О Haskell по-человечески

Д. Шевченко

О Haskell по-человечески

издание 2.0

Денис Шевченко

www.ohaskell.guide
2016

Оглавление

Глава 1

Приветствую!

Перед вами — книга о Haskell, удивительном и прекрасном языке программирования.

Почему эта книга появилась

Потому что меня откровенно достало. Почти все известные мне книги о Haskell начинаются с примера реализации быстрой сортировки и — куда ж без неё! — последовательности Фибоначчи. Эта книга не такая: минимум академизма, максимум практичности.

Цель

Функциональное программирование — своеобразное гетто посреди мегаполиса нашей индустрии. Доля функциональных языков пока ещё очень мала, и многие разработчики побаиваются знакомства с этими языками, и с Haskell в особенности. Моя цель — разрушить этот страх. Вероятно, вы слышали, что Haskell — это что-то архисложное, сугубо научное и непригодное для реальной жизни? Читайте дальше, и вскоре вы убедитесь в обратном.

О себе

Обыкновенный программист-самоучка. Разрабатываю с 2006 года. В 2012 году впервые услышал про Haskell, ужаснулся и поспешил о нём забыть. В 2013 вспомнил опять, в 2014 увлёкся всерьёз, а в 2015, после 8 лет жизни с C++, окончательно перешёл в Haskell-мир. Также я положил начало русскоязычному сообществу Haskell-разработчиков. И да, я действительно использую этот язык в своей каждодневной работе.

О вас

Знаете, что такое компилятор? Не боитесь командной строки? Слышали слово «функция»? Если да — смело продолжайте читать, никаких дополнительных навыков от вас не ожидается. И какой-либо математической подготовки — тоже.

Обещание

Возможно, вы по уши влюбитесь в Haskell. Возможно, он вызовет у вас отвращение. Обещаю одно — скучно не будет. Начнём.

Глава 2

Первые вопросы

Мне задавали их множество раз. Отвечаю.

«Что такое этот ваш Haskell?»

Haskell — чисто функциональный язык программирования общего назначения, может быть использован для решения самого широкого круга задач. Компилируемый, но может вести себя и как скриптовый. Кроссплатформенный. Ленивый, со строгой статической типизацией. И он не похож на другие языки. Совсем.

«Это что, какой-то новый язык?»

Вовсе нет. История Haskell началась ещё в 1987 году. Этот язык был рождён в математических кругах, когда группа людей решила создать лучший функциональный язык программирования. В 1990 году вышла первая версия языка, названного в честь известного американского математика Хаскелла Карри. В 1998 году язык был стандартизован, а начиная с 2000-х началось его медленное вхождение в мир практического программирования. За эти годы язык совершенствовался, и вот в 2010 мир увидел его обновлённый стандарт. Так что мы имеем дело с языком, который старше Java.

«И кто его сделал?»

Haskell создавался многими людьми. Наиболее известная реализация языка нашла своё воплощение в компиляторе GHC (The Glasgow Haskell Compiler), родившегося в 1989 году в Университете Глазго. У компилятора было несколько главных разработчиков, из которых наиболее известны двое, Simon Peyton Jones и Simon Marlow. Впоследствии весомый вклад в разработку GHC внесли ещё несколько сотен человек. Исходный код компилятора GHC открыт. Кстати, сам компилятор на 82% написан на Haskell.

Для любопытных: исчерпывающее повествование об истории Haskell и GHC читайте здесь.

«А библиотеки для Haskell имеются?»

О да! Их даже не сотни — их тысячи. В процессе чтения вы познакомитесь со многими из них.

«И что, его уже можно в production?»

Он уже в production. С момента выхода первого стандарта язык улучшался, развивалась его экосистема, появлялись новые библиотеки, выходили в свет книги. Haskell полностью готов к серьёзному коммерческому использованию, о чём свидетельствуют истории успешного внедрения Haskell в бизнесе, в том числе крупном.

«А порог вхождения в Haskell высокий?»

И да и нет. Освоение Haskell сложно в первую очередь из-за его непохожести на остальные языки, поэтому людям, имеющим опыт работы с другими языками, мозги поломать придётся. Именно поломать, а не просто пошевелить ими: Haskell заставляет иначе взглянуть даже на привычные вещи. С другой стороны, Haskell проще многих известных языков. Не верьте мне на слово, вскоре вы и сами в этом убедитесь. И знайте: многие люди, узнав вкус Haskell, категорически не желают возвращаться к другим языкам. Я вас предупредил.

«А я слышал ещё про какие-то монады...»

Да, есть такое дело. Некоторые вещи из мира Haskell не имеют прямых аналогов в других языках программирования, и это вводит новичков в ступор. Но не беспокойтесь: я сам прошёл через этот ступор и хорошо вас понимаю. Помните: новое лишь кажется страшным.

«А если сравнить его с C++/Python/Scala...»

Сравнение Haskell с другими языками выходит за рамки этой книги. Несколько раз вы встретите здесь кусочки кода на других языках, но я привожу их исключительно для того, чтобы подчеркнуть различие с Haskell, а вовсе не для сравнения в контексте «лучше/хуже». И вообще, я буду изо всех сил стараться не восхвалять Haskell без меры, я хочу лишь рассказать вам правду о нём. Мой вывод об этом языке я уже сделал, а свой вывод о нём вы должны сделать сами.

Глава 3

Об этой книге

В последние годы заметно возросло число книг, посвящённых Haskell, и это радует. Каждая из них преследует свою цель, поэтому трудно сказать, какая из них лучше. Цель этой книги двоякая.

Во-первых, я научу вас главному в Haskell. Основам, без освоения которых двигаться дальше никак не получится.

Во-вторых, я разрушу страх. Уже много лет вокруг Haskell витает дух страха, и я сполна ощутил его на себе. В действительности Haskell совсем не страшный, в нём нет чёрной магии, и чтобы программировать на нём, вам не нужна учёная степень. Более того, вы удивитесь, насколько просто в Haskell делать многие вещи, но эта простота откроется вам лишь после того, как вы близко познакомитесь с Тремя Китами Haskell, а также с госпожой Черепахой, поддерживающей оных. Имена этих Китов и Черепахи вы узнаете уже в следующей главе.

Эта книга не возведёт вас на вершины Haskell, но она откроет вам путь к этим вершинам.

Чего здесь нет

Трёх вещей вы не найдёте на страницах этой книги:

1. Исчерпывающего справочника по Haskell. Дублировать официальное описание стандарта Haskell 2010 я не стану.
2. Набора готовых рецептов. За рецептами пожалуйте на Stack Overflow.
3. Введения в математическую теорию. Несмотря на то, что Haskell корнями своими уходит в математику, в этой книге нет погружения в теорию категорий и в иные теории. Извините, если разочаровал.

О первом и втором издании

На обложке вы видели метку «издание 2.0». Перед вами второе издание, полностью переработанное и переосмысленное. Вот две причины, побудившие меня перепи-

сать книгу.

Первая — мои ошибки. Я убеждён, что обучать языку программирования могут лишь те, кто использует этот язык в своей каждодневной работе. На момент написания первой версии я ещё не работал с Haskell, а потому многого не знал и не понимал. В результате часть информации из первого издания была откровенно бедна, а несколько глав вообще вводили читателя в заблуждение.

Вторая причина — изменившаяся цель книги. Я намеренно сузил круг рассматриваемых здесь тем. Теперь книга всецело посвящена основам языка, поэтому не ищите здесь рассмотрения специфических тем. Я не очень-то верю в идею book-all-in-one, книга для новичков должна быть книгой для новичков. Вы не встретите здесь ни примеров реализации 3D-движка, ни рассказа о работе с PostgreSQL, ни повествования о проектировании игры для Android. Всё это можно делать с Haskell, но подобным темам посвящены другие публикации, которые несомненно будут вам по плечу после прочтения моей книги.

Читайте последовательно

И это важно. В процессе чтения вы заметите, что я периодически поднимаю вопросы и как бы оставляю их без ответа. Это делается вполне осознанно: ответы обязательно будут даны, но в последующих главах, там, где это будет наиболее уместно. Поэтому перепрыгивание с главы на главу может вас запутать.

Впрочем, в веб-версии книги есть «Предметный указатель», который поможет вам быстро найти нужное место, что особенно полезно при повторном прочтении книги.

Для любопытных

В конце большинства глав вы найдёте небольшой раздел, который так и называется — «Для любопытных». Читать его необязательно, но любознательным непременно понравится. В этом разделе я привожу некоторые технические подробности, исторические сведения и просто интересные факты.

И учтите, пожалуйста: содержимое раздела «Для любопытных» иногда чуток ломает последовательность изложения материала, это сделано осознанно. Помня о многих вопросах читателей к главам из предыдущего издания, я вынес ответы на некоторые из этих вопросов в данный раздел, и поэтому оный, скажем, в 12 главе может ссылаться на материал, изложенный лишь в 16 главе. Если сомневаетесь — не читайте.

О пояснениях

Во многих примерах исходного кода вы увидите пояснения вот такого вида:

```
type  String  =      [Char]

тип   этот    равен  тому
```

Такие пояснение следует читать слева направо и сверху вниз, и вы сразу поймёте что к чему. Каждая часть пояснения расположена строго под тем кусочком кода, к которому она относится.

Вот ещё один пример:

```
let (host, alias) = ("173.194.71.106", "www.google.com")

                    данное значение
     это
     хост
                                       а вот это
                                       значение
          это
          имя
```

Здесь я говорю вам: «Данное значение — это хост, а вот это значение — это имя». В ряде случаев я использую также различного вида подчёркивание:

```
(host, alias) = ("173.194.71.106", "www.google.com")

 ____           ________________

      =====                     ================
```

Здесь я провожу параллель: «Значение `host` ассоциировано со строкой `173.194.71.106`, а значение `alias` — со строкой `www.google.com`».

Благодарность

Эта книга — плод не только моих усилий. Многие члены нашего сообщества помогли мне советами, замечаниями и исправлениями. Большое спасибо вам, друзья!

А ещё я благодарю всех тех, кто создал Haskell, и всех тех, кто неустанно совершенствует его. Вашими усилиями наша профессия становится ещё более прекрасной!

Слово к читавшим первое издание

Если вы не читали его — можете переходить к следующей главе.

Как уже было сказано, цель книги поменялась. Я убеждён, что новичку следует дать фундамент, освоив который, он сможет уже самостоятельно изучать то, что нужно именно ему. Я больше не хочу давать читателям рыбу, я хочу дать им удочку. Поэтому здесь нет повествований обо всех имеющихся монадных трансформерах, или обо всех контейнерах, или о Кметтовских линзах, или о трубах Гонсалеса.

Я сделаю упор на теорию, но уже глубже. Так, в прошлом издании я часто использовал неточную терминологию, откровенно ступил с определением монады, прогнал какую-то пургу с ФВП, ни словом не обмолвился о функторных и иных законах, почти не рассказал о паттерн-матчинге и использовал мало примеров реального кода. В этом издании я постараюсь исправить все эти ошибки.

И я по-прежнему открыт для вашей критики.

Глава 4

Приготовимся

Мы не можем начать изучение языка без испытательного полигона. Установим Haskell.

Сделать это можно несколькими способами, мы выберем самый удобный. Называется он The Haskell Tool Stack. Эта маленькая утилита — всё, что вам понадобится для работы с Haskell.

Haskell — кроссплатформенный язык, работающий и в OS X, и в Linux, и даже в Windows. Однако в 2008 году я навсегда покинул мир Windows, поэтому все последующие примеры взаимодействия с командной строкой подразумевают Unix-way. Строго говоря, я уже и позабыл, что такое командная строка в Windows.

Вся конфигурация и примеры кода опробованы мною на OS X Yosemite.

Устанавливаем

Идём сюда и скачиваем архив для нужной нам ОС. Распаковываем архив и видим программку под названием `stack`. Для удобства располагаем её в каком-нибудь каталоге, доступном в `PATH`, и всё. Но если вы живёте в мире Mac и пользуетесь Homebrew, вам ещё проще. Делаете:

```
$ brew update
$ brew install haskell-stack
```

Всё.

На момент написания книги я использовал `stack` версии 1.0.2. Если у вас более старая версия — непременно обновитесь. Если же более новая — у вас теоретически что-нибудь может работать не в точности так, как описано ниже, поскольку `stack` активно развивается.

Главное (но не единственное), что умеет делать `stack`, это:

1. Разворачивать инфраструктуру.
2. Собирать проекты.
3. Устанавливать библиотеки.

Haskell-инфраструктура — экосистема, краеугольным камнем которой является ранее упомянутый компилятор GHC. Haskell является компилируемым языком: приложение представляет собой обыкновенный исполняемый (англ. executable) файл.

Haskell-проект — среда для создания приложений и библиотек.

Haskell-библиотеки — кем-то написанные решения, спасающие нас от изобретения велосипедов.

Разворачиваем инфраструктуру

Делаем:

```
$ stack setup
```

В результате на ваш компьютер будет установлена инфраструктура последней стабильной версии. Жить всё это хозяйство будет в только что созданном каталоге `~/.stack/`. Именно поэтому устанавливать инфраструктуру для последующих Haskell-проектов вам уже не придётся: единожды развернули, используем всегда. Пока вам не нужно знать об устройстве этой инфраструктуры, воспринимайте её как данность: теперь на вашем компьютере живёт Haskell.

Hi World

Создадим наш первый Haskell-проект:

```
$ stack new real
```

Здесь `real` — название проекта. В результате будет создан каталог `real`, внутри которого мы увидим это:

```
.
├── LICENSE
├── Setup.hs
├── app
│   └── Main.hs <- Главный модуль
├── real.cabal  <- Сборочный файл проекта
├── src
│   └── Lib.hs  <- Ещё один модуль
├── stack.yaml
└── test
    └── Spec.hs
```

О содержимом проекта вам пока знать не нужно, просто переходим в каталог `real` и собираем проект командой:

```
$ stack build
```

Запомните эту команду, мы будем использовать её постоянно. В результате сборки появится файл `real-exe`. Располагается он внутри скрытого каталога `.stack-work` в корне проекта. Чтобы сразу его запустить, не копаясь во внутренностях этого скрытого каталога, используем команду:

```
$ stack exec real-exe
someFunc
```

Команда `stack exec` запускает программу (в данном случае `real-exe`) в `stack`-окружении. В одной из последующих глав я подробнее расскажу об этом окружении. Впрочем, мы можем запустить нашу программу и напрямую, без `stack`. Исполняемый файл `real-exe` находится внутри скрытого каталога `.stack-work` в корне проекта. Например, на моём компьютере путь к исполняемому файлу такой:

```
.stack-work/dist/x86_64-osx/Cabal-1.22.4.0/build/real-exe/real-exe
```

Но можно и упростить себе жизнь, выполнив команду:

```
$ stack install
```

В результате исполняемый файл будет скопирован в каталог `~/.local/bin` (подразумевается, что такой каталог у вас уже имеется). Кстати, полезно добавить `~/.local/bin` в `PATH`, что позволит вам тут же запускать программу:

```
$ real-exe
someFunc
```

Вот мы и создали Haskell-проект и запустили нашу первую программу, выведшую строку `someFunc`. Но как же это работает? Пришла пора познакомиться с фундаментальной единицей проекта — модулем.

Модули: знакомство

Haskell-проект состоит из модулей. Модулем называется файл, содержащий исходный Haskell-код. Один файл — один модуль. Расширение `.hs` — стандартное расширения для модулей. В Haskell нет понятия «заголовочный файл»: каждый из модулей рассматривается как самостоятельная единица проекта, содержащая в себе разные полезные вещи. А чтобы воспользоваться этими вещами, необходимо один модуль импортировать в другой.

Откроем модуль `src/Lib.hs`:

```haskell
module Lib      -- Имя модуля
    ( someFunc  -- Интерфейс модуля
    ) where

someFunc :: IO ()
someFunc = putStrLn "someFunc"
```

В первой строке объявлено, что имя этого модуля — `Lib`. Далее в круглых скобках указан интерфейс данного модуля, то есть та его часть, которая видна всему миру. В данном случае это единственная функция `someFunc`, объявление и определение которой идёт далее, вслед за ключевым словом `where`. Пока вам не нужно знать о синтаксисе объявления и определений функции, в следующих главах мы разберём его тщательнейшим образом.

Теперь откроем модуль `app/Main.hs`:

```haskell
module Main where

import Lib      -- Импортируем модуль Lib...

main :: IO ()
main = someFunc -- Используем его содержимое...
```

Это модуль `Main`, главный модуль нашего приложения, ведь именно здесь определена функция `main`. С помощью директивы `import` мы включаем сюда модуль `Lib` и можем работать с содержимым этого модуля.

Запомните модуль `Main`, с ним мы будем работать чаще всего. Все примеры исходного кода, которые вы увидите на страницах этой книги, живут именно в модуле `Main`, если не оговорено иное.

Все модули в наших проектах можно разделить на две части: те, которые мы берём из библиотек и те, которые мы создали сами. Библиотеки — это уже кем-то написанные решения, в последующих главах мы познакомимся со многими из них. Среди библиотек следует выделить одну, так называемую стандартную библиотеку. Модули из стандартной библиотеки мы начнём использовать уже в ближайших главах. А одна из глав будет полностью посвящена рассказу о библиотеках: из неё мы подробно узнаем, откуда берутся библиотеки и как их можно использовать.

Для любопытных

До появления `stack` основным способом установки Haskell была так называемая Haskell Platform. Однако именно `stack`, несмотря на свою молодость (вышел в свет летом 2015 года), является предпочтительным путём в мир Haskell, особенно для новичков. Если вдруг так случилось, что к моменту прочтения этой главы у вас уже была установлена Haskell Platform — настоятельно рекомендую вам незамедлительно удалить это старьё. Если у вас OS X, вы можете воспользоваться вот этим

маленьким скриптом.

Как вы заметили, имена файлов с исходным кодом начинаются с большой буквы: `app/Main.hs` и `src/Lib.hs`. Строго говоря, это необязательно, можно и с маленькой буквы, однако для гармонии с именем модуля лучше придерживаться общепринятой практики и называть файл модуля по имени самого модуля:

```
app/Main.hs -> module Main ...
src/Lib.hs  -> module Lib ...
```

И ещё. При создании проекта мы могли бы использовать схему `simple` вместо предлагаемой по умолчанию. Для этого проект нужно было создать командой:

```
$ stack new real simple
```

где `simple` — имя схемы проекта. Дело в том, что команда `stack new` может создавать заготовки проектов для разных нужд. Простейшая из заготовок называется `simple`. В этом случае в проекте отсутствует модуль `src/Lib.hs`, а есть лишь `src/Main.hs`:

```
├── LICENSE
├── Setup.hs
├── real.cabal
├── src
│   └── Main.hs <- Единственный модуль
└── stack.yaml
```

Да, мы могли бы воспользоваться данной схемой, однако в этом случае мы не увидели бы механизма импорта одного модуля в другой. Я рад, что вы познакомились с импортом уже сейчас, ведь в последующих главах мы будем постоянно использовать различные модули из многих библиотек.

Глава 5

Киты и Черепаха

Итак, проект создали, теперь мы готовы начать наше путешествие.

Haskell стоит на Трёх Китах, имена которым: **Функция**, **Тип** и **Класс типов**. Они же, в свою очередь, покоятся на огромной Черепахе, имя которой — **Выражение**.

Черепаха

Haskell-программа представляет собой совокупность выражений (англ. expression). Взгляните:

```
1 + 2
```

Это — основной кирпич Haskell-программы, будь то Hello World или часть инфраструктуры международного банка. Конечно, помимо сложения единицы с двойкой существуют и другие выражения, но суть у них у всех одна:

> Выражение — это то, что может дать нам некий полезный результат.

Полезный результат мы получаем в результате вычисления (англ. evaluation) выражения. Все выражения можно вычислить, однако одни выражения в результате вычисления уменьшаются (англ. reduce), а другие — нет. Первые иногда называют редуцируемыми выражениями, а вторые — нередуцируемые. Так, выражение:

```
1 + 2
```

относится к редуцируемым, потому что оно в результате вычисления уменьшится и даст нам другое выражение:

```
3
```

Это выражение уже нельзя уменьшить, оно нередуцируемое и мы теперь лишь можем использовать его как есть.

Таким образом, выражения, составляющие программу, вычисляются/редуцируются до тех пор, пока не останется некое окончательное, корневое выражение. А запуск Haskell-программы на выполнение (англ. execution) — это запуск всей этой цепочки вычислений, причём с корнем этой цепочки мы уже познакомились ранее. Помните функцию `main`, определённую в модуле `app/Main.hs`? Вот эта функция и является главной точкой нашей программы, её Альфой и Омегой.

Первый Кит

Вернёмся к выражению `1 + 2`. Полезный результат мы получим лишь после того, как вычислим это выражение, то есть осуществим сложение. И как же можно «осуществить сложение» в рамках Haskell-программы? С помощью функции. Именно функция делает выражение вычислимым, именно она оживляет нашу программу, потому я и назвал Функцию Первым Китом Haskell. Но дабы избежать недоразумений, определимся с понятиями.

Что такое функция в математике? Вспомним школьный курс:

> Функция — это закон, описывающий зависимость одного значения от другого.

Рассмотрим функцию возведения целого числа в квадрат:

```
square v = v * v
```

Функция `square` определяет простую зависимость: числу `2` соответствует число `4`, числу `3` — `9`, и так далее. Схематично это можно записать так:

```
2 -> 4
3 -> 9
4 -> 16
5 -> 25
...
```

Входное значение функции называют аргументом. А так как функция определяет однозначную зависимость выходного значения от аргумента, её, функцию, называют ещё *отображением*: она отображает/проецирует входное значение на выходное. Получается как бы труба: кинули в неё `2` — с другой стороны вылетело `4`, кинули `5` — вылетело `25`.

Чтобы заставить функцию сделать полезную работу, её необходимо применить (англ. apply) к аргументу. Пример:

```
square 2
```

Мы применили функцию `square` к аргументу `2`. Синтаксис предельно прост: имя функции и через пробел аргумент. Если аргументов более одного — просто дописываем их так же, через пробел. Например, функция `sum`, вычисляющая сумму двух своих целочисленных аргументов, применяется так:

```
sum 10 20
```

Так вот выражение `1 + 2` есть ни что иное, как применение функции! И чтобы яснее это увидеть, перепишем выражение:

```
(+) 1 2
```

Это применение функции `(+)` к двум аргументам, `1` и `2`. Не удивляйтесь, что имя функции заключено в скобки, вскоре я расскажу об этом подробнее. А пока запомните главное:

> Вычислить выражение — это значит применить какие-то функции (одну или более) к каким-то аргументам (одному или более).

И ещё. Возможно, вы слышали о так называемом «вызове» функции. В Haskell функции не вызывают. Понятие «вызов» функции пришло к нам из почтенного языка C. Там функции действительно вызывают (англ. call), потому что в C, в отличие от Haskell, понятие «функция» не имеет никакого отношения к математике. Там это подпрограмма, то есть обособленный кусочек программы, доступный по некоторому адресу в памяти. Если у вас есть опыт разработки на C-подобных языках — забудьте о подпрограмме. В Haskell функция — это функция в математическом смысле слова, поэтому её не вызывают, а применяют к чему-то.

Второй Кит

Итак, любое редуцируемое выражение суть применение функции к некоторому аргументу (тоже являющемуся выражением):

```
square   2
функция  аргумент
```

Аргумент представляет собой некоторое значение, его ещё называют «данное» (англ. data). Данные в Haskell — это сущности, обладающие двумя главными характеристиками: типом и конкретным значением/содержимым.

Тип — это Второй Кит в Haskell. Тип отражает конкретное содержимое данных, а потому все данные в программе обязательно имеют некий тип. Когда мы видим данное типа `Double`, мы точно знаем, что перед нами число с плавающей точкой, а когда видим данные типа `String` — можем ручаться, что перед нами строки.

Отношение к типам в Haskell очень серьёзное, и работа с типами характеризуется тремя важными чертами:

1. статическая проверка,
2. сила,
3. выведение.

Три эти свойства системы типов Haskell — наши добрые друзья, ведь они делают нашу программистскую жизнь счастливее. Познакомимся с ними.

Статическая проверка

Статическая проверка типов (англ. static type checking) — это проверка типов всех данных в программе, осуществляемая на этапе компиляции. Haskell-компилятор упрям: когда ему что-либо не нравится в типах, он громко ругается. Поэтому если функция работает с целыми числами, применить её к строкам никак не получится. Так что если компиляция нашей программы завершилась успешно, мы точно знаем, что с типами у нас всё в порядке. Преимущества статической проверки невозможно переоценить, ведь она гарантирует отсутствие в наших программах целого ряда ошибок. Мы уже не сможем спутать числа со строками или вычесть метры из рублей.

Конечно, у этой медали есть и обратная сторона — время, затрачиваемое на компиляцию. Вам придётся свыкнуться с этой мыслью: внесли изменения в проект — будьте добры скомпилировать. Однако утешением вам пусть послужит тот факт, что преимущества статической проверки куда ценнее времени, потраченного на компиляцию.

