

О Haskell

По-человечески

Д. Шевченко

О Haskell по-человечески

издание 2.0

Денис Шевченко

www.ohaskell.guide

2016

Книга свободно распространяется на условиях
лицензии [CC BY-NC 4.0](#)

© Денис Шевченко, 2014-2016

Оглавление

1	Приветствую!	9
	Почему эта книга появилась	9
	Цель	9
	О себе	10
	О вас	10
	Обещание	10
2	Первые вопросы	11
	«Что такое этот ваш Haskell?»	11
	«Это что, какой-то новый язык?»	11
	«И кто его сделал?»	12
	«А библиотеки для Haskell имеются?»	12
	«И что, его уже можно в production?»	12

«А порог вхождения в Haskell высокий?»	13
«И что же в нём такого необычного?»	13
«А если сравнить его с C++/Python/Scala...»	13
3 Об этой книге	15
Чего здесь нет	16
О первом и втором издании	16
Читайте последовательно	17
О пояснениях	17
Благодарность	18
4 Приготовимся	19
Устанавливаем	19
Разворачиваем инфраструктуру	21
Hi World	21
Модули: первый взгляд	22
5 Киты и Черепаха	25
Черепаха	25
Первый Кит	27
Второй Кит	29
Третий Кит	32

6	Неизменность и чистота	33
	Объявляем и определяем	33
	Чисто функциональный	36
	«Присваивание? Не, не слышал...»	37
	Удивлены?	38
7	Выбираем и возвращаемся	40
	Выглянем во внешний мир	40
	Выбор и выход	42
8	Выбор и образцы	46
	Не только из двух	46
	Без Если	49
	Сравнение с образцом	51
	case	53
9	Пусть будет там, Где...	55
	Пусть	55
	Где	58
	Вместе	58
10	Мир операторов	62

Зачем это нужно?	64
11 Список: знакомство	65
Тип списка	67
Действия над списками	67
Неизменность списка	70
12 Кортеж	72
Тип кортежа	73
Действия над кортежами	73
Не всё	77
А если ошиблись?	79
13 Лямбда-функция	80
Истоки	80
Строение	82
Тип функции	83
Локальные функции	85
14 Композиция функций	88
Скобкам — бой!	88
Композиция и применение	89

Как работает композиция	93
15 ФВП	95
Отображение	96
Композиция для отображения	99
16 Namespace и библиотеки	101
Библиотеки большие и маленькие	101
Namespace	102
Иерархия в имени	104
Лицо	105
Импортируем по-разному	107
Оформление	111

Глава 1

Приветствую!

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

Почему эта книга появилась

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

Цель

Функциональное программирование — своеобразное гетто посреди мегаполиса нашей индустрии. Доля функциональных языков

пока ещё мала, однако эти языки — и в частности Haskell — являются мощными инструментами разработки, и в рамках этой книги я покажу вам эту мощь. Вероятно, вы слышали, что Haskell — это нечто сугубо научное и непригодное для жизни? Читайте дальше, и вскоре вы убедитесь в обратном.

О себе

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

О вас

Отличаете объявление функции от её определения? Знаете что такое компилятор? Умеете работать с командной строкой? Если да — смело продолжайте читать, никаких дополнительных навыков от вас не ожидается.

Обещание

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

Глава 2

Первые вопросы

С них и начнём.

«Что такое этот ваш Haskell?»

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

«Это что, какой-то новый язык?»

Вовсе нет. История Haskell началась ещё в 1987 году. Этот язык был рождён в математических кругах, когда группа людей решила создать лучший функциональный язык программирования. В 1990

году вышла первая версия языка, названного в честь известного американского математика [Хаскела Карри](#). В 1998 году язык был стандартизован, а начиная с 2000-х началось его медленное вхождение в мир практического программирования. За эти годы язык совершенствовался, и вот в 2010 мир увидел его обновлённый стандарт. Так что мы имеем дело с языком, который старше Java.

«И кто его сделал?»

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

«А библиотеки для Haskell имеются?»

Имеются. В процессе чтения вы познакомитесь со многими из них.

«И что, его уже можно в production?»

Можно, и уже давно. С момента выхода первого стандарта язык улучшался, развивалась его экосистема, появлялись новые биб-

лиотеки, выходили в свет книги. Сегодня, в 2016, можно уверенно заявить, что Haskell полностью готов к серьёзному коммерческому использованию, о чём убедительно свидетельствуют истории успешного внедрения Haskell в бизнесе, в том числе крупном.

«А порог вхождения в Haskell высокий?»

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

«И что же в нём такого необычного?»

Например, в Haskell нет оператора присваивания. Вообще. А что касается остальных странностей языка — вся книга им и посвящена.

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

Сравнение Haskell с другими языками выходит за рамки этой книги. Несколько раз вы встретите здесь кусочки кода на других

языках, но я привожу их исключительно для того, чтобы подчеркнуть различие с Haskell, а вовсе не для сравнения в контексте «лучше/хуже».

Глава 3

Об этой книге

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

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

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

Эта книга не возведёт вас на вершины Haskell, но она откроет вам

путь к этим вершинам.

Чего здесь нет

Трёх вещей вы не найдёте на страницах этой книги:

1. Справочника по Haskell. Дублировать [официальное описание стандарта Haskell 2010](#) я не стану.
2. Набора готовых рецептов. За рецептами пожалуйста на [Stackoverflow](#).
3. Введения в математическую теорию. Несмотря на то, что Haskell корнями своими уходит в математику, в этой книге нет погружения в теорию категорий и в иные теории. Извините, если разочаровал.

О первом и втором издании

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

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

Вторая причина — изменившаяся цель книги. Я намеренно сузил круг рассматриваемых здесь тем. Теперь книга всецело посвящена основам языка, поэтому не ждите от неё рассмотрения специ-

фических тем. Я не очень-то верю в идею book-all-in-one, книга для новичков должна быть книгой для новичков. Вы не встретите здесь ни примеров реализации 3D-движка, ни рассказа о работе с PostgreSQL, ни повествования о проектировании игры для Android. Всё это можно делать с Haskell, но подобным темам посвящены другие публикации, которые несомненно будут вам по плечу после прочтения моей книги.

