Денис Шевченко

О Haskell *по-человечески*

для обыкновенных программистов

О Haskell по-человечески

для обыкновенных программистов

издание 0.1

март 2014

Формат 70x100/16. Тираж ∞ экз.

Официальный сайт книги: http://ohaskell.ru

Содержание

Часть 4 **О списках**

Часть 5 **Пользовательские типы**

Часть 0 *Лирическое вступление*

Кто

«Я, барон Мюнхгаузен, обыкновенный человек…»

барон

Я, подобно Мюнхгаузену, обыкновенный программист. Самый заурядный самоучка. Когда-то я считал программирование самым скучным видом человеческой деятельности. Последние семь лет я считаю его одним из наиболее интересных дел.

Фредерик Брукс был прав, серебряной пули не существует. И всё же программисты-практики ищут новые инструменты для решения своих непростых задач. Вот и я, после 7 лет опыта программирования на C++, решил искать нечто новое, и именно ради стремления к большей эффективности.

Признаюсь, я начал уставать от сложности C++. Захотелось мне чего-нибудь эдакого. Компилируемость, строгость к типам, высокоуровневые конструкции, красивый синтаксис, универсальность и… что-нибудь попроще. Да, я хотел именно этого. Под руку случайно подвернулся Haskell — и зацепил меня сразу.

Во-первых, отсутствие оператора присваивания. Признаюсь, крышу мне снесло напрочь, и я решил разобраться.

Во-вторых, красота. Я люблю красивый код, а, как выяснилось, код на Haskell может быть очень красивым.

В-третьих, мощь. Продолжайте читать, и чуть позже вы сами в этом убедитесь.

И наконец, простота. Нет, я не оговорился. Мне известно, что к функциональному программированию эпитет «простое» применяется чуть реже, чем никогда. В частности, в отношении Haskell бытует мнение о чрезвычайной, прямо-таки фантастической сложности его освоения. И всё-таки я повторю: в этом языке меня поразила его простота. Но простота эта особенная, и скоро вы поймёте, что я имею в виду.

Почему

А в самом деле, почему? С чего это я решил написать ещё одну книгу о Haskell?

Причина первая: меня достало! Достало, что почти все известные мне руководства по Haskell начинаются с демонстрации того, как реализовать алгоритм быстрой сортировки. И ещё что-то там про факториал и числа Фибоначчи. Мне за все годы практики ни разу не приходилось реализовывать алгоритм быстрой сортировки. Поэтому я даже не знаю, что это такое.

Исторически сложилось так, что большинство из нас начали свой профессиональный путь именно с императивных языков. И вот вместо того, чтобы показать нам красоту функциональных языков в свете их реального применения, нас тыкают носом в числа Фибоначчи и в почти нами забытую математическую нотацию... Естественно, читая подобные материалы, обычный программист начинает чувствовать себя дебилом, и это чувство отбивает в нём всякую охоту осваивать эту *непонятную функциональщину*.

Именно поэтому я расскажу о Haskell нормальным, человеческим языком, с минимумом академизма и действительно понятными примерами.

Есть и вторая причина. Все известные мне книги по Haskell слишком объёмные. В них много лишнего. А у нас, программистов-практиков, не так много свободного времени, чтобы проглатывать очередной талмудоподобный труд в 500 страниц. Именно поэтому я расскажу о Haskell по возможности лаконично.

Зачем

Функциональное программирование — это своеобразное гетто посреди мирового мегаполиса программной разработки. Доля функциональных языков на рынке очень мала, а программистов, использующих эти языки, считают либо недосягаемой элитой, либо асоциальными идиотами. Цель данной книги — разрушить такое представление.

В частности, я докажу ложность двух представлений о языке Haskell, а именно а) представление о колоссальной сложности его освоения и б) убеждение в том, что этот язык пригоден исключительно для научных лабораторий MIT[1].

Да, *в прошлом* оба эти представления соответствовали действительности. Haskell официально существует с 1990 года, однако его выход в «широкий свет» начался лишь в 2003. Таким образом, в течение 13 лет этот язык действительно был уделом лабораторий, и изучить его было нелегко, поскольку вся имеющаяся на тот момент документация по нему была напичкана математикой. Тогда язык был медленным. Тогда было мало библиотек. Однако то, что было актуально *тогда*, уже неактуально *сегодня*.

И ещё об ожиданиях. Не ждите от этой книги всеохватной полноты рассмотрения Haskell и его экосистемы. Кроме того, это не справочник. Я не буду копировать сюда всё содержимое официального сайта Haskell[2] или переводить на русский язык стандарт Haskell 2010[3]. Да и к тому же господин Google знает о Haskell значительно больше, чем я.

Цель этой книги — протянуть новичкам руку помощи в самом начале их пути.

1 Massachusetts Institute of Technology
2 `http://www.haskell.org/haskellwiki/Haskell`
3 `http://www.haskell.org/onlinereport/haskell2010`

Для кого

Если вы дочитали до этого места — значит эта книга для вас. И не беспокойтесь об уровне своей квалификации: если вы *уже* знаете, что такое компилятор, зачем нужны пользовательские типы и чем объявление функции отличается от её определения — смело продолжайте читать.

Признаюсь вам: на момент написания этой книги я ещё не имел опыта разработки на Haskell и даже не завершил изучение этого языка. Многие удивятся: как же может человек, не имеющий весомого опыта и глубоких знаний в области функционального программирования, браться за написание книги о Haskell?!

Главное препятствие на пути популяризации этого языка (равно как и функционального программирования в целом) заключается в том, что рассказывающие о нём люди зачастую слишком далеки от обыкновенных разработчиков и от обыкновенных задач, решаемых этими разработчиками. И многие из нас, читая какой-нибудь труд, написанный аспирантом МФТИ, часто ловят себя на мысли, мол, куда уж мне до его мозгов...

Именно поэтому автор этой книги — самый обыкновенный программист. Я рассматриваю Haskell не как объект научного исследования, а как инструмент для решения моих повседневных задач. Таких как я — большинство. И если я смог ухватить суть этой функциональщины — значит и вы сможете.

Возможно, вы влюбитесь в этот язык. Возможно, он вызовет у вас отвращение. Могу обещать одно: скучно не будет.

Начнём.

Часть 1 Готовимся к работе

Создаём проект

Мы не можем начать изучение языка без полигона. Поэтому скачайте и установите Haskell Platform[1].

В состав Haskell Platform входит два важнейших компонента, о которых вам нужно знать:

1. `ghc`, компилятор Haskell (**G**lasgow **H**askell **C**ompiler);
2. `ghci`, интерпретатор Haskell.

Запомнили? А теперь можете забыть. Особенно про интерпретатор. Ведь вы планируете использовать Haskell в реальной работе, а это значит, все ваши проекты будут компилироваться. Однако и непосредственное использование компилятора `ghc` вам тоже едва ли понадобится.

В реальной работе вы не будете создавать файлик `Main.hs`[2] на рабочем столе для последующего скармливания его компилятору. Напротив, вы создадите нормальные рабочие проекты с логичной внутренней структурой. Так давайте и создадим такой с самого начала. А поможет нам в этом удобная утилита из Haskell Platform с необычным названием `cabal`.

Утилита `cabal`[3] предназначена для сборки проектов. Уверен, вы слышали о вещах типа `cmake` или `qmake`, так вот воспринимайте `cabal` как «`cmake` специально для Haskell».

Начнём творить. Разумеется, все описываемые ниже действия подразумевают вашу крепкую дружбу с командной строкой. Я буду приводить Unix-овые команды, если же вы используете Windows — адаптируйте примеры под себя.

Готовим структуру

Открываем терминал и творим:

```
mkdir Real
cd Real/
mkdir src
cd src/
touch Main.hs
```

1 `http://www.haskell.org/platform`
2 `.hs` — стандартное расширение исходников на Haskell
3 Аббревиатура от **c**ommon **a**rchitecture for **b**uilding **a**pplications and **l**ibraries.

```
mkdir Utils
cd Utils/
touch Helpers.hs
```

Итак, у нас есть каталог `Real` с привычной структурой:

```
.
└── src
    ├── Main.hs
    └── Utils
        └── Helpers.hs
```

Есть корневой каталог `src`, внутри которой лежат все наши исходники, сгруппированные по логическим признакам.

Кстати, об именах. Вам, вероятно, интересно, почему имена файлов и каталогов внутри каталога `src` начинаются с большой буквы? Чуть позже я объясню причину. А пока откроем файл `Main.hs` и напишем в нём:

```
main = putStrLn "Hi, haskeller!"
```

Закрываем, возвращаемся в корень проекта.

Настраиваем

Выполняем команду:

```
cabal init
```

Мы попадём в интерактивный диалог, в ходе которого нам будет предложено ответить на несколько вопросов о нашем проекте. В конце этого диалога будут автоматически созданы файлы проекта, и наш каталог приобретёт следующее содержимое:

```
.
├── Real.cabal
├── Setup.hs
└── src
    ├── Main.hs
    └── Utils
        └── Helpers.hs
```

Кстати, вы, вероятно, увидите вот такое предупреждение:

```
Generating LICENSE...
Warning: unknown license type, you must put a copy in LICENSE yourself.
```

Не беспокойтесь. Я выбрал вариант без лицензии, однако `cabal` почему-то не пропустила шаг создания оной молча, а вместо этого недовольно буркнула. Простим ей это, но, для поддержания классического вида нашего проекта, добавим файл `LICENSE` вручную.

Как уже было упомянуто, в корневом каталоге нашего проекта появились два новых файла, `Real.cabal` и `Setup.hs`. Второй файл нам не так интересен[4], а вот первый — это и есть сборочный файл нашего проекта. Откроем его:

```
-- Initial Real.cabal generated by cabal init.  For further documentation,
-- see http://haskell.org/cabal/users-guide/
name:                Real
version:             0.1.0.0
synopsis:            Real project in Haskell
-- description:
-- license:
license-file:        LICENSE
author:              Denis Shevchenko
maintainer:          me@dshevchenko.biz
-- copyright:
-- category:
build-type:          Simple
cabal-version:       >=1.8

executable Real
  -- main-is:
  -- other-modules:
  build-depends:       base ==4.6.*
  hs-source-dirs:      src
```

Здесь уже сохранены те самые значения, которые мы вводили в процессе вышеупомянутого диалога. Однако собрать проект прямо сейчас мы не сможем, потому что строка:

```
  -- main-is:
```

4 В соответствии с официальной документацией, трогать файл `Setup.hs` вам придётся крайне редко.

закомментирована. В этом файле принят синтаксис, подобный синтаксису Haskell, и поэтому однострочные комментарии здесь, как и в программном коде, начинаются с двух минусов подряд.

Нам необходимо раскомментировать эту строку и прописать в ней имя файла `Main.hs`, содержащего функцию `main`[5]:

```
  main-is: Main.hs
```

Конфигурируем

Выполняем:

```
cabal configure
```

В результате произойдёт подготовка проекта к сборке. Но прежде чем перейти к этой самой сборке, обращаю ваше внимание на последнюю часть файла `Real.cabal`:

```
executable Real
  main-is:             Main.hs
  -- other-modules:
  build-depends:       base ==4.6.*
  hs-source-dirs:      src
```

Видите отступ в два пробела перед четырьмя последними строчками? Оказывается, этот отступ необходим, и без него проект не соберётся. Кроме того, отступ этот должен быть не менее двух пробелов. Я рекомендую четыре, для красоты.

И ещё одна деталь. Это необязательно, но лишним не будет. Допишем в секцию `executable Real` ещё одну строку:

```
  ghc-options:         -W
```

Флаг `-W` вежливо попросит компилятор `ghc` показывать все основные предупреждения при компиляции. Не пренебрегайте этой возможностью.

5 Функция `main` — главная функция приложения, подобно `int main()` в языке C.

Собираем

Выполняем:

```
$ cabal build
Building Real-0.1.0.0...
Preprocessing executable 'Real' for Real-0.1.0.0...
[1 of 1] Compiling Main             ( src/Main.hs, dist/build/Real/Real-
tmp/Main.o )
Linking dist/build/Real/Real ...
```

Готово. В нашем каталоге появилось кое-что новенькое:

```
.
├── LICENSE
├── Real.cabal
├── Setup.hs
├── dist
│   ├── build
│   │   ├── Real
│   │   │   ├── Real  <<<--- Это и есть наш исполняемый файл.
...
```

Остальное содержимое каталога `dist` нас пока не интересует.

Запускаем

Пришло время запустить наше приложение. Находясь в корне проекта, выполняем:

```
./dist/build/Real/Real
```

Вывод будет таким:

```
Hi haskeller!
```

Вот и всё. Теперь вы знаете, как создавать, настраивать и собирать Haskell-проект. Вероятно, вас интересует, зачем мы создавали файл `Helpers.hs` в подкаталоге `Utils`? Какой в нём смысл, если он всё равно остался пустым? В следующей главе вы это узнаете.

О модулях, минимум

Настоящие проекты никогда не состоят из одного-единственного файла. Пришла пора узнать о модулях.

Исходные файлы в Haskell-проекте — это и есть модули. Один файл — один модуль. Таким образом, в нашем проекте сейчас есть два модуля: `Main.hs` и `Helpers.hs`.

В Haskell нет заголовочных файлов. Каждый из модулей рассматривается как самостоятельная единица проекта, содержащая в себе разные интересные вещи. И чтобы воспользоваться этими интересными вещами, нужно один модуль импортировать в другой.

Откроем наш пустой файл `Helpers.hs` и напишем в нём:

```
module Helpers where

hello user = "Hi, " ++ user
```

Первой строкой мы объявили, что имя этого модуля — `Helpers`. Далее, после ключевого слова `where`, мы наполнили модуль содержимым. Содержимое у нас предельно простое, но пока не спрашивайте меня, что такое `hello`. Скоро мы выясним это.

Импортируем

Откроем файл `Main.hs` и чуток изменим его:

```
import Helpers

main = putStrLn (hello "denis")
```

Мы включили наш модуль `Helpers` с помощью директивы `import`. Теперь можно воспользоваться содержимым этого модуля, а именно той самой штуковиной по имени `hello`.

Упоминаем

Теперь упомянем модуль `Helpers` в сборочном файле `Real.cabal`. Открываем его и прописываем наш модуль:

```
executable Real
    main-is:             Main.hs
    other-modules:       Helpers
    build-depends:       base ==4.6.*
    hs-source-dirs:      src
```

Мы раскомментировали строку `other-modules` и указали имя нашего модуля. Обращаю ваше внимание: указать нужно не имя файла, а имя модуля.

Но раз уж мы указали имя нашего модуля, необходимо указать и место, где его искать. Ведь он лежит не в каталоге `src`, а в подкаталоге `src/Utils`. Поэтому в сборочном файле ищем параметр `hs-source-dirs` и дописываем:

```
    hs-source-dirs:      src
                         src/Utils
```

Сохраняем, собираем:

```
Building Real-0.1.0.0...
Preprocessing executable 'Real' for Real-0.1.0.0...
[1 of 2] Compiling Helpers          ( src/Utils/Helpers.hs,
dist/build/Real/Real-tmp/Helpers.o )
[2 of 2] Compiling Main             ( src/Main.hs, dist/build/Real/Real-
tmp/Main.o )
Linking dist/build/Real/Real ...
```

Получилось — уже не один, а два модуля были скомпилированы. Теперь запускаем:

```
$ ./dist/build/Real/Real
Hi, denis
```

Работает.

Об именах

Здесь есть два правила.

Во-первых, имя модуля должно начинаться с большой буквы. В отношении имён подпапок внутри `src` принята та же практика.

Во-вторых, имя модуля должно совпадать с именем соответствующего ему файла. Именно поэтому файл, содержащий модуль `Helpers`, назван `Helpers.hs`.

Вот и всё. Теперь вы знаете, как организовать настоящий Haskell-проект. Позже я расскажу о модулях кое-что ещё, но на данный момент вам необходимо знать лишь это.

О Hackage

Если вы работали с Linux, вам знакомо понятие «репозиторий»: эдакое централизованное место, откуда можно взять много разных вкусностей. Так вот Hackage — это главный репозиторий в мире Haskell.

Название происходит от слияния слов **H**askell и p**ackage**. Существует он с 2008 года, и представляет собой большую-пребольшую кучу пакетов. Воспринимайте пакет как библиотеку, однако в мире Haskell закрепилось понятие «пакет» (package).

Среди этой кучи вы найдёте очень много готовых решений, как для стандартных задач, так и для узкоспециализированных.

Чтобы воспользоваться пакетом, нам необходимо сделать четыре шага:

1. найти этот пакет,
2. установить его,
3. добавить его в наш проект,
4. импортировать из него нужные нам модули.

Ищем

Рекомендую искать пакеты здесь:

1. Hoogle[1]
2. Hayoo![2]

Вбиваем в строке поиска нужное вам название, или категорию, или некое ассоциативное слово — и получаем много интересных результатов.

Для примера установим пакет `text`, продвинутый пакет для работы с... текстом, очевидно.

Устанавливаем

Существует инструмент для удобной установки пакетов из Hackage, и имя ему `cabal`. Да-да, та самая, уже знакомая нам утилита!

Переходим в корень нашего проекта и выполняем команду:

1 `http://www.haskell.org/hoogle`
2 `http://holumbus.fh-wedel.de/hayoo/hayoo.html`

```
cabal update
```

Этой командной мы обновляем список всех доступных пакетов. Рекомендуется периодически выполнять эту команду, чтобы всегда быть «на острие» развития Hackage.

После обновления списка устанавливаем наш пакет:

```
cabal install text
```

Чуток терпения — и пакет установлен.

Добавляем в проект

Открываем сборочный файл `Real.cabal` и прописываем в нём имя установленного пакета. Для этого находим параметр `build-depends` и через запятую дописываем имя пакета:

```
build-depends:   base == 4.6.*, text
```

И последний шаг.

Импортируем модули

Пакет состоит из модулей (а модули, как вы уже знаете, это файлы исходного кода). В пакете `text` модулей весьма много, мы выберем самый первый по счёту, `Data.Text`. Открываем `Main.hs` и пишем в самом начале:

```
import Data.Text
```

Готово. Теперь мы можем использовать разные вкусные вещи из этого модуля. А вот какие именно вещи и как их использовать — об этом вы узнаете в ближайшем будущем.

О прелюдии

Есть один стандартный модуль, который по умолчанию импортируется во все ваши модули. Имя ему — `Prelude`. В нём содержатся самые базовые Haskell-инструменты, многие из которых вы будете использовать постоянно.

Часть 2 Несколько слов о Haskell

Чистая функциональность

Haskell — чисто функциональный язык программирования общего назначения.

Исторически сложилось так, что наиболее популярным ныне подходом к написанию программ является *императивный* подход (от английского imperative, приказание). При таком подходе программа представляет собой набор инструкций, которые должны быть выполнены строго в том порядке, в котором эти инструкции указаны. Кроме этого, императивное программирование подразумевает наличие оператора присваивания, потому что программист часто меняет состояние множества переменных.

Однако существует принципиально иной подход к написанию программ, а именно *декларативный* (от английского declarative, описание), при котором программа представляет собой набор описаний того, что же она должна в итоге сделать. Функциональное программирование является одним из воплощений декларативного подхода. При таком подходе порядок выполнения инструкций зачастую неважен. Более того, в Haskell нет оператора присваивания, и все переменные в нём вовсе не переменные, а самые что ни на есть *постоянные*.

И чтобы окончательно сбить вас с толку, упомяну такие свойства Haskell, как:

1. наличие чистых функций,
2. разграничение чистых функций от функций с побочными эффектами,
3. ленивость вычислений.

Звучит весьма странно, поэтому прямо сейчас мы начнём разбираться.

Три кита типизации

К типам у Haskell отношение очень серьёзное. Его система типов зиждется на трёх китах:

1. статическая проверка,
2. строгость,
3. автоматическое выведение.

Кит первый

Статическая проверка типов — это проверка типа каждого выражения, выполняемая на стадии компиляции. И если компилятору что-то не понравится в типе какого-либо выражения, компиляция будет прервана с ошибкой.

Соответственно, если компиляция кода на Haskell прошла успешно, мы можем утверждать, что с типами у нас всё в порядке, потому что у нас есть второй кит.

Кит второй

Строгость типов — это требование соответствия того, что мы ожидаем, тому, что мы получаем.

Например, в языке C мы можем написать такую функцию:

```
int coefficient() {
    return 12.9;
}
```

Это пример неявного приведения типов. Мы ожидаем значение типа `int`, но фактически получаем значение типа `double`. Однако компилятор языка C спокойно проглотит это, при этом аккуратно отбросив дробную часть возвращаемого значения, ведь тип этого значения будет незримо приведён к `int`.

В Haskell подобный код не имеет ни малейших шансов пройти компиляцию, потому что в этом языке не существует неявного приведения типов: если мы ожидаем целое число — будь добр предоставить именно целое число.

Впрочем, явное приведение типов в Haskell тоже очень ограничено. В том же C++ мы можем написать так:

```
int main() {
    std::cout << (int)'1' << std::endl;
}
```

Взяли значение типа `char` — и грубо переделали его в значение типа `int`. Компилятор — молчок. Последствия такого рода ошибок уже стали притчею во языцех.

В Haskell мы *можем* явно указать тип некоторого значения, но только если этот тип ассоциативен со значением. То есть если это число `1`, мы можем явно указать лишь «числовой» тип (такой, как `Integer` или `Double`). А вот фокусы с приведением символа к целочисленному значению, как это было продемонстрировано выше, в Haskell совершенно невозможны.

Кит третий

Автоматическое выведение типов — это способность компилятора понять тип выражения по самому этому выражению.

Например, в языке C мы обязаны указывать тип явно:

```
int i = 10;
```

В Haskell этого делать не нужно. Мы просто пишем:

```
i = 10
```

Компилятор проанализирует значение `10` и сам поймёт, что тип `i` — это `Integer`. Впрочем, как уже было сказано, мы можем указать тип выражения явно (а иногда *должны* это сделать). Вскоре я продемонстрирую это.

