappend`, сохраняется значение слева, если значение слева не равно `EQ`. Если значение слева равно `EQ`, результатом будет значение справа. Единичным значением является `EQ`. На первый взгляд, такой выбор может показаться несколько случайным, но на самом деле он имеет сходство с тем, как мы сравниваем слова в алфавитном порядке. Мы смотрим на первые две буквы, и, если они отличаются, уже можем решить, какое слово шло бы первым в словаре. Если же первые буквы равны, то мы переходим к сравнению следующей пары букв, повторяя процесс[1].



[1] Специалисты по нечёткой логике могут увидеть в этом определении троичную логику Лукасевича. – *Прим. ред.*

Например, сравнивая слова «*ox*» и «*on*», мы видим, что первые две буквы каждого слова равны, а затем продолжаем сравнивать вторые буквы. Поскольку «*x*» в алфавите идёт после «*n*», мы знаем, в каком порядке должны следовать эти слова. Чтобы лучше понять, как `EQ` является единичным значением, обратите внимание, что если бы мы втиснули одну и ту же букву в одну и ту же позицию в обоих словах, их расположение друг относительно друга в алфавитном порядке осталось бы неизменным; к примеру, слово «*oix*» будет по-прежнему идти следом за «*oin*».

Важно, что в экземпляре класса `Monoid` для типа `Ordering` выражение ``x `mappend` y`` не равно выражению ``y `mappend` x``. Поскольку первый параметр сохраняется, если он не равен `EQ`, ``LT `mappend` GT`` в результате вернёт `LT`, тогда как ``GT `mappend` LT`` в результате вернёт `GT`:

```
ghci> LT `mappend` GT
LT
ghci> GT `mappend` LT
GT
ghci> mempty `mappend` LT
LT
ghci> mempty `mappend` GT
GT
```

Хорошо, так чем же этот моноид полезен? Предположим, мы пишем функцию, которая принимает две строки, сравнивает их длину и возвращает значение типа `Ordering`. Но если строки имеют одинаковую длину, то вместо того, чтобы сразу вернуть значение `EQ`, мы хотим установить их расположение в алфавитном порядке.

Вот один из способов записать это:

```
lengthCompare :: String -> String -> Ordering
lengthCompare x y = let a = length x `compare` length y
                        b = x `compare` y
                    in if a == EQ then b else a
```

Результат сравнения длин мы присваиваем образцу `a`, результат сравнения по алфавиту – образцу `b`; затем, если оказывается, что длины равны, возвращаем их порядок по алфавиту.

Но, имея представление о том, что тип `Ordering` является моноидом, мы можем переписать эту функцию в более простом виде:

```
import Data.Monoid

lengthCompare :: String -> String -> Ordering
lengthCompare x y = (length x `compare` length y) `mappend`(x `compare` y)
```

Давайте опробуем это:

```
ghci> lengthCompare "ямб" "хорей"
LT
ghci> lengthCompare "ямб" "хор"
GT
```

Вспомните, что когда мы используем функцию `mappend`, сохраняется её левый параметр, если он не равен значению `EQ`; если он равен `EQ`, сохраняется правый. Вот почему мы поместили сравнение, которое мы считаем первым, более важным критерием, в качестве первого параметра. Теперь предположим, что мы хотим расширить эту функцию, чтобы она также сравнивала количество гласных звуков, и установить это вторым по важности критерием для сравнения. Мы изменяем её вот так:

```
import Data.Monoid

lengthCompare :: String -> String -> Ordering
lengthCompare x y = (length x `compare` length y) `mappend`
                    (vowels x `compare` vowels y) `mappend`
                    (x `compare` y)
   where vowels = length . filter (`elem` "аеёиоуыэюя")
```

Мы создали вспомогательную функцию, которая принимает строку и сообщает нам, сколько она содержит гласных звуков, сначала отфильтровывая в ней только буквы, находящиеся в строке `"аеёиоуыэюя"`, а затем применяя функцию `length`.

```
ghci> lengthCompare "ямб" "абыр"
LT
ghci> lengthCompare "ямб" "абы"
LT
ghci> lengthCompare "ямб" "абр"
GT
```

В первом примере длины оказались различными, поэтому вернулось `LT`, так как длина слова `"ямб"` меньше длины слова `"абыр"`. Во

втором примере длины равны, но вторая строка содержит больше гласных звуков, поэтому опять возвращается `LT`. В третьем примере они обе имеют одинаковую длину и одинаковое количество гласных звуков, поэтому сравниваются по алфавиту, и слово `"ямб"` выигрывает.

Моноид для типа `Ordering` очень полезен, поскольку позволяет нам без труда сравнивать сущности по большому количеству разных критериев и помещать сами эти критерии по порядку, начиная с наиболее важных и заканчивая наименее важными.

Моноид Maybe

Рассмотрим несколько способов, которыми для типа `Maybe a` могут быть определены экземпляры класса `Monoid`, и обсудим, чем эти экземпляры полезны.

Один из способов состоит в том, чтобы обрабатывать тип `Maybe a` как моноид, только если его параметр типа `a` тоже является моноидом, а потом реализовать функцию `mappend` так, чтобы она использовала операцию `mappend` для значений, обёрнутых в конструктор `Just`. Мы используем значение `Nothing` как единичное, и поэтому если одно из двух значений, которые мы объединяем с помощью функции `mappend`, равно `Nothing`, мы оставляем другое значение. Вот объявление экземпляра:

```
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    Nothing `mappend` m = m
    m `mappend` Nothing = m
    Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

Обратите внимание на ограничение класса. Оно говорит, что тип `Maybe` является моноидом, только если для типа `a` определён экземпляр класса `Monoid`. Если мы объединяем нечто со значением `Nothing`, используя функцию `mappend`, результатом является это нечто. Если мы объединяем два значения `Just` с помощью функции `mappend`, то содержимое значений `Just` объединяется с помощью этой функции, а затем оборачивается обратно в конструктор `Just`. Мы можем делать это, поскольку ограничение класса гарантирует, что тип значения, которое находится внутри `Just`, имеет экземпляр класса `Monoid`.

```
ghci> Nothing `mappend` Just "андрей"
Just "андрей"
ghci> Just LT `mappend` Nothing
Just LT
ghci> Just (Sum 3) `mappend` Just (Sum 4)
Just (Sum {getSum = 7})
```

Это полезно, когда мы имеем дело с моноидами как с результатами вычислений, которые могли окончиться неуспешно. Из-за наличия этого экземпляра нам не нужно проверять, окончились ли вычисления неуспешно, определяя, вернули они значение Nothing или Just; мы можем просто продолжить обрабатывать их как обычные моноиды.

Но что если тип содержимого типа Maybe не имеет экземпляра класса Monoid? Обратите внимание: в предыдущем объявлении экземпляра единственный случай, когда мы должны полагаться на то, что содержимые являются моноидами, – это когда оба параметра функции mappend обёрнуты в конструктор Just. Когда мы не знаем, являются ли содержимые моноидами, мы не можем использовать функцию mappend между ними; так что же нам делать? Ну, единственное, что мы можем сделать, – это отвергнуть второе значение и оставить первое. Для этой цели существует тип First a. Вот его определение:

```
newtype First a = First { getFirst :: Maybe a }
   deriving (Eq, Ord, Read, Show)
```

Мы берём тип Maybe a и оборачиваем его с помощью декларации newtype. Экземпляр класса Monoid в данном случае выглядит следующим образом:

```
instance Monoid (Firsta) where
   mempty = First Nothing
   First (Just x) `mappend` _ = First (Just x)
   First Nothing `mappend` x = x
```

Значение mempty – это просто Nothing, обёрнутое с помощью конструктора First. Если первый параметр функции mappend является значением Just, мы игнорируем второй. Если первый параметр – Nothing, тогда мы возвращаем второй параметр в качестве результата независимо от того, является ли он Just или Nothing:

```
ghci> getFirst $ First (Just 'a') `mappend` First (Just 'b')
Just 'a'
ghci> getFirst $ First Nothing `mappend` First (Just 'b')
Just 'b'
ghci> getFirst $ First (Just 'a') `mappend` First Nothing
Just 'a'
```

Тип `First` полезен, когда у нас есть множество значений типа `Maybe` и мы хотим знать, является ли какое-либо из них значением `Just`. Для этого годится функция `mconcat`:

```
ghci> getFirst . mconcat . map First $ [Nothing, Just 9, Just 10]
Just 9
```

Если нам нужен моноид на значениях `Maybe a` – такой, чтобы оставался второй параметр, когда оба параметра функции `mappend` являются значениями `Just`, то модуль `Data.Monoid` предоставляет тип `Last a`, который работает, как и тип `First a`, но при объединении с помощью функции `mappend` и использовании функции `mconcat` сохраняется последнее значение, не являющееся `Nothing`:

```
ghci> getLast . mconcat . map Last $ [Nothing, Just 9, Just 10]
Just 10
ghci> getLast $ Last (Just "один") `mappend` Last (Just "два")
Just "two"
```

Свёртка на моноидах

Один из интересных способов ввести моноиды в работу заключается в том, чтобы они помогали нам определять свёртки над различными структурами данных. До сих пор мы производили свёртки только над списками, но списки – не единственная структура данных, которую можно свернуть. Мы можем определять свёртки почти над любой структурой данных. Особенно хорошо поддаются свёртке деревья.

Поскольку существует так много структур данных, которые хорошо работают со свёртками, был введён класс типов `Foldable`. Подобно тому как класс `Functor` предназначен для сущностей, которые можно отображать, класс `Foldable` предназначен для вещей, которые могут быть свёрнуты! Его можно найти в модуле `Data.Foldable`;

и, поскольку он экспортирует функции, имена которых конфликтуют с именами функций из модуля `Prelude`, его лучше импортировать, квалифицируя (и подавать с базиликом!):

```
import qualified Data.Foldable as F
```

Чтобы сэкономить драгоценные нажатия клавиш, мы импортировали его, квалифицируя как `F`.

Так какие из некоторых функций определяет этот класс типов? Среди них есть функции `foldr`, `foldl`, `foldr1` и `foldl1`. Ну и?.. Мы уже давно знакомы с ними! Что ж в этом нового? Давайте сравним типы функции `foldr` из модуля `Foldable` и одноимённой функции из модуля `Prelude`, чтобы узнать, чем они отличаются:

```
ghci> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
ghci> :t F.foldr
F.foldr :: (F.Foldable t) => (a -> b -> b) -> b -> t a -> b
```

А-а-а! Значит, в то время как функция `foldr` принимает список и сворачивает его, функция `foldr` из модуля `Data.Foldable` принимает любой тип, который можно свернуть, – не только списки! Как и ожидалось, обе функции `foldr` делают со списками одно и то же:

```
ghci> foldr (*) 1 [1,2,3]
6
ghci> F.foldr (*) 1 [1,2,3]
6
```

Другой структурой данных, поддерживающей свёртку, является `Maybe`, которую мы все знаем и любим!

```
ghci> F.foldl (+) 2 (Just 9)
11
ghci> F.foldr (||) False (Just True)
True
```

Но сворачивание значения `Maybe` не очень-то интересно. Оно действует просто как список с одним элементом, если это значение `Just`, и как пустой список, если это значение `Nothing`. Давайте рассмотрим чуть более сложную структуру данных.

Помните древовидную структуру данных из главы 7? Мы определили её так:

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show)
```

Вы узнали, что дерево – это либо пустое дерево, которое не содержит никаких значений, либо узел, который содержит одно значение, а также два других дерева. После того как мы его определили, мы сделали для него экземпляр класса `Functor`, и это дало нам возможность отображать его с помощью функций, используя функцию `fmap`. Теперь мы определим для него экземпляр класса `Foldable`, чтобы у нас появилась возможность производить его свёртку.

Один из способов сделать для конструктора типа экземпляр класса `Foldable` состоит в том, чтобы просто напрямую реализовать для него функцию `foldr`. Но другой, часто более простой способ состоит в том, чтобы реализовать функцию `foldMap`, которая также является методом класса типов `Foldable`. У неё следующий тип:

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

Её первым параметром является функция, принимающая значение того типа, который содержит наша сворачиваемая структура (обозначен здесь как `a`), и возвращающая моноидное значение. Второй её параметр – сворачиваемая структура, содержащая значения типа `a`. Эта функция отображает структуру с помощью заданной функции, таким образом, производя сворачиваемую структуру, которая содержит моноидные значения. Затем, объединяя эти моноидные значения с помощью функции `mappend`, она сводит их все в одно моноидное значение. На данный момент функция может показаться несколько странной, но вы увидите, что её очень просто реализовать. И такой реализации достаточно, чтобы определить для нашего типа экземпляр класса `Foldable`! Поэтому если мы просто реализуем функцию `foldMap` для какого-либо типа, то получаем функции `foldr` и `foldl` для этого типа даром!

Вот как мы делаем экземпляр класса `Foldable` для типа:

```
instance F.Foldable Tree where
   foldMap f EmptyTree = mempty
   foldMap f (Node x l r) = F.foldMap f l `mappend`
                            f x           `mappend`
                            F.foldMap f r
```

Если нам предоставлена функция, которая принимает элемент нашего дерева и возвращает моноидное значение, то как превратить

наше целое дерево в одно моноидное значение? Когда мы использовали функцию `fmap` с нашим деревом, мы применяли функцию, отображая с её помощью узел, а затем рекурсивно отображали с помощью этой функции левое поддерево, а также правое поддерево. Здесь наша задача состоит не только в отображении с помощью функции, но также и в соединении значений в одно моноидное значение с использованием функции `mappend`. Сначала мы рассматриваем случай с пустым деревом – печальным и одиноким деревцем, у которого нет никаких значений или поддеревьев. Оно не содержит значений, которые мы можем предоставить нашей функции, создающей моноид, поэтому мы просто говорим, что если наше дерево пусто, то моноидное значение, в которое оно будет превращено, равно значению `mempty`.

Случай с непустым узлом чуть более интересен. Он содержит два поддерева, а также значение. В этом случае мы рекурсивно отображаем левое и правое поддеревья с помощью одной и той же функции `f`, используя рекурсивный вызов функции `foldMap`. Вспомните, что наша функция `foldMap` возвращает в результате одно моноидное значение. Мы также применяем нашу функцию `f` к значению в узле. Теперь у нас есть три моноидных значения (два из наших поддеревьев и одно – после применения `f` к значению в узле), и нам просто нужно соединить их. Для этой цели мы используем функцию `mappend`, и естественным образом левое поддерево идёт первым, затем – значение узла, а потом – правое поддерево[2].

Обратите внимание, что нам не нужно было предоставлять функцию, которая принимает значение и возвращает моноидное значение. Мы принимаем эту функцию как параметр к `foldMap`, и всё, что нам нужно решить, – это где применить эту функцию и как соединить результирующие моноиды, которые она возвращает.

---

[2] Это определение представляет собой один из возможных способов обхода двоичного дерева: «левый – корень – правый». Читатель может самостоятельно реализовать экземпляры для представления других способов обхода двоичных деревьев. – *Прим. ред.*

Теперь, когда у нас есть экземпляр класса Foldable для нашего типа, представляющего дерево, мы получаем функции foldr и foldl даром! Рассмотрите вот это дерево:

```
testTree = Node 5
            (Node 3
               (Node 1 EmptyTree EmptyTree)
               (Node 6 EmptyTree EmptyTree)
            )
            (Node 9
               (Node 8 EmptyTree EmptyTree)
               (Node 10 EmptyTree EmptyTree)
            )
```

У него значение 5 в качестве его корня, а его левый узел содержит значение 3 со значениями 1 слева и 6 справа. Правый узел корня содержит значение 9, а затем значения 8 слева от него и 10 в самой дальней части справа. Используя экземпляр класса Foldable, мы можем производить всё те же свёртки, что и над списками:

```
ghci> F.foldl (+) 0 testTree
42
ghci> F.foldl (*) 1 testTree
64800
```

Функция foldMap полезна не только для создания новых экземпляров класса Foldable. Она также очень удобна для превращения нашей структуры в одно моноидное значение. Например, если мы хотим узнать, равно ли какое-либо из чисел нашего дерева 3, мы можем сделать следующее:

```
ghci> getAny $ F.foldMap (\x -> Any $ x == 3) testTree
True
```

Здесь анонимная функция \x -> Any $ x == 3 – это функция, которая принимает число и возвращает моноидное значение: значение Bool, обёрнутое в тип Any. Функция foldMap применяет эту функцию к каждому элементу нашего дерева, а затем превращает получившиеся моноиды в один моноид с помощью вызова функции mappend. Предположим, мы выполняем следующее:

```
ghci> getAny $ F.foldMap (\x -> Any $ x > 15) testTree
False
```

Все узлы нашего дерева будут содержать значение Any False после того, как к ним будет применена анонимная функция. Но чтобы получить в итоге значение True, реализация функции mappend для типа Any должна принять по крайней мере одно значение True в качестве параметра. Поэтому окончательным результатом будет False, что логично, поскольку ни одно значение в нашем дереве не превышает 15.

Мы также можем легко превратить наше дерево в список, просто используя функцию foldMap с анонимной функцией \x -> [x]. Сначала эта функция проецируется на наше дерево; каждый элемент становится одноэлементным списком. Действие функции mappend, которое имеет место между всеми этими одноэлементными списками, возвращает в результате один список, содержащий все элементы нашего дерева:

```
ghci> F.foldMap (\x -> [x]) testTree
[1,3,6,5,8,9,10]
```

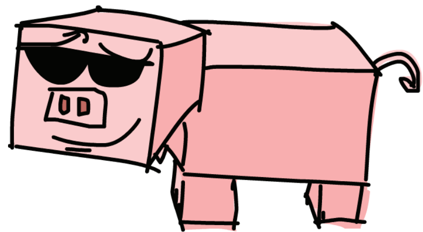
Самое классное, что все эти трюки не ограничиваются деревьями. Они применимы ко всем экземплярам класса Foldable!

13
ПРИГОРШНЯ МОНАД

Когда мы впервые заговорили о функторах в главе 7, вы видели, что они являются полезной концепцией для значений, которые можно отображать. Затем в главе 11 мы развили эту концепцию с помощью аппликативных функторов, которые позволяют нам воспринимать значения определённых типов данных как значения с контекстами и применять к этим значениям обычные функции, сохраняя смысл контекстов.

В этой главе вы узнаете о монадах, которые, по сути, представляют собой расширенные аппликативные функторы, так же как аппликативные функторы являются всего лишь расширенными функторами.

Совершенствуем наши аппликативные функторы



Когда мы начали с функторов, вы видели, что можно отображать разные типы данных с помощью функций, используя класс типов `Functor`. Введение в функторы заставило нас задаться вопросом: «Когда у нас есть функция типа `a -> b` и некоторый тип данных `f a`, как отобразить этот тип данных с помощью функции, чтобы получить значение типа `f b`?» Вы видели, как с по-

мощью чего-либо отобразить Maybe a, список [a], IO a и т. д. Вы даже видели, как с помощью функции типа a -> b отобразить другие функции типа r -> a, чтобы получить функции типа r -> b. Чтобы ответить на вопрос о том, как отобразить некий тип данных с помощью функции, нам достаточно было взглянуть на тип функции fmap:

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

А затем нам необходимо было просто заставить его работать с нашим типом данных, написав соответствующий экземпляр класса Functor.

Потом вы узнали, что возможно усовершенствование функторов, и у вас возникло ещё несколько вопросов. Что если эта функция типа a -> b уже обёрнута в значение функтора? Скажем, у нас есть Just (*3) – как применить это к значению Just 5? Или, может быть, не к Just 5, а к значению Nothing? Или, если у нас есть список [(*2),(+4)], как применить его к списку [1,2,3]? Как это вообще может работать?.. Для этого был введён класс типов Applicative:

```
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

Вы также видели, что можно взять обычное значение и обернуть его в тип данных. Например, мы можем взять значение 1 и обернуть его так, чтобы оно превратилось в Just 1. Или можем превратить его в [1]. Оно могло бы даже стать действием ввода-вывода, которое ничего не делает, а просто выдаёт 1. Функция, которая за это отвечает, называется pure.

Аппликативное значение можно рассматривать как значение с добавленным контекстом – «причудливое» значение, выражаясь техническим языком. Например, буква 'a' – это просто обычная буква, тогда как значение Just 'a' обладает неким добавленным контекстом. Вместо типа Char у нас есть тип Maybe Char, который сообщает нам, что его значением может быть буква; но значением может также быть и отсутствие буквы. Класс типов Applicative позволяет нам использовать с этими значениями, имеющими контекст, обычные функции, и этот контекст сохраняется. Взгляните на пример:

```
ghci> (*) <$> Just 2 <*> Just 8
Just 16
ghci> (++) <$> Just "клингон" <*> Nothing
Nothing
```

```
ghci> (-) <$> [3,4] <*> [1,2,3]
[2,1,0,3,2,1]
```

Поэтому теперь, когда мы рассматриваем их как аппликативные значения, значения типа `Maybe a` представляют вычисления, которые могли окончиться неуспешно, значения типа `[a]` – вычисления, которые содержат несколько результатов (недетерминированные вычисления), значения типа `IO a` – вычисления, которые имеют побочные эффекты, и т. д.

Монады являются естественным продолжением аппликативных функторов и предоставляют решение для следующей проблемы: если у нас есть значение с контекстом типа `m a`, как нам применить к нему функцию, которая принимает обычное значение `a` и возвращает значение с контекстом? Другими словами, как нам применить функцию типа `a -> m b` к значению типа `m a`? По существу, нам нужна вот эта функция:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

Если у нас есть причудливое значение и функция, которая принимает обычное значение, но возвращает причудливое, как нам передать это причудливое значение в данную функцию? Это является основной задачей при работе с монадами. Мы пишем `m a` вместо `f a`, потому что `m` означает `Monad`; но монады являются всего лишь аппликативными функторами, которые поддерживают операцию `>>=`. Функция `>>=` называется *связыванием*.

Когда у нас есть обычное значение типа `a` и обычная функция типа `a -> b`, передать значение функции легче лёгкого: мы применяем функцию к значению как обычно – вот и всё! Но когда мы имеем дело со значениями, находящимися в определённом контексте, нужно немного поразмыслить, чтобы понять, как эти причудливые значения передаются функциям и как учесть их поведение. Впрочем, вы сами убедитесь, что это так же просто, как раз, два, три.

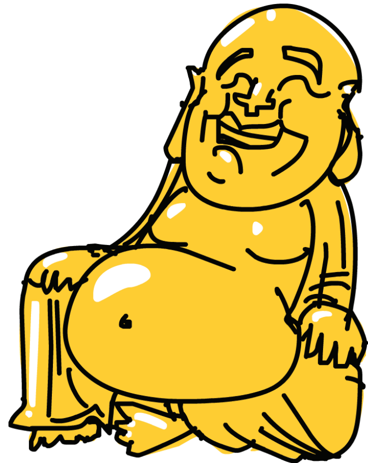
Приступаем к типу Maybe

Теперь, когда у вас появилось хотя бы смутное представление о том, что такое монады, давайте внесём в это представление несколько большую определённость. К великому удивлению, тип `Maybe` являет-

ся монадой. Здесь мы исследуем её чуть лучше, чтобы понять, как она работает в этой роли.

> **ПРИМЕЧАНИЕ.** *Убедитесь, что вы в настоящий момент понимаете, что такое аппликативные функторы (мы обсуждали их в главе 11). Вы должны хорошо разбираться в том, как работают различные экземпляры класса `Applicative` и какие виды вычислений они представляют. Для понимания монад вам понадобится развить уже имеющиеся знания об аппликативных функторах.*

Значение типа `Maybe a` представляет значение типа `a`, но с прикреплённым контекстом возможной неудачи в вычислениях. Значение `Just "дхарма"` означает, что в нём имеется строка `"дхарма"`. Значение `Nothing` представляет отсутствие значения, или, если вы посмотрите на строку как на результат вычисления, это говорит о том, что вычисление завершилось неуспешно.



Когда мы рассматривали тип `Maybe` как функтор, мы видели, что если нам нужно отобразить его с помощью функции, используя метод `fmap`, функция отображала содержимое, если это значение `Just`. В противном случае сохранялось значение `Nothing`, поскольку с помощью функции нечего отображать!

```
ghci> fmap (++"!") (Just "мудрость")
Just "мудрость!"
ghci> fmap (++"!") Nothing
Nothing
```

Тип `Maybe` функционирует в качестве аппликативного функтора аналогично. Однако при использовании аппликативных функторов сама функция находится в контексте наряду со значением, к которому она применяется. Тип `Maybe` является аппликативным функтором таким образом, что когда мы используем операцию `<*>` для применения функции внутри типа `Maybe` к значению, которое находится внутри типа `Maybe`, они оба должны быть значениями

Just, чтобы результатом было значение Just; в противном случае результатом будет значение Nothing. Это имеет смысл. Если недостаёт функции либо значения, к которому вы её применяете, вы не можете ничего получить «из воздуха», поэтому вы должны распространить неудачу.

```
ghci> Just (+3) <*> Just 3
Just 6
ghci> Nothing <*> Just "алчность"
Nothing
ghci> Justord <*> Nothing
Nothing
```

Использование аппликативного стиля, чтобы обычные функции работали со значениями типа Maybe, действует аналогичным образом. Все значения должны быть значениями Just; в противном случае всё это напрасно (Nothing)!

```
ghci> max <$> Just 3 <*> Just 6
Just 6
ghci> max <$> Just 3 <*> Nothing
Nothing
```

А теперь давайте подумаем над тем, как бы мы использовали операцию >>= с типом Maybe. Операция >>= принимает монадическое значение и функцию, которая принимает обычное значение. Она возвращает монадическое значение и умудряется применить эту функцию к монадическому значению. Как она это делает, если функция принимает обычное значение? Ну, она должна принимать во внимание контекст этого монадического значения.

В данном случае операция >>= принимала бы значение типа Maybe a и функцию типа a -> Maybe b и каким-то образом применяла бы эту функцию к значению Maybe a. Чтобы понять, как она это делает, мы будем исходить из того, что тип Maybe является аппликативным функтором. Скажем, у нас есть анонимная функция \x -> Just (x+1). Она принимает число, прибавляет к нему 1 и оборачивает его в конструктор Just:

```
ghci> (\x -> Just (x+1)) 1
Just 2
ghci> (\x -> Just (x+1)) 100
Just 101
```

Если мы передадим ей значение 1, она вернёт результат Just 2. Если мы дадим ей значение 100, результатом будет Just 101. Это выглядит очень просто. Но как нам передать этой функции значение типа Maybe? Если мы подумаем о том, как тип Maybe работает в качестве аппликативного функтора, ответить на этот вопрос будет довольно легко. Мы передаём функции значение Just, берём то, что находится внутри конструктора Just, и применяем к этому функцию. Если мы даём ей значение Nothing, то у нас остаётся функция, но к ней нечего (Nothing) применить. В этом случае давайте сделаем то же, что мы делали и прежде, и скажем, что результат равен Nothing.

Вместо того чтобы назвать функцию >>=, давайте пока назовём её applyMaybe. Она принимает значение типа Maybe a и функцию, которая возвращает значение типа Maybe b, и умудряется применить эту функцию к значению типа Maybe a. Вот она в исходном коде:

```
applyMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
applyMaybe Nothing f = Nothing
applyMaybe (Just x) f = f x
```

Теперь давайте с ней поиграем. Мы будем использовать её как инфиксную функцию так, чтобы значение типа Maybe было слева, а функция была справа:

```
ghci> Just 3 `applyMaybe` \x -> Just (x+1)
Just 4
ghci> Just "смайлик" `applyMaybe` \x -> Just (x ++ " :)")
Just "смайлик :)"
ghci> Nothing `applyMaybe` \x -> Just (x+1)
Nothing
ghci> Nothing `applyMaybe` \x -> Just (x ++ " :)")
Nothing
```

В данном примере, когда мы использовали функцию applyMaybe со значением Just и функцией, функция просто применялась к значению внутри конструктора Just. Когда мы попытались использовать её со значением Nothing, весь результат был равен Nothing. Что насчёт того, если функция возвращает Nothing? Давайте посмотрим:

```
ghci>Just 3 `applyMaybe` \x -> if x > 2 then Just x else Nothing
Just 3
```

```
ghci> Just 1 `applyMaybe` \x -> if x > 2 then Just x else Nothing
Nothing
```

Результаты оказались такими, каких мы и ждали! Если монадическое значение слева равно `Nothing`, то всё будет равно `Nothing`. А если функция справа возвращает значение `Nothing`, результатом опять будет `Nothing`. Это очень похоже на тот случай, когда мы использовали тип `Maybe` в качестве аппликативного функтора и в результате получали значение `Nothing`, если где-то в составе присутствовало значение `Nothing`.

Похоже, мы догадались, как взять причудливое значение, передать его функции, которая принимает обычное значение, и вернуть причудливое значение. Мы сделали это, помня, что значение типа `Maybe` представляет вычисление, которое могло окончиться неуспешно.

Вы можете спросить себя: «Чем это полезно?» Может показаться, что аппликативные функторы сильнее монад, поскольку аппликативные функторы позволяют нам взять обычную функцию и заставить её работать со значениями, имеющими контекст. В этой главе вы увидите, что монады, будучи усовершенствованными аппликативными функторами, тоже способны на такое. На самом деле они могут делать и кое-какие другие крутые вещи, на которые не способны аппликативные функторы.

Мы вернёмся к `Maybe` через минуту, но сначала давайте взглянем на класс типов, который относится к монадам.

Класс типов Monad

Как и функторы, у которых есть класс типов `Functor`, и аппликативные функторы, у которых есть класс типов `Applicative`, монады обладают своим классом типов: `Monad`! (Ух ты, кто бы мог подумать?)

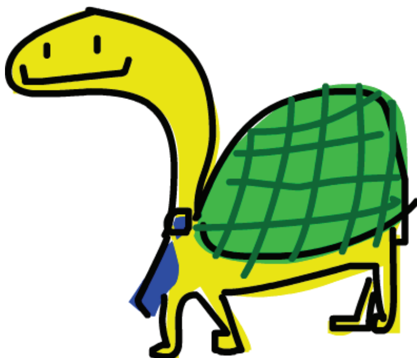
```
class Monad m where
    return :: a -> m a

    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg
```

В первой строке говорится `class Monad m where`. Стойте, не говорил ли я, что монады являются просто расширенными аппликативными функторами? Не надлежит ли здесь быть ограничению класса наподобие `class (Applicative m) => Monad m where`, чтобы тип должен был являться аппликативным функтором, прежде чем он может быть сделан монадой? Ладно, положим, надлежит, – но когда появился язык Haskell, людям не пришло в голову, что аппликативные функторы хорошо подходят для этого языка. Тем не менее будьте уверены: каждая монада является аппликативным функтором, даже если в объявлении класса `Monad` этого не говорится.

Первой функцией, которая объявлена в классе типов `Monad`, является `return`. Она аналогична функции `pure`, находящейся в классе типов `Applicative`. Так что, хоть она и называется по-другому, вы уже фактически с ней знакомы. Функция `return` имеет тип `(Monad m) => a -> m a`. Она принимает значение и помещает его в минимальный контекст по умолчанию, который по-прежнему содержит это значение. Другими словами, она принимает нечто и оборачивает это в монаду. Мы уже использовали функцию `return` при обработке действий ввода-вывода (см. главу 8). Там она понадобилась для получения значения и создания фальшивого действия ввода-вывода, которое ничего не делает, а только возвращает это значение. В случае с типом `Maybe` она принимает значение и оборачивает его в конструктор `Just`.

> **ПРИМЕЧАНИЕ.** *Функция `return` ничем не похожа на оператор `return` из других языков программирования, таких как C++ или Java. Она не завершает выполнение функции. Она просто принимает обычное значение и помещает его в контекст.*

Следующей функцией является `>>=`, или связывание. Она похожа на применение функции, но вместо того, чтобы получать



обычное значение и передавать его обычной функции, она принимает монадическое значение (то есть значение с контекстом) и передаёт его функции, которая принимает обычное значение, но возвращает монадическое.

Затем у нас есть операция `>>`. Мы пока не будем обращать на неё большого внимания, потому что она идёт в реализации по умолчанию, и её редко реализуют при создании экземпляров класса `Monad`. Мы подробно рассмотрим её в разделе «Банан на канате».

Последним методом в классе типов `Monad` является функция `fail`. Мы никогда не используем её в нашем коде явно. Вместо этого её использует язык Haskell, чтобы сделать возможным неуспешное окончание вычислений в специальной синтаксической конструкции для монад, с которой вы встретитесь позже. Нам не нужно сейчас сильно беспокоиться по поводу этой функции.

Теперь, когда вы знаете, как выглядит класс типов `Monad`, давайте посмотрим, каким образом для типа `Maybe` реализован экземпляр этого класса!

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f = f x
    fail _ = Nothing
```

Функция `return` аналогична функции `pure`, так что для работы с ней не нужно большого ума. Мы делаем то же, что мы делали в классе типов `Applicative`, и оборачиваем в конструктор `Just`. Операция `>>=` аналогична нашей функции `applyMaybe`. Когда мы передаём значение типа `Maybe a` нашей функции, то запоминаем контекст и возвращаем значение `Nothing`, если значением слева является `Nothing`. Ещё раз: если значение отсутствует, нет способа применить к нему функцию. Если это значение `Just`, мы берём то, что находится внутри, и применяем к этому функцию `f`.