Читайте последовательно

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

О пояснениях

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

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

тип этот равен тому

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

Вот ещё один пример:

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

данное значение

это

хост

а вот это

значение

это

имя

Здесь я говорю вам: «Данное значение — это хост, а вот это значение — это имя».

Благодарность

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

А ещё я благодарю всех разработчиков, неустанно совершенствующих мир Haskell. Вашими усилиями наша профессия становится ещё более прекрасной!

Глава 4

Приготовимся

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

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

Haskell — кроссплатформенный язык, работающий и в OS X, и в Linux, и даже в Windows. Однако в 2008 году я навсегда покинул мир Windows, поэтому все последующие примеры взаимодействия с командной строкой подразумевают Unix-way. Вся конфигурация и примеры кода опробованы мною на OS X Yosemite.

Устанавливаем

Идём [сюда](#) и скачиваем архив для нужной нам ОС. Распаковываем архив — и перед нами утилита под названием stack. Для удобства

располагаем её в каком-нибудь каталоге, доступном в PATH. Рекомендованный путь — `~/local/bin/`.

Если же вы живёте в мире Mac и пользуетесь [Homebrew](#) — вам ещё проще. Делаете:

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

Всё.

На момент написания книги я использовал `stack` версии 1.0.2. Если у вас более старая версия — непременно обновитесь. Если же более новая — у вас теоретически что-нибудь может работать не совсем так, как описано ниже, поскольку `stack` активно развивается, добавляются новые возможности, может быть где и ломают обратную совместимость.

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

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

Haskell-инфраструктура — экосистема, краеугольным камнем которой является компилятор GHC (Glasgow Haskell Compiler). Как было сказано ранее, Haskell — это компилируемый язык: приложение представляет собой обыкновенный исполняемый файл.

Haskell-проект — среда для создания приложений и библиотек.

Haskell-библиотеки — готовые решения, спасающие нас от изобретения велосипедов.

Разворачиваем инфраструктуру

Делаем:

```
$ stack setup
```

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

Hi World

Создадим наш первый Haskell-проект:

```
$ stack new real
```

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

```
.
├─ LICENSE
├─ Setup.hs
├─ app
│   └─ Main.hs <- Главный модуль
├─ real.cabal <- Сборочный файл
├─ src
│   └─ Lib.hs <- Вспомогательный модуль
└─ stack.yaml
```

```
└─ test
  └─ Spec.hs
```

О содержимом проекта вам пока знать не нужно, просто соберём его командой:

```
$ stack install
```

Запомните эту команду, мы будем использовать её постоянно. В результате её выполнения появится файл `real-exe`. А поскольку скопирован он будет в упомянутый выше каталог `~/.local/bin/`, мы сможем сразу запустить программу:

```
$ real-exe
someFunc
```

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

Модули: первый взгляд

Настоящие проекты никогда не состоят из одного-единственного файла. Файлы, содержащие Haskell-код — это и есть модули. Один файл — один модуль. В Haskell нет заголовочных файлов: каждый из модулей рассматривается как самостоятельная единица проекта, содержащая в себе разные полезные вещи. А чтобы воспользоваться этими вещами, необходимо один модуль импортировать в другой.

Откроем модуль `src/Lib.hs`:

```
module Lib      -- Имя модуля
( someFunc
```

```
) where
```

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

В первой строке объявлено, что имя этого модуля — `lib`. Далее, в круглых скобках упомянуто содержимое данного модуля, а именно имя функции `someFunc`. Затем, после ключевого слова `where`, мы видим определение функции `someFunc`. Пока вам не нужно знать о синтаксисе данной конструкции, в следующих главах мы разберём его тщательнейшим образом. Расширение `.hs` — стандартное расширения для модулей.

Теперь откроем модуль `app/Main.hs`:

```
module Main where  
  
import Lib          -- Импортируем модуль Lib...  
  
main :: IO ()  
main = someFunc -- Используем его содержимое...
```

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

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

Все модули в наших проектах можно разделить на две части: те, которые мы берём из библиотек и те, которые мы создали сами. Библиотеки — это уже кем-то созданные решения, в последующих главах мы познакомимся со многими из них. Среди библиотек сле-

дует выделить одну, так называемую стандартную библиотеку. Модули из стандартной библиотеки мы начнём использовать уже в ближайших главах. А одна из глав будет полностью посвящена рассказу о библиотеках: из неё мы подробно узнаем, откуда берутся библиотеки и как их использовать.

Глава 5

Киты и Черепаха

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

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

Черепаха

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

$1 + 2$

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

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

Все выражения можно разделить на две группы: те, которые (всё ещё) можно вычислить и те, которые (уже) нельзя. Вычисление (англ. *evaluation*) — это фундаментальное действие по отношению к выражению, ведь именно вычисление даёт нам тот самый полезный результат. Так, выражение:

$1 + 2$

может дать нам полезный результат, а именно сумму двух чисел. Вычислив это выражение, мы получаем результат:

3

Причём это не просто число 3, это тоже выражение. Подобное выражение уже нельзя вычислить, оно вычислено окончательно, до самого дна, и мы можем лишь использовать его как есть.

В результате вычисления выражение всегда уменьшается (англ. *reduce*). В русскоязычной литературе иногда так и пишут: «редукция выражения». Уменьшать выражение можно до тех пор, пока оно не достигнет своей нередуцируемой формы. Упомянутое выше выражение $1 + 2$ ещё можно редуцировать, а вот выражение 3 — уже нельзя.

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

Первый Кит

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

Вспомним математическое определение функции. Не пугайтесь, математики будет совсем немного:

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

Рассмотрим функцию возведения целого числа в квадрат:

```
sqaure v = v * v
```

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

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

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

Сама по себе функция абсолютно бесполезна. Чтобы заставить её сделать полезную работу, её необходимо применить (англ. *apply*) к аргументу. Ведь если на вход ничего не кинули, то и на выходе ничего не получим. Вот пример:

```
square 2
```

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

```
sum 10 20
```

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

```
(+) 1 2
```

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

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

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

применяют к чему-то.