Неизменность данных

Одним из фундаментальных свойств Haskell языка является отсутствие оператора присваивания.

Это именно то свойство, услышав о котором впервые, я не поверил своим ушам. Каким образом можно программировать без оператора присваивания? А как же мы будем изменять состояние наших переменных? Моё удивление можно было понять: в процессе написания кода на C++ я часто использую оператор присваивания.

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: «Возьми совокупность байтов, соответствующую значению `123`, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого.» Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: «Значение `a` равно `123`.»

Вы спросите, в чём разница? Ведь мы в любом случае получаем переменную `a` со значением `123`. А разница в том, что присваивание может происходить множество раз в отношении одной и той же переменной, в то время как объявление равенства может быть указано только единожды. Поэтому если мы объявили, что значение `a` равно `123`, то так оно и будет, раз и навсегда. Именно поэтому в языке Haskell нет ни понятия «переменная», ни ключевого слова `const`, ведь все значения в нём константны по своей сути.

Вероятно, вас интересует, как же мы сможем добавить элемент в какую-нибудь коллекцию, если у нас всё константное? Ответ: никак. Мы не можем изменить значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена[1].

Вскоре вы увидите, что можно прекрасно жить без оператора присваивания.

1 В Haskell есть сборщик мусора (garbage collector).

Лень

Язык Haskell — ленивый. Это означает, что он никогда не сделает работу, результат которой никому не нужен.

Начнём с C++

Допустим, нам нужен список из некоторого числа одинаковых IP-адресов. Да, в реальной жизни нам такое едва ли понадобится, но этот пример хорошо покажет нам суть ленивых вычислений.

Функция, возвращающая список адресов, на C++ может выглядеть так:

```
typedef std::vector<std::string> IP_addresses;

IP_addresses generate_addresses( size_t how_many,
                                 const std::string& address ) {
    const IP_addresses addresses( how_many, address );
    return addresses;
}
```

Теперь нам понадобилась функция, получающая заданное количество адресов из этого списка и выводящая их на экран. Например:

```
void take_and_print( size_t how_many,
                     const IP_addresses& addresses ) {
    for( size_t i = 0; i < how_many; ++i ) {
        std::cout << addresses[i] << std::endl;
    }
}

int main() {
    take_and_print( 2, generate_addresses( 100, "127.0.0.1" ) );
}
```

Функция `take_and_print` получает список, возвращённый нашей функцией `generate_addresses`, а потом печатает первые два адреса из этого списка.

Вывод будет таким:

```
127.0.0.1
127.0.0.1
```

И всё бы хорошо, но из 100 созданных строк фактически потребовались лишь первые две. Оставшиеся 98 строк были созданы абсолютно напрасно. Было затрачено время, была затрачена память — и всё впустую.

Это — следствие строгости вычислений, присущей языку C++. Функция `generate_addresses` прямолинейна и сразу рвётся в бой. Сказали ей создать 100 адресов — получите 100. Скажут создать миллион — пожалуйста, вот вам миллион. Скажут миллиард — ну что ж, потерпите чуток, но будет вам и миллиард.

Тем временем функция `take_and_print` столь же прямолинейна, и ей абсолютно наплевать на усилия трудолюбивой функции `generate_addresses`. Если ей сказали отобразить лишь первые два элемента полученного контейнера, именно это она и сделает. И ей без разницы, сколько там ещё осталось элементов, десять или полмиллиарда.

Результатом строгости вычислений является лишняя работа. Но функции в Haskell, в отличие от своих трудолюбивых коллег из C++, терпеть не могут лишней работы.

А вот как в Haskell

Откроем наш файл `Main.hs` и перепишем его:

```
main = print (take 2 (replicate 100 "127.0.0.1"))
```

Функция `replicate` создаёт список из 100 адресов вида `127.0.0.1`, а функция `take` берёт 2 первых адреса из этого списка (о чём свидетельствует число 2, переданное ей в качестве первого аргумента). Функция `print` приводит это хозяйство к строковому виду и выводит на экран. Не обращайте внимания на синтаксические непонятности этого кода. В последующих главах они будут разъяснены в высшей степени подробно.

Весь фокус в том, что функция `replicate` создаёт список вовсе не из 100 адресов, а всего из двух. Почему? Потому что именно столько понадобилось функции `take`.

Функция `replicate` — лентяйка. Несмотря на то, что мы попросили её создать список из 100 строк, она смотрит по сторонам и думает: «Так-с, кому тут нужны мои строки? Ага, функции `take` нужны. И сколько же ей нужно? А-а,

всего две. Ну так а чего я, глупая что ли, создавать сто строк, когда требуется всего две?! Вот тебе две строки и будь счастлива!»

Да, трудолюбие — это хорошо, а лень — это плохо, однако в данном случае мне более симпатична функция-лентяйка. Она, как хороший рационализатор, делает не столько, сколько её попросили, а столько, сколько реально нужно. В этом и заключается суть ленивых вычислений в Haskell.

Разумеется, если аппетиты функции `take` возрастут и она попросит первые пятьдесят элементов вместо первых двух, то функция `replicate` создаст список уже из 50 строк. Столько, сколько нужно, и ни капли больше.

Да, но откуда мы можем знать, что функция `replicate` создаёт лишь столько IP-адресов, сколько потребовалось? А вдруг это не так? Давайте проверим.

Ленивость языка Haskell позволяет нам оперировать бесконечно большими списками. Нет, не просто очень большими, но именно бесконечными. Перепишем наш пример следующим образом:

```
main = print (take 2 (repeat "127.0.0.1"))
```

Функция `repeat` создаст бесконечно большой список IP-адресов, элементами которого будет переданный ей адрес `127.0.0.1`. И вот если бы наша трудолюбивая функция `generate_addresses` из C++ захотела стать похожей на свою ленивую коллегу, ей пришлось бы стать примерно такой:

```
IP_addresses generate_addresses( size_t how_many,
                                 const std::string& address ) {
    IP_addresses addresses;
    for(;;) {
        addresses.push_back( address );
    }
    return addresses;
}
```

И всё бы хорошо, но это намертво зависнет. И причиной тому служит уже известное нам трудолюбие функции `generate_addresses`. Сказали ей создать бесконечно большой список — будет создавать до последнего вздоха.

Однако если мы соберём наш Haskell-проект и запустим его, то не будет никакого зависания, и на экран вновь выведутся уже знакомые нам два адреса.

А всё потому, что функция `repeat` столь же ленива и рациональна, как и её коллега `replicate`. Да, мы попросили её создать бесконечно большой список, однако на деле она создаст список вовсе не бесконечно большой, а настолько большой, насколько потребуется. И если в данном случае потребовался список

только из двух строк — получите список из двух строк. Конечно, если бы потребовался список из миллиона строк — извольте, будет вам миллион.

Вот суть ленивых вычислений в Haskell: не важно, сколько приказали сделать, ведь в конечном итоге будет сделано ровно столько, сколько реально понадобится.

Часть 3 О функциях

Чистые функции

Раз уж Haskell — чисто функциональный язык программирования, поговорим о чистых функциях, как об одном из краеугольных камней этого языка.

Для начала вспомним школьный курс математики и сформулируем простейшее определение функции:

Функция — это описание зависимости чего-то от чего-то.

Так вот чистые функции в Haskell — это и есть функции в математическом смысле. Они представляют собой описание того, как входное значение определяет выходное значение.

Отсюда вытекает важнейшая характеристика чистых функций, а именно *отсутствие побочных эффектов*. Значение на входе чистой функции всецело и полностью определяет значение на выходе. Поэтому если мы миллион раз подадим на вход одно и то же значение, то на выходе мы миллион раз гарантированно получим один и тот же результат.

Объявляем

Как и во многих других языках программирования, функцию сначала нужно объявить. Сделаем же это:

```
simple_sum :: Int -> Int
```

До символа `::` указывается имя функции, а после — тип.

```
simple_sum :: Int -> Int
|        |    |        |
   имя            тип
```

Я понимаю, словосочетание «тип функции» звучит странно, но чистая функция — это значение, имеющее тип функции. Рассмотрим описание этого типа:

```
Int -> Int
```

Обратите внимание на стрелочку. Именно эта стрелочка и говорит нам о том, что перед нами — чистая функция. Слева от неё указан тип единственного аргумента (в данном случае это стандартный тип `Int`), а справа от неё — тип выходного значения (тот же `Int`). Саму же стрелочку можно воспринимать как «ментальное указание» на поток информации, движущийся через функцию: от её входа к выходу, слева направо.

Напоминаю, что чистая функция обязана иметь хотя бы один аргумент и обязана что-то возвращать, ведь это и отражает суть математической функции: что-то обязательно подаём на вход и что-то обязательно получаем на выходе.

Кстати, о количестве аргументов. Разумеется, чистая функция может принимать и несколько аргументов. Вот тип функции, принимающей три аргумента:

```
Int -> Int -> Int -> Int
```

Читать эту запись следует так: ищем последнюю по счёту (самую правую) стрелочку — она-то и будет тем самым разделителем: слева от неё идёт список типов аргументов, справа — тип возвращаемого выражения:

```
Int -> Int -> Int -> Int
типы аргументов  |  |тип возвращаемого значения
```

Определяем

Теперь функцию необходимо определить. Кстати, определить нужно *обязательно*. Например, в языке C или C++ мы можем спокойно объявить функцию и не определять её (при условии, что она никогда не вызывается). В Haskell более строгий подход: если объявил функцию — будь добр и определить её, в противном случае компилятор выскажет своё категорическое недовольство.

Поэтому сразу же после объявления пишем определение:

```
simple_sum :: Int -> Int
simple_sum value = value + value
```

Здесь «ментальным разделителем» является знак равенства. Скелет данного выражения можно представить так:

```
NAME ARGUMENTS = BODY_EXPRESSION
```

где `NAME` — имя функции, `ARGUMENTS` — список имён аргументов (имён, а не типов), а `BODY_EXPRESSION` — тело функции. В данном случае у нас имеется

один-единственный аргумент по имени `value`, а также имеется простое тело, в котором мы просто складываем аргумент с самим собой.

Вызываем

Теперь нашу функцию можно вызывать. Сделаем же это с аргументом 4, или, как принято говорить в мире ФП, *применим* нашу функцию к аргументу 4:

```
main = putStrLn (show (simple_sum 4))
```

Результат:

```
8
```

Готово. А теперь необходимо уточнить некоторые важные детали.

Выход из функции

В языке C, если у нас есть функция с возвращаемым значением, мы обязаны где-то в её теле указать точку выхода с помощью инструкции `return`. Кроме того, точек выхода может быть несколько.

В Haskell всё обстоит совершенно иначе. Во-первых, точка выхода из чистой функции может быть только одна, а во-вторых, аналога инструкции `return` в Haskell нет. И если мы вспомним математическую природу чистой функции, то поймём, что иначе и быть не может. Ведь чистая функция представляет собой описание зависимости выходного значения от входных значений, поэтому её тело представляет собой совокупность выражений, которые вычисляются и в конечном итоге оставляют одно-единственное, последнее выражение. Так вот это последнее выражение и будет являться «точкой выхода» из функции.

Приведу пример:

```
indicate :: String -> String
indicate address =
    if address == "127.0.0.1" then "localhost" else address
```

Эта функция принимает единственный аргумент стандартного типа `String`, соответствующий некоторому IP-адресу. В теле функции происходит проверка аргумента на равенство адресу `127.0.0.1`, в результате чего мы окажемся в одной из двух логических ветвей. В C++ это выглядело бы так:

```
std::string indicate( const std::string& address ) {
    if( address == "127.0.0.1" ) {
        return "localhost";
    }
    return address;
}
```

Мы явно указали две точки выхода из функции. Но в Haskell этого делать не нужно, потому что когда мы окажемся в одной из двух логических ветвей, то выражение, на котором мы окажемся, и будет возвращено.

Чтобы стало понятнее, перепишем тело этой функции так, чтобы избавиться от выражения `if-then-else`:

```
indicate :: String -> String
indicate "127.0.0.1" = "localhost"
indicate address = address
```

Haskell позволяет вводить несколько определений для одной функции. Рассматривайте это как особый вариант перегрузки. Здесь мы говорим: «Если входной аргумент будет равен `127.0.0.1`, пусть будет использовано тело №1, в противном случае пусть будет использовано тело №2.» Следовательно, когда компилятор увидит вызов этой функции в коде, он просто подставит на место этого вызова соответствующее выражение: либо строку `localhost`, в случае использования первого тела, либо фактически переданный аргумент, в случае использования второго тела.

Теперь всё встало на свои места: явно определять точку выхода из чистой функции не нужно потому, что конечное выражение в теле этой функции просто заменит собою вызов функции. То есть если написано так:

```
main = putStrLn (indicate "127.0.0.1")
```

то это то же самое, как если бы было написано просто:

```
main = putStrLn "localhost"
```

Это — важное свойство чистых функций: мы всегда можем безопасно заменить места их вызова соответствующими возвращёнными значениями, и работа приложения при этом останется неизменной. Именно поэтому работать с чистой функцией легко.

Стража!

Существует ещё один способ задать выбор внутри функции без использования `if-then-else`. Называется он стража (guard), хотя можно перевести и как «защита» или «охрана». Перепишем нашу функцию:

```
indicate :: String -> String
indicate address
    | address == "127.0.0.1" = "localhost"
    | null address = "empty IP-address"
    | otherwise = address
```

Символ `'|'` отражает выбор, как если бы мы написали вместо него слово «либо». После него идёт логическое условие и соответствующее ему итоговое значение функции:

```
    | address == "127.0.0.1" = "localhost"
    | null address           = "empty IP-address"

        логическое условие   = итоговое значение
```

Кстати, ветку `otherwise` необходимо использовать всегда. Если вы её пропустите, код пройдёт компиляцию, однако в вашем коде поселится коварная ошибка. В частности, если вы напишете так:

```
indicate :: String -> String
indicate address
    | address == "127.0.0.1" = "localhost"
    | null address = "empty IP-address"
```

а потом примените эту функцию к непустой строке, отличающейся от `"127.0.0.1"`, вы получите ошибку времени выполнения:

```
Real: src/Main.hs:(23,1)-(25,36): Non-exhaustive patterns in function
indicate
```

Будьте внимательны.

Локальные выражения

Локальное выражение в теле функции — штука очень полезная, спасающая нас от магических чисел и от дубляжа.

Например, у нас есть такая функция:

```
prepare_length :: Double -> Double
prepare_length line =
    line * 0.4959
```

Здесь мы готовим длину некой линии путём умножения её первоначальной длины на заданный поправочный коэффициент. Но перед нами — классическое магическое число, смысл которого непонятен, и это плохо. Добавлять комментарий — не самое лучшее решение. Поэтому добавим локальное поясняющее выражение:

```
prepare_length :: Double -> Double
prepare_length line =
    line * coefficient
    where coefficient = 0.4959
```

Ключевое слово `where` вводит выражение, которое можно использовать в теле функции. Рассматривайте его как псевдоним: идентификатор `coefficient` теперь можно использовать как аналог числового значения `0.4959`.

Локальных выражений может быть и несколько:

```
...
    line * coefficient - correction
    where coefficient = 0.4959
          correction = 0.0012
```

Есть ещё один способ ввести локальное вспомогательное выражение, а именно с помощью ключевого слова `let`. На примере нашей последней функции это будет выглядеть так:

```
prepare_length :: Double -> Double
prepare_length line =
    let coefficient = 12.4959
        correction = 0.0012
    in
    line * coefficient - correction
```

Общая модель такая:

```
let bindings in expression,
```

где *bindings* — локальные выражения, а *expression* — то место, где мы собираемся использовать эти локальные выражения.

Вы спросите, а в чём же разница между `where` и `let`?

Во-первых, выражение `where` может быть только одно и только в конце тела функции, в то время как выражение `let` может присутствовать многократно и в любой части тела функции.

Во-вторых, выражение, введённое ключевым словом `where`, видимо в любой точке тела функции, в то время как выражение, введённое ключевым словом `let`, может быть «супер-локальным». Например:

```
...
    let coefficient = 12.4959
        correction = 0.0012
    in
    line * coefficient - correction - (let s = 10.9 in s + 1) - s
```

Здесь мы ввели «супер-локальное» выражение с именем `s`, которое существует только внутри круглых скобок. Именно поэтому этот код не пройдёт компиляцию, ведь второе выражение `s` находится уже за пределами круглых скобок.

Без объявления

Как вы помните, нельзя объявить функцию и при этом не определить её. А можно ли определить функцию без объявления? Ответ: можно, но не рекомендуется.

Общепринятой практикой является объявлять функцию и тут же определять её. И несмотря на то, что мы *можем* написать так:

```
-- Объявления нет, сразу определение
prepare_length line =
    ...
```

делать так не рекомендуется, поскольку определение становится беднее, ведь описание типов аргументов и возвращаемого значения помогает лучше понять

работу функции. Кроме того, если вы не укажете эти типы, они станут полиморфными, но об этом мы поговорим позже.

Вот и всё, теперь вы знаете о чистых функциях. Кстати, они нам очень пригодятся — в последующих главах мы к ним вернёмся.

λ-функции

Теперь мы должны познакомится с любопытной и важной концепцией, а именно с λ-функциями (лямбда-функциями).

Вспомним упомянутое в предыдущей главе определение математической функции:

Функция — это описание зависимости чего-то от чего-то.

Однако в языке C (и подобных ему языках) функция никогда не ассоциировалась с таким определением. Напротив, функция там есть ни что иное, как подпрограмма, а имя функции есть ни что иное, как указатель на первую инструкцию этой подпрограммы.

Кроме того, функция в языке C является глобальной в рамках текущей единицы трансляции. И поэтому вызов функции — это своего рода «глобальный `goto`» в её тело, с последующим возвратом из него. Именно поэтому функция в языке C не может быть безымянной, потому что иначе её невозможно было бы вызвать.

λ-функция — совсем другой зверь.

Что это такое

В основе λ-функций лежит λ-исчисление, названное так по имени красивой греческой буквы. У λ-исчисления довольно-таки долгая академическая история, но нас интересует практическая сторона, поэтому сразу приведу пример.

Допустим, нам нужна математическая функция, принимающая некое целочисленное значение и возвращающая квадрат этого значения. Такую функцию мы можем описать так:

```
5 -> f -> 25
```

Проще некуда: на входе — 5, на выходе — 25. Внутренности этой функции можно описать так:

```
5 -> (x * x) -> 25
```

А теперь главный вопрос: как такую функцию описать *формально*? Вот тут-то на сцену и выходит λ-исчисление, ибо оно как раз и предлагает формализованный способ записи функции. Для нашей функции эта запись будет такой:

```
λx.x * x
```

Буква λ — это признак λ-функции. А читать это выражение следует так: «λ-функция (от) одного аргумента x, возвращающая результат умножения этого аргумента на самого себя».

Разделителем здесь является точка. Выражение слева от этой точки — список аргументов (в данном случае он один), а выражение справа от неё — тело функции.

Простое и элегантное описание, ничего лишнего. Нет даже имени. Особенностью λ-функции является её безымянность, ведь имя ей не нужно. И это принципиально отличает её от «обыкновенной» функции.

Как это выглядит в коде

λ-функции присутствуют во многих языках, но в Haskell вид λ-выражения максимально приближен к математическому. Сравните:

```
λx .  x * x  -- Математическая форма
\x -> x * x  -- Haskell-форма
```

Прямое сходство. Даже backslash вначале подходит как нельзя лучше: рассматривайте его как «спинку» буквы λ. Единственное отличие — это замена точки стрелочкой.

А теперь возникает резонный вопрос: как мы можем вызвать такую функцию? Вероятно, ответ удивит вас, но λ-функции, строго говоря, не вызывают. Впрочем, это лишь игра слов. Вернёмся на минутку в математику.

Идея λ-функции базируется на математическом принципе «аппликации» (application), или «применения». λ-функцию не вызывают с аргументом, а применяют (апплицируют) её к аргументу. Поэтому запись вида:

```
f a
```

принято читать так: «Применение функции `f` к аргументу `a`.»

Вот как это выглядит в Haskell:

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

λ-выражение, находящееся в скобках, порождает λ-функцию, которая сразу же применяется к аргументу 5.

Множество аргументов

λ-функция может применяться и к нескольким аргументам. Пусть у нас теперь будет функция, возвращающая результат умножения первого значения на второе:

```
main =
    print (f 5 6)
    where f = \arg1 arg2 -> arg1 * arg2
              аргументы
```

Между backslash и стрелочкой идёт список имён аргументов функции.

Какая от них польза

В языке C принята стандартная последовательность из трёх шагов при работе с функцией:

1. объявление,
2. определение,
3. вызов.

Например:

```
int sq( int i ) {
    return i * i;
}

int main() {
    printf( "%d", sq( 5 ) );
}
```

Мы готовим нашу «глобальную подпрограмму», а потом заходим в неё через вызов.

А вот как это выглядит в Haskell:

```
main = print ((\x -> x * x) 5)
```

Мы ничего не готовим заранее. Напротив, мы создаём функцию как значение, локально и непосредственно перед использованием. Создаём — и тут же применяем её к аргументу 5.

Для простоты мы можем ввести пояснительное выражение для нашей функции:

```
main =
    print (f 5)
    where f = \x -> x * x
```

Одно из преимуществ λ-функции как раз и заключается в её локальности. Зачем нам заранее объявлять и определять функцию, если мы можем создать и сразу использовать её непосредственно в том месте, где она нужна?

Конечно, если λ-функция используется в нескольких местах, мы можем, во избежание дубляжа, определить её глобально, связав с некоторым именем. Например:

```
f = \x -> x * x

main = print ((f 5) + (f 6))
```

Выражение `f` равно нашей λ-функции, и теперь мы можем многократно применять это выражение к различным аргументам.