Сила

Сильная (англ. strong) система типов — это бескомпромиссный контроль соответствия ожидаемого действительному. Сила делает работу с типами ещё более аккуратной. Вот вам пример из мира C:

```
double coeff(double base) {
    return base * 4.9856;
}

int main() {
    int value = coeff(122.04);
    ...
}
```

Это канонический пример проблемы, обусловленной слабой (англ. weak) системой типов. Функция `coeff` возвращает значение типа `double`, однако вызывающая сторона ожидает почему-то целое число. Ну вот ошиблись мы, криво скопировали. В этом случае произойдёт жульничество, называемое скрытым приведением типов (англ. implicit type casting): число с плавающей точкой, возвращённое функцией `coeff`, будет грубо сломано путём приведения его к типу `int`, в результате чего дробная часть будет отброшена и мы получим не `608.4426`, а `608`. Подобная ошибка, кстати, приводила к серьёзным последствиям, таким как уничтожение космических аппаратов. Нет, это вовсе не означает, что слабая типизация ужасна сама по себе, просто есть иной путь.

Благодаря сильной типизации в Haskell подобный код не имеет ни малейших шансов пройти компиляцию. Мы всегда получаем то, что ожидаем, и если должно быть

число с плавающей точкой — расшибись, но предоставь именно его. Компилятор скрупулёзно отслеживает соответствие ожидаемого типа фактическому, поэтому когда компиляция завершается успешно, мы абсолютно уверены в гармонии между типами всех наших данных.

Выведение

Выведение (англ. inference) типов — это способность определить тип данных автоматически, по конкретному выражению. В том же языке C тип данных следует указывать явно:

```
double value = 122.04;
```

однако в Haskell мы напишем просто:

```
value = 122.04
```

В этом случае компилятор автоматически выведет тип `value` как `Double`.

Выведение типов делает наш код лаконичнее и проще в сопровождении. Впрочем, мы можем указать тип значения и явно, а иногда даже должны это сделать. В последующих главах я объясню, почему.

Да, кстати, вот простейшие стандартные типы, они нам понадобятся:

```
123         Int
23.5798     Double
'a'         Char
"Hello!"    String
True        Bool, истина
False       Bool, ложь
```

С типами `Int` и `Double` вы уже знакомы. Тип `Char` — это Unicode-символ. Тип `String` — строка, состоящая из Unicode-символов. Тип `Bool` — логический тип, соответствующий истине или лжи. В последующих главах мы встретимся ещё с несколькими стандартными типами, но пока хватит и этих. И заметьте: имя типа в Haskell всегда начинается с большой буквы.

Третий Кит

А вот о ТретьемКите, о **Классе типов**, я пока умолчу, потому что знакомиться с ним следует лишь после того, как мы поближе подружимся с первыми двумя.

Уверен, после прочтения этой главы у вас появилось множество вопросов. Ответы будут, но позже. Более того, следующая глава несомненно удивит вас.

Для любопытных

Если вы работали с объектно-ориентированными языками, такими как C++, вас удивит тот факт, что в Haskell между понятиями «тип» и «класс» проведено чёткое различие. А поскольку типам и классам типов в Haskell отведена колоссально важная роль, добрый вам совет: когда в будущих главах мы познакомимся с ними поближе, не пытайтесь проводить аналогии из других языков. Например, некоторые усматривают родство между классами типов в Haskell и интерфейсами в Java. Не делайте этого, во избежание путаницы.

Глава 6

Неизменность и чистота

В предыдущей главе мы познакомились с функциями и выражениями, увидев близкую связь этих понятий. В этой главе мы познакомимся с функциями поближе, а также узнаем, что такое «чисто функциональный» язык и почему в нём нет места оператору присваивания.

Объявляем и определяем

Применение функции нам уже знакомо, осталось узнать про объявление и определение, без них использовать функцию не получится. Помните функцию `square`, возводящую свой единственный аргумент в квадрат? Вот как выглядит её объявление и определение:

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

Первая строка содержит объявление, вторая — определение. Объявление (англ. declaration) — это весть всему миру о том, что такая функция существует, вот её имя и вот типы, с которыми она работает. Определение (англ. definition) — это весть о том, что конкретно делает данная функция.

Рассмотрим объявление:

```
square :: Int -> Int
```

Оно разделено двойным двоеточием на две части: слева указано имя функции, справа — типы, с которыми эта функция работает, а именно типы аргументов и тип вычисленного, итогового значения. Как вы узнали из предыдущей главы, все данные в Haskell-программе имеют конкретный тип, а поскольку функция работает с данными, её объявление содержит типы этих данных. Типы разделены стрелками. Схематично это выглядит так:

```
square :: Int      -> Int

имя       тип         тип
функции   аргумента  вычисленного
                      значения
```

Такое объявление сообщает нам о том, что функция `square` принимает единственный аргумент типа `Int` и возвращает значение того же типа `Int`. Если же аргументов более одного, объявление просто вытягивается. Например, объявление функции `prod`, возвращающей произведение двух целочисленных аргументов, могло бы выглядеть так:

```
prod    :: Int      -> Int      -> Int

имя       тип         тип         тип
функции   первого     второго     вычисленного
          аргумента   аргумента   значения
```

Идею вы поняли: ищем крайнюю правую стрелку, и всё что левее от неё — то типы аргументов, а всё что правее — то тип вычисленного значения.

Мы не можем работать с функцией, которая ничего не вычисляет. То есть аналога C-функции `void f(int i)` в Haskell быть не может, так как это противоречит математической природе. Однако мы можем работать с функцией, которая ничего не принимает, то есть с аналогом C-функции `int f(void)`. С такими функциями мы познакомимся в следующих главах.

Теперь рассмотрим определение функции `square`:

```
square v = v * v
```

Схема определения такова:

```
square   v          =    v * v

имя      имя        это  выражение
функции  аргумента
```

А функция `prod` определена так:

```
prod     x          y         =    x * y

имя      имя        имя       это  выражение
функции  первого    второго
         аргумента  аргумента
```

Определение тоже разделено на две части: слева от знака равенства — имя функции и имена аргументов (имена, а не типы), разделённые пробелами, а справа —

выражение, составляющее суть функции, её содержимое. Иногда эти части называют «головой» и «телом»:

```
square   v      =  v * v

голова функции     тело функции
(англ. head)       (англ. body)
```

Обратите внимание, речь здесь идёт именно о знаке равенства, а никак не об операторе присваивания. Мы ничего не присваиваем, мы лишь декларируем равенство левой и правой частей. Когда мы пишем:

```
prod x y = x * y
```

мы объявляем следующее: «Отныне выражение `prod x y` равно выражению `x * y`». Мы можем безопасно заменить выражение `prod 2 5` выражением `2 * 5`, а выражение `prod 120 500` — выражением `120 * 500`, и при этом работа программы гарантированно останется неизменной.

Но откуда у меня такая уверенность? А вот откуда.

Чисто функциональный

Haskell — чисто функциональный (англ. purely functional) язык. Чисто функциональным он называется потому, что центральное место в нём уделено чистой функции (англ. pure function). А чистой называется такая функция, которая предельно честна с нами: её выходное значение всецело определяется её аргументами и более ничем. Это и есть функция в математическом смысле. Вспомним функцию `prod`: когда на входе числа `10` и `20` — на выходе всегда будет `200`, и ничто не способно помешать этому. Функция `prod` является чистой, а потому характеризуется отсутствием побочных эффектов (англ. side effects): она не способна сделать ничего, кроме как вернуть произведение двух своих аргументов. Именно поэтому чистая функция предельно надёжна, ведь она не может преподнести нам никаких сюрпризов.

Скажу больше: чистые функции не видят окружающий мир. Вообще. Они не могут вывести текст на консоль, их нельзя заставить обработать HTTP-запрос, они не умеют дружить с базой данных и прочесть файл они также неспособны. Они суть вещь в себе.

А чтобы удивить вас ещё больше, открою ещё один секрет Haskell.

«Присваивание? Не, не слышал…»

В мире Haskell нет места оператору присваивания. Впрочем, этот факт удивителен лишь на первый взгляд. Задумаемся: если каждая функция в конечном итоге представляет собою выражение, вычисляемое посредством применения каких-то

других функций к каким-то другим аргументам, тогда нам просто не нужно ничего ничему присваивать.

Вспомним, что присваивание (англ. assignment) пришло к нам из императивных языков. Императивное программирование (англ. imperative programming) — это направление в разработке, объединяющее несколько парадигм программирования, одной из которых является знаменитая объектно-ориентированная парадигма. В рамках этого направления программа воспринимается как набор инструкций, выполнение которых неразрывно связано с изменением состояния (англ. state) этой программы. Вот почему в императивных языках обязательно присутствует понятие «переменная» (англ. variable). А раз есть переменные — должен быть и оператор присваивания. Когда мы пишем:

```
coeff = 0.569;
```

мы тем самым приказываем: «Возьми значение `0.569` и перезапиши им то значение, которое уже содержалось в переменной `coeff` до этого». И перезаписывать это значение мы можем множество раз, а следовательно, мы вынуждены внимательно отслеживать текущее состояние переменной `coeff`, равно как и состояния всех остальных переменных в нашем коде.

Однако существует принципиально иной подход к разработке, а именно декларативное программирование (англ. declarative programming). Данное направление также включает в себя несколько парадигм, одной из которых является функциональная парадигма, нашедшая своё воплощение в Haskell. При этом подходе программа воспринимается уже не как набор инструкций, а как набор выражений. А поскольку выражения вычисляются путём применения функций к аргументам (то есть, по сути, к другим выражениям), там нет места ни переменным, ни оператору присваивания. Все данные в Haskell-программе, будучи созданными единожды, уже не могут быть изменены. Поэтому нам не нужен не только оператор присваивания, но и ключевое слово `const`. И когда в Haskell-коде мы пишем:

```
coeff = 0.569
```

мы просто объявляем: «Отныне значение `coeff` равно `0.569`, и так оно будет всегда». Вот почему в Haskell-коде символ `=` — это знак равенства в математическом смысле, и с присваиванием он не имеет ничего общего.

Уверен, вы удивлены. Как же можно написать реальную программу на языке, в котором нельзя изменять данные? Какой прок от этих чистых функций, если они не способны ни файл прочесть, ни запрос по сети отправить? Оказывается, прок есть, и на Haskell можно написать очень даже реальную программу. За примером далеко ходить не буду: сама эта книга построена с помощью программы, написанной на Haskell, о чём я подробнее расскажу в следующих главах.

А теперь, дабы не мучить вас вопросами без ответов, мы начнём ближе знакомиться с Китами Haskell, и детали большой головоломки постепенно сложатся в красивую картину.

Для любопытных

В процессе работы Haskell-программы в памяти создаётся великое множество различных данных, ведь мы постоянно строим новые данные на основе уже имеющихся. За их своевременное уничтожение отвечает сборщик мусора (англ. garbage collector, GC), встраиваемый в программы компилятором GHC.

Глава 7

Выбираем и возвращаемся

В этой главе мы встретимся с условными конструкциями, выглянем в терминал, а также узнаем, почему из Haskell-функций не возвращаются (впрочем, последнее — не более чем игра слов).

Выглянем во внешний мир

Мы начинаем писать настоящий код. А для этого нам понадобится окно во внешний мир. Откроем модуль `app/Main.hs`, найдём функцию `main` и напишем в ней следующее:

```
main :: IO ()
main = putStrLn "Hi, real world!"
```

Стандартная функция `putStrLn` выводит строку на консоль. А если говорить строже, функция `putStrLn` применяется к значению типа `String` и делает так, чтобы мы увидели это значение в нашем терминале.

Да, я уже слышу вопрос внимательного читателя. Как же так, спросите вы, разве мы не говорили о чистых функциях в прошлой главе, неспособных взаимодействовать с внешним миром? Придётся признаться: функция `putStrLn` относится к особым функциям, которые могут-таки вылезти во внешний мир. Но об этом в следующих главах. Это прелюбопытнейшая тема, поверьте мне!

И ещё нам следует познакомиться с Haskell-комментариями, они нам понадобятся:

```
{-
    Я - сложный многострочный
     комментарий, содержащий
  нечто
        очень важное!
-}
main :: IO ()
main =
  -- А я - скромный однострочный комментарий.
  putStrLn "Hi, real world!"
```

Символы `{-` и `-}` скрывают многострочный комментарий, а символ `--` начинает комментарий однострочный.

На всякий случай напоминаю команду сборки, запускаемую из корня проекта:

```
$ stack build
```

После сборки запускаем:

```
$ stack exec real-exe
Hi, real world!
```

Выбор и выход

Выбирать внутри функции приходится очень часто. Существует несколько способов задания условной конструкции. Вот базовый вариант:

```
if CONDITION then EXPR1 else EXPR2
```

где `CONDITION` — логическое выражение, дающее ложь или истину, `EXPR1` — выражение, используемое в случае `True`, `EXPR2` — выражение, используемое в случае `False`. Пример:

```
checkLocalhost :: String -> String
checkLocalhost ip =
  -- True или False?
  if ip == "127.0.0.1" || ip == "0.0.0.0"
    -- Если True - идём туда...
    then "It's a localhost!"
    -- А если False - сюда...
    else "No, it's not a localhost."
```

Функция `checkLocalhost` применяется к единственному аргументу типа `String` и возвращает другое значение типа `String`. В качестве аргумента выступает строка,

содержащая IP-адрес, а функция проверяет, не лежит ли в ней localhost. Оператор `||` — стандартый оператор логического «ИЛИ», а оператор `==` — стандартный оператор проверки на равенство. Итак, если строка `ip` равна `127.0.0.1` или `0.0.0.0`, значит в ней localhost, и мы возвращаем первое выражение, то есть строку `It's a localhost!`, в противном случае возвращаем второе выражение, строку `No, it's not a localhost.`.

А кстати, что значит «возвращаем»? Ведь, как мы узнали, функции в Haskell не вызывают (англ. call), а значит, из них и не возвращаются (англ. return). И это действительно так. Если напишем:

```
main :: IO ()
main = putStrLn (checkLocalhost "127.0.0.1")
```

при запуске увидим это:

```
It's a localhost!
```

а если так:

```
main :: IO ()
main = putStrLn (checkLocalhost "173.194.22.100")
```

тогда увидим это:

```
No, it's not a localhost.
```

Круглые скобки включают выражение типа `String` по схеме:

```
main :: IO ()
main = putStrLn (checkLocalhost "173.194.22.100")

                └── выражение типа String ──┘
```

То есть функция `putStrLn` видит не применение функции `checkLocalhost` к строке, а просто выражение типа `String`. Если бы мы опустили скобки и написали так:

```
main :: IO ()
main = putStrLn checkLocalhost "173.194.22.100"
```

произошла бы ошибка компиляции, и это вполне ожидаемо: функция `putStrLn` применяется к одному аргументу, а тут их получается два:

```
main = putStrLn     checkLocalhost  "173.194.22.100"

       функция      к этому
       применяется  аргументу...
                                    и к этому??
```

Не знаю как вы, а я не очень люблю круглые скобки, при всём уважении к Lisp-программистам. К счастью, в Haskell существует способ уменьшить число скобок. Об этом способе — в одной из последующих глав.

Так что же с возвращением из функции? Вспомним о равенстве в определении:

```
checkLocalhost ip =
  if ip == ”127.0.0.1” || ip == ”0.0.0.0”
    then ”It’s a localhost!”
    else ”No, it’s not a localhost.”
```

То, что слева от знака равенства, равно тому, что справа. А раз так, эти два кода эквивалентны:

```
main :: IO ()
main = putStrLn (checkLocalhost ”173.194.22.100”)
```

```
main :: IO ()
main =
  putStrLn (if ”173.194.22.100” == ”127.0.0.1” ||
                ”173.194.22.100” == ”0.0.0.0”
              then ”It’s a localhost!”
              else ”No, it’s not a localhost.”)
```

Мы просто заменили применение функции `checkLocalhost` её внутренним выражением, подставив вместо аргумента `ip` конкретную строку `173.194.22.100`. В итоге, в зависимости от истинности или ложности проверок на равенство, эта условная конструкция будет также заменена одним из двух выражений. В этом и заключается идея: возвращаемое функцией значение — это её последнее, итоговое выражение. То есть если выражение:

```
”173.194.22.100” == ”127.0.0.1” ||
”173.194.22.100” == ”0.0.0.0”
```

даст нам результат `True`, то мы переходим к выражению из логической ветви `then`. Если же оно даст нам `False` — мы переходим к выражению из логической ветви `else`. Это даёт нам право утверждать, что условная конструкция вида:

```
if True
  then ”It’s a localhost!”
  else ”No, it’s not a localhost.”
```

может быть заменена на первое нередуцируемое выражение, строку `It’s a localhost!`, а условную конструкцию вида:

```
if False
  then "It's a localhost!"
  else "No, it's not a localhost."
```

можно спокойно заменить вторым нередуцируемым выражением, строкой `No, it's not a localhost.`. Поэтому код:

```
main :: IO ()
main = putStrLn (checkLocalhost "0.0.0.0")
```

эквивалентен коду:

```
main :: IO ()
main = putStrLn "It's a localhost!"
```

Аналогично, код:

```
main :: IO ()
main = putStrLn (checkLocalhost "173.194.22.100")
```

есть ни что иное, как:

```
main :: IO ()
main = putStrLn "No, it's not a localhost."
```

Каким бы сложным ни было логическое ветвление внутри функции `checkLocalhost`, в конечном итоге оно вернёт/вычислит какое-то одно итоговое выражение. Именно поэтому из функции в Haskell нельзя выйти в произвольном месте, как это принято в императивных языках, ведь она не является набором инструкций, она — выражение, состоящее из других выражений. Вот почему функции в Haskell так просто компоновать друг с другом, и позже мы встретим множество таких примеров.

Для любопытных

Внимательный читатель несомненно заметил необычное объявление главной функции нашего проекта, функции `main`:

```
main :: IO ()   -- Объявление?
main = putStrLn ...
```

Если `IO` — это тип, то что такое `()`? И почему указан лишь один тип? Что такое `IO ()`: аргумент функции `main`, или же то, что она вычисляет? Сожалею, но пока я вынужден сохранить это в секрете. Когда мы поближе познакомимся со Вторым Китом Haskell, я непременно расскажу про этот странный `IO ()`.

Глава 8

Выбор и образцы

Эта глава откроет нам другие способы выбора, а также познакомит нас с образцами. Уверяю, вы влюбитесь в них!

Не только из двух

Часто мы хотим выбирать не только из двух возможных вариантов. Вот как это можно сделать:

```
analyzeGold :: Int -> String
analyzeGold standard =
  if standard == 999
    then "Wow! 999 standard!"
    else if standard == 750
           then "Great! 750 standard."
           else if standard == 585
                  then "Not bad! 585 standard."
                  else "I don't know such a standard..."

main :: IO ()
main = putStrLn (analyzeGold 999)
```

Уверен, вы уже стираете плевок с экрана. Вложенная `if-then-else` конструкция не может понравиться никому, ведь она крайне неудобна в обращении. А уж если бы анализируемых проб золота было штук пять или семь, эта лестница стала бы поистине ужасной. К счастью, в Haskell можно написать по-другому:

```
analyzeGold :: Int -> String
analyzeGold standard =
  if | standard == 999 -> "Wow! 999 standard!"
     | standard == 750 -> "Great! 750 standard."
     | standard == 585 -> "Not bad! 585 standard."
     | otherwise -> "I don't know such a standard..."
```

Не правда ли, так красивее? Это — множественный `if`. Работает он по схеме:

```
if | COND1 -> EXPR1
   | COND2 -> EXPR2
   | ...
   | CONDn -> EXPRn
   | otherwise -> COMMON_EXPR
```

где `COND1..n` — выражения, дающие ложь или истину, а `EXPR1..n` — соответствующие им результирующие выражения. Особая функция `otherwise` соответствует общему случаю, когда ни одно из логических выражений не дало `True`, и в этой ситуации результатом условной конструкции послужит выражение `COMMON_EXPR`.

Не пренебрегайте `otherwise`! Если вы его не укажете и при этом примените функцию `analyzeGold` к значению, отличному от проверяемых:

```
analyzeGold :: Int -> String
analyzeGold standard =
  if | standard == 999 -> "Wow! 999 standard!"
     | standard == 750 -> "Great! 750 standard."
     | standard == 585 -> "Not bad! 585 standard."

main :: IO ()
main = putStrLn (analyzeGold 583)  -- Ой...
```

компиляция завершится успешно, однако в момент запуска программы вас ожидает неприятный сюрприз в виде ошибки:

```
Non-exhaustive guards in multi-way if
```

Проверка получилась неполной, вот и получите ошибку.

Кстати, видите слово `guards` в сообщении об ошибке? Вертикальные черты перед логическими выражениями — это и есть охранники (англ. guard), неусыпно охраняющие наши условия. Потешное название выбрали. Чтобы читать их было легче, воспринимайте их как аналог слова «ИЛИ».

А сейчас стоп. Вы ведь попробовали скомпилировать этот код, не так ли? А почему вы не ругаетесь? Ведь такой код не скомпилируется, так как не хватает одной маленькой, но важной детали. Вот как должен выглядеть модуль `Main`:

```haskell
{-# LANGUAGE MultiWayIf #-}  -- Что это??

module Main where

analyzeGold :: Int -> String
analyzeGold standard =
  if | standard == 999 -> "Wow! 999 standard!"
     | standard == 750 -> "Great! 750 standard."
     | standard == 585 -> "Not bad! 585 standard."
     | otherwise -> "I don't know such a standard..."

main :: IO ()
main = putStrLn (analyzeGold 999)
```

Вот теперь всё в порядке. Но что это за странный комментарий в первой строке модуля? Вроде бы оформлен как многострочный комментарий, но выглядит необычно. Перед нами — указание расширения языка Haskell.

Стандарт Haskell 2010 — это официальный стержень языка. Однако компилятор GHC, давно уж ставший компилятором по умолчанию при разработке на Haskell, обладает рядом особых возможностей. По умолчанию многие из этих возможностей выключены, а прагма `LANGUAGE` как раз для того и предназначена, чтобы их включать/активизировать. В данном случае мы включили расширение `MultiWayIf`. Именно это расширение позволяет нам использовать множественный `if`. Такого рода расширений существует очень много, и мы будем часто их использовать.

Помните: расширение, включённое с помощью прагмы `LANGUAGE`, действует лишь в рамках текущего модуля. И если я прописал его только в модуле `app/Main.hs`, то на модуль `src/Lib.hs` механизм `MultiWayIf` не распространяется.

Без Если

Множественный `if` весьма удобен, но есть способ более красивый. Взгляните:

```haskell
analyzeGold :: Int -> String
analyzeGold standard
  | standard == 999 = "Wow! 999 standard!"
  | standard == 750 = "Great! 750 standard."
  | standard == 585 = "Not bad! 585 standard."
  | otherwise = "I don't know such a standard..."
```

Ключевое слово `if` исчезло. Схема здесь такая:

```
function arg  -- Нет знака равенства?
  | COND1 = EXPR1
  | COND2 = EXPR2
  | ...
  | CONDn = EXPRn
  | otherwise = COMMON_EXPR
```

Устройство почти такое же, но, помимо исчезновения ключевого слова `if`, мы теперь используем знаки равенства вместо стрелок. Именно поэтому исчез знакомый нам знак равенства после имени аргумента `arg`. В действительности он, конечно, никуда не исчез, он лишь перешёл в выражения. А чтобы это легче прочесть, напишем выражения в строчку:

```
function arg  |    COND1    =     EXPR1     |    ...

эта           или           равна
функция
                                   этому
                                   выражению

                  в случае
                  истинности
                  этого
                  выражения
                                          или  и т.д.
```

То есть перед нами уже не одно определение функции, а цепочка определений, потому нам и не нужно ключевое слово `if`. Но и эту цепочку определений можно упростить.

Сравнение с образцом

Убрав слово `if`, мы и с нашими виртуальными «ИЛИ» можем расстаться. В этом случае останется лишь это:

```
analyzeGold :: Int -> String  -- Одно объявление.
-- И множество определений...
analyzeGold 999 = "Wow! 999 standard!"
analyzeGold 750 = "Great! 750 standard."
analyzeGold 585 = "Not bad! 585 standard."
analyzeGold _   = "I don't know such a standard..."
```

Мы просто перечислили определения функции `analyzeGold` одно за другим. На первый взгляд, возможность множества определений одной и той же функции удивляет, но если вспомнить, что применение функции суть выражение, тогда ничего удивительного. Вот как это читается:

```
      analyzeGold  999          =      "Wow! 999 standard!"

если  эта функция  применяется  тогда  этому выражению
                   к этому      она
                   аргументу    равна

      analyzeGold  750          =      "Wow! 999 standard!"

если  эта функция  применяется  тогда  другому выражению
                   к другому    она
                   аргументу    равна
...
       analyzeGold _ =      "I don't know such a standard..."

иначе  эта функция   равна  общему выражению
```

Когда функция `analyzeGold` применяется к конкретному аргументу, этот аргумент последовательно сравнивается с образцом (англ. pattern matching). Образца здесь три: `999`, `750` и `585`. И если раньше мы сравнивали аргумент с этими числовыми значениями явно, посредством функции `==`, теперь это происходит скрыто. Идея сравнения с образцом очень проста: что-то (в данном случае реальный аргумент) сопоставляется с образцом (или образцами) на предмет «подходит/не подходит». Если подходит — то есть сравнение с образцом даёт результат `True` — готово, используем соответствующее выражение. Если же не подходит — переходим к следующему образцу.