Второй Кит

Итак, любое редуцируемое выражение — это применение функции к некоторому аргументу (а по сути, тоже выражению):

`sqaure` 2

функция аргумент

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

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

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

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

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

Статическая проверка

Статическая проверка типов (англ. *static type checking*) — это проверка типов всех данных в программе на этапе компиляции. Haskell-компилятор упрям: когда ему что-либо не нравится в типах, он громко ругается. Поэтому если функция работает с целыми числами, применить её к строкам никак не получится. Таким образом, когда компиляция завершилась успешно, мы точно знаем, что с типами у нас всё в порядке.

Преимущества статической проверки невозможно переоценить, ведь она гарантирует отсутствие в наших программах целого ряда ошибок. Мы уже не сможем спутать числа со строками или вычесть метры из рублей.

Конечно, у этой медали есть и обратная сторона — время компиляции. Вам придётся привыкнуть к этой мыслию: внесли изменения в проект — будьте добры скомпилировать. Однако утешением вам пусть послужит тот факт, что преимущества статической проверки в реальном проекте куда ценнее времени, потраченного на компиляцию.

Сила

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

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

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

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

В Haskell подобный код не имеет ни малейших шансов пройти компиляцию. Мы всегда получаем то, что ожидаем, и если должно быть число с плавающей точкой — расшибись, но предоставь именно его. Компилятор скрупулёзно отслеживает соответствие между ожидаемым типом и фактическим, поэтому когда компиляция завершается успешно, мы абсолютно уверены в гармонии между типами всех наших данных.

Выведение

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

```
double value = 122.04;  
}
```

однако в Haskell мы напишем просто:

```
value = 122.04
```

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

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

Да, кстати, вот простейшие стандартные типы, они нам в любом случае понадобятся:

123	<code>Int</code>
23.5798	<code>Double</code>
'a'	<code>Char</code>
"Hello!"	<code>String</code>
<code>True</code>	<code>Bool</code> , истина
<code>False</code>	<code>Bool</code> , ложь

С типами `Int` и `Double` вы уже знакомы. Тип `Char` — это Unicode-символ. Тип `String` — обыкновенная строка. Тип `Bool` — логический тип, истина или ложь. В последующих главах мы встретимся ещё с несколькими стандартными типами, но пока хватит и этих.

Третий Кит

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

Уверен, после прочтения этой главы у вас появилось множество вопросов. Однако я пока не стану отвечать на них. Более того, следующая глава несомненно удивит вас. Вперёд.

Глава 6

Неизменность и чистота

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

Объявляем и определяем

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

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

Первая строка — объявление, вторая — определение. Объявление

(англ. declaration) — это весть всему миру о том, что такая функция существует, вот её имя и вот типы, с которыми она работает. Определение (англ. declaration) — это «тело» функции, её конкретное содержимое.

Рассмотрим объявление:

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

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

square	::	Int	->	Int
имя		тип		тип
функции		аргумента		вычисленного значения

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

product	::	Int	->	Int	->	Int
имя		тип		тип		тип
функции		первого аргумента		второго аргумента		вычисленного значения

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

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

Теперь рассмотрим определение функции `square`:

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

Схема определения такова:

square	v	=	v * v
имя	имя	это	выражение
функции	аргумента		

А функция `product` определена так:

product	x	y	=	x * y
имя	имя	имя	это	выражение
функции	первого	второго		
	аргумента	аргумента		

Определение тоже разделено на две части: слева от знака равенства — имя функции и имена аргументов (уже имена, а не типы), разделённые пробелами, а справа — выражение, составляющее суть функции, её содержимое. В C-подобных языках закрепилось понятие «тело функции» (англ. *function body*), однако в Haskell чаще говорят о выражении.

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

```
product x y = x * y
```

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

Но откуда у меня такая уверенность? А вот откуда.

Чисто функциональный

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

Скажу больше: чистая функция не видит окружающий мир. Вообще. Причём в данном случае под «окружающим миром» я подразумеваю не только внешний по отношению ко всей программе мир (например, файловая система или сеть), но и все остальные функции. Чистую функцию можно сравнить с чёрным ящиком: она знает только свои аргументы и вычисляемое ею значение. Это даёт нам второе преимущество чистых функций — композируемость. Раз функции полностью изолированы друг от друга, их очень просто

комбинировать, строя из более простых более сложные.

И раз уж я упомянул об этом вскользь, подчеркну: чистые функции не способны взаимодействовать с внешним по отношению к программе миром. Они не могут вывести текст на консоль, их нельзя заставить обработать HTTP-запрос, они не умеют дружить с базой данных. Они суть вещь в себе.

И чтобы удивить вас ещё больше, открою очередной секрет Haskell.

«Присваивание? Не, не слышал...»

В мире Haskell нет места оператору присваивания. Впрочем, этот факт удивителен лишь на первый взгляд. Задумаемся: раз уж каждая функция в конечном итоге представляет собою выражение, вычисляемое посредством применения каких-то других функций к каким-то другим аргументам, тогда нам просто не нужно ничего ничему присваивать.

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

```
coeff = 0.569;
```

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

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

```
coeff = 0.569;
```

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

Удивлены?

Полагаю, да. Как же можно написать реальную программу на языке, в котором нельзя изменять данные? Какой прок от этих чистых функций, если они не способны ни файл прочесть, ни запрос по сети отправить? Оказывается, прок есть, и на Haskell можно напи-

сать очень даже реальную программу. За примером далеко ходить не буду: сама эта книга построена с помощью Haskell, о чём я подробнее расскажу в следующих главах.

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

Глава 7

Выбираем и возвращаемся

В этой главе мы встретимся с условными конструкциями, выглянем в терминал, а также узнаем, почему из Haskell-функций не возвращаются.