Мы можем поиграть с типом Maybe как с монадой:

```
ghci> return "ЧТО" :: Maybe String
Just "ЧТО"
ghci> Just 9 >>= \x -> return (x*10)
Just 90
ghci> Nothing >>= \x -> return (x*10)
Nothing
```

В первой строке нет ничего нового или захватывающего, поскольку мы уже использовали функцию pure с типом Maybe, и мы знаем, что функция return – это просто функция pure под другим именем.

Следующая пара строк демонстрирует операцию >>= уже поинтереснее. Обратите внимание: когда мы передавали значение Just 9 анонимной функции \x -> return (x*10), то параметр x принимал значение 9 внутри функции. Выглядит это так, будто мы могли извлечь значение из обёртки Maybe без сопоставления с образцом. И мы всё ещё не потеряли контекст нашего значения Maybe, потому что когда оно равно Nothing, результатом использования операции >>= тоже будет Nothing.

Прогулка по канату

Теперь, когда вы знаете, как передавать значение типа Maybe a функции типа a -> Maybe b, учитывая контекст возможной неудачи в вычислениях, давайте посмотрим, как можно многократно использовать операцию >>= для обработки вычислений нескольких значений Maybe a.

Пьер решил сделать рабочий перерыв на рыбной ферме и попробовать заняться канатоходством. На удивление, ему это неплохо удаётся, но есть одна проблема: на балансировочный шест приземляются птицы! Они прилетают, немного отдыхают, болтают со своими пернатыми друзьями, а затем срываются в по-

исках хлебных крошек. Это не сильно беспокоило бы Пьера, будь количество птиц с левой стороны шеста всегда равным количеству птиц с правой стороны. Но порой всем птицам почему-то больше нравится одна сторона. В результате канатоходец теряет равновесие и падает (не волнуйтесь, он использует сетку безопасности!).

Давайте предположим, что Пьер удержит равновесие, если количество птиц на левой стороне шеста и на правой стороне шеста разнится в пределах трёх. Покуда, скажем, на правой стороне одна птица, а на левой – четыре, всё в порядке. Но стоит пятой птице опуститься на левую сторону, канатоходец теряет равновесие и кубарем летит вниз.

Мы сымитируем посадку и улёт птиц с шеста и посмотрим, останется ли Пьер на канате после некоторого количества прилётов и улётов птиц. Например, нам нужно увидеть, что произойдёт с Пьером, если первая птица прилетит на левую сторону, затем четыре птицы займут правую, а потом птица, которая была на левой стороне, решит улететь.

Код, код, код

Мы можем представить шест в виде простой пары целых чисел. Первый компонент будет обозначать количество птиц на левой стороне, а второй – количество птиц на правой:

```
type Birds = Int
type Pole = (Birds, Birds)
```

Сначала мы создали синоним типа для `Int`, названный `Birds`, потому что мы используем целые числа для представления количества имеющихся птиц. Затем создали синоним типа `(Birds, Birds)` и назвали его `Pole` (учтите: это означает «шест» – ничего общего ни с поляками, ни с человеком по имени Поль).

А теперь как насчёт того, чтобы добавить функции, которые принимают количество птиц и производят их приземление на одной стороне шеста или на другой?

```
landLeft :: Birds -> Pole -> Pole
landLeft n (left, right) = (left + n, right)

landRight :: Birds -> Pole -> Pole
landRight n (left, right) = (left, right + n)
```

Давайте проверим их:

```
ghci> landLeft 2 (0, 0)
(2,0)
ghci> landRight 1 (1, 2)
(1,3)
ghci> landRight (-1) (1,2)
(1,1)
```

Чтобы заставить птиц улететь, мы просто произвели приземление отрицательного количества птиц на одной стороне. Поскольку приземление птицы на `Pole` возвращает `Pole`, мы можем сцепить применения функций `landLeft` и `landRight`:

```
ghci> landLeft 2 (landRight 1 (landLeft 1 (0, 0)))
(3,1)
```

Когда мы применяем функцию `landLeft 1` к значению `(0, 0)`, у нас получается результат `(1, 0)`. Затем мы усаживаем птицу на правой стороне, что даёт в результате `(1, 1)`. Наконец, две птицы приземляются на левой стороне, что даёт в результате `(3, 1)`. Мы применяем функцию к чему-либо, сначала записывая функцию, а затем её параметр, но здесь было бы лучше, если бы первым шел шест, а потом функция посадки. Предположим, мы создали вот такую функцию:

```
x -: f = f x
```

Можно применять функции, сначала записывая параметр, а затем функцию:

```
ghci> 100 -: (*3)
300
ghci> True -: not
False
ghci> (0, 0) -: landLeft 2
(2,0)
```

Используя эту форму, мы можем многократно производить приземление птиц на шест в более «читабельном» виде:

```
ghci> (0, 0) -: landLeft 1 -: landRight 1 -: landLeft 2
(3,1)
```

Круто!.. Эта версия эквивалентна предыдущей, где мы многократно усаживали птиц на шест, но выглядит она яснее. Здесь очевиднее, что мы начинаем с (0, 0), а затем усаживаем одну птицу слева, потом одну – справа, и в довершение две – слева.

Я улечу

Пока всё идёт нормально, но что произойдёт, если десять птиц приземлятся на одной стороне?

```
ghci> landLeft 10 (0, 3)
(10,3)
```

Десять птиц с левой стороны и лишь три с правой?! Этого достаточно, чтобы отправить в полёт самого Пьера!.. Довольно очевидная вещь. Но что если бы у нас была примерно такая последовательность посадок:

```
ghci> (0, 0) -: landLeft 1 -: landRight 4 -: landLeft (-1) -: landRight (-2)
(0,2)
```

Может показаться, что всё хорошо, но если вы проследите за шагами, то увидите, что на правой стороне одновременно находятся четыре птицы – а на левой ни одной! Чтобы исправить это, мы должны ещё раз взглянуть на наши функции `landLeft` и `landRight`.

Необходимо дать функциям `landLeft` и `landRight` возможность завершаться неуспешно. Нам нужно, чтобы они возвращали новый шест, если равновесие поддерживается, но завершались неуспешно, если птицы приземляются неравномерно. И какой способ лучше подойдёт для добавления к значению контекста неудачи, чем использование типа `Maybe`? Давайте переработаем эти функции:

```
landLeft :: Birds -> Pole -> Maybe Pole
landLeft n (left,right)
    | abs ((left + n) - right) < 4 = Just (left + n, right)
    | otherwise                    = Nothing

landRight :: Birds -> Pole -> Maybe Pole
landRight n (left,right)
    | abs (left - (right + n)) < 4 = Just (left, right + n)
    | otherwise                    = Nothing
```

Вместо того чтобы вернуть значение типа `Pole`, эти функции теперь возвращают значения типа `Maybe Pole`. Они по-прежнему принимают количество птиц и прежний шест, как и ранее, но затем проверяют, выведет ли Пьера из равновесия приземление такого количества птиц. Мы используем охранные выражения, чтобы проверить, меньше ли разница в количестве птиц на новом шесте, чем 4. Если меньше, оборачиваем новый шест в конструктор `Just` и возвращаем это. Если не меньше, возвращаем значение `Nothing`, сигнализируя о неудаче.

Давайте опробуем этих деток:

```
ghci> landLeft 2 (0, 0)
Just (2,0)
ghci> landLeft 10 (0, 3)
Nothing
```

Когда мы приземляем птиц, не выводя Пьера из равновесия, мы получаем новый шест, обёрнутый в конструктор `Just`. Но когда значительное количество птиц в итоге оказывается на одной стороне шеста, в результате мы получаем значение `Nothing`. Всё это здорово, но, похоже, мы потеряли возможность многократного приземления птиц на шесте! Выполнить `landLeft 1 (landRight 1 (0, 0))` больше нельзя, потому что когда `landRight 1` применяется к `(0, 0)`, мы получаем значение не типа `Pole`, а типа `Maybe Pole`. Функция `landLeft 1` принимает параметр типа `Pole`, а не `Maybe Pole`.

Нам нужен способ получения `Maybe Pole` и передачи его функции, которая принимает `Pole` и возвращает `Maybe Pole`. К счастью, у нас есть операция `>>=`, которая делает именно это для типа `Maybe`. Давайте попробуем:

```
ghci> landRight 1 (0, 0) >>= landLeft 2
Just (2,1)
```

Вспомните, что функция `landLeft 2` имеет тип `Pole -> Maybe Pole`. Мы не можем просто передать ей значение типа `Maybe Pole`, которое является результатом вызова функции `landRight 1 (0, 0)`, поэтому используем операцию `>>=`, чтобы взять это значение с контекстом и отдать его функции `landLeft 2`. Операция `>>=` действительно позволяет нам обрабатывать значения типа `Maybe` как значения с контекстом. Если мы передадим значение `Nothing` в функцию `landLeft 2`, результатом будет `Nothing`, и неудача будет распространена:

```
ghci> Nothing >>= landLeft 2
Nothing
```

Используя это, мы теперь можем помещать в цепочку приземления, которые могут окончиться неуспешно, потому что оператор >>= позволяет нам передавать монадическое значение функции, которая принимает обычное значение. Вот последовательность приземлений птиц:

```
ghci> return (0, 0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
Just (2,4)
```

Вначале мы использовали функцию `return`, чтобы взять шест и обернуть его в конструктор `Just`. Мы могли бы просто применить выражение `landRight 2` к значению `(0, 0)` – это было бы то же самое, – но так можно добиться большего единообразия, используя оператор `>>=` для каждой функции. Выражение `Just (0, 0)` передаётся в функцию `landRight 2`, что в результате даёт результат `Just (0, 2)`. Это значение в свою очередь передаётся в функцию `landLeft 2`, что в результате даёт новый результат `(2, 2)`, и т. д.

Помните следующий пример, прежде чем мы ввели возможность неудачи в инструкции Пьера?

```
ghci> (0, 0) -: landLeft 1 -: landRight 4 -: landLeft (-1) -: landRight (-2)
(0,2)
```

Он не очень хорошо симулировал взаимодействие канатоходца с птицами. В середине его равновесие было нарушено, но результат этого не отразил. Давайте теперь исправим это, используя монадическое применение (оператор `>>=`) вместо обычного:

```
ghci> return (0, 0) >>= landLeft 1 >>= landRight 4 >>= landLeft (-1) >>= landRight (-2)
Nothing
```

Окончательный результат представляет неудачу, чего мы и ожидали. Давайте посмотрим, как этот результат был получен:

1. Функция `return` помещает значение `(0, 0)` в контекст по умолчанию, превращая значение в `Just (0, 0)`.
2. Происходит вызов выражения `Just (0, 0) >>= landLeft 1`. Поскольку значение `Just (0, 0)` является значением `Just`, функция `landLeft 1` применяется к `(0, 0)`, что в результате даёт

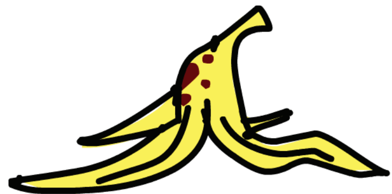
результат `Just (1, 0)`, потому что птицы всё ещё находятся в относительном равновесии.
3. Имеет место вызов выражения `Just (1, 0) >>= landRight 4`, и результатом является выражение `Just (1, 4)`, поскольку равновесие птиц пока ещё не затронуто, хотя Пьер уже удерживается с трудом.
4. Выражение `Just (1, 4)` передаётся в функцию `landLeft (-1)`. Это означает, что имеет место вызов `landLeft (-1) (1, 4)`. Теперь ввиду особенностей работы функции `landLeft` в результате это даёт значение `Nothing`, так как результирующий шест вышел из равновесия.
5. Теперь, поскольку у нас есть значение `Nothing`, оно передаётся в функцию `landRight (-2)`, но так как это `Nothing`, результатом автоматически становится `Nothing`, поскольку нам не к чему применить эту функцию.

Мы не смогли бы достигнуть этого, просто используя `Maybe` в качестве аппликативного функтора. Если вы попробуете так сделать, то застрянете, поскольку аппликативные функторы не очень-то позволяют аппликативным значениям взаимодействовать друг с другом. Их в лучшем случае можно использовать как параметры для функции, применяя аппликативный стиль.

Аппликативные операторы извлекут свои результаты и передадут их функции в соответствующем для каждого аппликативного функтора виде, а затем соберут окончательное аппликативное значение, но взаимодействие между ними не особенно заметно. Здесь, однако, каждый шаг зависит от результата предыдущего шага. Во время каждого приземления возможный результат предыдущего шага исследуется, а шест проверяется на равновесие. Это определяет, окончится ли посадка успешно либо неуспешно.

Банан на канате

Давайте разработаем функцию, которая игнорирует текущее количество птиц на балансировочном шесте и просто заставляет Пьера поскользнуться и упасть. Мы назовём её `banana`:



```
banana :: Pole -> Maybe Pole
banana _ = Nothing
```

Мы можем поместить эту функцию в цепочку вместе с нашими приземлениями птиц. Она всегда будет вызывать падение канатоходца, поскольку игнорирует всё, что ей передаётся в качестве параметра, и неизменно возвращает неудачу.

```
ghci> return (0, 0) >>= landLeft 1 >>= banana >>= landRight 1
Nothing
```

Функции `banana` передаётся значение `Just (1, 0)`, но она всегда производит значение `Nothing`, которое заставляет всё выражение возвращать в результате `Nothing`. Какая досада!..

Вместо создания функций, которые игнорируют свои входные данные и просто возвращают предопределённое монадическое значение, мы можем использовать функцию `>>`. Вот её реализация по умолчанию:

```
(>>) :: (Monad m) => m a -> m b -> m b
m >> n = m >>= \_ -> n
```

Обычно передача какого-либо значения функции, которая игнорирует свой параметр и всегда возвращает некое предопределённое значение, всегда даёт в результате это предопределённое значение. При использовании монад, однако, нужно принимать во внимание их контекст и значение. Вот как функция `>>` действует при использовании с типом `Maybe`:

```
ghci> Nothing >> Just 3
Nothing
ghci> Just 3 >> Just 4
Just 4
ghci> Just 3 >> Nothing
Nothing
```

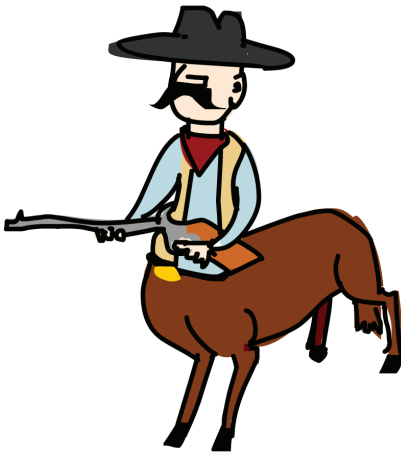
Если мы заменим оператор >> на вызов `>>= _ ->`, легко увидеть, что происходит.

Мы можем заменить нашу функцию `banana` в цепочке на оператор `>>` и следующее за ним значение `Nothing`, чтобы получить гарантированную и очевидную неудачу:

```
ghci> return (0, 0) >>= landLeft 1 >> Nothing >>= landRight 1
Nothing
```

Как бы это выглядело, если бы мы не сделали разумный выбор, обработав значения типа `Maybe` как значения с контекстом неудачи и передав их функциям? Вот какой была бы последовательность приземлений птиц:

```
routine :: Maybe Pole
routine = case landLeft 1 (0, 0) of
   Nothing -> Nothing
   Just pole1 -> case landRight 4 pole1 of
      Nothing -> Nothing
      Just pole2 -> case landLeft 2 pole2 of
         Nothing -> Nothing
         Just pole3 -> landLeft 1 pole3
```



Мы усаживаем птицу слева, а затем проверяем вероятность неудачи и вероятность успеха. В случае неудачи мы возвращаем значение `Nothing`. В случае успеха усаживаем птиц справа, а затем повторяем всё сызнова. Превращение этого убожества в симпатичную цепочку монадических применений с использованием функции `>>=` является классическим примером того, как монада `Maybe` экономит массу времени, когда вам необходимо последовательно выполнить вычисления, основанные на вычислениях, которые могли окончиться неуспешно.

Обратите внимание, каким образом реализация операции `>>=` для типа `Maybe` отражает именно эту логику, когда проверяется, равно ли значение `Nothing`, и действие производится на основе этих сведений. Если значение равно `Nothing`, она незамедлительно возвращает результат `Nothing`. Если значение не равно `Nothing`, она продолжает работу с тем, что находится внутри конструктора `Just`.

В этом разделе мы рассмотрели, как некоторые функции работают лучше, когда возвращаемые ими значения поддерживают неудачу. Превращая эти значения в значения типа `Maybe` и заменяя обычное применение функций вызовом операции `>>=`, мы практически даром получили механизм обработки вычислений, которые могут оканчиваться неудачно. Причина в том, что операция `>>=` должна сохранять контекст значения, к которому она применяет функции. В данном случае контекстом являлось то, что наши значения были значениями с неуспехом в вычислениях. Поэтому когда мы применяли к таким значениям функции, всегда учитывалась вероятность неуспеха.

Нотация do

Монады в языке Haskell настолько полезны, что они обзавелись своим собственным синтаксисом, который называется «*нотация do*». Вы уже познакомились с нотацией `do` в главе 8, когда мы использовали её для объединения нескольких действий ввода-вывода. Как оказывается, нотация `do` предназначена не только для системы ввода-вывода, но может использоваться для любой монады. Её принцип остаётся прежним: последовательное «склеивание» монадических значений.

Рассмотрим этот знакомый пример монадического применения:

```
ghci> Just 3 >>= (\x -> Just (show x ++ "!"))
Just "3!"
```

Это мы уже проходили! Передача монадического значения функции, которая возвращает монадическое значение, – ничего особенного. Заметьте, как параметр `x` становится равным значению `3` внутри анонимной функции, когда мы выполняем код. Как только мы внутри этой анонимной функции, это просто обычное значение, а не монадическое. А что если бы у нас был ещё один вызов оператора `>>=` внутри этой функции? Посмотрите:

```
ghci> Just 3 >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Just "3!"
```

Ага-а, вложенное использование операции `>>=`! Во внешней анонимной функции мы передаём значение `Just "!"` анонимной

функции `\y -> Just (show x ++ y)`. Внутри этой анонимной функции параметр `y` становится равным `"!"`. Параметр `x` по-прежнему равен `3`, потому что мы получили его из внешней анонимной функции. Всё это как будто напоминает мне о следующем выражении:

```
ghci> let x = 3; y = "!" in show x ++ y
"3!"
```

Главное отличие состоит в том, что значения в нашем примере с использованием оператора `>>=` являются монадическими. Это значения с контекстом неудачи. Мы можем заменить любое из них на неудачу:

```
ghci> Nothing >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Nothing
ghci> Just 3 >>= (\x -> Nothing >>= (\y -> Just (show x ++ y)))
Nothing
ghci> Just 3 >>= (\x -> Just "!" >>= (\y -> Nothing))
Nothing
```

В первой строке передача значения `Nothing` функции естественным образом даёт в результате `Nothing`. Во второй строке мы передаём значение `Just 3` функции, и параметр `x` становится равным `3`. Но потом мы передаём значение `Nothing` внутренней анонимной функции, и результатом становится `Nothing`, что заставляет внешнюю анонимную функцию тоже произвести `Nothing` в качестве своего результата. Это что-то вроде присвоения значений переменным в выражениях `let`, только значения, о которых идёт речь, являются монадическими.

Чтобы проиллюстрировать эту идею, давайте запишем следующие строки в сценарий так, чтобы каждое значение типа `Maybe` занимало свою собственную строку:

```
foo :: Maybe String
foo = Just 3 >>= (\x ->
      Just "!">>= (\y ->
      Just (show x ++ y)))
```

Чтобы уберечь нас от написания всех этих раздражающих анонимных функций, язык Haskell предоставляет нам нотацию `do`. Она позволяет нам записать предыдущий кусок кода вот так:

```
foo :: Maybe String
foo = do
    x <- Just 3
    y <- Just "!"
    Just (show x ++ y)
```

Могло показаться, что мы получили возможность временно извлекать сущности из значений типа `Maybe` без необходимости проверять на каждом шагу, являются ли значения типа `Maybe` значениями в конструкторе `Just` или значениями `Nothing`. Вот классно!.. Если какое-либо из значений, которые мы пытаемся извлечь, равно `Nothing`, всё выражение `do` в результате вернёт значение `Nothing`. Мы выдёргиваем наружу их значения (если таковые существуют) и перекладываем необходимость беспокойства о контексте, идущем с этими значениями, на плечи оператора `>>=`.

Выражения `do` – это просто другой синтаксис для сцепления монадических значений.

Делай как я

В выражении `do` каждая строка, не являющаяся строкой `let`, является монадическим значением. Чтобы просмотреть её результат, мы используем символ `<-`. Если у нас есть значение типа `Maybe String` и мы привязываем её к образцу с помощью символа `<-`, этот образец будет иметь тип `String` так же, как когда мы использовали операцию `>>=` для передачи монадических значений анонимным функциям.

Последнее монадическое значение в выражении `do` – такое как `Just (show x ++ y)` в этом примере – не может быть использовано с символом `<-` для привязки его результата, потому что если бы мы преобразовали выражение `do` обратно в цепочку применений оператора `>>=`, это не имело бы смысла. Наоборот, результат последнего выражения является результатом всего склеенного монадического значения, учитывая возможную неудачу вычисления каждого

из предыдущих монадических значений. Рассмотрите, например, следующую строку:

```
ghci> Just 9 >>= (\x -> Just (x > 8))
Just True
```

Поскольку левым параметром функции `>>=` является значение в конструкторе `Just`, анонимная функция применяется к значению `9`, и результатом становится значение `Just True`. Мы можем переписать это в нотации `do` следующим образом:

```
marySue :: Maybe Bool
marySue = do
   x <- Just 9
   Just (x > 8)
```

Сравнивая оба варианта, легко увидеть, почему результатом всего монадического значения является результат последнего монадического значения в выражении `do` со всеми предыдущими монадическими значениями, сцепленными с ним.

Пьер возвращается

Инструкция нашего канатоходца может также быть выражена с использованием нотации `do`. Функции `landLeft` и `landRight` принимают количество птиц и шест и производят шест, обёрнутый в `Just`. Исключение – это когда канатоходец соскальзывает, и тогда возвращается значение `Nothing`. Мы использовали операцию `>>=` для сцепления последовательных шагов, потому что каждый из них зависел от предыдущего и каждый обладал добавленным контекстом возможной неудачи. Здесь две птицы приземляются с левой стороны, затем две птицы – с правой, а потом одна птица – снова с левой:

```
routine :: Maybe Pole
routine = do
   start <- return (0, 0)
   first <- landLeft 2 start
   second <- landRight 2 first
   landLeft 1 second
```

Давайте посмотрим, окончится ли это удачно для Пьера:

```
ghci> routine
Just (3,2)
```

Окончилось удачно!

Когда мы выполняли эти инструкции, явно записывая вызовы оператора `>>=`, мы обычно писали что-то вроде `return (0, 0) >>= landLeft 2`, потому что функция `landLeft` является функцией, которая возвращает значение типа `Maybe`. Однако при использовании выражения `do` каждая строка должна представлять монадическое значение. Поэтому мы явно передаём предыдущее значение типа `Pole` функциям `landLeft` и `landRight`. Если бы мы проверили образцы, к которым привязали наши значения типа `Maybe`, то `start` был бы равен `(0, 0)`, `first` был бы равен `(2, 0)` и т. д.

Поскольку выражения `do` записываются построчно, некоторым людям они могут показаться императивным кодом. Но эти выражения просто находятся в последовательности, поскольку каждое значение в каждой строке зависит от результатов выражений в предыдущих строках вместе с их контекстами (в данном случае контекстом является успешное либо неуспешное окончание их вычислений).

Ещё раз давайте взглянем на то, как выглядел бы этот кусок кода, если бы мы не использовали монадические стороны типа `Maybe`:

```
routine :: Maybe Pole
routine =
   case Just (0, 0) of
      Nothing -> Nothing
      Just start -> case landLeft 2 start of
         Nothing -> Nothing
         Just first -> case landRight 2 first of
            Nothing -> Nothing
            Just second -> landLeft 1 second
```

Видите, как в случае успеха образец `start` получает значение кортежа внутри `Just (0, 0)`, образец `first` получает значение результата выполнения `landLeft 2 start` и т. д.?

Если мы хотим бросить Пьеру банановую кожуру в нотации `do`, можем сделать следующее:

```
routine :: Maybe Pole
routine = do
   start <- return (0, 0)
   first <- landLeft 2 start
   Nothing
   second <- landRight 2 first
   landLeft 1 second
```

Когда мы записываем в нотации do строку, не связывая монадическое значение с помощью символа <-, это похоже на помещение вызова функции >> за монадическим значением, результат которого мы хотим игнорировать. Мы помещаем монадическое значение в последовательность, но игнорируем его результат, так как нам неважно, чем он является. Плюс ко всему это красивее, чем записывать эквивалентную форму _ <- Nothing.

Когда использовать нотацию do, а когда явно использовать вызов операции >>=, зависит от вас. Я думаю, этот пример хорошо подходит для того, чтобы явно использовать операцию >>=, потому что каждый шаг прямо зависит от предыдущего. При использовании нотации do мы должны явно записывать, на каком шесте садятся птицы, но каждый раз мы просто используем шест, который был результатом предшествующего приземления. Тем не менее это дало нам некоторое представление о нотации do.

Сопоставление с образцом и неудача в вычислениях

Привязывая монадические значения к идентификаторам в нотации do, мы можем использовать сопоставление с образцом так же, как в выражениях let и параметрах функции. Вот пример сопоставления с образцом в выражении do:

```
justFirst :: Maybe Char
justFirst = do
   (x:xs) <- Just "привет"
   return x
```

Мы используем сопоставление с образцом для получения первого символа строки "привет", а затем возвращаем его в качестве результата. Поэтому justFirst возвращает значение Just 'п'.