Сравнение с образцом, называемое ещё «сопоставлением с образцом» используется в Haskell чрезвычайно широко. В русскоязычной литературе перевод словосочетания «pattern matching» не особо закрепился, вместо этого так и говорят «паттерн матчинг». Я поступлю так же.

Но что это за символ подчёркивания такой, в последнем варианте определения? Вот этот:

```
analyzeGold _ = "I don't know such a standard..."
            ^
```

С формальной точки зрения, это — универсальный образец, сравнение с которым всегда истинно (ещё говорят, что с ним матчится (англ. match) всё что угодно). А с неформальной — это символ, который можно прочесть как «мне всё равно». Мы как бы говорим: «В данном случае нас не интересует конкретное содержимое аргумента, нам всё равно, мы просто возвращаем строку `I don't know such a standard...`».

Важно отметить, что сравнение аргумента с образцами происходит последовательно, сверху вниз. Поэтому если мы напишем так:

```
analyzeGold :: Int -> String
analyzeGold _   = "I don't know such a standard..."
analyzeGold 999 = "Wow! 999 standard!"
analyzeGold 750 = "Great! 750 standard."
analyzeGold 585 = "Not bad! 585 standard."
```

наша функция будет всегда возвращать первое выражение, строку `I don't know such a standard...`, и это вполне ожидаемо: первая же проверка гарантированно даст нам `True`, ведь с образцом _ совпадает всё что угодно. Таким образом, общий образец следует располагать в самом конце, чтобы мы попали на него лишь после того, как не сработали все остальные образцы.

case

Существует ещё один вид паттерн матчинга, с помощью конструкции `case-of`:

```
analyzeGold standard =
  case standard of
    999 -> "Wow! 999 standard!"
    750 -> "Great! 750 standard."
    585 -> "Not bad! 585 standard."
    _   -> "I don't know such a standard..."
```

Запомните конструкцию `case-of`, мы встретимся с нею не раз. Работает она по модели:

```
case EXPRESSION of
  PATTERN1 -> EXPR1
  PATTERN2 -> EXPR2
  ...
  PATTERNn -> EXPRn
  _        -> COMMON_EXPR
```

где `EXPRESSION` — анализируемое выражение, последовательно сравниваемое с образцами `PATTERN1..n`. Если ни одно не сработало — как обычно, упираемся в универсальный образец _ и выдаём `COMMON_EXPR`.

В последующих главах мы встретимся и с другими видами паттерн матчинга, ведь он используется не только для выбора.

Глава 9

Пусть будет там, Где…

В этой главе мы узнаем, как сделать наши функции более удобными и читабельными.

Пусть

В нижеследующих примерах мы вновь будем использовать расширение GHC `MultiWayIf`, не забудьте включить его. Рассмотрим следующую функцию:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  if | timeInS <  40 -> timeInS + 120
     | timeInS >= 40 -> timeInS + 8 + 120
```

Мы считаем время некоторого события, и если исходное время меньше `40` секунд — результирующее время увеличено на `120` секунд, в противном случае — ещё на `8` секунд сверх того. Перед нами классический пример «магических чисел» (англ. magic numbers), когда смысл конкретных значений скрыт за семью печатями. Что за `40`, и что за `8`? Во избежание этой проблемы можно ввести временные выражения, и тогда код станет совсем другим:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  let threshold  = 40
      correction = 120
      delta      = 8
  in
  if | timeInS <  threshold -> timeInS + correction
     | timeInS >= threshold -> timeInS + delta + correction
```

Вот, совсем другое дело! Мы избавились от «магических чисел», введя поясняющие выражения `threshold`, `correction` и `delta`. Конструкция `let-in` вводит поясняющие выражения по схеме:

```
let DECLARATIONS in EXPRESSION
```

где `DECLARATIONS` — выражения, декларируемые нами, а `EXPRESSION` — выражение, в котором используется выражения из `DECLARATION`. Когда мы написали:

```
let threshold = 40
```

мы объявили: «Отныне выражение `threshold` равно выражению `40`». Выглядит как присваивание, но мы-то уже знаем, что в Haskell его нет. Теперь выражение `threshold` может заменить собою число `40` внутри выражения, следующего за словом `in`:

```
  let threshold = 40
      ...
  in if | timeInS <  threshold -> ...
        | timeInS >= threshold -> ...
```

Эта конструкция легко читается:

```
let    threshold  =      40       ... in ...

пусть  это        будет  этому        в  том
       выражение  равно  выражению       выражении
```

С помощью ключевого слова `let` можно ввести сколько угодно пояснительных/промежуточных выражений, что делает наш код понятнее, а во многих случаях ещё и короче.

И кстати, мы ведь можем упростить условную конструкцию, воспользовавшись `otherwise`:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  let threshold  = 40
      correction = 120
      delta      = 8
  in
  if | timeInS < threshold -> timeInS + correction
     | otherwise -> timeInS + delta + correction
```

Важно помнить, что введённое конструкцией `let-in` выражение существует лишь в рамках выражения, следующего за словом `in`. Изменим функцию:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  let threshold  = 40
      correction = 120
  in
  if | timeInS < threshold -> timeInS + correction
     | otherwise ->
       let delta     = 8 in timeInS
                            + delta
                            + correction

           это              существует лишь в
           выражение        рамках этого выражения
```

Мы сократили область видимости промежуточного выражения `delta`, сделав его видимым лишь в выражении `timeInS + delta + correction`.

При желании `let`-выражения можно записывать и в строчку:

```
  ...
  let threshold = 40; correction = 120
  in
  if | timeInS < threshold -> timeInS + correction
     | otherwise ->
       let delta = 8 in timeInS + delta + correction
```

В этом случае мы перечисляем их через точку с запятой. Лично мне такой стиль не нравится, но выбирать вам.

Где

Существует иной способ введения промежуточных выражений, взгляните:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  if | timeInS < threshold -> timeInS + correction
     | otherwise -> timeInS +
                    delta +
                    correction
  where
    threshold  = 40
    correction = 120
    delta      = 8
```

Ключевое слово `where` делает примерно то же, что и `let`, но промежуточные выражения задаются в конце функции. Такая конструкция читается подобно научной формуле:

```
  S = V * t,      -- Выражение
где
  -- Всё то, что
  -- используется
  -- в выражении.
  S = расстояние,
  V = скорость,
  t = время.
```

В отличие от `let`, которое может быть использовано для введения суперлокального выражение (как в последнем примере с `delta`), все `where`-выражения доступны в любой части выражения, предшествующего ключевому слову `where`.

Вместе

Мы можем использовать `let-in` и `where` совместно, в рамках одной функции:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  let threshold = 40 in
  if | timeInS < threshold -> timeInS + correction
     | otherwise -> timeInS + delta + correction
  where
    correction = 120
    delta      = 8
```

Часть промежуточных значений задана вверху, а часть — внизу. Общая рекомендация: не смешивайте `let-in` и `where` без особой надобности, такой код читается тяжело, избыточно.

Отмечу, что в качестве промежуточных могут выступать и более сложные выражения. Например:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  let threshold = 40 in
  if | timeInS < threshold -> timeInS + correction
     | otherwise -> timeInS + delta + correction
  where
    -- Это промежуточное выражение зависит от аргумента...
    correction = timeInS * 2
    -- А это - от другого выражения...
    delta      = correction - 4
```

Выражение `correction` равно `timeInS * 2`, то есть теперь оно зависит от значения аргумента функции. А выражение `delta` зависит в свою очередь от `correction`. Причём мы можем менять порядок задания выражений:

```
...
let threshold = 40
in
if | timeInS < threshold -> timeInS + correction
   | otherwise -> timeInS + delta + correction
where
  delta      = correction - 4
  correction = timeInS * 2
```

Выражение `delta` теперь задано первым по счёту, но это не имеет никакого значения. Ведь мы всего лишь объявляем равенства, и результат этих объявлений не влияет на конечный результат вычислений. Конечно, порядок объявления равенств не важен и для `let`-выражений:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  let delta     = correction - 4
      threshold = 40
  in
  if | timeInS < threshold -> timeInS + correction
     | otherwise -> timeInS + delta + correction
  where
    correction = timeInS * 2
```

Мало того, что мы задали `let`-выражения в другом порядке, так мы ещё и использовали в одном из них выражение `correction`! То есть в `let`-выражении использовалось `where`-выражение. А вот проделать обратное, увы, не получится:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  let delta     = correction - 4
      threshold = 40
  in
  if | timeInS < threshold -> timeInS + correction
     | otherwise -> timeInS + delta + correction
  where
    correction = timeInS * 2 * threshold -- Из let??
```

При попытке скомпилировать такой код мы получим ошибку:

```
Not in scope: ‘threshold’
```

Таково ограничение: использовать `let`-выражения внутри `where`-выражений невозможно, ведь последние уже не входят в выражение, следующее за словом `in`.

Ну что ж, пора двигаться дальше, ведь внутренности наших функций не ограничены условными конструкциями.

Глава 10

Мир операторов

Оператор (англ. operator) — частный случай функции. В предыдущих главах мы уже познакомились с ними, осталось объяснить подробнее.

Вспомним наше самое первое выражение:

```
1 + 2
```

Функция `+` записана в инфиксной (англ. infix) форме, то есть между своими аргументами. Такая запись выглядит естественнее, нежели обычная:

```
(+) 1 2
```

Видите круглые скобки? Они говорят о том, что данная функция предназначена для инфиксной записи. Автор этой функции изначально рассчитывал на инфиксную форму использования `1 + 2`, а не на обычную `(+) 1 2`, именно поэтому имя функции в определении заключено в круглые скобки:

```
(+) :: ...
```

Функции, предназначенные для инфиксной формы применения, называют операторами.

Если же имя функции не заключено в круглые скобки, подразумевается, что мы рассчитываем на обычную форму её применения. Однако и в этом случае можно применять её инфиксно, но имя должно заключаться в обратные одинарные кавычки (англ. backtick).

Определим функцию `isEqualTo`, являющуюся аналогом оператора проверки на равенство для двух целочисленных значений:

```
isEqualTo :: Int -> Int -> Bool
isEqualTo x y = x == y
```

При обычной форме её применение выглядело бы так:

```
...
if isEqualTo code1 code2 then ... else ...
where code1 = 123
      code2 = 124
...
```

Но давайте перепишем в инфиксной форме:

```
...
if code1 `isEqualTo` code2 then ... else ...
where code1 = 123
      code2 = 124
...
```

Гораздо лучше, ведь теперь код читается как обычный английский текст:

```
...
if code1 `isEqualTo` code2 ...
if code1 is equal to code2 ...
...
```

Строго говоря, название «оператор» весьма условно, мы можем его и не использовать. Говорить о функции сложения столь же корректно, как и об операторе сложения.

Зачем это нужно?

Почти все ASCII-символы (а также их всевозможные комбинации) можно использовать в качестве операторов в Haskell. Это даёт нам широкие возможности для реализации различных EDSL (англ. Embedded Domain Specific Language), своего рода «языков в языке». Вот пример:

```
div ! class_ "nav-wrapper" $
  a ! class_ "brand-logo sans" ! href "/" $
    "#ohaskell"
```

Любой, кто знаком с веб-разработкой, мгновенно узнает в этом коде HTML. Это кусочек кода, строящего HTML-шаблон для веб-варианта данной книги. То что вы видите — это совершенно легальный Haskell-код, в процессе работы которого генерируется реальный HTML: тег `<div>` с классом `nav-wrapper`, внутри которого лежит `<a>`-ссылка с двумя классами, корневым адресом и внутренним текстом `#ohaskell`.

Идентификаторы `div`, `class_` и `href` — это имена функций, а символы `!` и `$` — это операторы, записанные в инфиксной форме. Самое главное, что для понимания

этого кода нам абсолютно необязательно знать, где определены все эти функции/операторы и как они работают. Это важная мысль, которую я неоднократно буду повторять в последующих главах:

> Чтобы использовать функции, нам вовсе необязательно знать их внутренности.

А про EDSL запомните, мы с ними ещё встретимся.

Глава 11

Список

Помните, в одной из предыдущих глав я говорил, что познакомлю вас ещё с несколькими стандартными типами данных в Haskell? Пришло время узнать о списках.

Список (англ. list) — это стандартный тип, характеризующий уже не просто данные, но структуру данных (англ. data structure). Эта структура представляет собой набор данных одного типа, и едва ли хоть одна реальная Haskell-программа может обойтись без списков.

Структуры, содержащие данные одного типа, называют ещё гомогенными (в переводе с греческого: «одного рода»).

Вот список из трёх целых чисел:

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

Квадратные скобки и значения, разделённые запятыми. Вот так выглядит список из двух значений типа `Double`:

```
[1.3, 45.7899]
```

а вот и список из одного-единственного символа:

```
['H']
```

или вот из четырёх строк, отражающих имена протоколов транспортного уровня OSI-модели:

```
["TCP", "UDP", "DCCP", "SCTP"]
```

Если у вас есть опыт разработки на языке C, вы можете подумать, что список похож на массив. Однако, хотя сходства имеются, я намеренно избегаю слова «массив», потому что в Haskell существуют массивы (англ. array), это несколько иная структура данных.

Список — это тоже выражение, поэтому можно легко создать список списков произвольной вложенности. Вот так будет выглядеть список из ряда протоколов трёх уровней OSI-модели:

```
[ ["DHCP", "FTP", "HTTP"]
, ["TCP", "UDP", "DCCP", "SCTP"]
, ["ARP", "NDP", "OSPF"]
]
```

Это список списков строк. Форматирование в отношении квадратных скобок весьма вольное, при желании можно и так написать:

```
[["DHCP", "FTP", "HTTP"        ],
 ["TCP",  "UDP", "DCCP", "SCTP"],
 ["ARP",  "NDP", "OSPF"        ]]
```

Список может быть и пустым, то есть не содержать в себе никаких данных:

```
[]
```

Тип списка

Раз список представляет собой структуру, содержащую данные некоторого типа, каков же тип самого списка? Вот:

```
[Int]     -- Список целых чисел
[Char]    -- Список символов
[String]  -- Список строк
```

То есть тип списка так и указывается, в квадратных скобках. Упомянутый ранее список списков строк имеет такой тип:

```
[[String]]  -- Список списков строк
```

Модель очень проста:

```
[   [String]    ]

   |  Тип   |
   └ данных ┘

|     Тип       |
|    списка     |
└─ этих данных ─┘
```

Хранить данные разных типов в стандартном списке невозможно. Однако вскоре мы познакомимся с другой стандартной структурой данных, которая позволяет это.

Действия над списками

Если списки создаются — значит это кому-нибудь нужно. Со списком можно делать очень много всего. В стандартной Haskell-библиотеке существует отдельный модуль `Data.List`, включающий широкий набор функций, работающих со списком. Откроем модуль `Main` и импортируем в него модуль `Data.List`:

```
module Main where

-- Стандартный модуль для работы со списками.
import Data.List

main :: IO ()
main = putStrLn (head [”Vim”, ”Emacs”, ”Atom”])
```

Функция `head` возвращает голову списка, то есть его первый элемент. При запуске этой программы на выходе получим:

```
Vim
```

Модель такая:

```
[”Vim”  ,  ”Emacs”, ”Atom”]

 голова    └── хвост ──┘
```

Эдакая гусеница получается: первый элемент — голова, а всё остальное — хвост. Функция `tail` возвращает хвост:

```
main :: IO ()
main = print (tail [”Vim”, ”Emacs”, ”Atom”])
```

Вот результат:

```
[”Emacs”, ”Atom”]
```

Функция `tail` формирует другой список, представляющий собою всё от первоначального списка, кроме головы. Обратите внимание на новую функцию `print`. В данном случае мы не могли бы использовать нашу знакомую `putStrLn`, ведь она применяется к значению типа `String`, в то время как функция `tail` вернёт нам значение типа `[String]`. Мы ведь помним про строгость компилятора: что ожидаем, то

и получить должны. Функция `print` предназначена для «стрингификации» значения: она берёт значение некоторого типа и выводит это значение на консоль уже в виде строки.

Внимательный читатель спросит, каким же образом функция `print` узнаёт, как именно отобразить конкретное значение в виде строки? О, это интереснейшая тема, но она относится к Третьему Киту Haskell, до знакомства с которым нам ещё далеко.

Можно получить длину списка:

```
handleTableRow :: String -> String
handleTableRow row
  | length row == 2 = composeTwoOptionsFrom row
  | length row == 3 = composeThreeOptionsFrom row
  | otherwise       = invalidRow row
```

Это чуток видоизменённый кусочек одной моей программы, функция `handleTableRow` обрабатывает строку таблицы. Стандартная функция `length` даём нам длину списка (число элементов в нём). В данном случае мы узнаём число элементов в строке таблицы `row`, и в зависимости от этой длины применяем к этой строке функцию `composeTwoOptionsFrom` или `composeThreeOptionsFrom`.

Но постойте, а где же тут список? Функция `handleTableRow` применяется к строке и вычисляет строку. А всё дело в том, что строка есть ни что иное, как список символов. То есть тип `String` эквивалентен типу `[Char]`. Скажу более: `String` — это даже не самостоятельный тип, это всего лишь псевдоним для типа `[Char]`, и вот как он задан:

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

Ключевое слово `type` вводит синоним для уже существующего типа (англ. type synonym). Иногда его называют «псевдонимом типа». Читается это так:

```
type  String  =      [Char]

тип   этот    равен  тому
```

Таким образом, объявление функции `handleTableRow` можно было бы переписать так:

```
handleTableRow :: [Char] -> [Char]
```

При работе со списками мы можем использовать уже знакомые промежуточные выражения, например:

```
handleTableRow :: String -> String
handleTableRow row
  | size == 2 = composeTwoOptionsFrom row
  | size == 3 = composeThreeOptionsFrom row
  | otherwise = invalidRow row
  where size = length row
```

А можно и так:

```
handleTableRow :: String -> String
handleTableRow row
  | twoOptions   = composeTwoOptionsFrom row
  | threeOptions = composeThreeOptionsFrom row
  | otherwise    = invalidRow row
  where
    size         = length row  -- Узнаём длину
    twoOptions   = size == 2   -- ... сравниваем
    threeOptions = size == 3   -- ... и ещё раз
```

Здесь выражения `twoOptions` и `threeOptions` имеют уже знакомый нам стандартный тип `Bool`, ведь они равны результату сравнения значения `size` с числом.

Неизменность списка

Как вы уже знаете, все данные в Haskell неизменны, как Египетские пирамиды. Списки — не исключение: мы не можем изменить существующий список, мы можем лишь создать на его основе новый список. Например:

```
addTo :: String -> [String] -> [String]
addTo newHost hosts = newHost : hosts

main :: IO ()
main = print ("124.67.54.90" `addTo` hosts)
  where hosts = ["45.67.78.89", "123.45.65.54"]
```

Результат этой программы таков:

```
["124.67.54.90","45.67.78.89","123.45.65.54"]
```

Рассмотрим определение функции `addTo`:

```
addTo newHost hosts = newHost : hosts
```

Стандартный оператор : добавляет значение, являющееся левым операндом, в начало списка, являющегося правым операндом. Читается это так:

```
newHost   :         hosts

          этот
          оператор

берёт
это
значение
                    и добавляет
                    его в начало
                    этого списка
```

Естественно, тип значения слева обязан совпадать с типом значений, содержащихся в списке справа.

С концептуальной точки зрения функция `addTo` добавила новый IP-адрес в начало списка `hosts`. В действительности же никакого добавления не произошло, ибо списки неизменны. Оператор `:` взял значение `newHost` и список `hosts` и создал на их основе новый список, содержащий в себе уже три IP-адреса вместо двух.

Перечисление

Допустим, понадобился нам список целых чисел от одного до десяти. Пишем:

```
main :: IO ()
main = print tenNumbers
  where tenNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Неплохо, но избыточно, ведь чисел могло быть и сто, и тысяча. Есть лучший путь:

```
main :: IO ()
main = print tenNumbers
  where tenNumbers = [1..10]
```

Красиво, не правда ли? Выражение в квадратных скобках называется перечислением (англ. enumeration или сокращённо enum). Иногда её именуют также арифметической последовательностью. Идея предельно проста: зачем указывать содержимое списка целиком в той ситуации, когда можно указать лишь диапазон значений? Это мы и сделали:

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

Значение слева от `..` — это начало диапазона, а значение справа — его конец. Компилятор сам догадается, что шаг между числами в данной последовательности равен 1. Вот ещё пример:

```
[3..17] = [3,4,5,6,7,8,9,10,11,12,13,14,15,16,17]

 _         _

    ==                                        ==
```

Мы можем задать шаг и явно:

```
[2,4..10] = [2,4,6,8,10]
```

Получили только чётные значения. Схема проста:

```
[2,      4      .. 10]

 первый            конец
         второй

 |  разница   |
 └─ даёт шаг ─┘
```

Вот ещё пример:

```
[3,9..28] = [3,9,15,21,27]
```

Можно задать и нисходящий диапазон:

```
[9,8..1] = [9,8,7,6,5,4,3,2,1]
```

Или так:

```
[-9, -8.. -1] = [-9,-8,-7,-6,-5,-4,-3,-2,-1]
```

Да, отрицательные числа тоже работают. Можно взять также и числа с плавающей точкой:

```
[1.02,1.04..1.16] = [1.02,1.04,1.06,1.08,1.1,1.12,1.14,1.16]
```

В общем, идея ясна. Но что это мы всё с числами да с числами! Возьмём символы:

```
['a'..'z'] = "abcdefghijklmnopqrstuvwxyz"
```

Диапазон от `'a'` до `'z'` — получили английский алфавит в виде `[Char]` или, как мы уже знаем, просто `String`. При большом желании явно задать шаг можно и здесь:

```
['a','c'..'z'] = "acegikmoqsuwy"
```

Вот такая красота.

Теперь, после знакомства со списком, мы будем использовать их постоянно.

Для любопытных

В разделе про диапазоны для списка мы оперировали значениями типа `Int`, `Double` и `Char`. Возникает вопрос: а можно ли использовать значения каких-нибудь других типов? Отвечаю: можно, но с оговоркой. Попробуем проделать это со строкой:

```
main :: IO ()
main = print ["a","aa"..."aaaaaa"]  -- Ну-ну...
```

При попытке скомпилировать такой код увидим ошибку:

```
No instance for (Enum [Char])
  arising from the arithmetic sequence `"a", "aa" .. "aaaaaa"'
```

И удивляться тут нечему: шаг между строками абсурден, и компилятор в замешательстве. Не все типы подходят для перечислений в силу своей природы, однако в будущем, когда мы научимся создавать наши собственные типы, мы узнаем, что их вполне можно использовать в диапазонах. Наберитесь терпения.

Приоткрою секрет: этот странный пример с шагом между строками теоретически можно-таки заставить работать, но о том, как это сделать, мы узнаем во время знакомства с Третьим Китом Haskell.

Глава 12

Кортеж

В этой главе мы познакомимся с кортежем и ещё ближе подружимся с паттерн матчингом.

Кортеж (англ. tuple) — ещё одна стандартная структура данных, но, в отличие от списка, она может содержать данные как одного типа, так и разных.

Структуры, способные содержать данные разных типов, называют гетерогенными (в переводе с греческого: «разного рода»).

Вот как выглядит кортеж:

```
("Haskell", 2010)
```

Круглые скобки и значения, разделённые запятыми. Этот кортеж содержит значение типа `String` и ещё одно, типа `Int`. Вот ещё пример:

```
("Haskell", "2010", "Standard")
```

То есть ничто не мешает нам хранить в кортеже данные одного типа.

Тип кортежа

Тип списка строк, как вы помните, `[String]`. И не важно, сколько строк мы запихнули в список, одну или миллион — его тип останется неизменным. С кортежем же дело обстоит абсолютно иначе.

Тип кортежа зависит от количества его элементов. Вот тип кортежа, содержащего две строки:

```
(String, String)
```

Вот ещё пример:

```
(Double, Double, Int)
```

И ещё:

```
(Bool, Double, Int, String)
```

Тип кортежа явно отражает его содержимое. Поэтому если функция применяется к кортежу из двух строк, применить её к кортежу из трёх никак не получится, ведь типы этих кортежей различаются:

```
-- Разные типы
(String, String)
(String, String, String)
```

Действия над кортежами

Со списками можно делать много всего, а вот с кортежами — не очень. Самые частые действия — собственно формирование кортежа и извлечение хранящихся в нём данных. Например:

```
makeAlias :: String -> String -> (String, String)
makeAlias host alias = (host, alias)
```

Пожалуй, ничего проще придумать нельзя: на входе два аргумента, на выходе — двухэлементный кортеж с этими аргументами. Двухэлементный кортеж называют ещё парой (англ. pair). И хотя кортеж может содержать сколько угодно элементов, на практике именно пары встречаются чаще всего.