Выглянем во внешний мир

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

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

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

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

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

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

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

```
$ stack install
```

После чего запускаем:

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

Выбор и выход

Существует несколько способов задания условной конструкции. Вот базовый вариант:

```
if CONDITION then EXPRESSION1 else EXPRESSION2
```

где `CONDITION` — логическое выражение, дающее ложь или истину, `EXPRESSION1` — выражение, используемое в случае `True`, `EXPRESSION2` — выражение, используемое в случае `False`. Пример:

```
checkLocalhost :: String -> String
checkLocalhost ip =
  if ip == "127.0.0.1" || ip == "0.0.0.0"
    then "It's a localhost!"
    else "No, it's not a localhost."
```

Функция `checkLocalhost` применяется к единственному аргументу типа `String` и возвращает другое значение типа `String`. В качестве аргумента выступает строка, содержащая IP-адрес, а функция проверяет, не лежит ли в ней `localhost`. Оператор `||` — стандартный оператор логического «ИЛИ», а оператор `==` — стандартный оператор проверки на равенство. Итак, если строка `ip` равна `127.0.0.1` или `0.0.0.0`, значит в ней `localhost`, и мы возвращаем строку `It's a localhost!`, в противном случае возвращаем строку `No, it's not a localhost..`

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

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

при запуске увидим это:

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

а если так:

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

тогда увидим это:

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

Круглые скобки включают выражение типа `String` по схеме:

```
main :: IO ()
main = putStrLn (checkLocalhost "173.194.22.100")
                |--- выражение типа String ---|
```

То есть функция `putStrLn` видит в конечном итоге лишь результат применения функции `checkLocalhost`. Если бы мы опустили скобки и написали так:

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

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

```
main = putStrLn      checkLocalhost  "173.194.22.100"
      функция        к этому
      применяется   аргументу??

                        или к этому??
```

Не знаю как вы, а я не очень люблю круглые скобки, при всём уважении к Lisp-программистам. К счастью, в Haskell существует способ написать такой код без скобок, и он будет работать как ожидается. Об этом способе — в одной из последующих глав.

Вернёмся к возвращению из функции. Вспомним о равенстве в определении:

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

Применение функции `checkLocalhost` к строке объявлено равным условной конструкции. А если так, то эти два кода эквивалентны:

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

Мы просто заменили аргумент `ip` конкретным значением `173.194.22.100`. В итоге, в зависимости от истинности или ложности проверок на равенство, эта условная конструкция будет также заменена одним из двух выражений. В этом и заключается идея: возвращаемое функцией значение — это её последнее, итоговое выражение. То есть если выражение:

```
"173.194.22.100" == "127.0.0.1" ||  
"173.194.22.100" == "0.0.0.0"
```

даст нам результат `True`, то мы работаем с выражением из логической ветки `then`. Если же оно даст нам `False` — мы работаем с выражением из логической ветки `else`. Это даёт нам право утверждать, что условная конструкция вида:

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

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

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

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

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

эквивалентен коду:

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

Аналогично, код:

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

есть ни что иное, как:

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

Каким бы сложным ни было логическое ветвление внутри функции `checkLocalhost`, в конечном итоге оно вернёт/вычислит какое-то одно итоговое выражение. Вот почему функции в Haskell так просто компоновать друг с другом, и позже мы будем встречать всё больше таких примеров.

Глава 8

Выбор и образцы

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

Не только из двух

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

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

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

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

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

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

```
if | CONDITION1 -> EXPRESSION1
   | CONDITION2 -> EXPRESSION2
   | ...
   | CONDITIONn -> EXPRESSIONn
   | otherwise  -> COMMON_EXPRESSION
```

где `CONDITION1..n` — выражения, дающие ложь или истину, а `EXPRESSION1..n` — соответствующие им результирующие выражения. Слово `otherwise` соответствует общему случаю, когда ни одно из логических выражений не дало `True`, и в этой ситуации результатом условной конструкции послужит выражение `COMMON_EXPRESSION`.

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

```
analyzeGold :: Int -> String
```

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

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

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

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

Проверка получилась неполной, вот и получите ошибку.

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

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

```
{-# LANGUAGE MultiWayIf #-} -- ???  
  
module Main where  
  
analyzeGold :: Int -> String  
analyzeGold standard =  
  if | standard == 999 -> "Wow! 999 standard!"  
    | standard == 750 -> "Great! 750 standard."  
    | standard == 585 -> "Not bad! 585 standard."  
    | otherwise -> "I don't know such a standard..."
```



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

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

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

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

Без Если

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

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

```
| otherwise      = "I don't know such a standard..."
```

Ключевое слово `if` исчезло за ненадобностью. Схема здесь такая:

```
function arg -- <<< Нет знака равенства
| CONDITION1 = EXPRESSION1
| CONDITION2 = EXPRESSION2
| ...
| CONDITIONn = EXPRESSIONn
| otherwise  = COMMON_EXPRESSION
```

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

```
function arg |   CONDITION1 = EXPRESSION1 |   ...
эта           либо                               равна этому
функция                               выражению

                               в случае
                               истинности
                               этого
                               выражения

                               либо и т.д.
```

То есть перед нами уже не одно определение функции, а цепочка определений, потому нам и не нужно ключевое слово `if`. А ведь в ряде случаев эту цепочку можно сделать ещё более простой.

Сравнение с образцом

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

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

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

	<code>analyzeGold</code>	<code>999</code>	<code>=</code>	<code>"Wow! 999 standard!"</code>
если	эта функция	применяется	тогда	этому выражению
		вот к этому	она	
		аргументу	равна	
	<code>analyzeGold</code>	<code>750</code>	<code>=</code>	<code>"Wow! 999 standard!"</code>
если	эта функция	применяется	тогда	другому выражению
		к другому	она	
		аргументу	равна	
...				
	<code>analyzeGold</code>	<code>_</code>	<code>=</code>	<code>"I don't know such a standard..."</code>
в				
противном	эта функция	просто	общему	выражению
случае		равна		

Таким образом, когда функция `analyzeGold` применяется к конкретному аргументу, этот аргумент последовательно сравнивается с образцом (англ. `pattern matching`). Образца здесь три: `999`, `750` и `585`. И

если раньше мы сравнивали аргумент с этими числовыми значениями явно, посредством функции `==`, теперь это происходит скрыто. Идея сравнения с образцом очень проста: что-то (в данном случае реальный аргумент функции `analyzeGold`) сопоставляется с образцом (или образцами) на предмет «подходит/не подходит». Если подходит — то есть сравнение с образцом даёт результат `True` — готово, используем соответствующее выражение. Если же не подходит — переходим к следующему образцу.