Готово. Теперь вы знаете, что такое λ-функции. Однако самое интересное их применение связано с функциями высшего порядка, о которых мы поговорим прямо сейчас.

Функции высшего порядка

Функции высшего порядка (higher-order functions) занимают важное место в языке Haskell. Из предыдущих глав вы узнали, что чистые функции — это, в конечном итоге, значения. Следовательно, чистые функции можно, во-первых, передавать другим функциям в качестве аргументов, а во-вторых, возвращать их из других функций.

Функцией высшего порядка называют такую функцию, которая принимает другую функцию в качестве аргумента и/или возвращает другую функцию.

Разоблачение функций

Помните, в рассказе о чистых функциях было упомянуто, что они могут принимать как один, так и множество аргументов? Пришло время признаться в обмане, ибо правда такова:

Чистые функции в Haskell всегда принимают только один аргумент.

Да, но как же мы тогда смогли определить функции, принимающие по два и даже по три аргумента?

Это была хитрость, и называется она «каррирование» (currying). Слово это знаменитое, ибо происходит от имени Haskell Curry[1]. Каррирование — это превращение функции, принимающей множество аргументов, в функцию, принимающую все эти аргументы по одному.

Определим функцию деления двух чисел:

```
divide :: Double -> Double -> Double
divide arg1 arg2 = arg1 / arg2
```

Функция принимает два значения стандартного типа `Double` и возвращает результат деления первого значения на второе. Всё предельно просто. Но если мы заглянем «пот капот» вызова этой функции:

```
main = print (divide 10.03 2.1)
```

1 Это тот самый американский математик, в честь которого назван изучаемый нами язык.

то узнаем, что этот вызов происходит в *два* этапа:

1. Функция `divide` применяется к первому аргументу `10.03` и — внимание! — возвращает функцию типа `Double -> Double`.
2. Эта возвращённая функция, в свою очередь, применяется ко второму аргументу `2.1` и возвращает конечное значение `4.77`.

Мы можем явно отразить эту «двухэтапность», переписав вызов функции так:

```
(divide 10.03) 2.1
```

Функция применяется только к одному значению: сначала к `10.03`, а уже потом функция, возвращённая первым вызовом, применяется к `2.1`.

Именно по причине такой «двухэтапности» объявление функции divide содержит две стрелочки вместо одной:

```
divide :: Double -> Double -> Double
```

С концептуальной точки зрения такое объявление звучит так: «Функция `divide` принимает два значения типа `Double` и возвращает значение типа `Double`.» Однако правильнее читать его так: «Функция `divide` применяется к первому значению типа `Double` и возвращает функцию типа `Double -> Double`, которая применяется ко второму значению типа `Double` и возвращает конечное значение типа `Double`.»

Правильное прочтение объявления можно отразить и в самом этом объявлении:

```
divide :: Double -> (Double -> Double)
```

Теперь мы ясно видим, что на первом этапе происходит вызов функции от одного аргумента, возвращающей функцию типа `Double -> Double`, а на втором этапе происходит вызов второй функции, возвращённой на первом этапе.

По аналогии, если у нас есть функция, принимающая три аргумента:

```
total_sum :: Double -> Double -> Double -> Double
total_sum arg1 arg2 arg3 = arg1 + arg2 + arg3
```

то её вызов:

```
main = print (total_sum 10.03 2.1 45.7)
```

проходил бы в три этапа, и чтобы явно отразить этот факт, мы можем переписать объявление данной функции так:

```
total_sum :: Double -> (Double -> (Double -> Double))
```

а её вызов — так:

```
((total_sum 10.03) 2.1) 45.7
```

И чтобы всё окончательно прояснилось, изучим одну важную деталь.

Частичное применение функции

Несмотря на «двухэтапность» вызова функции `divide`, её тело будет выполнено один раз. Вызов один, просто он разделён на два последовательных шага. А чтобы понять суть этих шагов, изучим *частичное применение* функции (partial application).

Функцию называют частично применённой, если количество аргументов, к которым она применена, оказалось меньше ожидаемого ею количества аргументов. И здесь нам пригодятся уже известные нам *λ*-функции.

Применим функцию `divide` не к двум, а только к одному аргументу:

```
main =
    let temporary_function = divide 10.03 -- "запомнили" первое значение
    in
    print (temporary_function 2.1) -- а вот теперь можем выполнить работу!
```

Теперь всё встало на свои места. Здесь наглядно показано, что же на самом деле означает выражение вида:

```
(divide 10.03) 2.1
```

В результате первого вызова, когда мы применили функцию `divide` к первому аргументу, мы ещё не можем получить результат деления, ведь второго-то аргумента ещё нет! Вместо этого мы получили временную *λ*-функцию, которую для наглядности ассоциировали с выражением `temporary_function`. Эта временная *λ*-функция как бы запомнила значение первого аргумента, и только когда мы применим её ко второму аргументу, мы и получим результат деления.

По аналогии, вызов нашей функции `total_sum`, который происходит в три этапа, можно разложить так:

```
main =
    let first_function = total_sum 1.0       -- "запомнили" первый
        second_function = first_function 2.0 -- "запомнили" второй
    in
    print (second_function 3.0) -- а вот теперь можем складывать.
```

В процессе вызова у нас появилось уже две временные λ-функции, каждая из которых применялась к очередному аргументу и запоминала его. И только когда вторая промежуточная λ-функция была применена к третьему, последнему аргументу, мы и получили сумму.

Зачем это нужно

В подавляющем большинстве случаев знать вышеизложенную информацию о каррировании функций и о частичном применении не нужно. Главное преимущество такого подхода, при котором одна функция от нескольких аргументов раскладывается на цепочку функций от одного аргумента каждая, лежит в «академической плоскости»: проводить формальные математические доказательства гораздо легче, если договориться, что каждая из вычисляемых функций всегда принимает строго один аргумент и выдаёт строго одно значение.

Но нас с вами, как программистов-практиков, больше интересует аспект практический. И поэтому мы возвращаемся к рассмотрению функций высшего порядка (далее — ФВП).

Формально функции `divide` и `total_sum` являются ФВП, в силу тех самых промежуточных λ-функций. Фактически, все функции, принимающие более одного аргумента, являются ФВП. Но все эти промежуточные λ-функции — всего лишь «подкапотные» дела, они скрыты от наших глаз. Гораздо больший интерес для нас представляют «настоящие» ФВП, которые явно объявлены как принимающие на вход функциональные значения и/или возвращающие функциональные значения.

Рассмотрим небольшой пример:

```
type Login = String
type Password = String
type AvatarURL = String
type UserId = Integer

user_info :: Login -> Password -> AvatarURL -> UserId -> String
user_info login password avatar_URL user_id =
```

```
    "Full info about user @" ++ (show user_id) ++ ":" ++
    "\n login: " ++ login ++
    "\n password: " ++ password ++
    "\n avatar URL: " ++ avatar_URL

type EmptyInfo = Login -> Password -> AvatarURL -> UserId -> String
type WithLogin = Password -> AvatarURL -> UserId -> String
type AndWithPassword = AvatarURL -> UserId -> String
type AndWithAvatarURL = UserId -> String

store_login_in :: EmptyInfo -> UserId -> WithLogin
store_login_in empty_info user_id =
    empty_info "denis"
    -- В реальности логин будет получен в соответствии с переданным user_id

store_password_in :: WithLogin -> UserId -> AndWithPassword
store_password_in info_with_login user_id =
    info_with_login "123456789abc"
    -- В реальности пароль будет получен в соответствии с переданным user_id

store_avatar_URL_in :: AndWithPassword -> UserId -> AndWithAvatarURL
store_avatar_URL_in info_with_password user_id =
    info_with_password "http://dshevchenko.biz/denis_avatar.png"
    -- В реальности URL будет получен в соответствии с переданным user_id

main =
    let user_id = 1234
        info_with_login = store_login_in user_info user_id
        info_with_password = store_password_in info_with_login user_id
        info_with_avatar_URL = store_avatar_URL_in info_with_password user_id
        full_info_about_user = info_with_avatar_URL user_id
    in
    putStrLn full_info_about_user
```

А теперь разберём это хозяйство по косточкам.

Во-первых, появилась новая для нас конструкция:

```
type Login = String
```

Ключевое слово `type` добавляет псевдоним для уже известного типа. Теперь вместо типа `String` можно использовать идентификатор `Login`.

Далее мы определили функцию:

```
user_info :: Login -> Password -> AvatarURL -> UserId -> String
```

Тут всё просто: функция `user_info` ожидает на вход логин, пароль, адрес аватара и идентификатор пользователя, а на выходе выдаёт некую описывающую строку. Обратите внимание и на двойной плюс:

```
"\n login: " ++ login
```

Это оператор конкатенации двух строк в одну.

А вот теперь начинается самое интересное. Подразумевается, что изначально у нас имеется только идентификатор пользователя, а соответствующие ему логин, пароль и путь к аватару нам нужно откуда-то получить. К счастью, у нас есть три функции, каждая из которых знает, где взять логин, пароль и путь к аватару соответственно. И каждая из этих трёх функций является ФВП.

Рассмотрим псевдонимы:

```
type EmptyInfo = Login -> Password -> AvatarURL -> UserId -> String
type WithLogin =          Password -> AvatarURL -> UserId -> String
type AndWithPassword =                AvatarURL -> UserId -> String
type AndWithAvatarURL =                            UserId -> String
```

Каждый из них вводит упрощающее имя для функционального типа, образованного «урезанием» от типа функции `user_info`. Обратите внимание: каждый последующий тип ожидает на один аргумент меньше, чем предыдущий тип. Эти псевдонимы задают типы для очередной промежуточной λ-функции, которые нужны, как вы уже догадались, для частичного применения функции `user_info`.

Рассмотрим первый вызов:

```
info_with_login = store_login_in user_info user_id
```

Здесь мы передаём функцию `user_info` в качестве первого аргумента функции `store_login_in`, внутри которой мы применяем переданную функцию `user_info` к единственному аргументу, а именно к логину. Соответственно, на выходе из функции `store_login_in` мы получаем первую промежуточную λ-функцию, в которой мы сохранили значение логина (именно поэтому тип этой λ-функции ассоциирован со словом `WithLogin`).

Далее следует вызов:

```
info_with_password = store_password_in info_with_login user_id
```

Здесь мы передаём нашу промежуточную λ-функцию в качестве первого аргумента функции `store_password_in`. Эта функция, в свою очередь, применяет переданную ей λ-функцию к единственному аргументу, а именно к паролю. Таким образом, на выходе из функции `store_password_in` мы имеем вторую промежуточную λ-функцию, в которой сохранены уже два значения: полученный на предыдущем вызове логин и на этом вызове — пароль.

То же самое справедливо и для следующего вызова:

```
info_with_avatar_URL = store_avatar_URL_in info_with_password user_id
```

На выходе из функции `store_avatar_URL_in` мы получаем третью λ-функцию, в которой сохранены уже три значения: логин, пароль и путь к аватару.

В итоге мы применяем эту третью λ-функцию к последнему нужному аргументу, а именно к идентификатору пользователя:

```
full_info_about_user = info_with_avatar_URL user_id
```

Здесь и происходит «полноценный» вызов функции `user_info`, в результате которого мы и получаем описывающую строку:

```
Full info about user @1234:
 login: denis
 password: 123456789abc
 avatar URL: http://dshevchenko.biz/denis_avatar.png
```

Таким образом, функция `user_info` была частично применена трижды, каждый раз получая очередной аргумент, и лишь к четвёртому применению она получила все необходимые ей аргументы. Это можно сравнить с конвейерной цепочкой, на каждом шаге которой эта функция получала очередной аргумент.

Впрочем, нужны ли были такие сложности? Ведь мы можем передавать в каждую из этих трёх функций только значение `user_id`, а возвращать никакую не промежуточную λ-функцию, а непосредственно логин, пароль и адрес аватара соответственно. Например, вместо функции `store_login_in` можно определить функцию `obtain_login` следующего вида:

```
obtain_login :: UserId -> Login
obtain_login user_id =
    -- Получаем откуда-то логин и просто возвращаем его.
```

Ну а что если мы не хотим возвращать логин в явном виде? Ведь в случае с частичным применением мы упаковываем логин в промежуточную λ-функцию

(то есть фактически прячем логин в неё), а в этом случае мы явно возвращаем его на показ всему миру. Первое решение может оказаться более приемлемым.

Или другой пример:

```
type UserId = Integer
type Prefix = String

obtain_login :: UserId -> (Prefix -> String)
obtain_login user_id =
    login_storage "denis" -- Подразумевается, что логин как-то получен.
    where login_storage = \login prefix -> prefix ++ ": " ++ login

main =
    let user_id = 1234
    in
    putStrLn ((obtain_login user_id) "My login")
```

Рассмотрим функцию `obtain_login` подробнее:

```
obtain_login :: UserId -> (Prefix -> String)
obtain_login user_id =
    login_storage "denis"
    where login_storage = \login prefix -> prefix ++ ": " ++ login
```

Здесь мы, на основании полученного извне идентификатора пользователя, откуда-то извлекаем логин и сразу же прячем его в λ-функцию, тут же нами и созданную. В результате функция `obtain_login` возвращает частично применённую функцию, которую мы вторично применяем к строке-префиксу — и в результате на выходе мы получаем готовый результат:

```
My login: denis
```

Готово. Теперь вы знаете о функциях высшего порядка. Именно по причине того, что с функциями в Haskell можно работать как со значениями, мы можем составлять из них гибкие комбинации.

Функциональные цепочки

В отношении функций часто можно сказать: «Один в поле не воин». В этой главе мы рассмотрим два удобных способа организации взаимодействия функций.

Пример с URL

Известно, что вид URL обязан соответствовать особым правилам[1]. Но в реальной жизни это не всегда так, поэтому иногда URL нужно преобразовать к правильному виду. Вот как это может выглядеть:

```
import Data.Char
import Data.String.Utils

-- оборачиваем в свою функцию для красивого инфиксного использования
starts_with = \url prefix -> startswith prefix url

add_prefix :: String -> String
add_prefix url =
    if url `starts_with` prefix then url else prefix ++ url
    where prefix = "http://"

encode_all_spaces = \url -> replace " " "%20" url

make_it_lower_case = \url -> map toLower url

main =
    putStrLn (add_prefix (encode_all_spaces (make_it_lower_case url)))
    where url = "www.SITE.com/test me/Start page"
```

Вывод будет таким:

```
http://www.site.com/test%20me/start%20page
```

1 http://www.w3.org/Addressing/URL/uri-spec.html

Мы импортировали новый модуль `Data.String.Utils`. Этот модуль является частью пакета `MissingH`, содержащего кучу разных полезных утилит. Установим этот пакет:

```
cabal install MissingH
```

Упомянем его в `Real.cabal`:

```
build-depends: base ==4.6.*, MissingH
```

В модуле `Data.String.Utils` присутствуют различные вкусности для работы со строками, но нам понадобилась лишь одна функция `startswith`, проверяющая, является ли одна строка началом другой строки.

Итак, у нас имеются три функции, каждая из которых делает с нашим URL простую исправительную операцию: `make_it_lower_case` переводит все символы в нижний регистр, `encode_all_spaces` заменяет пробелы строкой `%20`, `add_prefix` добавляет префикс, если таковой отсутствует. То есть у нас есть цепочка из трёх функций: на входе этой цепочки неправильный URL, а на выходе — исправленный URL. Рассмотрим эту цепочку поближе:

```
add_prefix (encode_all_spaces (make_it_lower_case url))
```

Сначала вызывается `make_it_lower_case`, потом `encode_all_spaces`, потом `add_prefix`. Каждая из функций принимает на вход URL и возвращает обработанный ею URL, поступающий на вход следующей функции.

И всё бы хорошо, но есть в такой цепочке один минус — многовато круглых скобок[2]. Проблема усугубилась бы, если бы функций-исправителей было не три, а больше. Существует два способа сделать такие цепочки красивее.

Функция композиции

Функция композиции (function composition) выглядит как точка. Её назначение — компоновать функции в цепочку. Вот так:

```
(add_prefix . encode_all_spaces . make_it_lower_case) url
```

2 Да простят меня программисты Lisp.

Функция композиции берёт наши три функции и объединяет их в одну конструкцию, которая один раз применяется к нашему `url`, и результат будет точно таким же, как если бы мы написали так:

```
add_prefix (encode_all_spaces (make_it_lower_case url))
```

Можно сказать, что функция композиции создала стек из трёх функций: перечислены они слева направо, а вызываться будут справа налево. Таким образом, строка `url` едет по конвейеру, заезжая в него с правого края и выезжая с левого.

Функция применения

А ещё есть функция применения (function application), иногда говорят «функция аппликации». Выглядит она как значок доллара. Её назначение — компоновать функции в цепочку. Вот так:

```
add_prefix $ encode_all_spaces $ make_it_lower_case url
```

Здесь мы обошлись вообще без скобок. И такое написание также аналогично исходному:

```
add_prefix (encode_all_spaces (make_it_lower_case url))
```

Здесь тоже получился стек из функций: перечислены слева направо, а вызываются справа налево. Такой вызов справа налево называют ещё правоассоциативным (right-associative).

Вместе

Вероятно, вам интересно, а в чём же разница между этими двумя способами? Ведь и первый и второй предназначены для организации стековой цепочки функций.

Главная разница состоит в том, что функция применения позволяет объединять не только функции, но также функцию с её аргументом:

```
main = print $ "Hi master!"
```

Долларовый оператор объединил значение и применяемую к этому значению функцию. А вот функция композиции не позволит проделать такой фокус.

Вы спросите, зачем это нужно? Отвечаю:

```
main = print ("Hi master '" ++ name ++ "', have a nice day!")
```

Функция `print` готова работать исключительно с одним аргументом, поэтом три литерала, объединяющиеся в один, необходимо взять в скобки. Значительно удобнее написать так:

```
main = print $ "Hi master '" ++ name ++ "', have a nice day!"
```

Мы избавились от скобок, объединив функцию и её аргумент в маленькую цепочку. Именно благодаря такому свойству функции композиции и применения часто используют вместе:

```
add_prefix . encode_all_spaces . make_it_lower_case $ url
```

Точка объединяет функции, а доллар привязывает их к аргументу.

Часть 4 О списках

Списки — одним взглядом

Списки в Haskell — это наборы элементов одного типа. Приступим к их изучению.

Прежде всего знайте: когда вы видите в коде квадратные скобки — значит, список где-то рядом. Вот список из трёх целочисленных элементов:

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

а вот пустой список:

```
[]
```

Элементами списка могут быть значения любого типа, в том числе и другие списки. Мы даже можем создать список функций, но после прочтения предыдущих глав вас этот факт не должен удивлять.

Простейшие действия

Если списки создаются — значит это кому-нибудь нужно. Вот функция, возвращающая список из трёх строк:

```
list_of_names :: String -> [String]
list_of_names prefix =
    [prefix ++ "John", prefix ++ "Anna", prefix ++ "Andrew"]

main = print $ list_of_names "Dear "
```

Результат:

```
["Dear John","Dear Anna","Dear Andrew"]
```

Обратите внимание на объявление этой функции:

```
list_of_names :: String -> [String]
```

Тип `[String]` — это тип списка строк. А, например, список символов объявляется как `[Char]`. Кстати говоря, строка — это и есть список символов, то есть тип `String` эквивалентен типу `[Char]`. Поэтому объявление может быть и таким:

```
list_of_names :: String -> [[Char]]
```

Вот так можно узнать размер списка:

```
main =
    print $ length list_of_animals
    where list_of_animals = ["Bear", "Tiger", "Lion", "Wolf"]
```

А так можно узнать, есть ли заданное значение в списке:

```
this_is_a_wild_animal :: String -> Bool
this_is_a_wild_animal name =
    name `elem` wild_animals
    where wild_animals = ["Bear", "Tiger", "Lion", "Wolf"]

main = print $ if this_is_a_wild_animal "Cat" then "Yes!" else "No!"
```

Здесь функция `elem`, записанная в инфиксной форме[1], проверяет наличие строки `Cat` в списке диких животных.

Стандартная библиотека Haskell позволяет делать со списком самые разные вещи, такие как получение минимального значения, вычисления суммы элементов, извлечение части списка, проверка на пустоту и равенство и так далее и в том же духе.

Неизменность списка

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

```
add_new_host_to_front :: String -> [String] -> [String]
add_new_host_to_front new_host list_of_hosts =
    new_host : list_of_hosts
```

1 Это когда функция располагается между двумя аргументами, подобно тому как бинарный оператор располагается между операндами.

```
main =
    print $ add_new_host_to_front "124.67.54.90" list_of_hosts
    where list_of_hosts = ["45.67.78.89", "123.45.65.54", "127.0.0.1"]
```

Вывод:

```
["124.67.54.90","45.67.78.89","123.45.65.54","127.0.0.1"]
```

С концептуальной точки зрения функция `add_new_host_to_front` добавила новый адрес в начало переданного ей списка. Но в действительности никакого добавления не произошло: функция просто взяла элемент `new_host` и список `list_of_hosts` и создала на их основе новый список, содержащий уже четыре адреса вместо трёх.

Действия над элементами

Мы создаём список для того, чтобы что-то делать с его элементами. Например, такая функция:

```
remove_all_empty_names_from :: [String] -> [String]
remove_all_empty_names_from list_of_names =
    filter not_empty_name list_of_names
    where not_empty_name = \name -> not $ null name

main =
    print $ remove_all_empty_names_from list_of_names
    where list_of_names = ["John", "", "Ann"]
```

Стандартная функция `filter` последовательно применяет предикат `not_empty_name` к каждой строке в списке и конструирует новый список лишь из тех строк, которые удовлетворяют этому предикату. В качестве предиката выступает λ-функция, применяющаяся к одному аргументу и возвращающая значение `True` только в том случае, если он не `null`.