Обратите внимание, насколько легко создаётся кортеж. Причина тому — уже знакомый нам паттерн матчинг:

```
makeAlias host alias = (host, alias)

           ____          ____

                =====          =====
```

Мы просто указываем соответствие между левой и правой сторонами определения: «Пусть первый элемент пары будет равен аргументу `host`, а второй — аргументу `alias`». Ничего удобнее и проще и придумать нельзя. И если бы мы хотели получить кортеж из трёх элементов, это выглядело бы так:

```
makeAlias :: String -> String -> (String, String, String)
makeAlias host alias = (host, ”https://” ++ host, alias)

          ____          ____                ____

               =====                              =====
```

Оператор `++` — это оператор конкатенации, склеивающий две строки в одну. Строго говоря, он склеивает два списка, но мы-то с вами уже знаем, что `String` есть ни что иное, как `[Char]`. Таким образом, `”https://” ++ ”www.google.com”` даёт нам `”https://www.google.com”`.

Извлечение элементов из кортежа также производится через паттерн матчинг:

```
main :: IO ()
main =
  let (host, alias) = makeAlias ”173.194.71.106”
                                ”www.google.com”
  in print (host ++ ”, ” ++ alias)
```

Функция `makeAlias` даёт нам пару из хоста и имени. Но что это за странная запись возле уже знакомого нам слова `let`? Это промежуточное выражение, но выражение хитрое, образованное через паттерн матчинг. Чтобы было понятнее, сначала перепишем функцию без него:

```
main :: IO ()
main =
  let pair  = makeAlias ”173.194.71.106”
                        ”www.google.com”
      host  = fst pair  -- Берём первое...
      alias = snd pair  -- Берём второе...
  in print (host ++ ”, ” ++ alias)
```

При запуске этой программы получим:

```
”173.194.71.106, www.google.com”
```

Стандартные функции `fst` и `snd` возвращают первый и второй элемент кортежа соответственно. Выражение `pair` соответствует паре, выражение `host` — значению хоста, а `alias` — значению имени. Но не кажется ли вам такой способ избыточным? Мы в Haskell любим изящные решения, поэтому предпочитаем паттерн матчинг. Вот как получается вышеприведённый способ:

```
let (host, alias) = makeAlias "173.194.71.106" "www.google.com"

let (host, alias) = ("173.194.71.106", "www.google.com")

                     данное значение
     это
     хост
                                       а вот это значение
          это
          имя
```

Вот такая простая магия. Функция `makeAlias` даём там пару, и мы достоверно знаем это! А если знаем, нам не нужно вводить промежуточные выражения вроде `pair`. Мы сразу говорим:

```
let (host, alias) = makeAlias "173.194.71.106" "www.google.com"

                    мы точно знаем, что выражение,
                    вычисленное этой функцией
     это вот
     такая пара
```

Это «зеркальная» модель: через паттерн матчинг формируем:

```
-- Формируем правую сторону
-- на основе левой...
host alias = (host, alias)

>>>>          >>>>

     >>>>>          >>>>>
```

и через него же извлекаем:

```
-- Формируем левую сторону
-- на основе правой...
(host, alias) = ("173.194.71.106", "www.google.com")

 <<<<            <<<<<<<<<<<<<<<<

       <<<<<                      <<<<<<<<<<<<<<<<
```

Вот ещё один пример работы с кортежем через паттерн матчинг:

```
chessMove :: String
          -> (String, String)
          -> (String, (String, String))
chessMove color (from, to) = (color, (from, to))

main :: IO ()
main = print (color ++ ": " ++ from ++ "-" ++ to)
  where
    (color, (from, to)) = chessMove "white" ("e2", "e4")
```

И на выходе получаем:

```
"white: e2-e4"
```

Обратите внимание, объявление функции отформатировано чуток иначе: типы выстроены друг под другом через выравнивание стрелок под двоеточием. Вы часто встретите такой стиль в Haskell-проектах.

Функция `chessMove` даёт нам кортеж с кортежем, а раз мы точно знаем вид этого кортежа, сразу указываем `where`-выражение в виде образца:

```
(color, (from, to)) = chessMove "white" ("e2", "e4")

 _____                          _______

        ====                           ====

              ..                             ....
```

Не всё

Мы можем вытаскивать по образцу лишь часть нужной нам информации. Помните универсальный образец _? Взгляните:

```
-- Поясняющие псевдонимы
type UUID     = String
type FullName = String
type Email    = String
type Age      = Int
type Patient = (UUID, FullName, Email, Age)

patientEmail :: Patient -> Email
patientEmail (_, _, email, _) = email

main :: IO ()
main =
  putStrLn (patientEmail ( "63ab89d"
                         , "John Smith"
                         , "johnsm@gmail.com"
                         , 59
                         ))
```

Функция `patientEmail` даёт нам почту пациента. Тип `Patient` — это псевдоним для кортежа из четырёх элементов: уникальный идентификатор, полное имя, адрес почты и возраст. Дополнительные псевдонимы делают наш код яснее: одно дело видеть безликую `String` и совсем другое — `Email`.

Рассмотрим внутренность функции `patientEmail`:

```
patientEmail (_, _, email, _) = email
```

Функция говорит нам: «Да, я знаю, что мой аргумент — это четырёхэлементный кортеж, но меня в нём интересует исключительно третий по счёту элемент, соответствующий адресу почты, его я и верну». Универсальный образец _ делает наш код лаконичнее и понятнее, ведь он помогает нам игнорировать то, что нам неинтересно. Строго говоря, мы не обязаны использовать _, но с ним будет лучше.

А если ошиблись?

При использовании паттерн матчинга в отношении пары следует быть внимательным. Представим себе, что вышеупомянутый тип `Patient` был расширен:

```
type UUID      = String
type FullName  = String
type Email     = String
type Age       = Int
type DiseaseId = Int  -- Новый элемент.
type Patient = ( UUID
               , FullName
               , Email
               , Age
               , DiseaseId
               )
```

Был добавлен идентификатор заболевания. И всё бы хорошо, но внести изменения в функцию `patientEmail` мы забыли:

```
patientEmail :: Patient -> Email
patientEmail (_, _, email, _) = email

             ^  ^  ^      ^  -- А пятый где?
```

К счастью, в этом случае компилятор строго обратит наше внимание на ошибку:

```
Couldn't match type '(t0, t1, String, t2)'
               with '(UUID, FullName, Email, Age, DiseaseId)'
Expected type: Patient
  Actual type: (t0, t1, String, t2)
In the pattern: (_, _, email, _)
```

Оно и понятно: функция `patientEmail` использует образец, который уже некорректен. Вот почему при использовании паттерн матчинга следует быть внимательным.

На этом наше знакомство с кортежем считаю завершённым, в последующих главах мы будем использовать их периодически.

Для любопытных

Для работы с элементами многоэлементных кортежей можно использовать готовые библиотеки, во избежании длинных паттерн матчинговых цепочек. Например, пакет tuple:

```
Data.Tuple.Select

main :: IO ()
main = print (sel4 (123, 7, "hydra", "DC:4", 44, "12.04"))
```

Функция `sel4` из модуля `Data.Tuple.Select` извлекает четвёртый по счёту элемент кортежа, в данном случае строку "`DC:4`". Там есть функции вплоть до `sel32`, авторы вполне разумно сочли, что никто, находясь в здравом уме и твёрдой памяти, не станет оперировать кортежами, состоящими из более чем 32 элементов.

Кроме того, мы и обновлять элементы кортежа можем:

```
import Data.Tuple.Update

main :: IO ()
main = print (upd2 2 ("si", 45))
```

Естественно, по причине неизменности кортежа, никакого обновления тут не происходит, но выглядит симпатично. При запуске получаем результат:

```
("si",2)
```

Второй элемент кортежа изменился с `45` на `2`.

Глава 13

Лямбда-функция

Пришло время познакомиться с важной концепцией — лямбда-функцией. Именно с неё всё и началось. Приготовьтесь: в этой главе нас ждут новые открытия.

Истоки

В далёких 1930-х молодой американский математик Алонзо Чёрч задался вопросом о том, что значит «вычислить» что-либо. Плодом его размышлений явилась система для формализации понятия «вычисление», и назвал он эту систему «лямбда-исчислением» (англ. lambda calculus, по имени греческой буквы λ). В основе этой системы лежит лямбда-функция, которую в некотором смысле можно считать «матерью функционального программирования» в целом и Haskell в частности. Далее буду называть её ЛФ.

В отношении ЛФ можно смело сказать: «Всё гениальное просто». Идея ЛФ столь полезна именно потому, что она предельно проста. ЛФ — это анонимная функция. Вот как она выглядит в Haskell:

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

Обратный слэш в начале — признак ЛФ. Сравните с математической формой записи:

```
λx . x * x
```

Похоже, не правда ли? Воспринимайте обратный слэш в определении ЛФ как спинку буквы λ.

ЛФ представляет собой простейший вид функции, эдакая функция, раздетая догола. У неё забрали не только объявление, но и имя, оставив лишь необходимый минимум в виде имён аргументов и внутреннего выражения. Алонзо Чёрч понял: чтобы применить функцию, вовсе необязательно её именовать. И если у обычной функции сначала идёт объявление/определение, а затем (где-то) применение с ис-

пользованием имени, то у ЛФ всё куда проще: мы её определяем и тут же применяем, на месте. Вот так:

```
(\x -> x * x) 5
```

Помните функцию `square`? Вот это её лямбда-аналог:

```
(\x -> x * x)      5

лямбда-абстракция  аргумент
```

Лямбда-абстракция (англ. lambda abstraction) — это особое выражение, порождающее функция, которую мы сразу же применяем к аргументу `5`. ЛФ с одним аргументом, как и простую функцию, называют ещё «ЛФ от одного аргумента» или «ЛФ одного аргумента». Также можно сказать и о «лямбда-абстракции от одного аргумента».

Строение

Строение лямбда-абстракции предельно простое:

```
\        x          ->  x * x
признак  имя            выражение
ЛФ       аргумента
```

Соответственно, если ЛФ применяется к двум аргументам — пишем так:

```
\        x          y          ->  x * y
признак  имя 1      имя 2          выражение
ЛФ       аргумента  аргумента
```

И когда мы применяем такую функцию:

```
(\x y -> x * y) 10 4
```

то просто подставляем `10` на место `x`, а `4` — на место `y`, и получаем выражение `10 * 4`:

```
  (\x y -> x * y) 10 4
= 10 * 4
= 40
```

В общем, всё как с обычной функцией, даже проще.

Мы можем ввести промежуточное значение для лямбда-абстракции:

```haskell
main :: IO ()
main = print (mul 10 4)
  where mul = \x y -> x * y
```

Теперь мы можем применять `mul` так же, как если бы это была сама лямбда-абстракция:

```haskell
  mul 10 4
= (\x y -> x * y) 10 4
= 10 * 4
```

И здесь мы приблизились к одному важному открытию.

Тип функции

Мы знаем, что у всех данных в Haskell-программе обязательно есть какой-то тип, внимательно проверяемый на этапе компиляции. Вопрос: какой тип у выражения `mul` из предыдущего примера?

```haskell
  where mul = \x y -> x * y  -- Какой тип?
```

Ответ прост: тип `mul` такой же, как и у этой лямбда-абстракции. Из этого мы делаем важный вывод: ЛФ имеет тип, как и обычные данные. Но поскольку ЛФ является частным случаем функции — значит и у обыкновенной функции тоже есть тип!

В нефункциональных языках между функциями и данными проведена чёткая граница: вот это функции, а вон то — данные. Однако в Haskell между данными и функциями разницы нет, ведь и то и другое покоится на одной и той же Черепахе. Вот тип функции `mul`:

```haskell
mul :: a -> a -> a
```

Погодите, скажете вы, но ведь это же объявление функции! Совершенно верно: объявление функции — это и есть указание её типа. Помните, когда мы впервые познакомились с функцией, я уточнил, что её объявление разделено двойным двоеточием? Так вот это двойное двоеточие и представляет собой указание типа:

```haskell
mul  ::     a -> a -> a

вот  имеет  |   вот   |
это  тип    └─ такой ─┘
```

Точно так же мы можем указать тип любых других данных:

```
let coeff = 12 :: Double
```

Хотя мы знаем, что в Haskell типы выводятся автоматически, иногда мы хотим взять эту заботу на себя. В данном случае мы явно говорим: «Пусть выражение `coeff` будет равно `12`, но тип его пусть будет `Double`, а не `Int`». Так же и с функцией: когда мы объявляем её — мы тем самым указываем её тип.

Но вы спросите, можем ли мы не указывать тип функции явно? Можем:

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

Это наша старая знакомая, функция `square`. Когда она будет применена к значению типа `Int`, тип аргумента будет выведен автоматически как `Int`.

И раз функция характеризуется типом так же, как и прочие данные, мы делаем ещё одно важное открытие: функциями можно оперировать как данными. Например, можно создать список функций:

```
main :: IO ()
main = putStrLn ((head functions) "Hi")
  where
    functions = [ \x -> x ++ " val1"
                , \x -> x ++ " val2"
                ]
```

Выражение `functions` — это список из двух функций. Два лямбда-выражения порождают эти две функции, но до момента применения они ничего не делают, они безжизненны и бесполезны. Но когда мы применяем функцию `head` к этому списку, мы получаем первый элемент списка, то есть первую функцию. И получив, тут же применяем эту функцию к строке `"Hi"`:

```
putStrLn ((head functions)  "Hi")

          |    первая    |  её
          |   функция    |  аргумент
          └─ из списка ──┘
```

Это равносильно коду:

```
putStrLn ((\x -> x ++ " val1") "Hi")
```

При запуске программы мы получим:

```
Hi val1
```

Кстати, а каков тип списка `functions`? Его тип таков: `[String -> String]`. То есть список функций с одним аргументом типа `String`, возвращающих значение типа

`String`.

Локальные функции

Раз уж между ЛФ и простыми функциями фактически нет различий, а функции есть частный случай данных, мы можем создавать функции локально для других функций:

```
-- Здесь определены функции
-- isInfixOf и isSuffixOf.
import Data.List

validComEmail :: String -> Bool
validComEmail email =
    containsAtSign email && endsWithCom email
  where
    containsAtSign e = "@" `isInfixOf` e
    endsWithCom e = ".com" `isSuffixOf` e

main :: IO ()
main = putStrLn (if validComEmail my
                   then "It's ok!"
                   else "Non-com email!")
  where
    my = "haskeller@gmail.com"
```

Несколько наивная функция `validComEmail` проверяет `.com`-адрес. Её выражение образовано оператором `&&` и двумя выражениями типа `Bool`. Вот как образованы эти выражения:

```
containsAtSign e = "@" `isInfixOf` e
endsWithCom e = ".com" `isSuffixOf` e
```

Это — две функции, которые мы определили прямо в `where`-секции, поэтому они существуют только для основного выражения функции `validComEmail`. С простыми функциями так поступают очень часто: где она нужна, там её и определяют. Мы могли бы написать и более явно:

```
validComEmail :: String -> Bool
validComEmail email =
    containsAtSign email && endsWithCom email
  where
    -- Объявляем локальную функцию явно.
    containsAtSign :: String -> Bool
    containsAtSign e = "@" `isInfixOf` e

    -- И эту тоже.
    endsWithCom :: String -> Bool
    endsWithCom e = ".com" `isSuffixOf` e
```

Впрочем, указывать тип столь простых функций, как правило, необязательно.

Вот как этот код выглядит с лямбда-абстракциями:

```
validComEmail :: String -> Bool
validComEmail email =
    containsAtSign email && endsWithCom email
  where
    containsAtSign = \e -> "@" `isInfixOf` e
    endsWithCom = \e -> ".com" `isSuffixOf` e
```

Теперь выражения `containsAtSign` и `endsWithCom` приравнены к ЛФ от одного аргумента. В этом случае мы не указываем тип этих выражений. Впрочем, если очень хочется, можно и указать:

```
containsAtSign =
    (\e -> "@" `isInfixOf` e) :: String -> Bool

    лямбда-абстракция            тип этой абстракции
```

Лямбда-абстракция взята в скобки, чтобы указание типа относилось к функции в целом, а не только к аргументу `e`:

```
containsAtSign =
    \e -> "@" `isInfixOf` e :: String -> Bool

                               в этом случае это
                               тип аргумента e,
                               а вовсе не всей
                               функции!
```

Для типа функции тоже можно ввести псевдоним:

```
-- Псевдоним для типа функции.
type Func = String -> Bool

validComEmail :: String -> Bool
validComEmail email =
    containsAtSign email && endsWithCom email
  where
    containsAtSign = (\e -> "@" `isInfixOf` e) :: Func
    endsWithCom = (\e -> ".com" `isSuffixOf` e) :: Func
```

Впрочем, на практике указание типа для лямбда-абстракций встречается исключительно редко, ибо незачем.

Отныне, познакомившись с ЛФ, мы будем использовать их периодически.

И напоследок, вопрос. Помните тип функции `mul`?

```
mul :: a -> a -> a
```

Что это за буква `a`? Во-первых, мы не встречали такой тип ранее, а во-вторых, разве имя типа в Haskell не обязано начинаться с большой буквы? Обязано. А всё дело в том, что буква `a` в данном случае — это не совсем имя типа. А вот что это такое, мы узнаем в одной из ближайших глав.

Для любопытных

А почему, собственно, лямбда? Почему Чёрч выбрал именно эту греческую букву? По одной из версий, произошло это чисто случайно.

Шли 30-е годы прошлого века, компьютеров не было, и все научные работы набирались на печатных машинках. В первоначальном варианте, дабы выделять имя аргумента ЛФ, Чёрч ставил над именем аргументом символ, похожий на `^`. Но когда он сдавал работу наборщику, то вспомнил, что печатная машинка не сможет воспроизвести такой символ над буквой. Тогда он вынес эту «крышу» перед именем аргумента, и получилось что-то наподобие:

```
^x . x * 10
```

А наборщик, увидев такой символ, использовал заглавную греческую букву `Λ`:

```
Λx . x * 10
```

Вот так и получилось, лямбда-исчисление.

Глава 14

Композиция функций

Эта глава рассказывает о том, как объединять функции в цепочки, а также о том, как избавиться от круглых скобок.

Скобкам — бой!

Да, я не люблю круглые скобки. Они делают код визуально избыточным, к тому же нужно следить за симметрией скобок открывающих и закрывающих. Вспомним пример из главы про кортежи:

```
main :: IO ()
main =
  putStrLn (patientEmail ( "63ab89d"
           ^             , "John Smith"
                         , "johnsm@gmail.com"
                         , 59
                         ))
                          ^
```

Со скобками кортежа мы ничего сделать не можем, ведь они являются синтаксической частью кортежа. А вот скобки вокруг применения функции `patientEmail` мне абсолютно не нравятся. К счастью, мы можем избавиться от них. Но прежде чем искоренять скобки, задумаемся вот о чём.

Если применение функции представляет собой выражение, не можем ли мы как-нибудь компоновать их друг с другом? Конечно можем, мы уже делали это много раз, вспомните:

```
main :: IO ()
main = putStrLn (checkLocalhost "173.194.22.100")
```

Здесь компонуются две функции, `putStrLn` и `checkLocalhost`, потому что тип выражения на выходе функции `checkLocalhost` совпадает с типом выражения на входе функции `putStrLn`. Схематично это можно изобразить так:

```
          ┌─────────────────┐           ┌─────────┐
String ->|checkLocalhost|-> String ->|putStrLn|-> ...
          └─────────────────┘           └─────────┘

IP-адрес                    сообщение             текст
                            об этом               в нашем
                            IP-адресе             терминале
```

Получается эдакий конвейер: на входе строка с IP-адресом, на выходе — сообщение в нашем терминале. Существует иной способ соединения двух функций воедино.

Композиция и применение

Взгляните:

```
main :: IO ()
main = putStrLn . checkLocalhost $ "173.194.22.100"
```

Необычно? Перед нами два новых стандартных оператора, избавляющие нас от лишних скобок и делающие наш код проще. Оператор `.` — это оператор композиции функций (англ. function composition), а оператор `$` — это оператор применения (англ. application operator). Эти операторы часто используют совместно друг с другом. И отныне мы будем использовать их чуть ли не в каждой главе.

Оператор композиции объединяет две функции воедино (или компонует их, англ. compose). Когда мы пишем:

```
putStrLn . checkLocalhost
```

происходит маленькая «магия»: две функции объединяются в новую функцию. Вспомним наш конвейер:

```
          ┌─────────────────┐           ┌─────────┐
String ->|checkLocalhost|-> String ->|putStrLn|-> ...
          └─────────────────┘           └─────────┘

A                           B                     C
```

Раз нам нужно попасть из точки `A` в точку `C`, нельзя ли сделать это сразу? Можно, и в этом заключается суть композиции: мы берём две функции и объединяем их в третью функцию. Раз `checkLocalhost` приводит нас из точки `A` в точку `B`, а функция `putStrLn` — из точки `B` в `C`, тогда композиция этих двух функций будет представлять собой функцию, приводящую нас сразу из точки `A` в точку `C`:

```
          ┌──────────────────────────┐
String ->|checkLocalhost + putStrLn|-> ...
          └──────────────────────────┘

A                                    C
```

В данном случае знак + не относится к конкретному оператору, я лишь показываю факт «объединения» двух функций в третью. Теперь-то нам понятно, почему в типе функции, в качестве разделителя, используется стрелка:

```
checkLocalhost :: String -> String
```

в нашем примере это:

```
checkLocalhost :: A -> B
```

Она показывает наше движение из точки `A` в точку `B`. Поэтому часто говорят о «функции из `A` в `B`». Так, о функции `checkLocalhost` можно сказать как о «функции из `String` в `String`».

А оператор применения работает ещё проще. Без него код был бы таким:

```
main :: IO ()
main =
  (putStrLn . checkLocalhost)  "173.194.22.100"

  объединённая функция         аргумент
```

Но мы ведь хотели избавиться от круглых скобок, а тут они опять. Вот для этого и нужен оператор применения. Его схема проста:

```
FUNCTION  $           ARGUMENT
вот эта   применяется вот этому
функция   к           аргументу
```

Для нашей объединённой функции это выглядит так:

```
main :: IO ()
main =
  putStrLn . checkLocalhost  $           "173.194.22.100"

  объединённая функция       применяется
                             к           этому аргументу
```

Теперь получился настоящий конвейер: справа в него «заезжает» строка и движется «сквозь» функции, а слева «выезжает» результат:

```
main = putStrLn . checkLocalhost $  "173.194.22.100"

     <-         <-               <- аргумент
```

Чтобы было легче читать композицию, вместо оператора . мысленно подставляем фразу «применяется после»:

```
putStrLn  .            checkLocalhost

эта       применяется  этой
функция   после        функции
```

То есть композиция правоассоциативна (англ. right-associative): сначала применяется функция справа, а затем — слева.

Ещё одно замечание про оператор применения функции. Он весьма гибок, и мы можем написать так:

```
main = putStrLn . checkLocalhost $ "173.194.22.100"

       объединённая функция        └─ её аргумент ─┘
```

а можем и так:

```
main = putStrLn $ checkLocalhost "173.194.22.100"

       обычная     └──────── её аргумент ────────┘
       функция
```

Эти две формы, как вы уже поняли, эквивалентны. Я показываю это для того, чтобы вновь и вновь продемонстрировать вам, сколь гибко можно работать с данными и функциями в Haskell.

Длинные цепочки

Красота композиции в том, что компоновать мы можем сколько угодно функций:

```
logWarn :: String -> String
logWarn rawMessage =
  warning . correctSpaces . asciiOnly $ rawMessage

main :: IO ()
main = putStrLn $
  logWarn "Province   'Gia Vi□n' isn't on the map! "
```

Функция `logWarn` готовит переданную ей строку для записи в журнал. Функция `asciiOnly` готовит строку к выводу в нелокализованном терминале (да, в 2016 году такие всё ещё имеются), функция `correctSpaces` убирает дублирующиеся пробелы, а функция `warning` делает строку предупреждением (например, добавляет строку "`WARNING: `" в начало сообщения). При запуске этой программы мы увидим:

```
WARNING: Province 'Gia Vi?n' isn't on the map!
```

Здесь мы объединили в «функциональный конвейер» уже три функции, безо всяких скобок. Вот как это получилось:

```
warning . correctSpaces . asciiOnly $ rawMessage

                        ^
          └── первая композиция ──┘

        ^
└──────── вторая композиция ──────────┘
                                      аргумент
```

Первая композиция объединяет две простые функции, `correctSpaces` и `asciiOnly`. Вторая объединяет тоже две функции, простую `warning` и объединённую, являющуюся результатом первой композиции.

Более того, определение функции `logWarn` можно сделать ещё более простым:

```
logWarn :: String -> String
logWarn = warning . correctSpaces . asciiOnly
```

Погодите, но где же имя аргумента? А его больше нет, оно нам не нужно. Ведь мы знаем, что применение функции можно легко заменить внутренним выражением функции. А раз так, выражение `logWarn` может быть заменено на выражение `warning . correctSpaces . asciiOnly`. Сделаем же это:

```
   logWarn "Province   'Gia Vi□n' isn't on the map! "

 = (warning
    . correctSpaces
    . asciiOnly) "Province   'Gia Vi□n' isn't on the map! "

 =   warning
   . correctSpaces
   . asciiOnly $ "Province   'Gia Vi□n' isn't on the map! "
```

И всё работает! В мире Haskell принято именно так: если что-то может быть упрощено — мы это упрощаем.

