

О Haskell по-человечески

Денис Шевченко

Содержание

1 Добро пожаловать!	5
Почему эта книга появилась	5
Цель	5
О себе	6
О вас	6
Обещание	6
2 Первые вопросы	7
«Что такое этот ваш Haskell?»	7
«Это что, какой-то новый язык?»	7
«И кто его сделал?»	8
«А библиотеки для Haskell имеются?»	8
«Да, но я слышал, что Haskell ещё не готов к production...»	8
«А правда ли, что порог вхождения в Haskell высок?»	8
«И что же в нём такого необычного?»	9
«А если сравнить его с C++/Python/Scala...»	9
3 Об этой книге	11
Чего здесь нет	11
О первом и втором издании	12
Читайте последовательно	12
Благодарность	12
4 Приготовимся	13
Устанавливаем	13
Разворачиваем инфраструктуру	14
Hi World	14
Модули: первый взгляд	15

Имена модулей	16
5 Киты и Черепаха	19
Черепаха	19
Первый Кит	20
Второй Кит	22
Третий Кит	24
6 Неизменность и чистота	27
Объявляем и определяем	27
Чисто функциональный	29
«Присваивание? Не, не слышал...»	30
Удивлены?	31
7 Выбираем и возвращаемся	33
Выглянем во внешний мир	33
Выбор и выход	34
8 Выбор и образцы	39
Не только из двух	39
Без Если	42
Сравнение с образцом	43
case	44
9 Пусть будет там, Где...	47
Пусть	47
Где	49
Вместе	50

Глава 1

Добро пожаловать!

Перед вами — книга о Haskell, удивительном и красивом языке программирования. Я написал её для тех, кто плохо представляет себе, что такое функциональное программирование, но хочет наконец разобраться.

Почему эта книга появилась

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

Цель

Функциональное программирование — своеобразное гетто посреди мегаполиса нашей индустрии. Доля функциональных языков пока ещё мала, однако эти языки — и в частности Haskell — являются мощными инструментами разработки, и в рамках этой книги я покажу вам эту мощь. Вероятно, вы слышали, что Haskell — это что-то сугубо теоретическое/научное/супер-сложное/непригодное для жизни? Читайте дальше, и вскоре вы убедитесь, что эти предрассудки остались в прошлом.

О себе

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

О вас

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

Обещание

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

Глава 2

Первые вопросы

С них и начнём.

«Что такое этот ваш Haskell?»

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

«Это что, какой-то новый язык?»

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

«И кто его сделал?»

Главная реализация языка нашла своё воплощение в компиляторе GHC (The Glasgow Haskell Compiler), родившемся в недрах Microsoft Research. Впрочем, отнеситесь к слову «Microsoft» спокойно — к .NET и прочим известным творениям Компании-из-Редмонда компилятор GHC отношения не имеет.

«А библиотеки для Haskell имеются?»

Имеются, и весьма много. В процессе чтения вы познакомитесь со многими из них.

«Да, но я слышал, что Haskell ещё не готов к production...»

Готов, и уже не первый год. 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 убедительно свидетельствуют истории успешного внедрения Haskell в бизнесе, в том числе крупном.

«А правда ли, что порог вхождения в Haskell высок?»

Правда. Haskell настолько не похож на другие языки, что людям, пришедшим из мира других языков, мозги поломать придётся. Именно поломать, а не просто пошевелить ими: Haskell заставляет иначе взглянуть на, казалось бы, привычные вещи. В этом его сложность и в этом же его красота: многие люди, включая меня, узнав вкус Haskell, категорически не желают возвращаться к другим языкам. Я вас предупредил.

«И что же в нём такого необычного?»

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

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

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

Глава 3

Об этой книге

В последние годы заметно возросло число книг, посвящённых Haskell, и это радует. Каждая из них преследует свою цель, поэтому трудно сказать, какая из них лучше. Цель этой книги — научить вас главному в Haskell, основам, без глубокого усвоения которых двигаться вперёд не получится.

Как было сказано в предыдущей главе, порог вхождения в Haskell весьма высок, и в первую очередь в силу непохожести этого языка на остальных. Объективно, программировать на Haskell совсем не сложно, но лишь после того, как вы близко познакомились с Тремя Китами Haskell, а также с Госпожой Черепахой, поддерживающей оных. Вот этому знакомству и посвящена эта книга. А имена этих Китов и Черепахи вы узнаете уже в следующей главе.

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

Чего здесь нет

Трёх вещей вы не найдёте на страницах этой книги:

1. Справочника по Haskell. Лучшим справочником является официальное описание стандарта Haskell 2010.
2. Набора готовых рецептов. За рецептами пожалуйста на Stackoverflow.
3. Введения в математическую теорию. Несмотря на то, что Haskell корнями своими уходит в математику, в этой книге нет погружения в теорию категорий и в иные теории. Извините, если разочаровал.

О первом и втором издании

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

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

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

Читайте последовательно

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

Благодарность

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

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

Глава 4

Приготовимся

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

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

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

Устанавливаем

Идём сюда и скачиваем архив для нужной нам ОС. Распаковываем архив — и перед нами утилита под названием `stack`. Для удобства располагаем её в каком-нибудь каталоге, доступном в PATH. Рекомендованный путь — `~/local/bin/`.