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

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

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

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

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

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

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

case

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

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

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

```
case EXPRESSION of  
  PATTERN1 -> EXPRESSION1  
  PATTERN2 -> EXPRESSION2  
  ...  
  PATTERNn -> EXPRESSIONn  
  otherwise -> COMMON_EXPRESSION
```

где `EXPRESSION` — анализируемое выражение, последовательно сравниваемое с образцами `PATTERN1..n`. Если ни одно ни сработало — как обычно, упираемся в `otherwise` и выдаём `COMMON_EXPRESSION`.

В последующих главах мы встретимся с другими видами паттерн матчинга.

Глава 9

Пусть будет там, Где...

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

Пусть

Рассмотрим следующую функцию:

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

Мы считаем время некоторого события, и если исходное время меньше 40 секунд — результирующее время увеличено на 120 секунд, в противном случае ещё на 8 секунд сверх того. К сожалению, перед нами классический пример «магических чисел» (англ. *magic numbers*), когда смысл конкретных значений скрыт за семью печатями. Что за 40, и что за 8? Во избежание этой проблемы

можно ввести временные выражения, и тогда код станет совсем другим:

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

Вот, совсем другое дело! Мы избавились от «магических чисел», введя поясняющие выражения `threshold`, `correction` и `delta`, и код функции стал куда понятнее.

Конструкция `let-in` вводит поясняющие выражения по схеме:

```
let DECLARATIONS in EXPRESSION
```

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

```
let threshold = 40
```

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

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

Эта конструкция легко читается:

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


пусть это будет этому в том
 выражение равно выражению выражении

С помощью ключевого слова `let` можно ввести сколько угодно пояснительных/промежуточных выражений, что делает наш код, во-первых, понятнее, а во-вторых, короче. Да, в этом конкретном случае код стал чуть длиннее, но в последующих главах мы увидим ситуации, когда промежуточные значения сокращают код в разы.

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

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

В этом случае мы сократили область видимости промежуточного выражения `delta`, сделав его видимым лишь в выражении `timeInS + delta + correction`.

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

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

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

Где

Существует иной способ введения промежуточных выражений:

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

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

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

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

Вместе

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

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

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

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

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

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

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

```
correction = timeInS * 2
```

Выражение `delta` теперь задано первым по счёту, но это не имеет никакого значения. Ведь мы всего лишь объявляем равенства, и результат этих объявлений не влияет на конечный результат.

Запомните упоминание о неважности порядка введения выражений! К этой теме мы вернёмся в одной из следующих глав, которая откроет нам очередную тайну Haskell.

Конечно, порядок введения не важен и для `let`-выражений:

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

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

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

При попытке скомпилировать такой код мы получим ошибку:

Not in scope: ‘threshold’

Таково ограничение: использовать let-выражения внутри where-выражений невозможно.

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

Глава 10

Мир операторов

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

Вспомним наше самое первое выражение:

$1 + 2$

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

$(+) \ 1 \ 2$

Видите круглые скобки? Они говорят о том, что данная функция предназначена для инфиксной записи. Авторы этой функции изначально рассчитывали на инфиксную форму $1 + 2$, а не на $(+) \ 1 \ 2$, поэтому в определении имя функции заключено в круглые скобки:

$(+) :: \dots$

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

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

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

При обычной форме её применение выглядело бы так:

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

Но давайте перепишем в инфиксной форме:

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

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

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

Строго говоря, название «оператор» весьма условно, мы можем его

и не использовать. Говорить о функции сложения столь же корректно, как и об операторе сложения.

Зачем это нужно?

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

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

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

Идентификаторы `div`, `class_` и `href` — это имена функций, а символы `!` и `$` — это операторы, записанные в своей обычной, инфиксной форме. Самое главное, что нам абсолютно необязательно знать, как они определены и как работают, чтобы понять этот «Haskell-based HTML». А в дальнейших главах мы не только встретимся с другими EDSL, но и заглянем во внутренности некоторых из них.

Вы спросите, а откуда взялись все эти `div` и `!`? Отвечаю: они взялись из конкретной Haskell-библиотеки. С библиотеками мы вскоре познакомимся.

Глава 11

Список: знакомство

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

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

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

Вот список из трёх целых чисел:

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

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

```
[1.3, 45.7899]
```

а вот и список из одного-единственного символа:

```
[ 'H' ]
```

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

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

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

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

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

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

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

Список может быть и пустым:

```
[]
```

Тип списка

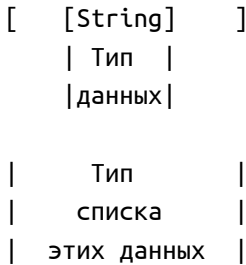
Раз список представляет собой структуру, содержащую данные некоторого типа, возникает вопрос: как указать тип списка? Вот так:

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

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

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

Модель очень проста:



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

Действия над списками

Если списки создаются — значит это кому-нибудь нужно. Со списком можно делать очень много всего. В стандартной Haskell-библиотеке существует отдельный модуль `Data.List`, включающий

широкий набор функций, работающих со списком. Откроем модуль `Main` и импортируем в него модуль `Data.List`:

```
module Main where
```

```
import Data.List
```

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

```
main = putStrLn (head ["Vim", "Emacs", "Atom"])
```

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

```
Vim
```

Модель такая:

```
["Vim",      "Emacs", "Atom"]  
 | голова|   |   хвост   |
```

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

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

```
main = print (tail ["Vim", "Emacs", "Atom"])
```

вот результат:

```
["Emacs", "Atom"]
```

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

и выводит это значение на консоль уже в виде строки.

