

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!	10
	Почему эта книга появилась	10
	Цель	10
	О себе	10
	О вас	11
	Обещание	11
2	Первые вопросы	12
	«Что такое этот ваш Haskell?»	12
	«Это что, какой-то новый язык?»	12
	«И кто его сделал?»	12
	«А библиотеки для Haskell имеются?»	13
	«И что, его уже можно в production?»	13
	«А порог вхождения в Haskell высокий?»	13
	«А я слышал ещё про какие-то монады...»	13
	«А если сравнить его с C++/Python/Scala...»	13
3	Об этой книге	14
	Чего здесь нет	14
	О первом и втором издании	14
	Читайте последовательно	15
	Для любопытных	15
	О пояснениях	15

Благодарность	16
Слово к читавшим первое издание	16
4 Приготовимся	18
Устанавливаем	18
Разворачиваем инфраструктуру	19
Hi World	19
Модули: знакомство	20
Для любопытных	21
5 Киты и Черепаха	22
Черепаха	22
Первый Кит	23
Второй Кит	24
Третий Кит	26
Для любопытных	26
6 Неизменность и чистота	27
Объявляем и определяем	27
Чисто функциональный	29
«Присваивание? Не, не слышал...»	29
Для любопытных	30
7 Выбираем и возвращаемся	31
Выглянем во внешний мир	31
Выбор и выход	32
Для любопытных	35
8 Выбор и образцы	36
Не только из двух	36
Без Если	38
Сравнение с образцом	39
case	40

9 Пусть будет там, Где...	42
Пусть	42
Где	44
Вместе	44
10 Мир операторов	47
Зачем это нужно?	48
11 Список	49
Тип списка	50
Действия над списками	50
Неизменность списка	52
Перечисление	53
Для любопытных	55
12 Кортеж	56
Тип кортежа	56
Действия над кортежами	57
Не всё	60
А если ошиблись?	60
Для любопытных	61
13 Лямбда-функция	63
Истоки	63
Строение	64
Тип функции	64
Локальные функции	66
Для любопытных	68
14 Композиция функций	69
Скобкам — бой!	69
Композиция и применение	70
Длинные цепочки	72

Как работает композиция	73
15 ФВП	75
Отображение	75
Частичное применение	78
Композиция для отображения	80
16 Hackage и библиотеки	82
Библиотеки большие и маленькие	82
Hackage	83
Иерархия в имени	84
Лицо	84
Импортируем по-разному	85
Оформление	88
17 Лень	89
Начнём с C++	89
Вот как в Haskell	90
Аргументы	91
Строгость	93
18 Рекурсия	94
19 Генераторы списков	95
Полнота и красота	95
Хитрый список	95
Добавляем предикат	96
Больше списков	97
Добавляем условие	98
Добавляем локальное выражение	98
Пример	98
В сухом остатке	99
20 Диапазоны	100

Суть	100
Умные диапазоны	100
Без конца	101
В сухом остатке	102
21 Наши типы	103
Знакомство	103
Значение-пустышка	104
22 АТД	106
Извлекаем значение	107
Строим	108
23 АТД: поля с метками	110
Метки	110
Getter и Setter?	112
Без меток	114
24 Конструктор типа	116
25 Новый тип	117
Один конструктор значения	117
Одно поле	117
Для чего он нужен	118
В сухом остатке	118
26 Stackage	119
Проблема	119
Что это такое?	119
Неизменность	120
Утилита stack?	120
А если не из репозитория?	120
Ещё почитать	121

27 О форматировании	122
Функция	122
Тип	124
Класс типов	124
Константа	125
Условие	125
Локальные выражения	126
Вывод	127

Глава 1

Приветствую!

Перед вами — книга о Haskell, удивительном и прекрасном 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 неё! — последовательности Фибоначчи. Эта книга не такая: минимум академизма, максимум практичности.

Цель

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

О себе

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

О вас

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

Обещание

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

Глава 2

Первые вопросы

Мне задавали их множество раз. Отвечаю.

«Что такое этот ваш Haskell?»

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

«Это что, какой-то новый язык?»

Вовсе нет. История Haskell началась ещё в 1987 году. Этот язык был рождён в математических кругах, когда группа людей решила создать лучший функциональный язык программирования. В 1990 году вышла первая версия языка, названного в честь известного американского математика [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. Наиболее известная реализация языка нашла своё воплощение в компиляторе GHC (The Glasgow Haskell Compiler), родившегося в 1989 году в Университете Глазго. У компилятора было несколько главных разработчиков, из которых наиболее известны двое, [Simon Peyton Jones](#) и [Simon Marlow](#). Впоследствии весомый вклад в разработку GHC внесли ещё несколько сотен человек. Исходный код компилятора GHC [открыт](#). Кстати, сам компилятор на 82% написан на Haskell.

Для любопытных: исчерпывающее повествование об истории Haskell и GHC читайте [здесь](#).

«А библиотеки для Haskell имеются?»

О да! 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 библиотеки, выходили в свет книги. Сегодня, в 2016, можно уверенно заявить, что Haskell полностью готов к серьёзному коммерческому использованию, о чём свидетельствуют истории успешного внедрения Haskell в бизнесе, в том числе [крупном](#).

«А порог вхождения в Haskell высокий?»

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

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

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

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

Сравнение Haskell с другими языками выходит за рамки этой книги. Несколько раз вы встретите здесь кусочки кода на других языках, но я привожу их исключительно для того, чтобы подчеркнуть различие с Haskell, а вовсе не для сравнения в контексте «лучше/хуже».

Глава 3

Об этой книге

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

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

Во-вторых, я разрушу страх. Уже много лет вокруг Haskell витает дух страха, и я сполна ощутил его на себе. В действительности Haskell совсем 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. Исчерпывающего 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 перепи-

сать книгу.

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

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

Читайте последовательно

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

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

Для любопытных

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

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

О пояснениях

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

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

тип этот равен тому

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

Вот ещё один пример:

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

	данное значение	
это		
хост		
		а вот это
		значение
это		
имя		

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

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

_____	_____
=====	=====

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

Благодарность

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

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

Слово к читавшим первое издание

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

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

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

И я по-прежнему открыт для вашей критики.

Глава 4

Приготовимся

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

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

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

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

Устанавливаем

Идём [сюда](#) и скачиваем архив для нужной нам ОС. Распаковываем архив и видим программку под названием `stack`. Для удобства располагаем её в каком-нибудь каталоге, доступном в PATH. 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 в точности так, как описано ниже, поскольку `stack` активно развивается.

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

1. Разворачивать инфраструктуру.
2. Собирать проекты.

3. Устанавливать библиотеки.

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

Haskell-проект — среда для создания приложений и библиотек.

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

Разворачиваем инфраструктуру

Делаем:

```
$ stack setup
```

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

Hi World

Создадим наш первый Haskell-проект:

```
$ stack new real
```

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

```
.
├─ LICENSE
├─ Setup.hs
├─ app
│   └─ Main.hs <- Главный модуль
├─ real.cabal <- Сборочный файл проекта
├─ src
│   └─ Lib.hs <- 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 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: знакомство

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

Откроем модуль `src/Lib.hs`:

```
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 этой конструкции, в следующих главах мы разберём его тщательнейшим образом.

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 иное.

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

Для любопытных

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

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

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

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. И как же можно «осуществить сложение» в рамках Haskell-программы? С помощью функции. Именно функция делает выражение вычислимым, именно она оживляет нашу программу, потому я и назвал Функцию Первым Китом Haskell. Но дабы избежать недоразумений, определимся с понятиями.

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

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

```
sqaure 2
```

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

```
sum 10 20
```

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

```
(+) 1 2
```

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

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

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

Второй Кит

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

```
square 2  
функция аргумент
```

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

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

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

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

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

Статическая проверка

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

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

Сила

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

```
double coeff(double base) {  
    return base * 4.9856;  
}  
  
int main() {  
    int value = coeff(122.04);  
    ...  
}
```

Это канонический пример проблемы, обусловленной слабой (англ. *weak*) системой типов. Функция `coeff` возвращает значение типа `double`, однако вызывающая сторона ожидает почему-то целое число. Ну вот ошиблись мы, криво скопировали. В этом случае произойдёт жульничество, называемое скрытым приведением типов (англ. *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 ожидаемого типа фактическому, поэтому когда компиляция завершается успешно, мы абсолютно уверены в гармонии между типами всех наших данных.

Выведение

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

```
double value = 122.04;
```

однако в Haskell мы напишем просто:

```
value = 122.04
```

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

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

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

123	<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, соответствующий истине или лжи. В последующих главах мы встретимся ещё с несколькими стандартными типами, но пока хватит и этих. И заметьте: имя типа в Haskell всегда начинается с большой буквы.