Если же вы живёте в мире Mac и пользуетесь Homebrew — вам ещё проще. Делаете:

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

Всё.

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

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

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

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

Haskell-проект — среда для создания приложений и библиотек.

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

Разворачиваем инфраструктуру

Делаем:

```
$ stack setup
```

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

Hi World

Создадим наш первый Haskell-проект:

```
$ stack new real
```

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

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

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

```
$ stack install
```

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

```
$ real-exe
someFunc
```

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

Модули: первый взгляд

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

Откроем модуль `src/Lib.hs`:

```
module Lib
  ( someFunc
  ) where

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

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

Как вы уже поняли, расширение `.hs` — стандартное расширения для модулей.

Теперь откроем модуль `app/Main.hs`:

```
module Main where

import Lib      -- Импортируем модуль Lib...

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

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

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

Имена модулей

Есть два правила.

Во-первых, имя модуля должно начинаться с большой буквы. Всегда.

Во-вторых, желательно, чтобы имя модуля совпадало с именем соответствующего ему файла. Так, файл, содержащий модуль `Main`, назван `Main.hs`. Это очень удобно, помогает избежать путаницы.

Всё. В будущих главах вы узнаете о модулях кое-что ещё, но пока достаточно этого. Теперь пора познакомиться с пакетами, ведь мы будем использовать их в наших проектах постоянно.

Глава 5

Киты и Черепаха

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

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

Черепаха

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

`1 + 2`

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

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

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

`1 + 2`

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

3

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

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

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

Первый Кит

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

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

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

Рассмотрим функцию возведения целого числа в квадрат:

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

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

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

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

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

```
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 главное:

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

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

математическом смысле слова, поэтому её не вызывают, а применяют к чему-то.

Второй Кит

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

```
sqaare  2  
функция аргумент
```

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

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

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

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

Три эти свойства системы типов Haskell — наши 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 на этапе компиляции. Haskell-компилятор упрям: когда ему что-либо не нравится в типах, он громко ругается. Поэтому если функция работает с целыми числами, применить её к строкам никак не получится. Таким образом, когда компиляция завершилась успешно, мы точно знаем, что с типами у нас всё в порядке.

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

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

Сила

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

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

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

В Haskell подобный код не имеет ни малейших шансов пройти компиляцию. Мы всегда получаем то, что ожидаем, и если должно быть число с плавающей точкой — расшибись, но предоставь именно его. Компилятор скрупулёзно отслеживает соответствие между ожидаемым типом и фактическим, поэтому

когда компиляция завершается успешно, мы абсолютно уверены в гармонии между типами всех наших данных.

Выведение

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

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

однако в Haskell мы напишем просто:

```
value = 122.04
```

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

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

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

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

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

Третий Кит

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

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

Глава 6

Неизменность и чистота

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

Объявляем и определяем

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

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

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

Рассмотрим объявление:

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

Оно разделено двойным двоеточием на две части: слева указано имя функции, справа — типы, с которыми эта функция работает, а именно типы аргументов

и тип вычисленного, итогового значения. Как вы узнали из предыдущей главы, все данные в Haskell-программе имеют конкретный тип, а поскольку функция работает с данными, её объявление содержит типы этих данных. Типы разделены стрелками. Схематично это выглядит так:

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

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

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

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

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

Теперь рассмотрим определение функции `square`:

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

Схема определения такова:

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

А функция `product` определена так:

```
product x      y      =  x * y
имя     имя     имя     это выражение
```

функции первого второго
 аргумента аргумента

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

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

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

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

Но откуда у меня такая уверенность? А вот откуда.

Чисто функциональный

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

Скажу больше: чистая функция не видит окружающий мир. Вообще. Причём в данном случае под «окружающим миром» я подразумеваю не только внешний по отношению ко всей программе мир (например, файловая система или сеть), но и все остальные функции. Чистую функцию можно сравнить с чёрным

ящиком: она знает только свои аргументы и вычисляемое ею значение. Это даёт нам второе преимущество чистых функций — композируемость. Раз функции полностью изолированы друг от друга, их очень просто комбинировать, строя из более простых более сложные.

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

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

«Присваивание? Не, не слышал...»

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

Вспомним, что присваивание (англ. assignment) пришло к нам из императивных языков. При императивном программировании (англ. imperative programming) программа воспринимается как набор инструкций, выполнение которых неразрывно связано с изменением состояния этой программы. Вот почему в императивных языках присутствует понятие «переменная» (англ. variable). А раз есть переменные — должен быть и инструмент для изменения их конкретных значений, а именно 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 всех остальных переменных в нашем коде.

Однако существует принципиально иной подход к разработке, а именно декларативное программирование (англ. declarative programming). Haskell воплотил в себе именно этот подход, при котором программа воспринимается уже не как набор инструкций, а как набор выражений. А поскольку выражения

вычисляются путём применения функций к аргументам (то есть, по сути, к другим выражениям), там нет места ни переменным, ни оператору присваивания. Все данные в Haskell-программе, будучи созданными единожды, уже не могут быть изменены. Поэтому когда в Haskell-коде мы пишем:

```
coeff = 0.569;
```

мы просто объявляем: «Отныне значение `coeff` равно 0.569, и так оно будет всегда». Вот почему в Haskell-коде символ `=` — это знак равенства, а вовсе не присваивание.

Удивлены?

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

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

Глава 7

Выбираем и возвращаемся

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

Выглянем во внешний мир

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

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

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

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

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

```
{-
  Я - сложный многострочный
```

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

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

```
$ stack install
```

После чего запускаем:

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

Выбор и выход

Существует несколько способов задания условной конструкции. Вот базовый вариант:

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

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

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

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

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

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

при запуске увидим это:

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

а если так:

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

тогда увидим это:

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

Круглые скобки включают выражение типа String по схеме:

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

То есть функция putStrLn видит в конечном итоге лишь результат применения функции checkLocalhost.

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

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

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

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

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

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

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

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

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

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

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

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

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

эквивалентен коду:

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

Аналогично, код:

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

есть ни что иное, как:

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

Каким бы сложным ни было логическое ветвление внутри функции `checkLocalhost`, в конечном итоге оно вернёт/вычислит какое-то одно итоговое выражение.

Вот почему функции в Haskell так просто компоновать друг с другом, и позже мы будем встречать всё больше таких примеров.

Глава 8

Выбор и образцы

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

Не только из двух

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

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

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

Уверен, вы уже стираете плевок с экрана. Вложенная if-then-else конструкция не может понравиться никому, ведь она крайне неудобна в обращении. А уж если

бы анализируемых проб золота было штук пять или семь, эта лестница стала бы поистине ужасной. К счастью, в Haskell можно написать по-другому:

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

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

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

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

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

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

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

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

Non-exhaustive guards in multi-way if

Проверка получилась неполной, вот и получите ошибку.

Видите слово `guards` в сообщении об ошибке? Вертикальные черты перед

логическими выражениями — это и есть охранники (англ. *guard*), неусыпно охраняющие наши условия. А чтобы читать их было легче, воспринимайте их как аналог слова «ИЛИ».

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

```
{-# LANGUAGE MultiWayIf #-}  -- ???

module Main where

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

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

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

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

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

Без Если

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

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

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

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

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

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

                                в случае
                                истинности
                                этого
                                выражения

                                либо и т.д.
```

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

Сравнение с образцом

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

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

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

	<code>analyzeGold</code>	<code>999</code>	<code>=</code>	<code>"Wow! 999 standard!"</code>
если	эта функция	применяется	тогда	этому выражению
		вот к этому	она	
		аргументу	равна	

	<code>analyzeGold</code>	<code>750</code>	<code>=</code>	<code>"Wow! 999 standard!"</code>
если	эта функция	применяется	тогда	другому выражению
		к другому	она	
		аргументу	равна	

...

	<code>analyzeGold</code>	<code>_</code>	<code>=</code>	<code>"I don't know such a standard..."</code>
--	--------------------------	----------------	----------------	--

в

противном	эта функция	просто	общему выражению
случае		равна	

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

образцу.

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

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

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

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

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

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

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

case

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

```
analyzeGold standard =
  case standard of
    999      -> "Wow! 999 standard!"
    750      -> "Great! 750 standard."
```

```
585      -> "Not bad! 585 standard."  
otherwise -> "I don't know such a standard..."
```

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

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

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

В последующих главах мы встретимся с другими видами паттерн матчинга.

Глава 9

Пусть будет там, Где...

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

Пусть

Рассмотрим следующую функцию:

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

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

```
calculateTime :: Int -> Int
calculateTime timeInS =
  let threshold = 40
      correction = 120
      delta     = 8
```

```
in if | timeInS < threshold -> timeInS + correction
    | timeInS >= threshold -> timeInS + delta + correction
```

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

Конструкция `let-in` вводит поясняющее выражение по схеме:

```
let DECLARATION in EXPRESSION
```

где `DECLARATION` — выражение, декларируемое нами, а `EXPRESSION` — выражение, в котором используется `DECLARATION`. Так, когда мы написали:

```
let threshold = 40
```

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

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

Эта конструкция легко читается:

```
let   threshold =      40      ... in ...
      пусть  это      будет  этому      в  том
      выражение равно выражению выражении
```

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

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

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



```
| timeInS >= threshold ->
  let delta = 8 in timeInS + delta + correction
```

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

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

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

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

Где

Существует иной способ введения промежуточных выражений:

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

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

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

В отличие от `let`, которое может быть использовано для введения супер-локального выражения (как в последнем примере с `delta`), все `where`-выражения

доступны в любой части выражения, предшествующего ключевому слову `where`.

Вместе

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

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

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

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

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

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

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

```
delta      = correction - 4
correction = timeInS * 2
```

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

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

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

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

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

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

При попытке скомпилировать такой код мы получим ошибку:

```
Not in scope: 'threshold'
```

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

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