Справедливости ради следует заметить, что не все Haskell-разработчики любят избавляться от круглых скобок, некоторые предпочитают использовать именно их. Что ж, это лишь вопрос стиля и привычек.

Как работает композиция

Если вдруг вы подумали, что оператор композиции уникален и встроен в Haskell — спешу вас разочаровать. Никакой магии, всё предельно просто. Этот стандартный оператор определён так же, как и любая другая функция. Вот его определение:

```
(.) f g = \x -> f (g x)
```

Опа! Да тут и вправду нет ничего особенного. Оператор композиции применяется к двум функциям. Стоп, скажете вы, как это? Применяется к функциям? Да, именно так. Ведь мы уже выяснили, что функциями можно оперировать как данными. А раз так, что нам мешает передать функцию в качестве аргумента другой функции? Что нам мешает вернуть функцию из другой функции? Ничего.

Оператор композиции получает на вход две функции, а потом всего лишь даёт нам ЛФ, внутри которой происходит обыкновенный последовательный вызов этих двух функций через скобки. И никакой магии:

```
(.)    f        g        =  \x -> f (g x)

берём  эту      и эту       и возвращаем
       функцию  функцию     ЛФ, внутри
                            которой
                            вызываем их
```

Подставим наши функции:

```
(.) putStrLn checkLocalhost = \x -> putStrLn (checkLocalhost x)
```

Вот так и происходит «объединение» двух функций: мы просто возвращаем ЛФ от одного аргумента, внутри которой правоассоциативно вызываем обе функции. А аргументом в данном случае является та самая строка с IP-адресом:

```
(\x -> putStrLn (checkLocalhost x)) ”173.194.22.100” =

putStrLn (checkLocalhost ”173.194.22.100”))
```

Но если я вас ещё не убедил, давайте определим собственный оператор композиции функций! Помните, я говорил вам, что ASCII-символы можно гибко объединять в операторы? Давайте возьмём плюс со стрелками, он чем-то похож на объединение. Пишем:

```
-- Наш собственный оператор композиции.
(<+>) f g = \x -> f (g x)

...

main :: IO ()
main = putStrLn <+> checkLocalhost $ "173.194.22.100"
```

Выглядит необычно, но работать будет так, как и ожидается: мы определили собственный оператор `<+>` с тем же функционалом, что и стандартный оператор композиции. Поэтому можно написать ещё проще:

```
(<+>) f g = f . g
```

Мы говорим: «Пусть оператор `<+>` будет эквивалентен стандартному оператору композиции функций.». И так оно и будет. А можно — не поверите — ещё проще:

```
f <+> g = f . g
```

И это будет работать! Раз оператор предназначен для инфиксного применения, то мы, определяя его, можно сразу указать его в инфиксной форме:

```
f <+> g    =      f . g

           пусть

такое
выражение
           будет
           равно
                  такому
                  выражению
```

Теперь мы видим, что в композиции функций нет ничего сверхъестественного. Эту мысль я подчёркиваю на протяжении всей книги: в Haskell нет никакой магии, он логичен и последователен.

Глава 15

ФВП

ФВП, или Функции Высшего Порядка (англ. HOF, Higher Order Functions) — важная концепция в Haskell, с которой, однако, мы уже знакомы. Как мы узнали из предыдущих глав, функциями можно оперировать как значениями. Так вот функции, оперирующие другими функциями как аргументами и/или как результирующим выражением, носят название функций высшего порядка.

Так, оператор композиции функций является ФВП, потому что он, во-первых, принимает функции в качестве аргументов, а во-вторых, возвращает другую функцию (в виде ЛФ) как результат своего применения. Использование функций в качестве аргументов — чрезвычайно распространённая практика в Haskell.

Отображение

Рассмотрим функцию `map`. Эта стандартная функция используется для отображения (англ. mapping) функции на элементы списка. Пусть вас не смущает такой термин: отображение функции на элемент фактически означает её применение к этому элементу.

Вот объявление функции `map`:

```
map :: (a -> b) -> [a] -> [b]
```

Вот опять эти маленькие буквы! Помните, я обещал рассказать о них? Рассказываю: малой буквой принято именовать полиморфный (англ. polymorphic) тип. Полиморфизм — это многообразность, многоформенность. В данном случае речь идёт не об указании конкретного типа, а о «типовой заглушке». Мы говорим: «Функция `map` применяется к функции из какого-то типа `a` в какой-то тип `b` и к списку типа `[a]`, а результат её работы — это другой список типа `[b]`». Типовой заглушкой я назвал их потому, что на их место встают конкретные типы, что делает функцию `map` очень гибкой. Например:

```
import Data.Char

toUpperCase :: String -> String
toUpperCase str = map toUpper str

main :: IO ()
main = putStrLn . toUpperCase $ "haskell.org"
```

Результатом работы этой программы будет строка:

```
HASKELL.ORG
```

Функция `map` применяется к двум аргументам: к функции `toUpper` и к строке `str`. Функция `toUpper` из стандартного модуля `Data.Char` переводит символ типа `Char` в верхний регистр:

```
toUpper 'a' = 'A'
```

Вот её объявление:

```
toUpper :: Char -> Char
```

Функция из `Char` в `Char` выступает первым аргументом функции `map`, подставим сигнатуру:

```
map :: (a    -> b)    -> [a] -> [b]
       (Char -> Char)
```

Ага, уже теплее! Мы сделали два новых открытия: во-первых, заглушки `a` и `b` могут быть заняты одним и тем же конкретным типом, а во-вторых, сигнатура позволяет нам тут же понять остальные типы. Подставим их:

```
map :: (a    -> b)    -> [a]    -> [b]
       (Char -> Char)    [Char]    [Char]

        ____              ____

                ____                ____
```

А теперь вспомним о природе типа `String`:

```
map :: (a    -> b)    -> [a]    -> [b]
       (Char -> Char)    String    String
```

Всё встало на свои места. Функция `map` в данном случае берёт функцию `toUpper` и бежит по списку, последовательно применяя эту функцию к его элементам:

```
map toUpper ['h','a','s','k','e','l','l','.','o','r','g']
```

Так, на первом шаге функция `toUpper` будет применена к элементу `'h'`, на втором — к элементу `'a'`, и так далее до последнего элемента `'g'`. Когда функция `map` бежит по этому списку, результат применения функции `toUpper` к его элементам служит элементами для второго списка, который и будет в конечном итоге возвращён. Так, результатом первого шага будет элемент `'H'`, результатом второго — элемент `'A'`, а результатом последнего — элемент `'G'`. Схема такова:

```
map toUpper [ 'h'  >>  [ 'H'
            , 'a'  >>  , 'A'
            , 's'  >>  , 'S'
            , 'k'  >>  , 'K'
            , 'e'  >>  , 'E'
            , 'l'  >>  , 'L'
            , 'l'  >>  , 'L'
            , '.'  >>  , '.'
            , 'o'  >>  , 'O'
            , 'r'  >>  , 'R'
            , 'g'  >>  , 'G'
            ]          ]
```

Вот и получается:

```
map toUpper "haskell.org" = "HASKELL.ORG"
```

Работа функции `map` выглядит как изменение списка, однако, в виду неизменности последнего, в действительности формируется новый список. Что самое интересное, функция `toUpper` пребывает в полном неведении о том, что ею в конечном итоге изменяют регистр целой строки, она знает лишь об отдельных символах этой строки. То есть функция, являющаяся аргументом функции `map`, ничего не знает о функции `map`, и это очень хорошо! Чем меньше функции знают друг о друге, тем проще и надёжнее использовать их друг с другом.

Рассмотрим другой пример, когда типовые заглушки `a` и `b` замещаются разными типами:

```
toStr :: [Double] -> [String]
toStr numbers = map show numbers

main :: IO ()
main = print . toStr $ [1.2, 1,4, 1.6]
```

Функция `toStr` работает уже со списками разных типов: на входе список чисел с плавающей точкой, на выходе список строк. При запуске этой программы мы увидим следующее:

```
["1.2","1.0","4.0","1.6"]
```

Уже знакомая нам стандартная функция `show` переводит свой единственный аргумент в строковый вид:

```
show 1.2 = ”1.2”
```

В данном случае, раз уж мы работаем с числами типа `Double`, тип функции `show` такой:

```
show :: Double -> String
```

Подставим в сигнатуру функции `map`:

```
map :: (a      -> b)      -> [a]      -> [b]
       (Double -> String)    [Double]    [String]

        ______                ______

                ======                ======
```

Именно так, как у нас и есть:

```
map show [1.2, 1,4, 1.6] = [”1.2”,”1.0”,”4.0”,”1.6”]
```

Функция `map` применяет функцию `show` к числам из первого списка, на выходе получаем второй список, уже со строками. И как и в случае с `toUpper`, функция `show` ничего не подозревает о том, что ею оперировали в качестве аргумента функции `map`.

Разумеется, в качестве аргумента функции `map` мы можем использовать и наши собственные функции:

```
ten :: [Double] -> [Double]
ten = map (\n -> n * 10)

main :: IO ()
main = print . ten $ [1.2, 1,4, 1.6]
```

Результат работы:

```
[12.0,10.0,40.0,16.0]
```

Мы передали функции `map` нашу собственную ЛФ, умножающую свой единственный аргумент на `10`. Обратите внимание, мы вновь использовали краткую форму определения функции `ten`, опустив имя её аргумента. Раскроем подробнее:

```
main = print .          ten        $ [1.2, 1,4, 1.6] =
                  _____/  \_____
                 /              \
                /                \
main = print . map (\n -> n * 10) $ [1.2, 1,4, 1.6]
```

Вы спросите, как же вышло, что оператор применения расположен между двумя аргументами функции `map`? Разве он не предназначен для применения функции к единственному аргументу? Совершенно верно. Пришло время открыть ещё один секрет Haskell.

Частичное применение

Функция `map` ожидает два аргумента, это отражено в её типе. Но что будет, если применить её не к двум аргументам, а лишь к одному? В этом случае произойдёт ещё одно «магическое» превращение, называющееся частичным применением (англ. partial application) функции. Частичным называют такое применение, когда аргументов меньше чем ожидается.

Вспомним сокращённое определение функции `ten`:

```
ten = map (\n -> n * 10)

          первый         а где же
          аргумент       второй??
          есть
```

Функция `map` получила лишь первый аргумент, а где же второй? Второй, как мы уже знаем, будет получен ею уже потом, после того, как мы подставим это выражение на место функции `ten`. Но что же происходит с функцией `map` до этого? А до этого с ней происходит частичное применение. Понятно, что она ещё не может выполнить свою работу, поэтому, будучи применённой лишь к одному аргументу, она возвращает ЛФ! Сопоставим с типом функции `map`, и всё встанет на свои места:

```
map :: (a -> b)        -> [a]           -> [b]

map    (\n -> n * 10)

       только первый
       аргумент

|      частично     |
|    применённая    |
└──────── map ──────┘
                         аргумент          ответ
                         для частично
                         применённой
                         функции map

                         [1.2, 1,4, 1.6]
```

Тип ЛФ, возвращённой после применения `map` к первому аргументу — `[a] -> [b]`. Это «типовой хвост», оставшийся от полного типа функции `map`:

```
map :: (a -> b) -> [a]  ->  [b]

       голова      └── хвост ──┘
```

Поскольку голова в виде первого аргумента типа `(a -> b)` уже дана, осталось получить второй аргумент. Поэтому ЛФ, порождённая частичным применением, ожидает единственный аргумент, которым и будет тот самый второй, а именно список `[1.2, 1,4, 1.6]`.

Сопоставим тип функции `ten` с типом `map`, чтобы понять, где наш хвост:

```
ten ::             [Double] -> [Double]

map :: (a -> b) -> [a]      -> [b]

       голова      └──────── хвост ──────┘
```

Вот почему мы можем использовать краткую форму определения для функции `ten`: она уже является нашим хвостом!

Рассмотрим ещё один пример частичного применения, дабы закрепить наше понимание:

```
replace :: String -> String -> String -> String
```

Это объявление функции `replace`, принимающей три строки: первая содержит то, что ищем, вторая содержит то, на что заменяем, а в третьей лежит то, где ищем. Например:

```
replace "http"
        "https"
        "http://google.com" = "https://google.com"
```

Определение функции `replace` нас сейчас не интересует, рассмотрим пошаговое применение:

```
main :: IO ()
main = putStrLn result
  where
    first  = replace "http"
    second = first   "https"
    result = second  "http://google.com"
```

Тип выражения `first` — `String -> String -> String`, оно явилось результатом частичного применения функции `replace` к первому аргументу, строке `"http"`. Тип выражения `second` — `String -> String`, оно явилось результатом вторичного частичного применения функции `first` к уже второму аргументу, строке `"https"`. И наконец, применив функцию `second` к третьему аргументу, строке `"http://google.com"`, мы наконец-то получаем конечный результат, ассоциированный с выражением `result`.

Из этого мы делаем интересное открытие:

> Функция от нескольких аргументов может быть разложена на последовательность применений временных функций от одного аргумента каждая.

Поэтому мы и смогли подставить частично применённую `map` на место выражения `ten`. Используем круглые скобки, дабы яснее показать, что есть что:

```
main = print . (map (\n -> n * 10)) $ [1.2, 1,4, 1.6]

               |     частично     |
               └─ применённая map ┘

        |    композиция функции    |
        |     print и частично     |
        └──── применённой map ─────┘
                                    аргумент для
                                    композиции
```

Гибко, не правда ли? Теперь мы знакомы с частичным применением функции.

Композиция для отображения

Вернёмся к функции `map`. Если мы можем передать ей некую функцию для работы с элементами списка, значит мы можем передать ей и композицию двух или более

функций. Например:

```
import Data.Char

pretty :: [String] -> [String]
pretty = map (stars . big)
  where
    big = map toUpper
    stars = \s -> "* " ++ s ++ " *"

main :: IO ()
main = print . pretty $ ["haskell", "lisp", "coq"]
```

Мы хотим украсить имена трёх языков программирования. Для этого мы пробегаемся по списку композицией двух функций, `big` и `stars`. Функция `big` переводит сроки в верхний регистр, а функция `stars` украшает имя двумя звёздочками в начале и в конце. В результате имеем:

```
["* HASKELL *","* LISP *","* COQ *"]
```

Пройтись по списку композицией `stars . big` равносильно тому, как если бы мы прошлись сначала функцией `big`, а затем функцией `stars`. При этом, как мы уже знаем, обе эти функции ничего не знают ни о том, что их скомпоновали, ни о том, что эту композицию передали функции `map`.

Ну что ж, теперь мы знаем о функции `map`, и последующих главах мы увидим множество других ФВП. Отныне они будут нашими постоянными спутниками.

Глава 16

Hackage и библиотеки

Ранее я уже упоминал о библиотеках, пришло время познакомиться с ними поближе, ведь в последующих главах мы будем использовать их постоянно.

Библиотеки большие и маленькие

За годы существования Haskell разработчики со всего мира создали множество библиотек. Библиотеки избавляют нас от необходимости вновь и вновь писать то, что уже написано до нас. Для любого живого языка программирования написано множество библиотек. В мире Haskell их, конечно, не такая туча, как для той же Java, но порядочно: стабильных есть не менее двух тысяч, многие из которых очень качественные и уже многократно испытаны в серьёзных проектах.

С модулями — файлами, содержащими Haskell-код, — мы уже знакомы, они являются основным кирпичом любого Haskell-проекта. Библиотека, также являясь Haskell-проектом, тоже состоит из модулей (не важно, из одного или из сотен). Поэтому использование библиотеки сводится к использованию входящих в неё модулей. И мы уже неоднократно делали это в предыдущих главах.

Вспомним пример из главы про ФВП:

```
import Data.Char

toUpperCase :: String -> String
toUpperCase str = map toUpper str

main :: IO ()
main = putStrLn . toUpperCase $ ”haskell.org”
```

Функция `toUpper` определена в модуле `Data.Char`, который, в свою очередь, живёт в стандартной библиотеке. Библиотек есть множество, но стандартная лишь одна. Она содержит самые базовые, наиболее широко используемые инструменты. А прежде чем продолжить, зададимся важным вопросом: «Где живут все эти библиотеки?» Они живут в разных местах, но главное из них — Hackage.

Hackage

Hackage — это центральный репозиторий Haskell-библиотек, или, как принято у нас называть, пакетов (англ. package). Название репозитория происходит от слияния слов `Haskell` и `package`. Hackage существует с 2008 года и живёт здесь. Ранее упомянутая стандартная библиотека тоже живёт в Hackage и называется она `base`. Каждой библиотеке выделена своя страница.

Каждый из Hackage-пакетов живёт по адресу, сформированному по неизменной схеме: `http://hackage.haskell.org/package/ИМЯПАКЕТА`. Так, дом стандартной библиотеки — `http://hackage.haskell.org/package/base`. Hackage — открытый репозиторий: любой разработчик может добавить туда свои пакеты.

Стандартная библиотека включает в себя более сотни модулей, но есть среди них самый известный, носящий имя `Prelude`. Этот модуль по умолчанию всегда с нами: всё его содержимое автоматически импортируется во все модули нашего проекта. Например, уже известные нам `map` или операторы конкатенации списков живут в модуле `Prelude`, поэтому доступны нам всегда. Помимо них (и многих-многих десятков других функций) в `Prelude` располагаются функции для работы с вводом-выводом, такие как наши знакомые `putStrLn` и `print`.

Hackage весьма большой, поэтому искать пакеты можно двумя способами. Первый — на единой странице всех пакетов. Здесь перечислены все пакеты, а для нашего удобства они расположены по тематическим категориям.

Второй способ — через специальный поисковик, коих существует два:

1. Hoogle
2. Hayoo!

Эти поисковики скрупулёзно просматривают внутренности Hackage, и вы будете часто ими пользоваться. Лично я предпочитаю Hayoo!. Пользуемся оным как обычным поисковиком: например, знаем мы имя функции, а в каком пакете/модуле она живёт — забыли. Вбиваем в поиск — получаем результаты.

Чтобы воспользоваться пакетом в нашем проекте, нужно для начала включить его в наш проект. Для примера рассмотрим пакет `text`, предназначенный для работы с текстом. Он нам в любом случае понадобится, поэтому включим его в наш проект незамедлительно.

Открываем сборочный файл проекта `real.cabal`, находим секцию `executable real-exe` и в поле `build-depends` через запятую дописываем имя пакета:

```
build-depends:   base  -- Уже здесь!
               , real
               , text  -- А это новый пакет.
```

Файл с расширением `.cabal` — это обязательный сборочный файл Haskell-проекта. Он содержит главные инструкции, касающиеся сборки проекта. С синтаксисом сборочного файла мы будем постепенно знакомиться в следующих главах.

Как видите, пакет `base` уже тут. Включив пакет `text` в секцию `build-depends`, мы объявили тем самым, что наш проект отныне зависит от этого пакета. Теперь, нахо-

дясь в корне проекта, выполняем уже знакомую нам команду:

```
$ stack build
```

Помните, когда мы впервые настраивали проект, я упомянул, что утилита `stack` умеет ещё и библиотеки устанавливать? Она увидит новую зависимость нашего проекта и установит как сам пакет `text`, так и все те пакеты, от которых, в свою очередь, зависит пакет `text`. После сборки мы можем импортировать модули из этого пакета в наши модули. И теперь пришла пора узнать, как это можно делать.

Иерархия в имени

Когда мы пишем:

```
import Data.Char
```

в имени модуля отражена иерархия пакета. `Data.Char` означает, что внутри пакета `base` есть каталог `Data`, внутри которого живёт файл `Char.hs`, открыв который, мы увидим:

```
module Data.Char
...
```

Таким образом, точка в имени модуля отражает файловую иерархию внутри данного пакета. Можете воспринимать эту точку как слэш в Unix-пути. Есть пакеты со значительно более длинными именами, например:

```
module GHC.IO.Encoding.UTF8
```

Соответственно, имена наших собственных модулей тоже отражают место, в котором они живут. Так, один из модулей в моём рабочем проекте носит название `Common.Performers.Click`. Это означает, что живёт этот модуль здесь: `src/Common/Performers/Click.hs`.

Лицо

Вернёмся к нашему примеру:

```
import Data.Char
```

Импорт модуля `Data.Char` делает доступным для нас всё то, что включено в интерфейс этого модуля. Откроем наш собственный модуль `Lib`:

```
module Lib
    ( someFunc
    ) where

someFunc :: IO ()
someFunc = putStrLn "someFunc"
```

Имя функции `someFunc` упомянуто в интерфейсе модуля, а именно между круглыми скобками, следующими за именем модуля. Чуток переформатируем скобки:

```
module Lib (
    someFunc
) where
```

В настоящий момент только функция `someFunc` доступна всем импортёрам данного модуля. Если же мы определим в этом модуле другую функцию `anotherFunc`:

```
module Lib (
    someFunc
) where

someFunc :: IO ()
someFunc = putStrLn "someFunc"

anotherFunc :: String -> String
anotherFunc s = s ++ "!"
```

она останется невидимой для внешнего мира, потому что её имя не упомянуто в интерфейсе модуля. И если в модуле `Main` мы напишем так:

```
module Main

import Lib

main :: IO ()
main = putStrLn . anotherFunc $ "Hi"
```

компилятор справедливо ругнётся, мол, не знаю функцию `anotherFunc`. Если же мы добавим её в интерфейс модуля `Lib`:

```
module Lib (
    someFunc,
    anotherFunc
) where
```

тогда функция `anotherFunc` тоже станет видимой всему миру. Интерфейс позволяет нам показывать окружающим лишь то, что мы хотим им показать, оставляя служебные внутренности нашего модуля тайной за семью печатями.

Импортируем по-разному

В реальных проектах мы импортируем множество модулей из различных пакетов. Иногда это является причиной конфликтов, с которыми приходится иметь дело.

Вспомним функцию `putStrLn`: она существует не только в незримом модуле `Prelude`, но и в модуле `Data.Text.IO` из пакета `text`:

```
-- Здесь тоже есть функция по имени putStrLn.
import Data.Text.IO

main :: IO ()
main = putStrLn ...  -- И откуда эта функция?
```

При попытке скомпилировать такой код мы упрёмся в ошибку:

```
Ambiguous occurrence ‘putStrLn’
It could refer to either ‘Prelude.putStrLn’,
                         imported from ‘Prelude’ ...
                      or ‘Data.Text.IO.putStrLn’,
                         imported from ‘Data.Text.IO’ ...
```

Нам необходимо как-то указать, какую из функций `putStrLn` мы имеем в виду. Это можно сделать несколькими способами.

Можно указать принадлежность функции конкретному модулю. Из сообщения об ошибке уже видно, как это можно сделать:

```
-- Здесь тоже есть функция по имени putStrLn.
import Data.Text.IO

main :: IO ()
main = Data.Text.IO.putStrLn ...  -- Сомнений нет!
```

Теперь уже сомнений не осталось: используемая нами `putStrLn` принадлежит модулю `Data.Text.IO`, поэтому коллизий нет.

Впрочем, не кажется ли вам подобная форма слишком длинной? В упомянутом ранее стандартном модуле `GHC.IO.Encoding.UTF8` есть функция `mkUTF8`, и представьте себе:

```
import GHC.IO.Encoding.UTF8

main :: IO ()
main =
  let enc = GHC.IO.Encoding.UTF8.mkUTF8 ...
```

Слишком длинно, нужно укоротить. Импортируем модуля под коротким именем:

```
import    Data.Text.IO  as   TIO

включить  этот модуль   как  это

main :: IO ()
main = TIO.putStrLn ...
```

Вот, так значительно лучше. Короткое имя может состоять даже из одной буквы, но как и полное имя модуля, оно обязательно должно начинаться с большой буквы, поэтому:

```
import Data.Text.IO as tIO  -- Ошибка
import Data.Text.IO as i    -- Тоже ошибка
import Data.Text.IO as I    -- Порядок!
```

Иногда, для большего порядка, используют qualified-импорт:

```
import qualified Data.Text.IO as TIO
```

Ключевое слово `qualified` используется для «строгого» включения модуля: в этом случае мы обязаны указывать принадлежность к нему. Например:

```
import qualified Data.Text as T

main :: IO ()
main = T.justifyLeft ...
```

Даже несмотря на то, что функция `justifyLeft` есть только в модуле `Data.Text` и никаких коллизий с `Prelude` нет, мы обязаны указать, что эта функция именно из `Data.Text`. В больших модулях qualified-импорт бывает полезен: с одной стороны, гарантированно не будет никаких конфликтов, с другой, мы сразу видим, откуда родом та или иная функция.

