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 да, 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, что называется, 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 в 2000-х началось его медленное вхождение в мир практического программирования. За эти годы язык совершенствовался и в 2010 мир увидел его обновлённый стандарт. Так что мы имеем дело вовсе не с молоденьким выскочкой.

«И кто ж 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 язык Haskell отношения не имеет.

«Хорошо, а библиотеки для Haskell имеются? Или велосипедить придётся?»

Библиотеки имеются, и очень много. А ещё имеется единый открытый репозиторий библиотек с удобным механизмом их установки.

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

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 и 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 странностях языка... Да, собственно, вся книга им и посвящена.

«Ну ладно, а если сравнить его с C++/Python/Scala/Assembler...»

Сравнение Haskell с другими языками выходит за рамки этой книги. Да, несколько раз вы увидите 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».

Об этой книге

В последние годы заметно возросло число число книг, посвящённых Haskell, и это радует. Каждая из них преследует свою цель, и поэтому трудно сказать, какая из них лучше. Цель этой книги в том, чтобы дать вам понимание фундаментальных основ Haskell. Без усвоения этих основ 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. Важно отметить, что с определённой точки зрения программировать на Haskell как раз-таки очень просто, но лишь после того, как вы близко познакомились с Тремя Китами Haskell, имена которых: **Функция**, **Тип** и **Класс типов**. Всё в этом языке зиждется на этих трёх основах. Именно поэтому рассмотрение оных - важнейшая цель настоящей книги. И именно поэтому объём её относительно невелик, ведь, к счастью для нас, разобраться с этими основами не так уж и тяжело. Тем более что сфокусированы мы будем не на академизме, а на практичности.

Разумеется, помимо Трёх Китов, в Haskell есть много чего другого. В частности, ленивая стратегия вычислений, жёсткое отделение чистого кода от кода с побочными эффектами, великолепная способность к параллельному и конкурентному программированию. Однако без понимания Трёх Китов путь к высотам Haskell для вас закрыт.

И ещё, добрый вам совет: не пытайтесь понять Haskell-термины, опираясь на ваш предыдущий опыт. Например, в ряде ОО-языков понятия «тип» и «класс» фактически взаимозаменяемы, в то время как в Haskell эти понятия абсолютно

различны. Попытайтесь, хоть это и тяжело, учить Haskell как бы с нуля, с чистого листа. Если же Haskell оказался вашим первым языком программирования, тогда я искренне рад за вас, ведь вам не придётся ломать сознание.

О первом и втором издании

На обложке книги вы видели метку «издание 2.0». Перед вами второе издание, полностью переработанное и переосмысленное. Да, на момент публикации второй версии 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. На момент написания первой версии я ещё много не знал и не понимал, поэтому некоторая информация из первого издания откровенно бедна, а несколько глав вообще вводят читателя в заблуждение. К тому же тогда я не имел опыта реальной работы с Haskell, потому и отношение к языку было несколько иллюзорное, розоватое.

Вторая - изменившаяся цель книги. Я намеренно сузил круг рассматриваемых здесь тем. Теперь эта книга полностью посвящена основам языка, она далеко не обо всём, что есть в современном Haskell. Поэтому не ждите от неё рассмотрения специфических тем. Например, Haskell действительно силён в области параллельного и конкурентного программирования, однако здесь вы не найдёте об этом ни строчки. Да и зачем, если этим важным темам посвящена отдельная книга, да ещё и переведённая на русский?

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 этим интересным темам посвящены другие книги и статьи. И если вы освоите изложенное здесь, то все остальные публикации о Haskell будут вам по плечу.

Про теорию категорий, раз и навсегда

Не имея шансов обойти эту тему стороной, сразу расставлю точки над «ё».

Действительно, язык Haskell имеет глубокие математические корни. И ряд важных понятий в Haskell родом из раздела современной математики, называемого теорией категорий. Однако вам знать эту теорию вовсе необязательно. Математическое происхождение таких понятий, как «функтор» или «монада», с которыми мы познакомимся позже, не обязывает нас знать научные корни этих понятий. Главное - увидеть, как эти понятия пригодятся нам в нашей повседневной работе.

Я вовсе не против ознакомления с теорией. Знать корни - это неплохо. Однако знакомство с теорией едва ли сделает вас лучшим практиком. Я могу сравнить это с музыкой, на примере гитары. Знание истории происхождения нотной записи, или истоков названия нот, или имени изобретателя первой испанской гитары - всё это

весьма любопытно, но это знание не улучшит ваши навыки игры на гитаре. Ведь для того, чтобы научиться играть на гитаре, необходимо играть на гитаре. Вот этим мы и займёмся - Практикой.

Как читать эту книгу

Настоятельно рекомендую читать её строго последовательно, от начала до конца. 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.

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, прекрасно 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 утилитка 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 - обязательно 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 Haskell-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 настоящих приложений и библиотек.