Но как же, спросите вы, функция `print` узнаёт, как именно отобразить значение в виде строки? О, это интереснейшая тема, но она относится к Третьему Киту Haskell, до знакомства с которым нам ещё далеко.

Можно получить длину списка:

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

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

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

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

Ключевое слово `type` вводит псевдоним для уже существующего типа. Читается это так:

```
type String = [Char]
тип этот равен тому
```

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

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

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

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

А можно и так:

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

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

Неизменность списка

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

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

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

Результат этой программы таков:

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

Рассмотрим определение функции `addTo`:

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

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

<code>newHost</code>	<code>:</code>	<code>hosts</code>
беру	и	в начало
это	добавляю	этого
значение	его	списка

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

Теперь, после знакомства со списком, мы будем использовать их постоянно. И в Haskell эта простая структура данных куда мощнее, чем может показаться на первый взгляд.

Глава 12

Кортеж

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

Кортеж (англ. tuple) — ещё одна стандартная структура данных, с которой нам следует познакомиться. В отличие от списка, кортеж может содержать данные как одного типа, так и разных.

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

Вот как он выглядит:

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

Круглые скобки, в отличие от списка. Этот кортеж содержит два элемента типа `String` и `Int` соответственно. Можно написать и так:

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

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

Тип кортежа

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

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

```
(String, String)
```

Вот ещё пример:

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

И ещё:

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

То есть тип кортежа явно отражает его содержимое. Поэтому если функция применяется к кортежу из двух строк, применить её к кортежу из трёх не получится, потому что типы таких кортежей различаются:

```
-- Несовместимые типы
```

```
(String, String)
```

```
(String, String, String)
```

Действия над кортежами

Со списками можно делать много всего, а вот с кортежами — не очень. Самое частое действие в отношении уже созданного кортежа — извлечение хранящихся в нём данных. Например:

```
makeAlias :: String -> String -> (String, String)
```

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

Пожалуй, ничего проще придумать нельзя: на входе два аргумента, на выходе — двуэлементный кортеж с этими аргументами.

Двуэлементный кортеж называют ещё парой (англ. pair). И хотя кортеж может содержать сколько угодно элементов, на практике пары встречаются чаще всего.

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

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

это к этому

а это к тому

Мы просто пишем прямое соответствие между левой и правой сторонами определения. Ничего удобнее и проще и придумать нельзя. И если бы мы хотели получить кортеж из трёх элементов с дублированием хоста (ну вдруг), это выглядело бы так:

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

это к этому и к этому

а вот к тому
это

Таким же образом, через паттерн матчинг, мы извлекаем элементы из кортежа. Например:

```
main :: IO ()
main =
    let (host, alias) = makeAlias "173.194.71.106"
```

```

                                "www.google.com"
in print (host ++ ", " ++ alias)

```

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

```

main :: IO ()
main =
    let pair = makeAlias "173.194.71.106"
                        "www.google.com"
        host = fst pair
        alias = snd pair
    in print (host ++ ", " ++ alias)

```

При запуске этой программы мы получим:

```
"173.194.71.106, www.google.com"
```

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

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

```

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

```

это

ХОСТ

а вот это значение

это

имя

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

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

то, что ты в конечном итоге вычислишь

это вот

такая пара

Это «зеркальная» модель: через паттерн матчинг формируем:

```
-- Формируем правую сторону
```

```
-- на основе левой...
```

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

и через него же извлекаем:

```
-- Формируем левую сторону
```

```
-- на основе правой...
```

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

Вот ещё один пример работы с кортежем через паттерн матчинг:

```
chessMove :: String -> (String, String) -> (String, (String, String))
```

```
chessMove color (from, to) = (color, (from, to))
```

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

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

И на выходе получаем:

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

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

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

Кстати, я об этом не упомянул, но теоретически кортеж может состоять из одного-единственного элемента:

```
useless :: String -> (String)
useless s = (s)
```

```
main :: IO ()
main = putStrLn s
  where (s) = useless "some"
```

Хотя, учитывая гетерогенность кортежа, мне трудно представить ситуацию, в которой одноэлементный кортеж был бы реально полезен.

Не всё

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

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

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

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

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

Рассмотрим внутренность функции `patientEmail`:

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

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

А если ошиблись?

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

```
type UUID      = String
type FullName  = String
type Email     = String
type Age       = Int
type DiseaseId = Int
type Patient = (UUID, FullName, Email, Age, DiseaseId)
```

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

```
-- А где пятый элемент?
patientEmail :: Patient -> Email
patientEmail (_, _, email, _) = email
```

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

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

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

На этом наше знакомство с кортежем считаю завершённым. Далее мы будем использовать их периодически, а в одной из следующих глав мы узнаем ещё об одном действии в отношении кортежа.

Глава 13

Лямбда-функция

Теперь мы должны познакомиться с важной концепцией — с лямбда-функцией. Потому что именно с неё всё и началось. Приготовьтесь: в этой главе нас ждут новые открытия.

Истоки

В далёких 1930-х молодой американский математик [Алонзо Чёрч](#) задался вопросом о том, что значит «вычислить» что-то. Плодом его размышлений явилась система для формализации понятия «вычисление», и назвал он её «лямбда-исчислением» (англ. lambda calculus, по имени греческой буквы λ).

В основе этой системы лежит лямбда-функция, которую можно считать «матерью функционального программирования» в целом и Haskell в частности. Далее буду называть её ЛФ.

В отношении ЛФ можно смело сказать: «Всё гениальное просто». Идея ЛФ столь полезна именно потому, что она предельно проста.