Третий Кит

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

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

Для любопытных

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

Глава 6

Неизменность и чистота

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

Объявляем и определяем

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

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

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

Рассмотрим объявление:

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

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

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

имя	тип	тип
функции	аргумента	вычисленного значения

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

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

имя	тип	тип	тип
функции	первого	второго	вычисленного
	аргумента	аргумента	значения

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

Мы не можем работать с функцией, которая ничего не вычисляет. То есть аналога С-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 С-функции `int f(void)`. С такими функциями мы познакомимся в следующих главах.

Теперь рассмотрим определение функции `square`:

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

Схема определения такова:

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

имя	имя	это	выражение
функции	аргумента		

А функция `prod` определена так:

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

имя	имя	имя	это	выражение
функции	первого	второго		
	аргумента	аргумента		

Определение тоже разделено на две части: слева от знака равенства — имя функции и имена аргументов (имена, а не типы), разделённые пробелами, а справа — выражение, составляющее суть функции, её содержимое. Иногда эти части называют «головой» и «телом»:

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

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

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

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

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

Но откуда у меня такая уверенность? А вот откуда.

Чисто функциональный

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

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

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

«Присваивание? Не, не слышал...»

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

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

```
coeff = 0.569;
```

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

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

```
coeff = 0.569
```

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

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

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

Для любопытных

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

Глава 7

Выбираем и возвращаемся

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

Выглянем во внешний мир

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

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

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

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

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

```
{-
    Я - сложный многострочный
    комментарий, содержащий
    нечто
    очень важное!
-}
main :: IO ()
main =
```

```
-- А я - скромный однострочный комментарий.
putStrLn "Hi, real world!"
```

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

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

```
$ stack install
```

После сборки проект запускаем:

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

Выбор и выход

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

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

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

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

Функция `checkLocalhost` применяется к единственному аргументу типа `String` и возвращает другое значение типа `String`. В качестве аргумента выступает строка, содержащая IP-адрес, а функция проверяет, не лежит ли в ней `localhost`. Оператор `||` — 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 первое выражение, то есть строку `It's a localhost!`, в противном случае возвращаем второе выражение, строку `No, it's not a localhost..`

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

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

при запуске увидим это:

`It's a localhost!`

а если так:

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

тогда увидим это:

`No, it's not a localhost.`

Круглые скобки включают выражение типа `String` по схеме:

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

|__ выражение типа `String` __|

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

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

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

```
main = putStrLn      checkLocalhost "173.194.22.100"
```

функция к этому
применяется аргументу...

и к этому??

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

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

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

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

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

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

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

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

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

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

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

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

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

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

эквивалентен коду:

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

Аналогично, код:

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

есть ни что иное, как:

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

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

Для любопытных

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

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

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

Глава 8

Выбор и образцы

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

Не только из двух

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

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

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

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

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

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

```

if | CONDITION1 -> EXPR1
   | CONDITION2 -> EXPR2
   | ...
   | CONDITIONn -> EXPRn
   | otherwise -> COMMON_EXPRESSION

```

где `CONDITION1..n` — выражения, дающие ложь или истину, а `EXPR1..n` — соответствующие им результирующие выражения. Особая функция `otherwise` соответствует общему случаю, когда ни одно из логических выражений не дало `True`, и в этой ситуации результатом условной конструкции послужит выражение `COMMON_EXPRESSION`.

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

```

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

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

```

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

Non-exhaustive guards in multi-way if

Проверка получилась неполной, вот и получите ошибку.

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

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

```

{-# LANGUAGE MultiWayIf #-} -- А это что?

module Main where

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

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

```

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

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

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

Без Если

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

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

Ключевое слово `if` исчезло. Схема здесь такая:

```
function arg  -- Нет знака равенства?
  | CONDITION1 = EXPR1
  | CONDITION2 = EXPR2
  | ...
  | CONDITIONn = EXPRn
  | otherwise  = COMMON_EXPRESSION
```

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

```
function arg | CONDITION1 = EXPR1 | ...
эта          или          равна
функция                                этому
                                         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`. Но и эту цепочку определений можно упростить.

Сравнение с образцом

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

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

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

	<code>analyzeGold</code>	<code>999</code>	<code>=</code>	<code>"Wow! 999 standard!"</code>
если	эта функция	применяется	тогда	этому выражению
		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 же случае эта функция просто некоторому общему выражению
равна

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

образцу.

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

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

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

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

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

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

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

case

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

```
analyzeGold standard =
  case standard of
    999      -> "Wow! 999 standard!"
    750      -> "Great! 750 standard."
    585      -> "Not bad! 585 standard."
    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 -> EXPR1
  PATTERN2 -> EXPR2
```



```
...  
PATTERNn -> EXPRn  
otherwise -> COMMON_EXPRESSION
```

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

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

Глава 9

Пусть будет там, Где...

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

Пусть

Рассмотрим следующую функцию:

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

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

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

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

```
let DECLARATIONS in EXPRESSION
```

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

```
let threshold = 40
```

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

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

Эта конструкция легко читается:

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

пусть это будет этому в том
выражение равно выражению выражении

С помощью ключевого слова `let` можно ввести сколько угодно пояснительных/промежуточных выражений, что делает наш код, 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
```

это существует лишь в
выражение рамках этого выражения

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

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

```
...
let threshold = 40; correction = 120
in if | timeInS < threshold -> timeInS + correction
    | 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, взгляните:

```
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 задана сверху, 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
  -- Это промежуточное выражение зависит от аргумента...
  correction = timeInS * 2
  -- А это - от другого выражения...
  delta      = correction - 4

```

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

```

...
let threshold = 40
in if | timeInS < threshold -> timeInS + correction
    | 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 вычислений. Конечно, порядок объявления равенств не важен и для `let`-выражений:

```

calculateTime :: Int -> Int
calculateTime timeInS =
  let delta      = correction - 4
      threshold = 40
  in if | timeInS < threshold -> timeInS + correction
      | 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, ведь последние уже не входят в выражение, следующее за словом `in`.

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

Глава 10

Мир операторов

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

Вспомним наше самое первое выражение:

```
1 + 2
```

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

```
(+) 1 2
```

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

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

Функции, предназначенные для инфиксной формы применения, называют операторами.

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

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

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

При обычной форме её применения выглядело бы так:

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

Но давайте перепишем в инфиксной форме:

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

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

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

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

Зачем это нужно?

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

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

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

Идентификаторы `div`, `class_` и `href` — это имена функций, а символы `!` и `$` — это операторы, записанные в инфиксной форме. Самое главное, что для понимания этого кода нам абсолютно необязательно знать, где определены все эти функции/операторы и как они работают. Это важная мысль, которую я неоднократно буду повторять в последующих главах:

Чтобы использовать функции, нам вовсе необязательно знать их внутренности.

А про EDSL запомните, мы с ними ещё встретимся.

Глава 11

Список

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

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

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

Вот список из трёх целых чисел:

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

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

```
[1.3, 45.7899]
```

а вот и список из одного-единственного символа:

```
['H']
```

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

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

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

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

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

```
, ["ARP", "NDP", "OSPF"]
]
```

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

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

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

```
[]
```

Тип списка

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

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

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

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

Модель очень проста:

```
[ [String] ]
```

```
| Тип |
|-----|
| данных |
```

```
|      Тип      |
|      списка   |
|_ этих данных _|
```

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

Действия над списками

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

```
module Main where
```

```
-- Стандартный модуль для работы со списками.
```

```
import Data.List
```

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

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

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

```
Vim
```

Модель такая:

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

```
голова |__ хвост __|
```

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

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

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

Вот результат:

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

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

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

Можно получить длину списка:

```
handleTableRow :: String -> String
```

```
handleTableRow row
```

```
  | length row == 2 = composeTwoOptionsFrom row
```

```
  | length row == 3 = composeThreeOptionsFrom row
```

```
  | otherwise      = invalidRow row
```

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

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

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

Ключевое слово `type` вводит 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 уже знакомые промежуточные выражения, например:

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

А можно и так:

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

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

Неизменность списка

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

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

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

Результат этой программы таков:

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

Рассмотрим определение функции `addTo`:

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

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

```
newHost  :      hosts

          этот
          оператор

берёт
это
значение

          и добавляет
          его в начало
          этого списка
```

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

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

Перечисление

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

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

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

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

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

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

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

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

```

-           -
      ==                ==

```

Мы можем задать шаг и явно:

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

Получили только чётные значения. Схема проста:

```
[2,      4      .. 10]
```

```

первый          конец
      второй

```

```

|  разница  |
|_ даёт шаг _|

```

Вот ещё пример:

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

Можно задать и нисходящий диапазон:

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

Или так:

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

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

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

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

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

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

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

Вот такая красота.

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

Для любопытных

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

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

При попытке скомпилировать такой код увидим ошибку:

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

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

Глава 12

Кортеж

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

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

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

Вот как выглядит кортеж:

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

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

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

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

Тип кортежа

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

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