Впрочем, некоторым Haskell-программистам любое указание принадлежности к модулю кажется избыточным. Поэтому они идут по другому пути: выборочное включение/выключение. Например:

```
import Data.Char
import Data.Text (pack)  -- Только её!

main :: IO ()
main = putStrLn $ map toUpper "haskell.org"
```

Мы подразумеваем стандартную функцию `map`, однако в модуле `Data.Text` тоже содержится функция по имени `map`. К счастью, никакой коллизии не будет, ведь мы импортировали не всё содержимое модуля `Data.Text`, а лишь одну его функцию `pack`:

```
import       Data.Text  (pack)

импортируем  отсюда     только
                        это
```

Если же мы хотим импортировать две или более функции, перечисляем их через запятую:

```
import Data.Text (pack, unpack)
```

Существует и прямо противоположный путь: вместо выборочного включения — выборочное выключение. Избежать коллизии между функциями `putStrLn` можно было бы и так:

```
import Data.Text.IO hiding (putStrLn)

main :: IO ()
main = putStrLn ...  -- Сомнений нет: из Prelude.
```

Слово `hiding` позволяет скрывать кое-что из импортируемого модуля:

```
import       Data.Text.IO  hiding (putStrLn)

импортируем  всё отсюда    кроме   этого
```

Можно и несколько функций скрыть:

```
import Data.Text.IO hiding ( readFile
                           , writeFile
                           , appendFile
                           )
```

При желании можно скрыть и из `Prelude`:

```
import Prelude hiding (putStrLn)
import Data.Text.IO

main :: IO ()
main = putStrLn ...  -- Она точно из Data.Text.IO.
```

Оформление

Общая рекомендация такова — оформляйте так, чтобы было легче читать. В реальном проекте в каждый из ваших модулей будет импортироваться довольно много всего. Вот кусочек из одного моего рабочего модуля:

```
import qualified Test.WebDriver.Commands    as WDC
import           Test.WebDriver.Exceptions
import qualified Data.Text                  as T
import           Data.Maybe                 (fromJust)
import           Control.Monad.IO.Class
import           Control.Monad.Catch
import           Control.Monad              (void)
```

Как полные, так и краткие имена модулей выровнены, такой код проще читать и изменять. Не все программисты согласятся с таким стилем, но попробуем убрать выравнивание:

```
import qualified Test.WebDriver.Commands as WDC
import Test.WebDriver.Exceptions
import qualified Data.Text as T
import Data.Maybe (fromJust)
import Control.Monad.IO.Class
import Control.Monad.Catch
import Control.Monad (void)
```

Теперь код выглядит скомканным, его труднее воспринимать. Впрочем, выбор за вами.

Глава 17

Рекурсия

> Чтобы понять рекурсию, нужно сначала понять рекурсию.

Эта старая шутка про рекурсию иногда пугает новичков, как в своё время напугала и меня. В действительности в рекурсии нет ничего страшного, и в этой главе мы познакомимся с этим важным механизмом.

Цикл

Удивительно, но в Haskell нет встроенных циклических конструкций, столь привычных для других языков. Ни тебе `for`, ни тебе `while`. Однако обойтись без циклов в нашем коде мы не сможем. Как же нам их организовывать?

К счастью, чаще всего нам это и не нужно. Вспомним нашу знакомую, функцию `map`:

```
map toUpper someList
```

Ну и чем же не цикл? На том же C это выглядело бы как-то так:

```
int length = ...
for(int i = 0; i < length; ++i) {
  char result = toUpper(someList[i]);
  ...
}
```

Функции наподобие `map` в подавляющем большинстве случаев избавляют нас от написания явных циклических конструкций, и это не может не радовать. Однако изредка нам всё-так придётся писать циклы явно. В Haskell, из-за отсутствия `for`-конструкции, сделать это можно только одним способом — через рекурсию (англ. recursion).

Идея рекурсии предельно проста:

> Если нам нужно повторить вычисление, производимое некой функцией, мы должны применить эту функцию внутри себя самой. И получится зацикливание.

Взглянем на определение функции `map`:

```
map _ []     = []
map f (x:xs) = f x : map f xs
```

А теперь разберём это интереснейшее определение по косточкам.

Правда о списке

Первым аргументом, как мы помним, выступает некая функция, а вторым — список, к элементам которого применяется эта функция. Но что это за странного вида конструкция в круглых скобках?

```
(x:xs)
```

Это — особый образец, используемый для работы со списками. И чтобы он стал понятен, я должен рассказать вам правду о формировании списка.

Как мы помним, формируется список предельно просто:

```
[1, 2, 3]  -- Список из трёх целых чисел.
```

Однако в действительности он формируется несколько иначе. Привычная нам конструкция в квадратных скобках есть ни что иное, как синтаксический сахар (англ. syntactic sugar). Синтаксическим сахаром называют некое упрощение кода, делающее его слаще, приятнее для нас. Если же мы уберём сахар (или, как ещё говорят, рассахарим код), то увидим вот что:

```
1 : 2 : 3 : []
```

Именно так список из трёх целых чисел формируется на самом деле. Стандартный оператор : нам уже знаком, мы встретились с ним в главе о списках:

```
newHost   :         hosts

          этот
          оператор

берёт
это
значение
                    и добавляет
                    его в начало
                    этого списка
```

То есть список строится путём добавления элемента в его «голову», начиная с пустого списка:

```
  1 : 2 : 3 : []

= 1 : 2 : [3]

= 1 : [2, 3]

= [1, 2, 3]
```

Начиная с правого края, мы сначала применяем оператор `:` к `3` и пустому списку, в результате чего получаем список с единственным элементом `[3]`. Затем, применяя второй оператор `:` к `2` и к только что полученному списку `[3]`, мы получаем новый список `[2, 3]`. И в конце, вновь применив оператор `:` к `1` и к списку `[2, 3]`, мы получаем итоговый список `[1, 2, 3]`. Вот почему столь удобно оперировать «головой» и «хвостом» списка. И именно поэтому был создан особый образец для паттерн-матчинговой работы со списком:

```
(head : tail)
```

В данном случае слова `head` и `tail` не относятся к стандартным функциям, я лишь показываю назначение элементов данного образца. Вот более живой пример:

```
main :: IO ()
main = print first
  where
    (first:others) = [”He”, ”Li”, ”Be”]

     _____            ____

           ======           ==========
```

Поскольку мы точно знаем, что справа у нас список, слева мы пишем образец для списка, в котором `first` ассоциирован с первым элементом, с «головой», а шаблон `others` — с оставшимися элементами, с «хвостом».

Но вы спросите, зачем нам это нужно? Если уж мы так хотим работать со списком через паттерн матчинг, можно ведь воспользоваться явным образцом:

```
main :: IO ()
main = print first
  where
    [first, second, third] = [”He”, ”Li”, ”Be”]

     _____                    ____

           ======                  ====

                  +++++                ++++
```

Всё верно, однако образец с круглыми скобками чрезвычайно удобен именно для рекурсивной работы со списком, и вот почему. Вспомним определение функции `map`:

```
map f (x:xs) = f x : map f xs

       _         _

         ==                ==
```

Подставим реальные значения на основе примера про перевод символов строки в верхний регистр:

```
map f       (x:xs) = f       x    : map f       xs

map toUpper ”neon” = toUpper ’n’ : map toUpper ”eon”

             _                _

              ===                              ===
```

Вот теперь-то мы видим, каким образом функция `map` пробегается по всему списку. Пройдёмся по итерациям, чтобы всё окончательно встало на свои места. У нас же цикл, верно? А где цикл — там итерации.

На первой из них оператор `:` применяется к выражениям `toUpper ’n’` и `map toUpper ”eon”`. Выражение слева вычисляется и даёт нам символ `’N’`:

```
toUpper ’n’ : map toUpper ”eon”

’N’         : map toUpper ”eon”
```

Выражение справа содержит применение той же функции `map`, то есть мы входим в цикл, во вторую его итерацию:

```
map toUpper "eon" = toUpper 'e' : map toUpper "on"
```

Выражение слева вычисляется и даёт нам `'E'`:

```
toUpper 'e' : map toUpper "on"

'E'         : map toUpper "on"
```

Вычисляем выражение справа — и входим в следующую итерацию:

```
map toUpper "on" = toUpper 'o' : map toUpper "n"
```

Выражение слева даёт нам `'O'`:

```
toUpper 'o' : map toUpper "n"

'O'         : map toUpper "n"
```

Справа вновь применение `map` — и наша последняя итерация:

```
map toUpper "n" = toUpper 'n' : map toUpper []
```

Выражение слева даёт нам `'N'`:

```
toUpper 'n' : map toUpper []

'N'         : map toUpper []
```

Мы вытащили из списка последний из четырёх символов, и список остался пустым. Что же мы будем делать дальше? А дальше мы вспоминаем первый вариант определения функции `map`:

```
map _ [] = []
```

Здесь функция говорит: «Как только я вторым аргументом получу пустой список, я, игнорируя первый аргумент, немедленно дам тот же самый пустой список». Поэтому оставшееся на последней итерации выражение справа:

```
map toUpper []
```

подойдёт под данный случай и просто даст нам пустой список. Всё, готово, работа функции завершена. На каждой итерации мы откусываем «голову» списка и передаём её функции `toUpper`, «хвост» же передаём вновь функции `map`. На четвёртой итерации упираемся в пустой список и возвращаем его же. Совместив все итерации воедино, получаем вот что:

```
'N' : 'E' : 'O' : 'N' : []
```

Узнаёте? Это же наш рассахаренный список, соединяющийся воедино:

```
['N', 'E', 'O', 'N']
```

Вот мы и пришли к нашему равенству:

```
  map toUpper "neon"

= map toUpper ['n', 'e', 'o', 'n']

= ['N', 'E', 'O', 'N']

= "NEON"
```

Туда и обратно

Определяя рекурсивную функцию, важно помнить о том, что в ней должно быть как правило зацикливания, так и правило выхода из цикла:

```
map _ []     = []              -- Выходим из цикла.
map f (x:xs) = f x : map f xs  -- Зацикливаемся,
                               -- применяя саму себя.
```

Если бы мы опустили первое определение, компилятор предусмотрительно сообщил бы нам о проблеме:

```
Pattern match(es) are non-exhaustive
```

И это совершенно правильно: если на каждой итерации мы уменьшаем список, то рано или поздно список точно останется пустым, а следовательно, мы обязаны объяснить, что же делать в этом случае.

Для любопытных

Открою секрет: рекурсивными в Haskell бывают не только функции, но и типы. Но об этом в последующих главах.

Глава 18

Лень

Помните, в главе с первыми вопросами о Haskell я упомянул, что этот язык является ленивым? Сейчас мы наконец-то узнаем о ленивых вычислениях и познакомимся с их светлой и тёмной сторонами.

Две модели вычислений

Как мы уже знаем, Haskell-программа состоит из выражений, а запуск программы суть начало длинной цепочки вычислений. Вспомним функцию `square`, возводящую свой единственный аргумент в квадрат:

```
main :: IO ()
main = print . square $ 4
```

Здесь всё просто: функция `square` применяется к нередуцируемому выражению `4` и даёт нам `16`. А если так:

```
main :: IO ()
main = print . square $ 2 + 2
```

Теперь функция `square` применяется уже к редуцируемому выражению:

```
square   $            2 + 2

функция  применяется  редуцируемому
         к            выражению
```

Как вы думаете, что произойдёт раньше? Применение оператора сложения или же применение функции `square`? Вопрос хитрый, ведь правильного ответа на него нет, поскольку существует две модели вычисления аргументов, а именно энергичная (англ. eager) и ленивая (англ. lazy).

При энергичной модели (называемой ещё «жадной» или «строгой») выражение, являющееся аргументом функции, будет вычислено ещё до того, как попадёт в тело

функции. На фоне определения функции `square` будет яснее:

```
square     x   = x * x
         /   \
square $ 2 + 2
         \   /
           4   = 4 * 4 = 16
```

То есть видим выражение `2 + 2`, жадно на него набрасываемся, полностью вычисляем, а уже потом результат этого вычисления передаём в функцию `square`.

При ленивой же модели всё наоборот: выражение, являющееся аргументом функции, передаётся в функцию прямо так, без вычисления. Изобразить это можно следующим образом:

```
square     x   =    x    *    x
         /   \    /   \     /   \
square $ 2 + 2 = (2 + 2) * (2 + 2) = 16
```

Но какая разница, спросите вы? Всё равно в итоге получим `16`, хоть там сложили, хоть тут. Так и есть: модель вычисления не влияет на результат этого вычисления, но она влияет на путь к этому результату.

Жадная модель нашла своё воплощение практически во всех современных языках программирования. Напишем на C:

```c
#include <stdio.h>

int strange(int i) {
  return 22;
}

int main() {
  printf("%d\n", strange(2 / 0));
}
```

Функция `strange` действительно странная, ведь она игнорирует свой аргумент и просто возвращает число `22`. И всё же при запуске этой программы вы гарантированно получите ошибку `Floating point exception`, ибо компилятор языка C категорически не терпит деления на ноль. А всё потому, что язык C придерживается энергичной модели вычислений: оператор деления `2` на `0` будет вызван ещё до того, как мы войдём в тело функции `strange`, поэтому программа упадёт.

Такой подход прямолинеен и строг: сказали нам сначала разделить на ноль — разделим, не задумываясь. Ленивая же модель придерживается иного подхода. Взгляните на Haskell-вариант:

```
strange :: Int -> Int
strange i = 22

main :: IO ()
main = print . strange $ 2 `div` 0
```

Удивительно, но при запуске этой программы мы увидим:

```
22
```

Впрочем, почему удивительно? Функция `strange`, проигнорировав свой аргумент, дала нам значение `22`, которое, попав на вход функции `print`, вылетело в наш терминал. Но где же ошибка деления `2` на `0`, спросите вы? Её нет.

Ленивый подход вполне гармонирует со своим названием: нам лень делать работу сразу же. Вместо этого мы, подобно ребёнку, которого заставили убрать разбросанные по комнате игрушки, откладываем работу до последнего. Ленивая модель гарантирует, что работа будет выполнена лишь тогда, когда результат этой работы кому-то понадобится. Если же он никому не понадобится, тогда работа не будет выполнена вовсе.

Функция `strange` ленива и потому рациональна. Она смотрит на свой аргумент `i`:

```
strange i = 22
```

и понимает, что он нигде не используется в её теле. Значит, он не нужен. А раз так, то и вычислен он не будет. Кстати, если аргумент функции игнорируется, определение принято писать с универсальным образцом:

```
strange _     = 22

        ^
        нам
        всё
        равно
```

Так и получается:

```
strange       _     = 22
          /       \
strange $ 2 `div` 0 = 22
```

Выражение, содержащее деление на ноль, попадает внутрь функции, будучи ещё невычисленным, но поскольку в теле функции оно нигде не используется, оно так и останется невычисленным. Девиз лени: если результат работы никому не нужен — зачем же её делать? Вот почему фактического деления на ноль здесь не произойдёт и программа не рухнет.

Разумеется, если бы мы определили функцию `strange` иначе:

```
strange :: Int -> Int
strange i = i + 1
```

тогда другое дело: значение аргумента уже используется в теле функции, а значит вычисление аргумента непременно произойдёт:

```
strange       i     =      i      + 1
          /       \    /       \
strange $ 2 `div` 0 = (2 `div` 0) + 1
```

Оператору сложения требуется значение обоих своих аргументов, в том числе левого, а потому получите ошибку деления на ноль.

Как можно меньше

До тех пор, пока результат вычисления никому не нужен, оно не производится. Однако даже тогда, когда результат кому-то понадобился, вычисление происходит не до конца. Помните, выше я сказал, что при жадной модели вычисления выражение, являющееся аргументом, вычисляется «полностью»? А вот при ленивой модели мы вычисляем выражение лишь настолько, насколько это необходимо. Как вышеупомянутый ребёнок, убирающий игрушки в комнате, убирает их вовсе не до конца, а лишь до такой степени, чтобы его не ругали родители.

С точки зрения вычисления любое выражение в Haskell проходит через три стадии:

1. невычисленное,
2. вычисленное не до конца,
3. вычисленное до конца.

Невычисленным называется такое выражение, которое вообще не трогали. Вспомним вышеупомянутое деление на ноль:

```
2 `div` 0
```

Мы увидели, что программа не упала, и это говорит нам о том, что деления не было. То есть функция `div` так и не была применена к своим аргументам. Вообще. Такое выражение называют thunk (можно перевести как «задумка»). То есть мы задумали применить функцию `div` к `2` и к `0`, приготовились сделать это — но в итоге так и не сделали.

Вычисленным до конца называют такое выражение, которое вычислено до своей окончательной, нередуцируемой формы. О таком выражении говорят как о выражении в «нормальной форме» (англ. normal form).

А вот вычисленным не до конца называют такое выражение, которое начали было вычислять, но сделали это не до конца, то есть не до нормальной формы, а до

так называемой «слабой головной формы» (англ. Weak Head Normal Form, WHNF). Вы спросите, как же это можно вычислить выражение не до конца? Рассмотрим пример:

```
main :: IO ()
main =
  let cx     = 2 / 6.054   -- thunk
      nk     = 4 * 12.003  -- thunk
      coeffs = [cx, nk]    -- thunk
  in putStrLn "Nothing..."
```

Есть у нас два коэффициента, `cx` и `nk`, и ещё список `coeffs`, в который мы поместили эти коэффициенты. Но, как мы видим, в итоге ни эти коэффициенты, ни этот список нам не понадобились: мы просто вывели строку и тихо вышли. В этом случае ни одно из этих выражений так и не было вычислено, оставшись в виде thunk. То есть оператор деления так и не был применён к `2` и `6.054`, оператор умножения не прикоснулся ни к `4`, ни к `12.003`, а список остался лишь в наших умах. Ленивая стратегия рациональна: зачем тратить компьютерные ресурсы на создание того, что в итоге никому не понадобится?

Изменим код:

```
main :: IO ()
main =
  let cx     = 2 / 6.054   -- thunk
      nk     = 4 * 12.003  -- thunk
      coeffs = [cx, nk]    -- WHNF
  in print $ length coeffs
```

Ага, уже интереснее. В этот раз захотелось нам узнать длину списка `coeffs`. В этом случае нам уже не обойтись без списка, иначе как же мы узнаем его длину? Однако фокус в том, что выражение `[cx, nk]` вычисляется не до конца, а лишь до той своей формы, которая удовлетворит функцию `length`.

Задумаемся: функция `length` возвращает число элементов списка, но какое ей дело до содержимого этих элементов? Ровным счётом никакого. Поэтому в данном случае список формируется из thunk-ов:

```
coeffs = [thunk, thunk]
```

Первым элементом этого списка является thunk, ассоциированный с невычисленным выражением `2 / 6.054`, а вторым элементом списка является thunk, ассоциированный с невычисленным выражением `4 * 12.003`. Фактически, список `coeffs` получился как бы не совсем настоящим, пустышечным: он был сформирован в памяти как корректный список, однако внутри обоих его элементов — вакуум. И всё же даже такая его форма вполне подходит для функции `length`, которая и так прекрасно поймёт, что в списке два элемента. О таком списке говорят как о выражении в слабой головной форме.

Ещё чуток изменим код:

```
main :: IO ()
main =
  let cx     = 2 / 6.054   -- thunk
      nk     = 4 * 12.003  -- normal
      coeffs = [cx, nk]    -- WHNF
  in print $ coeffs !! 1
```

Необычного вида оператор `!!` извлекает из списка элемент по индексу, в данном случае нас интересует второй по счёту элемент. Теперь нам уже недостаточно просто сформировать список, нам действительно нужен его второй элемент, иначе как бы мы смогли вывести его на консоль? В этом случае выражение `4 * 12.003` будет вычислено до своей окончательной, нормальной формы, а результат этого вычисления ляжет вторым элементом списка, вот так:

```
coeffs = [thunk, 48.012]
```

Однако первый элемент списка так и остался невостребованным, поэтому выражение `2 / 6.054` по-прежнему остаётся лишь нашей мыслью, не более чем. В этом случае список `coeffs` всё равно остаётся в слабой головной форме, ведь внутри первого его элемента всё ещё вакуум.

И теперь напишем так:

```
main :: IO ()
main =
  let cx     = 2 / 6.054   -- normal
      nk     = 4 * 12.003  -- normal
      coeffs = [cx, nk]    -- normal
  in print coeffs
```

Вот, теперь никакой лени. Список `coeffs` должен быть выведен на консоль полностью, а следовательно, оба его элемента должны быть вычислены до своей нормальной формы, в противном случае мы не смогли бы показать их в консоли.

Вот философия ленивой стратегии: даже если нам нужно вычислить выражение, мы вычисляем его лишь до той формы, достаточной в конкретных условиях, и не более того.

Рациональность

Как уже было упомянуто, ленивая стратегия помогает программе быть рациональной и не делать лишнюю работу. Рассмотрим пример:

```
main :: IO ()
main = print $ take 5 evens
  where evens = [2, 4 .. 100]
```

Список `evens`, формируемый через арифметическую последовательность, содержит в себе чётные числа от `2` до `100` включительно. Используется этот список в качестве второго аргумента стандартной функции `take`, которая даёт нам N первых элементов из переданного ей списка:

```
take    5          evens

возьми  лишь
        пять
        элементов  из этого
                   списка
```

При запуске этой программы мы получим ожидаемый результат:

```
[2,4,6,8,10]
```

В чём же здесь рациональность, спросите вы? А в том, что список `evens` в итоге содержал в себе лишь 5 элементов. Да, но ведь чётных чисел от `2` до `100` куда больше, нежели пять! Совершенно верно, но лень позволяет нам сделать лишь столько работы, сколько реально требуется. Раз уж список `evens` нужен лишь функции `take`, которая, в свою очередь, хочет только пять первых его элементов — зачем же создавать оставшиеся элементы? Нужно первые пять — получи пять. Если же напишем так:

```
main :: IO ()
main = print $ take 50 evens
  where evens = [2, 4 .. 100]
```

тогда в списке `evens` окажется уже пятьдесят элементов, потому что именно столько запросила функция `take`. Повторю философию ленивого рационализма: сделаем не столько, сколько нам сказали, а лишь столько, сколько действительно понадобится.

Бесконечность

А что будет, если мы запросим из списка `evens` 500 элементов? Вот так:

```
main :: IO ()
main = print $ take 500 evens
  where evens = [2, 4 .. 100]
```

Ничего страшного не случится, функция `take` проверяет выход за границы и в случае, если её первый аргумент превышает длину списка, она просто даёт нам тот же список. Да, но ведь мы хотим увидеть пятьсот чётных чисел, а не пятьдесят! Можно было бы увеличить список:

```
main :: IO ()
main = print $ take 500 evens
  where evens = [2, 4 .. 100000]
```

но это ненадёжно, ведь потом опять может потребоваться ещё больше. Нужно что-нибудь универсальное, и в Haskell есть подходящее решение:

```
main :: IO ()
main = print $ take 500 evens
  where evens = [2, 4 ..]  -- Что это?
```

Теперь не сомневайтесь: в списке `evens` будет не менее пятисот чётных чисел. Но что это за конструкция такая? Начало дано, шаг дан, а где же конец? Познакомьтесь, это бесконечный список:

```
[2, 4 ..]
```

Ленивая модель вычислений позволяет нам работать с бесконечными структурами данных. Вот прямо так, начиная с двойки и, с шагом через один, уходим в бесконечные дали... Шучу. На самом деле, список получится вовсе не бесконечным, а настолько большим, насколько нам это понадобится.

В самом деле, если функция `take` требует от нас N элементов — зачем нам вообще задавать окончание диапазона списка? Всё равно в нём будет не более чем N. Бесконечная структура данных тем и полезна, что из неё всегда можно взять столько, сколько требуется.

Конечно, если бы мы решили похулиганить:

```
main :: IO ()
main = print evens  -- Дай нам всё!
  where evens = [2, 4 ..]
```

в этом случае в нашу консоль быстро посыпалось бы очень много чисел...

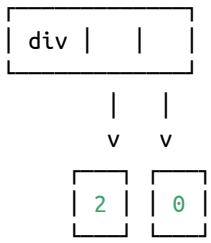
Space leak

Да, я должен рассказать вам правду: есть у ленивой стратегии вычислений тёмная сторона, получившая название space leak (букв. «утечка пространства»). И вот в чём её суть.

Вспомним пример с делением:

```
main :: IO ()
main = print . strange $ 2 `div` 0
```

Как мы помним, деления на ноль так и не произошло за ненадобностью его результата. В этом случае выражение осталось в виде thunk. Возникает вопрос: что же с ним стало? У нас есть функция `div` и есть два значения типа `Int`, `2` и `0`. Если функция `div` так и не была применена к ним, где же всё это хозяйство находилось в процессе работы нашей программы? Оно находилось в памяти, в виде особого графа, который можно изобразить так:



То есть сама функция и два значения, которые должны были занять место двух её аргументов. И вот этот граф в памяти так и остался невостребованным. Казалось бы, ну и в чём проблема? А проблема в количестве. Если мы смогли написать код, при работе которого в память отложился один thunk, значит теоретически мы можем написать и такой код, количество thunk-ов при работе которого будет исчисляться миллионами. А учитывая тот факт, что каждый thunk занимает в памяти хотя бы несколько байт, вы можете себе представить масштаб проблемы.