Помимо функций `map` и `filter`, в стандартной библиотеке Haskell есть и другие вкусности для работы с элементами (проверки, замены, сортировки, перестановки и тому подобное).

Всё. Теперь вы знаете о списках. Осталось рассмотреть несколько деликатесов.

Диапазоны

Диапазон — это конструкция, автоматически создающая список по заданному признаку.

Суть

Если нам нужно создать список целых чисел от 1 до 10, мы можем написать так:

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

а можем просто задать диапазон:

```
[1..10]
```

Готово.

Разумеется, такой фокус можно проделать не только с числами. Например, вот так мы получим список всех букв английского алфавита в нижнем регистре:

```
main = print ['a'..'z']
```

На выходе получим красивый список символов, или простую строку:

```
"abcdefghijklmnopqrstuvwxyz"
```

Умные диапазоны

Диапазоны можно задавать весьма гибко.

Вот так мы можем получить список всех чётных чисел от 2 до 30:

```
main = print [2,4..30]
```

Мы задали шаг между значениями элементов, а остальные значения были созданы уже автоматически.

Можно и в порядке убывания:

```
main = print [120,110..10]
```

На выходе получим список с десятками:

```
[120,110,100,90,80,70,60,50,40,30,20,10]
```

А вот чего компилятор не потерпит, так это излишних указаний с вашей стороны. Поэтому не пишите так:

```
main = print [2,4,6..30]
```

и так тоже не пишите:

```
main = print [2,4..28,30]
```

Компилятор обидится на вас, ведь вы пытаетесь сказать ему то, что он и сам прекрасно понимает.

Без конца

Как вы помните, ленивость языка Haskell позволяет нам оперировать бесконечными списками. И мы можем создать такой список через диапазон.

Например, вот такой диапазон:

```
[1..]
```

создаст бесконечный список целых чисел, начиная с 1. Но, как вы уже знаете, в действительности созданный этим диапазоном список будет вовсе не бесконечным, а лишь *достаточно* большим:

```
main = print $ take 5 [1..]
```

Вывод:

```
[1,2,3,4,5]
```

Мы можем задать и шаг:

```
main = print $ take 5 [2,4..]
```

В этом случае вывод будет таким:

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

Вот такие они, эти диапазоны.

Кортежи

Кортеж — это особый вид списка. Он тоже хранит в себе набор элементов, однако имеет три фундаментальных отличия от списка:

1. круглые скобки вместо квадратных;
2. гетерогенность;
3. тип, зависящий от размера.

С первым отличием всё очевидно:

```
["Denis", "Shevchenko"]  -- Это список из двух строк
("Denis", "Shevchenko")  -- Это кортеж из двух строк
```

Второе отличие — это способность кортежа хранить в себе элементы разных типов:

```
["Denis", 1.234]  -- Будьте уверены, компиляция не пройдёт
("Denis", 1.234)  -- А тут – без проблем
```

Теперь о третьем отличии. Если у нас есть два разных по размеру списка строк:

```
["Denis", "Vasil`evich", "Shevchenko"]
["Denis", "Shevchenko"]
```

тип обоих этих списков одинаков, а именно `[String]`. Тип списка не зависит от количества элементов в нём.

С кортежами всё обстоит совершенно иначе. Если у нас есть два кортежа, разных по длине:

```
("Denis", "Vasil`evich", "Shevchenko")
("Denis", "Shevchenko")
```

типы этих кортежей будут абсолютно разными: тип первого будет `(String, String, String)`, а тип второго `(String, String)`. Поэтому если функцию, в качестве аргумента ожидающую кортеж из двух строк, применить к кортежу из трёх строк, компилятор выразит свой категорически протест:

```
    Couldn't match expected type `(String, String)'
                with actual type `([Char], [Char], [Char])'
```

Оно и понятно: ожидали кортеж из двух строк, а тут вдруг — из трёх!

Что с ними можно делать

Единственное, что можно сделать с кортежем — извлечь хранящиеся в нём элементы. Всё.

На практике чаще всего используют кортежи из двух элементов. Такой кортеж ещё называют парой (pair). Чтобы извлечь хранящиеся в нём элементы, используются стандартные функции `fst` и `snd`.

Определим функцию, применяющуюся к кортежу, хранящему две части шахматного хода:

```
chess_move :: (String, String) -> String
chess_move pair =
    (fst pair) ++ "-" ++ (snd pair)

main = print $ chess_move ("e2", "e4")
```

Мы последовательно извлекли первый и второй элементы из полученной пары и сделали из них единую строку.

Но что же мы будем делать, если количество элементов в кортеже больше двух? Ведь функции `fst` и `snd` работают только с парами. Если элементов больше двух, извлекать их нужно иным способом.

Неудобный способ

Первый способ неудобен, ибо нам придётся самим определять необходимые функции. Но нас трудности не страшат, поэтому сделаем это:

```
get1 = \(elem, _, _, _) -> elem
get2 = \(_, elem, _, _) -> elem
get3 = \(_, _, elem, _) -> elem
get4 = \(_, _, _, elem) -> elem
```

Подразумевается, что мы хотим работать с кортежем из четырёх элементов. В этом случае у нас есть лишь четыре варианта извлечения, поэтому определим функцию для извлечения первого элемента, второго, третьего и четвёртого.

Кстати, говоря «первый элемент», мы подразумеваем именно первый по счёту, поэтому цифра `1` в имени `get1` — это порядковый номер, а не индекс.

Нас совершенно не интересуют типы элементов в нашем кортеже, поэтому определим наши функции в виде λ-функций, ассоциированных с соответствующими названиями. Это сделает наш код более компактным.

А теперь рассмотрим определение первой λ-функции:

```
\(elem, _, _, _) -> elem
```

Эта функция применяется к кортежу из четырёх элементов и возвращает первый из них. Обратите внимание на странные символы подчёркивания. Воспринимайте этот символ как «нечто», «что бы то ни было». Мы говорим: «Да, в этом кортеже есть четыре элемента, но нас абсолютно не интересует, что там под номером два, и что под номером три, и что под номером четыре. Нас интересует только то, что под номером один. Вот этот номер один мы и вернём.»

Так же и вторая функция:

```
get2 = \(_, elem, _, _) -> elem
```

Получаем четыре элемента, и хотя что-то там стоит под номерами один, три и четыре, нас это не волнует, нам нужен только элемент под номером два, поэтому именно его и возвращаем.

А теперь можем написать так:

```
main = print $ get3 ("One", "Two", "Three", "Four")
```

и получаем ожидаемый результат:

```
"Three"
```

Удобный способ

Зачем делать самому то, что уже сделали другие? А другие уже сделали пакет `tuple` из Hackage.

Установим его командой:

```
cabal install tuple
```

Затем добавим имя этого пакета к параметру `build-depends` в нашем `.cabal`-файле:

```
    build-depends:       base ==4.6.*, tuple
```

Импортируем модуль `Data.Tuple.Select`, и сразу же можем приступить:

```
import Data.Tuple.Select

main = print $ sel3 ("One", "Two", "Three", "Four")
```

Метод `sel3` извлекает третий элемент кортежа. Просто и удобно. Кстати, в модуле `Data.Tuple.Select` определены функции от `sel1` до `sel15`. Авторы вполне резонно предположили, что создавать кортеж из более чем 15 элементов никакому вменяемому программисту в голову не придёт…

А кстати, как же насчёт безопасности? Что будет, если мы по ошибке попытаемся извлечь из этого кортежа пятый элемент? Попробуем:

```
import Data.Tuple.Select

main = print $ sel5 ("One", "Two", "Three", "Four")
```

Итак, пытаемся извлечь пятый элемент при наличии только четырёх. Получили трудноуловимую ошибку? Или, может, будет брошено исключение? Вовсе нет. Такой код просто не пройдёт компиляцию:

```
src/Main.hs:23:12:
    No instance for (Sel5 ([Char], [Char], [Char], [Char]) a0)
      arising from a use of `sel5'
```

Тип кортежа жёстко завязан на количество хранящихся в нём значений. Именно поэтому такого рода ошибки будут выявлены на стадии компиляции.

Теперь вы и о кортежах знаете.

List comprehension

Не удивляйтесь, что название этой главы не переведено на русский. Корректный перевод понятия «list comprehension» я так и не смог подобрать, долго размышлял — и в итоге решил оставить как есть.

Речь пойдёт об одной хитрой конструкции, предназначенной для прохода по элементам списка(ов) и применения к ним некоторых действий. Да-да, это похоже на уже известные нам функции `map` и `filter`, но есть некоторые дополнительные вкусности.

Хитрый список

Вот как это выглядит:

```
import Data.Char

main = print [toUpper c | c <- "http"]
```

На выходе получим:

```
"HTTP"
```

Рассмотрим поближе:

```
[toUpper c | c <- "http"]
```

Мы видим квадратные скобки... То есть перед нами список? Не совсем. Можно сказать, что перед нами генератор списка. Скелет такой конструкции можно представить так:

```
[OPERATION ELEM | ELEM <- LIST]
```

где `LIST` — список, `ELEM` — элемент этого списка, а `OPERATION` — действие, применяемое к каждому элементу. Мы говорим: «Возьми список `LIST`, последовательно пройдись по всем его элементам и примени к каждому из них функцию

`OPERATION`.» В результате значения, возвращаемые функцией `OPERATION`, породят новый список.

В данном случае мы пройдём по всем символам строки `http` и применим к каждому из её символов функцию `toUpper`, которая в свою очередь переведёт этот символ в верхний регистр. В результате мы получим новую строку `HTTP`.

Добавляем предикат

Мы можем добавить предикат в эту конструкцию. Тогда её скелет станет таким:

```
[OPERATION ELEM | ELEM <- LIST, PREDICATE]
```

В этом случае мы говорим: «Возьми список `LIST`, последовательно пройдись по всем его элементам и примени функцию `OPERATION` только к тем элементам, которые удовлетворят предикату `PREDICATE`.»

Например:

```
import Data.Char

main = print [toUpper c | c <- "http", c == 't']
```

На выходе будет:

```
"TT"
```

Мы прошлись по всем четырём символам строки `http`, но функция `toUpper` была применена только к тем символам, которые удовлетворили предикату `c == 't'`. Именно поэтому на выходе мы получили строку из двух символов, ибо только они удовлетворили этому предикату.

Предикатов, кстати, может быть несколько. Например, так:

```
[toUpper c | c <- "http", c /= 'h', c /= 'p']
```

Вывод в этом случае будет таким же:

```
"TT"
```

Здесь два предиката, `c /= 'h'` и `c /= 'p'`. Они соединяются в единый предикат через логическое «И», поэтому мы можем написать и так:

```
[toUpper c | c <- "http", c /= 'h' && c /= 'p']
```

Результат будет таким же.

Обратите внимание на комбинацию символов `/=`. Это оператор «не равно», аналог оператора `!=` в языке C. Кстати, он тоже носит математический окрас. Сравните:

```
/=   - Haskell-форма
≠    - математическая форма
```

Симпатично, не правда ли? Прямое сходство, мы лишь передвинули перечеркивающую косую палочку.

Больше списков

Мы можем использовать эту конструкцию для совместной работы с несколькими списками. Скелет в этом случае будет таким:

```
[OPERATION_with_ELEMs | ELEM1 <- LIST1, ..., ELEMN <- LISTN ]
```

Здесь мы работаем сразу с `N` списками, а `OPERATION_with_ELEMs` представляет собой функцию, в которую передаются все элементы наших списков. Например:

```
main =
    print [prefix ++ name | name <- names, prefix <- name_prefix]
    where names = ["James", "Victor", "Denis", "Michael"]
          name_prefix = ["Mr. "]
```

На выходе получим:

```
["Mr. James","Mr. Victor","Mr. Denis","Mr. Michael"]
```

Мы последовательно прошлись по всем элементам списков `names` и `name_prefix`.

Обратите внимание, в списке `name_prefix` лишь один префикс. Вот что будет, если префиксов два:

```
main =
    print [prefix ++ name | name <- names, prefix <- name_prefix]
    where names = ["James", "Victor", "Denis", "Michael"]
          name_prefix = ["Mr. ", "sir "]  -- Теперь префиксов два
```

В этом случае на выходе будет:

```
["Mr. James","sir James","Mr. Victor","sir Victor","Mr. Denis","sir
Denis","Mr. Michael","sir Michael"]
```

Мы последовательно использовали каждый элемент из списка `names` *и* каждый элемент из списка `name_prefix`.

Добавляем условие

Предикат не всегда применим к элементам списка. В ряде случаев нам нужно условие. Добавим его:

```
main =
    print [if car == "Bentley" then "Wow!" else "Good!" | car <- cars]
    where cars = ["Mercedes",
                  "BMW",
                  "Bentley",
                  "Audi",
                  "Bentley"]
```

Результат:

```
["Good!","Good!","Wow!","Good!","Wow!"]
```

Мы прошлись по списку марок автомобилей и применили к каждой из них условие, которое вернуло строку `"Wow!"` или строку `"Good!"`.

Пример

Разберём более практичный пример:

```
import Data.String.Utils

ends_with :: String -> String -> Bool
```

```
ends_with str suffix = endswith suffix str

check_googler_by :: String -> String
check_googler_by email =
    if email `ends_with` "gmail.com"
    then name_from email ++ " is a Googler!"
    else email
    where name_from = \full_email -> takeWhile (/= '@') full_email

main = print [check_googler_by email | email <- ["adam@gmail.com",
                                                 "bob@yahoo.com",
                                                 "richard@gmail.com",
                                                 "elena@yandex.ru",
                                                 "denis@gmail.com"]]
```

Результат:

```
["adam is a Googler!","bob@yahoo.com","richard is a
Googler!","elena@yandex.ru","denis is a Googler!"]
```

Мы проанализировали список email-адресов, и заменили все gmail-адреса фразой, начинающейся с имени пользователя. А теперь по шагам.

Из уже знакомого нам модуля `Data.String.Utils` мы возьмём функцию `endswith`, проверяющую, завершается ли одна строка другой строкой. Для красивого инфиксного использования мы обернули её собственной функцией:

```
ends_with :: String -> String -> Bool
ends_with str suffix = endswith suffix str
```

В результате наш код приобретает литературно точный вид:

```
if "adam@gmail.com" `ends_with` "gmail.com"
```

Теперь рассмотрим эту строку:

```
takeWhile (/= '@') full_email
```

Скелет стандартной функции `takeWhile` можно отобразить так:

```
takeWhile PREDICATE LIST
```

Здесь мы говорим: «Последовательно забирай (take) элементы из списка `LIST` до тех пор (While), пока `PREDICATE`, применённый к этим элементам, возвращает `True`. Если наткнёшься на элемент, не соответствующий этому предикату, прекращай работу и возвращай список из ранее полученных элементов.» Мы хотим извлечь имя пользователя из его email-адреса, а значит, мы бежим по email до тех пор, пока символы не равны `'@'`, что и отражается предикатом `(/= '@')`. Как только натыкаемся на собачку — возвращаем всё, находящееся перед ней.

Вот и всё. Теперь вы знаете, что такое list comprehension и как его можно использовать в вашем коде.

Часть 5 О пользовательских типах

Типы — одним взглядом

В настоящих проектах нам обязательно понадобятся наши собственные типы. О них и поговорим.

Прежде всего рассмотрим новые ключевые слова. Слово `data` служит для определения *типа*. Слово `class` используется для определения *класса типов*. А слово `instance` необходимо для определения *экземпляра класса типов*. Приступим.

Собственный тип

Определим тип для IP-адреса:

```
data IP_address = IP_address String
```

Готово. Перед нами — простейший пользовательский тип. Значение этого типа фактически будет представлять собой значение типа `String` с меткой `IP_address`. Рассматривайте метку как пояснительный идентификатор. Многие типы не имеют метки, поэтому их можно инициализировать значениями непосредственно. Например, если у нас есть некая функция, принимающая значение типа `Int`, то при её вызове мы будем писать просто:

```
show 6
```

Однако если эта же функция будет применена к значению нашего типа `IP_address`, мы должны будем явно указать это:

```
show (IP_address "127.0.0.1")
```

Выражение `(IP_address "127.0.0.1")` породит значение типа `IP_address`, содержащее в себе строку со значением `"127.0.0.1"`.

Кстати, метку принято называть конструктором значения.

Запомните: имя типа не может начинаться с маленькой буквы. Поэтому такой код:

```
data ip_address = ip_address String
```

будет отвергнут компилятором.

Класс типов

Проблема в том, что тип `IP_address` в его нынешнем виде настолько примитивен, что не представляет для нас никакого интереса. Мы, создав значение этого типа, даже не сможем вывести его на экран. И если мы прямо сейчас напишем так:

```
main = putStrLn $ show $ IP_address "127.0.0.1"
```

компилятор выдаст нам следующее:

```
No instance for (Show IP_address) arising from a use of `show'
```

И мы не имеем права обижаться на компилятор, он поступил совершенно правильно: стандартная функция `show`, преобразующая переданный ей аргумент в строковый вид, не имеет ни малейшего понятия о том, как представить объект типа `IP_address` в виде строки. И она не узнает это до тех пор, пока мы явно не расскажем ей об этом. Вот тут-то и выходят на сцену классы типов.

Класс типов — это логическая группа типов, отражающая общие для всех этих типов черты. Класс типов предоставляет набор методов, и каждый тип, имеющий отношение к данному классу, должен предоставлять свою реализацию этих методов. Класс типов можно рассматривать как интерфейс.

Например, в стандартной библиотеке Haskell есть класс типов `Show`. Он обобщает все типы, объекты которых могут быть «показаны» (shown), то есть отображены в виде стандартной строки. Вот определение этого класса:

```
class Show a where
    show :: a -> String
```

Здесь присутствует один-единственный метод `show`, принимающий в качестве аргумента объект типа `a` и возвращающий строковое отображение этого объекта.

Вы спросите, что это за тип такой — `a`? Ранее я уже упоминал термин *полиморфный тип*, так вот обозначенное буквой a — это он и есть. Благодаря этому метод `show` может применяться к значениям самых разных типов.

Возникает вопрос, откуда метод show узнаёт, *как* ему отобразить объект конкретного типа в виде строки? Ведь мы можем применить его к объектам разных

типов, для которых понятие «показать» может иметь абсолютно разный смысл. На сцену выходит экземпляр класса типов.

Экземпляр класса типов

Если класс типов `Show` — это логическая группа для типов, объекты которых можно показать, то экземпляр класса типов `Show` — это реальное объяснение того, *как* можно показать объект того или иного типа. И если мы хотим показывать объекты типа `IP_address`, мы обязаны предоставить экземпляр класса типов `Show` для типа `IP_address`. Сделаем же это:

```
instance Show IP_address where
    show (IP_address address) =
        if address == "127.0.0.1" then "localhost" else address
```

Мы использовали ключевое слово `instance`, а полиморфный тип `a` заменили на тип `IP_address`. Далее следует определение метода `show`, которое и объясняет, как отобразить значение типа `IP_address` в виде строки.

Взглянем ещё раз на это определение:

```
    show (IP_address address) =
        if address == "127.0.0.1" then "localhost" else address
```

Первая строка показывает, что метод `show` принимает объект типа `IP_address`, порождённый выражением `(IP_address address)`. Далее следует простое условие, выводящее слово `"localhost"` в том случае, если адрес равен `"127.0.0.1"`.

А теперь пишем:

```
main = putStrLn $ show $ IP_address "127.0.0.1"
```

Экземпляр класса типов `Show`, определённый для типа `IP_address`, прекрасно справится со своими обязанностями, поэтому мы получим ожидаемый вывод:

```
localhost
```

Всё. Теперь вы кое-что знаете о типах.

О конструкторах значений

Конструктор значения (value constructor) — штука полезная. Как мы увидели в предыдущей главе, конструктор позволяет идентифицировать значения наших собственных типов.

Иные имена

Конструктор значения может отличаться от имени самого типа. Взгляните:

```
data IP_address = IP String  -- Имя типа осталось неизменным

instance Show IP_address where
    show (IP address) =
        if address == "127.0.0.1" then "localhost" else address
```

Теперь мы будем создавать значения нашего типа так:

```
main = putStrLn $ show $ IP "127.3.0.1"
```

Такой подход позволяет писать более краткий, но при этом ничуть не менее понятный код. Обратите внимание: конструктор `IP` используется лишь в местах конструирования объекта, в то время как в описании экземпляра класса типов `Show` мы по-прежнему использовали имя самого типа `IP_address`.

Множество конструкторов

Конструкторов может быть более одного, и эта возможность используется очень часто. Например:

```
data IP_address = IP String | Host String
```

Мы ввели два конструктора, `IP` и `Host`. Здесь используется оператор `'|'`, знакомый программистам C как оператор бинарного «ИЛИ». Выглядит довольно логично, и поэтому мы можем прочесть эту строку так:

```
data IP_address  =  IP String  |  Host String
тип  IP_address это IP String или Host String
```

Определим метод `show` для каждого из этих конструкторов:

```
instance Show IP_address where
    show (IP address) =
        address

    show (Host address) =
        if address == "127.0.0.1" then "localhost" else address
```

Теперь, если мы напишем так:

```
main = putStrLn $ show $ IP "127.0.0.1"
```

вывод будет таким:

```
127.0.0.1
```

Если же мы используем второй конструктор:

```
main = putStrLn $ show $ Host "127.0.0.1"
```

вывод будет таким:

```
localhost
```

О нульарных конструкторах

Нульарный конструктор (nullary constructor) используется в Haskell-коде довольно часто. Суть его проста.

Вспомним наш демонстрационный тип:

```
data IP_address = IP String
```

За конструктором `IP` следует одно значение, поэтому такой конструктор называют одинарным. А нульарным называют такой конструктор, за которым никакого значения нет.