Что если бы это сопоставление с образцом окончилось неуспешно? Когда сопоставление с образцом в функции оканчивается неуспешно, происходит сопоставление со следующим образцом. Если сопоставление проходит по всем образцам для данной функции с невыполнением их условий, выдаётся ошибка и происходит аварийное завершение работы программы. С другой стороны, сопоставление с образцом, окончившееся неудачей в выражениях let, приводит к незамедлительному возникновению ошибки, по-

тому что в выражениях let отсутствует механизм прохода к следующему образцу при невыполнении условия.

Когда сопоставление с образцом в выражении do завершается неуспешно, функция fail (являющаяся частью класса типов Monad) позволяет ему вернуть в результате неудачу в контексте текущей монады, вместо того чтобы привести к аварийному завершению работы программы. Вот реализация функции по умолчанию:

```
fail :: (Monad m) => String -> m a
fail msg = error msg
```

Так что по умолчанию она действительно заставляет программу завершаться аварийно. Но монады, содержащие в себе контекст возможной неудачи (как тип Maybe), обычно реализуют её самостоятельно. Для типа Maybe она реализована следующим образом:

```
fail _ = Nothing
```

Она игнорирует текст сообщения об ошибке и производит значение Nothing. Поэтому, когда сопоставление с образцом оканчивается неуспешно в значении типа Maybe, записанном в нотации do, результат всего значения будет равен Nothing. Предпочтительнее, чтобы ваша программа завершила свою работу неаварийно. Вот выражение do, включающее сопоставление с образцом, которое обречено на неудачу:

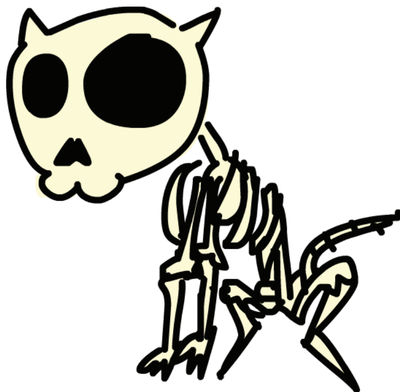
```
wopwop :: Maybe Char
wopwop = do
   (x:xs) <- Just ""
   return x
```

Сопоставление с образцом оканчивается неуспешно, поэтому эффект аналогичен тому, как если бы вся строка с образцом была заменена значением Nothing. Давайте попробуем это:

```
ghci> wopwop
Nothing
```

Неуспешно окончившееся сопоставление с образцом вызвало неуспех только в контексте нашей монады, вместо того чтобы вызвать неуспех на уровне всей программы. Очень мило!..

Списковая монада



До сих пор вы видели, как значения типа `Maybe` могут рассматриваться в качестве значений с контекстом неудачи, и как мы можем ввести в код обработку неуспешно оканчивающихся вычислений, используя оператор `>>=` для передачи их функциям. В этом разделе мы посмотрим, как использовать монадическую сторону списков, чтобы внести в код недетерминированность в ясном и «читабельном» виде.

В главе 11 мы говорили о том, каким образом списки представляют недетерминированные значения, когда они используются как аппликативные функторы. Значение вроде 5 является детерминированным – оно имеет только один результат, и мы точно знаем, какой он. С другой стороны, значение вроде `[3,8,9]` содержит несколько результатов, поэтому мы можем рассматривать его как одно значение, которое в то же время, по сути, является множеством значений. Использование списков в качестве аппликативных функторов хорошо демонстрирует эту недетерминированность:

```
ghci> (*) <$> [1,2,3] <*> [10,100,1000]
[10,100,1000,20,200,2000,30,300,3000]
```

В окончательный список включаются все возможные комбинации умножения элементов из левого списка на элементы правого. Когда дело касается недетерминированности, у нас есть много вариантов выбора, поэтому мы просто пробуем их все. Это означает, что результатом тоже является недетерминированное значение, но оно содержит намного больше результатов.

Этот контекст недетерминированности очень красиво переводится в монады. Вот как выглядит экземпляр класса `Monad` для списков:

```
instance Monad [] where
   return x = [x]
   xs >>= f = concat (map f xs)
   fail _ = []
```

Как вы знаете, функция `return` делает то же, что и функция `pure`, и вы уже знакомы с функцией `return` для списков. Она принимает значение и помещает его в минимальный контекст по умолчанию, который по-прежнему возвращает это значение. Другими словами, функция `return` создаёт список, который содержит только одно это значение в качестве своего результата. Это полезно, когда нам нужно просто обернуть обычное значение в список, чтобы оно могло взаимодействовать с недетерминированными значениями.

Суть операции `>>=` состоит в получении значения с контекстом (монадического значения) и передаче его функции, которая принимает обычное значение и возвращает значение, обладающее контекстом. Если бы эта функция просто возвращала обычное значение вместо значения с контекстом, то операция `>>=` не была бы столь полезна: после первого применения контекст был бы утрачен.

Давайте попробуем передать функции недетерминированное значение:

```
ghci> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
```

Когда мы использовали операцию `>>=` со значениями типа `Maybe`, монадическое значение передавалось в функцию с заботой о возможных неудачах. Здесь она заботится за нас о недетерминированности.

Список `[3,4,5]` является недетерминированным значением, и мы передаём его в функцию, которая тоже возвращает недетерминированное значение. Результат также является недетерминированным, и он представляет все возможные результаты получения элементов из списка `[3,4,5]` и передачи их функции `\x -> [x,-x]`. Эта функция принимает число и производит два результата: один взятый со знаком минус и один неизменный. Поэтому когда мы используем операцию `>>=` для передачи этого списка функции, каждое число берётся с отрицательным знаком, а также сохраняется неизменным. Образец `x` в анонимной функции принимает каждое значение из списка, который ей передаётся.

Чтобы увидеть, как это достигается, мы можем просто проследить за выполнением. Сначала у нас есть список `[3,4,5]`. Потом мы

отображаем его с помощью анонимной функции и получаем следующий результат:

```
[[3,-3],[4,-4],[5,-5]]
```

Анонимная функция применяется к каждому элементу, и мы получаем список списков. В итоге мы просто сглаживаем список – и вуаля, мы применили недетерминированную функцию к недетерминированному значению!

Недетерминированность также включает поддержку неуспешных вычислений. Пустой список в значительной степени эквивалентен значению `Nothing`, потому что он означает отсутствие результата. Вот почему неуспешное окончание вычислений определено просто как пустой список. Сообщение об ошибке отбрасывается. Давайте поиграем со списками, которые приводят к неуспеху в вычислениях:

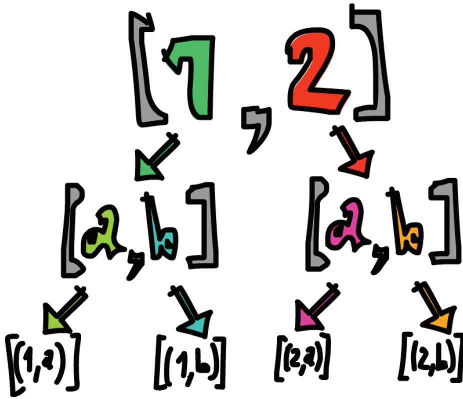
```
ghci> [] >>= \x -> ["плохой","бешеный","крутой"]
[]
ghci> [1,2,3] >>= \x -> []
[]
```

В первой строке пустой список передаётся анонимной функции. Поскольку список не содержит элементов, нет элементов для передачи функции, а следовательно, результатом является пустой список. Это аналогично передаче значения `Nothing` функции, которая принимает тип `Maybe`. Во второй строке каждый элемент передаётся функции, но элемент игнорируется, и функция просто возвращает пустой список. Поскольку функция завершается неуспехом для каждого элемента, который в неё попадает, результатом также является неуспех.

Как и в случае со значениями типа `Maybe`, мы можем сцеплять несколько списков с помощью операции `>>=`, распространяя недетерминированность:

```
ghci> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Числа из списка `[1,2]` связываются с образцом `n`; символы из списка `['a','b']` связываются с образцом `ch`. Затем мы выполняем выражение `return (n, ch)` (или `[(n, ch)]`), что означает получение



пары `(n, ch)` и помещение её в минимальный контекст по умолчанию. В данном случае это создание наименьшего возможного списка, который по-прежнему представляет пару `(n, ch)` в качестве результата и обладает наименее возможной недетерминированностью. Его влияние на контекст минимально. Мы говорим: «Для каждого элемента в списке `[1,2]` обойти каждый элемент из `['a','b']` и произвести кортеж, содержащий по одному элементу из каждого списка».

Вообще говоря, поскольку функция `return` принимает значение и оборачивает его в минимальный контекст, она не обладает какими-то дополнительными эффектами (вроде приведения к неуспешному окончанию вычислений в типе `Maybe` или получению ещё большей недетерминированности для списков), но она действительно возвращает что-то в качестве своего результата.

Когда ваши недетерминированные значения взаимодействуют, вы можете воспринимать их вычисление как дерево, где каждый возможный результат в списке представляет отдельную ветку. Вот предыдущее выражение, переписанное в нотации `do`:

```
listOfTuples :: [(Int,Char)]
listOfTuples = do
   n <- [1,2]
   ch <- ['a','b']
   return (n,ch)
```

Такая запись делает чуть более очевидным то, что образец `n` принимает каждое значение из списка `[1,2]`, а образец `ch` – каждое значение из списка `['a','b']`. Как и в случае с типом `Maybe`, мы извлекаем элементы из монадического значения и обрабатываем их как обычные значения, а операция `>>=` беспокоится о контексте за нас. Контекстом в данном случае является недетерминированность.

Нотация do и генераторы списков

Использование списков в нотации do может напоминать вам о чём-то, что вы уже видели ранее. Например, посмотрите на следующий кусок кода:

```
ghci> [(n,ch) | n <- [1,2], ch <- ['a','b']]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Да! Генераторы списков! В нашем примере, использующем нотацию do, образец n принимал значения всех результатов из списка [1,2]. Для каждого такого результата образцу ch был присвоен результат из списка ['a','b'], а последняя строка помещала пару (n, ch) в контекст по умолчанию (одноэлементный список) для возврата его в качестве результата без привнесения какой-либо дополнительной недетерминированности. В генераторе списка произошло то же самое, но нам не нужно было писать вызов функции return в конце для возврата пары (n, ch) в качестве результата, потому что выводящая часть генератора списка сделала это за нас.

На самом деле генераторы списков являются просто синтаксическим сахаром для использования списков как монад. В конечном счёте генераторы списков и списки, используемые в нотации do, переводятся в использование операции >>= для осуществления вычислений, которые обладают недетерминированностью.

Класс MonadPlus и функция guard

Генераторы списков позволяют нам фильтровать наши выходные данные. Например, мы можем отфильтровать список чисел в поиске только тех из них, которые содержат цифру 7:

```
ghci> [x | x <- [1..50], '7' `elem` show x]
[7,17,27,37,47]
```

Мы применяем функцию show к параметру x чтобы превратить наше число в строку, а затем проверяем, является ли символ '7' частью этой строки.

Чтобы увидеть, как фильтрация в генераторах списков преобразуется в списковую монаду, мы должны рассмотреть функцию guard и класс типов MonadPlus.

Класс типов MonadPlus предназначен для монад, которые также могут вести себя как моноиды. Вот его определение:

```
class Monad m => MonadPlus m where
   mzero :: m a
   mplus :: m a -> m a -> m a
```

Функция `mzero` является синонимом функции `mempty` из класса типов `Monoid`, а функция `mplus` соответствует функции `mappend`. Поскольку списки являются моноидами, а также монадами, их можно сделать экземпляром этого класса типов:

```
instance MonadPlus [] where
   mzero = []
   mplus = (++)
```

Для списков функция `mzero` представляет недетерминированное вычисление, которое вообще не имеет результата – неуспешно окончившееся вычисление. Функция `mplus` сводит два недетерминированных значения в одно. Функция `guard` определена следующим образом:

```
guard :: (MonadPlus m) => Bool -> m ()
guard True = return ()
guard False = mzero
```

Функция `guard` принимает значение типа `Bool`. Если это значение равно `True`, функция `guard` берёт пустой кортеж `()` и помещает его в минимальный контекст, который по-прежнему является успешным. Если значение типа `Bool` равно `False`, функция `guard` создаёт монадическое значение с неудачей в вычислениях. Вот эта функция в действии:

```
ghci> guard (5 > 2) :: Maybe ()
Just ()
ghci> guard (1 > 2) :: Maybe ()
Nothing
ghci> guard (5 > 2) :: [()]
[()]
ghci> guard (1 > 2) :: [()]
[]
```

Выглядит интересно, но чем это может быть полезно? В списковой монаде мы используем её для фильтрации недетерминированных вычислений:

```
ghci> [1..50] >>= (\x -> guard ('7' `elem` show x) >> return x)
```

```
[7,17,27,37,47]
```

Результат аналогичен тому, что был возвращен нашим предыдущим генератором списка. Как функция `guard` достигла этого? Давайте сначала посмотрим, как она функционирует совместно с операцией `>>`:

```
ghci> guard (5 > 2) >> return "клёво" :: [String]
["клёво"]
ghci> guard (1 > 2) >> return "клёво" :: [String]
[]
```

Если функция `guard` срабатывает успешно, результатом, находящимся в ней, будет пустой кортеж. Поэтому дальше мы используем операцию `>>`, чтобы игнорировать этот пустой кортеж и предоставить что-нибудь другое в качестве результата. Однако если функция `guard` не срабатывает успешно, функция `return` впоследствии тоже не сработает успешно, потому что передача пустого списка функции с помощью операции `>>=` всегда даёт в результате пустой список. Функция `guard` просто говорит: «Если это значение типа `Bool` равно `False`, верни неуспешное окончание вычислений прямо здесь. В противном случае создай успешное значение, которое содержит в себе значение-пустышку `()`». Всё, что она делает, – позволяет вычислению продолжиться.

Вот предыдущий пример, переписанный в нотации `do`:

```
sevensOnly :: [Int]
sevensOnly = do
   x <- [1..50]
   guard ('7' `elem` show x)
   return x
```

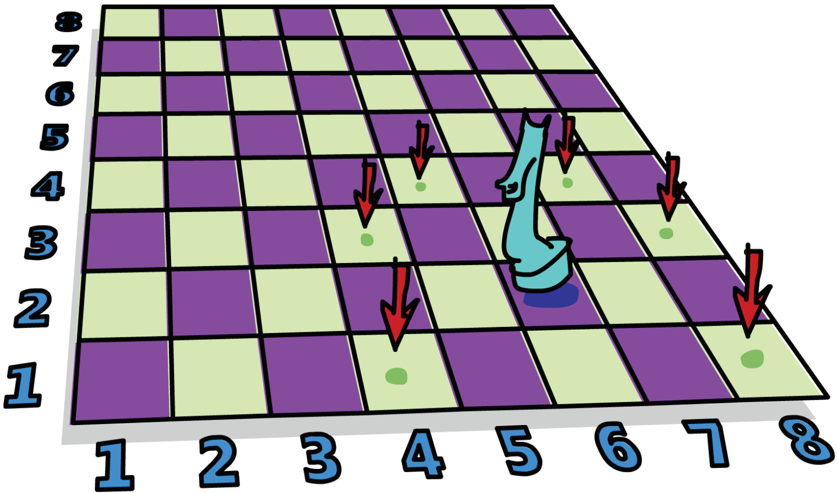
Если бы мы забыли представить образец `x` в качестве окончательного результата, используя функцию `return`, то результирующий список состоял бы просто из пустых кортежей. Вот определение в форме генератора списка:

```
ghci> [x | x <- [1..50], '7' `elem` show x]
[7,17,27,37,47]
```

Поэтому фильтрация в генераторах списков – это то же самое, что использование функции `guard`.

Ход конём

Есть проблема, которая очень подходит для решения с помощью недетерминированности. Скажем, у нас есть шахматная доска и на ней только одна фигура – конь. Мы хотим определить, может ли конь достигнуть определённой позиции в три хода. Будем использовать пару чисел для представления позиции коня на шахматной доске. Первое число будет определять столбец, в котором он находится, а второе число – строку.



Создадим синоним типа для текущей позиции коня на шахматной доске.

```
type KnightPos = (Int, Int)
```

Теперь предположим, что конь начинает движение с позиции `(6, 2)`. Может ли он добраться до `(6, 1)` именно за три хода? Какой ход лучше сделать следующим из его нынешней позиции? Я знаю: как насчёт их всех?! К нашим услугам недетерминированность, поэтому вместо того, чтобы выбрать один ход, давайте просто выберем их все сразу! Вот функция, которая берёт позицию коня и возвращает все его следующие ходы:

```
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c,r) = do
```

```
   (c',r') <- [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
              ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)
              ]
   guard (c' `elem` [1..8] && r' `elem` [1..8])
   return (c',r')
```

Конь всегда может перемещаться на одну клетку горизонтально или вертикально и на две клетки вертикально или горизонтально, причём каждый его ход включает движение и по горизонтали, и по вертикали. Пара `(c', r')` получает каждое значение из списка перемещений, а затем функция `guard` заботится о том, чтобы новый ход, а именно пара `(c', r')`, был в пределах доски. Если движение выходит за доску, она возвращает пустой список, что приводит к неудаче, и вызов `return (c', r')` не обрабатывается для данной позиции.

Эта функция может быть записана и без использования списков в качестве монад. Вот как записать её с использованием функции `filter`:

```
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c,r) = filter onBoard
   [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
   ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)
   ]
  where onBoard (c,r) = c `elem` [1..8] && r `elem` [1..8]
```

Обе версии делают одно и то же, так что выбирайте ту, которая кажется вам лучше. Давайте опробуем функцию:

```
ghci> moveKnight (6, 2)
[(8,1),(8,3),(4,1),(4,3),(7,4),(5,4)]
ghci> moveKnight (8, 1)
[(6,2),(7,3)]
```

Работает чудесно! Мы берём одну позицию и просто выполняем все возможные ходы сразу, так сказать.

Поэтому теперь, когда у нас есть следующая недетерминированная позиция, мы просто используем операцию `>>=`, чтобы передать её функции `moveKnight`. Вот функция, принимающая позицию и возвращающая все позиции, которые вы можете достигнуть из неё в три хода:

```
in3 :: KnightPos -> [KnightPos]
```

```
in3 start = do
   first <- moveKnight start
   second <- moveKnight first
   moveKnight second
```

Если вы передадите ей пару (6, 2), результирующий список будет довольно большим. Причина в том, что если есть несколько путей достигнуть определённой позиции в три хода, ход неожиданно появляется в списке несколько раз.

Вот предшествующий код без использования нотации do:

```
in3 start = return start >>= moveKnight >>= moveKnight >>= moveKnight
```

Однократное использование операции >>= даёт нам все возможные ходы с начала. Когда мы используем операцию >>= второй раз, то для каждого возможного первого хода вычисляется каждый возможный следующий ход; то же самое верно и в отношении последнего хода.

Помещение значения в контекст по умолчанию с применением к нему функции return, а затем передача его функции с использованием операции >>= – то же самое, что и обычное применение функции к данному значению; но мы сделали это здесь, во всяком случае, ради стиля.

Теперь давайте создадим функцию, которая принимает две позиции и сообщает нам, можем ли мы попасть из одной в другую ровно в три хода:

```
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

Мы производим все возможные позиции в пределах трёх ходов, а затем проверяем, находится ли среди них искомая.

Вот как проверить, можем ли мы попасть из (6,2) в (6,1) в три хода:

```
ghci> (6, 2) `canReachIn3` (6, 1)
True
```

Да! Как насчёт из (6, 2) в (7, 3)?

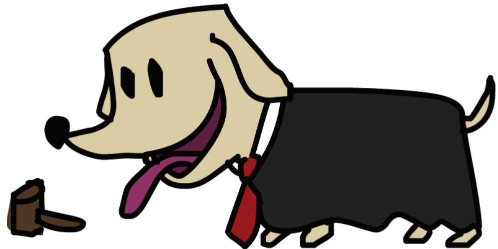
```
ghci> (6, 2) `canReachIn3` (7, 3)
False
```

Нет! В качестве упражнения вы можете изменить эту функцию так, чтобы она показывала вам ходы, которые нужно совершить, когда вы можете достигнуть одной позиции из другой. В главе 14 вы увидите, как изменить эту функцию, чтобы также передавать ей число ходов, которые необходимо произвести, вместо того чтобы кодировать это число жёстко, как сейчас.

Законы монад

Так же, как в отношении функторов и аппликативных функторов, в отношении монад действует несколько законов, которым должны подчиняться все экземпляры класса `Monad`. Даже если что-то сделано экземпляром класса типов `Monad`, это ещё не означает, что на самом деле перед нами монада. Чтобы тип по-настоящему был монадой, для него должны выполняться законы монад. Эти законы позволяют нам делать обоснованные предположения о типе и его поведении.

Язык Haskell позволяет любому типу быть экземпляром любого класса типов, пока типы удаётся проверить. Впрочем, он не может проверить, выполняются ли законы монад для типа, поэтому если мы создаём новый экземпляр класса типов `Monad`, мы должны обладать достаточной уверенностью в том, что с выполнением законов монад для этого типа всё хорошо. Можно полагаться на то, что типы в стандартной библиотеке удовлетворяют законам, но когда мы перейдем к созданию собственных монад, нам необходимо будет проверять выполнение законов вручную. Впрочем, не беспокойтесь – эти законы совсем не сложны!



Левая единица

Первый закон монад утверждает, что если мы берём значение, помещаем его в контекст по умолчанию с помощью функции `return`, а затем передаём его функции, используя операцию `>>=`, это равнозначно тому, как если бы мы просто взяли значение и применили

к нему функцию. Говоря формально, `return x >>= f` – это то же самое, что и `f x`.

Если вы посмотрите на монадические значения как на значения с контекстом и на функцию `return` как на получение значения и помещение его в минимальный контекст по умолчанию, который по-прежнему возвращает это значение в качестве результата функции, то закон имеет смысл. Если данный контекст действительно минимален, передача этого монадического значения функции не должна сильно отличаться от простого применения функции к обычному значению – и действительно, вообще ничем не отличается.

Функция `return` для монады `Maybe` определена как вызов конструктора `Just`. Вся суть монады `Maybe` состоит в возможном неуспехе в вычислениях, и если у нас есть значение, которое мы хотим поместить в такой контекст, есть смысл в том, чтобы обрабатывать его как успешное вычисление, поскольку мы знаем, каким является значение. Вот некоторые примеры использования функции `return` с типом `Maybe`:

```
ghci> return 3 >>= (\x -> Just (x+100000))
Just 100003
ghci> (\x -> Just (x+100000)) 3
Just 100003
```

Для списковой монады функция `return` помещает что-либо в одноэлементный список. Реализация операции `>>=` для списков проходит по всем значениям в списке и применяет к ним функцию. Однако, поскольку в одноэлементном списке лишь одно значение, это аналогично применению функции к данному значению:

```
ghci> return "WoM" >>= (\x -> [x,x,x])
["WoM","WoM","WoM"]
ghci> (\x -> [x,x,x]) "WoM"
["WoM","WoM","WoM"]
```

Вы знаете, что для монады `IO` использование функции `return` создаёт действие ввода-вывода, которое не имеет побочных эффектов, но просто возвращает значение в качестве своего результата. Поэтому вполне логично, что этот закон выполняется также и для монады `IO`.

Правая единица

Второй закон утверждает, что если у нас есть монадическое значение и мы используем операцию `>>=` для передачи его функции `return`, результатом будет наше изначальное монадическое значение. Формально `m >>= return` является не чем иным, как просто `m`.

Этот закон может быть чуть менее очевиден, чем первый. Давайте посмотрим, почему он должен выполняться. Когда мы передаём монадические значения функции, используя операцию `>>=`, эти функции принимают обычные значения и возвращают монадические. Функция `return` тоже является такой, если вы рассмотрите её тип.

Функция `return` помещает значение в минимальный контекст, который по-прежнему возвращает это значение в качестве своего результата. Это значит, что, например, для типа `Maybe` она не вносит никакого неуспеха в вычислениях; для списков – не вносит какую-либо дополнительную недетерминированность.

Вот пробный запуск для нескольких монад:

```
ghci> Just "двигайся дальше" >>= (\x -> return x)
Just "двигайся дальше"
ghci> [1,2,3,4] >>= (\x -> return x)
[1,2,3,4]
ghci> putStrLn "Вах!" >>= (\x -> return x)
Вах!
```

В этом примере со списком реализация операции `>>=` выглядит следующим образом:

```
xs >>= f = concat (map f xs)
```

Поэтому когда мы передаём список `[1,2,3,4]` функции `return`, сначала она отображает `[1,2,3,4]`, что в результате даёт список списков `[[1],[2],[3],[4]]`. Затем это конкатенируется, и мы получаем наш изначальный список.

Левое тождество и правое тождество являются, по сути, законами, которые описывают, как должна вести себя функция `return`. Это важная функция для превращения обычных значений в монадические, и было бы нехорошо, если бы монадическое значение, которое она произвела, имело больше, чем необходимый минимальный контекст.

Ассоциативность

Последний монадический закон говорит, что когда у нас есть цепочка применений монадических функций с помощью операции `>>=`, не должно иметь значения то, как они вложены. В формальной записи выполнение `(m >>= f) >>= g` – точно то же, что и выполнение `m >>= (\x -> f x >>= g)`.

Гм-м, что теперь тут происходит? У нас есть одно монадическое значение, `m`, и две монадические функции, `f` и `g`. Когда мы выполняем выражение `(m >>= f) >>= g`, то передаём значение `m` в функцию `f`, что даёт в результате монадическое значение. Затем мы передаём это новое монадическое значение функции `g`. В выражении `m >>= (\x -> f x >>= g)` мы берём монадическое значение и передаём его функции, которая передаёт результат применения `f x` функции `g`. Нелегко увидеть, почему обе эти записи равны, так что давайте взглянем на пример, который делает это равенство немного более очевидным.

Помните нашего канатоходца Пьера, который пытался удержать равновесие, в то время как птицы приземлялись на его балансировочный шест? Чтобы симулировать приземление птиц на балансировочный шест, мы создали цепочку из нескольких функций, которые могли вызывать неуспешное окончание вычислений:

```
ghci> return (0, 0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
Just (2,4)
```

Мы начали со значения `Just (0, 0)`, а затем связали это значение со следующей монадической функцией `landRight 2`. Результатом было другое монадическое значение, связанное со следующей монадической функцией, и т. д. Если бы надлежало явно заключить это в скобки, мы написали бы следующее:

```
ghci> ((return (0, 0) >>= landRight 2) >>= landLeft 2) >>= landRight 2
Just (2,4)
```

Но мы также можем записать инструкцию вот так:

```
return (0, 0) >>= (\x ->
landRight 2 x >>= (\y ->
landLeft 2 y >>= (\z ->
landRight 2 z)))
```

Вызов `return (0, 0)` – то же самое, что `Just (0, 0)`, и когда мы передаём это анонимной функции, образец `x` принимает значение `(0, 0)`.

Функция `landRight` принимает количество птиц и шест (кортеж, содержащий числа) – и это то, что ей передаётся. В результате мы имеем значение `Just (0, 2)`, и, когда передаём его следующей анонимной функции, образец `y` становится равен `(0, 2)`. Это продолжается до тех пор, пока последнее приземление птицы не вернёт в качестве результата значение `Just (2, 4)`, что в действительности является результатом всего выражения.

Поэтому неважно, как у вас вложена передача значений монадическим функциям. Важен их смысл. Давайте рассмотрим ещё один способ реализации этого закона. Предположим, мы производим композицию двух функций, `f` и `g`:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = (\x -> f (g x))
```

Если функция `g` имеет тип `a -> b` и функция `f` имеет тип `b -> c`, мы компонуем их в новую функцию типа `a -> c`, чтобы её параметр передавался между этими функциями. А что если эти две функции – монадические? Что если возвращаемые ими значения были бы монадическими? Если бы у нас была функция типа `a -> m b`, мы не могли бы просто передать её результат функции типа `b -> m c`, потому что эта функция принимает обычное значение `b`, не монадическое. Чтобы всё-таки достичь нашей цели, можно воспользоваться операцией `<=<`:

```
(<=<) :: (Monad m) => (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g = (\x -> g x >>= f)
```

Поэтому теперь мы можем производить композицию двух монадических функций:

```
ghci> let f x = [x,-x]
ghci> let g x = [x*3,x*2]
ghci> let h = f <=< g
ghci> h 3
[9,-9,6,-6]
```

Ладно, всё это здорово. Но какое это имеет отношение к закону ассоциативности? Просто, когда мы рассматриваем этот закон как закон композиций, он утверждает, что `f <=< (g <=< h)` должно быть равнозначно `(f <=< g) <=< h`. Это всего лишь ещё один способ доказать, что для монад вложенность операций не должна иметь значения.

Если мы преобразуем первые два закона так, чтобы они использовали операцию `<=<`, то закон левого тождества утверждает, что для каждой монадической функции `f` выражение `f <=< return` означает то же самое, что просто вызвать `f`. Закон правого тождества говорит, что выражение `return <=< f` также ничем не отличается от простого вызова `f`. Это подобно тому, как если бы `f` являлась обычной функцией, и тогда `(f . g) . h` было бы аналогично `f . (g . h)`, выражение `f . id` – всегда аналогично `f`, и выражение `id . f` тоже ничем не отличалось бы от вызова `f`.

В этой главе мы в общих чертах ознакомились с монадами и изучили, как работают монада `Maybe` и списковая монада. В следующей главе мы рассмотрим целую кучу других крутых монад, а также создадим нашу собственную.