ЛФ — это анонимная функция. Вот как она выглядит в Haskell:

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

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

```
λx . x * x
```

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

ЛФ представляет собой простейший вид функции, эдакая функция, раздетая догола. У неё забрали не только объявление, но и имя, оставив лишь необходимый минимум в виде имён аргументов и внутреннего выражения. Алонзо Чёрч понял: чтобы применить функцию, вовсе необязательно её именовать. И если у обычной функции сначала идёт объявление/определение, а затем (где-то) применение с использованием имени, то у ЛФ всё куда проще: мы её определяем и тут же применяем, на месте. Вот так:

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

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

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

лямбда-выражение аргумент

Лямбда-выражение порождает временную функцию, которую мы сразу же применяем к аргументу 5.

ЛФ с одним аргументом ещё называют «ЛФ от одного аргумента» или «ЛФ одного аргумента».

Строение

Строение ЛФ предельно простое:

<code>\</code>	<code>x</code>	<code>-></code>	<code>x * x</code>
признак	имя		выражение
ЛФ	аргумента		

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

<code>\</code>	<code>x</code>	<code>y</code>	<code>-></code>	<code>x * y</code>
признак	имя 1	имя 2		выражение
ЛФ	аргумента	аргумента		

И когда мы применяем такую функцию:

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

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

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

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

Мы можем ввести промежуточное значение для ЛФ:

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

Здесь выражение `mul` объявляется равным ЛФ, и теперь мы применяем `mul` так же, как если бы это было само лямбда-выражение:

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

И здесь мы приблизились к одному важному открытию.

Тип функции

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

```
where mul = \x y -> x * y -- Тип?
```

Ответ прост: тип `mul` такой же, как и у этой ЛФ. Из этого мы делаем важный вывод: ЛФ имеет тип, как и обычные данные. Но если ЛФ тоже является функцией (просто предельно лаконичной) — значит и у обыкновенной функции тоже есть тип!

В императивных языках между функциями и данными проведена чёткая граница: вот это функции, а вон то — данные. Однако в Haskell между данными и функциями разницы нет, ведь и то и другое порождается неким выражением. И вот каков тип функции `mul`:

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

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

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

```
вот  имеет  | вот такой |
это  тип
```

Точно так же мы можем указать тип любых других данных:

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

Хотя мы знаем, что в Haskell типы выводятся автоматически, ино-

гда мы хотим взять эту заботу на себя. В данном случае мы явно говорим: «Пусть выражение `coeff` будет равно 12, но тип пусть имеет `Double`, а не `Int`». Так же и с функцией: когда мы объявляем её — мы тем самым указываем её тип.

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

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

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

И раз функция характеризуется типом как и все другие данные, мы делаем ещё одно важное открытие: функциями можно оперировать как данными. И запомните эту важную и простую мысль!

Например, можно создать список функций:

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

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

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

| первая | её

	функция		аргумент
	из списка		

Это равносильно коду:

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

Поэтому при запуске программы мы получим:

```
Hi val1
```

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

Локальные функции

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

```
import Data.List

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

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

```
where
  my = "haskeller@gmail.com"
```

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

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

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

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

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

Вот, теперь уже сомнений не возникает. Указывать тип примитивных функций, как правило, необязательно.

А вот как этот код выглядит с ЛФ:

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

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

```
containsAtSign = (\e -> "@" `isInfixOf` e) :: String -> Bool
endsWithCom = (\e -> ".com" `isSuffixOf` e) :: String -> Bool
```

Лямбда-выражение взято в скобки, чтобы указание типа относилось к выражению в целом, а не только к аргументу `e`. Впрочем, на практике указание типа ЛФ встречается нечасто, ибо незачем.

Вот мы и познакомились с «матерью Haskell». Теперь мы будем использовать ЛФ периодически.

И напоследок, вопрос. Помните тип функции `mul` в начале главы?

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

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

Глава 14

Композиция функций

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

Скобкам — бой!

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

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


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

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

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

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

```

      +-----+               +-----+
String ->| checkLocalhost |-> String ->| putStrLn |-> ...
      +-----+               +-----+

```

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

Композиция и применение

Взгляните:

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

Необычно? Перед нами два новых стандартных оператора, избавляющие нас от лишних скобок и делающие наш код проще.

Оператор `.` — это оператор композиции функций (англ. function composition), а оператор `$` — это оператор применения (англ. application operator). Эти операторы часто используют совместно друг с другом.

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

```
putStrLn . checkLocalhost
```

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

```

      +-----+      +-----+
String ->| checkLocalhost |-> String ->| putStrLn |-> ...
      +-----+      +-----+

```

A B C

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

```

+-----+
String ->| checkLocalhost + putStrLn |-> ...
+-----+

```

A C

В данном случае знак + не относится к конкретному оператору, я лишь показываю факт «объединения» двух функций в третью. Разумеется, промежуточная точка в никуда не исчезла, просто она теперь скрыта от наших глаз.

И теперь нам стало понятнее, почему в типе функции, в качестве разделителя, используется стрелка:

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

в нашем примере это:

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

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

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

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

объединённая функция аргумент

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

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

объединённая функция применяется к аргументу

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

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

<- <- <- аргумент

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

```
putStrLn . checkLocalhost
```

эта применяется этой
функция после функции

То есть композиция правоассоциативна: сначала применяется функция справа, а затем — слева. Красота композиции в том, что компоновать мы можем сколько угодно функций:

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

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

Функция `logWarn` готовит переданную ей строку для записи в журнал. Функция `asciiOnly` готовит строку к выводу в нелокализованном терминале, функция `correctSpaces` убирает дублирующиеся пробелы, а функция `warning` делает строку предупреждением (например, добавляет строку “WARNING:” в начало сообщения). Таким образом, при запуске этой программы мы увидим:

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

Здесь мы объединили в «функциональный конвейер» уже три функции, безо всяких скобок. Более того, определение функции `logWarn` можно сделать ещё более простым:

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

Погодите, но где же имя аргумента? Его больше нет, оно нам не

нужно. Вспомните, что применение функции может быть легко заменено внутренним выражением функции. А раз так, выражение `logWarn` может быть заменено на выражение `warning . correctSpaces . asciiOnly`. Сделаем же это:

```
logWarn "Province  'Gia Vi\n' isn't on the map! " =
(warning
 . correctSpaces
 . asciiOnly) "Province  'Gia Vi\n' isn't on the map! " =
warning
 . correctSpaces
 . asciiOnly $ "Province  'Gia Vi\n' isn't on the map! "
```

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

Как работает композиция

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

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

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

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

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

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

Подставим наши функции:

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

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

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

  putStrLn (checkLocalhost "173.194.22.100")
```

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

Глава 15

ФВП

ФВП, или Функции Высшего Порядка (англ. HOF, Higher Order Functions) — важная концепция в Haskell. Однако, как ни странно, мы с ней уже познакомились. Как мы узнали из предыдущих глав, функциями можно оперировать как значениями, в частности, использовать их в качестве аргументов функций и возвращать как результат функции. Так вот функции, оперирующие другими функциями как аргументами и/или как результирующим выражением, носят название функций высшего порядка.