```
(String, String)
```

Вот ещё пример:

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

И ещё:

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


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

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

Действия над кортежами

Со списками можно делать много всего, а вот с кортежами — не очень. 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 — двухэлементный кортеж с этими аргументами. Двухэлементный кортеж называют ещё парой (англ. pair). И хотя кортеж может содержать сколько угодно элементов, на практике именно пары встречаются чаще всего.

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

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

[illegible]

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

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

[illegible]

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

Извлечение элементов из кортежа также производится через паттерн матчинг:

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

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

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

При запуске этой программы получим:

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

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

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

```

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

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

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

```

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

Это «зеркальная» модель: через паттерн матчинг формируем:

```
-- Формируем правую сторону
-- на основе левой...
```



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

Впрочем, 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 — `Email`.

Рассмотрим внутренность функции `patientEmail`:

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

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

А если ошиблись?

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

```

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

```

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

```

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

      ^   ^   ^           ^   -- А пятый где?

```

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

```

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

```

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

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

Для любопытных

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

```
Data.Tuple.Select
```

```

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

```

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

Кроме того, мы и обновлять элементы кортежа можем:

```
import Data.Tuple.Update
```

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

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

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

Второй элемент кортежа изменился с 45 на 2.

Глава 13

Лямбда-функция

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

Истоки

В далёких 1930-х молодой американский математик [Алонзо Чёрч](#) задался вопросом о том, что значит «вычислить» что-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:

```
 $\lambda x . x * x$ 
```

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

ЛФ представляет собой простейший вид функции, эдакая функция, раздетая до гола. 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 предельно простое:

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

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

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

И когда мы применяем такую функцию:

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

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

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

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

Мы можем ввести промежуточное значение для ЛФ:

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

Здесь выражение `mul` объявляется равным ЛФ, и теперь мы можем применять `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 выражения `mul` из предыдущего примера?


```
where mul = \x y -> x * y -- Какой 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 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 покоеится на одной и той же Черепахе. Вот 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   |
это тип   |_ такой _|
```

Точно так же мы можем указать тип любых других данных:

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

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

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

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

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

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

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

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

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

	первая		её
	функция		аргумент
	__ из списка __		

Это равносильно коду:

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

При запуске программы мы получим:

```
Hi val1
```

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

Локальные функции

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

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

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

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

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

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

Это — две функции, которые мы определили прямо в `where`-секции, поэтому они существуют только для основного выражения функции `validComEmail`. С простыми

функциями так поступают очень часто: где она нужна, там её и определяют. Мы могли бы написать и более явно:

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

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

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

Вот как этот код выглядит с 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 к ЛФ от одного аргумента. В этом случае мы конечно же не указываем тип этих выражений. Впрочем, если очень хочется, можно и указать, например:

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

выражение, порождающее тип этой функции
функцию

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

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

тип аргумента `e`,
а вовсе не всей
функции!

Для типа функции тоже можно ввести псевдоним:

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

validComEmail :: String -> Bool
validComEmail email =
    containsAtSign email && endsWithCom email
    where
```

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

Впрочем, на практике указание типа ЛФ встречается крайне редко, ибо незначем.

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

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

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

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

Для любопытных

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

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

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

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

```
Λx . x * 10
```

Вот так и получилось, лямбда-исчисление.

Глава 14

Композиция функций

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

Скобкам — бой!

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

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

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

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

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

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

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

```

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

```

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

Композиция и применение

Взгляните:

```

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

```

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

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

```
putStrLn . checkLocalhost
```

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

```

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

```

A B C

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

```

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

```

A C

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

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

в нашем примере это:

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

Она показывает наше движение из точки 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. Его схема проста:

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

Для нашей объединённой функции это выглядит так:

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

объединённая функция применяется к этому аргументу

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

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

`<-` `<-` `<-` аргумент

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

```
putStrLn "checkLocalhost"
```

эта функция применяется после этой функции

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

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

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

объединённая функция |_ её аргумент _|

а можем и так:

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

обычная |_____ её аргумент _____|
функция

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

Длинные цепочки

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

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

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

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

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

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

```
warning . correctSpaces . asciiOnly $ rawMessage
```

```
===== ^ =====
```

```
|__ первая композиция __|
```

```
+++++++ ^ ++++++
```

```
|_____ вторая композиция _____|
```

аргумент

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

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

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

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

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

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

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

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

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

Как работает композиция

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

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

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

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

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

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

Подставим наши функции:

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

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

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

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

```
-- Наш собственный оператор композиции.  
(<+>) f g = \x -> f (g x)  
  
...  
  
main :: IO ()  
main = putStrLn <+> checkLocalhost $ "173.194.22.100"
```

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

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

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

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

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

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

пусть

такое
выражение

будет
равно

такому
выражению

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

Глава 15

ФВП

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

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

Отображение

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

Вот объявление функции `map`:

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

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

```
import Data.Char
```

```
toUpperCase :: String -> String
```

```
toUpperCase str = map toUpper str
```

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

Результатом работы этой программы будет строка:

HASKELL.ORG

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

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

Вот её объявление:

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

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

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

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

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

```
_____
          _____
          _____
```

А теперь вспомним о природе типа `String`:

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

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

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

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

```
map toUpper [ 'h' >> [ 'H'
                    , 'a' >> , 'A'
                    , 's' >> , 'S'
                    , 'k' >> , 'K'
                    , 'e' >> , 'E'
```

```

    , 'l' >> , 'L'
    , 'l' >> , 'L'
    , '.' >> , '.'
    , 'o' >> , 'O'
    , 'r' >> , 'R'
    , 'g' >> , 'G'
  ]
]

```

Вот и получается:

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

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

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

```

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

```

```

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

```

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

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

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

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

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

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

Подставим в сигнатуру функции `map`:

```

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

```

```

_____
          =====

```

Именно так, как у нас и есть:

```
map show [1.2, 1.4, 1.6] = ["1.2", "1.0", "4.0", "1.6"]
```

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

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

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

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

Результат работы:

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

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

```
main = print .      ten      $ [1.2, 1.4, 1.6] =
      /      \
     /        \
main = print . map (\n -> n * 10) $ [1.2, 1.4, 1.6]
```

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

Частичное применение

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

Вспомним сокращённое определение функции `ten`:

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

первый	а где же
аргумент	второй??
есть	

Функция `map` получила лишь первый аргумент, а где же второй? Вторым, как мы уже знаем, будет получен ею уже потом, после того, как мы подставим это выражение на место функции `ten`. Но что же происходит с функцией `map` до этого? А до

этого с ней происходит частичное применение. Понятно, что она ещё не может выполнить свою работу, поэтому, будучи применённой лишь к одному аргументу, она возвращает ЛФ! Сопоставим с типом функции `map`, и всё встанет на свои места:

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

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

только первый
аргумент

	частично			
	применённая			
	_____ map _____		аргумент	ответ
			для частично	
			применённой	
			функции map	
				[1.2, 1.4, 1.6]

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

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

голова |_ хвост _|

Поскольку голова в виде первого аргумента типа `(a -> b)` уже дана, осталось получить второй аргумент. Поэтому ЛФ, порождённая частичным применением, ожидает единственный аргумент, которым и будет тот самый второй, а именно список `[1.2, 1.4, 1.6]`.

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

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

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

голова |_____ хвост _____|

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

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

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

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

```
replace "http"
       "https"
```

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

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

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

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

Из этого мы делаем интересное открытие:

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

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

```
main = print . (map (\n -> n * 10)) $ [1.2, 1.4, 1.6]
```

	частично	
	применённая map	
	композиция функции	
	print и частично	
	_____ применённой map _____	
	аргумент для	
	композиции	

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

Композиция для отображения

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

```
import Data.Char

pretty :: [String] -> [String]
pretty = map (stars . big)
```



```
where
  big = map toUpper
  stars = \s -> "*" ++ s ++ " *"
```

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

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

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

Пройтись по списку композицией `stars . big` равносильно тому, как если бы мы 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 любого Haskell-проекта. Библиотека, также являясь Haskell-проектом, тоже состоит из модулей (не важно, из одного или из сотен). Поэтому использование библиотеки сводится к использованию входящих в неё модулей. И мы уже неоднократно делали это в предыдущих главах.

Вспомним пример из главы про ФВП:

```
import Data.Char

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

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

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

Hackage

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

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

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

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

Второй способ — через специальный поисковик, коих существует два:

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

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

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

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

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

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

Как видите, пакет `base` уже тут. Включив пакет `text` в секцию `build-depends`, мы объявили тем самым, что наш проект отныне зависит от этого пакета. Теперь, 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 импортёрам данного модуля. Если же мы определим в этом модуле другую функцию `anotherFunc`:

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

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

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

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

```
module Main
```

```
import Lib
```

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

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

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

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

Импортируем по-разному

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

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

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

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

При попытке скомпилировать такой код мы упрёмся в ошибку:

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

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

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

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

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

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

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

```
import GHC.IO.Encoding.UTF8

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

Слишком длинно, нужно укоротить. Импортируем 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.

