

# О Haskell по-человечески

Денис Шевченко



# Содержание

<b>1 Добро пожаловать!</b>	<b>5</b>
Почему эта книга появилась . . . . .	5
Цель . . . . .	5
О себе . . . . .	6
О вас . . . . .	6
Обещание . . . . .	6
<b>2 Первые вопросы</b>	<b>7</b>
«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
<b>3 Об этой книге</b>	<b>11</b>
Чего здесь нет . . . . .	11
О первом и втором издании . . . . .	12
Читайте последовательно . . . . .	12
Благодарность . . . . .	12
<b>4 Приготовимся</b>	<b>13</b>
Устанавливаем . . . . .	13
Разворачиваем инфраструктуру . . . . .	14
Hi World . . . . .	14
Модули: первый взгляд . . . . .	15

Имена модулей . . . . .	16
<b>5 Киты и Черепаха</b>	<b>19</b>
Черепаха . . . . .	19
Первый Кит . . . . .	20
Второй Кит . . . . .	22
Третий Кит . . . . .	24
<b>6 Неизменность и чистота</b>	<b>27</b>
Объявляем и определяем . . . . .	27
Чисто функциональный . . . . .	29
«Присваивание? Не, не слышал...» . . . . .	30
Удивлены? . . . . .	31
<b>7 Hackage</b>	<b>33</b>
Ищем . . . . .	33
Включаем . . . . .	34
Прелюдия . . . . .	35

# 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 русскоязычного сообщества Haskell-разработчиков 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 поломают 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. Познакомимся с модулями.

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.

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



кстати, очень удобно, помогает избегать 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 C тип данных следует указывать явно:

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

однако в Haskell мы напишем просто:

```
value = 122.04  
}
```

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

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

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

```
123          Int  
23.5798      Double  
'a'         Char  
"Hello!"     String
```

С типами `Int` и `Double` вы уже знакомы. Тип `Char` — это Unicode-символ. Тип `String` — обыкновенная строка. 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.

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 С-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.

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





## Глава 7

# Hackage

Hackage — это главный 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 с тех пор увеличился с нескольких десятков пакетов до почти 9 тысяч.

По сути, пакет представляет собой совокупность модулей. Существуют большие пакеты, состоящие из многих десятков модулей, но есть и такие, в которых модуль всего один.

Чтобы воспользоваться пакетом, необходимо сделать три шага:

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

## Ищем

Искать пакет можно двумя способами: на единой Hackage-странице или через специальный поисковик.

Все пакеты, живущий в Hackage, можно увидеть на одной странице. Это весьма удобно: браузерный поиск поможет вам искать пакеты по всем доступным тематическим категориям.

А ещё есть особые поисковики по Hackage. Названы они в честь поисковых гигантов:

1. Hoogle
2. Hayoo!

Эти поисковики скрупулёзно осматривают внутренности Hackage, и мы будем очень часто ими пользоваться. Сейчас, в качестве примера, возьмём пакет `text`, предназначенный для работы с текстом. Этот пакет, кстати, очень нам пригодится, живёт он тут.

Каждый из Hackage-пакетов живёт по адресу, сформированному по неизменной схеме: `http://hackage.haskell.org/package/ИМЯПАКЕТА`.

## Включаем

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

```
build-depends:      base
                   , real
                   , text
```

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

Теперь выполняем:

```
$ stack install
```

Начнётся повторная сборка проекта, но перед нею `stack`, увидев новую зависимость, сделает то что нам и нужно: установит сам пакет `text` и все те пакеты, от которых он в свою очередь зависит.

Готово. Отныне мы можем импортировать нужные нам модули из пакета `text`. В следующих главах мы сделаем это неоднократно.

## **Прелюдия**

Существует один стандартный модуль, который по умолчанию автоматически импортируется во все наши модули. Имя ему — `Prelude`. В этом модуле содержатся самые базовые Haskell-инструменты, многие из которых мы будем использовать постоянно.