14
ЕЩЁ НЕМНОГО МОНАД

Мы видели, как монады могут быть использованы для получения значений с контекстами и применения их к функциям и как использование оператора `>>=` или нотации `do` позволяет нам сфокусироваться на самих значениях, в то время как контекст обрабатывается за нас.

Мы познакомились с монадой `Maybe` и увидели, как она добавляет к значениям контекст возможного неуспеха в вычислениях. Мы узнали о списковой монаде и увидели, как легко она позволяет нам вносить недетерминированность в наши программы. Мы также научились работать в монаде `IO` даже до того, как вообще выяснили, что такое монада!

В этой главе мы узнаем ещё о нескольких монадах. Мы увидим, как они могут сделать наши программы понятнее, позволяя нам обрабатывать все типы значений как монадические значения. Исследование ряда примеров также укрепит наше понимание монад.

Все монады, которые нам предстоит рассмотреть, являются частью пакета *mtl*. В языке Haskell пакетом является совокупность модулей. Пакет *mtl* идёт в поставке

с Haskell Platform, так что он у вас, вероятно, уже есть. Чтобы проверить, так ли это, выполните команду `ghc-pkg list` в командной строке. Эта команда покажет, какие пакеты для языка Haskell у вас уже установлены; одним из таких пакетов должен являться *mtl*, за названием которого следует номер версии.

Writer? Я о ней почти не знаю!

Итак, мы зарядили наш пистолет монадой `Maybe`, списковой монадой и монадой `IO`. Теперь давайте поместим в патронник монаду `Writer` и посмотрим, что произойдёт, когда мы выстрелим ею!

Между тем как `Maybe` предназначена для значений с добавленным контекстом неуспешно оканчивающихся вычислений, а список – для недетерминированных вычислений, монада `Writer` предусмотрена для значений, к которым присоединено другое значение, ведущее себя наподобие журнала. Монада `Writer` позволяет нам производить вычисления, в то же время обеспечивая слияние всех журнальных значений в одно, которое затем присоединяется к результату.

Например, мы могли бы снабдить наши значения строками, которые объясняют, что происходит, возможно, для отладочных целей. Рассмотрите функцию, которая принимает число бандитов в банде и сообщает нам, является ли эта банда крупной. Это очень простая функция:

```
isBigGang :: Int -> Bool
isBigGang x = x > 9
```

Ну а что если теперь вместо возвращения значения `True` или `False` мы хотим, чтобы функция также возвращала строку журнала, которая сообщает, что она сделала? Что ж, мы просто создаём эту строку и возвращаем её наряду с нашим значением `Bool`:

```
isBigGang :: Int -> (Bool, String)
isBigGang x = (x > 9, "Размер банды сравнён с 9.")
```

Так что теперь вместо того, чтобы просто вернуть значение типа `Bool`, мы возвращаем кортеж, первым компонентом которого является само значение, а вторым компонентом – строка, сопутствующая этому значению. Теперь у нашего значения появился некоторый добавленный контекст. Давайте опробуем функцию:

```
ghci> isBigGang 3
(False,"Размер банды сравнён с 9.")
ghci> isBigGang 30
(True,"Размер банды сравнён с 9.")
```

Пока всё нормально. Функция isBigGang принимает нормальное значение и возвращает значение с контекстом. Как мы только что увидели, передача ей нормального значения не составляет сложности. Теперь предположим, что у нас уже есть значение, у которого имеется журнальная запись, присоединенная к нему – такая как (3, "Небольшая банда.") – и мы хотим передать его функции isBigGang. Похоже, перед нами снова встаёт вопрос: если у нас есть функция, которая принимает нормальное значение и возвращает значение с контекстом, как нам взять нормальное значение с контекстом и передать его функции?

Исследуя монаду Maybe, мы создали функцию applyMaybe, которая принимала значение типа Maybe a и функцию типа a -> Maybe b и передавала это значение Maybe a в функцию, даже если функция принимает нормальное значение типа a вместо Maybe a. Она делала это, следя за контекстом, имеющимся у значений типа Maybe a, который означает, что эти значения могут быть значениями с неуспехом вычислений. Но внутри функции типа a -> Maybe b мы могли обрабатывать это значение как нормальное, потому что applyMaybe (которая позже стала функцией >>=) проверяла, являлось ли оно значением Nothing либо значением Just.

В том же духе давайте создадим функцию, которая принимает значение с присоединенным журналом, то есть значением типа (a,String), и функцию типа a -> (b,String), и передаёт это значение в функцию. Мы назовём её applyLog. Однако поскольку значение типа (a,String) не несёт с собой контекст возможной неудачи, но несёт контекст добавочного значения журнала, функция applyLog

будет обеспечивать сохранность первоначального значения журнала, объединяя его со значением журнала, возвращаемого функцией. Вот реализация этой функции:

```
applyLog :: (a,String) -> (a -> (b,String)) -> (b,String)
applyLog (x,log) f = let (y,newLog) = f x in (y,log ++ newLog)
```

Когда у нас есть значение с контекстом и мы хотим передать его функции, то мы обычно пытаемся отделить фактическое значение от контекста, затем пытаемся применить функцию к этому значению, а потом смотрим, сбережён ли контекст. В монаде `Maybe` мы проверяли, было ли значение равно `Just x`, и если было, мы брали это значение `x` и применяли к нему функцию. В данном случае очень просто определить фактическое значение, потому что мы имеем дело с парой, где один компонент является значением, а второй – журналом. Так что сначала мы просто берём значение, то есть `x`, и применяем к нему функцию `f`. Мы получаем пару `(y,newLog)`, где `y` является новым результатом, а `newLog` – новым журналом. Но если мы вернули это в качестве результата, прежнее значение журнала не было бы включено в результат, так что мы возвращаем пару `(y,log ++ newLog)`. Мы используем операцию конкатенации ++, чтобы добавить новый журнал к прежнему.

Вот функция `applyLog` в действии:

```
ghci> (3, "Небольшая банда.") `applyLog` isBigGang
(False,"Небольшая банда.Размер банды сравнён с 9.")
ghci> (30, "Бешеный взвод.") `applyLog` isBigGang
(True,"Бешеный взвод.Размер банды сравнён с 9.")
```

Результаты аналогичны предшествующим, только теперь количеству бандитов сопутствует журнал, который включен в окончательный журнал.

Вот ещё несколько примеров использования `applyLog`:

```
ghci> ("Тобин","Вне закона.") `applyLog` (\x -> (length x "Длина."))
(5,"Вне закона.Длина.")
ghci> ("Котопёс","Вне закона.") `applyLog` (\x -> (length x "Длина."))
(7,"Вне закона.Длина.")
```

Смотрите, как внутри анонимной функции образец `x` является просто нормальной строкой, а не кортежем, и как функция `applyLog` заботится о добавлении записей журнала.

Моноиды приходят на помощь

Убедитесь, что вы на данный момент знаете, что такое моноиды!

Прямо сейчас функция `applyLog` принимает значения типа `(a,String)`, но есть ли смысл в том, чтобы тип журнала был `String`? Он использует операцию ++ для добавления записей журнала – не будет ли это работать и в отношении любого типа списков, не только списка символов? Конечно же, будет! Мы можем пойти дальше и изменить тип этой функции на следующий:

```
applyLog :: (a,[c]) -> (a -> (b,[c])) -> (b,[c])
```

Теперь журнал является списком. Тип значений, содержащихся в списке, должен быть одинаковым как для изначального списка, так и для списка, который возвращает функция; в противном случае мы не смогли бы использовать операцию ++ для «склеивания» их друг с другом.

Сработало бы это для строк байтов? Нет причины, по которой это не сработало бы! Однако тип, который у нас имеется, работает только со списками. Похоже, что нам пришлось бы создать ещё одну функцию `applyLog` для строк байтов. Но подождите! И списки, и строки байтов являются моноидами. По существу, те и другие являются экземплярами класса типов `Monoid`, а это значит, что они реализуют функцию `mappend`. Как для списков, так и для строк байтов функция `mappend` производит конкатенацию. Смотрите:

```
ghci> [1,2,3] `mappend` [4,5,6]
[1,2,3,4,5,6]
ghci> B.pack [99,104,105] `mappend` B.pack [104,117,97,104,117,97]
Chunk "chi" (Chunk "huahua" Empty)
```

Круто! Теперь наша функция `applyLog` может работать для любого моноида. Мы должны изменить тип, чтобы отразить это, а также реализацию, потому что следует заменить вызов операции ++ вызовом функции `mappend`:

```
applyLog :: (Monoid m) => (a,m) -> (a -> (b,m)) -> (b,m)
applyLog (x,log) f = let (y,newLog) = f x
                     in (y,log `mappend` newLog)
```

Поскольку сопутствующее значение теперь может быть любым моноидным значением, нам больше не нужно думать о кортеже как

о значении и журнале, но мы можем думать о нём как о значении с сопутствующим моноидным значением. Например, у нас может быть кортеж, в котором есть имя предмета и цена предмета в виде моноидного значения. Мы просто используем определение типа `newtype Sum`, чтобы быть уверенными, что цены добавляются, пока мы работаем с предметами. Вот функция, которая добавляет напиток к обеду какого-то ковбоя:

```
import Data.Monoid

type Food = String
type Price = Sum Int

addDrink :: Food -> (Food,Price)
addDrink "бобы" = ("молоко", Sum 25)
addDrink "вяленое мясо" = ("виски", Sum 99)
addDrink _ = ("пиво", Sum 30)
```

Мы используем строки для представления продуктов и тип `Int` в обёртке типа `newtype Sum` для отслеживания того, сколько центов стоит тот или иной продукт. Просто напомню: выполнение функции `mappend` для значений типа `Sum` возвращает сумму обёрнутых значений.

```
ghci> Sum 3 `mappend` Sum 9
Sum {getSum = 12}
```

Функция `addDrink` довольно проста. Если мы едим бобы, она возвращает `"молоко"` вместе с `Sum 25`; таким образом, `25` центов завёрнуты в конструктор `Sum`. Если мы едим вяленое мясо, то пьём виски, а если едим что-то другое – пьём пиво. Обычное применение этой функции к продукту сейчас было бы не слишком интересно, а вот использование функции `applyLog` для передачи продукта с указанием цены в саму функцию представляет интерес:

```
ghci> ("бобы", Sum 10) `applyLog` addDrink
("молоко",Sum {getSum = 35})
ghci> ("вяленое мясо", Sum 25) `applyLog` addDrink
("виски",Sum {getSum = 124})
ghci> ("собачатина", Sum 5) `applyLog` addDrink
("пиво",Sum {getSum = 35})
```

Молоко стоит 25 центов, но если мы заедаем его бобами за 10 центов, это обходится нам в 35 центов. Теперь ясно, почему присоединенное значение не всегда должно быть журналом – оно может быть любым моноидным значением, и то, как эти два значения объединяются, зависит от моноида. Когда мы производили записи в журнал, они присоединялись в конец, но теперь происходит сложение чисел.

Поскольку значение, возвращаемое функцией `addDrink`, является кортежем типа `(Food,Price)`, мы можем передать этот результат функции `addDrink` ещё раз, чтобы функция сообщила нам, какой напиток будет подан в сопровождение к блюду и сколько это нам будет стоить. Давайте попробуем:

```
ghci> ("собачатина", Sum 5) `applyLog` addDrink `applyLog` addDrink
("пиво",Sum {getSum = 65})
```

Добавление напитка к какой-нибудь там собачатине вернёт пиво и дополнительные 30 центов, то есть `("пиво", Sum 35)`. А если мы используем функцию `applyLog` для передачи этого результата функции `addDrink`, то получим ещё одно пиво, и результатом будет `("пиво", Sum 65)`.

Тип Writer

Теперь, когда мы увидели, что значение с присоединенным моноидом ведёт себя как монадическое значение, давайте исследуем экземпляр класса `Monad` для типов таких значений. Модуль `Control.Monad.Writer` экспортирует тип `Writer w a` со своим экземпляром класса `Monad` и некоторые полезные функции для работы со значениями такого типа.

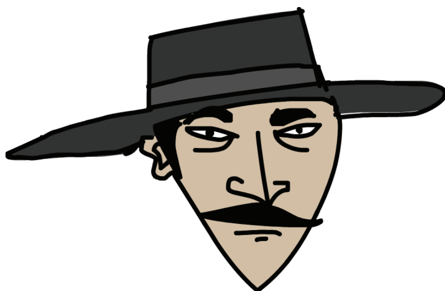
Прежде всего, давайте исследуем сам тип. Для присоединения моноида к значению нам достаточно поместить их в один кортеж. Тип `Writer w a` является просто обёрткой `newtype` для кортежа. Его определение несложно:

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

Чтобы кортеж мог быть сделан экземпляром класса `Monad` и его тип был отделён от обычного кортежа, он обёрнут в `newtype`. Параметр типа `a` представляет тип значения, параметр типа `w` – тип присоединенного значения моноида.

Экземпляр класса Monad для этого типа определён следующим образом:

```
instance (Monoid w) => Monad (Writer w) where
   return x = Writer (x, mempty)
   (Writer (x,v)) >>= f = let (Writer (y, v')) = f x
                          in Writer (y, v `mappend` v')
```



Во-первых, давайте рассмотрим операцию >>=. Её реализация по существу аналогична функции applyLog, только теперь, поскольку наш кортеж обёрнут в тип newtype Writer, мы должны развернуть его перед сопоставлением с образцом. Мы берём значение x и применяем к нему функцию f. Это даёт нам новое значение Writer w a, и мы используем выражение let для сопоставления его с образцом. Представляем y в качестве нового результата и используем функцию mappend для объединения старого моноидного значения с новым. Упаковываем его вместе с результирующим значением в кортеж, а затем оборачиваем с помощью конструктора Writer, чтобы нашим результатом было значение Writer, а не просто необёрнутый кортеж.

Ладно, а что у нас с функцией return? Она должна принимать значение и помещать его в минимальный контекст, который по-прежнему возвращает это значение в качестве результата. Так каким был бы контекст для значений типа Writer? Если мы хотим, чтобы сопутствующее моноидное значение оказывало на другие моноидные значения наименьшее влияние, имеет смысл использовать функцию mempty. Функция mempty используется для представления «единичных» моноидных значений, как, например, "", Sum 0 и пустые строки байтов. Когда мы выполняем вызов функции mappend между значением mempty и каким-либо другим моноидным значением, результатом будет это второе моноидное значение. Так что если мы используем функцию return для создания значения монады Writer, а затем применяем оператор >>= для передачи этого значения функции, окончательным моноидным значением будет только то, что возвращает функция. Давайте используем функцию

return с числом 3 несколько раз, только каждый раз будем соединять его попарно с другим моноидом:

```
ghci> runWriter (return 3 :: Writer String Int)
(3,"")
ghci> runWriter (return 3 :: Writer (Sum Int) Int)
(3,Sum {getSum = 0})
ghci> runWriter (return 3 :: Writer (Product Int) Int)
(3,Product {getProduct = 1})
```

Поскольку у типа Writer нет экземпляра класса Show, нам пришлось использовать функцию runWriter для преобразования наших значений типа Writer в нормальные кортежи, которые могут быть показаны в виде строки. Для строк единичным значением является пустая строка. Для типа Sum это значение 0, потому что если мы прибавляем к чему-то 0, это что-то не изменяется. Для типа Product единичным значением является 1.

В экземпляре класса Monad для типа Writer не имеется реализация для функции fail; значит, если сопоставление с образцом в нотации do оканчивается неудачно, вызывается функция error.

Использование нотации do с типом Writer

Теперь, когда у нас есть экземпляр класса Monad, мы свободно можем использовать нотацию do для значений типа Writer. Это удобно, когда у нас есть несколько значений типа Writer и мы хотим с ними что-либо делать. Как и в случае с другими монадами, можно обрабатывать их как нормальные значения, и контекст сохраняется для нас. В этом случае все моноидные значения, которые идут в присоединенном виде, объединяются с помощью функции mappend, а потому отражаются в окончательном результате. Вот простой пример использования нотации do с типом Writer для умножения двух чисел:

```
import Control.Monad.Writer

logNumber :: Int -> Writer [String] Int
logNumber x = Writer (x, ["Получено число: " ++ show x])

multWithLog :: Writer [String] Int
multWithLog = do
   a <- logNumber 3
```

```
   b <- logNumber 5
   return (a*b)
```

Функция logNumber принимает число и создаёт из него значение типа Writer. Для моноида мы используем список строк и снабжаем число одноэлементным списком, который просто говорит, что мы получили это число. Функция multWithLog – это значение типа Writer, которое перемножает 3 и 5 и гарантирует включение прикрепленных к ним журналов в окончательный журнал. Мы используем функцию return, чтобы вернуть значение (a*b) в качестве результата. Поскольку функция return просто берёт что-то и помещает в минимальный контекст, мы можем быть уверены, что она ничего не добавит в журнал. Вот что мы увидим, если выполним этот код:

```
ghci> runWriter multWithLog
(15,["Получено число: 3","Получено число: 5"])
```

Добавление в программы функции журналирования

Иногда мы просто хотим, чтобы некое моноидное значение было включено в каком-то определённом месте. Для этого может пригодиться функция tell. Она является частью класса типов MonadWriter и в случае с типом Writer берёт монадическое значение вроде ["Всё продолжается"] и создаёт значение типа Writer, которое возвращает значение-пустышку () в качестве своего результата, но прикрепляет желаемое моноидное значение. Когда у нас есть монадическое значение, которое в качестве результата содержит значение (), мы не привязываем его к переменной. Вот определение функции multWithLog с включением некоторых дополнительных сообщений:

```
multWithLog :: Writer [String] Int
multWithLog = do
   a <- logNumber 3
   b <- logNumber 5
   tell ["Перемножим эту парочку"]
   return (a*b)
```

Важно, что вызов return (a*b) находится в последней строке, потому что результат последней строки в выражении do является результатом всего выражения do. Если бы мы поместили вызов функции tell на последнюю строку, результатом этого выражения do

было бы (). Мы бы потеряли результат умножения. Однако журнал остался бы прежним. Вот функция в действии:

```
ghci> runWriter multWithLog
(15,["Получено число: 3","Получено число: 5","Перемножим эту парочку"])
```

Добавление журналирования в программы

Алгоритм Евклида – это алгоритм, который берёт два числа и вычисляет их наибольший общий делитель, то есть самое большое число, на которое делятся без остатка оба числа. В языке Haskell уже имеется функция `gcd`, которая проделывает это, но давайте реализуем её сами, а затем снабдим её возможностями журналирования. Вот обычный алгоритм:

```
gcd' :: Int -> Int -> Int
gcd' a b
   | b == 0 = a
   | otherwise = gcd' b (a `mod` b)
```

Алгоритм очень прост. Сначала он проверяет, равно ли второе число 0. Если равно, то результатом становится первое число. Если не равно, то результатом становится наибольший общий делитель второго числа и остаток от деления первого числа на второе. Например, если мы хотим узнать, каков наибольший общий делитель 8 и 3, мы просто следуем изложенному алгоритму. Поскольку 3 не равно 0, мы должны найти наибольший общий делитель 3 и 2 (если мы разделим 8 на 3, остатком будет 2). Затем ищем наибольший общий делитель 3 и 2. Число 2 по-прежнему не равно 0, поэтому теперь у нас есть 2 и 1. Второе число не равно 0, и мы выполняем алгоритм ещё раз для 1 и 0, поскольку деление 2 на 1 даёт нам остаток равный 0. И наконец, поскольку второе число равно 0, финальным результатом становится 1. Давайте посмотрим, согласуется ли наш код:

```
ghci> gcd' 8 3
1
```

Согласуется. Очень хорошо! Теперь мы хотим снабдить наш результат контекстом, а контекстом будет моноидное значение, которое ведёт себя как журнал. Как и прежде, мы используем список строк в качестве моноида. Поэтому тип нашей новой функции `gcd'` должен быть таким:

```
gcd' :: Int -> Int -> Writer [String] Int
```

Всё, что осталось сделать, – снабдить нашу функцию журнальными значениями. Вот код:

```
import Control.Monad.Writer

gcd' :: Int -> Int -> Writer [String] Int
gcd' a b
   | b == 0 = do
      tell ["Закончили: " ++ show a]
      return a
   | otherwise = do
      tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
      gcd' b (a `mod` b)
```

Эта функция принимает два обычных значения `Int` и возвращает значение типа `Writer [String] Int`, то есть целое число, обладающее контекстом журнала. В случае, когда параметр `b` принимает значение `0`, мы, вместо того чтобы просто вернуть значение `a` как результат, используем выражение `do` для сборки значения `Writer` в качестве результата. Сначала используем функцию `tell`, чтобы сообщить об окончании, а затем – функцию `return` для возврата значения `a` в качестве результата выражения `do`. Вместо данного выражения `do` мы также могли бы написать следующее:

```
Writer (a, ["Закончили: " ++ show a])
```

Однако я полагаю, что выражение `do` проще читать. Далее, у нас есть случай, когда значение `b` не равно `0`. В этом случае мы записываем в журнал, что используем функцию `mod` для определения остатка от деления `a` и `b`. Затем вторая строка выражения `do` просто рекурсивно вызывает `gcd'`. Вспомните: функция `gcd'` теперь, в конце концов, возвращает значение типа `Writer`, поэтому вполне допустимо наличие строки `gcd' b (a `mod` b)` в выражении `do`.

Хотя отслеживание выполнения этой новой функции `gcd'` вручную может быть отчасти полезным для того, чтобы увидеть, как записи присоединяются в конец журнала, я думаю, что лучше будет взглянуть на картину крупным планом, представляя эти значения как значения с контекстом, и отсюда понять, каким будет окончательный результат.

Давайте испытаем нашу новую функцию gcd'. Её результатом является значение типа Writer [String] Int, и если мы развернём его из принадлежащего ему newtype, то получим кортеж. Первая часть кортежа – это результат. Посмотрим, правильный ли он:

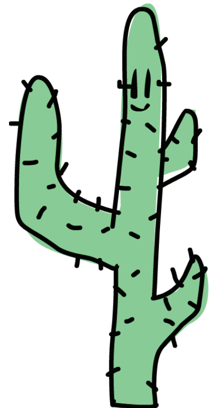
```
ghci> fst $ runWriter (gcd 8 3)
1
```

Хорошо! Теперь что насчёт журнала? Поскольку журнал является списком строк, давайте используем вызов mapM_ putStrLn для вывода этих строк на экран:

```
ghci> mapM_ putStrLn $ snd $ runWriter (gcd 8 3)
8 mod 3 = 2
3 mod 2 = 1
2 mod 1 = 0
Закончили: 1
```

Даже удивительно, как мы могли изменить наш обычный алгоритм на тот, который сообщает, что он делает по мере развития, просто превращая обычные значения в монадические и возлагая беспокойство о записях в журнал на реализацию оператора >>= для типа Writer!.. Мы можем добавить механизм журналирования почти в любую функцию. Всего лишь заменяем обычные значения значениями типа Writer, где мы хотим, и превращаем обычное применение функции в вызов оператора >>= (или выражения do, если это повышает «читабельность»).

Неэффективное создание списков



При использовании монады Writer вы должны внимательно выбирать моноид, поскольку использование списков иногда очень замедляет работу программы. Причина в том, что списки задействуют оператор конкатенации ++ в качестве реализации метода mappend, а использование данного оператора для присоединения чего-либо в конец списка заставляет программу существенно медлить, если список длинный.

В нашей функции gcd' журналирование происходит быстро, потому что добавление списка в конец в итоге выглядит следующим образом:

```
a ++ (b ++ (c ++ (d ++ (e ++ f))))
```

Списки – это структура данных, построение которой происходит слева направо, и это эффективно, поскольку мы сначала полностью строим левую часть списка и только потом добавляем более длинный список справа. Но если мы невнимательны, то использование монады `Writer` может вызывать присоединение списков, которое выглядит следующим образом:

```
((((a ++ b) ++ c) ++ d) ++ e) ++ f
```

Здесь связывание происходит в направлении налево, а не направо. Это неэффективно, поскольку каждый раз, когда функция хочет добавить правую часть к левой, она должна построить левую часть полностью, с самого начала!

Следующая функция работает аналогично функции `gcd'`, но производит журналирование в обратном порядке. Сначала она создаёт журнал для остальной части процедуры, а затем добавляет текущий шаг к концу журнала.

```
import Control.Monad.Writer

gcdReverse :: Int -> Int -> Writer [String] Int
gcdReverse a b
  | b == 0 = do
     tell ["Закончили: " ++ show a]
     return a
  | otherwise = do
     result <- gcdReverse b (a `mod` b)
     tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
     return result
```

Сначала она производит рекурсивный вызов и привязывает его значение к значению `result`. Затем добавляет текущий шаг в журнал, но текущий попадает в конец журнала, который был произведен посредством рекурсивного вызова. В заключение функция возвращает результат рекурсии как окончательный. Вот она в действии:

```
ghci> mapM_ putStrLn $ snd $ runWriter (gcdReverse 8 3)
Закончили: 1
2 mod 1 = 0
3 mod 2 = 1
```

```
8 mod 3 = 2
```

Она неэффективна, поскольку производит ассоциацию вызовов оператора ++ влево, вместо того чтобы делать это вправо.

Разностные списки

Поскольку списки иногда могут быть неэффективными при добавлении подобным образом, лучше использовать структуру данных, которая всегда поддерживает эффективное добавление. Одной из таких структур являются *разностные списки*. Разностный список аналогичен обычному списку, только он является функцией, которая принимает список и присоединяет к нему другой список спереди. Разностным списком, эквивалентным списку `[1,2,3]`, была бы функция `\xs -> [1,2,3] ++ xs`. Обычным пустым списком является значение `[]`, тогда как пустым разностным списком является функция `\xs -> [] ++ xs`.

Прелесть разностных списков заключается в том, что они поддерживают эффективную конкатенацию. Когда мы «склеиваем» два списка с помощью оператора ++, приходится проходить первый список (левый операнд) до конца и затем добавлять другой.

```
f `append` g = \xs -> f (g xs)
```

Вспомните: `f` и `g` – это функции, которые принимают списки и добавляют что-либо в их начало. Так, например, если `f` – это функция `("соба"++)` – просто другой способ записи `\xs -> "dog" ++ xs`, а `g` – это функция `("чатина"++)`, то ``f `append` g`` создаёт новую функцию, которая аналогична следующей записи:

```
\xs -> "соба" ++ ("чатина" ++ xs)
```

Мы соединили два разностных списка, просто создав новую функцию, которая сначала применяет один разностный список к какому-то одному списку, а затем к другому.

Давайте создадим обёртку `newtype` для разностных списков, чтобы мы легко могли сделать для них экземпляры класса `Monoid`:

```
newtype DiffList a = DiffList { getDiffList :: [a] -> [a] }
```

Оборачиваемым нами типом является тип `[a] -> [a]`, поскольку разностный список – это просто функция, которая принимает список и возвращает другой список. Преобразовывать обычные списки в разностные и обратно просто:

```
toDiffList :: [a] -> DiffList a
toDiffList xs = DiffList (xs++)

fromDiffList :: DiffList a -> [a]
fromDiffList (DiffList f) = f []
```

Чтобы превратить обычный список в разностный, мы просто делаем то же, что делали ранее, превращая его в функцию, которая добавляет его в начало другого списка. Поскольку разностный список – это функция, добавляющая нечто в начало другого списка, то если мы просто хотим получить это нечто, мы применяем функцию к пустому списку!

Вот экземпляр класса `Monoid`:

```
instance Monoid (DiffList a) where
   mempty = DiffList (\xs -> [] ++ xs)
   (DiffList f) `mappend` (DiffList g) = DiffList (\xs -> f (g xs))
```

Обратите внимание, что для разностных списков метод `mempty` – это просто функция `id`, а метод `mappend` на самом деле – всего лишь композиция функций. Посмотрим, сработает ли это:

```
ghci> fromDiffList (toDiffList [1,2,3,4] `mappend` toDiffList [1,2,3])
[1,2,3,4,1,2,3]
```

Превосходно! Теперь мы можем повысить эффективность нашей функции `gcdReverse`, сделав так, чтобы она использовала разностные списки вместо обычных:

```
import Control.Monad.Writer

gcdReverse :: Int -> Int -> Writer (DiffList String) Int
gcdReverse a b
   | b == 0 = do
      tell (toDiffList ["Закончили: " ++ show a])
      return a
   | otherwise = do
      result <- gcdReverse b (a `mod` b)
```

```
    tell (toDiffList [show a ++ " mod " ++ show b ++ " = "
                      ++ show (a `mod` b)])
    return result
```

Нам всего лишь нужно было изменить тип моноида с [String] на DiffList String, а затем при использовании функции tell преобразовать обычные списки в разностные с помощью функции toDiffList. Давайте посмотрим, правильно ли соберётся журнал:

```
ghci> mapM_ putStrLn . fromDiffList . snd . runWriter $ gcdReverse 110 34
Закончили: 2
8 mod 2 = 0
34 mod 8 = 2
110 mod 34 = 8
```

Мы выполняем вызов выражения gcdReverse 110 34, затем используем функцию runWriter, чтобы развернуть его результат из newtype, потом применяем к нему функцию snd, чтобы просто получить журнал, далее – функцию fromDiffList, чтобы преобразовать его в обычный список, и в заключение выводим его записи на экран.

Сравнение производительности

Чтобы почувствовать, насколько разностные списки могут улучшить вашу производительность, рассмотрите следующую функцию. Она просто в обратном направлении считает от некоторого числа до нуля, но производит записи в журнал в обратном порядке, как функция gcdReverse, чтобы числа в журнале на самом деле считались в прямом направлении.

```
finalCountDown :: Int -> Writer (DiffList String) ()
finalCountDown 0 = tell (toDiffList ["0"])
finalCountDown x = do
   finalCountDown (x-1)
   tell (toDiffList [show x])
```

Если мы передаём ей значение 0, она просто записывает это значение в журнал. Для любого другого числа она сначала вычисляет предшествующее ему число в обратном направлении до 0, а затем добавляет это число в конец журнала. Поэтому если мы применим функцию finalCountDown к значению 100, строка "100" будет идти в журнале последней.