Глава 17

Лень

В языке Haskell по умолчанию используются ленивые вычисления. Ленивым называется такое вычисление, которое откладывается до тех пор, пока кому-нибудь не понадобится результат этого вычисления. Соответственно, если он не понадобился никому, то и вычисляться ничего не будет.

Начнём с C++

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

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

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

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

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

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

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

Функция `take_and_print` получает список, возвращённый функцией `generate_addresses`, а потом печатает первые два адреса из этого списка. Вывод будет таким:

```
127.0.0.1
```

```
127.0.0.1
```

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

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

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

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

Вот как в Haskell

Откроем файл `Main.hs` и перепишем его:

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

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

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

Функция `replicate` - лентяйка. Несмотря на то, что мы попросили её создать список из 100 строк, она смотрит по сторонам и думает: "Так-с, кому тут нужны мои строки? Ага, функции `take` нужны. И сколько же ей нужно? А-а, всего две. Ну так а чего я, глупая что ли, создавать сто строк, когда требуется всего две?! Вот тебе две строки и будь счастлива!"

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

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

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

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

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

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

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

    return addresses;
}
```

И всё бы хорошо, но это намертво зависнет. И причиной тому служит уже известное нам трудолюбие функции `generate_addresses`. Сказали ей создать бесконечно большой список - будет создавать до последнего вздоха, ведь цикл `for` в данном случае не имеет выхода.

Однако если мы соберём наш `Haskell`-проект и запустим его, то не будет никакого зависания, и на экран вновь выведутся уже знакомые нам два адреса. А всё потому, что функция `repeat` столь же ленива и рациональна, как и её коллега `replicate`. Да, мы попросили её создать бесконечно большой список, однако на деле она создаст список вовсе не бесконечный, а такой, который реально нужен. И если в данном случае нужен список только из двух строк - получите список из двух строк. Конечно, если бы потребовался список из миллиона строк - извольте, будет вам миллион.

Аргументы

Ленивые вычисления распространяются и на работу с аргументами функции. И это очень полезное свойство `Haskell`.

Вернёмся к C++. Пусть у нас есть функция, принимающая два аргумента. И в

качестве этих двух аргументов используются уже известные нам списки IP-адресов:

```
void take_and_print_two_lists( const IPAddresses& first_list,
                              const IPAddresses& second_list ) {
    for( auto ip_address : first_list ) {
        std::cout << ip_address << std::endl;
    }

    for( auto ip_address : second_list ) {
        std::cout << ip_address << std::endl;
    }
}

int main() {
    take_and_print_two_lists( generate_addresses( 1000, "127.0.0.1" ),
                              generate_addresses( 2000, "22.34.47.65" ) );
}
```

Итак, мы дважды вызываем функцию `generate_addresses`, в итоге в качестве первого аргумента функция `take_and_print_two_lists` принимает список из 1000 адресов, а в качестве второго - список из 2000 адресов. В итоге адреса из обоих списков будут выведены на экран.

А теперь представим себе, что печатать адреса из второго списка нам больше не нужно:

```
void take_and_print_two_lists( const IPAddresses& first_list,
                              const IPAddresses& second_list ) {
    for( auto ip_address : first_list ) {
        std::cout << ip_address << std::endl;
    }
}
```

Второй аргумент уже не используется. Вопрос: будет ли вызвана функция `generate_addresses`, создающая 2000 адресов? Ответ: конечно да! Ведь, как мы выяснили ранее, эта функция прямолинейна и трудолюбива. Сказали создать 2000 адресов - получите. Что же мы имеем в этом случае? Опять-таки лишнюю работу: целых 2000 адресов были созданы, а в итоге это оказалось никому не нужным. А если бы их было 10 миллионов?..

В языках, похожих на C++, все аргументы вычисляются *до* их передачи в тело функции, независимо от того, используются ли значения этих аргументов или нет. В Haskell же аргументы вычисляются только в том случае, если они реально нужны в теле функции. Поэтому в Haskell функция `generate_addresses`, создающая 2000 адресов, не будет вызвана, ведь результат её работы никому не понадобился. Да здравствуют рационализм и оптимизация!

А, кстати, вдруг я пошутил? Откуда мы знаем, что аргумент не будет вычислен в случае его неиспользуемости? А это очень легко проверить.

В Haskell, как и в любом языке программирования, нельзя делить на 0. Воспользуемся же этим опасным трюком. Откроем файл `Main.hs` и напишем в нём следующую

щее:

```
f x y = x + y
```

```
main = print (f (2 `div` 1) (3 `div` 0))
```

Функция `f` принимает два аргумента и возвращает их сумму. И вот мы передаём ей в качестве первого аргумента результат деления 2 на 1, а в качестве второго - результат деления 3 на 0. Эта программа скомпилируется, но если вы её запустите, получите ожидаемую ошибку выполнения:

```
$ ./dist/build/Real/Real
Real: divide by zero
```

Функция `f` использует оба свои аргумента, а значит, оба выражения, результат которых будет передан в качестве этих аргументов, будут вычислены, и мы поймем ошибку деления на 0. А теперь давайте чуток перепишем функцию `f`:

```
f x y = x
```

Теперь эта функция просто возвращает значение своего первого аргумента, абсолютно игнорируя второй. Эта программа тоже скомпилируется, но при запуске мы увидим:

```
$ ./dist/build/Real/Real
2
```

Всё верно, перед нами первый аргумент функции `f`. Но вы спросите, куда же делась ошибка деления на 0? А никакого деления на 0 не было. Ведь второй аргумент функции `f` не используется, а следовательно, результат выражения:

```
(3 `div` 0)
```

никому не понадобился. Ну а раз он никому не понадобился, то и вычислено это выражение не будет.

Строгость

Как было упомянуто выше, ленивость вычислений - это *умолчальное* поведение в Haskell. Однако иногда нам нужна строгость. Поэтому существуют способы избежать ленивого рационализма и действовать напролом. Но об этом будет рассказано в одной из следующих глав.

Глава 18

Рекурсия

Глава 19

Генераторы списков

Со списками мы уже знакомы, а в этой главе мы узнаем о более продвинутых техниках работы с ними. Они пригодятся нам в дальнейшем.

Полнота и красота

Понятие «list comprehension» в русскоязычной литературе переводят по-разному, один из переводов — «полнота, всеохватность». Мне нравится такой перевод, он неплохо отражает суть конструкции, которую мы сейчас рассмотрим. Сразу пример:

```
import Data.Char

main :: IO ()
main = print [toUpper c | c <- "haskell.org"]
```

Не удивляйте

Понятие “list comprehension” в русскоязычной документации чаще всего переводится как “генератор списка”. Строго говоря, это не лучший перевод, но я не смог подобрать ничего лучшего.

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

Хитрый список

Вот как это выглядит:

```
import Data.Char

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

На выходе получим:

```
"HTTP"
```

Рассмотрим поближе:

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

Мы видим квадратные скобки... То есть перед нами список? Ну почти. Перед нами - генератор списка. Скелет такой конструкции можно представить так:

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

где `LIST` - список, `ELEM` - элемент этого списка, а `OPERATION` - функция, применяемая к каждому элементу. Мы говорим: “Возьми список `LIST`, последовательно пройди по всем его элементам и примени к каждому из них функцию `OPERATION`”. В результате значения, возвращаемые функцией `OPERATION`, породят новый список.

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

Добавляем предикат

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

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

В этом случае мы говорим: “Возьми список `LIST`, последовательно пройди по всем его элементам и примени функцию `OPERATION` только к тем элементам, которые удовлетворяют предикату `PREDICATE`”.

Например:

```
import Data.Char
```

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

На выходе будет:

```
"TT"
```

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

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

```
import Data.Char
```

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

Вывод в этом случае будет таким же:

```
"TT"
```


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

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

Результат будет таким же.

Обратите внимание на комбинацию символов `/=`. Это функция проверки на неравенство, аналог оператора `!=` в языке С. Кстати, он тоже носит математический окрас. Сравните:

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

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

Больше списков

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

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

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

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

На выходе получим:

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

Мы последовательно прошлись по всем элементам списков `names` и `namePrefix`. Обратите внимание, в списке `namePrefix` лишь один префикс. Вот что будет, если префиксов два:

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

В этом случае на выходе будет:

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

В этом случае мы последовательно использовали *каждый* элемент из списка `names` и *каждый* элемент из списка `namePrefix`.

Добавляем условие

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

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

Результат:

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

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

Добавляем локальное выражение

Мы можем добавить сюда и локальное выражение с помощью уже известного нам `let`. Например так:

```
import Data.Char

main = print [toUpper c | c <- "http",
                      let hletter = 'h' in c /= hletter]
```

Промежуточное значение может быть использовано во избежание дуближа при наличии нескольких предикатов.