Допустим, нам нужен тип, ассоциированный с протоколами транспортного слоя (Transport layer) модели OSI. Вот как это будет выглядеть:

```
data Transport_layer = TCP | UDP | SCTP | DCCP | SPX
```

Символ `'|'` вы уже знаете, но значений, как видите, здесь нет. Тип `Transport_layer` имеет пять нульарных конструкторов. И чтобы продемонстрировать вам практическую пользу такого типа, разберём пример.

Определим функцию, описывающую конкретный протокол:

```
description_of :: Transport_layer -> String
description_of protocol =
    case protocol of
        TCP  -> "Transmission Control Protocol"
        UDP  -> "User Datagram Protocol"
        SCTP -> "Stream Control Transmission Protocol"
        DCCP -> "Datagram Congestion Control Protocol"
        SPX  -> "Sequenced Packet Exchange"
```

Мы принимаем значение типа `Transport_layer` и возвращаем соответствующее ему строковое описание. Обратите внимание на конструкцию с ключевым словом `case`. Это — родственница конструкции `switch-case` из языка C.

Теперь проверяем:

```
main = print $ description_of TCP
```

Для создания значения типа `Transport_layer`, к которому будет применена функция `description_of`, мы использовали один из нульарных конструкторов. Никакого дополнительного значения здесь нет, потому что сам нульарный конструктор представляет собой значение.

Результат такой:

```
"Transmission Control Protocol"
```

Если вам нужен тип, подразумевающий фиксированное множество именованных значений — нульарные конструкторы вам помогут.

Контекст типа

Как вы уже знаете, тип аргумента функции может быть полиморфным, что позволит нам применить такую функцию к значению *любого* типа. Но на практике нам редко нужна столь радикальная гибкость, ведь в большинстве случаев, говоря об аргументе *любого* типа, мы подразумеваем вовсе не *любой* тип, а тип из некоторой группы. Об этом и поговорим.

Любой, да не совсем

Рассмотрим стандартную функцию `elem`. Это простой предикат, возвращающий `True` в том случае, если заданное значение является элементом списка. Например:

```
main = print $ if "yellow" `elem` colors
               then "Yello is here!"
               else "There's no yellow..."
       where colors = ["red",
                       "black",
                       "yellow",
                       "green"]
```

Предельно простая функция. Взглянем на её объявление:

```
elem :: Eq a => a -> [a] -> Bool
```

Тут вроде бы всё понятно: принимаем значение некоторого типа `a`, а также список из значений этого же типа, и возвращаем логический признак.

И всё-таки кое-что новенькое тут присутствует, а именно символ `=>`. Этот символ и говорит нам о наличии контекста типа:

```
Eq a => a
```

Слева от символа `=>` находится ограничение, наложенное на полиморфный тип `a`. Мы говорим: «Да, мы готовы принять аргумент *любого* типа, но при условии, что этот тип относится к классу типов `Eq`.»

Стандартный класс типов `Eq` (от английского equal, «равный») предоставляет два метода:

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

Именно поэтому функция `elem` может быть применена лишь к значению такого типа, который предоставляет свои реализации методов «равно» `(==)` и «не равно» `(/=)`. Оно и понятно: далеко не для всех типов операция проверки на равенство имеет смысл.

Проверим это на нашем типе `IP_address`:

```
nothing :: a -> a
nothing val = val

main = print $ nothing $ IP "127.0.0.1"
```

У нас есть функция `nothing`, которая ничего не делает, просто возвращает свой аргумент. Поскольку она объявлена с полиморфным типом без каких-либо ограничений, мы можем применить эту функцию даже к значению нашего типа `IP_address`.

Но как только мы добавим в объявление этой функции ограничение:

```
nothing :: Eq a => a -> a
nothing val = val
```

функция `nothing` уже не согласится работать с нашим типом, и правильно сделает: наш тип не имеет никакого отношения к классу `Eq`. В этом случае говорят, что тип `IP_address` не подходит под ограничение класса `Eq`, или не входит в контекст класса `Eq`.

Механизм ограничения класса типа — полезный механизм, к тому же делающий наш код самодокументируемым, ведь видя ограничение типа в объявлении функции, мы сразу же понимаем, каким характеристикам должен соответствовать тип аргумента.

Множественность

Контекстов может быть и несколько. В этом случае мы перечисляем требуемые классы типов в виде кортежа:

```
nothing :: (Show a, Eq a) => a -> a
nothing val = val
```

Здесь мы требуем, чтобы тип аргумента входил как в контекст класса `Show`, так и в контекст класса `Eq`.

Более того, мы можем указать разные ограничения для разных полиморфных типов. Если бы наша функция `nothing` принимала два аргумента, её объявление могло быть таким:

```
nothing :: (Show a, Show b, Eq b) => a -> b -> String
nothing val1 val2 = (show val1) ++ (show val2)
```

Тип первого аргумента должен входит в контекст класса `Show`, в то время как тип второго должен входить в контексты сразу двух классов, `Show` и `Eq`.

Готово. Теперь вы знаете о контексте типов. Однако чуть позже вы узнаете о нём кое-что ещё.

Составные типы

Вспомним наш демонстрационный тип `IP_address`, который мы использовали до сих пор:

```
data IP_address = IP_address String
```

Будем честны: реальная польза от такого типа невелика. Да, мы можем кое-что с ним сделать, например, определить экземпляр класса `Show` и вывести значение этого типа на экран. Ну и что с того? Фактически, этот тип представляет собою именную обёртку вокруг одной-единственной строки. Так *ли* нам нужна строка с именным ярлыком `IP_address`? Едва *ли*. В реальных проектах нам понадобятся более интересные типы.

Поля

Определим тип User:

```
data User = User { first_name :: String
                 , last_name :: String
                 , email :: String
                 }
```

Это уже значительно интереснее. Перед нами — классический составной тип. Он может выглядеть очень похожим на структуру в языке C, но на деле это не совсем так. Рассмотрим вот эту конструкцию:

```
first_name :: String
```

Перед нами — поле. Впрочем, при кажущейся простоте, оно как шляпа фокусника, с двойным дном. С одной стороны, это значение типа `String`. С другой стороны — это (автоматически создаваемая компилятором) функция, позволяющая получить доступ к этому значению. Поэтому при создании значения типа `User` мы сначала инициализируем все три его поля, а потом сможем получить доступ к значениям этих полей. Вот так:

```
main =
    print $ first_name user ++ " " ++
            last_name user ++ ", " ++
            email user
    where user = User { first_name = "Denis"
                      , last_name = "Shevchenko"
                      , email = "me@dshevchenko.biz"
                      }
```

Вывод:

```
"Denis Shevchenko, me@dshevchenko.biz"
```

Применяя каждую из «полевых» функций к значению типа `User`, мы получаем доступ к соответствующему «полевому» содержимому, заданному при инициализации. Разумеется, все три значения константны.

Что с ними можно делать

Как обычно: создавать, сохранять, использовать, изменять.

Стоп... Я сказал «изменять»? Так значит, поле можно изменить??

Конечно нет. Когда мы пишем:

```
email = "me@dshevchenko.biz"
```

мы тем самым объявляем: «Всё! С этого мгновения и до конца времён функция `email`, применённая к этому значению типа `User`, будет возвращать значение `"me@dshevchenko.biz"` и никакое другое.» Однако есть один способ, создающий *впечатление* изменяемости поля.

Смотрите:

```
change_email :: User -> String -> User
change_email user new_email = user { email = new_email }
```

Эта функция меняет текущее значение поля `email` на значение `new_email`, используя тот же синтаксис фигурных скобок, что и при создании `user`. Однако, сказав об отсутствии присваивания в Haskell, я не обманул вас. Поэтому значение поля `email` у аргумента user не изменится.

Обратите внимание на сигнатуру этой функции:

```
change_email :: User -> String -> User
```

Мы возвращаем значение типа `User`, но это уже не то значение, которое идёт первым аргументом. Здесь мы говорим: «Возьми аргумент `user` и создай на его основе такое же значение, но лишь с тем исключением, что значение поля `email` в этом новом значении будет равным `new_email`.»

Укороченная запись типов полей

Мы должны явно указывать типы всех наших полей. Однако если эти типы одинаковы, можно использовать сокращённую запись.

Вспомним наш тип:

```
data User = User { first_name :: String
                 , last_name :: String
                 , email :: String
                 }
```

Все три поля имеют одинаковый тип, мы можем переписать объявление так:

```
data User = User { first_name
                 , last_name
                 , email :: String
                 }
```

Перечисляем поля и в конце указываем их тип. Кроме того, поля могут объединяться и по нескольким «типовым группам»:

```
data User = User { first_name
                 , last_name
                 , email :: String
                 , account
                 , uid :: Integer
                 }
```

Здесь у нас появились два новых поля, и мы опять использовали сокращённую типовую запись: группа из первых трёх полей имеет тип `String`, в то время как группа из полей `account` и `uid` имеет тип `Integer`.

Конструктор типа

Помните, мы уже говорили о конструкторах? Но там речь шла о конструкторах значений, а теперь поговорим о конструкторах типов. Различаются они так: конструктор значения используется при создании значений, а конструктор типа — при создании типов.

Добавим в тип `User` чуток гибкости:

```
data User year = User { first_name             -- строка
                      , last_name              -- опять строка
                      , email :: String        -- ещё одна строка
                      , year_of_birth :: year  -- а это что??
                      , account                -- целое число
                      , uid :: Integer         -- тоже целое число
                      }
```

Вот эта строка:

```
data User year = ...
```

говорит нам о том, что перед нами конструктор типа (type constructor). Тип поля `year_of_birth` задан полиморфным типом `year`[1]. Это позволит нам инициализировать это поле как числом `1981`, так и, например, строкой `"1981"`.

Однако нас поджидает один неприятный сюрприз. Теперь функция `change_email` наотрез откажется работать с типом `User`. Но не ругайтесь на неё, она поступает абсолютно правильно. Вспомним её объявление:

```
change_email :: User -> String -> User
```

Она ожидает, что первым аргументом будет значение типа `User`, но ведь `User` — это уже не тип, а конструктор типа. Теперь мы должны любезно попросить его сконструировать для нас конкретный тип. Чтобы это сделать, мы должны применить конструктор типа к типу. Если функция применяется к значению, то конструктор типа применяется непосредственно к типу. Прямо так и пишем:

```
User String
```

В результате применения конструктора `User` к типу `String` был создан тип, аналогичный такому:

1 Да-да, имя полиморфного типа можно задавать не только одной буквой.

```
data User = User { first_name
                 , last_name
                 , email
                 , year_of_birth :: String
                 , account
                 , uid :: Integer
                 }
```

Больше нет никакого полиморфного типа `year`, потому что тип поля `year_of_birth` теперь равен типу `String`.

Следовательно, чтобы наша функция `change_email` смогла работать с типами, сконструированными конструктором `User`, это нужно явно указать в её объявлении:

```
change_email :: User String -> String -> User String
```

Можно написать и так:

```
change_email :: (User String) -> String -> (User String)
```

Мы говорим нашей функции: «Первым аргументом ты теперь принимаешь значение типа, сконструированного путём применения конструктора `User` к типу `String`. Возвращаешь то же самое хозяйство.»

Разумеется, мы можем воспользоваться полиморфным типом и здесь:

```
change_email :: (User a) -> String -> (User a)
```

Теперь мы говорим этой функции: «Теперь в качестве аргумента ты принимаешь значение типа, который был создан конструктором `User`, применённым к некоторому типу `a`.» Такой подход удобен в том случае, если мы не знаем заранее, какой конкретный тип будет подставлен вместо полиморфного.

Всё. О составных типах вы теперь знаете.

Наследуемые типы

Как вы помните, вывести на экран значение нашего типа `IP_address` мы смогли лишь после того, как определили собственный экземпляр класса типов `Show`:

```
data IP_address = IP String

instance Show IP_address where
    show (IP address) =
        address
```

Однако существует ещё один способ обеспечить «печатаемость» нашего `IP_address`.

Наследуем

На сцену выходит ключевое слово `deriving`. Перепишем определение нашего типа:

```
data IP_address = IP String
                  deriving Show
```

Всё. Мы можем сразу напечатать наше значение:

```
main = print $ IP "127.0.0.1"
```

Вывод:

```
IP "127.0.0.1"
```

Готово. Никаких экземпляров. Мы просто определили наш тип как *наследуемый* от класса `Show`. Именно поэтому нам не нужно определять собственную версию метода `show`, ведь компилятор уже сделал это за нас.

Вас, вероятно, интересует, откуда компилятор знает, *как* нужно выводить на экран значение нашего типа? А он этого и не знает, поэтому идёт по пути наименьшего сопротивления. Обратите внимание на вывод:

```
IP "127.0.0.1"
```

Фактически, наш объект явно «стрингифицировался» в том же виде, в каком и был создан.

Вспомним наш составной тип и сделаем его «печатаемым»:

```
data User = User { first_name
                 , last_name
                 , email
                 , year_of_birth :: String
                 , account
                 , uid :: Integer
                 } deriving Show

main =
    print (user)
    where user = User { first_name = "Denis"
                      , last_name = "Shevchenko"
                      , email = "me@dshevchenko.biz"
                      , year_of_birth = "27.01.1981"
                      , account = 1234567890
                      , uid = 123
                      }
```

Вывод будет таким:

```
User {first_name = "Denis", last_name = "Shevchenko", email =
"me@dshevchenko.biz", year_of_birth = "27.01.1981", account = 1234567890, uid
= 123}
```

Прямая «стрингификация», как создали — так и получили.

Кстати, наследоваться можно только от нескольких классов. В соответствии со стандартом Haskell 2010 к таковым относятся: `Eq`, `Ord`, `Enum`, `Bounded`, `Read` и `Show`. Рассмотрим, что сделает с нашим типом наследование от этих классов.

Eq и Ord

Наследование от этих двух классов позволит нам сравнивать объекты нашего типа на (не)равенство, а также по признаку больше/меньше. То есть к объекту нашего типа можно будет применять следующие стандартные функции: `(==)`, `(/=)`, `compare`, `(<)`, `(<=)`, `(>)`, `(>=)`, `max`, `min`.

Эти классы — братья-близнецы: если наследуетесь от одного, то скорее всего нужно и от второго. Да, я ведь не сказал: вы можете наследоваться от нескольких классов одновременно. В этом случае они перечисляются в виде кортежа:

```
data IP_address = IP String
                  deriving (Eq, Ord)
```

Enum

Наследование от `Enum` сделает объекты нашего типа перечисляемыми. Однако этот тип должен иметь только нульарные конструкторы.

Вспомним наш "протокольный" тип и описательную функцию к нему:

```
data Transport_layer = TCP | UDP | SCTP | DCCP | SPX

description_of :: Transport_layer -> String
description_of protocol =
    case protocol of
        TCP  -> "Transmission Control Protocol"
        UDP  -> "User Datagram Protocol"
        SCTP -> "Stream Control Transmission Protocol"
        DCCP -> "Datagram Congestion Control Protocol"
        SPX  -> "Sequenced Packet Exchange"
```

Поработаем со списком протоколов:

```
main = print [description_of protocol | protocol <- [TCP, UDP]]
```

Вывод:

```
["Transmission Control Protocol","User Datagram Protocol"]
```

Здесь мы использовали нашего старого друга, list comprehension, чтобы пройтись по всем элементам списка протоколов и вернуть список с соответствующими описаниями.

Но что мы будем делать, если захотим получить описание всех протоколов транспортного уровня? Нам придётся вручную указывать все пять. Ничего страшного в этом нет, однако если бы это были протоколы физического уровня, то их было бы уже порядка двадцати. Писать их вручную — скучно. Но есть у нас один инструмент, позволяющий создать список малыми усилиями. Речь идёт о диапазонах. Вот тут-то и выходит на сцену класс `Enum`.

Наследуем от него наш тип:

```
data Transport_layer = TCP | UDP | SCTP | DCCP | SPX
                       deriving Enum
```

и теперь мы можем использовать его так:

```
main = print [description_of protocol | protocol <- [TCP ..]]
```

Здесь мы использовали бесконечный диапазон, указав лишь первый из протоколов. В результате наш список включит в себя все имеющиеся значения типа `Transport_layer`, и вывод будет таким:

```
["Transmission Control Protocol","User Datagram Protocol","Stream Control
Transmission Protocol","Datagram Congestion Control Protocol","Sequenced
Packet Exchange"]
```

Обращаю ваше внимание на маленькую деталь:

```
[TCP ..]
```

Видите пробел между именем протокола и двумя точками? Он обязателен. Если уберёте — компилятор выразит своё несогласие.

Bounded

Когда мы наследуем наш тип от класса `Bounded`, мы получаем возможность применять к нашему типу две стандартные функции, `minBound` и `maxBound`. Обратите внимание: эти функции применяются именно к типу, а не к значению, и возвращают они минимальное и максимальное значение данного типа.

Например:

```
main = print $ "minimal Int value: " ++ show (minBound :: Int) ++
               ", maximum Int value: " ++ show (maxBound :: Int)
```

Вывод будет таким:

```
"minimal Int value: -9223372036854775808, maximum Int value:
9223372036854775807"
```

И самое интересное, что мы можем применять эти две функции и к нашим собственным типам. Сделаем это с нашими протоколами:

```
data Transport_layer = TCP | UDP | SCTP | DCCP | SPX
                       deriving (Show, Enum, Bounded)

main = print $ "first protocol: " ++
               show (minBound :: Transport_layer) ++
               ", last protocol: " ++
               show (maxBound :: Transport_layer)
```

Вывод:

```
"first protocol: TCP, last protocol: SPX"
```

Мы применили эти функции к нашему перечисляемому типу, и они вернули, соответственно, "наименьшее" (первое по счёту) и "наибольшее" (последнее по счёту) значения этого типа.

Read и Show

Эти два класса наделяют значение нашего типа диаметрально противоположными способностями. `Show`, как вы уже знаете, позволяет представлять значение в виде строки, а `Read`, напротив, позволяет извлекать объект из строки. Ничего не напоминает? Ведь это сериализация. `Show` даёт возможность сериализовать объект в строку, а `Read` — десериализовать его из этой строки.

Например:

```
data User = User { first_name
                 , last_name
                 , email
                 , year_of_birth :: String
                 , account
                 , uid :: Integer
                 } deriving (Show, Read, Eq)

main =
    let object = user
        serialized_object = show object
        deserialized_object = read serialized_object
    in
    print $ object == deserialized_object -- Объекты равны? Не сомневайтесь!
    where user = User { first_name = "Denis"
                      , last_name = "Shevchenko"
                      , email = "me@dshevchenko.biz"
                      , year_of_birth = "27.01.1981"
                      , account = 1234567890
                      , uid = 123
```

Теперь вы знаете, что такое `deriving`. Открою вам секрет: наследоваться можно не только от шести вышеперечисленных классов, но и от нескольких других. Однако это касается довольно-таки редких случаев, поэтому мы не будем их рассматривать.

Собственные классы типов

До этого мы рассматривали лишь стандартные классы типов. А теперь поговорим о собственных классах типов.

Перцы

Объявим класс типов `Pepper` (перец):

```
type SHU = Integer  -- SHU (Scoville Heat Units), единица жгучести перца

class Pepper a_pepper where
    color :: a_pepper -> String
    pungency :: a_pepper -> SHU
```

У этого класса два метода, `color` (цвет) и `pungency` (жгучесть). Создадим два разных перца:

```
data Poblano = Poblano  -- распространён в национальных блюдах Мексики
data TrinidadScorpion = TrinidadScorpion  -- самый жгучий перец в мире

instance Pepper Poblano where
    color Poblano = "green"
    pungency Poblano = 1500

instance Pepper TrinidadScorpion where
    color TrinidadScorpion = "red"
    pungency TrinidadScorpion = 855000
```

Теперь мы можем работать с этими перцами как обычно:

```
main =
    putStrLn $ show (pungency trinidad) ++ ", " ++ color trinidad
    where trinidad = TrinidadScorpion
```

Зачем они нужны

В самом деле, разве мы не можем использовать каждый из типов перца самостоятельно?

Конечно можем. Главная цель определения собственного класса типов — указание контекста типов. Определим функцию, выводящую информацию о конкретном перце:

```
pepper_info :: Pepper a_pepper => a_pepper -> String
pepper_info a_pepper =
    show (pungency a_pepper) ++ ", " ++ color a_pepper
```

Контекст полиморфного типа `a_pepper` говорит нам о том, что эта функция предназначена только для работы с перцами. Более того, сам класс тоже может иметь контекст типа:

```
class Pepper a_pepper => Chili a_pepper where
    kind :: a_pepper -> String
```

Класс типов `Chili` объявлен с контекстом `Pepper`. Следовательно, занять место полиморфного типа `a_pepper` смогут лишь те типы, которые относятся к классу `Pepper`, и никакие другие. Это позволит нам ограничить набор типов, которые смогут иметь отношение к классу `Chili`.

Разумеется, контекст может состоять и из нескольких классов. В этом случае они, как обычно, перечисляются в виде кортежа:

```
class (Pepper a_pepper, Capsicum a_pepper) => Chili a_pepper where
    kind :: a_pepper -> String
```

Константы

Добавим в наш класс константу:

```
type SHU = Integer

class Pepper a_pepper where
    simple :: a_pepper       -- это константное значение, а не функция
    color :: a_pepper -> String
    pungency :: a_pepper -> SHU
    name :: a_pepper -> String
```

```
data Poblano = Poblano String   -- унарный конструктор вместо нульарного

instance Pepper Poblano where
    simple = Poblano "ancho"      -- готовим простое значение
    color (Poblano name) = "green"
    pungency (Poblano name) = 1500
    name (Poblano name) = name

main =
    putStrLn $ name (simple :: Poblano)  -- обращаемся к простому значению
```

`simple` — это константное значение, к которому можно обращаться напрямую. Мы говорим: «Каждый тип, относящийся к классу `Pepper`, обязан предоставлять константу `simple` своего собственного типа.» Поэтому при определении экземпляра класса `Pepper` мы готовим это константное значение:

```
simple = Poblano "ancho"
```

Поскольку теперь конструктор значения `Poblano` у нас унарный, он принимает строку (например, название перца или ассоциативного блюда). Здесь мы говорим: «Когда кто-нибудь обратится к константе `simple`, принадлежащей типу `Poblano`, он получит такое значение, как если бы мы написали просто `Poblanlo "ancho"`.» Таким образом, эту константу можно рассматривать как конструктор по умолчанию для значения типа `Poblano`. И это весьма удобно: если для нашего типа имеет смысл некоторое умолчальное значение, лучше задавать его прямо внутри нашего типа.