Причём возникнуть эта проблема может из весьма невинного на первый взгляд кода:

```
bad :: [Int] -> Int -> Int
bad []         c = c
bad (_:others) c = bad others $ c + 1
```

Простенькая рекурсивная функция, пробегающаяся по ненужному ей списку и увеличивающаяся свой второй аргумент на единицу. Но я не просто так назвал её `bad`. Давайте применим её:

```
bad [1, 2, 3] 0
```

Подставим в определение, содержащее зацикливание:

```
bad (_: others) c = bad others $ c + 1

bad [1, 2, 3]   0 = bad [2, 3] $ 0 + 1

        ____            ______

                =               =
```

«Голова» списка откусывается и игнорируется, а к `0` прибавляется `1`. Но поскольку результат сложения пока что никому не нужен, сложение не производится. Вместо этого, на второй итерации, мы видим следующее:

```
bad [2, 3] $ 0 + 1 = bad [3] $ (0 + 1) + 1
```

К предыдущему выражению вновь прибавляется единица — и мы опять входим в очередную итерацию, так и не выполнив сложения:

```
bad [3] $ (0 + 1) + 1 = bad [] $ ((0 + 1) + 1) + 1
```

Опа! Упёрлись в пустой список, вспоминаем правило выхода из рекурсии:

```
bad [] c = c
```

Итак, в этом случае мы просто возвращаем значение второго аргумента. Сделаем же это:

```
bad [] $ ((0 + 1) + 1) + 1 = ((0 + 1) + 1) + 1 = 3
```

И вот только здесь мы реально вычисляем второй аргумент, складывая три единицы. Вы спросите, почему же мы накапливали эти сложения вместо того, чтобы делать их сразу? Потому что мы ленивы: раз результат сложения понадобился нам лишь на последней итерации, значит до этой итерации никакого сложения не будет, ведь лень вынуждает нас откладывать работу до конца.

Вот в этом-то накоплении вся беда. Представим, что мы написали так:

```
main :: IO ()
main = print $ bad [1..50000000] 0
```

50 миллионов элементов, а значит, 50 миллионов раз сложение второго аргумента с единицей будет откладываться, накапливая гигантский «хвост» из (пока что) невычисленных выражений. Хотите знать, что произойдёт при запуске такой программы? Её выполнение, на MacBook Pro 2014 года, займёт приблизительно 63 секунды и скушает, ни много ни мало, 6,4 ГБ памяти! А теперь представьте, что случилось бы, если бы элементов в списке было не 50 миллионов, а 50 миллиардов...

Иногда space leak ошибочно путают с другой проблемой, называемой memory leak (англ. «утечка памяти»), однако это вовсе не одно и то же. Утечка памяти — это ошибка, характерная для языков с ручным управлением памятью, например, C. Если мы выделим память в куче (англ. heap), а затем потеряем указатель, связывающий нас с этой памятью — всё, выделенная память утекла, она потеряна для нас навеки. Но в случае space leak мы не теряем память: когда весь этот «хвост» из сложений в конце концов вычислится, память, занимаемая миллионами thunk-ов, освободится. Мы не теряем память, мы просто используем её слишком много.

Борьба

Проблема space leak вытекает из самой природы ленивых вычислений. Многие программисты, узнав об этой проблеме, отворачиваются от Haskell. Мол, если в этом языке можно легко написать код, сжирающий уймищу памяти, значит этот язык точно не подходит для серьёзного использования. Но не так страшен чёрт, как его малюют. Я расскажу о двух способах борьбы со space leak.

Впрочем, с концептуальной точки зрения способ всего один. Задумаемся: если в примере выше лень явилась причиной откладывания сложений на потом, что же можно сделать? Ответ прост: мы должны убрать излишнюю ленивость и заменить её строгостью. В этом случае применение оператора сложения уже не будет откладываться до последнего, а будет производиться тут же, как в языках со строгой моделью вычислений.

И как же мы можем разбавить лень строгостью? Вот два способа.

Оптимизация

Первый способа самый простой — оптимизация. Когда компилятор превращает наш код в программу, его можно попросить оптимизировать наш код, сделав его более эффективным, по тем или иным критериям. Чтобы попросить компилятор провести оптимизацию, мы должны использовать специальный флаг. Откроем сборочный файл нашего проекта `real.cabal`, найдём секцию `executable real-exe`, в которой есть строка:

```
ghc-options:         ...
```

Эта строка содержит различные опции компилятора GHC, и оптимизационный флаг дописывается именно сюда. Попробуем подставить туда сначала флаг `-O0`, а затем `-O2`. Результаты запуска программы будут такими:

Оптимизация	Время	Память
-O0	63 с	6,4 ГБ
-O2	3,2 с	104 кБ

Впечатляющая разница, не правда ли? Флаг `-O0` говорит компилятору о том, чтобы тот не производил никакую оптимизацию, в этом случае говорят о нулевом уровне оптимизации. Флаг `-O2`, напротив, устанавливает стандартный для production-проектов уровень оптимизации. Так вот при стандартном уровне компилятор способен распознать излишнюю ленивость в нашем коде и добавить чуток жадности. В примере выше компилятор увидит накопление thunk-ов сложения и пресечёт оное. Согласитесь, с гигабайтов прыгнуть сразу на килобайты — это круто.

Так что же, проблемы нет? Ну, если оптимизация `-O2` и так стандартна — так давайте ставить её в наши проекты и забудем про space leak! К сожалению, не всё так

просто.

Во-первых, компиляторная оптимизация сродни чёрной магии, на неё трудно полагаться. Мы очень благодарны компилятору GHC за попытку помочь нам, но эта помощь не всегда соответствует нашим ожиданиям. И во-вторых, к сожалению, компилятор не всегда способен распознать излишнюю лень в нашем коде, и в этом случае нам приходится-таки прибегнуть ко второму способу борьбы со space leak.

Вручную

Вернёмся к определению функции `bad`:

```
bad :: [Int] -> Int -> Int
bad []         c = c
bad (_:others) c = bad others $ c + 1
```

Проблема, как мы уже поняли, во втором аргументе:

```
bad others $ c + 1

             накопление
             thunk-ов...
```

Превратим же злую функцию в добрую:

```
good :: [Int] -> Int -> Int
good []         c = c
good (_:others) c = good others $! c + 1
```

Этот код даст нам приблизительно такой же выигрыш, что и оптимизация уровня `-O2`: секунды вместо минуты и килобайты вместо гигабайтов. Что же изменилось? Смотрим внимательно:

```
good others $! c + 1

            ^
```

Вместо привычного оператора применения `$` мы видим оператор строго применения `$!` (англ. strict application operator). Этот оператор говорит аргументу: «Забудь о лени, я приказываю тебе немедленно вычислиться до слабой головной формы»:

```
good others $!       c + 1

            вычисли  этот
                     аргумент

            строго,
            а не
            лениво!
```

Вот потому-то наш «хвост» из thunk-ов и не будет накапливаться, ведь на каждой из 50 миллионов итераций будет происходить незамедлительное применение оператора сложения. Таким образом, заставить аргумент тут же вычислиться до слабой головной или нормальной формы можно как посредством того, что этот аргумент прямо сейчас кому-то понадобился, так и посредством строгого применения.

Лень и строгость вместе

Функцию называют ленивой по тем аргументам, которые не вычисляется, и строгой по тем аргументам, которые вычисляются. Примитивный пример:

```haskell
fakeSum :: Int -> Int -> Int
fakeSum x _ = x + 100
```

Функция `fakeSum` строга по своему первому аргументу и ленива по своему второму аргументу. Первый аргумент `x` непременно будет вычислен, ведь он передаётся оператору сложения. Второй же аргумент игнорируется, оставшись невычисленным. И кстати, существует простой способ проверить, строга ли функция по некоторому аргументу или ленива.

В стандартной библиотеке Haskell определена особая функция `undefined`. Это — чёрная дыра: при попытке прикоснуться к ней программа гарантированно падает с ошибкой. Проверяем:

```haskell
main :: IO ()
main = print $ fakeSum 1 undefined
```

В этом случае мы получим результат:

```
101
```

Чёрная дыра была проигнорирована, ведь функция `fakeSum` ленива по второму аргументу. Если же мы напишем так:

```
main :: IO ()
main = print $ fakeSum undefined 45
```

программа, попытавшись передать `undefined` оператору сложения, аварийно остановится. Или вот другой пример:

```
main :: IO ()
main = print . head $ [23, undefined, undefined]
```

Не сомневайтесь: программа спокойно вернёт нам `23`, ведь функция `head` строга лишь по первому элементу переданного ей списка, остальное содержимое оного её абсолютно не интересует. Но если попробуете вытащить второй или третий элемент из подобного списка — крах неминуем.

Для любопытных

Haskell — не первый язык с ленивой стратегией вычислений. Открою вам исторический факт: у языка Haskell был предшественник, язык программирования с красивым женским именем Miranda. Лень и чистая функциональность пришли в Haskell именно из Miranda, и лишь в этих двух языках ленивая стратегия вычисления аргументов используется по умолчанию. На сегодняшний день, насколько мне известно, язык Miranda мёртв. Впрочем, как сугубо исследовательский язык он, может быть, кем-то и используется.

Что же касается проблемы space leak, то к счастью, существуют способы обнаружения функций, шибко прожорливых до памяти. В самом деле, представьте себе большой проект, тысячи функций, и что-то кушает гигабайты памяти. Как найти виновного? Этот процесс называют ещё «space leak профилированием». Рассказывать об этом здесь я не стану, материал довольно объёмный. Но для особо любопытных привожу ссылку на неплохую англоязычную статью по теме: Chasing a Space Leak in Shake.

И ещё вспомним вот это:

```
square     x   =    x      *    x
         /   \    /   \       /   \
square $ 2 + 2 = (2 + 2)   * (2 + 2)    = 16

                вычисляем   и что,
                            опять
                            вычисляем?!
```

Внимательный читатель удивится, мол, неужели выражение `2 + 2` вычисляется дважды?! Ведь это нерационально. Конечно нерационально, поэтому в действительности оно будет вычислено единожды. В Haskell есть особый механизм «шаринга» (англ. sharing), позволяющий избежать напрасной работы. И если у нас есть несколько одинаковых выражений, вычисление оного происходит один

раз, результат же сохраняется и потом просто подставляется в нужные места. Например:

```
main :: IO ()
main =
  let x = sin 2 in print x * x
```

Если бы не sharing-механизм, функция `sin` была бы применена к `2` дважды. К счастью, значение синуса будет вычислено единожды и тут же сохранено, чтобы потом просто встать на места тех двух `x`.

Глава 19

Наши типы

Вот мы и добрались до Второго Кита Haskell — до **Типов**. Конечно, мы работали с типами почти с самого начала, но вам уже порядком надоели все эти `Int` и `String`, не правда ли? Пришла пора познакомиться с типами куда ближе.

Знакомство

Удивительно, но в Haskell очень мало встроенных типов, то есть таких, о которых компилятор знает с самого начала. Есть `Int`, есть `Double`, `Char`, ну и ещё несколько. Все же остальные типы, даже носящие статут стандартных, не являются встроенными в язык. Вместо этого они определены в стандартной или иных библиотеках, причём определены точно так же, как мы будем определять и наши собственные типы. А поскольку без своих типов написать сколь-нибудь серьёзное приложение у нас не получится, тема эта достойна самого пристального взгляда.

Определим тип `Transport` для двух известных протоколов транспортного уровня модели OSI:

```
data Transport = TCP | UDP
```

Перед нами — очень простой, но уже наш собственный тип. Рассмотрим его внимательнее.

Ключевое слово `data` — это начало определение типа. Далее следует название типа, в данном случае `Transport`. Имя любого типа обязано начинаться с большой буквы. Затем идёт знак равенства, после которого начинается фактическое описание типа, его «тело». В данном случае оно состоит из двух простейших конструкторов. Конструктор значения (англ. data constructor) — это то, что строит значение данного типа. Здесь у нас два конструктора, `TCP` и `UDP`, каждый из которых строит значение типа `Transport`. Имя конструктора тоже обязано начинаться с большой буквы. Иногда для краткости конструктор значения называют просто конструктором.

Подобное определение легко читается:

```
data  Transport  =    TCP  |    UDP

тип   Transport  это  TCP  или  UDP
```

Теперь мы можем использовать тип `Transport`, то есть создавать значения этого типа и что-то с ними делать. Например, в `let`-выражении:

```
let protocol = TCP
```

Мы создали значение `protocol` типа `Transport`, использовав конструктор `TCP`. А можно и так:

```
let protocol = UDP
```

Хотя мы использовали разные конструкторы, тип значения `protocol` в обоих случаях один и тот же — `Transport`.

Расширить подобный тип предельно просто. Добавим новый протокол SCTP (Stream Control Transmission Protocol):

```
data Transport = TCP | UDP | SCTP
```

Третий конструктор значения дал нам третий способ создать значение типа `Transport`.

Значение-пустышка

Задумаемся: говоря о значении типа `Transport` — о чём в действительности идёт речь? Казалось бы, значения-то фактического нет: ни числа никакого, ни строки, просто три конструктора. Так вот они и есть значения. Когда мы пишем:

```
let protocol = SCTP
```

мы создаём значение типа `Transport` с конкретным содержимым в виде `SCTP`. Конструктор — это и есть содержимое. Данный вид конструктора называется нульарным (англ. nullary). Тип `Transport` имеет три нульарных конструктора. И даже столь простой тип уже может быть полезен нам:

```
checkProtocol :: Transport -> String
checkProtocol transport = case transport of
  TCP  -> ”That’s TCP protocol.”
  UDP  -> ”That’s UDP protocol.”
  SCTP -> ”That’s SCTP protocol.”

main :: IO ()
main = putStrLn . checkProtocol $ TCP
```

В результате увидим:

```
That’s TCP protocol.
```

Функция `checkProtocol` объявлена как принимающая аргумент типа `Transport`, а применяется она к значению, порождённому конструктором `TCP`. В данном случае конструкция `case-of` сравнивает аргумент с конструкторами. Именно поэтому нам не нужна функция `otherwise`, ведь никаким иным способом, кроме как с помощью трёх конструкторов, значение типа `Transport` создать невозможно, а значит, один из конструкторов гарантированно совпадёт.

Тип, состоящий только из нульарных конструкторов, называют ещё перечислением (англ. enumeration). Конструкторов может быть сколько угодно, в том числе один-единственный (хотя польза от подобного типа была бы невелика). Вот ещё один известный пример:

```
data Day = Sunday
         | Monday
         | Tuesday
         | Wednesday
         | Thursday
         | Friday
         | Saturday
```

Обратите внимание на форматирование, когда ментальные «ИЛИ» выровнены строго под знаком равенства. Такой стиль вы встретите во многих реальных Haskell-проектах.

Значение типа `Day` отражено одним из семи конструкторов. Сделаем же с ними что-нибудь:

```haskell
data WorkMode = FiveDays | SixDays

workingDays :: WorkMode -> [Day]
workingDays FiveDays = [ Monday
                       , Tuesday
                       , Wednesday
                       , Thursday
                       , Friday
                       ]
workingDays SixDays = [ Monday
                      , Tuesday
                      , Wednesday
                      , Thursday
                      , Friday
                      , Saturday
                      ]
```

Функция `workingDays` возвращает список типа `[Day]`, и в случае пятидневной рабочей недели, отражённой конструктором `FiveDays`, этот список сформирован пятью конструкторами, а в случае шестидневной — шестью конструкторами.

Польза от типов, сформированных нульарными конструкторами, не очень велика, хотя встречаться с такими типами вы будете часто.

Приоткрою секрет: новый тип можно определить не только с помощью ключевого слова `data`, но об этом узнаем в одной из следующих глав.

А теперь мы можем познакомиться с типами куда более полезными.

Глава 20

АТД

АТД, или Алгебраические Типы Данных (англ. ADT, Algebraic Data Type), занимают почётное место в мире типов Haskell. Абсолютно подавляющее большинство ваших собственных типов будут алгебраическими, и то же можно сказать о типах из множества Haskell-пакетов. Алгебраическим типом данных называют такой тип, которые составлены из других типов. Мы берём простые типы и строим из них, как из кирпичей, типы сложные, а из них — ещё более сложные. Это даёт нам невероятный простор для творчества.

Оставим сетевые протоколы и дни недели, рассмотрим такой пример:

```
data IPAddress = IPAddress String
```

Тип `IPAddress` использует один-единственный конструктор значения, но кое-что изменилось. Во-первых, имена типа и конструктора совпадают. Это вполне легально, вы встретите такое не раз. Во-вторых, конструктор уже не нульарный, а унарный (англ. unary), потому что теперь он связан с одним значением типа `String`. И вот как создаются значения типа `IPAddress`:

```
let ip = IPAddress "127.0.0.1"
```

Значение `ip` типа `IPAddress` образовано конструктором и конкретным значением некоего типа:

```
let ip = IPAddress       "127.0.0.1"

         конструктор     значение
         значения        типа
         типа IPAddress  String

         └ значение типа IPAddress ┘
```

Значение внутри нашего типа называют ещё полем (англ. field):

```
data IPAddress = IPAddress    String

     тип         конструктор  поле
```

Расширим тип `IPAddress`, сделав его более современным:

```
data IPAddress = IPv4 String | IPv6 String
```

Теперь у нас два конструктора, соответствующие разным IP-версиям. Это позволит нам создавать значение типа `IPAddress` так:

```
  let ip = IPv4 "127.0.0.1"
```

или так:

```
  let ip = IPv6 "2001:0db8:0000:0042:0000:8a2e:0370:7334"
```

Сделаем тип ещё более удобным. Так, при работе с IP-адресом нам часто требуется `localhost`. И чтобы явно не писать `"127.0.0.1"` и `"0:0:0:0:0:0:0:1"`, введём ещё два конструктора:

```
data IPAddress = IPv4 String
               | IPv4Localhost
               | IPv6 String
               | IPv6Localhost
```

Поскольку значения `localhost` нам заведомо известны, нет нужды указывать их явно. Вместо этого, когда нам понадобится `IPv4-localhost`, пишем так:

```
    let ip = IPv4Localhost
```

Извлекаем значение

Допустим, мы создали значение `google`:

```
  let google = IPv4 "173.194.122.194"
```

Как же нам потом извлечь конкретное строковое значение из `google`? С помощью нашего старого друга, паттерн матчинга:

```
checkIP :: IPAddress -> String
checkIP (IPv4 address) = "IP is '" ++ address ++ "'."

main :: IO ()
main = putStrLn . checkIP $ IPv4 "173.194.122.194"
```

Результат:

```
IP is '173.194.122.194'.
```

Взглянем на определение:

```
checkIP (IPv4 address) = "IP is '" ++ address ++ "'."
```

Здесь мы говорим: «Мы знаем, что значение типа `IPAddress` сформировано с конструктором и строкой». Однако внимательный компилятор сделает нам замечание:

```
Pattern match(es) are non-exhaustive
In an equation for 'checkIP':
    Patterns not matched:
        IPv4Localhost
        IPv6 _
        IPv6Localhost
```

В самом деле, откуда мы знаем, что значение, к которому применили функцию `checkIP`, было сформировано именно с помощью конструктора `IPv4`? У нас же есть ещё три конструктора, и нам следует проверить их все:

```
checkIP :: IPAddress -> String
checkIP (IPv4 address) = "IPv4 is '" ++ address ++ "'."
checkIP IPv4Localhost  = "IPv4, localhost."
checkIP (IPv6 address) = "IPv6 is '" ++ address ++ "'."
checkIP IPv6Localhost  = "IPv6, localhost."
```

С каким конструктором совпало — с таким и было создано значение. Можно, конечно, и так проверить:

```
checkIP :: IPAddress -> String
checkIP addr = case addr of
    IPv4 address  -> "IPv4 is '" ++ address ++ "'."
    IPv4Localhost -> "IPv4, localhost."
    IPv6 address  -> "IPv6 is '" ++ address ++ "'."
    IPv6Localhost -> "IPv6, localhost."
```

Строим

Определим тип для сетевой точки:

```
data EndPoint = EndPoint String Int
```

Конструктор `EndPoint` — бинарный, ведь здесь уже два значения. Создаём обычным образом:

```
  let googlePoint = EndPoint "173.194.122.194" 80
```

Конкретные значения извлекаем опять-таки через паттерн матчинг:

```
main :: IO ()
main = putStrLn $ "The host is: " ++ host
  where
    EndPoint host _ = EndPoint "173.194.122.194" 80

    └── образец ──┘   └──────── значение ─────────┘
```

Обратите внимание, что второе поле, соответствующее порту, отражено универсальным образцом _, потому что в данном случае нас интересует только значение хоста, а порт просто игнорируется.

И всё бы хорошо, но тип `EndPoint` мне не очень нравится. Есть в нём что-то некрасивое. Первым полем выступает строка, содержащая IP-адрес, но зачем нам строка? У нас же есть прекрасный тип `IPAddress`, он куда лучше безликой строки. Это общее правило для Haskell-разработчика: чем больше информации несёт в себе тип, тем он лучше. Давайте заменим определение:

```
data EndPoint = EndPoint IPAddress Int
```

Тип стал понятнее, и вот как мы теперь будем создавать значения:

```
  let google = EndPoint (IPv4 "173.194.122.194") 80
```

Красиво. Извлекать конкретные значения будем так:

```
main :: IO ()
main = putStrLn $ "The host is: " ++ ip
  where
    EndPoint (IPv4 ip) _ = EndPoint (IPv4 "173.194.122.194") 80
             ~~~~                    ~~~~

                 ==                      =================
```

Здесь мы опять-таки игнорируем порт, но значение IP-адреса извлекаем уже на основе образца с конструктором `IPv4`.

Это простой пример того, как из простых типов строятся более сложные. Но сложный тип вовсе не означает сложную работу с ним, паттерн матчинг элегантен как всегда. А вскоре мы узнаем о другом способе работы с полями типов, без паттерн матчинга.

Любопытно, что конструкторы типов тоже можно компоновать, взгляните:

```
main :: IO ()
main = putStrLn $ "The host is: " ++ ip
  where
    EndPoint (IPv4 ip) _ = (EndPoint . IPv4 $ "173.194.122.194") 80
```

Это похоже на маленькое волшебство, но конструкторы типов можно компоновать знакомым нам оператором композиции функций:

```
(EndPoint . IPv4 $ "173.194.122.194") 80

            |       значение типа      |
            └──────── IPAddress ────────┘
```

Вам это ничего не напоминает? Это же в точности так, как мы работали с функциями! Из этого мы делаем вывод: конструктор значения можно рассматривать как особую функцию. В самом деле:

```
EndPoint   (IPv4 "173.194.122.194")  80

"функция"  |        первый         | второй
           └─────── аргумент ───────┘ аргумент
```

Мы как бы применяем конструктор к конкретным значениям как к аргументам, в результате чего получаем значение нашего типа. А раз так, мы можем компоновать конструкторы так же, как и обычные функции, лишь бы их типы были комбинируемыми. В данном случае всё в порядке: тип значения, возвращаемого конструктором `IPv4`, совпадает с типом первого аргумента конструктора `EndPoint`.

Вот мы и познакомились с настоящими типами. Пришло время узнать о более удобной работе с полями типов.

Глава 21

АТД: поля с метками

Многие типы в реальных проектах довольно велики. Взгляните:

```
data Arguments = Arguments Port
                           Endpoint
                           RedirectData
                           FilePath
                           FilePath
                           Bool
                           FilePath
```

Значение типа `Arguments` хранит в своих полях некоторые значения, извлечённые из параметров командной строки, с которыми запущена одна из моих программ. И всё бы хорошо, но работать с таким типом абсолютно неудобно. Он содержит семь полей, и паттерн матчинг был бы слишком громоздким, представьте себе:

```
...
  where
    Arguments _ _ _ redirectLib _ _ xpi = arguments
```

Более того, когда мы смотрим на определение типа, назначение его полей остаётся тайной за семью печатями. Видите предпоследнее поле? Оно имеет тип `Bool` и, понятное дело, отражает какой-то флаг. Но что это за флаг, читатель не представляет. К счастью, существует способ, спасающих нас от обеих этих проблем.

Метки

Мы можем снабдить наши поля метками (англ. label). Вот как это выглядит:

```
data Arguments = Arguments { runWDServer    :: Port
                           , withWDServer   :: Endpoint
                           , redirect       :: RedirectData
                           , redirectLib    :: FilePath
                           , screenshotsDir :: FilePath
                           , noScreenshots  :: Bool
                           , harWithXPI     :: FilePath
                           }
```

Теперь назначение меток куда понятнее. Схема определения такова:

```
data Arguments = Arguments   { runWDServer :: Port }

тип  такой-то    конструктор   метка поля     тип
                                              поля
```

Теперь поле имеет не только тип, но и название, что и делает наше определение значительно более читабельным. Поля в этом случае разделены запятыми и заключены в фигурные скобки.

Если подряд идут два или более поля одного типа, его можно указать лишь для последней из меток. Так, если у нас есть вот такой тип:

```
data Patient = Patient { firstName :: String
                       , lastName  :: String
                       , email     :: String
                       }
```

его определение можно чуток упростить и написать так:

```
data Patient = Patient { firstName
                       , lastName
                       , email     :: String
                       }
```

Раз тип всех трёх полей одинаков, мы указываем его лишь для последней из меток. Ещё пример полной формы:

```
data Patient = Patient { firstName    :: String
                       , lastName     :: String
                       , email        :: String
                       , age          :: Int
                       , diseaseId    :: Int
                       , isIndoor     :: Bool
                       , hasInsurance :: Bool
                       }
```

и тут же упрощаем:

```
data Patient = Patient { firstName
                       , lastName
                       , email       :: String
                       , age
                       , diseaseId   :: Int
                       , isIndoor
                       , hasInsurance :: Bool
                       }
```

Поля `firstName`, `lastName` и `email` имеют тип `String`, поля `age` и `diseaseId` — тип `Int`, и оставшиеся два поля — тип `Bool`.