Если вы загрузите эту функцию в интерпретатор GHCi и примените её к большому числу, например к значению 500 000, то увидите, что она быстро начинает счёт от 0 и далее:

```
ghci> mapM_ putStrLn . fromDiffList .snd . runWriter $ finalCountDown 500000
0
1
2
...
```

Однако если вы измените её, чтобы она использовала обычные списки вместо разностных, например, так:

```
finalCountDown :: Int -> Writer [String] ()
finalCountDown 0 = tell ["0"]
finalCountDown x = do
   finalCountDown (x-1)
   tell [show x]
```

а затем скажете интерпретатору GHCi, чтобы он начал отсчёт:

```
ghci> mapM_ putStrLn . snd . runWriter $ finalCountDown 500000
```

вы увидите, что вычисления идут очень медленно.

Конечно же, это ненаучный и неточный способ проверять скорость ваших программ. Однако мы могли видеть, что в этом случае использование разностных списков начинает выдавать результаты незамедлительно, тогда как использование обычных занимает нескончаемо долгое время.

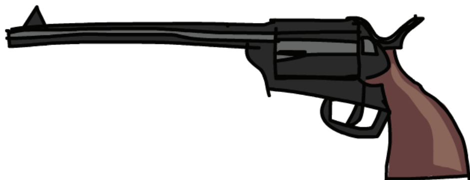
Ну, теперь в вашей голове наверняка засела песня «Final Countdown» группы Europe. Балдейте!

Монада Reader? Тьфу, опять эти шуточки!

В главе 11 вы видели, что тип функции (->) r является экземпляром класса Functor. Отображение функции g с помощью функции f создаёт функцию, которая принимает то же, что и g, применяет к этому g, а затем применяет к результату f. В общем, мы создаём новую функцию, которая похожа на g, только перед возвращением своего результата также применяет к этому результату f. Вот пример:

```
ghci> let f = (*5)
ghci> let g = (+3)
ghci> (fmap f g) 8
55
```

Вы также видели, что функции являются аппликативными функторами. Они позволяют нам оперировать окончательными результатами функций так, как если бы у нас уже были их результаты. И снова пример:



```
ghci> let f = (+) <$> (*2) <*> (+10)
ghci> f 3
19
```

Выражение `(+) <$> (*2) <*> (+10)` создаёт функцию, которая принимает число, передаёт это число функциям `(*2)` и `(+10)`, а затем складывает результаты. К примеру, если мы применим эту функцию к `3`, она применит к `3` и `(*2)`, и `(+10)`, возвращая `6` и `13`. Затем она вызовет операцию (+) со значениями `6` и `13`, и результатом станет `19`.

Функции в качестве монад

Тип функции `(->) r` является не только функтором и аппликативным функтором, но также и монадой. Как и другие монадические значения, которые вы встречали до сих пор, функцию можно рассматривать как значение с контекстом. Контекстом для функции является то, что это значение ещё не представлено и нам необходимо применить эту функцию к чему-либо, чтобы получить её результат.

Поскольку вы уже знакомы с тем, как функции работают в качестве функторов и аппликативных функторов, давайте прямо сейчас взглянем, как выглядит их экземпляр для класса `Monad`. Он расположен в модуле `Control.Monad.Instances` и похож на нечто подобное:

```
instance Monad ((->) r) where
   return x = \_ -> x
```

```
   h >>= f = \w -> f (h w) w
```

Вы видели, как функция pure реализована для функций, а функция return – в значительной степени то же самое, что и pure. Она принимает значение и помещает его в минимальный контекст, который всегда содержит это значение в качестве своего результата. И единственный способ создать функцию, которая всегда возвращает определённое значение в качестве своего результата, – это заставить её совсем игнорировать свой параметр.

Реализация для операции >>= может выглядеть немного загадочно, но на самом деле она не так уж и сложна. Когда мы используем операцию >>= для передачи монадического значения функции, результатом всегда будет монадическое значение. Так что в данном случае, когда мы передаём функцию другой функции, результатом тоже будет функция. Вот почему результат начинается с анонимной функции.

Все реализации операции >>= до сих пор так или иначе отделяли результат от монадического значения, а затем применяли к этому результату функцию f. То же самое происходит и здесь. Чтобы получить результат из функции, нам необходимо применить её к чему-либо, поэтому мы используем здесь (h w), а затем применяем к этому f. Функция f возвращает монадическое значение, которое в нашем случае является функцией, поэтому мы применяем её также и к значению w.

Монада Reader

Если в данный момент вы не понимаете, как работает операция >>=, не беспокойтесь. Несколько примеров позволят вам убедиться, что это очень простая монада. Вот выражение do, которое её использует:

```
import Control.Monad.Instances

addStuff :: Int -> Int
addStuff = do
   a <- (*2)
   b <- (+10)
   return (a+b)
```

Это то же самое, что и аппликативное выражение, которое мы записали ранее, только теперь оно полагается на то, что функции

являются монадами. Выражение do всегда возвращает монадическое значение, и данное выражение ничем от него не отличается. Результатом этого монадического значения является функция. Она принимает число, затем к этому числу применяется функция (*2) и результат записывается в образец a. К тому же самому числу, к которому применялась функция (*2), применяется теперь уже функция (+10), и результат записывается в образец b. Функция return, как и в других монадах, не имеет никакого другого эффекта, кроме создания монадического значения, возвращающего некий результат. Она возвращает значение выражения (a+b) в качестве результата данной функции. Если мы протестируем её, то получим те же результаты, что и прежде:

```
ghci> addStuff 3
19
```

И функция (*2), и функция (+10) применяются в данном случае к числу 3. Выражение return (a+b) применяется тоже, но оно игнорирует это значение и всегда возвращает (a+b) в качестве результата. По этой причине функциональную монаду также называют *монадой-читателем*. Все функции читают из общего источника. Чтобы сделать это ещё очевиднее, мы можем переписать функцию addStuff вот так:

```
addStuff :: Int -> Int
addStuff x = let a = (*2) x
                 b = (+10) x
             in a+b
```

Вы видите, что монада-читатель позволяет нам обрабатывать функции как значения с контекстом. Мы можем действовать так, как будто уже знаем, что вернут функции. Суть в том, что монада-читатель «склеивает» функции в одну, а затем передаёт параметр этой функции всем тем, которые её составляют. Поэтому если у нас есть множество функций, каждой из которых недостаёт всего лишь одного параметра, и в конечном счёте они будут применены к одному и тому же, то мы можем использовать монаду-читатель, чтобы как бы извлечь их будущие результаты. А реализация операции >>= позаботится о том, чтобы всё это сработало.

Вкусные вычисления с состоянием

Haskell является чистым языком, и вследствие этого наши программы состоят из функций, которые не могут изменять какое бы то ни было глобальное состояние или переменные – они могут только производить какие-либо вычисления и возвращать результаты. На самом деле данное ограничение упрощает задачу обдумывания наших программ, освобождая нас от необходимости заботиться о том, какое значение имеет каждая переменная в определённый момент времени.

Тем не менее некоторые задачи по своей природе обладают состоянием, поскольку зависят от какого-то состояния, изменяющегося с течением времени. Хотя это не проблема для Haskell, такие вычисления могут быть немного утомительными для моделирования. Вот почему в языке Haskell есть монада `State`, благодаря которой решение задач с внутренним состоянием становится сущим пустяком – и в то же время остаётся красивым и чистым.

Когда мы разбирались со случайными числами в главе 9, то имели дело с функциями, которые в качестве параметра принимали генератор случайных чисел и возвращали случайное число и новый генератор случайных чисел. Если мы хотели сгенерировать несколько случайных чисел, нам всегда приходилось использовать генератор случайных чисел, который возвращала предыдущая функция вместе со своим результатом. Например, чтобы создать функцию, которая принимает значение типа `StdGen` и трижды «подбрасывает монету» на основе этого генератора, мы делали следующее:

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
```

```
let (firstCoin, newGen) = random gen
    (secondCoin, newGen') = random newGen
    (thirdCoin, newGen'') = random newGen'
in  (firstCoin, secondCoin, thirdCoin)
```

Эта функция принимает генератор `gen`, а затем вызов `random gen` возвращает значение типа `Bool` наряду с новым генератором. Для подбрасывания второй монеты мы используем новый генератор, и т. д.

В большинстве других языков нам не нужно было бы возвращать новый генератор вместе со случайным числом. Мы могли бы просто изменить имеющийся! Но поскольку Haskell является чистым языком, этого сделать нельзя, поэтому мы должны были взять какое-то состояние, создать из него результат и новое состояние, а затем использовать это новое состояние для генерации новых результатов.

Можно подумать, что для того, чтобы не иметь дела с вычислениями с состоянием вручную подобным образом, мы должны были бы отказаться от чистоты языка Haskell. К счастью, такую жертву приносить не нужно, так как существует специальная небольшая монада под названием `State`. Она превосходно справляется со всеми этими делами с состоянием, никоим образом не влияя на чистоту, благодаря которой программирование на языке Haskell настолько оригинально и изящно.

Вычисления с состоянием

Чтобы лучше продемонстрировать вычисления с внутренним состоянием, давайте просто возьмём и дадим им тип. Мы скажем, что вычисление с состоянием – это функция, которая принимает некое состояние и возвращает значение вместе с неким новым состоянием. Данная функция имеет следующий тип:

```
s -> (a, s)
```

Идентификатор `s` обозначает тип состояния; `a` – это результат вычислений с состоянием.

ПРИМЕЧАНИЕ. *В большинстве других языков присваивание значения может рассматриваться как вычисление с состоянием. Например, когда мы выполняем выражение `x = 5` в императивном*

языке, как правило, это присваивает переменной x значение 5, и в нём также в качестве выражения будет фигурировать значение 5. Если рассмотреть это действие с функциональной точки зрения, получается нечто вроде функции, принимающей состояние (то есть все переменные, которым ранее были присвоены значения) и возвращающей результат (в данном случае 5) и новое состояние, которое представляло бы собой все предшествующие соответствия переменных значениям плюс переменную с недавно присвоенным значением.

Это вычисление с состоянием – функцию, которая принимает состояние и возвращает результат и новое состояние – также можно воспринимать как значение с контекстом. Действительным значением является результат, тогда как контекстом является то, что мы должны предоставить некое исходное состояние, чтобы фактически получить этот результат, и то, что помимо результата мы также получаем новое состояние.

Стеки и чебуреки

Предположим, мы хотим смоделировать стек. *Стек* – это структура данных, которая содержит набор элементов и поддерживает ровно две операции:

- проталкивание элемента в стек (добавляет элемент на верхушку стека);
- выталкивание элемента из стека (удаляет самый верхний элемент из стека).

Для представления нашего стека будем использовать список, «голова» которого действует как вершина стека. Чтобы решить эту задачу, создадим две функции:

- функция `pop` будет принимать стек, выталкивать один элемент и возвращать его в качестве результата. Кроме того, она возвращает новый стек без вытолкнутого элемента;
- функция `push` будет принимать элемент и стек, а затем проталкивать этот элемент в стек. В качестве результата она будет возвращать значение `()` вместе с новым стеком.

Вот используемые функции:

```
type Stack = [Int]

pop :: Stack -> (Int, Stack)
```

```
pop (x:xs) = (x, xs)

push :: Int -> Stack -> ((), Stack)
push a xs = ((), a:xs)
```

При проталкивании в стек в качестве результата мы использовали значение `()`, поскольку проталкивание элемента на вершину стека не несёт какого-либо существенного результирующего значения – его основная задача заключается в изменении стека. Если мы применим только первый параметр функции `push`, мы получим вычисление с состоянием. Функция `pop` уже является вычислением с состоянием вследствие своего типа.

Давайте напишем небольшой кусок кода для симуляции стека, используя эти функции. Мы возьмём стек, протолкнем в него значение `3`, а затем вытолкнем два элемента просто ради забавы. Вот оно:

```
stackManip :: Stack -> (Int, Stack)
stackManip stack = let
   ((), newStack1) = push 3 stack
   (a , newStack2) = pop newStack1
   in pop newStack2
```

Мы принимаем стек, а затем выполняем выражение `push 3 stack`, что даёт в результате кортеж. Первой частью кортежа является значение `()`, а второй частью является новый стек, который мы называем `newStack1`. Затем мы выталкиваем число из `newStack1`, что даёт в результате число `a` (равно `3`), которое мы протолкнули, и новый стек, названный нами `newStack2`. Затем мы выталкиваем число из `newStack2` и получаем число и новый стек. Мы возвращаем кортеж с этим числом и новым стеком. Давайте попробуем:

```
ghci> stackManip [5,8,2,1]
(5,[8,2,1])
```

Результат равен `5`, а новый стек – `[8,2,1]`. Обратите внимание, как функция `stackManip` сама является вычислением с состоянием. Мы взяли несколько вычислений с состоянием и как бы «склеили» их вместе. Хм-м, звучит знакомо.

Предшествующий код функции `stackManip` несколько громоздок, потому как мы вручную передаём состояние каждому вычислению

с состоянием, сохраняем его, а затем передаём следующему. Не лучше ли было бы, если б вместо того, чтобы передавать стек каждой функции вручную, мы написали что-то вроде следующего:

```
stackManip = do
   push 3
   a <- pop
   pop
```

Ла-адно, монада State позволит нам делать именно это!.. С её помощью мы сможем брать вычисления с состоянием, подобные этим, и использовать их без необходимости управлять состоянием вручную.

Монада State

Модуль Control.Monad.State предоставляет тип newtype, который оборачивает вычисления с состоянием. Вот его определение:

```
newtype State s a = State { runState :: s -> (a, s) }
```

Тип State s a – это тип вычисления с состоянием, которое манипулирует состоянием типа s и имеет результат типа a.

Как и модуль Control.Monad.Writer, модуль Control.Monad.State не экспортирует свой конструктор значения. Если вы хотите взять вычисление с состоянием и обернуть его в newtype State, используйте функцию state, которая делает то же самое, что делал бы конструктор State.

Теперь, когда вы увидели, в чём заключается суть вычислений с состоянием и как их можно даже воспринимать в виде значений с контекстами, давайте рассмотрим их экземпляр класса Monad:

```
instance Monad (State s) where
   return x = State $ \s -> (x, s)
   (State h) >>= f = State $ \s -> let (a, newState) = h s
                                       (State g) = f a
                                   in  g newState
```

Наша цель использования функции return состоит в том, чтобы взять значение и создать вычисление с состоянием, которое всегда содержит это значение в качестве своего результата. Поэтому мы просто создаём анонимную функцию \s -> (x, s). Мы

всегда представляем значение `x` в качестве результата вычисления с состоянием, а состояние остаётся неизменным, так как функция `return` должна помещать значение в минимальный контекст. Потому функция `return` создаст вычисление с состоянием, которое представляет определённое значение в качестве результата, а состояние сохраняет неизменным.

А что насчёт операции `>>=`? Ну что ж, результатом передачи вычисления с состоянием функции с помощью операции `>>=` должно быть вычисление с состоянием, верно? Поэтому мы начинаем с обёртки `newtype State`, а затем вызываем анонимную функцию. Эта анонимная функция будет нашим новым вычислением с состоянием. Но что же в ней происходит? Нам каким-то образом нужно извлечь значение результата из первого вычисления с состоянием. Поскольку прямо сейчас мы находимся в вычислении с состоянием, то можем передать вычислению с состоянием `h` наше текущее состояние `s`, что в результате даёт пару из результата и нового состояния: `(a, newState)`.

До сих пор каждый раз, когда мы реализовывали операцию `>>=`, сразу же после извлечения результата из монадического значения мы применяли к нему функцию `f`, чтобы получить новое монадическое значение. В случае с монадой `Writer` после того, как это сделано и получено новое монадическое значение, нам по-прежнему нужно позаботиться о контексте, объединив прежнее и новое моноидные значения с помощью функции `mappend`. Здесь мы выполняем вызов выражения `f a` и получаем новое вычисление с состоянием `g`. Теперь, когда у нас есть новое вычисление с состоянием и новое состояние (известное под именем `newState`), мы просто применяем это вычисление с состоянием `g` к `newState`. Результатом является кортеж из окончательного результата и окончательного состояния!

Итак, при использовании операции `>>=` мы как бы «склеиваем» друг с другом два вычисления, обладающих состоянием. Второе вычисление скрыто внутри функции, которая принимает результат

предыдущего вычисления. Поскольку функции pop и push уже являются вычислениями с состоянием, легко обернуть их в обёртку State:

```
import Control.Monad.State

pop :: State Stack Int
pop = state $ \(x:xs) -> (x, xs)

push :: Int -> State Stack ()
push a = state $ \xs -> ((), a:xs)
```

Обратите внимание, как мы задействовали функцию state, чтобы обернуть функцию в конструктор newtype State, не прибегая к использованию конструктора значения State напрямую.

Функция pop – уже вычисление с состоянием, а функция push принимает значение типа Int и возвращает вычисление с состоянием. Теперь мы можем переписать наш предыдущий пример проталкивания числа 3 в стек и выталкивания двух чисел подобным образом:

```
import Control.Monad.State

stackManip :: State Stack Int
stackManip = do
   push 3
   a <- pop
   pop
```

Видите, как мы «склеили» проталкивание и два выталкивания в одно вычисление с состоянием? Разворачивая его из обёртки newtype, мы получаем функцию, которой можем предоставить некое исходное состояние:

```
ghci> runState stackManip [5,8,2,1]
(5,[8,2,1])
```

Нам не требовалось привязывать второй вызов функции pop к образцу a, потому что мы вовсе не использовали этот образец. Значит, это можно было записать вот так:

```
stackManip :: State Stack Int
stackManip = do
```

```
   push 3
   pop
   pop
```

Очень круто! Но что если мы хотим сделать что-нибудь посложнее? Скажем, вытолкнуть из стека одно число, и если это число равно 5, просто протолкнуть его обратно в стек и остановиться. Но если число *не* равно 5, вместо этого протолкнуть обратно 3 и 8. Вот он код:

```
stackStuff :: State Stack ()
stackStuff = do
   a <- pop
   if a == 5
      then push 5
      else do
         push 3
         push 8
```

Довольно простое решение. Давайте выполним этот код с исходным стеком:

```
ghci> runState stackStuff [9,0,2,1,0]
((),[8,3,0,2,1,0])
```

Вспомните, что выражения do возвращают в результате монадические значения, и при использовании монады State одно выражение do является также функцией с состоянием. Поскольку функции stackManip и stackStuff являются обычными вычислениями с состоянием, мы можем «склеивать» их вместе, чтобы производить дальнейшие вычисления с состоянием:

```
moreStack :: State Stack ()
moreStack = do
   a <- stackManip
   if a == 100
      then stackStuff
      else return ()
```

Если результат функции stackManip при использовании текущего стека равен 100, мы вызываем функцию stackStuff; в противном случае ничего не делаем. Вызов return () просто сохраняет состояние как есть и ничего не делает.

Получение и установка состояния

Модуль `Control.Monad.State` определяет класс типов под названием `MonadState`, в котором присутствуют две весьма полезные функции: `get` и `put`. Для монады `State` функция `get` реализована вот так:

```
get = state $ \s -> (s, s)
```

Она просто берёт текущее состояние и представляет его в качестве результата.

Функция `put` принимает некоторое состояние и создаёт функцию с состоянием, которая заменяет им текущее состояние:

```
put newState = state $ \s -> ((), newState)
```

Поэтому, используя их, мы можем посмотреть, чему равен текущий стек, либо полностью заменить его другим стеком – например, так:

```
stackyStack :: State Stack ()
stackyStack = do
   stackNow <- get
   if stackNow == [1,2,3]
      then put [8,3,1]
      else put [9,2,1]
```

Также можно использовать функции `get` и `put`, чтобы реализовать функции `pop` и `push`. Вот определение функции `pop`:

```
pop :: State Stack Int
pop = do
   (x:xs) <- get
   put xs
   return x
```

Мы используем функцию `get`, чтобы получить весь стек, а затем – функцию `put`, чтобы новым состоянием были все элементы за исключением верхнего. После чего прибегаем к функции `return`, чтобы представить значение `x` в качестве результата.

Вот определение функции `push`, реализованной с использованием `get` и `put`:

```
push :: Int -> State Stack ()
push x = do
```

```
    xs <- get
    put (x:xs)
```

Мы просто используем функцию get, чтобы получить текущее состояние, и функцию put, чтобы установить состояние в такое же, как наш стек с элементом x на вершине.

Стоит проверить, каким был бы тип операции >>=, если бы она работала только со значениями монады State:

```
(>>=) :: State s a -> (a -> State s b) -> State s b
```

Видите, как тип состояния s остаётся тем же, но тип результата может изменяться с a на b? Это означает, что мы можем «склеивать» вместе несколько вычислений с состоянием, результаты которых имеют различные типы, но тип состояния должен оставаться тем же. Почему же так?.. Ну, например, для типа Maybe операция >>= имеет такой тип:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

Логично, что сама монада Maybe не изменяется. Не имело бы смысла использовать операцию >>= между двумя разными монадами. Для монады State монадой на самом деле является State s, так что если бы этот тип s был различным, мы использовали бы операцию >>= между двумя разными монадами.

Случайность и монада State

В начале этого раздела мы говорили о том, что генерация случайных чисел может иногда быть неуклюжей. Каждая функция, использующая случайность, принимает генератор и возвращает случайное число вместе с новым генератором, который должен затем быть использован вместо прежнего, если нам нужно сгенерировать ещё одно случайное число. Монада State намного упрощает эти действия.

Функция random из модуля System.Random имеет следующий тип:

```
random :: (RandomGen g, Random a) => g -> (a, g)
```

Это значит, что она берёт генератор случайных чисел и производит случайное число вместе с новым генератором. Нам видно, что это вычисление с состоянием, поэтому мы можем обернуть его в конструктор newtype State при помощи функции state, а затем

использовать его в качестве монадического значения, чтобы передача состояния обрабатывалась за нас:

```
import System.Random
import Control.Monad.State

randomSt :: (RandomGen g, Random a) => State g a
randomSt = state random
```

Поэтому теперь, если мы хотим подбросить три монеты (`True` – это «решка», а `False` – «орёл»), то просто делаем следующее:

```
import System.Random
import Control.Monad.State

threeCoins :: State StdGen (Bool, Bool, Bool)
threeCoins = do
   a <- randomSt
   b <- randomSt
   c <- randomSt
   return (a, b, c)
```

Функция `threeCoins` – это теперь вычисление с состоянием, и после получения исходного генератора случайных чисел она передаёт этот генератор в первый вызов функции `randomSt`, которая производит число и новый генератор, передаваемый следующей функции, и т. д. Мы используем выражение `return (a, b, c)`, чтобы представить значение `(a, b, c)` как результат, не изменяя самый последний генератор. Давайте попробуем:

```
ghci> runState threeCoins (mkStdGen 33)
((True,False,True),680029187 2103410263)
```

Теперь выполнение всего, что требует сохранения некоторого состояния в промежутках между шагами, в самом деле стало доставлять значительно меньше хлопот!

Свет мой, Error, скажи, да всю правду доложи

К этому времени вы знаете, что монада `Maybe` используется, чтобы добавить к значениям контекст возможной неудачи. Значением

может быть Just <*нечто*> либо Nothing. Как бы это ни было полезно, всё, что нам известно, когда у нас есть значение Nothing, – это состоявшийся факт некоей неудачи: туда не втиснуть больше информации, сообщающей нам, что именно произошло.

И тип Either e a позволяет нам включать контекст возможной неудачи в наши значения. С его помощью тоже можно прикреплять значения к неудаче, чтобы они могли описать, что именно пошло не так, либо предоставить другую полезную информацию относительно ошибки. Значение типа Either e a может быть либо значением Right (правильный ответ и успех) либо значением Left (неудача). Вот пример:

```
ghci> :t Right 4
Right 4 :: (Num t) => Either a t
ghci> :t Left "ошибка нехватки сыра"
Left "ошибка нехватки сыра" :: Either [Char] b
```

Это практически всего лишь улучшенный тип Maybe, поэтому имеет смысл, чтобы он был монадой. Он может рассматриваться и как значение с добавленным контекстом возможной неудачи, только теперь при возникновении ошибки также имеется прикреплённое значение.

Его экземпляр класса Monad похож на экземпляр для типа Maybe и может быть обнаружен в модуле Control.Monad.Error[1]:

```
instance (Error e) => Monad (Either e) where
   return x = Right x
   Right x >>= f = f x
   Left err >>= f = Left err
   fail msg = Left (strMsg msg)
```

Функция return, как и всегда, принимает значение и помещает его в минимальный контекст по умолчанию. Она оборачивает наше значение в конструктор Right, потому что мы используем его для представления успешных вычислений, где присутствует

[1] Если версия пакетов языка Haskell base и mtl, установленных в вашей системе, выше соответственно 4.3.1.0 и 2.0.1.0, вам нужно импортировать модуль Control.Monad.Error в ваш скрипт или Control.Monad.Instances в интерпретатор GHCi, перед тем как вы сможете использовать функции экземпляра класса Monad для типа Either. Это связано с тем, что в этих версиях пакетов объявления экземпляров были перенесены в модуль Control.Monad.Instances. – *Прим. перев.*

результат. Это очень похоже на определение метода return для типа Maybe.

Оператор >>= проверяет два возможных случая: Left и Right. В случае Right к значению внутри него применяется функция f, подобно случаю Just, где к его содержимому просто применяется функция. В случае ошибки сохраняется значение Left вместе с его содержимым, которое описывает неудачу.

Экземпляр класса Monad для типа Either e имеет дополнительное требование. Тип значения, содержащегося в Left, – тот, что указан параметром типа e, – должен быть экземпляром класса Error. Класс Error предназначен для типов, значения которых могут действовать как сообщения об ошибках. Он определяет функцию strMsg, которая принимает ошибку в виде строки и возвращает такое значение. Хороший пример экземпляра Error – тип String! В случае со String функция strMsg просто возвращает строку, которую она получила:

```
ghci> :t strMsg
strMsg :: (Error a) => String -> a
ghci> strMsg "Бум!" :: String
"Бум!"
```

Но поскольку при использовании типа Either для описания ошибки мы обычно задействуем тип String, нам не нужно об этом сильно беспокоиться. Когда сопоставление с образцом терпит неудачу в нотации do, то для оповещения об этой неудаче используется значение Left.

Вот несколько практических примеров:

```
ghci> Left "Бум" >>= \x ->return (x+1)
Left "Бум"
ghci> Left "Бум " >>= \x -> Left "нет пути!"
Left "Бум "
ghci> Right 100 >>= \x -> Left "нет пути!"
Left "нет пути!"
```

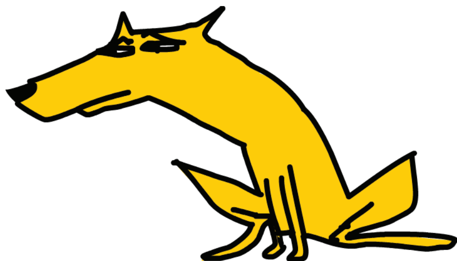
Когда мы используем операцию >>=, чтобы передать функции значение Left, функция игнорируется и возвращается идентичное значение Left. Когда мы передаём функции значение Right, функция применяется к тому, что находится внутри, но в данном случае эта функция всё равно произвела значение Left!

Использование монады `Error` очень похоже на использование монады `Maybe`.

> **ПРИМЕЧАНИЕ.** *В предыдущей главе мы использовали монадические аспекты типа `Maybe` для симуляции приземления птиц на балансировочный шест канатоходца. В качестве упражнения вы можете переписать код с использованием монады `Error`, чтобы, когда канатоходец поскальзывался и падал, вы запоминали, сколько птиц было на каждой стороне шеста в момент падения.*

Некоторые полезные монадические функции

В этом разделе мы изучим несколько функций, которые работают с монадическими значениями либо возвращают монадические значения в качестве своих результатов (или и то, и другое!). Такие функции обычно называют *монадическими*. В то время как некоторые из них будут для вас совершенно новыми, другие являются монадическими аналогами функций, с которыми вы уже знакомы – например, `filter` и `foldl`. Ниже мы рассмотрим функции `liftM`, `join`, `filterM` и `foldM`.



liftM и компания

Когда мы начали своё путешествие на верхушку Горы Монад, мы сначала посмотрели на *функторы*, предназначенные для сущностей, которые можно отображать. Затем рассмотрели улучшенные функторы – *аппликативные*, которые позволяют нам применять обычные функции между несколькими аппликативными значениями, а также брать обычное значение и помещать его в некоторый контекст по умолчанию. Наконец, мы ввели *монады* как улучшенные аппликативные функторы, которые добавляют возможность тем или иным образом передавать эти значения с контекстом в обычные функции.

Итак, каждая монада – это аппликативный функтор, а каждый аппликативный функтор – это функтор. Класс типов `Applicative` имеет

такое ограничение класса, ввиду которого наш тип должен иметь экземпляр класса `Functor`, прежде чем мы сможем сделать для него экземпляр класса `Applicative`. Класс `Monad` должен иметь то же самое ограничение для класса `Applicative`, поскольку каждая монада является аппликативным функтором – однако не имеет, потому что класс типов `Monad` был введён в язык Haskell задолго до класса `Applicative`.

Но хотя каждая монада – функтор, нам не нужно полагаться на то, что у неё есть экземпляр для класса `Functor`, в силу наличия функции `liftM`. Функция `liftM` берёт функцию и монадическое значение и отображает монадическое значение с помощью функции. Это почти одно и то же, что и функция `fmap`! Вот тип функции `liftM`:

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
```

Сравните с типом функции `fmap`:

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

Если экземпляры классов `Functor` и `Monad` для типа подчиняются законам функторов и монад, между этими двумя нет никакой разницы (и все монады, которые мы до сих пор встречали, подчиняются обоим). Это примерно как функции `pure` и `return`, делающие одно и то же, – только одна имеет ограничение класса `Applicative`, тогда как другая имеет ограничение `Monad`.