Всё. Теперь вы знаете о пользовательских классах типов.

Новый тип

Помимо ключевого ключевого `data` существует ещё одно слово, предназначенное для определения нового типа. Оно так и называется — `newtype`. И между этими двумя словами есть несколько важных отличий.

Один конструктор значения

Тип, определяемый с помощью слова `newtype`, может иметь один и только один конструктор значения. Мы можем написать так:

```
newtype IP_address = IP String
                     deriving Show
```

а вот такой код будет категорически отвергнут компилятором:

```
newtype IP_address = IP String | Host String
                     deriving Show
```

Одно поле

Следующее ограничение: одно и только одно поле. Мы можем написать так:

```
newtype IP_address = IP String
                     deriving Show
```

или так:

```
newtype IP_address = IP { value :: String }
                     deriving Show
```

А вот такой код не будет принят компилятором:

```
newtype IP_address = IP String Int
                     deriving Show
```

Тип с нульарным конструктором тоже не пройдёт компиляцию:

```
newtype Color = Red
```

Для чего он нужен

Вы спросите, зачем же нужно слово `newtype`, если с ним связаны такие ограничения? И чем вообще обусловлены эти ограничения?

Фундаментальное назначение `newtype`, строго говоря, не в том, чтобы создавать новый тип, а в том, чтобы оборачивать один, уже существующий тип. Именно поэтому оно требует унарного конструктора значения и никакого иного. Грубо говоря, ключевое слово `newtype` может рассматриваться как нечто среднее между словами `data` и `type`. И это даёт нам одно преимущество, а именно эффективность времени выполнения.

Если мы определили вот такой тип:

```
data IP_address = IP String
```

то с точки зрения программиста мы всего лишь обернули строковое значение в именную обёртку. Однако с точки зрения компилятора мы создали совершенно новый тип, хранящий в себе значение стандартного типа `String`. Именно поэтому работа с такой именной обёрткой связана с дополнительными накладными расходами на стадии выполнения (обусловленными «оборачиванием» и «разворачиванием» той самой внутренней строки). Да, эти расходы крошечны, но всё же…

Если же мы написали так:

```
newtype IP_address = IP String
```

мы сказали компилятору: «`IP_address` — это всего лишь именная обёртка вокруг стандартной строки. Именно так её и воспринимай, и никаких лишних телодвижений не делай». Поэтому работа с таким типом будет чуток более эффективной.

Вот и всё. Теперь вы знаете: если нужно создать новый стиль «с нуля» — используйте `data`, если же нужна обыкновенная именная обёртка вокруг одного-единственного значения существующего типа — `newtype` к вашим услугам.

Часть 6 Ввод и вывод

Функции с побочными эффектами

О чистых функциях вы уже знаете. Пришла пора поговорить о функциях, имеющих побочные эффекты.

Чистота vs нечистота

Как вы помните, чистые функции не имеют побочных эффектов, что делает их отражением математического понятия «функция»: полученное на входе однозначно определяет то, что будет на выходе. И всё бы замечательно, но наше приложение обязано взаимодействовать с внешним миром, а чистые функции никак не подходят на роль «послов во внешний мир».

Допустим, у нас есть функция чтения текста из файла: она принимает строку с путём к этому файлу, а возвращает строку с содержимым файла:

```
read_file :: String -> String
read_file path =
    -- открываем соответствующий файл,
    -- читаем его и возвращаем строку с его содержимым...
```

Однако такая функция не может быть чистой. Если мы два раза применим её к строке с путём к одному и тому же файлу, можем ли мы гарантировать, что возвращённое содержимое будет одним и тем же? Конечно же нет, ведь между первым и вторым применениями этот файл мог быть трижды изменён. В этом случае математичность такой функции лопнет как мыльный пузырь: дважды применили к одному и тому же аргументу, а результат на выходе получили разный.

Именно поэтому взаимодействие с внешним миром — это царство функций с побочными эффектами.

Действие vs бездействие

Чистая функция — это мир бездействия. Мир спокойствия и тишины. Как вы уже знаете, такая функция состоит из совокупности выражений, которые вычисляются в некотором порядке и в конце концов оставляют некое последнее, ито-

говое значение, которым компилятор просто заменяет место вызова этой функции.

А вот функции с побочными эффектами — это мир действия. Мир, в котором всё меняется. Именно поэтому при работе с вводом и выводом нам понадобятся действия (actions). Действие — это то, что соприкасается с внешним миром, а зачастую ещё и оказывает на него влияние. Нужно прочитать файл? Добро пожаловать во внешний мир. Нужно отправить UDP-дейтаграмму? Вам во внешний мир. Нужно прочесть строку, введённую пользователем с клавиатуры? Внешний мир ждёт вас.

IO a

Итак, для работы с внешним миром нам нужны действия. А действие представляет собой значение типа `IO a`, где `IO` — это стандартный тип действия, а `a` — это полиморфный тип значения, возвращённого этим действием. Как вы уже поняли, `IO` — это конструктор типа. Поэтому тип действия, возвращающего строку, такой: `IO String`.

С логической точки зрения, действие — это наш посол, который по нашей просьбе уходит во внешний мир, делает там какую-то работу, а потом приносит нам из внешнего мира что-нибудь интересное. Впрочем, иногда он может вернуться из внешнего мира и с пустыми руками.

Стандартные ввод и вывод

Начнём со стандартных каналов `stdout` и `stdin`. Выведем строку на экран:

```
main = putStrLn "Hi Haskeller!"
```

Взглянем на объявление функции `putStrLn`:

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

Перед нами `IO`, а значит, данная функция имеет побочное действие. На входе у нас строка, а на выходе — действие. Рассмотрим его поближе:

```
IO ()
```

Одинокие круглые скобки говорят нам о пустом кортеже. Следовательно, перед нами действие, которое, сделав во внешнем мире свою работу, ничего нам не принесёт. Мы, посылая это действие во внешний мир, говорим ему: «Пойди и просто напечатай на экране компьютера переданную тебе строку.» Но раз уж создатели Haskell договорились о том, что действие обязано *что-то* вернуть — пусть оно возвращает пустой кортеж. Это как `void`-функция в языке C: можно сказать, что она не возвращает ничего, а можно сказать, что она возвращает `void`.

Теперь взглянем на объявление функции `getLine`, получающей строку со стандартного ввода:

```
getLine :: IO String
```

Тут противоположная ситуация. Эта функция ничего не принимает от нас, потому что нам нечего ей дать, и порождённое этой функцией действие идёт во внешний мир с пустыми руками. Мы говорим ему: «Пойди, получи со стандартного ввода строку и принеси её нам.»

Объявляем `main`

Теперь мы наконец-то можем взглянуть на её объявление:

```
main :: IO ()
```

Функция `main` тоже совершает действие — работу всего нашего приложения. Понятно, что она ничего не возвращает в приложение, ведь при её завершении заканчивается всё. Разумеется, все действия в нашем приложении спят крепким сном и ничего не делают до тех пор, пока не будет запущено действие функции `main`.

Мы до сих пор не писали объявление этой функции, потому что только сейчас узнали об `IO`. Но, строго говоря, мы должны это делать. Хотя бы для порядка.

Совместная работа

Вот она:

```
main :: IO ()
main = do
    putStrLn "Input your text, please:"
    line_from_user <- getLine
    putStrLn $ "Not bad: " ++ line_from_user
```

Тут всё предельно понятно, за исключением двух новых вещей.

Во-первых, обратная стрелочка `'<-'`. Взглянем на неё:

```
line_from_user <- getLine
```

Это — ассоциация. Мы говорим действию, порождённому функцией `getLine`: «Пойди, получи введённую пользователем строку, принеси её нам и привяжи (bind) её к идентификатору `line_from_user`, чтобы мы смогли прочесть эту строку.»

Вторая новизна в этом коде — ключевое слово `do`. И о нём стоит поговорить отдельно.

do: императивный мир

Вы прочли правильно: императивный мир. Несмотря на то, что Haskell является чисто функциональным языком, в случае необходимости мы *можем* написать императивный код. А при работе с внешним миром необходимость такая возникает постоянно.

Как вы знаете, императивный подход подразумевает выполнение программных инструкций в чётко указанном порядке. В чистых функциях такой подход излишен, потому что в них неважен порядок вычисления выражений. Однако в функциях, взаимодействующих с внешним миром, ситуация кардинально меняется.

Вспомним наш пример работы со стандартными вводом и выводом:

```
main :: IO ()
main = do
    putStrLn "Input your text, please:"
    line_from_user <- getLine
    putStrLn $ "Not bad: " ++ line_from_user
```

Здесь мы делаем три шага:

1. выводим на экран приветственную строку;
2. дожидаемся, пока пользователь введёт свой текст;
3. выводим на экран итоговую строку.

Разумеется, мы ожидаем, что эти три шага будут выполнены именно в таком порядке. Согласитесь, было бы странно выводить на экран итоговую строку, не дождавшись введённого пользователем текста. При работе с внешним миром мы всегда подразумеваем определённый порядок наших шагов. Например, сервер, получив запрос от клиента, должен сначала его обработать, потом сформировать ответ, и только потом отправить его клиенту.

Именно для этой цели и введено ключевое слово `do`: оно связывает наши действия в последовательную цепочку. Говоря об этом ключевом слове, обычно используют термин «`do`-нотация».

Не только `main`

Мы можем использовать `do`-нотацию в *любой* функции с побочными эффектами. Например:

```
obtain_user_text :: String -> IO String
obtain_user_text prompt = do
    putStrLn prompt  -- Выведи приглашение ввести строку
    getLine          -- Получи от пользователя некую строку

main :: IO ()
main = do
    first_text <- obtain_user_text "Enter your text, please: "
    second_text <- obtain_user_text "One more, please: "
    putStrLn $ "You said '" ++ first_text ++ "' and '" ++ second_text ++ "'"
```

Функция `obtain_user_text` заключает в себе два последовательных шага, поэтому в ней тоже используется слово `do`. Мы ожидаем, что сначала будет выведено соответствующее приглашение на экран, и только после этого действие, порождённое функцией `getLine`, отправится во внешний мир и вернётся оттуда с введённой пользователем строкой.

О функции `return`

В языке C есть ключевое слово `return`, задающее точку возврата из функции. В Haskell нет такого ключевого слова, зато есть такая функция. И чтобы продемонстрировать её назначение, рассмотрим другой пример:

```
obtain_two_texts_from_user :: IO String
obtain_two_texts_from_user  = do
    putStrLn "Enter your text, please: "
    first_text <- getLine
    putStrLn "One more, please: "
    second_text <- getLine
    "'" ++ first_text ++ "' and '" ++ second_text ++ "'" -- простая строка??

main :: IO ()
main = do
    two_texts <- obtain_two_texts_from_user
    putStrLn $ "You said " ++ two_texts
```

Функция `obtain_two_texts_from_user` берёт на себя ответственность последовательно получить от пользователя два текста и вернуть составленную из них строку.

К сожалению, такой код не пройдёт компиляцию, ибо эта функция возвращает действие, однако последней по счёту инструкцией идёт вовсе не действие, а обыкновенная строка. Тут-то и приходит нам на помощь стандартная функция `return`.

Перепишем нашу функцию:

```
obtain_two_texts_from_user :: IO String
obtain_two_texts_from_user  = do
    putStrLn "Enter your text, please: "
    first_text <- getLine
    putStrLn "One more, please: "
    second_text <- getLine
    return $ "'" ++ first_text ++ "' and '" ++ second_text ++ "'"
```

Императивно выглядит, не правда ли? Только не забывайте, что с ключевым словом `return` в языке C такая запись не имеет ничего общего.

Функция `return` берёт значение и оборачивает его в действие, возвращающее это значение. В нашем случае мы передали ей строку, составленную из пользовательских текстов, а на выходе получили действие, возвращающее эту строку. Поэтому теперь наш код успешно скомпилируется.

Кстати, чтобы доказать вам, что функция `return` действительно не имеет никакого отношения к ключевому слову `return` в C-подобных языках, я позволю себе небольшое хулиганство:

```
obtain_two_texts_from_user :: IO String
obtain_two_texts_from_user  = do
    putStrLn "Enter your text, please: "
    first_text <- getLine
    putStrLn "One more, please: "
    second_text <- getLine
    return $ "'" ++ first_text ++ "' and '" ++ second_text ++ "'"
    putStrLn "And third text, please: " -- мы всё ещё продолжаем наш диалог!
    getLine
```

Это может сбить с толку программистов, имеющих опыт в императивном программировании, но, как уже и было сказано выше, функция `return` всего лишь оборачивает значение в действие, возвращающее это значение. Она не прерывает ход наших действий, и если за ней есть другие действия, они спо-

койно продолжат выполняться. Фактически, в этой хулиганской функции мы потеряем два первых пользовательских текста и вернём действие, которое принесёт нам только третий текст.

Готово. Теперь вы знаете о ключевом слове `do`.

Обработка исключений

Мы все стремимся создавать программные системы, свободные от ошибок. И всё же иногда они появляются, а значит, нам приходится иметь с ними дело. Поговорим про исключения, тем более что знать о них необходимо: многие пакеты из Hackage содержат код, кидающийся исключениями.

Нам понадобится модуль `Control.Exception`:

```
import Control.Exception
```

Этот стандартный модуль предназначен для кидания и ловли исключений, причём как стандартных, так и наших собственных. Важно отметить: мы можем *бросить* исключение как из чистых функций, так и из функций с побочными эффектами, однако *поймать* его в чистой функции мы не сможем.

Проблема с файлом

Чаще всего мы будем сталкиваться с исключениями, брошенными из функций, взаимодействующих с внешним миром. Канонический пример: мы хотим прочесть содержимое файла, который отсутствует:

```
main :: IO ()
main = do
    file_content <- readFile "Users/shevchenko/test.c"  -- неверный путь
    putStrLn file_content
```

Вывод будет таким:

```
Real: Users/shevchenko/test.c: openFile: does not exist (No such file or
directory)
```

Функция `readFile` бросила исключение, ведь сообщить о проблеме с файлом она может только так. Исключение, не найдя преград на своём пути, было поймано уже на самом верхнем уровне приложения, после чего сообщение об ошибке было выведено нам, а само приложение скоропостижно скончалось.

Ловим

Итак:

```
try_to_open_file :: FilePath -> IO String
try_to_open_file path =
    readFile path `catch` possible_errors
    where
        possible_errors :: IOException -> IO String
        possible_errors error = return $ show error

main :: IO ()
main = do
    file_content <- try_to_open_file "Users/shevchenko/test.c"
    putStrLn file_content
```

Теперь у нас есть функция `try_to_open_file`, которая открывает файл по заданному пути, но делает это осторожно, используя функцию `catch`. Как вы уже поняли, функция `catch`, определённая в модуле `Control.Exception`, умеет ловить исключения. Вот её объявление:

```
catch :: Exception e => IO a -> (e -> IO a) -> IO a
```

Функция принимает два аргумента: первым идёт `IO`-действие, вторым идёт функция-обработчик. Если действие бросило исключение, отражённое полиморфным типом `e`, оно, это исключение, будет передано обработчику.

Обратите внимание: тип исключения входит в контекст класса `Exception`, определённого в том же модуле `Control.Exception`. Любое исключение в вашем приложении обязано входить в контекст этого класса.

Для большей читабельности мы используем инфиксную запись функции `catch`, с которой гармонирует имя нашего обработчика:

```
readFile path `catch` possible_errors
```

Если при чтении файла возникла проблема, соответствующее исключение поступает на вход нашего обработчика:

```
possible_errors :: IOException -> IO String
possible_errors error = return (show error)
```

Обратите внимание и на то, что функция-обработчик объявлена и определена локально, прямо в теле функции `try_to_open_file`. В ряде случаев это удобно.

Обработчик принимает значение типа `IOException`. В теле обработчика мы «стрингифицируем» полученное исключение, а затем готовую строку, содержащую описание произошедшей ошибки, заворачиваем в действие `IO String`. Если же нас не устраивает уже имеющееся сообщение об ошибке — предоставим своё:

```
possible_errors error = return "Unable to open this file... Please check it."
```

Это сообщение и будет выведено на экран в случае возникновения проблемы.

Ловим наоборот

Перепишем нашу функцию:

```
try_to_open_file :: FilePath -> IO String
try_to_open_file path =
    handle possible_errors (readFile path)  -- то же самое, но наоборот...
    where
        possible_errors :: IOException -> IO String
        possible_errors error = return "Aaaaa!!! Please check file."
```

Мы заменили функцию `catch` на функцию `handle`. Никакой разницы между этими функциями нет, за исключением порядка следования аргументов: `catch` принимает обработчик вторым по счёту, а `handle` — первым. Таким образом, `catch` читабельнее в инфиксной форме, а `handle` — в простой. Так что выбирайте на вкус.

Пытаемся

Функция `try` использует иной подход. Вот пример:

```
main :: IO ()
main = do
    result <- try $ readFile path :: IO (Either IOException String)
    case result of
        Left exception -> putStrLn $ "Fault: " ++ show exception
```

```
        Right content -> putStrLn content
    where path = "Users/dshevchenko/test.c"
```

Разберём по полочкам.

Объявление функции `try` выглядит так:

```
try :: Exception e => IO a -> IO (Either e a)
```

Эта функция принимает наше `IO`-действие, а возвращает другое `IO`-действие, которое в свою очередь возвращает значение стандартного типа `Either e a`. `Either` — это конструктор типа, предназначенный для хранения одного из двух значений, каждое из которых соответствует хорошему результату или плохому. Обратите внимание, мы явно указали тип значения, возвращённого функцией `try`:

```
try $ readFile path :: IO (Either IOException String)
```

Мы сказали: «Пусть действие, возвращённое функцией `try`, вернёт нам значение типа `Either IOException String`, в котором будет лежать либо значение типа `IOException` (в случае, если при чтении файла что-то случилось), либо значение типа `String` с содержимым файла.»

Далее смотрим, получилось ли у нас:

```
case result of
    Left exception -> putStrLn $ "Fault: " ++ show exception
    Right content -> putStrLn content
```

Тип `Either` имеет два конструктора, `Left` и `Right`. В нашем случае это можно изобразить так:

```
Either IOException String
       |           |
       Left        Right
```

Используя эти два конструктора, мы можем понять, что же произошло. Конструкция `case-of` поможет нам сделать это. Мы говорим: «Если `result` соответствует левому значению — это значение типа `IOException`. Что-то пошло не так, выводим исключение на экран! Ну а если `result` соответствует правому значению — перед нами `String`. Всё прошло успешно, выводим на экран содержимое прочитанного файла.»

В чистом мире

Иногда нам нужно поймать исключение, брошенное из чистой функции. Например:

```
main :: IO ()
main = do
    result <- try $ 2 `div` 0 :: IO (Either SomeException Integer)
    case result of
        Left exception -> putStrLn $ "Fault: " ++ show exception
        Right value -> print value
```

Здесь мы попытались проверить результат деления числа 2 на ноль. К сожалению, этот код не пройдёт компиляцию. Ведь функция `try` ожидает на вход `IO`-действие, однако стандартная функция `div` чиста и возвращает обыкновенное число. Следовательно, нам нужен маленький трюк:

```
main :: IO ()
main = do
    result <- try $ evaluate $ 2 `div` 0 :: IO (Either SomeException Integer)
    case result of
        Left exception -> putStrLn $ "Fault: " ++ show exception
        Right value -> print value
```

Мы обернули вызов функции `div` в функцию `evaluate`. Теперь всё скомпилируется, и при запуске мы получим ожидаемое нами гневное сообщение:

```
Fault: divide by zero
```

Функция `evaluate` объявлена так:

```
evaluate :: a -> IO a
```

Эта функция играет роль адаптера: она как бы превращает результат в `IO`-действие, возвращающее этот результат. И после того, как функция `div` вернула нам обыкновенное число, функция `evaluate` обернула это число в действие, ожидаемое функцией `try`.

Почти готово. Но наше рассмотрение исключений не было бы полным без изучения наших собственных исключений.

Собственные исключения

До сих пор мы *лишь ловили* исключения. Теперь поговорим о том, как их бросать, а также о том, как создать свои собственные типы исключений.

Создаём

Определяем:

```
import Control.Exception
import Data.String.Utils
import Data.Typeable

type Repo = String

data InvalidRepository = InvalidRepository Repo
                       deriving (Show, Typeable)

instance Exception InvalidRepository
```

Подразумевается, что мы анализируем некий репозиторий, и если он некорректен, в дело вступает исключение `InvalidRepository`. Мы наследовались от двух классов, `Show` и `Typeable`. Сделать это необходимо, потому что наш тип обязан предоставить свой экземпляр класса типов `Exception`, а этот класс устанавливает контекст из этих двух классов.

Но нас поджидает одна неожиданная деталь:

```
instance Exception InvalidRepository
```

Перед нами — экземпляр класса типов `Exception`, но в этом экземпляре нет ни ключевого слова `where`, ни последующих реализаций соответствующих методов. Класс `Exception` содержит в себе два метода, но мы говорим: «Вот наш экземпляр класса типов `Exception`, но предоставлять реализации его методов мы не хотим.»

Бросаем

Для того, чтобы бросить исключение, используем функцию `throw`. Не забывайте, что даже в том случае, если исключение было брошено из чистой функции, поймать его мы сможем только в `IO`-функции.