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

Отображение

Рассмотрим функцию `map`. Эта стандартная функция используется для отображения (англ. *mapping*) функции на элементы списка. Вот её объявление:

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

Вот опять эти маленькие буквы! Помните, я обещал рассказать о них? Рассказываю: малой буквой принято именовать полиморфный (англ. *polymorphic*) тип. Полиморфизм — это многообразность, многоформенность. В данном случае речь идёт не об указании конкретного типа, а о «типовой заглушке». Мы говорим: «Функция `map` применяется к функции из какого-то типа `a` в какой-то тип `b` и к списку типа `[a]`, а результат её работы — это другой список типа `[b]`». На место типовых заглушек могут встать любые конкретные типы, в зависимости от того, к чему мы применим функцию `map`. Это делает данную функцию очень гибкой.

Например:

```
import Data.Char

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

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

Результатом работы этой программы будет строка:

```
HASKELL.ORG
```

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


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

Вот её объявление:

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

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

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

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

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

А теперь вспомним о природе типа String:

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

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

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

Так, на первом шаге функция toUpper будет применена к элементу 'h', на втором — к элементу 'a', и так далее до последнего элемента 'g'. Когда функция map бежит по этому списку, результат применения функции toUpper к его элементам служит элементами для второго списка, который будет в конечном итоге возвращён. Так,

результатом первого шага будет элемент 'н', результатом второго — элемент 'а', а результатом последнего — элемент 'г'. Вот и получается:

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

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

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

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

```
main :: IO ()
main = print . toStr $ [1.2, 1.4, 1.6]
```

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

```
["1.2", "1.4", "1.6"]
```

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

```
show 1.2 = "1.2"
```

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

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

Подставим в сигнатуру функции `map`:

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

Именно так, как у нас и есть:

```
map show [1.2, 1.4, 1.6] = ["1.2", "1.4", "1.6"]
```

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

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

```
main :: IO ()
main = print . ten $ [1.2, 1.4, 1.6]
```

Результат работы:

```
[12.0, 14.0, 16.0]
```

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

Композиция для отображения

Если мы можем передать функции `map` некую функцию для работы с элементами списка, значит мы можем передать ей и композицию

функций. Вот как это может выглядеть:

```
import Data.Char

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

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

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

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

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

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

Глава 16

Наскаге и библиотеки

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

Библиотеки большие и маленькие

За годы существования Haskell разработчики со всего мира создали множество библиотек. Библиотеки избавляют нас от необходимости вновь и вновь писать то, что уже написано до нас.

Для любого живого языка программирования написано множество библиотек. В мире Haskell их, конечно, не такая туча, как для той же Java, но порядочно: стабильных есть не менее двух тысяч, многие из которых очень качественные и уже многократно испытаны в серьёзных проектах.

С модулями — файлами, содержащими Haskell-код, — мы уже знакомы, они являются основным кирпичом Haskell-проекта. Библио-

тека, также являясь Haskell-проектом, тоже состоит из модулей (не важно, из одного или из сотен). Поэтому использование библиотеки сводится к использованию входящих в неё модулей. И мы уже неоднократно делали это в предыдущих главах.

Вспомним пример из главы про ФВП:

```
import Data.Char

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

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

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

Hackage

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

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

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

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

Второй способ — через специальный поисковик, коих существует два:

1. [Hoogle](#)
2. [Hayoo!](#)

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

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

чае понадобится, поэтому включим его в наш проект незамедлительно.

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

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

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

Как видите, пакет `base` уже тут. Включив пакет `text` в секцию `build-depends`, мы объявили тем самым, что наш проект отныне зависит от этого пакета. Теперь, находясь в корне проекта, выполняем уже знакомую нам команду:

```
$ stack install
```

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

Иерархия в имени

Когда мы пишем:


```
import Data.Char
```

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

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

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

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

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

Лицо

Вернёмся к нашему примеру:

```
import Data.Char
```

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

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

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

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

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

В настоящий момент только функция `someFunc` доступна всем импортирующим данный модуль. Если же мы определим в этом модуле другую функцию `anotherFunc`:

```
module Lib (
    someFunc
) where

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

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

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

```
module Main

import Lib

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

компилятор справедливо ругнётся, мол, не знаю функцию

`anotherFunc`. Если же мы добавим её в интерфейс модуля `Lib`:

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

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

Импортируем по-разному

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

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

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

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

При попытке скомпилировать такой код мы упрёмся в ошибку:

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

```
imported from 'Data.Text.IO' ...
```

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

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

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

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

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

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

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

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

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

включить этот модуль как это

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

Вот, так значительно лучше. Короткое имя может состоять даже из

одной буквы, но как и полное имя модуля, оно обязательно должно начинаться с большой буквы, поэтому:

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

Иногда, для большего порядка, используют `qualified`-импорт:

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

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

```
import qualified Data.Text as T
```

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

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

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

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

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

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

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

импортируем отсюда только
это

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

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

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

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

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

```
main = putStrLn ... -- Сомнений нет: из Prelude.
```

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

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

импортируем всё отсюда кроме этого

Можно и несколько функций скрыть:

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

При желании можно скрыть и из Prelude:

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

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

Оформление

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

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

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

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

```
import Control.Monad (void)
```

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