О Haskell по-человечески

Денис Шевченко

Содержание

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 фунции 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 фукнциональный 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 приведёт вас к вершинам Haskell, но она откроет вам путь к этим вершинам.

Чего здесь нет

Трёх вещей вы не найдёте на страницах этой книги:

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

О первом и втором издании

На обложке вы видели метку «издание 2.0». Перед вами второе издание, полностью переработанное и переосмысленное. На момент публикации второй версии книги первая всё ещё доступна онлайн, но дни её сочтены: она переведена в статус Deprecated и её поддержка полностью прекращена.

Есть 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 много не знал и не понимал. Это привело к тому, что часть информации из первого издания откровенно бедна, а несколько глав вообще вводят читателя в заблуждение.

Вторая — изменившаяся цель книги. Я намеренно сузил круг рассматриваемых здесь тем. Теперь книга всецело посвящена основам, 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. Да, всё это можно делать на Haskell, но подобным темам посвящены другие публикации, которые несомненно будут вам по плечу после прочтения моей книги.

Как её читать

Строго последовательно, это важно. В процессе чтения вы заметите, что я периодически поднимаю вопросы и как бы оставляю их без ответа. Я делаю это 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.

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 OX S 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 разивается, 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 познакомится с фундаментальной единицей проекта — модулем.

Глава 5

Модули: знакомство

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`:

```haskell
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 познакомится с пакетами, ведь мы будем использовать их в наших проектах постоянно.

Глава 6

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 этому модуле содержатся самые базовые Haskell-инструменты, многие из которых мы будем использовать постоянно.

Глава 7

Мир выражений и функций

Итак, проект создали, с пакетами познакомились, теперь мы готовы начать путешествие. Ремни пристёгивать необязательно, мы будем двигаться не торопясь.

Первой важной особенностью Haskell является тот факт, что программа - это мир выражений (англ. expression). Когда вы смотрите на исходный код, сколь бы простым или сложным он ни был - вы видите не совокупность инструкций, а совокупность выражений. Вот примеры:

```
1 + 2

(12.3 - 12.067) * 456 * 0.234

length myList

"/usr" </> pathToLib

9
```

Всё это - выражения. И даже то, что вы видите на последней строке - это не просто число `9`, это тоже выражение.

Любое выражение может дать нам некий полезный результат, в противном случае оно бессмысленно. Все выражения можно разделить на две группы: такие, которые ещё можно вычислить (англ. evaluate), и такие, вычислить которые уже нельзя. Например, выражение:

```
1 + 2
```

можно вычислить, то есть произвести сложение. Однако когда мы получим результирующее выражение, а именно:

```
3
```

его уже нельзя вычислить, оно вычислено окончательно, что называется, до самого дна.

Когда мы вычисляем выражение, оно всегда уменьшается (англ. reduce). И уменьшать его мы можем до тех пор, пока оно не достигнет своей простейшей, окончательной формы. Например, выражение:

```
(12.3 - 12.067) * 456 * 0.234
```

вычисляется в три шага:

```
   (12.3 - 12.067) * 456 * 0.234
-> 0.233           * 456 * 0.234
-> 106.248               * 0.234
-> 24.862
```

Всё, дошли до дна, ничего более сделать с выражением `24.862` мы уже не способны, мы можем лишь использовать его как оно есть. Таким образом, выражения, составляющие Haskell-программу, вычисляются до тех пор, пока не останется некое окончательное, корневое выражение. Следовательно, запуск Haskell-программы на выполнение (англ. execution) - это запуск всей этой цепочки вычислений, а 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.

Госпожа Функция

Именно функция делает выражение вычислимым, именно она оживляет нашу программу, потому я и назвал Функцию Первым Китом, на котором зиждется Haskell. Но дабы избежать недоразумений, следует определиться с понятиями.

Для начала вспомним математическое определение функции. Не пугайтесь, математики будет совсем немного:

> Функция - это закон, описывающий зависимость одного значения от другого.

Рассмотрим функцию возведения целого числа в квадрат:

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

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

```
square 2 = 4
square 3 = 9
square 4 = 16
```

Входное значение функции называют аргументом. И так как функция определяет однозначную зависимость выходного значения от аргумента, её, функцию, называют иногда *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` быть не может.

Чтобы заставить функцию сделать полезную работу, её нужно применить (англ. apply) к некоторому аргументу. Ведь если на вход ничего не кинули, то и на выходе ничего не получим. Вот пример:

```
square 2
```

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

```
sum 10 20
```

Вы спросите, к чему я всё это рассказываю? Взгляните на применение функции `square` ещё раз. Да ведь это же выражение! То самое выражение, о котором мы узнали в самом начале главы. Вспомним:

```
1 + 2
```

Это ни что иное, как применение функции. И чтобы яснее это увидеть, перепишем выражение так:

```
(+) 1 2
```

Это применение функции `(+)` к двум аргументам, `1` и `2`. Теперь вы знаете: вычислить выражение - значит применить какие-то функции к каким-то аргументам.

«Да, но разве функции не вызывают?»

В Haskell - нет. Понятие «вызов» функции пришло к нам из почтенного языка C. Там функции действительно вызывают (англ. call), потому что в языке 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. Так вот если вы обладаете опытом разработки на C-подобных языках - забудьте о подпрограмме. В Haskell функция - это функция в математическом смысле этого слова. Поэтому её не вызывают, а применяют к чему-то.

А теперь отложим в сторону математику и вернёмся к программированию. Перед тем, как использовать функцию, её следует объявить и определить. Вот как это выглядит на примере `square`:

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

Здесь первая строка - это объявление функции, а вторая - её определение.

Объявляем

Схема объявления следующая:

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

Можно было бы сказать «тип возвращённого значения», но поскольку в Haskell функции не вызывают, из них и не возвращаются. Чуть позже я поясню это.

Итак, на входе у функции `square` - единственный аргумент типа `Int` (стандартный тип в Haskell для обычных целых чисел), а на выходе - значение того же типа `Int`. Если же аргументов более одного, объявление просто вытягивается. Например, объявление упомянутой выше функции `sum` выглядит так:

```
sum :: Int -> Int -> Int
```

Схема объявления вытянулась, но суть осталась прежней:

```
sum         :: Int             -> Int             -> Int
имя функции    Тип 1 аргумента    Тип 2 аргумента    Тип вычисленного значения
```

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

Строго говоря, объявлять функцию необязательно, мы могли бы этого и не делать (и скоро я объясню, почему). Однако возьмите за правило всегда писать объявление, это сделает ваш код значительно более понятным и легко сопровождаемым.

Теперь рассмотрим определение функции.

Определяем

Вспомним определение функции `square`:

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

Схема определения такова:

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

А функция `sum` определена так:

```
sum          x                y                =    x + y
имя функции  имя 1 аргумента  имя 2 аргумента  это  выражение
```

Определение 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 языках семейства C закрепилось понятие «тело функции» (англ. function body), однако в Haskell 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:

```
sum x y = x + y
```

тем самым мы декларуем следующее: «Отныне выражение `sum 10 20` равно выражению `10 + 20`». То есть везде, где написано `sum 10 20`, мы можем совершенно спокойно подставить `10 + 20`, и работа программы гарантированно останется прежней.

А впрочем, откуда у меня такая уверенность? Ответ ищите в следующей главе, и приготовьтесь к удивлению.

Глава 8

Неизменность и чистота

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

Чисто функциональный?

Haskell - чисто функциональный (англ. purely functional) язык. И суть этого понятия уже изложена в предыдущей главе, осталось лишь сформулировать.

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 `sum`:

```
sum x y = x + y
```

Когда на входе `10` и `20` - на выходе всегда `30`, и ничто не способно помешать этому. Функция `sum` является чистой функцией, а потому характеризуется отсутствием побочных эффектов (англ. side effects). Именно поэтому чистая функция предельно надёжна: мы всегда можем заменить её применение её внутренним выражением, и результат останется неизменным. То есть если написано так:

```
square (sum 1 2)
```

мы можем написать и так:

```
square (1 + 2)
```

Казалось бы, в других языках, например в C, функция тоже может быть чистой:

```
int sum(int x, int y) {
    return x + y;
}
```

Однако это не совсем так. Да, результат функции `sum` здесь тоже определяется двумя аргументами, однако мы не можем гарантировать этого. И причиной тому являются переменные. Допустим:

```
int sum(int x, int y) {
    return x + y + 0.0049;
}
```

Мы ввели поправочный коэффициент `0.0049`, и теперь значение на выходе зависит (в том числе) и от него. Однако позже появилась другая функция, в теле которой понадобился тот же самый коэффициент. Мы, исповедуя принцип DRY (англ. Don't Repeat Yourself, Не Повторяй Себя), вынесли коэффициент в отдельную переменную `coeff` и переписали функцию так:

```
int sum(int x, int y) {
    return x + y + coeff;
}
```

Тут-то нас и поджидает проблема. Что будет, если значение `coeff` изменится в процессе работы программы? В этом случае и значение на выходе станет другим, поэтому мы, вызвав такую функцию с одинаковыми аргументами N раз, уже не можем быть полностью уверены в одинаковости результата. Хуже того, переменная `coeff` может оказаться глобальной, и её значение может изменяться совершенно другой функцией. В этом случае результат работы `sum` вполне может преподнести нам неприятный сюрприз, а потому она не может считаться чистой функцией.

Но, спросите вы, разве в Haskell мы не можем вынести общее значение в такой же `coeff`? Можем, но функция `sum` гарантированно останется чистой. И причина тому - неизменность данных (англ. data immutability).

«Присваивание? Не, не слышал...»

В мире Haskell нет места оператору присваивания. Этот факт, удивительный на первый взгляд, предельно логичным, если мы помним о математической природе функций в Haskell. Если каждая функция в конечном итоге представляет собою некое 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. Суть императивного программирования (англ. imperative programming) в том, что программа воспринимается как набор инструкций, работа которых неразрывно связана с изменением состояния этой программы. Вот почему в императивных языках обязательно присутствует понятие «переменная». А раз есть переменные - должен быть и инструмент для изменения их 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`.

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. И поскольку выражения вычисляются путём применения функций к аргументам (то есть, по сути, к другим выражениям), там нет места ни переменным, ни оператору присваивания. Любое конкретное значение в Haskell-программе, будучи созданным единожды, уже не может быть изменено, никогда и никем. Поэтому когда в Haskell-коде мы пишем:

```
coeff = 0.569;
```

мы просто объявляем/декларируем: «Отныне значение `coeff` равно `0.569`, и так оно будет всегда». В этом случае функции, использующие `coeff`, уже не способны преподнести нам сюрприз. Когда вы видите в Haskell-коде символ `=` - перед вам объявление равенства, а вовсе не присваивание.

Вы спросите, как же можно написать реальную программу на языке, в котором нельзя изменять данные? Оказывается, можно. Да, нам не дано изменить уже созданное значение, однако мы можем создать на его основе новое значение. А обязанности по уничтожению уже не нужных данных возложены на встроенный в Haskell сборщик мусора (англ. garbage collector).

Порядок вычислений

Императивный подход к разработке характеризуется ещё одним свойством, а именно жёстким порядком вычислений. Когда программа представляет собой набор инструкций, изменяющих состояние переменных, тогда и порядок выполнения этих инструкций жёстко задаётся программистом. Например:

```
int main() {
    int result1 = sum(10, 20);
    int result2 = mul(30, 40);
    /* Что-то делаем с результатами result1 и result2. */
    return 0;
}
```

Здесь функция `mul` делает почти то же, что и `sum`, только возвращает произведение двух аргументов, а не сумму. Вопрос: в каком порядке будут вызваны функции `sum` и `mul`? В том, в каком написаны: сначала `sum`, а потом `mul`.

При декларативном же подходе порядок вызова чистых функций нам, во-первых, не всегда известен, а во-вторых - и это главное! - он нам неинтересен.

Погодите, возразите вы, как это не всегда известен?! А вот так. Природа чистых функций такова, что нам больше не нужно думать о порядке вызова функций наподобие `sum` и `mul`, ведь результат их вычислений в конечном итоге окажется гарантированно правильным.

Я знаю, это удивляет. Мы привыкли давать чёткие приказания в нашем коде: «Сейчас вызови эту функцию, потом измени значение той переменной, после чего вызови вон ту функцию». То есть мы сфокусированы на том, **как** работает наша программа. А в декларативном мире мы не хотим думать о том, как работает наша программа, мы думаем лишь о том, **что** она в итоге сделает. Мы сфокусированы не на процессе вычислений, а на их результате, не на последовательности шагов, а на том, к чему они нас приведут.

Уверен, у вас уже уйма вопросов. Не спешите задавать их - по мере чтения книги всё встанет на свои места, и очень скоро вы убедитесь в том, что можно прекрасно обходиться без оператора присваивания.