Пример

Разберём более практичный пример:

```
import Data.List

checkGooglerBy :: String -> String
checkGooglerBy email =
    if "gmail.com" `isSuffixOf` email
    then nameFrom email ++ " is a Googler!"
    else email
    where nameFrom fullEmail = takeWhile (/= '@') fullEmail

main = print [checkGooglerBy email | email <- ["adam@gmail.com",
        "bob@yahoo.com",
        "richard@gmail.com"]]
```

```
"elena@yandex.ru",  
"denis@gmail.com"]]
```

Результат:

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

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

Рассмотрим эту строку:

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

Скелет стандартной функции `takeWhile` можно отобразить так:

```
takeWhile PREDICATE LIST
```

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

В сухом остатке

1. Генератор списка - это конструкция, порождающая новый список из одного или нескольких имеющихся списков.
2. Новый список порождается в результате применения различных функций к элементам имеющегося списка/списков.

Глава 20

Диапазоны

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

Суть

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

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

а можем просто задать диапазон:

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

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

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

На выходе получим красивый список символов (то есть обыкновенную строку):

```
"abcdefghijklmnopqrstuvwxyz"
```

Умные диапазоны

Диапазоны можно задавать весьма гибко. Например, так мы можем получить список всех чётных чисел от 2 до 30:

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

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

Конечно, этот фокус работает не только с целыми числами, мы вполне можем написать и так:

```
main = print [1.1, 1.2..2.9]
```

В результате получим список чисел с шагом в 0.1.

Можно, кстати, и в порядке убывания:

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

На выходе получим список с десятками:

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

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

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

и так тоже не пишите:

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

Такого рода уточнения компилятору не нужны.

Без конца

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

Например, вот такой диапазон:

```
[1..]
```

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

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

Вывод:

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

Мы можем задать и шаг:

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

В этом случае вывод будет таким:

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

В сухом остатке

1. Диапазон служит для автоматического создания списка по заданным критериям.
2. Учитывая ленивую природу Haskell, мы можем оперировать диапазонами, создающими бесконечные списки.

Глава 21

Наши типы

Вот мы и добрались до Второго Кита Haskell — до **Типов**. Конечно, мы работали с типами почти с самого начала, но вам уже порядком надоели все эти `Int` и `String`, не правда ли? Пришла пора познакомиться с типами куда ближе.

Знакомство

Удивительно, но в Haskell очень мало встроенных типов, то есть таких, о которых компилятор знает с самого начала. Есть `Int`, есть `Double`, `Char`, ну и ещё несколько. Все же остальные типы, даже носящие статус стандартных, не являются встроенными в язык. Вместо этого они определены в стандартной или иных библиотеках, причём определены точно так же, как мы будем определять и наши собственные типы. А поскольку без своих типов написать сколь-нибудь серьёзное приложение у нас не получится, тема эта достойна самого пристального взгляда.

Определим тип `Transport` для двух известных протоколов транспортного уровня модели OSI:

```
data Transport = TCP | UDP
```

Перед нами — очень простой, но уже наш собственный тип. Рассмотрим его внимательнее.

Ключевое слово `data` — это начало определение типа. Далее следует название типа, в данном случае `Transport`. Имя любого типа обязано начинаться с большой буквы. Затем идёт знак равенства, после которого начинается фактическое описание типа, его «тело». В данном случае оно состоит из двух простейших конструкторов. Конструктор значения (англ. *data constructor*) — это то, что строит значение данного типа. Здесь у нас два конструктора, `TCP` и `UDP`, каждый из которых строит значение типа `Transport`. Имя конструктора тоже обязано начинаться с большой буквы.

Такое определение легко читается:

```
data Transport = TCP | UDP
```

тип `Transport` это `TCP` или `UDP`

Теперь мы можем использовать тип `Transport`, то есть создавать значения этого типа и что-то с ними делать. Например, в `let`-выражении:

```
let protocol = TCP
```

Мы создали значение `protocol` типа `Transport`, использовав конструктор `TCP`. А можно и так:

```
let protocol = UDP
```

Хотя мы использовали разные конструкторы, тип значения `protocol` в обоих случаях один и тот же — `Transport`.

Расширить подобный тип предельно просто. Добавим новый протокол `SCTP` (Stream Control Transmission Protocol):

```
data Transport = TCP | UDP | SCTP
```

Третий конструктор значения дал нам третий способ создать значение типа `Transport`.

Значение-пустышка

Задумаемся: говоря о значении типа `Transport` — о чём в действительности идёт речь? Казалось бы, значения-то фактического нет: ни числа никакого, ни строки, просто три конструктора. Так вот они и есть значения. Когда мы пишем:

```
let protocol = SCTP
```

мы создаём значение типа `Transport` с конкретным содержимым в виде `SCTP`. Конструктор — это и есть содержимое. Данный вид конструктора называется нулевым (англ. `nullary`). Тип `Transport` имеет три нулевых конструктора. И даже столь простой тип уже может быть полезен нам:

```
checkProtocol :: Transport -> String
checkProtocol transport = case transport of
  TCP   -> "That's TCP protocol."
  UDP   -> "That's UDP protocol."
  SCTP  -> "That's SCTP protocol."
```

```
main :: IO ()
main = putStrLn . checkProtocol $ TCP
```

В результате увидим:

```
That's TCP protocol.
```

Функция `checkProtocol` объявлена как принимающая аргумент типа `Transport`, а применяется она к значению, порождённому конструктором `TCP`. В данном случае конструкция `case-of` сравнивает аргумент с конструкторами. Именно поэтому нам не нужна функция `otherwise`, ведь никаким иным способом, кроме как с помощью трёх конструкторов, значение типа `Transport` создать невозможно, а значит, один из конструкторов гарантированно совпадёт.

Тип, состоящий только из нульарных конструкторов, называют ещё перечислением (англ. enumeration). Конструкторов может быть сколько угодно, в том числе один-единственный (хотя польза от подобного типа была бы невелика). Вот ещё один известный пример:

```
data Day = Sunday
        | Monday
        | Tuesday
        | Wednesday
        | Thursday
        | Friday
        | Saturday
```

Обратите внимание на форматирование, когда ментальные «ИЛИ» выровнены строго под знаком равенства. Такой стиль вы встретите во многих реальных Haskell-проектах.

Значение типа `Day` отражено одним из семи конструкторов. Сделаем же с ними что-нибудь:

```
data WorkMode = FiveDays | SixDays
```

```
workingDays :: WorkMode -> [Day]
workingDays FiveDays = [ Monday
                        , Tuesday
                        , Wednesday
                        , Thursday
                        , Friday
                        ]
workingDays SixDays = [ Monday
                      , Tuesday
                      , Wednesday
                      , Thursday
                      , Friday
                      , Saturday
                      ]
```

Функция `workingDays` возвращает список типа `[Day]`, и в случае пятидневной рабочей недели, отражённой конструктором `FiveDays`, этот список сформирован пятью конструкторами, а в случае шестидневной — шестью конструкторами.

Польза от типов, сформированных нульарными конструкторами, не очень велика, хотя встречаться с такими типами вы будете часто.

Новый тип можно определить не только с помощью ключевого слова `data`, но об этом узнаем в одной из следующих глав.

А теперь мы можем познакомиться с типами куда более полезными.

Глава 22

АТД

АТД, или Алгебраические Типы Данных (англ. ADT, Algebraic Data Type), занимают почётное место в мире типов Haskell. Абсолютно подавляющее большинство ваших собственных типов будут алгебраическими, и то же можно сказать о типах из множества Haskell-пакетов. Алгебраическим типом данных называют такой тип, которые составлены из других типов. Мы берём простые типы и строим из них, как из кирпичей, типы сложные, а из них — ещё более сложные. Это даёт нам невероятный простор для творчества.

Оставим сетевые протоколы и дни недели, рассмотрим такой пример:

```
data IPAddress = IPAddress String
```

Тип `IPAddress` использует один-единственный конструктор значения, но кое-что изменилось. Во-первых, имена типа и конструктора совпадают. Это вполне легально, вы встретите такое не раз. Во-вторых, конструктор уже не нулевой, а унарный (англ. unary), потому что теперь он связан с одним значением типа `String`. И вот как создаются значения типа `IPAddress`:

```
let ip = IPAddress "127.0.0.1"
```

Значение `ip` типа `IPAddress` образовано конструктором и конкретным значением некоего типа:

```
let ip = IPAddress "127.0.0.1"
```

конструктор	значение
значения	типа
типа <code>IPAddress</code>	<code>String</code>

| значение типа `IPAddress` |

Значение внутри нашего типа называют ещё полем (англ. field):

```
data IPAddress = IPAddress String
```

тип	конструктор	поле
-----	-------------	------

Расширим тип `IPAddress`, сделав его более современным:

```
data IPAddress = IPv4 String | IPv6 String
```

Теперь у нас два конструктора, соответствующие разным IP-версиям. Это позволит нам создавать значение типа `IPAddress` так:

```
let ip = IPv4 "127.0.0.1"
```