Getter и Setter?

Что же представляют собой метки? Фактически, это особые функции, сгенерированные автоматически. Эти функции имеют три предназначения: создавать, извлекать и изменять. Да, я не оговорился, изменять. Но об этом чуть позже, пусть будет маленькая интрига.

Вот как мы создаём значение типа `Patient`

```
main :: IO ()
main = print $ diseaseId patient
  where
    patient = Patient {
        firstName    = "John"
      , lastName     = "Doe"
      , email        = "john.doe@gmail.com"
      , age          = 24
      , diseaseId    = 431
      , isIndoor     = True
      , hasInsurance = True
    }
```

Метки полей используются как своего рода setter (от англ. set, «устанавливать»):

```
patient = Patient { firstName    =      "John"
в этом    типа      поле с
значении  Patient   этой меткой  равно  этой строке
```

Кроме того, метку можно использовать и как getter (от англ. get, «получать»):

```
main = print $ diseaseId  patient

               метка как  аргумент
               функция
```

Мы применяем метку к значению типа `Patient` и получаем значение соответствующего данной метке поля. Поэтому для получения значений полей нам уже не нужен паттерн матчинг.

Но что же за интригу я приготовил под конец? Выше я упомянул, что метки используются не только для задания значений полей и для их извлечения, но и для изменения. Вот что я имел в виду:

```
main :: IO ()
main = print $ email patientWithChangedEmail
  where
    patientWithChangedEmail = patient {
      email = "j.d@gmail.com"  -- Изменяем???
    }

    patient = Patient {
        firstName    = "John"
      , lastName     = "Doe"
      , email        = "john.doe@gmail.com"
      , age          = 24
      , diseaseId    = 431
      , isIndoor     = True
      , hasInsurance = True
    }
```

При запуске программы получим:

```
j.d@gmail.com
```

Но постойте, что же тут произошло? Ведь в Haskell, как мы знаем, нет оператора присваивания, однако значение поля с меткой `email` поменялось. Помню, когда я впервые увидел подобный пример, то очень удивился, мол, уж не ввели ли меня в заблуждение по поводу неизменности значений в Haskell?!

Нет, не ввели. Подобная запись:

```
patientWithChangedEmail = patient {
  email = "j.d@gmail.com"
}
```

действительно похожа на изменение поля через присваивание ему нового значения, но в действительности никакого изменения не произошло. Когда я назвал метку setter-ом, я немного слукавил, ведь классический setter из мира ООП был бы невозможен в Haskell. Посмотрим ещё раз внимательнее:

```
...
  where
    patientWithChangedEmail = patient {
      email = "j.d@gmail.com"  -- Изменяем???
    }

    patient = Patient {
        firstName    = "John"
      , lastName     = "Doe"
      , email        = "john.doe@gmail.com"
      , age          = 24
      , diseaseId    = 431
      , isIndoor     = True
      , hasInsurance = True
    }
```

Взгляните, ведь у нас теперь два значения типа `Patient`, `patient` и `patientWithChangedEmail`. Эти значения не имеют друг ко другу ни малейшего отношения. Вспомните, как я говорил, что в Haskell нельзя изменить имеющееся значение, а можно лишь создать на основе имеющегося новое значение. Это именно то, что здесь произошло: мы взяли имеющееся значение `patient` и на его основе создали уже новое значение `patientWithChangedEmail`, значение поля `email` в котором теперь другое. Понятно, что поле `email` в значении `patient` осталось неизменным.

Будьте внимательны при инициализации значения с полями: вы обязаны предоставить значения для всех полей. Если вы напишете так:

```
main :: IO ()
main = print $ email patientWithChangedEmail
  where
    patientWithChangedEmail = patient {
      email = "j.d@gmail.com"  -- Изменяем???
    }

    patient = Patient {
        firstName    = "John"
      , lastName     = "Doe"
      , email        = "john.doe@gmail.com"
      , age          = 24
      , diseaseId    = 431
      , isIndoor     = True
    }

    -- Поле hasInsurance забыли!
```

код скомпилируется, но внимательный компилятор предупредит вас о проблеме:

```
Fields of 'Patient' not initialised: hasInsurance
```

Пожалуйста, не пренебрегайте подобным предупреждением, ведь если вы проигнорируете его и затем попытаетесь обратиться к неинициализированному полю:

```
main = print $ hasInsurance patient
  ...
```

ваша программа аварийно завершится на этапе выполнения с ожидаемой ошибкой:

```
Missing field in record construction hasInsurance
```

Не забывайте: компилятор — ваш добрый друг.

Без меток

Помните, что метки полей — это синтаксический сахар, без которого мы вполне можем обойтись. Даже если тип был определён с метками, как наш `Patient`, мы можем работать с ним по-старинке:

```
data Patient = Patient { firstName    :: String
                       , lastName     :: String
                       , email        :: String
                       , age          :: Int
                       , diseaseId    :: Int
                       , isIndoor     :: Bool
                       , hasInsurance :: Bool
                       }

main :: IO ()
main = print $ hasInsurance patient
  where
    -- Создаём по-старинке...
    patient = Patient "John"
                      "Doe"
                      "john.doe@gmail.com"
                      24
                      431
                      True
                      True
```

Соответственно, извлекать значения полей тоже можно по-старинке, через паттерн матчинг:

```haskell
main :: IO ()
main = print insurance
  where
    -- Жутко неудобно, но если желаете...
    Patient _ _ _ _ _ _ insurance = patient
    patient = Patient "John"
                      "Doe"
                      "john.doe@gmail.com"
                      24
                      431
                      True
                      True
```

С другими видами синтаксического сахара мы встретимся ещё не раз, на куда более продвинутых примерах.

Глава 22

Новый тип

Помимо `data` существует ещё одно ключевое слово, предназначенное для определения нового типа. Оно так и называется — `newtype`. Эти слова похожи друг на друга «в одну сторону»: вы можете поставить `data` на место `newtype`, но не наоборот.

Различия

Тип, определяемый с помощью слова `newtype`, обязан иметь один и только один конструктор значения. Мы можем написать так:

```
newtype IPAddress = IP String
```

А вот так не можем:

```
newtype IPAddress = IP String | Localhost
```

Компилятор заупрямится:

```
A newtype must have exactly one constructor,
  but ‘IPAddress’ has two
In the newtype declaration for ‘IPAddress’
```

Кроме того, в таком типе должно быть одно и лишь одно поле. То есть можно так:

```
newtype IPAddress = IP String
```

Или же так, с меткой:

```
newtype IPAddress = IP { value :: String }
```

А вот два или более полей запихнуть не удастся:

```
newtype EndPoint = EndPoint String Int
```

Компилятор вновь обратит наше внимание на проблему:

```
The constructor of a newtype must have exactly one field
  but 'EndPoint' has two
In the definition of data constructor 'EndPoint'
In the newtype declaration for 'EndPoint'
```

Более того, нульарный конструктор тоже не подойдёт:

```
newtype HardDay = Monday
```

И вновь ошибка:

```
The constructor of a newtype must have exactly one field
  but 'Monday' has none
```

Зачем он нужен?

В самом деле, зачем нам нужно такое хозяйство? Это нельзя, то нельзя. Какой смысл?

Смысл в оптимизации. Обратите внимание на модель `newtype`:

```
newtype IPAddress = IP           String

новый   название    конструктор  Поле
тип                 значения
```

Фактически, `newtype` берёт одно-единственное значение некоторого существующего типа и всего лишь оборачивает его в свой конструктор. Именно поэтому тип, введённый с помощью `newtype`, не относится к АТД, и с точки зрения компилятора он является лишь переименованием типа (англ. type renaming). Это делает такой тип более простым и эффективным с точки зрения представления в памяти, нежели тип, определяемый с `data`.

Когда мы пишем так:

```
data IPAddress = IP String
```

мы говорим компилятору: «`IPAddress` — это абсолютно новый и самобытный тип, которого никогда не было ранее». А когда пишем так:

```
newtype IPAddress = IP String
```

мы говорим: «`IPAddress` — это всего лишь обёртка для значения уже существующего типа `String`».

type vs newtype

Внимательный читатель спросит, в чём же фундаментальное отличие типов, вводимых с помощью `newtype`, от типов, вводимых с помощью `type`? Там синоним, тут — обёртка. Отличие вот в чём.

Когда мы пишем так:

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

мы объявляем: «Тип `String` — это эквивалентная замена типу `[Char]`». И поэтому везде, где в коде стоит `[Char]`, мы можем поставить `String`, и везде, где стоит `String`, мы можем поставить `[Char]`. Например, если функция объявлена так:

```
replace :: String
        -> String
        -> String
        -> String
```

мы можем спокойно переписать объявление:

```
replace :: [Char]
        -> [Char]
        -> [Char]
        -> [Char]
```

и ничего не изменится.

Когда же мы пишем так:

```
newtype MyInt = MyInt Int
```

мы объявляем: «Тип `MyInt` — это новый тип, представление которого такое же, как у типа `Int`». Мы не можем просто взять и поставить `MyInt` на место `Int`, потому что эти типы равны лишь с точки зрения представления в памяти, с точки зрения системы типов они абсолютно различны.

А зачем же нам нужно это? Для простоты и надёжности кода. Допустим, есть такая функция:

```
getBuildsInfo :: String -> Int -> BuildsInfo
getBuildsInfo projectName limit = ...
```

Эта функция запрашивает у CI-сервиса (через REST API) информацию о сборках проекта. Из определения мы видим, что первым аргументом выступает имя проекта, а вторым — количество сборок. Однако в месте применения функции это может быть не столь очевидным:

```
    let info = getBuildsInfo "ohaskell.guide" 4
```

Что такое первая строка? Что такое второе число? Неясно, нужно глядеть в определение, ведь даже объявление не расскажет нам правду:

```
getBuildsInfo :: String  -> Int    -> BuildsInfo

                 что за     что за
                 строка?    число?
```

Вот тут нам и помогают наши типы, ведь стандартные `String` и `Int` сами по себе не несут никакой полезной информации о своём содержимом. Конечно, мы могли бы обойтись и без типов, просто введя промежуточные выражения:

```
    let project = "ohaskell.guide"
        limit   = 4
        info    = getBuildsInfo project limit
```

Однако программист может этого и не сделать, и тогда мы получим «магические значения», смысл которых нам неизвестен. Куда лучше ввести собственные типы:

```
newtype Project = Project String
newtype Limit = Limit Int

getBuildsInfo :: Project -> Limit  -> BuildsInfo

                 уже не     уже не
                 просто     просто
                 строка     число
```

Это заставит нас писать явно:

```
    let info = getBuildsInfo (Project "ohaskell.guide")
                             (Limit 4)
```

Теперь, даже без промежуточных выражений, смысл строки и числа вполне очевиден. Это важный принцип в Haskell: безликие типы наподобие `String` или `Int` заменять на типы, имеющие конкретный смысл для нас.

Кроме того, `newtype`-типы помогают нам не допускать глупых ошибок. Например, есть другая функция:

```
getArtifacts :: String -> Int -> Int -> [Project]
getArtifacts projectName limit offset = ...
```

Мало того, что перед нами вновь безликие `Int`, так их ещё и два. И вот какая нелепая ошибка может нас поджидать:

```
    let project = "ohaskell.guide"
        limit   = 4
        offset  = 1
        info    = getArtifacts project offset limit
```

Заметили? Мы случайно перепутали аргументы местами, поставив `offset` на место `limit`. Работа функции при этом нарушится, однако компилятор останется нем как рыба, ведь с точки зрения системы типов ошибки не произошло: и там `Int`, и тут `Int`. Синонимы для `Int` также не помогли бы. Однако если у нас будут `newtype`-типы:

```
newtype Limit = Limit Int
newtype Offset = Offset Int
```

тогда подобная ошибка не пройдёт незамеченной:

```
    let project = "ohaskell.guide"
        limit   = Limit 4
        offset  = Offset 1
        info    = getArtifacts offset limit
```

Типы аргументов теперь разные, а значит, путаница между ними гарантированно прервёт компиляцию.

Вот такие они, `newtype`-типы. В последующих главах мы увидим ещё большую мощь системы типов Haskell.

Глава 23

Конструктор типа

В предыдущих главах мы познакомились с АТД, которые сами по себе уже весьма полезны. И всё же есть в них одно ограничение: они напрочь лишены гибкости. Вот тебе конкретные поля, а вот тебе конкретные типы, будь счастлив. Но существует способ наделить наши тип куда большей силой. Эта глава станет для нас переломной, ведь с неё начнётся наш путь в мир действительно мощных типов.

Опциональный тип

Допустим, у нас есть список пар следующего вида:

```
type Chapters = [(FilePath, String)]

chapters :: Chapters
chapters = [ ("/list.html",  "Список")
           , ("/tuple.html", "Кортеж")
           , ("/hof.html",   "ФВП")
           ]
```

Тип `FilePath` есть не более чем стандартный синоним для типа `String`, но он более информативен. Итак, этот список содержит названия трёх глав данной книги и пути к ним. И вот понадобилась нам функция, которая извлекает название главы по её пути:

```
lookupChapterNameBy :: FilePath -> Chapters -> String
lookupChapterNameBy _ [] = ""  -- Так ничего и не нашли...
lookupChapterNameBy path ((realPath, name) : others)
  | path == realPath = name -- Пути совпадают, вот вам имя.
  | otherwise        = lookupChapterNameBy path others
```

Всё предельно просто: рекурсивно бежим по списку пар `chapters`, на каждом шаге извлекая через паттерн матчинг путь ко главе и её имя. Сравниваем пути и, ежели совпадают — на выходе получается имя, соответствующее заданному пути. Если

же, пройдя весь список, мы так и не нашли соответствующего пути, на выходе будет пустая строка.

Используем так:

```
main :: IO ()
main = putStrLn $
  if | null name -> "No such chapter, sorry..."
     | otherwise -> "This is chapter name: " ++ name
  where
    name = lookupChapterNameBy "/tuple.html" chapters
```

Если на выходе функции `lookupChapterNameBy` пустая строка, значит мы ничего не нашли, в противном же случае показываем найденное имя.

Ну и как вам такое решение? Вроде бы красивое, но почему, собственно, пустая строка? Я вполне мог написать заготовку для очередной главы и ещё не дать ей имя:

```
chapters :: Chapters
chapters = [ ("/list.html",  "Список")
           , ("/tuple.html", "Кортеж")
           , ("/hof.html",   "ФВП")
           , ("/monad.html", "")  -- Заготовка
           ]
```

В этом случае наше решение ломается: пустая строка на выходе функции `lookupChapterNameBy` может означать теперь как то, что мы не нашли главы с таким путём, так и то, что глава-то существует, просто её имя пока не задано. Следовательно, нам нужен другой механизм проверки результата поиска, более однозначный.

Определим опциональный тип. Опциональным (англ. optional) называют такой тип, внутри которого либо есть нечто полезное, либо нет. Выглядеть он будет так:

```
data Optional = NoSuchChapter
              | Chapter String
```

Если значение типа `Optional` создано с помощью нульарного конструктора `NoSuchChapter`, это означает, что внутри ничего нет, перед нами значение-пустышка. Это и будет соответствовать тому случаю, когда нужную главу мы не нашли. А вот если значение было создано с помощью унарного конструктора `Chapter`, это несомненно будет означать то, что мы нашли интересующую нас главу. Перепишем функцию `lookupChapterNameBy`:

```
lookupChapterNameBy :: FilePath -> Chapters -> Optional
lookupChapterNameBy _ [] = NoSuchChapter -- Пустышка
lookupChapterNameBy path ((realPath, name) : others)
  | path == realPath = Chapter name      -- Реальное имя
  | otherwise        = lookupChapterNameBy path others
```

Код стал более понятным. И вот как мы будем работать с этой функцией:

```
main :: IO ()
main = putStrLn $
  case result of
    NoSuchChapter -> "No such chapter, sorry..."
    Chapter name  -> "This is chapter name: " ++ name
  where
    result = lookupChapterNameBy "/tuple.html" chapters
```

Отныне функция `lookupChapterNameBy` сигнализирует о неудачном поиске не посредством пустой строки, а посредством нульарного конструктора. Это и надёжнее, и читабельнее.

Красиво, но в этом элегантном решении всё-таки остаётся один изъян: оно намертво привязано к типу `String`:

```
data Optional = NoSuchChapter
              | Chapter String
```

Почему именно `String`?

В самом деле, почему? Например, в Haskell широкое применение получил тип `Text` из одноимённого пакета. Этот тип, кстати, значительно мощнее и эффективнее стандартной `String`. Значит, если мы захотим определить опциональный тип и для `Text`, придётся дублировать:

```
data Optional = NoSuchChapter | Chapter String

data Optional = NoSuchChapter | Chapter Text
```

Однако компилятор наотрез откажется принимать такой код:

```
Multiple declarations of ‘Optional’
```

Имена-то типов одинаковые! Хорошо, уточним:

```
data OptionalString = NoSuchChapter | Chapter String

data OptionalText   = NoSuchChapter | Chapter Text
```

Но и в этом случае компиляция не пройдёт:

```
Multiple declarations of ‘NoSuchChapter’

...

Multiple declarations of ‘Chapter’
```

Конструкторы значений тоже одноимённые, опять уточняем:

```
data OptionalString = NoSuchChapterString
                    | ChapterString String

data OptionalText   = NoSuchChapterText
                    | ChapterText Text
```

Вот теперь это работает, но код стал избыточным. А вдруг мы пожелаем добавить к двум строковым типам ещё и третий? Или четвёртый? Что ж нам, для каждого типа вот так вот уточнять? Нет, умный в гору не пойдёт, есть лучший путь.

Может быть

В стандартной библиотеке живёт тип по имени `Maybe`:

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

Тип `Maybe` (от англ. maybe, «может быть») нужен для создания тех самых опциональных значений. Впрочем, я выразился неточно, ведь, несмотря на ключевое слово `data`, `Maybe` — это не совсем тип, это конструктор типа (англ. type constructor). Данная концепция используется в Haskell чрезвычайно часто, и, как и большинство концепций в этом языке, она столь полезна потому, что очень проста.

Конструктор типа — это то, что создаёт новый тип (потенциально, бесконечное множество типов). Когда мы явно определяем тип, он прямолинеен и однозначен:

```
data Optional = NoSuchChapter | Chapter      String

     имя типа   нульарный       унарный      поле
                конструктор     конструктор  типа
                значения        значения     String
```

Когда же мы определяем конструктор типа, мы создаём концептуальный скелет для будущих типов. Взглянем ещё раз (к-тор — это конструктор, для краткости):

```
                _____________________________
               /                             'v

data Maybe  a        = Nothing   | Just      a

     к-тор  типовая    нульарный   унарный   поле
     типа   заглушка   к-тор       к-тор     типа
                       значения    значения  a
```

Здесь присутствует уже знакомая нам типовая заглушка `a`, она-то и делает `Maybe` конструктором типа. Как мы помним, на место типовой заглушки всегда встаёт какой-то тип. Перепишем функцию `lookupChapterNameBy` для работы с `Maybe`:

```
lookupChapterNameBy :: FilePath -> Chapters -> Maybe String
lookupChapterNameBy _ [] = Nothing  -- Пустышка
lookupChapterNameBy path ((realPath, name) : others)
  | path == realPath = Just name    -- Реальное имя
  | otherwise        = lookupChapterNameBy path others
```

Рассмотрим обновлённое объявление:

```
lookupChapterNameBy :: FilePath
                    -> Chapters -> Maybe String

                                   это тип такой,
                                   называется
                                   Maybe String
```

На выходе видим значение типа `Maybe String`. Этот тип был порождён конструктором `Maybe`, применённым к типу `String`. Стоп, я сказал «применённым»? Да, именно так: вы можете воспринимать конструктор типа как особую «функцию», назовём её «типовая функция». Нет, это не официальный термин из Haskell, это просто аналогия: обычная функция работает с данными, а типовая функция работает с типами. Сравните это:

```
length   [1, 2, 3] = 3

функция  данное    = другое данное
```

и это:

```
Maybe    String    = Maybe String

типовая  тип       = другой тип
функция
```

Применение конструктора типа к существующему типу порождает некий новый тип, и это очень мощная техника, используемая в Haskell почти на каждом шагу. Например, если нам нужно завернуть в опциональное значение уже не `String`, а ранее упомянутый `Text`, мы ничего не должны менять в конструкторе `Maybe`:

```
Maybe    Text = Maybe Text

типовая  тип  = другой тип
функция
```

Какой тип подставляем на место `a`, такой тип и станет опциональным. В этом и заключается красота конструкторов типов, ведь они дают нам колоссальный простор для творчества.

А теперь мы подошли к очень важной теме.

Этажи

Что такое тип `Maybe String`? Да, мы уже знаем, это АТД. Но что это такое по сути? Зачем мы конструируем сложные типы из простых? Я предлагаю вам аналогию, которая поможет нам взглянуть на этот вопрос несколько иначе. Эта аналогия отнюдь не аксиома, просто я нашёл её полезной для себя самого. Думаю, вам она тоже будет полезна. Конечно, предлагать аналогии — дело неблагодарное, ведь любая из них несовершенна и может быть так или иначе подвергнута критике. Поэтому не воспринимайте мою аналогию как единственно верную.

С точки зрения типов любую Haskell-программу можно сравнить с многоэтажным домом. И вот представьте, мы смотрим на этот дом со стороны.

На самом нижнем этаже расположены простейшие стандартные типы, такие как `Int`, `Double`, `Char` или список. Возьмём, например, тип `Int`. Что это такое? Целое число. Оно не несёт в себе никакого смысла, это всего лишь число в вакууме. Или вот строка — что она такое? Это просто набор каких-то символов в том же вакууме, и ничего более. И если бы мы были ограничены лишь этими типами, наша программистская жизнь была бы весьма грустной.

А вот на втором и последующих этажах живут типы куда более интересные. Например, на одном из этажей живёт тип `Maybe String`. При создании типа `Maybe String` происходит важное событие: мы поднимаемся с первого на более высокий этаж. Считайте эти этажи уровнями абстракции. Если тип `String` — это всего лишь безликая строка, то тип `Maybe String` — это уже не просто строка, это опциональная строка, или, если хотите, строка, наделённая опциональностью. Подняться на тот или иной этаж в нашем типовом небоскрёбе — это значит взять более простой тип и наделить его новым смыслом, новыми возможностями.

Или вот вспомним тип `IPAddress`:

```
data IPAddress = IPAddress String
```

Мы опять-таки взяли ничего не значащую строку и подняли её на этаж под названием `IPAddress`, и теперь это уже не просто какая-то строка, это IP-адрес. Новый тип наделил бессмысленную строку вполне определённым смыслом. А когда мы вытаскиваем внутреннюю строку из `IPAddress` с помощью паттерн матчинга, мы вновь оказываемся на первом этаже.

А вот ещё наш тип, `EndPoint`:

```
data EndPoint = EndPoint IPAddress Int
```

Тут мы поднялись ещё чуток: сначала подняли строку на этаж IP-адреса, а затем взяли его и тип `Int` и подняли их на следующий этаж под названием `EndPoint`, и на этом этаже перед нами уже не просто какой-то IP-адрес и какое-то число, перед нами уже связанные друг с другом адрес и порт.

А вот ещё один пример, знакомство с которым я откладывал до сих пор. Вспомним определение главной функции `main`:

```
main :: IO ()
```

Я обещал рассказать о том, что такое `IO`, и вот теперь рассказываю: `IO` — это тоже конструктор типа. Правда, конструктор особенный, непохожий на наши `IPAddress` или `EndPoint`, но об этом подробнее в следующих главах. Так вот поднявшись на этаж под названием `IO`, мы получаем очень важную способность — способность взаимодействовать с внешним миром: файл прочесть, на консоль текст вывести, и в том же духе. И потому тип `IO String` — это уже не просто невесть откуда взявшаяся строка, но строка, полученная из внешнего мира (например, из файла). И единственная возможность наделить наши функции способностью взаимодействовать с внешним миром — поднять (ну или опустить) их на `IO`-этаж. Вот так и получается: в процессе работы программы мы постоянно прыгаем в лифт и переезжаем с одного типового этажа на другой.

Но запомните: не все этажи одинаковы! Не со всякого этажа можно напрямую попасть на любой другой. Более того, есть такие этажи, оказавшись на котором, мы в конечном итоге обязаны на него и вернуться. Понимаю, сейчас это порождает больше вопросов, нежели ответов, но не беспокойтесь: ответы ждут нас в последующих главах.

Глава 24

Продолжение следует…

Работа над книгой идёт полным ходом, вас ждёт ещё много интересного! Следите за новостями об обновлениях в нашем чате и в моём Твиттере.