Давайте опробуем функцию `liftM`:

```
ghci> liftM (*3) (Just 8)
Just 24
ghci> fmap (*3) (Just 8)
Just 24
ghci> runWriter $ liftM not $ Writer (True, "горох")
(False,"горох")
ghci> runWriter $ fmap not $ Writer (True, "горох")
(False,"горох")
ghci> runState (liftM (+100) pop) [1,2,3,4]
(101,[2,3,4])
ghci> runState (fmap (+100) pop) [1,2,3,4]
(101,[2,3,4])
```

Вы уже довольно хорошо знаете, как функция `fmap` работает со значениями типа `Maybe`. И функция `liftM` делает то же самое. При использовании со значениями типа `Writer` функция отобра-

жает первый компонент кортежа, который является результатом. Выполнение функций fmap или liftM с вычислением, имеющим состояние, даёт в результате другое вычисление с состоянием, но его окончательный результат изменяется добавленной функцией. Если бы мы не отобразили функцию pop с помощью (+100) перед тем, как выполнить её, она бы вернула (1, [2,3,4]).

Вот как реализована функция liftM:

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = m >>= (\x -> return (f x))
```

Или с использованием нотации do:

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = do
   x <- m
   return (f x)
```

Мы передаём монадическое значение m в функцию, а затем применяем функцию к его результату, прежде чем поместить его обратно в контекст по умолчанию. Ввиду монадических законов гарантируется, что функция не изменит контекст; она изменяет лишь результат, который представляет монадическое значение.

Вы видите, что функция liftM реализована совсем не ссылаясь на класс типов Functor. Значит, мы можем реализовать функцию fmap (или liftM – называйте, как пожелаете), используя лишь те блага, которые предоставляют нам монады. Благодаря этому можно заключить, что монады, по крайней мере, настолько же сильны, насколько и функторы.

Класс типов Applicative позволяет нам применять функции между значениями с контекстами, как если бы они были обычными значениями, вот так:

```
ghci> (+) <$> Just 3 <*> Just 5
Just 8
ghci> (+) <$> Just 3 <*> Nothing
Nothing
```

Использование этого аппликативного стиля всё упрощает. Операция <$> – это просто функция fmap, а операция <*> – это функция из класса типов Applicative, которая имеет следующий тип:

```
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

Так что это вроде fmap, только сама функция находится в контексте. Нам нужно каким-то образом извлечь её из контекста и с её помощью отобразить значение f a, а затем вновь собрать контекст. Поскольку все функции в языке Haskell по умолчанию каррированы, мы можем использовать сочетание из операций <$> и <*> между аппликативными значениями, чтобы применять функции, принимающие несколько параметров.

Однако, оказывается, как и функция fmap, операция <*> тоже может быть реализована, используя лишь то, что даёт нам класс типов Monad. Функция ap, по существу, – это <*>, только с ограничением Monad, а не Applicative. Вот её определение:

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap mf m = do
   f <- mf
   x <- m
   return (fx)
```

Функция ap – монадическое значение, результат которого – функция. Поскольку функция, как и значение, находится в контексте, мы берём функцию из контекста и называем её f, затем берём значение и называем его x, и, в конце концов, применяем функцию к значению и представляем это в качестве результата. Вот быстрая демонстрация:

```
ghci> Just (+3) <*> Just 4
Just 7
ghci> Just (+3) `ap` Just 4
Just 7
ghci> [(+1),(+2),(+3)] <*> [10,11]
[11,12,12,13,13,14]
ghci> [(+1),(+2),(+3)] `ap` [10,11]
[11,12,12,13,13,14]
```

Теперь нам видно, что монады настолько же сильны, насколько и аппликативные функторы, потому что мы можем использовать методы класса Monad для реализации функций из класса Applicative. На самом деле, когда обнаруживается, что определённый тип является монадой, зачастую сначала записывают экземпляр класса Monad, а затем создают экземпляр класса Applicative, просто говоря,

что функция `pure` – это `return`, а операция `<*>` – это `ap`. Аналогичным образом, если у вас уже есть экземпляр класса `Monad` для чего-либо, вы можете сделать для него экземпляр класса `Functor`, просто говоря, что функция `fmap` – это `liftM`.

Функция `liftA2` весьма удобна для применения функции между двумя аппликативными значениями. Она определена вот так:

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f x y = f <$> x <*> y
```

Функция `liftM2` делает то же, но с использованием ограничения `Monad`. Есть также функции `liftM3`, `liftM4` и `liftM5`.

Вы увидели, что монады не менее сильны, чем функторы и аппликативные функторы – и, хотя все монады, по сути, являются функторами и аппликативными функторами, у них необязательно имеются экземпляры классов `Functor` и `Applicative`. Мы изучили монадические эквиваленты функций, которые используются функторами и аппликативными функторами.

Функция join

Есть кое-какая пища для размышления: если результат монадического значения – ещё одно монадическое значение (одно монадическое значение вложено в другое), можете ли вы «разгладить» их до одного лишь обычного монадического значения? Например, если у нас есть `Just (Just 9)`, можем ли мы превратить это в `Just 9`? Оказывается, что любое вложенное монадическое значение может быть разглажено, причём на самом деле это свойство уникально для монад. Для этого у нас есть функция `join`. Её тип таков:

```
join :: (Monad m) => m (m a) -> m a
```

Значит, функция `join` принимает монадическое значение в монадическом значении и отдаёт нам просто монадическое значение; другими словами, она его разглаживает. Вот она с некоторыми значениями типа `Maybe`:

```
ghci> join (Just (Just 9))
Just 9
ghci> join (Just Nothing)
Nothing
ghci> join Nothing
Nothing
```

В первой строке – успешное вычисление как результат успешного вычисления, поэтому они оба просто соединены в одно большое успешное вычисление. Во второй строке значение Nothing представлено как результат значения Just. Всякий раз, когда мы раньше имели дело со значениями Maybe и хотели объединить несколько этих значений – будь то с использованием операций <*> или >>= – все они должны были быть значениями конструктора Just, чтобы результатом стало значение Just. Если на пути возникала хоть одна неудача, то и результатом являлась неудача; нечто аналогичное происходит и здесь. В третьей строке мы пытаемся разгладить то, что возникло вследствие неудачи, поэтому результат – также неудача.

Разглаживание списков осуществляется довольно интуитивно:

```
ghci> join [[1,2,3],[4,5,6]]
[1,2,3,4,5,6]
```

Как вы можете видеть, функция join для списков – это просто concat. Чтобы разгладить значение монады Writer, результат которого сам является значением монады Writer, нам нужно объединить моноидное значение с помощью функции mappend:

```
ghci> runWriter $ join (Writer (Writer (1, "aaa"), "bbb"))
(1,"bbbaaa")
```

Внешнее моноидное значение "bbb" идёт первым, затем к нему конкатенируется строка "aaa". На интуитивном уровне, когда вы хотите проверить результат значения типа Writer, сначала вам нужно записать его моноидное значение в журнал, и только потом вы можете посмотреть, что находится внутри него.

Разглаживание значений монады Either очень похоже на разглаживание значений монады Maybe:

```
ghci> join (Right (Right 9)) :: Either String Int
Right 9
ghci> join (Right (Left "ошибка")) :: Either String Int
Left "ошибка"
ghci> join (Left "ошибка") :: Either String Int
Left "ошибка"
```

Если применить функцию join к вычислению с состоянием, результат которого является вычислением с состоянием, то результатом будет вычисление с состоянием, которое сначала выполняет

внешнее вычисление с состоянием, а затем результирующее. Взгляните, как это работает:

```
ghci> runState (join (state $ \s -> (push 10, 1:2:s))) [0,0,0]
((),[10,1,2,0,0,0])
```

Здесь анонимная функция принимает состояние, помещает 2 и 1 в стек и представляет `push 10` как свой результат. Поэтому когда всё это разглаживается с помощью функции `join`, а затем выполняется, всё это выражение сначала помещает значения 2 и 1 в стек, а затем выполняется выражение `push 10`, проталкивая число 10 на верхушку.

Реализация для функции `join` такова:

```
join :: (Monad m) => m (m a) -> m a
join mm = do
   m <- mm
   m
```

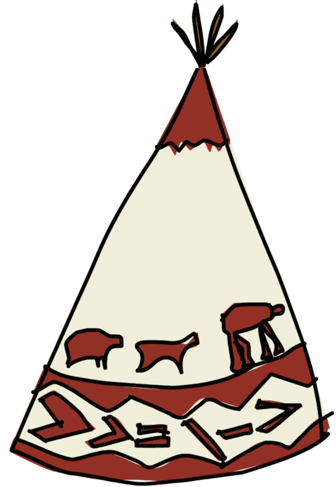
Поскольку результат `mm` является монадическим значением, мы берём этот результат, а затем просто помещаем его на его собственную строку, потому что это и есть монадическое значение. Трюк здесь в том, что когда мы вызываем выражение `m <- mm`, контекст монады, в которой мы находимся, будет обработан. Вот почему, например, значения типа `Maybe` дают в результате значения `Just`, только если и внешнее, и внутреннее значения являются значениями `Just`. Вот как это выглядело бы, если бы значение `mm` было заранее установлено в `Just (Just 8)`:

```
joinedMaybes :: Maybe Int
joinedMaybes = do
   m <- Just (Just 8)
   m
```

Наверное, самое интересное в функции `join` – то, что для любой монады передача монадического значения в функцию с помощью операции `>>=` представляет собой то же самое, что и просто отображение значения с помощью этой функции, а затем использование функции `join` для разглаживания результирующего вложенного монадического значения! Другими словами, выражение `m >>= f` – всегда то же самое, что и `join (fmap f m)`. Если вдуматься, это имеет смысл.

При использовании операции `>>=` мы постоянно думаем, как передать монадическое значение функции, которая принимает

обычное значение, а возвращает монадическое. Если мы просто отобразим монадическое значение с помощью этой функции, то получим монадическое значение внутри монадического значения. Например, скажем, у нас есть `Just 9` и функция `\x -> Just (x+1)`. Если с помощью этой функции мы отобразим `Just 9`, у нас останется `Just (Just 10)`.



То, что выражение `m >>= f` всегда равно `join (fmap f m)`, очень полезно, если мы создаём свой собственный экземпляр класса `Monad` для некоего типа. Это связано с тем, что зачастую проще понять, как мы бы разгладили вложенное монадическое значение, чем понять, как реализовать операцию `>>=`.

Ещё интересно то, что функция `join` не может быть реализована, всего лишь используя функции, предоставляемые функторами и аппликативными функторами. Это приводит нас к заключению, что монады не просто *сопоставимы* по своей силе с функторами и аппликативными функторами – они на самом деле *сильнее*, потому что с ними мы можем делать больше, чем просто с функторами и аппликативными функторами.

Функция filterM

Функция `filter` – это просто хлеб программирования на языке Haskell (при том что функция `map` – масло). Она принимает предикат и список, подлежащий фильтрации, а затем возвращает новый список, в котором сохраняются только те элементы, которые удовлетворяют предикату. Её тип таков:

```
filter :: (a -> Bool) -> [a] -> [a]
```

Предикат берёт элемент списка и возвращает значение типа `Bool`. А вдруг возвращённое им значение типа `Bool` было на самом деле монадическим? Что если к нему был приложен контекст?.. Например, каждое значение `True` или `False`, произведённое предикатом, имело также сопутствующее моноидное значение вроде

`["Принято число 5"]` или `["3 слишком мало"]`? Если бы это было так, мы бы ожидали, что к результирующему списку тоже прилагается журнал всех журнальных значений, которые были произведены на пути. Поэтому если бы к списку, возвращённому предикатом, возвращающим значение типа `Bool`, был приложен контекст, мы ожидали бы, что к результирующему списку тоже прикреплён некоторый контекст. Иначе контекст, приложенный к каждому значению типа `Bool`, был бы утрачен.

Функция `filterM` из модуля `Control.Monad` делает именно то, что мы хотим! Её тип таков:

```
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

Предикат возвращает монадическое значение, результат которого – типа `Bool`, но поскольку это монадическое значение, его контекст может быть всем чем угодно, от возможной неудачи до недетерминированности и более! Чтобы обеспечить отражение контекста в окончательном результате, результат тоже является монадическим значением.

Давайте возьмём список и оставим только те значения, которые меньше 4. Для начала мы используем обычную функцию `filter`:

```
ghci> filter (\x -> x < 4) [9,1,5,2,10,3]
[1,2,3]
```

Это довольно просто. Теперь давайте создадим предикат, который помимо представления результата `True` или `False` также предоставляет журнал своих действий. Конечно же, для этого мы будем использовать монаду `Writer`:

```
keepSmall :: Int -> Writer [String] Bool
keepSmall x
   | x < 4 = do
       tell ["Сохраняем " ++ show x]
       return True
   | otherwise = do
       tell [show x ++ " слишком велико, выбрасываем"]
       return False
```

Вместо того чтобы просто возвращать значение типа `Bool`, функция возвращает значение типа `Writer [String] Bool`. Это монадический предикат. Звучит необычно, не так ли? Если число

меньше числа 4, мы сообщаем, что оставили его, а затем возвращаем значение True.

Теперь давайте передадим его функции filterM вместе со списком. Поскольку предикат возвращает значение типа Writer, результирующий список также будет значением типа Writer.

```
ghci> fst $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
[1,2,3]
```

Проверяя результат результирующего значения монады Writer, мы видим, что всё в порядке. Теперь давайте распечатаем журнал и посмотрим, что у нас есть:

```
ghci> mapM_ putStrLn $ snd $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
9 слишком велико, выбрасываем
Сохраняем 1
5 слишком велико, выбрасываем
Сохраняем 2
10 слишком велико, выбрасываем
Сохраняем 3
```

Итак, просто предоставляя монадический предикат функции filterM, мы смогли фильтровать список, используя возможности применяемого нами монадического контекста.

Очень крутой трюк в языке Haskell – использование функции filterM для получения множества-степени списка (если мы сейчас будем думать о нём как о множестве). *Множеством – степенью* некоторого множества называется множество всех подмножеств данного множества. Поэтому если у нас есть множество вроде [1,2,3], его множество-степень включает следующие множества:

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

Другими словами, получение множества-степени похоже на получение всех сочетаний сохранения и выбрасывания элементов из

множества. Например, [2,3] – это исходное множество с исключением числа 1; [1,2] – это исходное множество с исключением числа 3 и т. д.

Чтобы создать функцию, которая возвращает множество-степень какого-то списка, мы положимся на недетерминированность. Мы берём список [1,2,3], а затем смотрим на первый элемент, который равен 1, и спрашиваем себя: «Должны ли мы его сохранить или отбросить?» Ну, на самом деле мы хотели бы сделать и то и другое. Поэтому мы отфильтруем список и используем предикат, который сохраняет и отбрасывает каждый элемент из списка недетерминированно. Вот наша функция powerset:

```
powerset :: [a] -> [[a]]
powerset xs = filterM (\x -> [True, False]) xs
```

Стоп, это всё?! Угу! Мы решаем отбросить и оставить каждый элемент независимо от того, что он собой представляет. У нас есть недетерминированный предикат, поэтому результирующий список тоже будет недетерминированным значением – и потому будет списком списков. Давайте попробуем:

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

Вам потребуется немного поразмыслить, чтобы понять это. Просто воспринимайте списки как недетерминированные значения, которые толком не знают, чем быть, поэтому решают быть сразу всем, – и эту концепцию станет проще усвоить!

Функция foldM

Монадическим аналогом функции foldl является функция foldM. Если вы помните свои свёртки из главы 5, вы знаете, что функция foldl принимает бинарную функцию, исходный аккумулятор и сворачиваемый список, а затем сворачивает его слева в одно значение, используя бинарную функцию. Функция foldM делает то же самое, только она принимает бинарную функцию, производящую монадическое значение, и сворачивает список с её использованием. Неудивительно, что результирующее значение тоже является монадическим. Тип функции foldl таков:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Тогда как функция foldM имеет такой тип:

```
foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
```

Значение, которое возвращает бинарная функция, является монадическим, поэтому результат всей свёртки тоже является монадическим. Давайте сложим список чисел с использованием свёртки:

```
ghci> foldl (\acc x -> acc + x) 0 [2,8,3,1]
14
```

Исходный аккумулятор равен 0, затем к аккумулятору прибавляется 2, что даёт в результате новый аккумулятор со значением 2. К этому аккумулятору прибавляется 8, что даёт в результате аккумулятор равный 10 и т. д. Когда мы доходим до конца, результатом является окончательный аккумулятор.

А ну как мы захотели бы сложить список чисел, но с дополнительным условием: если какое-то число в списке больше 9, всё должно окончиться неудачей? Имело бы смысл использовать бинарную функцию, которая проверяет, больше ли текущее число, чем 9. Если больше, то функция оканчивается неудачей; если не больше – продолжает свой радостный путь. Из-за этой добавленной возможности неудачи давайте заставим нашу бинарную функцию возвращать аккумулятор Maybe вместо обычного.

Вот бинарная функция:

```
binSmalls :: Int -> Int -> Maybe Int
binSmalls acc x
   | x > 9     = Nothing
   | otherwise = Just (acc + x)
```

Поскольку наша бинарная функция теперь является монадической, мы не можем использовать её с обычной функцией foldl; следует использовать функцию foldM. Приступим:

```
ghci> foldM binSmalls 0 [2,8,3,1]
Just 14
ghci> foldM binSmalls 0 [2,11,3,1]
Nothing
```

Клёво! Поскольку одно число в списке было больше 9, всё дало в результате значение Nothing. Свёртка с использованием бинарной функции, которая возвращает значение Writer, – тоже круто, потому что в таком случае вы журналируете что захотите по ходу работы вашей свёртки.

Создание безопасного калькулятора выражений в обратной польской записи

Решая задачу реализации калькулятора для обратной польской записи в главе 10, мы отметили, что он работал хорошо до тех пор, пока получаемые им входные данные имели смысл. Но если что-то шло не так, это приводило к аварийному отказу всей нашей программы. Теперь, когда мы знаем, как сделать уже существующий код монадическим, давайте возьмём наш калькулятор и добавим в него обработку ошибок, воспользовавшись монадой Maybe.

Мы реализовали наш калькулятор обратной польской записи, получая строку вроде "1 3 + 2 *" и разделяя её на слова, чтобы получить нечто подобное: ["1","3","+","2","*"]. Затем мы сворачивали этот список, начиная с пустого стека и используя бинарную функцию свёртки, которая добавляет числа в стек либо манипулирует числами на вершине стека, чтобы складывать их или делить и т. п.

Вот это было основным телом нашей функции:

```
import Data.List

solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words
```

Мы превратили выражение в список строк и свернули его, используя нашу функцию свёртки. Затем, когда у нас в стеке остался лишь один элемент, мы вернули этот элемент в качестве ответа. Вот такой была функция свёртки:

```
foldingFunction :: [Double] -> String -> [Double]
foldingFunction (x:y:ys) "*" = (y * x):ys
foldingFunction (x:y:ys) "+" = (y + x):ys
foldingFunction (x:y:ys) "-" = (y - x):ys
foldingFunction xs numberString = read numberString:xs
```

Аккумулятором свёртки был стек, который мы представили списком значений типа `Double`. Если по мере того, как функция проходила по выражению в обратной польской записи, текущий элемент являлся оператором, она снимала два элемента с верхушки стека, применяла между ними оператор, а затем помещала результат обратно в стек. Если текущий элемент являлся строкой, представляющей число, она преобразовывала эту строку в фактическое число и возвращала новый стек, который был как прежний, только с этим числом, протолкнутым на верхушку.

Давайте сначала сделаем так, чтобы наша функция свёртки допускала мягкое окончание с неудачей. Её тип изменится с того, каким он является сейчас, на следующий:

```
foldingFunction :: [Double] -> String -> Maybe [Double]
```

Поэтому она либо вернёт новый стек в конструкторе `Just`, либо потерпит неудачу, вернув значение `Nothing`.

Функция `reads` похожа на функцию `read`, за исключением того, что она возвращает список с одним элементом в случае успешного чтения. Если ей не удалось что-либо прочитать, она возвращает пустой список. Помимо прочитанного ею значения она также возвращает ту часть строки, которую она не потребила. Мы сейчас скажем, что она должна потребить все входные данные для работы, и превратим её для удобства в функцию `readMaybe`. Вот она:

```
readMaybe :: (Read a) => String -> Maybe a
readMaybe st = case reads st of [(x, "")] -> Just x
                                _ -> Nothing
```

Теперь протестируем её:

```
ghci> readMaybe "1" :: Maybe Int
Just 1
ghci> readMaybe "ИДИ К ЧЁРТУ" :: Maybe Int
Nothing
```

Хорошо, кажется, работает. Итак, давайте превратим нашу функцию свёртки в монадическую функцию, которая может завершаться неудачей:

```
foldingFunction :: [Double] -> String -> Maybe [Double]
foldingFunction (x:y:ys) "*" = return ((y * x):ys)
foldingFunction (x:y:ys) "+" = return ((y + x):ys)
foldingFunction (x:y:ys) "-" = return ((y - x):ys)
foldingFunction xs numberString = liftM (:xs) (readMaybe numberString)
```

Первые три случая – такие же, как и прежние, только новый стек обёрнут в конструктор `Just` (для этого мы использовали здесь функцию `return`, но могли и просто написать `Just`). В последнем случае мы используем вызов `readMaybe numberString`, а затем отображаем это с помощью `(:xs)`. Поэтому если стек равен `[1.0,2.0]`, а выражение `readMaybe numberString` даёт в результате `Just 3.0`, то результатом будет `[3.0,1.0,2.0]`. Если же `readMaybe numberString` даёт в результате значение `Nothing`, результатом будет `Nothing`.

Давайте проверим функцию свёртки отдельно:

```
ghci> foldingFunction [3,2] "*"
Just [6.0]
ghci> foldingFunction [3,2] "-"
Just [-1.0]
ghci> foldingFunction [] "*"
Nothing
ghci> foldingFunction [] "1"
Just [1.0]
ghci> foldingFunction [] "1 ya-ya-ya-ya"
Nothing
```

Похоже, она работает! А теперь пришла пора для новой и улучшенной функции `solveRPN`. Вот она перед вами, дамы и господа!

```
import Data.List

solveRPN :: String -> Maybe Double
solveRPN st = do
```

```
  [result] <- foldM foldingFunction [] (words st)
  return result
```

Как и в предыдущей версии, мы берём строку и превращаем её в список слов. Затем производим свёртку, начиная с пустого стека, но вместо выполнения обычной свёртки с помощью функции `foldl` используем функцию `foldM`. Результатом этой свёртки с помощью функции `foldM` должно быть значение типа `Maybe`, содержащее список (то есть наш окончательный стек), и в этом списке должно быть только одно значение. Мы используем выражение `do`, чтобы взять это значение, и называем его `result`. В случае если функция `foldM` возвращает значение `Nothing`, всё будет равно `Nothing`, потому что так устроена монада `Maybe`. Обратите внимание на то, что мы производим сопоставление с образцом в выражении `do`, поэтому если список содержит более одного значения либо ни одного, сопоставление с образцом окончится неудачно и будет произведено значение `Nothing`. В последней строке мы просто вызываем выражение `return result`, чтобы представить результат вычисления выражения в обратной польской записи как результат окончательного значения типа `Maybe`.

Давайте попробуем:

```
ghci> solveRPN "1 2 * 4 +"
Just 6.0
ghci> solveRPN "1 2 * 4 + 5 *"
Just 30.0
ghci> solveRPN "1 2 * 4"
Nothing
ghci> solveRPN "1 8 трам-тарарам"
Nothing
```

Первая неудача возникает из-за того, что окончательный стек не является списком, содержащим один элемент: в выражении `do` сопоставление с образцом терпит фиаско. Вторая неудача возникает потому, что функция `readMaybe` возвращает значение `Nothing`.

Композиция монадических функций

Когда мы говорили о законах монад в главе 13, вы узнали, что функция `<=<` очень похожа на композицию, но вместо того чтобы

работать с обычными функциями типа a -> b, она работает с монадическими функциями типа a -> m b. Вот пример:

```
ghci> let f = (+1) . (*100)
ghci> f 4
401
ghci> let g = (\x -> return (x+1)) <=< (\x -> return (x*100))
ghci> Just 4 >>= g
Just 401
```

В данном примере мы сначала произвели композицию двух обычных функций, применили результирующую функцию к 4, а затем произвели композицию двух монадических функций и передали результирующей функции Just 4 с использованием операции >>=.

Если у вас есть набор функций в списке, вы можете скомпоновать их все в одну большую функцию, просто используя константную функцию id в качестве исходного аккумулятора и функцию (.) в качестве бинарной. Вот пример:

```
ghci> letf = foldr (.) id [(+1),(*100),(+1)]
ghci> f 1
201
```

Функция f принимает число, а затем прибавляет к нему 1, умножает результат на 100 и прибавляет к этому 1.

Мы можем компоновать монадические функции так же, но вместо обычной композиции используем операцию <=<, а вместо id – функцию return. Нам не требуется использовать функцию foldM вместо foldr или что-то вроде того, потому что функция <=< гарантирует, что композиция будет происходить монадически.

Когда вы знакомились со списковой монадой в главе 13, мы использовали её, чтобы выяснить, может ли конь пройти из одной позиции на шахматной доске на другую ровно в три хода. Мы создали функцию под названием moveKnight, которая берёт позицию коня на доске и возвращает все ходы, которые он может сделать в дальнейшем. Затем, чтобы произвести все возможные позиции, в которых он может оказаться после выполнения трёх ходов, мы создали следующую функцию:

```
in3 start = return start >>= moveKnight >>= moveKnight >>= moveKnight
```

И чтобы проверить, может ли конь пройти от `start` до `end` в три хода, мы сделали следующее:

```
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

Используя композицию монадических функций, можно создать функцию вроде `in3`, только вместо произведения всех позиций, которые может занимать конь после совершения трёх ходов, мы сможем сделать это для произвольного количества ходов. Если вы посмотрите на `in3`, то увидите, что мы использовали нашу функцию `moveKnight` трижды, причём каждый раз применяли операцию `>>=`, чтобы передать ей все возможные предшествующие позиции. А теперь давайте сделаем её более общей. Вот так:

```
import Data.List

inMany :: Int -> KnightPos -> [KnightPos]
inMany x start = return start >>= foldr (<=<) return (replicate x moveKnight)
```

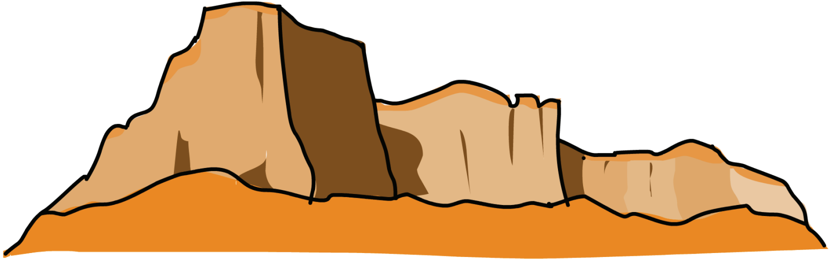
Во-первых, мы используем функцию `replicate`, чтобы создать список, который содержит `x` копий функции `moveKnight`. Затем мы монадически компонуем все эти функции в одну, что даёт нам функцию, которая берёт исходную позицию и недетерминированно перемещает коня `x` раз. Потом просто превращаем исходную позицию в одноэлементный список с помощью функции `return` и передаём его исходной функции.

Теперь нашу функцию `canReachIn3` тоже можно сделать более общей:

```
canReachIn :: Int -> KnightPos -> KnightPos -> Bool
canReachIn x start end = end `elem` inMany x start
```

Создание монад

В этом разделе мы рассмотрим пример, показывающий, как тип создаётся, опознаётся как монада, а затем для него создаётся подходящий экземпляр класса `Monad`. Обычно мы не намерены создавать монаду с единственной целью – создать монаду. Наоборот, мы создаём тип, цель которого – моделировать аспект некоторой проблемы, а затем, если впоследствии мы видим, что этот тип представляет



значение с контекстом и может действовать как монада, мы определяем для него экземпляр класса `Monad`.

Как вы видели, списки используются для представления недетерминированных значений. Список вроде `[3,5,9]` можно рассматривать как одно недетерминированное значение, которое просто не может решить, чем оно будет. Когда мы передаём список в функцию с помощью операции `>>=`, это просто создаёт все возможные варианты получения элемента из списка и применения к нему функции, а затем представляет эти результаты также в списке.

Если мы посмотрим на список `[3,5,9]` как на числа `3`, `5`, и `9`, встречающиеся одновременно, то можем заметить, что нет никакой информации в отношении того, какова вероятность встретить каждое из этих чисел. Что если бы нам было нужно смоделировать недетерминированное значение вроде `[3,5,9]`, но при этом мы бы хотели показать, что `3` имеет 50-процентный шанс появиться, а вероятность появления `5` и `9` равна 25%? Давайте попробуем провести эту работу!

Скажем, что к каждому элементу списка прилагается ещё одно значение: вероятность того, что он появится. Имело бы смысл представить это значение вот так:

```
[(3,0.5),(5,0.25),(9,0.25)]
```

Вероятности в математике обычно выражают не в процентах, а в вещественных числах между 0 и 1. Значение 0 означает, что чему-то ну никак не суждено сбыться, а значение 1 – что это что-то непременно произойдёт. Числа с плавающей запятой могут быстро создать путаницу, потому что они стремятся к потере точности, но язык Haskell предлагает тип данных для вещественных чисел. Он называется `Rational`, и определён он в модуле `Data.Ratio`. Чтобы создать значение типа `Rational`, мы записываем его так, как будто

это дробь. Числитель и знаменатель разделяются символом %. Вот несколько примеров:

```
ghci> 1 % 4
1 % 4
ghci> 1 % 2 + 1 % 2
1 % 1
ghci> 1 % 3 + 5 % 4
19 % 12
```

Первая строка – это просто одна четвёртая. Во второй строке мы складываем две половины, чтобы получить целое. В третьей строке складываем одну третью с пятью четвёртыми и получаем девять двенадцатых. Поэтому давайте выбросим эти плавающие запятые и используем для наших вероятностей тип Rational:

```
ghci> [(3,1 % 2),(5,1 % 4),(9,1 % 4)]
[(3,1 % 2),(5,1 % 4),(9,1 % 4)]
```

Итак, 3 имеет один из двух шансов появиться, тогда как 5 и 9 появляются один раз из четырёх. Просто великолепно!

Мы взяли списки и добавили к ним некоторый дополнительный контекст, так что это тоже представляет значения с контекстами. Прежде чем пойти дальше, давайте обернем это в newtype, ибо, как я подозреваю, мы будем создавать некоторые экземпляры.