или так:

```
let ip = IPv6 "2001:0db8:0000:0042:0000:8a2e:0370:7334"
```

Сделаем тип ещё более удобным. Так, при работе с IP-адресом нам часто требуется `localhost`. И чтобы явно не писать `"127.0.0.1"` и `"0:0:0:0:0:0:0:1"`, введём ещё два конструктора:

```
data IPAddress = IPv4 String
                | IPv4Localhost
                | IPv6 String
                | IPv6Localhost
```

Поскольку значения `localhost` нам заведомо известны, нет нужды указывать их явно. Вместо этого, когда нам понадобится IPv4-localhost, пишем так:

```
let ip = IPv4Localhost
```

Извлекаем значение

Допустим, мы создали значение `google`:

```
let google = IPv4 "173.194.122.194"
```

Как же нам потом извлечь конкретное строковое значение из `google`? С помощью нашего старого друга, паттерн матчинга:

```
checkIP :: IPAddress -> String
checkIP (IPv4 address) = "IP is " ++ address ++ "."
```

```
main :: IO ()
main = putStrLn . checkIP $ IPv4 "173.194.122.194"
```

Результат:

```
IP is '173.194.122.194'.
```

Взглянем на определение:

```
checkIP (IPv4 address) = "IP is " ++ address ++ "."
```

Здесь мы говорим: «Мы знаем, что значение типа `IPAddress` сформировано с конструктором и строкой». Однако внимательный компилятор сделает нам замечание:

```
Pattern match(es) are non-exhaustive
In an equation for ‘checkIP’:
  Patterns not matched:
    IPv4Localhost
```

```
IPv6 _
IPv6Localhost
```

В самом деле, откуда мы знаем, что значение, к которому применили функцию `checkIP`, было сформировано именно с помощью конструктора `IPv4`? У нас же есть ещё три конструктора, и нам следует проверить их все:

```
checkIP :: IPAddress -> String
checkIP (IPv4 address) = "IPv4 is '" ++ address ++ "'. "
checkIP IPv4Localhost = "IPv4, localhost."
checkIP (IPv6 address) = "IPv6 is '" ++ address ++ "'. "
checkIP IPv6Localhost = "IPv6, localhost."
```

С каким конструктором совпало — с таким и было создано значение. Можно, конечно, и так проверить:

```
checkIP :: IPAddress -> String
checkIP addr = case addr of
    IPv4 address -> "IPv4 is '" ++ address ++ "'. "
    IPv4Localhost -> "IPv4, localhost."
    IPv6 address -> "IPv6 is '" ++ address ++ "'. "
    IPv6Localhost -> "IPv6, localhost."
```

Строим

Определим тип для сетевой точки:

```
data EndPoint = EndPoint String Int
```

Конструктор `EndPoint` — бинарный, ведь здесь уже два значения. Создаём обычным образом:

```
let googlePoint = EndPoint "173.194.122.194" 80
```

Конкретные значения извлекаем опять-таки через паттерн матчинг:

```
main :: IO ()
main = putStrLn $ "The host is: " ++ host
  where
    EndPoint host _ = EndPoint "173.194.122.194" 80

    |-- образец --| |----- значение -----|
```

Обратите внимание, что второе поле, соответствующее порту, отражено универсальным образцом `_`, потому что в данном случае нас интересует только значение хоста, а порт просто игнорируется.

И всё бы хорошо, но тип `EndPoint` мне не очень нравится. Есть в нём что-то некрасивое. Первым полем выступает строка, содержащая IP-адрес, но зачем нам строка? У нас же есть прекрасный тип `IPAddress`, он куда лучше безликой строки. Это общее правило для Haskell-разработчика: чем больше информации несёт в себе тип, тем он лучше. Давайте заменим определение:

```
data EndPoint = EndPoint IPAddress Int
```

Тип стал понятнее, и вот как мы теперь будем создавать значения:

```
let google = EndPoint (IPv4 "173.194.122.194") 80
```

Красиво. Извлекать конкретные значения будем так:

```
main :: IO ()
main = putStrLn $ "The host is: " ++ ip
  where
    EndPoint (IPv4 ip) _ = EndPoint (IPv4 "173.194.122.194") 80
    _____
    ==
    =====
```

Здесь мы опять-таки игнорируем порт, но значение IP-адреса извлекаем уже на основе образца с конструктором IPv4.

Это простой пример того, как из простых типов строятся более сложные. Но сложный тип вовсе не означает сложную работу с ним, паттерн матчинг элегантен как всегда. А вскоре мы узнаем о другом способе работы с полями типов, без паттерн матчинга.

Любопытно, что конструкторы типов тоже можно компоновать, взгляните:

```
main :: IO ()
main = putStrLn $ "The host is: " ++ ip
  where
    EndPoint (IPv4 ip) _ = (EndPoint . IPv4 $ "173.194.122.194") 80
```

Это похоже на маленькое волшебство, но конструкторы типов можно компоновать знакомым нам оператором композиции функций:

```
(EndPoint . IPv4 $ "173.194.122.194") 80
```

	значение типа	
	_____ IPAddress _____	

Вам это ничего не напоминает? Это же в точности так, как мы работали с функциями! Из этого мы делаем вывод: конструктор значения можно рассматривать как особую функцию. В самом деле:

```
EndPoint (IPv4 "173.194.122.194") 80
```

"функция"		первый		второй
		_____ аргумент _____		аргумент

Мы как бы применяем конструктор к конкретным значениям как к аргументам, в результате чего получаем значение нашего типа. А раз так, мы можем компоновать конструкторы так же, как и обычные функции, лишь бы их типы были комбинировемыми. В данном случае всё в порядке: тип значения, возвращаемого конструктором IPv4, совпадает с типом первого аргумента конструктора EndPoint.

Вот мы и познакомились с настоящими типами. Пришло время узнать о более удобной работе с полями типов.

Глава 23

АТД: поля с метками

Многие типы в реальных проектах довольно велики. Взгляните:

```
data Arguments = Arguments Port
                  Endpoint
                  RedirectData
                  FilePath
                  FilePath
                  Bool
                  FilePath
```

Значение типа `Arguments` хранит в своих полях некоторые значения, извлечённые из параметров командной строки, с которыми запущена одна из моих программ. И всё бы хорошо, но работать с таким типом абсолютно неудобно. Он содержит семь полей, и паттерн матчинг был бы слишком громоздким, представьте себе:

```
...
where
    Arguments _ _ _ redirectLib _ _ xpi = arguments
```

Более того, когда мы смотрим на определение типа, назначение его полей остаётся тайной за семью печатями. Видите предпоследнее поле? Оно имеет тип `Bool` и, понятное дело, отражает какой-то флаг. Но что это за флаг, читатель не представляет. К счастью, существует способ, спасающих нас от обеих этих проблем.

Метки

Мы можем снабдить наши поля метками (англ. `label`). Вот как это выглядит:

```
data Arguments = Arguments { runWDServer    :: Port
                           , withWDServer  :: Endpoint
                           , redirect      :: RedirectData
                           , redirectLib   :: FilePath
                           , screenshotsDir :: FilePath
                           , noScreenshots :: Bool }
```

```
    , harWithXPI    :: FilePath
  }
```

Теперь назначение меток куда понятнее. Схема определения такова:

```
data Arguments = Arguments { runWDServer :: Port }
```

тип	такой-то	конструктор	метка поля	тип
				поля

Теперь поле имеет не только тип, но и название, что и делает наше определение значительно более читабельным. Поля в этом случае разделены запятыми и заключены в фигурные скобки.

Если подряд идут два или более поля одного типа, его можно указать лишь для последней из меток. Так, если у нас есть вот такой тип:

```
data Patient = Patient { firstName :: String
                        , lastName  :: String
                        , email     :: String
                        }
```

его определение можно чуток упростить и написать так:

```
data Patient = Patient { firstName
                        , lastName
                        , email    :: String
                        }
```

Раз тип всех трёх полей одинаков, мы указываем его лишь для последней из меток. Ещё пример полной формы:

```
data Patient = Patient { firstName  :: String
                        , lastName   :: String
                        , email      :: String
                        , age        :: Int
                        , diseaseId  :: Int
                        , isIndoor   :: Bool
                        , hasInsurance :: Bool
                        }
```

и тут же упрощаем:

```
data Patient = Patient { firstName
                        , lastName
                        , email    :: String
                        , age
                        , diseaseId :: Int
                        , isIndoor
                        , hasInsurance :: Bool
                        }
```

Поля `firstName`, `lastName` и `email` имеют тип `String`, поля `age` и `diseaseId` — тип `Int`, и оставшиеся два поля — тип `Bool`.

Getter и Setter?

Что же представляют собой метки? Фактически, это особые функции, сгенерированные автоматически. Эти функции имеют три предназначения: создавать, извлекать и изменять. Да, я не оговорился, изменять. Но об этом чуть позже, путь будет маленькая интрига.