Напишем:

```
extract_protocol :: String -> String
extract_protocol path =
    if path `starts_with` "git" || path `starts_with` "ssh"
    then takeWhile (/= ':') path
    else throw $ InvalidRepository path  -- протокол неверный, кидаем...
    where starts_with = \url prefix -> startswith prefix url

main :: IO ()
main = do
    result <- try $ evaluate $ extract_protocol "ss://ul@sch/proj.git"
                :: IO (Either SomeException String)
    case result of
        Left exception -> putStrLn $ "Fault: " ++ show exception
        Right protocol -> putStrLn protocol
```

Вывод будет таким:

```
Fault: InvalidRepository "ss://ul@sch/proj.git"
```

Мы пытаемся извлечь протокол из полного пути к репозиторию, и если там не то, что нам нужно, мы кидаем исключение, перехватывающееся функцией `try`.

Всё. Теперь мы можете создавать собственные типы исключений и кидаться ими на здоровье.

Часть 7 Деликатесы

Монады: суть

Произнесите это слово: «монада». Вам страшно? Нет?? Быть не может! Вам должно быть страшно. Все знают, что монады — это самое страшное и самое сложное что есть в языке Haskell!

Позвольте открыть вам тайну: ничего сложного в этих монадах нет.

Почему их так боятся

Термин «монада» (от греческого μονάς, «единица») пришло в Haskell из мира математики, а именно из теории категорий. Послушайте, как это звучит:

Монада может быть определена через общее понятие моноида в моноидальной категории. Монада над категорией K — это моноид в моноидальной категории эндофункторов End(K).

Вот поэтому их и боятся: с таким же успехом это определение могло быть написано на китайском. Лично я понял из него чуть меньше чем ничего. Да, я прекрасно понял бы это определение, если бы был знаком с той самой теорией категорий. К счастью, понять суть монад можно и без изучения этой теории.

Определение

Монады — это механизм, привносящий императивный подход в чисто функциональный язык. Вот и всё. Если нам нужно связать некие шаги в чёткую последовательную цепочку — значит нам нужен монадический механизм.

Впрочем, разве мы не говорили об императивном подходе, рассматривая `do`-нотацию и взаимодействия с внешним миром? Говорили. Потому что упомянутый в предыдущих главах тип `IO` имеет самое прямое отношения к монадам. А если вспомнить, что ни одно приложение на Haskell не может обойтись без взаимодействия с внешним миром, становится ясно: даже однострочное приложение уровня `"Hello World"` использует монадический механизм.

Иллюстрация

Если вы используете Unix-подобную операционную систему, откройте терминал и введите:

```
cd /usr/lib
ls | grep xml
```

Обратите внимание на вторую команду. Это — Unix-канал, связывающий две системные утилиты, а если точнее, он связывает не утилиты, а результаты работы этих утилит. Взгляните:

```
ls         |      grep xml
процесс 1  канал  процесс 2
```

Когда первый процесс, соответствующий системной утилите `ls`, выполнится, он вернёт текст, отображающий содержимое текущего каталога. Этот текст поступает на вход Unix-канала и затем, выходя из него, поступает на вход второго процесса, соответствующего системной утилите `grep`. Второй процесс фильтрует полученный текст по слову `"xml"` и возвращает некий результат.

Таким образом, Unix-канал создал цепочку из двух звеньев, на выходе из которой мы получаем итоговое «значение», явившееся результатом последовательного выполнения двух «вычислений». И эта цепочка характеризуется двумя важными свойствами: последовательность и изоляция.

Последовательность даёт нам твёрдую уверенность в том, что до тех пор, пока утилита `ls` не завершит свою работу, мы не перейдём к утилите `grep`.

А изоляция обеспечивает взаимодействие звеньев, ничего не знающих друг о друге. Утилита `ls` не догадывается о том, что результат её работы будет подан на вход утилите `grep`. Да и утилита `grep` остаётся в полном неведении о том, что имеет дело с результатом работы утилиты `ls`.

Взаимодействие двух этих утилит стало возможным только благодаря тому, что обе они имеют дело с общим «типом данных», а именно с текстом: утилита слева от канала *возвращает* текст, а утилита справа от канала *принимает* текст.

В этом и заключается суть монадического механизма: связать в последовательную цепочку вычисления, ничего друг о друге не знающие. Теперь вы понимаете, почему работа с вводом и выводом неразрывно связана с монадами, ведь `IO` — это и есть одна из монад.

А теперь перейдём к примерам.

Монады: на примере IO

Раз уж монады `IO` самые распространённые, и ни одно приложение не может без них обойтись, дальнейшие рассуждения о монадах продолжим на их примере.

Класс типов `Monad`

Все монады представлены в лице класса типов `Monad`. Вот из чего он состоит:

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a
```

А поскольку мы решили использовать в качестве примера монаду `IO`, то для наглядности заменим полиморфный тип `m` реальным:

```
class Monad IO where
    (>>=)  :: IO a -> (a -> IO b) -> IO b
    (>>)   :: IO a -> IO b -> IO b
    return :: a -> IO a
    fail   :: String -> IO a
```

Вспоминая пример с Unix-каналом, мы понимаем, что `IO`-действие — это монадическая обёртка для результатов функций, взаимодействующих с внешним миром.

Исследуем эти четыре метода.

Компоновка

Метод `(>>=)` — это оператор последовательной компоновки (sequentially composition). Иногда его ещё называют оператором связывания (bind). Именно этот оператор играет роль Unix-канала из нашей иллюстрации: он связывает

два IO-действия воедино, извлекая результат, возвращённый левым действием, и передавая его в качестве аргумента правому действию.

Рассмотрим объявление этого метода:

```
(>>=)  :: IO a -> (a -> IO b) -> IO b
```

Первый аргумент — это IO-действие, которое, выполнив свою работу, вернёт значение некоторого типа a. Второй аргумент — это функция, принимающая значение типа a в качестве аргумента и возвращающая IO-действие, которое, выполнив свою работу, вернёт значение некоторого типа b. А чтобы стало понятнее, позвольте мне прямо сейчас разоблачить do-нотацию.

Ключевое слово do — это всего лишь синтаксический сахар для монадических операторов. Возьмём простейших пример двух последовательных действий:

```
main :: IO ()
main = do
    text <- getLine
    putStrLn $ "You said '" ++ text ++ "'"
```

А вот как этот код выглядит «по-настоящему», без do-нотации:

```
main :: IO ()
main = getLine >>= \text -> putStrLn $ "You said '" ++ text ++ "'"
```

Теперь всё встало на свои места:

```
getLine    >>=  \text   -> putStrLn $ "You said '" ++ text ++ "'"

IO String  >>=  (String -> IO ())
```

Когда «запускается» функция getLine, возвращающая IO-монаду, содержащую полученную от пользователя строку, оператор (>>=) вытаскивает эту строку из монады и сразу же передаёт её в качестве аргумента λ-функции, «запускающей» функцию putStrLn, которая в свою очередь вернёт другую IO-монаду.

Это именно то, что мы наблюдали в нашем примере с Unix-каналом:

```
ls | grep xml
```

Функция getLine заняла место утилиты ls, а λ-функция заняла место утилиты grep.

Затем

Монадический оператор `(>>)` — это оператор «затем» (then). Это простейший случай связывания: действия связываются без извлечения значений.

Вот такое связывание с `do`-нотацией:

```
main :: IO ()
main = do
    putStrLn "Denis"
    putStrLn "Shevchenko"
```

А вот — без неё:

```
main :: IO ()
main = putStrLn "Denis" >> putStrLn "Shevchenko"
```

Именно поэтому этот оператор и называется «затем». Мы говорим: «Сначала выведи на экран имя, а *затем* — фамилию.» Никакой передачи значения здесь не происходит, ведь стоящая слева от оператора функция `putStrLn` возвращает пустое действие `IO ()`.

Если вспомнить аналогию с Unix-терминалом, то это можно изобразить так:

```
whoami ; pwd
```

Оператор связки команд (command concatenator), представленный точкой с запятой — это и есть подобие оператора «затем». Сначала запускается `whoami`, а затем запускается `pwd`, никакой передачи значения слева направо тут нет.

return

Метод `return` вам уже знаком. Теперь мы понимаем, что английское слово «return» хорошо отражает действие этой функции: она возвращает значение в монадическую обёртку. Вспомните наш пример:

```
obtain_text_from_user :: IO String
obtain_text_from_user = do
    putStrLn "Enter your text, please: "
    first_text <- getLine
    return $ "'" ++ first_text ++ "' and '"
```

Функция `getLine` вернёт нам монаду, из которой оператор компоновки вытащит введённую пользователем строку. Эта строка поступит на вход λ-функции, которая в свою очередь создаст новую строку на основе строки, введённой пользователем, после чего — внимание! — функция `return` *вернёт* эту новоиспечённую строку в `IO`-монаду. Вытащили значение из монады, что-то с ним сделали, а потом вернули в монаду.

Вы спросите, в чём же разница между `return` и упомянутой ранее `evaluate`? Разница в том, что функция `evaluate` заворачивает значение исключительно в монаду `IO`, в то время как функция `return` — в *любую* монаду, в зависимости от контекста. Рассматривайте `evaluate` как частный случай `return` для `IO`.

fail

О методе `fail` мы говорить не будем. Во-первых, этот метод не имеет отношения к концептуальной сути монад. А во-вторых, поговаривают, будто в стандарте Haskell 2014 этот метод будет вообще убран из класса `Monad`.

Вот и всё. Теперь вы знаете суть монад. В принципе, монадический механизм — это своего рода паттерн проектирования, унифицирующий процесс связывания вычислений.

Перейдём к примерам.

Монады: практика

Зачем же ради соединения вычислений в последовательную цепочку обматывать эти вычисления в какие-то монадические обёртки?

Главная цель такого подхода: гибкость и упрощение кода. Приступим.

Разоблачение списков

Списки в Haskell — это тоже монады. Рассмотрим, какую пользу нам может принести этот факт.

Начнём со строки. Она ведь есть ни что иное, как `[Char]`, а это значит, она тоже является монадой. Помните наш пример про исправление URL? Напишем нечто похожее:

```
import Data.Char

to_lower_case = \char -> return $ toLower char
underline_spaces = \char -> if char == ' ' then return '_' else return char

main :: IO ()
main =
    print $ name >>= to_lower_case >>= underline_spaces
    where name = "Lorem ipsuM"
```

На выходе у нас:

```
lorem_ipsum
```

Теперь проанализируем. Наше внимание приковывает вот эта строка:

```
name >>= to_lower_case >>= underline_spaces
```

В глаза бросаются операторы компоновки, а это значит, перед нами монадический конвейер. Слева в него въезжает наша строка `name`, то есть монада `[Char]`, и едет по нему. А за конвейером её ждут два работника, `to_lower_case` и `underline_spaces`, каждый из которых вносит в `name` свои изменения.

Вы спросите, в чём же тут соль? Чем это отличается от функций композиции и применения, рассмотренных нами ранее[1]? Ведь там мы тоже конструировали конвейер из трёх функций:

```
add_prefix . encode_all_spaces . make_it_lower_case $ url
```

Однако отличия имеются, и главное из них в том, что эти три функции работают со строкой, а функции `to_lower_case` и `underline_spaces` работают с *элементом* строки. Взгляните:

```
to_lower_case = \char -> return $ toLower char
```

Эта λ-функция ожидает на вход символ, а не строку. И это очень важное свойство функций, компонуемых в монадическую цепочку: они работают со значением, содержащимся в монаде, а не с самой монадой. Поскольку в данном случае монадой является `[Char]`, функция `to_lower_case` работает только с `Char`, извлекаемым из списка оператором компоновки. Понятно, что оператор компоновки, определённый для списка, подразумевает «прогон» монадной функции через все элементы этого списка.

Монадическая цепочка предоставляет нам большую гибкость, ведь функции `to_lower_case` и `underline_spaces` вообще не знают о том, что работают они в конечном итоге со строкой. А значит, эти функции можно соединять для работы с самыми разными монадами, содержащими в себе значение(я) типа `Char`.

Кстати, важное уточнение: функции такого рода могут работать только с монадами, о чём красноречиво говорит функция `return`. Взглянем на определение ещё раз:

```
to_lower_case = \char -> return $ toLower char
```

Функция говорит нам: «Да, я работаю с аргументом типа `Char`, но в конце своей работы я, с помощью функции `return`, возвращаю итоговое значение типа `Char` обратно в *какую-то* монаду.» А вот в какую именно — это уже неважно. Поэтому мы можем написать, например, так:

```
main :: IO ()
main =
    print $ name >>= to_lower_case
    where name = Just 'A'
```

1 В главе «Функциональные цепочки».

Кстати, чтобы это заработало, откройте файл `Real.cabal`, найдите в нём секцию `executable Real` и допишите в ней новый параметр:

```
    extensions:          NoMonomorphismRestriction
```

Функция `to_lower_case` остаётся в счастливом неведении о том, что теперь она фактически работает уже не со списком символов, а с монадой `Maybe`[2], содержащей в себе символ. И поэтому функция `return` в теле функции:

```
to_lower_case = \char -> return $ toLower char
```

завернёт итоговый символ уже не списочную монаду, а в монаду `Maybe`. Именно поэтому я и сказал выше, что функцию, задуманную для работы с одной монадой, можно использовать для работы с другими.

Меняем тип

Ещё один пример:

```
import Data.Char

to_real_numbers = \char -> return $ digitToInt char

main :: IO ()
main =
    print $ numbers >>= to_real_numbers
    where numbers = "1234567890"
```

На выходе нас ждёт:

```
[1,2,3,4,5,6,7,8,9,0]
```

Здесь произошло изменение типа монады. Функция `to_real_numbers` превращает символ в его цифровое представление. И вновь оператор компоновки и функция `return` сделали своё красивое дело: на вход была подана монада `[Char]`, а на выходе получили монаду `[Int]`. Таким образом, на протя-

2 О `Maybe` мы подробнее поговорим в следующей главе.

жении всей цепочки монадическая обёртка остаётся неизменной[3], а вот наполнение этой обёртки может изменять не только своё значение, но и свой тип.

Зеркальная компоновка

В стандартном пакете `Prelude` определён ещё один монадический оператор, который можно назвать «зеркальной компоновкой». Всё то же самое, но справа налево. Вот как это будет выглядеть в нашем примере:

```
main :: IO ()
main =
    print $ to_lower_case =<< underline_spaces =<< name
    where name = "Lorem ipsuM"
```

Мы просто развернули оператор компоновки наоборот, и теперь значение `name` заезжает в конвейер справа налево. Лично мне классический вариант кажется более удобным, так что выбор между обычной и зеркальной компоновкой — это вопрос эстетический.

Монадические цепочки — красивый и гибкий инструмент связки вычислений. Как мы смогли убедиться, аналогия с Unix-каналом оказалась весьма точной.

3 То есть `[Char]` никак не сможет превратиться, скажем, в `Maybe Char`.

Может быть

Есть в Haskell ещё один механизм обработки ошибок, не связанный ни с исключениями, ни с `IO`. Используется он весьма часто, поэтому знать о нём нужно. Речь идёт о стандартном типе `Maybe`.

Что за зверь

Тип `Maybe` — это опциональное значение. Оно говорит нам: «Может быть во мне есть некое реальное значение, а может быть и нету.» Вот его определение:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Ord)
```

Тип представлен двумя конструкторами: нульарным `Nothing` и унарным `Just`. Если значение было создано с конструктором `Nothing` — значит оно представляет собой пустышку, если же с конструктором `Just` — оно содержит в себе нечто полезное.

Для чего он

Простой пример:

```
import Data.Char

coefficient_from_string :: String -> Maybe Int
coefficient_from_string str =
    if isNumber first_char
    then Just (digitToInt first_char)
    else Nothing
    where first_char = str !! 0  -- извлекаем символ с индексом 0

check :: Maybe Int -> String
check a_coefficient
    | a_coefficient == Nothing = "Invalid string!"  -- коэффициент пустой!
    | otherwise = show a_coefficient
```

```
main :: IO ()
main = print $ check $ coefficient_from_string "0"
```

Мы пытаемся извлечь цифру из полученной строки, но что делать, если первым символом этой строки является нечисловой символ? Возвращать `-1`? Это не всегда приемлемо. Поэтому мы возвращаем значение типа `Maybe Int`, которое может содержать в себе извлечённую цифру, а может и не содержать. И уже в функции `check` мы проверяем, с каким конструктором было создано значение `a_coefficient`. Если с конструктором `Nothing` — значит извлечь цифру не удалось.

Обратите внимание: мы явно проверяем на равенство с нульарным конструктором:

```
a_coefficient == Nothing
```

Однако в модуле `Data.Maybe` есть два удобных предиката, `isJust` и `isNothing`. Поэтому мы могли бы написать так:

```
isNothing a_coefficient
```

Ещё и монада

Как уже было сказано в предыдущей главе, тип `Maybe` — это не просто опциональная обёртка, это ещё и монада. И благодаря этому факту мы можем делать вот такие вещи:

```
import Data.String.Utils
import Data.Maybe

result :: Maybe String -> String
result email = if isNothing email then "Bad email" else "Good!"

main :: IO ()
main =
    print $ result $ Just "me@gmail.com" >>= check_format >>= check_domain
    where check_format = \email ->
              if '@' `elem` email then return email else Nothing
          check_domain = \email ->
              if email `ends_with` ".com" then return email else Nothing
          ends_with = \str suffix -> endswith suffix str
```

Обмотали почтовый адрес в `Maybe`, прогнали его через проверочный конвейер и сделали вывод о корректности адреса. Если адрес некорректен — нам неважно, на какой стадии будет обнаружена ошибка, в *любом* случае конечный вывод будет однозначен.

Ну вот, теперь вы знаете о `Maybe`.

Функторы

Поговорим о функторах, одной из важных концепций языка Haskell. На самом деле мы их уже использовали, осталось лишь разобраться в сути.

Разбираемся

Функторами называют такие типы, значения которых могут быть маппированы (mapped over). Помните стандартную функцию `map`, позволяющую последовательно применить функцию к каждому элементу списка? Вот это тот самый случай.

Все функторы представлены стандартным классом `Functor`:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Один-единственный метод `fmap`, который говорит: «Дайте мне функцию, принимающую значение `a` и возвращающую значение `b`, а ещё дайте значение `a` в функторной обёртке, а я верну вам значение `b` в такой же функторной обёртке.»

Многие стандартные типы являются функторами.

Зачем это нам

Функция `fmap` — это волшебница. Она говорит: «Дайте мне ~~волшебную палочку~~ функцию и *нечто*, а я прикоснусь функцией к этому *нечто* и изменю его.» Это *нечто* и есть функтор.

Разумеется, говоря об изменении, я выражаюсь концептуально. Если мы видим строку:

```
fmap toLower ['A'..'Z']
```

мы можем выразиться технически точно: «Функция `fmap` последовательно применит функцию `toLower` к каждому элементу списка, в результате чего будет сконструирован новый список со значениями, являющимися результатом применения функции `toLower` к элементам изначального списка.»

Но так слишком длинно. Поэтому мы можем выразиться концептуально: «Функция `fmap` прикоснулась функцией `toLower` к списку и изменила его.» В этом и заключается фундаментальный смысл функторов: посредством `fmap` мы прикасаемся к ним некой функцией и изменяем их.

Более того, такое изменение может касаться не только содержимого функтора, но и его типа. Например, если взять функтор типа `[Char]` и прикоснуться к нему такой функцией:

```
fmap toLower ['A'..'Z']
```

на выходе у нас тоже получится функтор того же типа `[Char]`. Однако если прикоснуться такой функцией:

```
fmap digitToInt ['1'..'9']
```

на выходе у нас получится функтор типа `[Int]`.

Создаём свой

Путь это будет год:

```
data Year value = Year value
                  deriving Show
```

Тип `Year` содержит в себе год. Параметризация в данном случае весьма полезна: значение года может быть задано как числом, так и строкой. И поскольку значение года может (с концептуальной точки зрения) меняться, сделаем тип `Year` функторным:

```
instance Functor Year where
    fmap magic_wand (Year value) = Year (magic_wand value)
```

Функция `magic_wand` будет прикасаться не к году, а к его реальному содержимому. Например так:

```
increase :: Int -> Int
increase year = year + 1

main :: IO ()
main =
```

```
    print $ fmap increase year
    where year = Year 1981
```

Мы определили функцию `increase`, увеличивающую значение года на один. Казалось бы, зачем делать тип `Year` функторным? Ведь мы могли бы определить функцию, работающую с этим типом напрямую. Однако в этом случае мы жёстко привяжемся к типу `Year`. А фундаментальное преимущество функтора как раз в том, что функция, изменяющая его содержимое, ничего не знает об этом функторе.

Следовательно, в качестве волшебной палочки, прикасающейся к нашему году, могут выступать совершенно посторонние функции. Это и открывает перед нами огромный творческий простор.

Вот, теперь вы знаете о функторах. Рассматривайте их тоже как своего рода проектный паттерн, унифицирующий применение разных функций к разным значениям.

Часть 8 Остальное

О модулях

Как вы помните, в самом начале мы уже немного говорили о модулях. Пришло время изучить их более основательно.

Об иерархии

Зайдём в каталог `src/Utils` и откроем файл `Helpers.hs`. Внесём в него следующие изменения:

```
module Utils.Helpers (
    calibrate,
    graduate
) where

coefficient :: Double
coefficient = 0.99874

calibrate = length -> length * coefficient

graduate = length -> length / coefficient
```

Имя модуля уже не `Helpers`, а `Utils.Helpers`, то есть оно теперь отражает иерархию наших исходников. И хотя мы можем и не делать этого[1], но общая практика в мире Haskell именно такова: указывать имя модуля с полным путём к нему от корня. Именно поэтому все модули из Hackage, которые мы уже использовали, именовались полным путём:

```
import Data.String.Utils
```

Теперь вы знаете, что такое длинное название — это всего лишь путь: в исходниках данного пакета есть каталог `Data`, в нём — каталог `String`, а уже в нём лежит модуль `Utils.hs`.