```
import Data.Ratio

newtype Prob a = Prob { getProb :: [(a, Rational)] } deriving Show
```

Это функтор?.. Ну, раз список является функтором, это тоже должно быть функтором, поскольку мы только что добавили что-то в список. Когда мы отображаем список с помощью функции, то применяем её к каждому элементу. Тут мы тоже применим её к каждому элементу, но оставим вероятности как есть. Давайте создадим экземпляр:

```
instance Functor Prob where
   fmap f (Prob xs) = Prob $ map (\(x, p) -> (f x, p)) xs
```

Мы разворачиваем его из newtype при помощи сопоставления с образцом, затем применяем к значениям функцию f, сохраняя

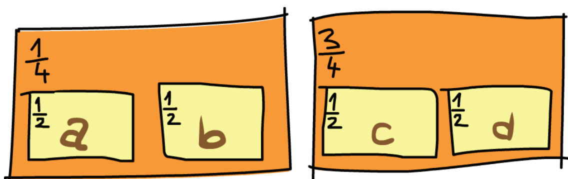
вероятности как есть, и оборачиваем его обратно. Давайте посмотрим, работает ли это:

```
ghci> fmap negate (Prob [(3,1 % 2),(5,1 % 4),(9,1 % 4)])
Prob {getProb = [(-3,1 % 2),(-5,1 % 4),(-9,1 % 4)]}
```

Обратите внимание, что вероятности должны давать в сумме 1. Если все эти вещи могут случиться, не имеет смысла, чтобы сумма их вероятностей была чем-то отличным от 1. Думаю, выпадение монеты на решку 75% раз и на орла 50% раз могло бы происходить только в какой-то странной Вселенной.

А теперь главный вопрос: это монада? Учитывая, что список является монадой, похоже, и это должно быть монадой. Во-первых, давайте подумаем о функции `return`. Как она работает со списками? Она берёт значение и помещает его в одноэлементный список. Что здесь происходит? Поскольку это должен быть минимальный контекст по умолчанию, она тоже должна создавать одноэлементный список. Что же насчёт вероятности? Вызов выражения `return x` должен создавать монадическое значение, которое всегда представляет `x` как свой результат, поэтому не имеет смысла, чтобы вероятность была равна `0`. Если оно всегда должно представлять это значение как свой результат, вероятность должна быть равна `1`!

А что у нас с операцией `>>=`? Выглядит несколько мудрёно, поэтому давайте воспользуемся тем, что для монад выражение `m >>= f` всегда равно выражению `join (fmap f m)`, и подумаем, как бы мы разгладили список вероятностей списков вероятностей. В качестве примера рассмотрим список, где существует 25-процентный шанс, что случится именно `'a'` или `'b'`. И `'a'`, и `'b'` могут появиться с равной вероятностью. Также есть шанс 75%, что случится именно `'c'` или `'d'`. То есть `'c'` и `'d'` также могут появиться с равной вероятностью. Вот рисунок списка вероятностей, который моделирует данный сценарий:



Каковы шансы появления каждой из этих букв? Если бы мы должны были изобразить просто четыре коробки, каждая из которых содержит вероятность, какими были бы эти вероятности? Чтобы узнать это, достаточно умножить каждую вероятность на все вероятности, которые в ней содержатся. Значение 'a' появилось бы один раз из восьми, как и 'b', потому что если мы умножим одну четвёртую на одну четвёртую, то получим одну восьмую. Значение 'c' появилось бы три раза из восьми, потому что три четвёртых, умноженные на одну вторую, – это три восьмых. Значение 'd' также появилось бы три раза из восьми. Если мы сложим все вероятности, они по-прежнему будут давать в сумме единицу.

Вот эта ситуация, выраженная в форме списка вероятностей:

```
thisSituation :: Prob (Prob Char)
thisSituation = Prob
   [(Prob [('a',1 % 2),('b',1 % 2)], 1 % 4)
   ,(Prob [('c',1 % 2),('d',1 % 2)], 3 % 4)
   ]
```

Обратите внимание, её тип – `Prob (Prob Char)`. Поэтому теперь, когда мы поняли, как разгладить вложенный список вероятностей, всё, что нам нужно сделать, – написать для этого код. Затем мы можем определить операцию `>>=` просто как `join (fmap f m)`, и заполучим монаду! Итак, вот функция `flatten`, которую мы будем использовать, потому что имя `join` уже занято:

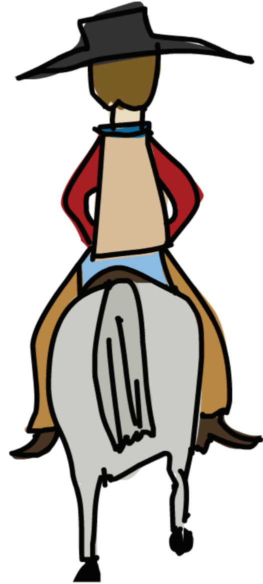
```
flatten :: Prob (Prob a) -> Prob a
flatten (Prob xs) = Prob $ concat $ map multAll xs
   where multAll (Prob innerxs, p) = map (\(x, r) -> (x, p*r)) innerxs
```

Функция `multAll` принимает кортеж, состоящий из списка вероятностей и вероятности `p`, которая к нему приложена, а затем умножает каждую внутреннюю вероятность на `p`, возвращая список пар элементов и вероятностей. Мы отображаем каждую пару в нашем списке вероятностей с помощью функции `multAll`, а затем просто разглаживаем результирующий вложенный список.

Теперь у нас есть всё, что нам нужно. Мы можем написать экземпляр класса `Monad`!

```
instance Monad Prob where
   return x = Prob [(x,1 % 1)]
```

```
    m >>= f = flatten (fmap f m)
    fail _ = Prob []
```



Поскольку мы уже сделали всю тяжелую работу, экземпляр очень прост. Мы определили функцию `fail`, которая такова же, как и для списков, поэтому если при сопоставлении с образцом в выражении `do` происходит неудача, неудача случается в контексте списка вероятностей.

Важно также проверить, что для только что созданной нами монады выполняются законы монад:

1. Первое правило говорит, что выражение `return x >>= f` должно равняться выражению `f x`. Точное доказательство было бы довольно громоздким, но нам видно, что если мы поместим значение в контекст по умолчанию с помощью функции `return`, затем отобразим это с помощью функции, используя `fmap`, а потом отобразим результирующий список вероятностей, то каждая вероятность, являющаяся результатом функции, была бы умножена на вероятность `1 % 1`, которую мы создали с помощью функции `return`, так что это не повлияло бы на контекст.
2. Второе правило утверждает, что выражение `m >> return` ничем не отличается от `m`. Для нашего примера доказательство того, что выражение `m >> return` равно просто `m`, аналогично доказательству первого закона.
3. Третий закон утверждает, что выражение `f <=< (g <=< h)` должно быть аналогично выражению `(f <=< g) <=< h`. Это тоже верно, потому что данное правило выполняется для списковой монады, которая составляет основу для монады вероятностей, и потому что умножение ассоциативно. `1 % 2 * (1 % 3 * 1 % 5)` равно `(1 % 2 * 1 % 3) * 1 % 5`.

Теперь, когда у нас есть монада, что мы можем с ней делать? Она может помочь нам выполнять вычисления с вероятностями.

Мы можем обрабатывать вероятностные события как значения с контекстами, а монада вероятностей обеспечит отражение этих вероятностей в вероятностях окончательного результата.

Скажем, у нас есть две обычные монеты и одна монета, с одной стороны налитая свинцом: она поразительным образом выпадает на решку девять раз из десяти и на орла – лишь один раз из десяти. Если мы подбросим все монеты одновременно, какова вероятность того, что все они выпадут на решку? Во-первых, давайте создадим значения вероятностей для подбрасывания обычной монеты и для монеты, налитой свинцом:

```
data Coin = Heads | Tails deriving (Show, Eq)

coin :: Prob Coin
coin = Prob [(Heads,1 % 2),(Tails,1 % 2)]

loadedCoin :: Prob Coin
loadedCoin = Prob [(Heads,1 % 10),(Tails,9 % 10)]
```

И наконец, действие по подбрасыванию монет:

```
import Data.List (all)

flipThree :: Prob Bool
flipThree = do
   a <- coin
   b <- coin
   c <- loadedCoin
   return (all (==Tails) [a,b,c])
```

При попытке запустить его видно, что вероятность выпадения решки у всех трёх монет не так высока, даже несмотря на жульничество с нашей налитой свинцом монетой:

```
ghci> getProb flipThree
[(False,1 % 40),(False,9 % 40),(False,1 % 40),(False,9 % 40),
(False,1 % 40),(False,9 % 40),(False,1 % 40),(True,9 % 40)]
```

Все три приземлятся решкой вверх 9 раз из 40, что составляет менее 25%!.. Видно, что наша монада не знает, как соединить все исходы `False`, где все монеты не приземляются решкой вверх,

в один исход. Впрочем, это не такая серьёзная проблема, поскольку написание функции для вставки всех одинаковых исходов в один исход довольно просто (это упражнение я оставляю вам в качестве домашнего задания).

В этом разделе мы перешли от вопроса («Что если бы списки также содержали информацию о вероятностях?») к созданию типа, распознанию монады и, наконец, созданию экземпляра и выполнению с ним некоторых действий. Думаю, это очаровательно! К этому времени у вас уже должно сложиться довольно неплохое понимание монад и их сути.

15
ЗАСТЁЖКИ

Хотя чистота языка Haskell даёт море преимуществ, вместе с тем он заставляет нас решать некоторые проблемы не так, как мы решали бы их в нечистых языках.

Из-за прозрачности ссылок одно значение в языке Haskell всё равно что другое, если оно представляет то же самое. Поэтому если у нас есть дерево, заполненное пятёрками (или, может, пятернями?), и мы хотим изменить одну из них на шестёрку, мы должны каким-то образом понимать, какую именно пятёрку в нашем дереве мы хотим изменить. Нам нужно знать, где в нашем дереве она находится. В нечистых языках можно было бы просто указать, где в памяти находится пятёрка, и изменить её. Но в языке Haskell одна пятёрка – всё равно что другая, поэтому нельзя проводить различие исходя из их расположения в памяти.

К тому же на самом деле мы не можем что-либо *изменять*. Когда мы говорим, что «изменяем дерево», то на самом деле имеем в виду, что мы берём дерево и возвращаем новое, аналогичное первоначальному, но немного отличающееся.

Одна вещь, которую мы *можем* сделать, – запомнить путь от корня дерева до элемента, который следует изменить. Мы могли бы сказать:

«Возьми это дерево, пойди влево, пойди вправо, а затем опять влево и измени находящийся там элемент». Хотя это и работает, но может быть неэффективно. Если позднее мы захотим изменить элемент, находящийся рядом с элементом, изменённым нами ранее, нам снова нужно будет пройти весь путь от корня дерева до нашего элемента!

В этой главе вы увидите, как взять некую структуру данных и снабдить её тем, что называется *застёжкой*, чтобы фокусироваться на части структуры данных таким образом, который делает изменение её элементов простым, а прохождение по ней – эффективным. Славно!

Прогулка

Как вы узнали на уроках биологии, есть множество различных деревьев, поэтому давайте выберем зёрнышко, которое мы используем, чтобы посадить наше. Вот оно:

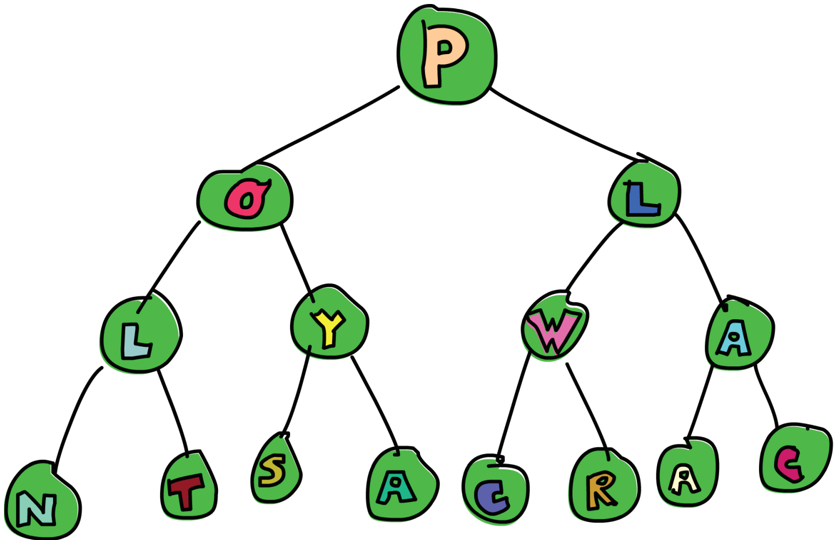
```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)
```

Наше дерево или пусто, или является узлом, содержащим элемент и два поддерева. Вот хороший пример такого дерева, которое я отдаю вам, читатель, просто задаром!

```
freeTree :: Tree Char
freeTree =
  Node 'P'
    (Node 'O'
       (Node 'L'
          (Node 'N' Empty Empty)
          (Node 'T' Empty Empty)
       )
       (Node 'Y'
          (Node 'S' Empty Empty)
          (Node 'A' Empty Empty)
       )
    )
    (Node 'L'
       (Node 'W'
          (Node 'C' Empty Empty)
          (Node 'R' Empty Empty)
       )
```

```
    (Node 'A'
      (Node 'A' Empty Empty)
      (Node 'C' Empty Empty)
    )
)
```

А вот это дерево, представленное графически:



Заметили символ W в дереве? Предположим, мы хотим заменить его символом P. Как нам это сделать? Ну, один из подходящих способов – сопоставление нашего дерева с образцом до тех пор, пока мы не найдём элемент, сначала двигаясь вправо, а затем влево. Вот соответствующий код:

```
changeToP :: Tree Char -> Tree Char
changeToP (Node x l (Node y (Node _ m n) r)) = Node x l (Node y (Node 'P' m n) r)
```

Тьфу, какая гадость! Это не только некрасиво, но к тому же несколько сбивает с толку. Что здесь на самом деле происходит? Мы сопоставляем наше дерево с образцом и даём его корневому элементу идентификатор x (который превращается в символ 'P' из корня), а левому поддереву – идентификатор l. Вместо того чтобы дать имя правому поддереву, мы опять же сопоставляем его

с образцом. Мы продолжаем это сопоставление с образцом до тех пор, пока не достигнем поддерева, корнем которого является наш искомый символ 'W'. Как только мы произвели сопоставление, мы перестраиваем дерево; только поддерево, которое содержало символ 'W', теперь содержит символ 'P'.

Есть ли лучший способ? Как насчёт того, чтобы наша функция принимала дерево вместе со списком направлений? Направления будут кодироваться символами L или R, представляя левую и правую стороны соответственно, и мы изменим элемент, которого достигнем, следуя переданным направлениям. Посмотрите:

```
data Direction = L | R deriving (Show)
type Directions = [Direction]

changeToP :: Directions -> Tree Char -> Tree Char
changeToP (L:ds) (Node x l r) = Node x (changeToP ds l) r
changeToP (R:ds) (Node x l r) = Node x l (changeToP ds r)
changeToP [] (Node _ l r) = Node 'P' l r
```

Если первый элемент в списке направлений – L, мы строим новое дерево, похожее на прежнее, только элемент в его левом поддереве заменён символом 'P'. Когда мы рекурсивно вызываем функцию changeToP, то передаём ей только «хвост» списка направлений, потому что мы уже переместились влево. То же самое происходит в случае с направлением R. Если список направлений пуст, это значит, что мы дошли до нашего места назначения, так что мы возвращаем дерево, аналогичное переданному, за исключением того, что в качестве корневого элемента оно содержит символ 'P'.

Чтобы не распечатывать дерево целиком, давайте создадим функцию, которая принимает список направлений и сообщает нам об элементе в месте назначения:

```
elemAt :: Directions -> Tree a -> a
elemAt (L:ds) (Node _ l _) = elemAt ds l
elemAt (R:ds) (Node _ _ r) = elemAt ds r
elemAt [] (Node x _ _) = x
```

Эта функция на самом деле очень похожа на функцию changeToP. С одной только разницей: вместо запоминания того, что встречается на пути, и воссоздания дерева она игнорирует всё, кроме своего

места назначения. Здесь мы заменяем символ 'W' символом 'P' и проверяем, сохраняется ли изменение в нашем новом дереве:

```
ghci> let newTree = changeToP [R,L] freeTree
ghci> elemAt [R,L] newTree
'P'
```

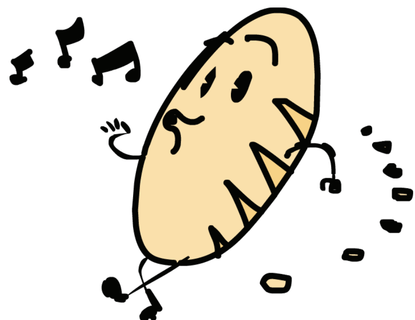
Кажется, работает! В этих функциях список направлений служит чем-то вроде *фокуса*, потому как в точности указывает на одно поддерево нашего дерева. Например, список направлений [R] фокусируется на поддереве, находящемся справа от корня. Пустой список направлений фокусируется на самом главном дереве.

Хотя эта техника с виду весьма хороша, она может быть довольно неэффективной, особенно если мы хотим часто изменять элементы. Скажем, у нас есть огромное дерево и длинный список направлений, который указывает весь путь до некоего элемента в самом низу дерева. Мы используем список направлений, чтобы пройтись по дереву и изменить элемент внизу. Если мы хотим изменить другой элемент, который близок к только что изменённому нами элементу, нужно начать с корня дерева и снова пройти весь путь вниз. Какая тоска!..

В следующем разделе мы найдём более удачный способ фокусироваться на поддереве – способ, который позволяет нам эффективно переводить фокус на близлежащие поддеревья.

Тропинка из хлебных крошек

Чтобы фокусироваться на поддереве, нам нужно что-то лучшее, нежели просто список направлений, по которому мы следуем из корня нашего дерева. А могло бы помочь, если бы мы начали с корня дерева и двигались на один шаг влево или вправо за раз, оставляя по пути «хлебные крошки»? Используя этот подход, когда мы идём влево, мы запоминаем, что пошли влево; а когда идём вправо, мы запоминаем, что пошли вправо. Давайте попробуем.



Чтобы представить «хлебные крошки», мы также будем

использовать список со значениями направлений (значения `L` и `R`), называя их, однако, не `Directions`, а `Breadcrumbs`, потому что наши направления теперь будут переворачиваться по мере того, как мы оставляем их, проходя вниз по нашему дереву.

```
type Breadcrumbs = [Direction]
```

Вот функция, которая принимает дерево и какие-то «хлебные крошки» и перемещается в левое поддерево, добавляя код `L` в «голову» списка, который представляет наши хлебные крошки:

```
goLeft :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goLeft (Node _ l _, bs) = (l, L:bs)
```

Мы игнорируем элемент в корне и правое поддерево и просто возвращаем левое поддерево вместе с прежними «хлебными крошками», где код `L` присутствует в качестве «головы».

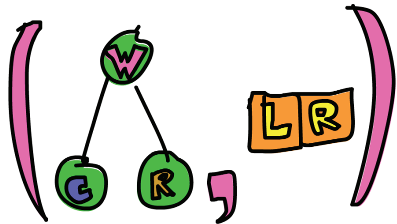
Вот функция для перемещения вправо:

```
goRight :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goRight (Node _ _ r, bs) = (r, R:bs)
```

Она работает так же, как и функция для перемещения влево.

Давайте используем эти функции, чтобы взять наше дерево `freeTree` и переместиться вправо, а затем влево.

```
ghci> goLeft (goRight (freeTree, []))
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```



Теперь у нас есть дерево с символом `'W'`, находящимся в его корне, символом `'C'` – в корне его левого поддерева и символом `'R'` – в корне правого поддерева. «Хлебными крошками» являются коды `[L,R]`, потому что сначала мы пошли вправо, а затем влево.

Чтобы сделать обход нашего дерева более ясным, мы можем использовать оператор `-:` из главы 13, который мы определили следующим образом:

```
x -: f = f x
```

Это позволяет нам применять функции к значениям, сначала записывая значение, потом `-:`, а затем функцию. Поэтому вместо выражения `goRight (freeTree, [])` мы можем написать `(freeTree, []) -: goRight`. Используя эту форму, перепишем предыдущий пример так, чтобы было более очевидно, что мы идём вправо, а затем влево:

```
ghci> (freeTree, []) -: goRight -: goLeft
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```

Движемся обратно вверх

Что, если мы хотим пойти обратно вверх по нашему дереву? Благодаря «хлебным крошкам» нам известно, что текущее дерево является левым поддеревом своего родителя, а последнее является правым поддеревом своего родителя – собственно, это всё, что нам известно. «Хлебные крошки» не сообщают нам достаточно сведений о родителе текущего поддерева, чтобы была возможность пойти вверх по дереву. Похоже, что помимо направления, по которому мы пошли, отдельная «хлебная крошка» должна также содержать все остальные сведения, которые необходимы для обратного движения вверх. В таком случае необходимыми сведениями являются элемент в родительском дереве вместе с его правым поддеревом.

Вообще у отдельной «хлебной крошки» должны быть все сведения, необходимые для восстановления родительского узла. Так что она должна иметь информацию из всех путей, которыми мы не пошли, а также знать направление, по которому мы пошли. Однако она не должна содержать поддерево, на котором мы фокусируемся в текущий момент, – потому что у нас уже есть это поддерево в первом компоненте кортежа. Если бы оно присутствовало у нас и в «хлебной крошке», мы бы имели копию уже имеющейся информации.

А нам такая копия не нужна, поскольку если бы мы изменили несколько элементов в поддереве, на котором фокусируемся, то имеющаяся в «хлебных крошках» информация не согласовывалась бы с произведёнными нами изменениями. Копия имеющейся информации устаревает, как только мы изменяем что-либо в нашем фокусе. Если наше дерево содержит много элементов, это также может забрать много памяти.

Давайте изменим наши «хлебные крошки», чтобы они содержали информацию обо всём, что мы проигнорировали ранее, когда двигались влево и вправо. Вместо типа Direction создадим новый тип данных:

```
data Crumb a = LeftCrumb a (Tree a) | RightCrumb a (Tree a) deriving (Show)
```

Теперь вместо кода L у нас есть значение LeftCrumb, содержащее также элемент узла, из которого мы переместились, и не посещённое нами правое поддерево. Вместо кода R есть значение RightCrumb, содержащее элемент узла, из которого мы переместились, и не посещённое нами левое поддерево.

Эти «хлебные крошки» уже содержат все сведения, необходимые для воссоздания дерева, по которому мы прошли. Теперь это не обычные «хлебные крошки» – они больше похожи на дискеты, которые мы оставляем при перемещении, потому что они содержат гораздо больше информации, чем просто направление, по которому мы шли!

В сущности, каждая такая «хлебная крошка» – как узел дерева, имеющий отверстие. Когда мы двигаемся вглубь дерева, в «хлебной крошке» содержится вся информация, которая имелась в покинутом нами узле, *за исключением* поддерева, на котором мы решили сфокусироваться. Нужно также указать, где находится отверстие. В случае со значением LeftCrumb нам известно, что мы переместились влево, так что отсутствующее поддерево – правое.

Давайте также изменим наш синоним типа Breadcrumbs, чтобы отразить это:

```
type Breadcrumbs a = [Crumb a]
```

Затем нам нужно изменить функции goLeft и goRight, чтобы они сохраняли информацию о путях, по которым мы не пошли, в наших «хлебных крошках», а не игнорировали эту информацию, как они делали это раньше. Вот новое определение функции goLeft:

```
goLeft :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goLeft (Node x l r, bs) = (l, (LeftCrumb x r):bs)
```

Как вы можете видеть, она очень похожа на нашу предыдущую функцию goLeft, но вместо того чтобы просто добавлять код L в «голову» нашего списка «хлебных крошек», мы добавляем туда зна-

чение LeftCrumb, чтобы показать, что мы пошли влево. Мы также снабжаем наше значение LeftCrumb элементом узла, из которого мы переместились (то есть значением x), и правым поддеревом, которое мы решили не посещать.

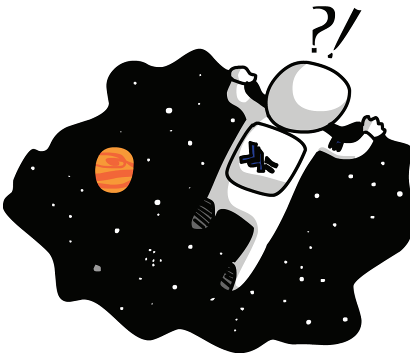
Обратите внимание: эта функция предполагает, что текущее дерево, находящееся в фокусе, – не Empty. Пустое дерево не содержит никаких поддеревьев, поэтому если мы попытаемся пойти влево из пустого дерева, возникнет ошибка. Причина в том, что сравнение значения типа Node с образцом будет неуспешным, и нет образца, который заботится о конструкторе Empty.

Функция goRight аналогична:

```
goRight :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goRight (Node x l r, bs) = (r, (RightCrumb x l):bs)
```

Ранее мы могли двигаться влево или вправо. То, чем мы располагаем сейчас, – это возможность действительно возвращаться вверх, запоминая информацию о родительских узлах и путях, которые мы не посетили. Вот определение функции goUp:

```
goUp :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goUp (t, LeftCrumbx r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```



Мы фокусируемся на дереве t и проверяем последнее значение типа Crumb. Если это значение равно LeftCrumb, мы строим новое дерево, используя наше дерево t в качестве левого поддерева и информацию о правом поддереве и элементе, которые мы не посетили, чтобы заполнить остальные части Node. Поскольку мы «переместились обратно» и подняли последнюю «хлебную крошку», а затем использовали её, чтобы воссоздать родительское дерево, в новом списке эта «хлебная крошка» не содержится.

Обратите внимание, что данная функция вызывает ошибку, если мы уже находимся на вершине дерева и хотим переместиться

выше. Позже мы используем монаду Maybe, чтобы представить возможную неудачу при перемещении фокуса.

С парой, состоящей из значений типов Tree a и Breadcrumbs a, у нас есть вся необходимая информация для восстановления дерева; кроме того, у нас есть фокус на поддереве. Эта схема позволяет нам легко двигаться вверх, влево и вправо.

Пару, содержащую часть структуры данных в фокусе и её окружение, называют *застёжкой*, потому что перемещение нашего фокуса вверх и вниз по структуре данных напоминает работу застёжки-молнии на брюках. Поэтому круто будет создать такой синоним типа:

```
type Zipper a = (Tree a, Breadcrumbs a)
```

Я бы предпочел назвать этот синоним типа Focus, поскольку это наглядно показывает, что мы фокусируемся на части структуры данных. Но так как для описания такой структуры чаще всего используется имя Zipper, будем придерживаться его.

Манипулируем деревьями в фокусе

Теперь, когда мы можем перемещаться вверх и вниз, давайте создадим функцию, изменяющую элемент в корне поддерева, на котором фокусируется застёжка.