Вот как мы создаём значение типа `Patient`

```
main :: IO ()
main = print $ diseaseId patient
  where
    patient = Patient {
      firstName    = "John"
    , lastName     = "Doe"
    , email        = "john.doe@gmail.com"
    , age          = 24
    , diseaseId    = 431
    , isIndoor     = True
    , hasInsurance = True
    }
```

Метки полей используются как своего рода `setter` (от англ. `set`, «устанавливать»):

```
patient = Patient { firstName = "John"
в этом   типа   поле с
значении Patient этой меткой равно этой строке
```

Кроме того, метку можно использовать и как `getter` (от англ. `get`, «получать»):

```
main = print $ diseaseId patient
```

метка как аргумент
функция

Мы применяем метку к значению типа `Patient` и получаем значение соответствующего данной метке поля. Поэтому для получения значений полей нам уже не нужен паттерн матчинг.

Но что же за интригу я приготовил под конец? Выше я упомянул, что метки используются не только для задания значений полей и для их извлечения, но и для изменения. Вот что я имел в виду:

```
main :: IO ()
main = print $ email patientWithChangedEmail
  where
    patientWithChangedEmail = patient {
      email = "j.d@gmail.com" -- Изменяем???
    }

    patient = Patient {
      firstName = "John"
```



```

    , lastName    = "Doe"
    , email       = "john.doe@gmail.com"
    , age         = 24
    , diseaseId   = 431
    , isIndoor    = True
    , hasInsurance = True
  }

```

При запуске программы получим:

```
j.d@gmail.com
```

Но постойте, что же тут произошло? Ведь в Haskell, как мы знаем, нет оператора присваивания, однако значение поля с меткой `email` поменялось. Помню, когда я впервые увидел подобный пример, то очень удивился, мол, уж не ввели ли меня в заблуждение по поводу неизменности значений в Haskell?!

Нет, не ввели. Подобная запись:

```

patientWithChangedEmail = patient {
  email = "j.d@gmail.com"
}

```

действительно похожа на изменение поля через присваивание ему нового значения, но в действительности никакого изменения не произошло. Когда я назвал метку `setter`-ом, я немного слукавил, ведь классический `setter` из мира ООП был бы невозможен в Haskell. Посмотрим ещё раз внимательнее:

```

...
where
  patientWithChangedEmail = patient {
    email = "j.d@gmail.com"  -- Изменяем???
  }

  patient = Patient {
    firstName    = "John"
    , lastName   = "Doe"
    , email      = "john.doe@gmail.com"
    , age        = 24
    , diseaseId   = 431
    , isIndoor    = True
    , hasInsurance = True
  }

```

Взгляните, ведь у нас теперь два значения типа `Patient`, `patient` и `patientWithChangedEmail`. Эти значения не имеют друг ко другу ни малейшего отношения. Вспомните, как я говорил, что в Haskell нельзя изменить имеющееся значение, а можно лишь создать на основе имеющегося новое значение. Это именно то, что здесь произошло: мы взяли имеющееся значение `patient` и на его основе создали уже новое значение `patientWithChangedEmail`, значение поля `email` в котором теперь другое. Понятно, что поле `email` в значении `patient` осталось неизменным.

Будьте внимательны при инициализации значения с полями: вы обязаны предоставить значения для всех полей. Если вы напишете так:

```

main :: IO ()
main = print $ email patientWithChangedEmail
  where
    patientWithChangedEmail = patient {
      email = "j.d@gmail.com" -- Изменяем???
    }

    patient = Patient {
      firstName  = "John"
    , lastName   = "Doe"
    , email      = "john.doe@gmail.com"
    , age        = 24
    , diseaseId  = 431
    , isIndoor   = True
    }

    -- Поле hasInsurance забыли!

```

код скомпилируется, но внимательный компилятор предупредит вас о проблеме:

```
Fields of 'Patient' not initialised: hasInsurance
```

Пожалуйста, не пренебрегайте подобным предупреждением, ведь если вы проигнорируете его и затем попытаетесь обратиться к неинициализированному полю:

```

main = print $ hasInsurance patient
...

```

ваша программа аварийно завершится на этапе выполнения с ожидаемой ошибкой:

```
Missing field in record construction hasInsurance
```

Не забывайте: компилятор — ваш добрый друг.

Без меток

Помните, что метки полей — это синтаксический сахар (англ. syntactic sugar), и мы можем обойтись без него. Даже если тип был определён с метками, как наш `Patient`, мы можем работать с ним по-старинке:

```

data Patient = Patient { firstName  :: String
                        , lastName   :: String
                        , email      :: String
                        , age        :: Int
                        , diseaseId  :: Int
                        , isIndoor   :: Bool
                        , hasInsurance :: Bool
                        }

```

```
main :: IO ()
main = print $ hasInsurance patient
  where
    -- Создаём по-старинке...
    patient = Patient "John"
                  "Doe"
                  "john.doe@gmail.com"
                  24
                  431
                  True
                  True
```

Соответственно, извлекать значения полей тоже можно по-старинке, через паттерн матчинг:

```
main :: IO ()
main = print insurance
  where
    -- Жутко неудобно, но если желаеме...
    Patient _ _ _ _ _ insurance = patient
    patient = Patient "John"
                  "Doe"
                  "john.doe@gmail.com"
                  24
                  431
                  True
                  True
```

С понятием «синтаксический сахар» мы встретимся ещё не раз, на куда более продвинутых примерах.

Глава 24

Конструктор типа

В предыдущих двух главах мы познакомились с АТД, которые, согласитесь, уже очень полезны. И всё-таки есть в них одно ограничение: они напрочь лишены гибкости. Все поля чётко заданы: вот тебе имя, а вот тебе тип, и будь счастлив. Однако есть способ наделить наши типы большей красотой.

Допустим есть у нас тип для даты:

```
data Date = Date { year  :: Int
                  , month :: Int
                  , day   :: Int
                  }

main :: IO ()
main = print $ year date
  where
    date = Date { year  = 2016
                  , month = 04
                  , day   = 12
                  }
```

Предельно простой тип: дата задаётся значениями года, месяца и дня. Но с чего мы взяли, что задавать, скажем, значения года или месяца следует целыми числами? А может лучше строкой? Или типом из готовой библиотеки для работы с датами? Или вообще нашим собственным типом? Добавим гибкости в наш тип:

Глава 25

Новый тип

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

Один конструктор значения

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

```
newtype IPAddress = IP String
                  deriving Show
```

а вот такой код будет категорически отвергнут компилятором:

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

Одно поле

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

```
newtype IPAddress = IP String
                  deriving Show
```

или так:

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

А вот такой код не будет принят компилятором:

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

Тип с нульарным конструктором тоже не пройдёт компиляцию:

```
newtype Color = Red
```

Для чего он нужен

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

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

Если мы определили вот такой тип:

```
data IPAddress = IP String
```

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

Если же мы написали так:

```
newtype IPAddress = IP String
```

мы сказали компилятору: “`IPAddress` - это всего лишь именная обёртка вокруг строки. Именно так её и воспринимай, и никаких лишних телодвижений не делай”. Поэтому работа с таким типом будет чуток более эффективной.

В сухом остатке

1. Ключевое слово `newtype` создаёт тип-обёртку вокруг некоторого уже существующего типа.
2. Эта тип-обёртка должна иметь только одно поле и только один конструктор значения.
3. Работа с типом, созданным с помощью `newtype`, эффективнее, нежели с созданным с помощью `data`.

Глава 26

Stackage

В предыдущей главе мы познакомились с Hackage, репозиторием Haskell-пакетов. Теперь же настало время познакомиться со “стабильным Hackage”.

Проблема

В становившиеся с годами времена существовала в мире Haskell большая проблема, связанная с пакетами, и называлась она “dependency hell”. Вот в чём её суть.

Как вы уже поняли, каждый из Haskell-пакетов тоже имеет ряд зависимостей от неких других пакетов, и эти зависимости перечислены в его .cabal-файле. Зависимость может быть указана “версионной вилкой”, например:

```
http-client >= 0.3 && < 0.5
```

Это означает, что данный пакет зависит от пакета http-client любой версии, входящей в промежуток от 0.3 включительно до 0.5. Однако зависимость может быть указана точно:

```
http-client == 0.3.6.1
```

В этом случае нам нужна версия 0.3.6.1 и никакая другая. Вот тут-то нас и поджидает ад.

Допустим, наш проект зависит от двух пакетов, А и В, каждый из которых в свою очередь зависит от третьего пакета С. Но к сожалению, звёзды оказались неблагоприятны к нашему проекту, и выяснилось, что пакет А зависит от пакета С версии 1.8.1, в то время как пакет В - от пакета С версии 2.0.0... Эту проблему пытались решить несколькими способами, но самым удобным оказался способ под названием Stackage.

Что это такое?

Название Stackage происходит от слияния слов “Stable” и “Hackage”. Идея в том, чтобы предоставить разработчику надёжный набор пакетов, на который гаранти-

рованно можно положиться.