1 Ведь в нашем сборочном файле подкаталог `src/Utils` уже указан в качестве хранилища модулей.

Кстати, не забудьте открыть ваш сборочный файл `Real.cabal` и внести изменения в параметр `other-modules`, а именно заменить `Helpers` на `Utils.Helpers`.

О лице

У каждого модуля, помимо имени, есть и лицо. Лицо — это набор всего того, что может быть импортировано в другие модули. По умолчанию всё содержимое модуля является его лицом, то есть доступно всему миру. Однако в реальных модулях у вас, скорее всего, будут некоторые служебные функции и типы, которые вы не захотите показывать всему миру.

Взглянем ещё раз на модуль `Helpers.hs`:

```
module Utils.Helpers (
    calibrate,
    graduate
) where
```

Имена двух функций в круглых скобках — это и есть лицо нашего модуля, поэтому лишь эти две функции можно будет импортировать. Всё остальное, что есть в этом модуле, останется тайной за семью печатями. В частности, наше служебное значение `coefficient`: при попытке импортировать его в другой модуль компилятор удивлённо упрекнёт вас, мол, не знаю никакого `coefficient`. Но если оно окажется кому-то нужным — допишем его имя в круглых скобках:

```
module Utils.Helpers (
    calibrate,
    graduate,
    coefficient
) where
```

и всё заработает. Теперь пропишем в лице модуля наш собственный тип:

```
module Utils.Helpers (
    calibrate,
    graduate,
    Color( Red, Green, Blue )
) where

data Color = Red | Green | Blue
```

```
              deriving Show
```

Обратите внимание: недостаточно прописать в *лице* модуля имя типа, необходимо также перечислить его конструкторы в виде кортежа. Впрочем, достаточно перечислить *лишь* те конструкторы, которые вы реально собираетесь использовать в *других модулях для* создания значений типа `Color`.

Ничего, кроме…

В ряде случаев вам нельзя (или необязательно) импортировать всё то, что есть в модуле. Откроем `Main.hs` и напишем в нём:

```
import Utils.Helpers (calibrate)  -- импортируем только calibrate

main :: IO ()
main = print $ calibrate 12.4
```

Мы импортировали *лишь* то, что перечислено в виде кортежа сразу за именем модуля. Всё остальное содержимое `Utils.Helpers` осталось невидимым.

Вы спросите, зачем это нужно? В конце концов, ну и пусть импортируется всё, а уж мы решим, что нам использовать.

Главная цель частичного импорта — исключение конфликта имён. В разных модулях зачастую присутствуют одноимённые сущности. В этом случае мы можем взять из «конфликтных» модулей только то, что нам необходимо.

Всё, кроме…

Существует также противоположный подход, а именно частичный импорт всего содержимого модуля, кроме указанного. В этом случае нам понадобится ключевое слово `hiding`:

```
import Utils.Helpers hiding (graduate)  -- graduate скрыта
```

После слова `hiding` перечислены в виде кортежа те сущности, которые будут недоступны (скрыты) в текущем модуле. Как вы уже догадались, такой подход так же используется во избежание конфликтов между одноимёнными сущностями из разных модулей.

Принадлежность

В реальных проектах вы столкнётесь с ситуацией, когда вам очень нужно будет совместно использовать одноимённые функции из разных модулей. Просто так это сделать не получится, компилятор проявит принципиальность и потребует уточнений. В этом случае нам необходимо явно указать принадлежность функции к конкретному модулю:

```
import Utils.Helpers
import Utils.Math  -- а вдруг здесь тоже есть функция calibrate?

main :: IO ()
main = print $ Utils.Helpers.calibrate 12.4
```

В этом случае конфликта не будет.

Короткая принадлежность

В уже известном нам пакете `MissingH` есть модули с весьма длинным именем, например `System.Console.GetOpt.Utils`. Согласитесь, длинновато писать такой «префикс» всякий раз, когда нужно указать принадлежность. К счастью, есть способ ввести короткий псевдоним для модуля:

```
import Utils.Helpers as H

main :: IO ()
main = print $ H.graduate 23
```

Ключевое слово `as` вводит короткое имя для `Utils.Helpers`. Кстати, на это имя действует общее для всех типов правило: только с большой буквы. Поэтому такой вариант не пройдёт:

```
import Utils.Helpers as h
```

Обязательная принадлежность

В ряде случаев бывает полезным призвать пользователя к строгому порядку и обязать его указывать принадлежность сущностей к модулю. Например:

```
import qualified Utils.Helpers as H

main :: IO ()
main = print $ graduate 23
```

Мы импортировали наш модуль с ключевым словом `qualified`. Именно поэтому такой код не пройдёт компиляцию. Слово `qualified` обязывает нас уточнять принадлежность всех используемых сущностей к соответствующим им модулям. Поэтому даже если функция `graduate` представлена в единственном экземпляре, при `qualified`-импорте мы обязаны явно указать, к какому модулю она принадлежит:

```
print $ H.graduate 23
```

О модуле `Main`

Если каждый наш модуль должен задаваться неким именем:

```
module Utils.Helpers where ...
```

то почему же безымянным остался главный модуль `Main`? Пришло время узнать правду: этот модуль тоже нужно именовать. Прямо так и пишем:

```
module Main where

...

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

Вы спросите, почему же мы не делали этого раньше? Дело в том, что компилятор `ghc` сам может понять, который из всего множества модулей есть модуль `Main`. Однако, в соответствии со стандартом Haskell 2010, правильнее будет указывать имя модуля `Main` явно. Я думаю, это логично, а то получается, что все модули названные, а самый главный модуль — безымянный.

Ну вот, теперь вы знаете о модулях всё.

Рекурсивные функции

В языке Haskell нет циклических конструкций. Никаких `for`, никаких `while`. Единственный способ явно организовать цикл — рекурсивные функции.

Вы спросите, почему о них не было рассказано раньше, в разделе о функциях? Да потому что в реальных проектах вам редко придётся иметь дело с рекурсивными функциями, ведь почти все повторяющиеся действия вы будете делать без них.

Например, самый распространённый случай циклического действия — итерирование всей последовательности элементов. На практике это будет проход некоторой функцией по всем элементам списка. Но вы уже знаете, что для этого есть функции `map`, `filter` и подобные им. А для итерирования последовательности с условием(ями) вы сможете использовать уже известные вам list comprehensions. А о таких простых вещах, как получение суммы или произведения элементов списка, и говорить нечего.

Иными словами, в большинстве случаев вы и не вспомните о рекурсивных функциях. И всё-таки знать о них полезно, тем более что в некоторых случаях они вам понадобятся.

Сама себя

Рекурсивной называется функция, в теле которой присутствует вызов её самой. Конечно, мы будем говорить только о простой рекурсии[1], косвенную же рекурсию[2] мы рассматривать не будем.

Например:

```
make_list_from :: a -> Int -> [a]
make_list_from value how_many =
    if how_many > 0
    then value : make_list_from value (how_many - 1)
    else []
```

Эта функция строит список, элементами которого будут значения `value`, количество же элементов будет равным `how_many`. И если мы вызовем её так:

1 Когда функция вызывает саму себя непосредственно.
2 Когда функция A вызывает функцию B, которая в свою очередь вызывает функцию A.

```
main :: IO ()
main = print $ make_list_from 2 3
```

на выходе получим список из трёх двоек:

```
[2,2,2]
```

Основное правило

Всё, что имеет начало, имеет и конец. Когда рекурсия запущена, нам нужен способ остановить её. Поэтому тело рекурсивной функции должно содержать не только зацикливающий код, но и код, обеспечивающий выход из этого цикла.

Рассмотрим тело нашей функции:

```
if how_many > 0
then value : make_list_from value (how_many - 1)  -- запускаю цикл
else []                                           -- останавливаю цикл
```

Перед нами условие, приводящее нас в одну из логических ветвей. Первая ветвь запускает цикл, вторая ветвь останавливает его. Рассмотрим совместную работу этих двух ветвей.

Погружаемся

Разберём вызов:

```
make_list_from 2 3
```

В самом начале мы заходим в эту функцию с аргументами `2` и `3`. Следовательно, на этом шаге внутренности данной функции вот такие:

```
make_list_from 2 3 =
    if 3 > 0
    then 2 : make_list_from 2 (3 - 1)
    else []
```

Поскольку условие `3 > 0` выполняется, мы попадаем в первую логическую ветвь:

```
2 : make_list_from 2 (3 – 1)
```

Здесь используется оператор `':'`, добавляющий левый операнд в начало списка, выступающего правым операндом. То есть мы хотим добавить значение `2` в начало списка, порождённого правым выражением. Но этим правым выражением идёт повторный вызов нашей функции. Следовательно, запускается цикл, и мы погружаемся.

Взглянем на внутренности нашей функции во время второго вызова:

```
make_list_from 2 2 =
    if 2 > 0
    then 2 : make_list_from 2 (2 – 1)
    else []
```

Условие `2 > 0` опять выполняется, значит, мы опять попадаем в первую логическую ветвь:

```
2 : make_list_from 2 (2 – 1)
```

Мы только собрались добавить значение `2` в начало списка, порождённого правым выражением — и тут опять входим в очередной вызов нашей функции. Её внутренности такие:

```
make_list_from 2 1 =
    if 1 > 0
    then 2 : make_list_from 2 (1 – 1)
    else []
```

И снова условие `1 > 0` выполняется, поэтому мы опять входим в первую ветвь:

```
2 : make_list_from 2 (1 – 1)
```

И снова мы, желая добавить значение `2` в начало списка, порождённого правым выражением, погружаемся в очередной вызов нашей функции. Вот что внутри:

```
make_list_from 2 0 =
    if 0 > 0
    then 2 : make_list_from 2 (0 – 1)
    else []
```

Условие `0 > 0` уже не выполнится, поэтому мы окажемся во второй логической ветви. А в ней — пустой список. Всё, мы дошли до дна, наша рекурсия остановилась.

Всплываем

Теперь начинается «обратная логическая раскрутка» наших вложенных вызовов. Именно в процессе это обратной раскрутки и происходит вся работа, ведь до тех пор, пока мы не дошли до «рекурсивного дна», никакой работы мы ещё не сделали. Мы каждый раз собирались добавить значение в начало списка — и тут же погружались в очередной вызов. Поэтому формирование готового списка начинается с того самого пустого списка, который был возвращён последним вызовом нашей функции.

Схематично наше всплытие можно изобразить так:

```
make_list_from 2 3                  -- зашли в функцию впервые
    2 : make_list_from 2 2          -- первый рекурсивный вызов
        2 : make_list_from 2 1      -- второй рекурсивный вызов
            2 : make_list_from 2 0  -- третий рекурсивный вызов
                []                  -- последний рекурсивный вызов
```

Четвёртый вызов вернул пустой список, поэтому всплытие приобрело следующий вид:

```
make_list_from 2 3
    2 : make_list_from 2 2      -- первый рекурсивный вызов
        2 : make_list_from 2 1  -- второй рекурсивный вызов
            2 : []              -- третий рекурсивный вызов
```

Наша функция на третьем рекурсивном вызове получила пустой список и добавила в его начало значение `2`, в результате чего появится список с одним значением.

Далее будет так:

```
make_list_from 2 3
    2 : make_list_from 2 2  -- первый рекурсивный вызов
        2 : [2]             -- второй рекурсивный вызов
```

На втором рекурсивном вызове функция получила список, состоящий из одного значения, и добавила в его начало значение 2, в результате чего появился список уже с двумя значениями.

Следующая ступень:

```
make_list_from 2 3
    2 : [2,2]  -- первый рекурсивный вызов
```

На первом рекурсивном вызове мы получаем уже список, состоящий из двух значений, и опять добавляем в его начало значение 2.

Таким образом, завершив наше всплытие с «рекурсивного дна», в месте вызова нашей функции

```
make_list_from 2 3
```

мы получим наш итоговый список:

```
[2,2,2]
```

Всё. Теперь вы знаете о рекурсивных функциях. Конечно, рекурсия немного выворачивает мозг программисту, привыкшему к `for`, но, как заметил один из корифеев программирования Laurence Peter Deutsch, «итерация свойственна человеку, а рекурсия божественна.»

Про апостроф

Есть в Haskell одна особенность, связанная с именованием. Признаюсь, я удивился, когда о ней узнал. Оказывается, частью имени любой программной сущности может выступать апостроф. Да-да, та самая одинарная кавычка, в которую мы помещаем отдельный символ `Char`.

Мы можем использовать один или более апострофов в имени функции:

```
strange_function' :: Int -> Int
strange_function' arg = arg
```

в имени типа:

```
data Strange_'type' = Strange_'type' String
```

в имени класса типов:

```
class Stran''ge_Class'' a where
    f :: a -> String
```

и даже в имени значения:

```
strange_value''' :: Integer
strange_value''' = 123
```

На мой взгляд, нет никакого рационального зерна в том, чтобы включать апостроф в какое-либо имя. Тем более что такой символ разрывает «текстовую целостность» слова[1]. Однако мне довелось исследовать исходный код некоторых Haskell-проектов, и там я видел имена функций с апострофами. Поэтому вы должны знать об этой особенности, хотя бы для того, чтобы сопровождать чужой код.

1 Во многих текстовых редакторах двойной клик на слове выделяет его целиком. При наличии апострофа в имени сделать такое уже не получится.

О форматировании

Код на Haskell является форматно-зависимым: мы не можем расставлять пробелы и отступы там, где нам заблагорассудится. Необходимо придерживаться определённых принципов.

Функция

Если мы напишем так:

```
main :: IO ()
main =
putStrLn "Hi Haskeller!"
```

компилятор выскажет своё несогласие:

```
parse error (possibly incorrect indentation or mismatched brackets)
```

Следующий пример:

```
 main :: IO ()
 main =
 putStrLn "Hi Haskeller!"
```

Здесь мы поставили один пробел перед каждой из трёх строк, однако и в этом случае компилятор закапризничает:

```
parse error on input `main'
```

Или вот так:

```
main :: IO ()
 main =
 putStrLn "Hi Haskeller!"
```

В этом случае мы получим ещё более странную ошибку:

```
Illegal type signature: `IO () main'
```

Как видите, из-за пробела перед именем функции компилятор принял это имя за часть сигнатуры.

Когда в теле функции несколько строк, появляются дополнительные ограничения. Если напишем так:

```
main :: IO ()
main = do
    putStrLn "Hi Haskeller!"
     putStrLn "Hi again!"
```

получим вот это:

```
Couldn't match expected type `(String -> IO ()) -> [Char] -> IO ()'
               with actual type `IO ()'
    The function `putStrLn' is applied to three arguments,
    but its type `String -> IO ()' has only one
```

Из-за сдвига второй функции по отношению к первой компилятор подумал, что первая по счёту `putStrLn` применяется к трём аргументам. Если же напишем так:

```
main :: IO ()
main = do
     putStrLn "Hi Haskeller!"
    putStrLn "Hi again!"
```

получим уже знакомую нам ошибку:

```
parse error on input `putStrLn'
```

Здесь компилятор ругнулся уже на вторую по счёту `putStrLn`.

В общем, экспериментальным путём я выяснил, что форматирование кода функции должно соответствовать следующим правилам:

1. Объявление и определение функции, должны начинаться с первого (самого левого) символа строки.

2. Если тело функции начинается со следующей строки после имени, перед этим телом должен присутствовать отступ от первого символа строки, хотя бы в один пробел.
3. Если тело функции состоит из нескольких выражений, стоящих на отдельной строке каждая, эти выражения должны быть вертикально выровнены по левому краю.

Поэтому придерживайтесь приблизительно такого шаблона:

```
main :: IO ()
main = do
    putStrLn "Hi Haskeller!"
    putStrLn "Hi again!"
```

и компилятор будет просто счастлив.

Тип

На код, связанный с типами, также наложены некоторые форматные ограничения.

```
 data IP_address = IP String
```

Перед словом `data` стоит лишний пробел, и компилятор вновь вспоминает нас недобрым словом:

```
parse error on input `data'
```

Вот такой код тоже не пройдёт компиляцию:

```
data
IP_address = IP String
```

равно как и такой:

```
data IP_address =
IP String
```

и даже такой:

```
data IP_address
```

```
= IP String
```

В ходе экспериментов было выяснено, что правила для кода определения типа схожи с вышеупомянутыми правилами для кода функции:

1. Ключевое слово `data` начинается с самого левого символа строки.
2. Если объявление переходит на следующую строку, то перед ним должен быть хотя бы один пробел.

Поэтому пишите приблизительно так:

```
data IP_address = IP String
                  deriving Show
```

и компилятор будет вам благодарен.

Класс типов

С классами типов — та же история. Если напишем так:

```
class Note n where
write :: n -> Bool
```

получим экзотическую ошибку:

```
The type signature for `write' lacks an accompanying binding
```

Если вздумаем написать так:

```
class Note n where
  write :: n -> Bool
 read :: n -> String
```

снова получим по башке:

```
parse error on input `read'
```

И если так напишем:

```
class Note n where
    write :: n -> Bool
    read
```

```
    :: n -> String
```

и даже если так:

```
  class Note n where
    write :: n -> Bool
    read :: n -> String
```

компилятор будет принципиален до крайности и не пропустит такой код.

В общем, тут правила точно такие же:

1. Начинаем с самого левого символа строки.
2. Перед методами — хотя бы однопробельный отступ.
3. Методы должны быть вертикально выровнены по левому краю.

Следовательно, ублажаем компилятор и пишем примерно так:

```
class Note n where
    write :: n -> Bool
    read :: n -> String
```

Константа

Для отдельной константы правила точно такие же, как и для функции. Поэтому пишем:

```
coefficient :: Double
coefficient = 0.0036
```

и всё будет хорошо.

Условие

Тут я выявил лишь одно ограничение — край ключевого слова `if` должен быть самым левым по отношению ко всем остальным частям выражения. То есть можно написать так:

```
main :: IO ()
main = do
    if 2 /= 2
      then
```

```
        putStrLn "Impossible"
      else
        putStrLn "I believe"
```

и так:

```
main :: IO ()
main = do
    if 2 /= 2
        then
      putStrLn "Impossible"
        else
      putStrLn "I believe"
```

и даже так:

```
main :: IO ()
main = do
    if 2 /= 2

     then
     putStrLn "Impossible"
     else
          putStrLn "I believe"
```

Но вот такого компилятор не потерпит:

```
main :: IO ()
main = do
    if 2 /= 2
   then
      putStrLn "Impossible"
   else
      putStrLn "I believe"
```

равно как и такого:

```
main :: IO ()
main = do
    if 2 /= 2
    then
    putStrLn "Impossible"
```

```
    else
    putStrLn "I believe"
```

Локальные выражения

Эти друзья менее прихотливы. В отношении выражения `where` я нашёл только одно ограничение:

```
prepare :: String -> String
prepare str =
    str ++ helper
where
    helper = "dear. "
```

Получим ошибку:

```
parse error on input `where'
```

Такого же рода ограничение действует и на `let`:

```
prepare :: String -> String
prepare str =
let helper = "dear. "
    in
    str ++ helper
```

Однако ошибка будет другой:

```
parse error (possibly incorrect indentation or mismatched brackets)
```

Суть вы уловили: пусть `where` и `let` гармонируют с остальным кодом тела функции.

Вывод

Пишите аккуратно, без ненужных изысков. Да, многие разработчики не любят, когда синтаксис языка форматно-зависимый, но, как говорится, что есть, то есть. Кстати, упомянутые выше ограничения в некотором смысле дисциплинируют программиста, так что в них тоже можно усмотреть плюс.

Заключение

И что, это всё??

Нет. Мы изучили многое, но далеко не всё.

В будущих изданиях мы узнаем о таких вещах, как монадные трансформеры, аппликативные функторы, многопоточное программирование, оптимизация скорости выполнения кода, построение собственных DSEL и интеграция с другими языками. Ждите новостей.

Кроме того, перед вами лежит необъятный Hackage[1], со множеством готовых программных решений. Их вы можете изучать самостоятельно, в зависимости от решаемых вами задач.

Главная задача этой книги выполнена, если после её прочтения вы поняли все эти «странности Haskell».

P.S. Я буду очень признателен вам за любые отзывы об этой книге.
Если есть вопросы, критика или предложения — напишите мне[2].

1 http://hackage.haskell.org/packages
2 me@dshevchenko.biz

Благодарности

Эта книга — плод не только моих усилий. Я благодарю всех тех разработчиков, которые помогли мне понять и полюбить Haskell. И несмотря на то, что некоторые из этих людей даже не подозревают о том, что оказали мне помощь[1], я всё равно им признателен.

Зарубежные имена привожу в оригинале.

Благодарю авторов книги «Real World Haskell»[2] Bryan O'Sullivan, Don Stewart, и John Goerzen. Несмотря на то, что эта книга считается устаревшей, именно благодаря ей я убедился в том, что Haskell пригоден не только для реализации алгоритма быстрой сортировки.

Благодарю автора книги «Learn You a Haskell for Great Good!»[3] Miran Lipovača. Эта веселая книга со слоном доказала мне, что понять Haskell могут не только аспиранты МФТИ.

Благодарю автора русскоязычного учебника по Haskell[4] Антона Холомьёва.

Также благодарю Артёма Петрова за ценные замечания о книге и помощь в её написании.

А ещё я говорю «спасибо» вам, уважаемый читатель. За то, что вы потратили своё время на изучение моего скромного труда, и за то, что согласились послушать о Haskell из уст обыкновенного программиста.

Удачи на профессиональном поприще!

Шевченко Денис
март 2014

1 Ведь мы незнакомы друг с другом.
2 `http://book.realworldhaskell.org`
3 `http://learnyouahaskell.com`
4 `http://anton-k.github.io/ru-haskell-book/book/toc.html`