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Node x l r, bs) = (Node (f x) l r, bs)
modify f (Empty, bs) = (Empty, bs)
```

Если мы фокусируемся на узле, мы изменяем его корневой элемент с помощью функции f. Фокусируясь на пустом дереве, мы оставляем его как есть. Теперь мы можем начать с дерева, перейти куда захотим и изменить элемент, одновременно сохраняя фокус на этом элементе, чтобы можно было легко переместиться далее вверх или вниз. Вот пример:

```
ghci> let newFocus = modify (\_ -> 'P') (goRight (goLeft (freeTree, [])))
```

Мы идём влево, затем вправо, а потом изменяем корневой элемент, заменяя его на 'P'. Если мы используем оператор -:, это будет читаться ещё лучше:

```
ghci> let newFocus = (freeTree, []) -: goLeft -: goRight -: modify (\_ -> 'P')
```

Затем мы можем перейти вверх, если захотим, и заменить имеющийся там элемент таинственным символом `'X'`:

```
ghci> let newFocus2 = modify (\_ -> 'X') (goUp newFocus)
```

Либо можем записать это, используя оператор `-:` следующим образом:

```
ghci> let newFocus2 = newFocus -: goUp -: modify (\_ -> 'X')
```

Перемещаться вверх просто, потому что «хлебные крошки», которые мы оставляем, формируют часть структуры данных, на которой мы не фокусируемся, но она вывернута наизнанку подобно носку. Вот почему когда мы хотим переместиться вверх, нам не нужно начинать с корня и пробираться вниз. Мы просто берём верхушку нашего вывернутого наизнанку дерева, при этом выворачивая обратно его часть и добавляя её в наш фокус.

Каждый узел имеет два поддерева, даже если эти поддеревья пусты. Поэтому, фокусируясь на пустом поддереве, мы по крайней мере можем сделать одну вещь: заменить его непустым поддеревом, таким образом прикрепляя дерево к листу. Код весьма прост:

```
attach :: Tree a -> Zipper a -> Zipper a
attach t (_, bs) = (t, bs)
```

Мы берём дерево и застёжку и возвращаем новую застёжку, фокус которой заменён переданным деревом. Можно не только расширять деревья, заменяя пустые поддеревья новыми, но и заменять существующие поддеревья. Давайте прикрепим дерево к дальнему левому краю нашего дерева `freeTree`:

```
ghci> let farLeft = (freeTree, []) -: goLeft -: goLeft -: goLeft -: goLeft
ghci> let newFocus = farLeft -: attach (Node 'Z' Empty Empty)
```

Значение `newFocus` теперь сфокусировано на дереве, которое мы только что прикрепили, а остальная часть дерева находится в «хлебных крошках» в вывернутом наизнанку виде. Если бы мы использовали функцию `goUp` для прохода всего пути к вершине дерева, оно было бы таким же деревом, как и `freeTree`, но с дополнительным символом `'Z'` в дальнем левом краю.

Идём прямо на вершину, где воздух чист и свеж!

Создать функцию, которая проходит весь путь к вершине дерева, независимо от того, на чём мы фокусируемся, очень просто. Вот она:

```
topMost :: Zipper a -> Zipper a
topMost (t, []) = (t, [])
topMost z = topMost (goUp z)
```

Если наша расширенная тропинка из «хлебных крошек» пуста, это значит, что мы уже находимся в корне нашего дерева, поэтому мы просто возвращаем текущий фокус. В противном случае двигаемся вверх, чтобы получить фокус родительского узла, а затем рекурсивно применяем к нему функцию `topMost`.

Итак, теперь мы можем гулять по нашему дереву, двигаясь влево, вправо и вверх, применяя функции `modify` и `attach` во время нашего путешествия. Затем, когда мы покончили с нашими изменениями, используем функцию `topMost`, чтобы сфокусироваться на вершине дерева и увидеть произведённые нами изменения в правильной перспективе.

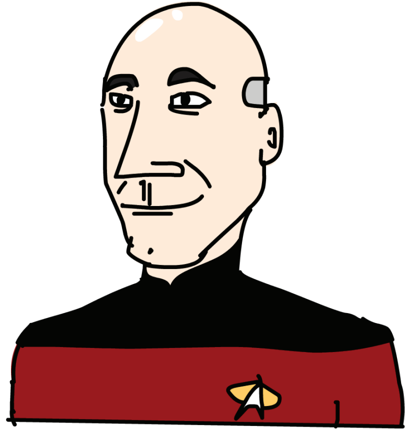
Фокусируемся на списках

Застёжки могут использоваться почти с любой структурой данных, поэтому неудивительно, что они работают с подсписками списков. В конце концов, списки очень похожи на деревья, только узел дерева содержит (или не содержит) элемент и несколько поддеревьев, а узел списка – элемент и лишь один подсписок. Когда мы реализовывали свои собственные списки в главе 7, то определили наш тип данных вот так:

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

Сравните это с определением нашего бинарного дерева – и легко увидите, что списки можно воспринимать в качестве деревьев, где каждый узел содержит лишь одно поддерево.

Список вроде `[1,2,3]` может быть записан как `1:2:3:[]`. Он состоит из «головы» списка равной `1` и «хвоста», который равен `2:3:[]`. `2:3:[]` также имеет «голову», которая равна `2`, и «хвост», который равен `3:[]`. Для `3:[]` «голова» равна `3`, а «хвост» является пустым списком `[]`.



Давайте создадим застёжку для списков. Чтобы изменить фокус на подсписках списка, мы перемещаемся или вперёд, или назад (тогда как при использовании деревьев мы перемещались вверх, влево или вправо). Помещённой в фокус частью будет подсписок, а кроме того, мы будем оставлять «хлебные крошки» по мере нашего движения вперёд.

А из чего состояла бы отдельная «хлебная крошка» для списка? Когда мы имели дело с бинарными деревьями, нужно было, чтобы «хлебная крошка» хранила элемент, содержащийся в корне родительского узла, вместе со всеми поддеревьями, которые мы не выбрали. Она также должна была запоминать, куда мы пошли, – влево или вправо. Поэтому требовалось, чтобы в ней содержалась вся имеющаяся в узле информация, за исключением поддерева, на которое мы решили навести фокус.

Списки проще, чем деревья. Нам не нужно запоминать, пошли ли мы влево или вправо, потому что вглубь списка можно пойти лишь одним способом. Поскольку для каждого узла существует только одно поддерево, нам также не нужно запоминать пути, по которым мы не пошли. Кажется, всё, что мы должны запоминать, – это предыдущий элемент. Если у нас есть список вроде `[3,4,5]` и мы знаем, что предыдущим элементом было значение `2`, мы можем пойти назад, просто поместив этот элемент в «голову» нашего списка, получая `[2,3,4,5]`.

Поскольку отдельная «хлебная крошка» здесь – просто элемент, нам на самом деле не нужно помещать её в тип данных, как мы делали это, когда создавали тип данных `Crumb`, использовавшийся застёжками для деревьев.

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

Первый список представляет список, на котором мы фокусируемся, а второй – это список «хлебных крошек». Давайте создадим функции, которые перемещаются вперёд и назад по спискам:

```
goForward :: ListZipper a -> ListZipper a
goForward (x:xs, bs) = (xs, x:bs)

goBack :: ListZipper a -> ListZipper a
goBack (xs, b:bs) = (b:xs, bs)
```

Когда мы движемся вперёд, мы фокусируемся на «хвосте» текущего списка и оставляем головной элемент в качестве «хлебной крошки». При движении назад мы берём самую последнюю «хлебную крошку» и помещаем её в начало списка. Вот эти две функции в действии:

```
ghci> let xs = [1,2,3,4]
ghci> goForward (xs, [])
([2,3,4], [1])
ghci> goForward ([2,3,4], [1])
([3,4], [2,1])
ghci> goForward ([3,4], [2,1])
([4], [3,2,1])
ghci> goBack ([4], [3,2,1])
([3,4], [2,1])
```

Вы можете видеть, что «хлебные крошки» в случае со списками – просто перевёрнутая часть вашего списка. Элемент, от которого мы удаляемся, всегда помещается в «голову» «хлебных крошек». Потом легко переместиться назад, просто вынимая этот элемент из их «головы» и делая его «головой» нашего фокуса. На данном примере опять-таки легко понять, почему мы называем это *застёжкой*: действительно очень напоминает перемещающийся вверх-вниз замок застёжки-молнии!

Если бы вы создавали текстовый редактор, можно было бы использовать список строк для представления строк текста, которые в текущий момент открыты, а затем использовать застёжку, чтобы знать, на какой строке в данный момент установлен курсор. Использование застёжки также облегчило бы вставку новых строк в любом месте текста или удаление имеющихся строк.

Очень простая файловая система

Для демонстрации работы застёжек давайте используем деревья, чтобы представить очень простую файловую систему. Затем мы

можем создать застёжку для этой файловой системы, которая позволит нам перемещаться между каталогами, как мы это делаем при переходах по реальной файловой системе.

Обычная иерархическая файловая система состоит преимущественно из файлов и каталогов. *Файлы* – это элементы данных, снабжённые именами. *Каталоги* используются для организации этих файлов и могут содержать файлы или другие каталоги. Для нашего простого примера достаточно сказать, что элементами файловой системы являются:

- файл под некоторым именем, содержащий некие данные;
- каталог под некоторым именем, содержащий другие элементы, которые сами являются или файлами, или каталогами.

Вот соответствующий тип данных и некоторые синонимы типов, чтобы было понятно, что к чему:

```
type Name = String
type Data = String
data FSItem = File Name Data | Folder Name [FSItem] deriving (Show)
```

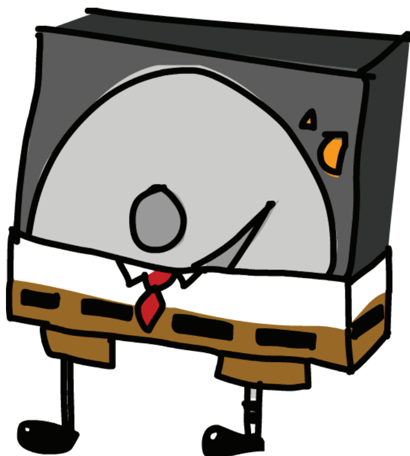
К файлу прилагаются две строки, представляющие его имя и содержимое. К каталогу прилагаются строка, являющаяся его именем, и список элементов. Если этот список пуст, значит, мы имеем пустой каталог.

Вот каталог с некоторыми файлами и подкаталогами (на самом деле это то, что в настоящую минуту содержится на моём диске):

```
myDisk :: FSItem
myDisk =
   Folder "root"
      [ File "goat_yelling_like_man.wmv" "baaaaaaa"
      , File "pope_time.avi" "Боже, благослови"
      , Folder "pics"
         [ File "ape_throwing_up.jpg" "блин..."
         , File "watermelon_smash.gif" "шмяк!!"
         , File "skull_man(scary).bmp" "Ой!"
         ]
      , File "dijon_poupon.doc" "лучшая горчица"
      , Folder "programs"
         [ File "sleepwizard.exe" "10 пойти поспать"
         , File "owl_bandit.dmg" "move ax, 42h"
         , File "not_a_virus.exe" "точно не вирус"
```

```
    , Folder "source code"
        [ File "best_hs_prog.hs" "main = print (fix error)"
        , File "random.hs" "main = print 4"
        ]
    ]
]
```

Создаём застёжку для нашей файловой системы



Теперь, когда у нас есть файловая система, всё, что нам нужно, – это застёжка, чтобы мы могли застёгивать файловую систему и брать её крупным планом, а также добавлять, изменять и удалять файлы и каталоги. Как и в случае с использованием бинарных деревьев и списков, наши «хлебные крошки» будут содержать информацию обо всём, что мы решили не посещать. Отдельная «хлебная крошка» должна хранить всё, кроме поддерева, на котором мы фокусируемся в данный момент. Она также должна указывать, где находится отверстие, чтобы при перемещении обратно вверх мы смогли вставить в отверстие наш предыдущий фокус.

В этом случае «хлебная крошка» должна быть похожа на каталог – только выбранный нами в данный момент каталог должен в нём отсутствовать. Вы спросите: «А почему не на файл?» Ну, потому что, когда мы фокусируемся на файле, мы не можем углубляться в файловую систему, а значит, не имеет смысла оставлять «хлебную крошку», которая говорит, что мы пришли из файла. Файл – это что-то вроде пустого дерева.

Если мы фокусируемся на каталоге `"root"`, а затем на файле `"dijon_poupon.doc"`, как должна выглядеть «хлебная крошка», которую мы оставляем? Она должна содержать имя своего родительского каталога вместе с элементами, идущими перед файлом, на котором мы фокусируемся, и следом за ним. Поэтому всё, что нам

требуется, – значение Name и два списка элементов. Храня два отдельных списка для элементов, идущих перед элементом, на котором мы фокусируемся, и для элементов, идущих за ним, мы будем точно знать, где мы его поместили, при перемещении обратно вверх. Таким образом, нам известно местоположение отверстия.

Вот наш тип «хлебной крошки» для файловой системы:

```
data FSCrumb = FSCrumb Name [FSItem] [FSItem]
deriving (Show)
```

А вот синоним типа для нашей застёжки:

```
type FSZipper = (FSItem, [FSCrumb])
```

Идти обратно вверх по иерархии очень просто. Мы берём самую последнюю «хлебную крошку» и собираем новый фокус из текущего фокуса и «хлебной крошки» следующим образом:

```
fsUp :: FSZipper -> FSZipper
fsUp (item, FSCrumb name ls rs:bs) = (Folder name (ls ++ [item] ++ rs), bs)
```

Поскольку нашей «хлебной крошке» были известны имя родительского каталога, а также элементы, которые шли перед находящимся в фокусе элементом каталога (то есть ls), и элементы, которые шли за ним (то есть rs), перемещаться вверх было легко.

Как насчёт продвижения вглубь файловой системы? Если мы находимся в "root" и хотим сфокусироваться на файле "dijon_poupon.doc", оставляемая нами «хлебная крошка» будет включать имя "root" вместе с элементами, предшествующими файлу "dijon_poupon.doc", и элементами, идущими за ним. Вот функция, которая, получив имя, фокусируется на файле или каталоге, расположенном в текущем каталоге, куда в текущий момент наведен фокус:

```
import Data.List (break)

fsTo :: Name -> FSZipper -> FSZipper
fsTo name (Folder folderName items, bs) =
   let (ls, item:rs) = break (nameIs name) items
   in (item, FSCrumb folderName ls rs:bs)

nameIs :: Name -> FSItem -> Bool
nameIs name (Folder folderName _) = name == folderName
nameIs name (File fileName _) = name == fileName
```

Функция fsTo принимает значения Name и FSZipper и возвращает новое значение FSZipper, которое фокусируется на файле с заданным именем. Этот файл должен присутствовать в текущем каталоге, находящемся в фокусе. Данная функция не производит поиск везде – она просто смотрит в текущем каталоге.

Сначала мы используем функцию break, чтобы разбить список элементов в каталоге на те, что предшествуют искомому нами файлу, и те, что идут за ним. Функция break принимает предикат и список и возвращает пару списков. Первый список в паре содержит элементы, для которых предикат возвращает значение False. Затем, когда предикат возвращает значение True для элемента, функция помещает этот элемент и остальную часть списка во второй элемент пары. Мы создали вспомогательную функцию nameIs, которая принимает имя и элемент файловой системы и, если имена совпадают, возвращает значение True.

Теперь ls – список, содержащий элементы, предшествующие искомому нами элементу; item является этим самым элементом, а rs – это список элементов, идущих за ним в его каталоге. И вот сейчас, когда они у нас есть, мы просто представляем элемент, полученный нами из функции break, как фокус и строим «хлебную крошку», которая содержит все необходимые ей данные.

Обратите внимание, что если имя, которое мы ищем, не присутствует в каталоге, образец item:rs попытается произвести сопоставление с пустым списком, и мы получим ошибку. А если наш текущий фокус – файл, а не каталог, мы тоже получим ошибку, и программа завершится аварийно.

Итак, мы можем двигаться вверх и вниз по нашей файловой системе. Давайте начнём движение с корня и перейдем к файлу "skull_man(scary).bmp":

```
ghci> let newFocus = (myDisk, []) -: fsTo "pics" -: fsTo "skull_man(scary).bmp"
```

Значение `newFocus` теперь – застёжка, сфокусированная на файле `skull_man(scary).bmp`. Давайте получим первый компонент застёжки (сам фокус) и посмотрим, так ли это на самом деле.

```
ghci> fst newFocus
File "skull_man(scary).bmp" "Ой!"
```

Переместимся выше и сфокусируемся на соседнем с ним файле `"watermelon_smash.gif"`:

```
ghci> let newFocus2 = newFocus -: fsUp -: fsTo "watermelon_smash.gif"
ghci> fst newFocus2
File "watermelon_smash.gif" "шмяк!!"
```

Манипулируем файловой системой

Теперь, когда мы можем передвигаться по нашей файловой системе, ею легко манипулировать. Вот функция, которая переименовывает находящийся в данный момент в фокусе файл или каталог:

```
fsRename :: Name -> FSZipper -> FSZipper
fsRename newName (Folder name items, bs) = (Folder newName items, bs)
fsRename newName (File name dat, bs) = (File newName dat, bs)
```

Давайте переименуем наш каталог `"pics"` в `"cspi"`:

```
ghci> let newFocus = (myDisk, []) -: fsTo "pics" -: fsRename "cspi" -: fsUp
```

Мы спустились к каталогу `"pics"`, переименовали его, а затем поднялись обратно вверх.

Как насчёт функции, которая создаёт новый элемент в текущем каталоге? Встречайте:

```
fsNewFile :: FSItem -> FSZipper -> FSZipper
fsNewFile item (Folder folderName items, bs) =
   (Folder folderName (item:items), bs)
```

Проще пареной репы! Обратите внимание, что если бы мы попытались добавить элемент, но фокусировались бы на файле, а не на каталоге, это привело бы к аварийному завершению программы.

Давайте добавим в наш каталог `"pics"` файл, а затем поднимемся обратно к корню:

```
ghci> let newFocus =
   (myDisk, []) -: fsTo "pics" -: fsNewFile (File "heh.jpg" "лол") -: fsUp
```

Что действительно во всём этом здорово, так это то, что когда мы изменяем нашу файловую систему, наши изменения на самом деле не производятся на месте – напротив, функция возвращает совершенно новую файловую систему. Таким образом, мы имеем доступ к нашей прежней файловой системе (в данном случае `myDisk`), а также к новой (первый компонент `newFocus`).

Используя застёжки, мы бесплатно получаем контроль версий. Мы всегда можем обратиться к старым версиям структур данных даже после того, как изменили их. Это не уникальное свойство застёжек; оно характерно для языка Haskell в целом, потому что его структуры данных неизменяемы. При использовании застёжек, однако, мы получаем возможность легко и эффективно обходить наши структуры данных, так что неизменность структур данных языка Haskell действительно начинает сиять во всей красе!

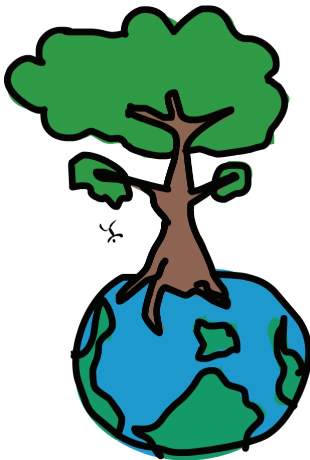
Осторожнее – смотрите под ноги!

До сих пор при обходе наших структур данных – будь они бинарными деревьями, списками, или файловыми системами – нам не было дела до того, что мы прошагаем слишком далеко и упадём. Например, наша функция `goLeft` принимает застёжку бинарного дерева и передвигает фокус на его левое поддерево:

```
goLeft :: Zipper a -> Zipper a
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

Но что если дерево, с которого мы сходим, является пустым? Что если это не значение `Node`, а `Empty`? В этом случае мы получили бы ошибку времени исполнения, потому что сопоставление с образцом завершилось бы неуспешно, а образец для обработки пустого дерева, у которого нет поддеревьев, мы не создавали.

До сих пор мы просто предполагали, что никогда не пытались бы навести фокус на левое поддерево пустого дерева, так как его левого поддерева просто не существует. Но переход к левому поддереву пустого дерева не имеет какого-либо смысла, и мы до сих пор это удачно игнорировали.



Ну или вдруг мы уже находимся в корне какого-либо дерева, и у нас нет «хлебных крошек», но мы всё же пытаемся переместиться вверх? Произошло бы то же самое! Кажется, при использовании застёжек каждый наш шаг может стать последним (не хватает только зловещей музыки). Другими словами, любое перемещение может привести к успеху, но также может привести и к неудаче. Вам это что-нибудь напоминает? Ну конечно же: монады! А конкретнее, монаду `Maybe`, которая добавляет к обычным значениям контекст возможной неудачи.

Давайте используем монаду `Maybe`, чтобы добавить к нашим перемещениям контекст возможной неудачи. Мы возьмём функции, которые работают с нашей застёжкой для двоичных деревьев, и превратим в монадические функции.

Сначала давайте позаботимся о возможной неудаче в функциях `goLeft` и `goRight`. До сих пор неуспешное окончание выполнения функций, которые могли окончиться неуспешно, всегда отражалось в их результате, и этот пример – не исключение.

Вот определения функций `goLeft` и `goRight` с добавленной возможностью неудачи:

```
goLeft :: Zipper a -> Maybe (Zipper a)
goLeft (Node x l r, bs) = Just (l, LeftCrumb x r:bs)
goLeft (Empty, _) = Nothing

goRight :: Zipper a -> Maybe (Zipper a)
goRight (Node x l r, bs) = Just (r, RightCrumb x l:bs)
goRight (Empty, _) = Nothing
```

Теперь, если мы попытаемся сделать шаг влево относительно пустого дерева, мы получим значение `Nothing`!

```
ghci> goLeft (Empty, [])
Nothing
ghci> goLeft (Node 'A' Empty Empty, [])
Just (Empty, [LeftCrumb 'A' Empty])
```

Выглядит неплохо! Как насчёт движения вверх? Раньше возникала проблема, если мы пытались пойти вверх, но у нас больше не было «хлебных крошек», что значило, что мы уже находимся в корне дерева. Это функция goUp, которая выдаст ошибку, если мы выйдем за пределы нашего дерева:

```
goUp :: Zipper a -> Zipper a
goUp (t, LeftCrumbx r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```

Давайте изменим её, чтобы она завершалась неудачей мягко:

```
goUp :: Zipper a -> Maybe (Zipper a)
goUp (t, LeftCrumbx r:bs) = Just (Node x t r,bs)
goUp (t, RightCrumb x l:bs) = Just (Node x l t, bs)
goUp (_, []) = Nothing
```

Если у нас есть хлебные крошки, всё в порядке, и мы возвращаем успешный новый фокус. Если у нас нет хлебных крошек, мы возвращаем неудачу.

Раньше эти функции принимали застёжки и возвращали застёжки, что означало, что мы можем сцеплять их следующим образом для осуществления обхода:

```
gchi> let newFocus = (freeTree, []) -: goLeft -: goRight
```

Но теперь вместо того, чтобы возвращать значение типа Zipper a, они возвращают значение типа Maybe (Zipper a), и сцепление функций подобным образом работать не будет. У нас была похожая проблема, когда в главе 13 мы имели дело с нашим канатоходцем. Он тоже проходил один шаг за раз, и каждый из его шагов мог привести к неудаче, потому что несколько птиц могли приземлиться на одну сторону его балансировочного шеста, что приводило к падению.

Теперь шутить будут над нами, потому что мы – те, кто производит обход, и обходим мы лабиринт нашей собственной разработки. К счастью, мы можем поучиться у канатоходца и сделать то, что сделал он: заменить обычное применение функций оператором >>=. Он берёт значение с контекстом (в нашем случае это значение типа Maybe (Zipper a), которое имеет контекст возможной неудачи) и передаёт его в функцию, обеспечивая при этом обработку контекста. Так что, как и наш канатоходец, мы отдадим все наши старые опера-

торы -: в счёт приобретения операторов >>=. Затем мы вновь сможем сцеплять наши функции! Смотрите, как это работает:

```
ghci> let coolTree = Node 1 Empty (Node 3 Empty Empty)
ghci> return (coolTree, []) >>= goRight
Just (Node 3 Empty Empty,[RightCrumb 1 Empty])
ghci> return (coolTree, []) >>= goRight >>= goRight
Just (Empty,[RightCrumb 3 Empty,RightCrumb 1 Empty])
ghci> return (coolTree, []) >>= goRight >>= goRight >>= goRight
Nothing
```

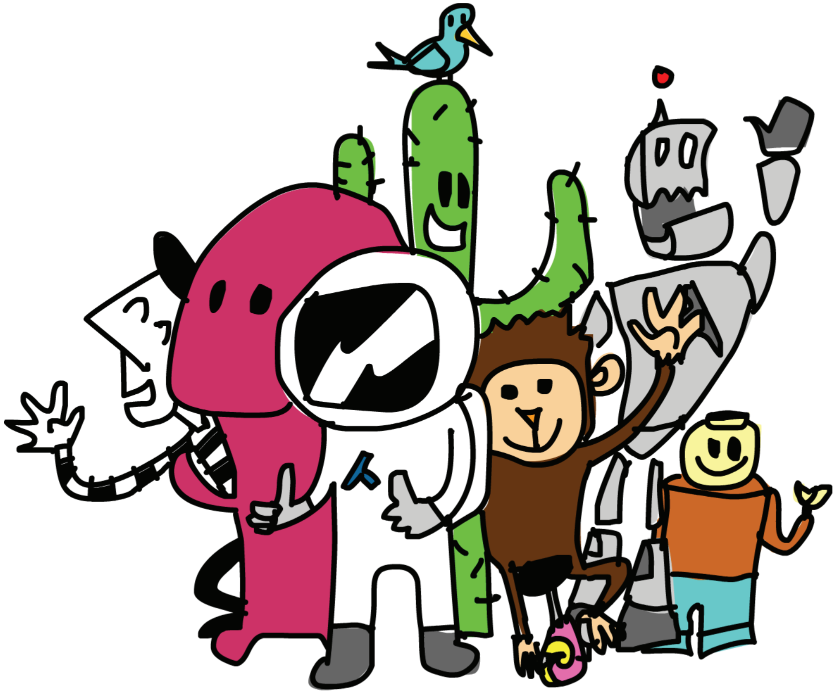
Мы использовали функцию `return`, чтобы поместить застёжку в конструктор `Just`, а затем прибегли к оператору `>>=`, чтобы передать это дело нашей функции `goRight`. Сначала мы создали дерево, которое в своей левой части имеет пустое поддерево, а в правой – узел, имеющий два пустых поддерева. Когда мы пытаемся пойти вправо один раз, результатом становится успех, потому что операция имеет смысл. Пойти вправо во второй раз – тоже нормально. В итоге мы получаем фокус на пустом поддереве. Но идти вправо третий раз не имеет смысла: мы не можем пойти вправо от пустого поддерева! Поэтому результат – `Nothing`.

Теперь мы снабдили наши деревья «сеткой безопасности», которая поймает нас, если мы свалимся. (Ух ты, хорошую метафору я подобрал.)

ПРИМЕЧАНИЕ. *В нашей файловой системе также имеется много случаев, когда операция может завершиться неуспешно, как, например, попытка сфокусироваться на несуществующем файле или каталоге. В качестве упражнения вы можете снабдить нашу файловую систему функциями, которые завершаются неудачей мягко, используя монаду* `Maybe`.

Благодарю за то, что прочитали!

...Или, по крайней мере, пролистали до последней страницы! Я надеюсь, вы нашли эту книгу полезной и весёлой. Я постарался дать вам хорошее понимание языка Haskell и его идиом. Хотя при изучении этого языка всегда открывается что-то новое, вы теперь сможете писать классный код, а также читать и понимать код других людей. Так что скорее приступайте к делу! Увидимся в мире программирования!



Миран Липовача

Изучай Haskell во имя добра

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Перевод с английского *Леушин Д., Синицын А., Арсанукаев Я.*
Научные редакторы *Душкин Р. В., Брагилевский В. Н.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 23.02.2012. Формат 60×90 1/16.
Гарнитура «NewBaskervilleC». Печать офсетная.
Усл. печ. л. 30.02. Тираж 500 экз.

Web-сайт издательства: www.dmk-press.ru

Отображения, монады, моноиды и другое!

Всё сказано в названии: «Изучай Хаскель во имя добра!» – веселый иллюстрированный самоучитель по этому сложному функциональному языку.

С помощью оригинальных рисунков автора, отсылке к поп-культуре, и, самое главное, благодаря полезным примерам кода, эта книга обучает основам функционального программирования так, как вы никогда не смогли бы себе представить.

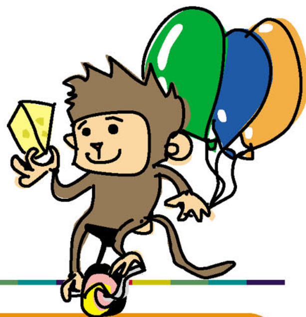
Вы начнете изучение с простого материала: основы синтаксиса, рекурсия, типы и классы типов. Затем, когда вы преуспеете в основах, начнется настоящий мастер-класс от профессионала: вы изучите, как использовать аппликативные функторы, монады, застежки, и другие легендарные конструкции Хаскеля, о которых вы читали только в сказках.

Продираясь сквозь образные (и порой безумные) примеры автора, вы научитесь:

- Смеяться в лицо побочным эффектам, поскольку вы овладеете техниками чистого функционального программирования.
- Использовать волшебство «ленивости» Хаскеля для игры с бесконечными наборами данных.
- Организовывать свои программы, создавая собственные типы, классы типов и модули.
- Использовать элегантную систему ввода-вывода Хаскеля, чтобы делиться гениальностью ваших программ с окружающим миром.

Нет лучшего способа изучить этот мощный язык, чем чтение «Изучай Хаскель во имя добра!», кроме, разве что, поедания мозга его создателей.

Миран Липовача (Miran Lipovača) изучает информатику в Любляне (Словения). Помимо его любви к Хаскелю, ему нравится заниматься боксом, играть на бас-гитаре и, конечно же, рисовать. У него есть увлечение танцующими скелетами и числом 71, а когда он проходит через автоматические двери, он притворяется, что на самом деле открывает их силой своей мысли.



ISBN 978-5-94074-749-9

Internet-магазин: www.alians-kniga.ru

Книга – почтой: orders@alians-kniga.ru

Оптовая продажа: «Альянс-книга»
тел. (499)725-54-09
books@alians-kniga.ru