В основе этой идеи лежит LTS Haskell snapshot, или LTS-снимок (от Long-Term Support). LTS-снимок представляет собой большой список пакетов жёстко заданных версий. И подобраны эти версии таким образом, чтобы все пакеты были версионно совместимы друг с другом. Гарантированно и железобетонно. То есть, возвращаясь к нашему примеру, если в LTS-снимке перечислены пакеты *a* и *b*, то их версии подобраны так, чтобы они зависели от одной и той же версии пакета *c*. В этом случае у нас никогда не возникнет никаких версионных коллизий.

Откроем файл `stack.yaml` в корне нашего проекта и найдём там строчку вида:

```
resolver: lts-5.2
```

В этой строке мы задаём номер LTS-снимка, используемого в данном проекте. Да, у каждого снимка есть собственный номер, или, если хотите, версия. В данном случае мы объявляем, что наш проект “живёт” в рамках LTS-снимка под номером 5.2. Это означает, что если наш проект зависит от пакета *text*, то не от абы какой версии *text*, но именно от версии 1.2.2.0, ведь именно эта версия упомянута в данном LTS-снимке.

Кстати, содержимое снимка 5.2 можно посмотреть [здесь](#). В конце адреса меняем номер - и видим содержимое другого снимка.

Неизменность

LTS-снимки хороши и тем, что неизменны. В упомянутом снимке 5.2 перечислены 1766 пакетов конкретных версий, и так оно будет всегда. Следовательно, если члены команды Haskell-разработчиков, работающие на одном и тем же проектом, используют один и тот же снимок - у всех у них всё будет компилироваться. Сегодня, завтра и через пять лет. Создатели Stackage провозгласили это одним из девизов данной идеи: “То, что работает сегодня, должно работать и завтра”.

Утилита stack?

Да, совершенно верно: уже знакомая нам утилита `stack` является частью проекта Stackage, оттого и схожесть в названии. Поэтому, используя `stack`, вы уже используете какой-то LTS-снимок.

А если не из репозитория?

В самом деле, представим себе такую ситуацию. Некий разработчик сделал полезную Haskell-библиотеку, но не стал включать её в единый репозиторий, а вместо этого поселил её в своём GitHub-профиле. А нам эта библиотека очень уж приглянулась. Как же включить её в проект? Открываем `stack.yaml` и пишем:

- `location:`

`git:` <https://github.com/user/project.git>

`commit:` 018919f43854b1d6f30d4270f5471db807ac1a41

Думаю, тут всё понятно без комментариев: пакет будет вытянут из данного репозитория (с использованием указанного коммита), собран и включён в наш проект. И не забудьте включить имя данного пакета в списке зависимостей в `.cabal`-файле.

Ещё почитать

Я написал небольшую [обзорную статью](#), посвящённую `stack` и работе с LTS-снимками.

Глава 27

О форматировании

Haskell-код является форматно-зависимым, поэтому мы не можем расставлять отступы там, где нам заблагорассудится. Необходимо придерживаться определённых правил.

Функция

Если мы напишем так:

```
main :: IO ()
main =
  putStrLn "Hi Haskell!"
```

компилятор выскажет своё несогласие:

```
parse error (possibly incorrect indentation or mismatched brackets)
```

Следующий пример:

```
main :: IO ()
main =
  putStrLn "Hi Haskell!"
```

Здесь мы поставили один пробел перед каждой из трёх строк, однако и в этом случае компилятор закапризничает:

```
parse error on input `main'
```

Или вот так:

```
main :: IO ()
main =
  putStrLn "Hi Haskell!"
```

В этом случае мы получим ещё более странную ошибку:

```
Illegal type signature: `IO () main'
```

Из-за пробела перед именем функции компилятор принял это имя за часть сигнатуры.

Когда в теле функции несколько строк, появляются дополнительные ограничения. Если напишем так:

```
main :: IO ()
main = do
    putStrLn "Hi Haskeller!"
    putStrLn "Hi again!"
```

получим вот это:

```
Couldn't match expected type `(String -> IO ()) -> [Char] -> IO ()'
      with actual type `IO ()'
The function `putStrLn' is applied to three arguments,
but its type `String -> IO ()' has only one
```

Из-за сдвига второй функции по отношению к первой компилятор подумал, что первая по счёту `putStrLn` применяется к трём аргументам. Если же напишем так:

```
main :: IO ()
main = do
    putStrLn "Hi Haskeller!"
    putStrLn "Hi again!"
```

получим уже знакомую нам ошибку:

```
parse error on input `putStrLn'
```

Здесь компилятор ругнулся уже на вторую по счёту `putStrLn`.

В общем, экспериментальным путём я выяснил, что форматирование кода функции должно соответствовать следующим правилам:

1. Объявление и определение функции, должны начинаться с первого (самого левого) символа строки.
2. Если тело функции начинается со следующей строки после имени, перед этим телом должен присутствовать отступ от первого символа строки, хотя бы в один пробел.
3. Если тело функции состоит из нескольких выражений, стоящих на отдельной строке каждая, эти выражения должны быть вертикально выровнены по левому краю.

Поэтому придерживайтесь приблизительно такого шаблона:

```
main :: IO ()
main = do
    putStrLn "Hi Haskeller!"
    putStrLn "Hi again!"
```

и компилятор будет просто счастлив.

Тип

На код, связанный с типами, также наложены некоторые форматные ограничения.

```
data IPAddress = IP String
```

Перед словом `data` стоит лишний пробел, и компилятор вновь вспоминает нас недобрым словом:

```
parse error on input `data'
```

Вот такой код тоже не пройдет компиляцию:

```
data
IPAddress = IP String
```

равно как и такой:

```
data IPAddress =
IP String
```

и даже такой:

```
data IPAddress
= IP String
```

В ходе экспериментов было выяснено, что правила для кода определения типа схожи с вышеупомянутыми правилами для кода функции:

1. Ключевое слово `data` начинается с самого левого символа строки.
2. Если объявление переходит на следующую строку, то перед ним должен быть хотя бы один пробел.

Поэтому пишите приблизительно так:

```
data IPAddress = IP String
                deriving Show
```

и компилятор будет вам благодарен.

Класс типов

С классами типов - та же история. Если напишем так:

```
class Note n where
write :: n -> Bool
```

получим экзотическую ошибку:

```
The type signature for `write' lacks an accompanying binding
```

Если вздумаем написать так:

```
class Note n where
  write :: n -> Bool
  read  :: n -> String
```

снова получим по башке:

```
parse error on input `read'
```

И если так напишем:

```
class Note n where
  write :: n -> Bool
  read
    :: n -> String
```

и даже если так:

```
class Note n where
  write :: n -> Bool
  read :: n -> String
```

компилятор будет принципиален до крайности и не пропустит такой код.

В общем, тут правила точно такие же:

1. Начинаем с самого левого символа строки.
2. Перед методами - хотя бы однопробельный отступ.
3. Методы должны быть вертикально выровнены по левому краю.

Следовательно, ублажаем компилятор и пишем примерно так:

```
class Note n where
  write :: n -> Bool
  read  :: n -> String
```

Константа

Для отдельной константы правила точно такие же, как и для функции. Поэтому пишем:

```
coefficient :: Double
coefficient = 0.0036
```

и всё будет хорошо.

Условие

Тут я выявил лишь одно ограничение - край ключевого слова `if` должен быть самым левым по отношению ко всем остальным частям выражения. То есть можно написать так:

```
main :: IO ()
main = do
  if 2 /= 2
  then
    putStrLn "Impossible"
```

```
else
    putStrLn "I believe"
```

и так:

```
main :: IO ()
main = do
    if 2 /= 2
    then
        putStrLn "Impossible"
    else
        putStrLn "I believe"
```

и даже так:

```
main :: IO ()
main = do
    if 2 /= 2

    then
        putStrLn "Impossible"
    else
        putStrLn "I believe"
```

Но вот такого компилятор не потерпит:

```
main :: IO ()
main = do
    if 2 /= 2
    then
        putStrLn "Impossible"
    else
        putStrLn "I believe"
```

равно как и такого:

```
main :: IO ()
main = do
    if 2 /= 2
    then
        putStrLn "Impossible"
    else
        putStrLn "I believe"
```

Локальные выражения

Эти друзья менее прихотливы. В отношении выражения `where` я нашёл только одно ограничение:

```
prepare :: String -> String
prepare str =
    str ++ helper
```

```
where
  helper = "dear. "
```

В этом случае получим ошибку:

```
parse error on input `where'
```

Такого же рода ограничение действует и на `let`:

```
prepare :: String -> String
prepare str =
  let helper = "dear. "
    in
    str ++ helper
```

Однако ошибка будет другой:

```
parse error (possibly incorrect indentation or mismatched brackets)
```

Суть вы уловили: пусть `where` и `let` гармонируют с остальным кодом тела функции.

Вывод

Пишите аккуратно, придерживаясь единого стиля без лишних изысков.