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

Разворачиваем инфраструктуру

Делаем:

```
$ stack setup
```

В результате на ваш компьютер будет установлена Haskell-инфраструктура последней стабильной версии. Жить всё это хозяйство будет в только что созданном скрытом 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 Haskell-проект:

```
$ stack new real
```

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

```
.
  LICENSE
  Setup.hs
  app
      Main.hs
  real.cabal
  src
      Lib.hs
  stack.yaml
  test
       Spec.hs
```

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

```
$ 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 же запустить программу:

```
$ real-exe
someFunc
```

Только что мы создали настоящий Haskell-проект и запустили нашу первую программу, выведшую строку “someFunc”. Но как же это работает? Пришла пора познакомится с модулями.

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, нужно 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.

Теперь откроем файл `app/Main.hs`:

```
module Main where

import Lib       --            Lib...

main :: IO ()
main = someFunc  --                  Lib...
```

Это - модуль `Main`, главный модуль наш приложения, ведь именно здесь определена функция `main`. С помощью `import` мы включаем содержимое модуля `Lib` и можем работать с этим содержимым.

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

Существует 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, во избежание путаницы.

Всё. В будущих главах вы узнаете о модулях кое-что ещё, но пока достаточно этого. А теперь пора познакомится с тем, что мы будем использовать в наших проектах постоянно, а именно с пакетами.

Hackage

Hackage - это главный репозиторий Haskell-библиотек, или, как у нас принято называть, пакетов (англ. package). Название 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 (напоминаю, модуль - это `.hs`-файл). Существуют большие пакеты, состоящие из многих десятков модулей, но есть и такие, в которых модуль всего один.

Итак, 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.

Существует также особый поисковик по Hackage. Строго говоря, их два, названных в честь поисковых гигантов:

1. Hoogle
2. Hayoo!

Эти поисковые движки скрупулёзно осматривают внутренности Hackage, и мы будем очень часто ими пользоваться. Особенно полезны они тогда, когда мы хотим найти информацию по некоторому API. Например, когда мы знаем имя функции и хотим найти пакет, в котором она живёт.

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 следующей схеме: http://hackage.haskell.org/package/ИМЯПАКЕТА.

Включаем

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

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

Теперь выполняем:

```
$ stack install
```

Начнётся повторная сборка нашего 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.

О Прелюдии

Существует один стандартный модуль, который по умолчанию автоматически импортируется во все наши модули (поэтому нам не придётся импортировать его с помощью `import`). Имя ему - Prelude. В этому модуле содержатся самые базовые Haskell-инструменты, многие из которых мы будем использовать постоянно.

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
```

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
               1                2
```

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
```

А функция `sum` определена так:

```
sum          x                y                =    x + y
           1           2
```

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.

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);
    /*  -               result1  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.

Функции внутри

В предыдущей главе я одним махом открыл вам много нового. Пришло время разбираться более детально. В этой главе рассмотрим содержимое функций. Ведь раз уж Haskell-программа состоит из выражений, а те, в свою очередь, представляют

собой применение неких функций к неким аргументам, то можно сказать, что всё, происходящее внутри программы, происходит внутри функций. Именно поэтому содержимое функций представляет для нас исключительный интерес.

В прошлой главе мы увидели простейшие функции, такие как `sum`:

```
sum :: Int -> Int -> Int
sum x y = x + y
```

Вот её содержимое:

```
x + y
```

Согласитесь, не очень интересно. Да, нужны и такие, но в реальных программах нужны штуки помощнее. Но прежде чем погрузиться в функции, познакомимся с некоторыми простейшими типами в Haskell, без них нам не обойтись.

С типом `Int` вы уже знакомы, значения этого типа содержат обыкновенные целые числа. Тип `Double` предназначен для чисел с плавающей точкой. Значения типа `Char` - это Unicode-символы. Значение типа `String` - обычная строка. Значение типа `Bool` отражает истину (`True`) или ложь (`False`). Вот примеры:

```
123                 -- Int

2.0678              -- Double

'A'                 -- Char

"Hello, Haskeller!" -- String

True                -- Bool,

False               -- Bool,
```

Пока достаточно. В Haskell есть ещё несколько важных стандартных типов, но о них узнаем чуть позже. И кстати, вы уже второй раз видите конструкцию, состоящую из двух минусов `--`. Это - начало однострочного комментария. А между символами `{-` и `-}` живёт многострочный комментарий:

```
--   -               ,

{-
      -
         ,
            !
-}
```

Ну а теперь прыгнем в функции.

Условие

Так выглядит условная конструкция:

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

где `CONDITION` - 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" || ip == "::1"
        then "It's a localhost!"
        else "No, it's not a localhost."
```

Функция `checkLocalhost` применяется к единственному аргументу типа `String` и возвращает/вычисляет другое значение типа `String`. В качестве аргумента выступает строка, содержащая IP-адрес, а функция проверяет, не лежит ли в ней localhost. Оператор `||` - стандартый оператор логического «ИЛИ», а оператор `==` - стандартный оператор проверки на равенство.

Теперь откроем наш модуль `app/Main.hs`, найдём функцию `main` и напишем следующее:

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

Вспоминаем команду сборки, которую мы запускаем, находясь в корне нашего проекта:

```
$ stack install
```

После сборки запускаем:

```
$ real-exe
It's a localhost!
```

А если перепишем так:

```
main :: IO ()
main = putStrLn (checkLocalhost "173.194.222.100")
```

тогда при запуске увидим это:

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

Функция `putStrLn` - это стандартная функция, выводящая на консоль некую строку. В данном случае она вывела строку, возвращённую функцией `checkLocalhost`.

Возврат из функции

В императивных языках, подобных C, существует особая инструкция `return`, предназначенная для явного возврата значения из функции. Поскольку функции

там - это подпрограммы, наличие такой инструкции необходимо потому, что точек выхода/возврата из функции может быть несколько. Например, на C++ функция `checkLocalhost` могла бы выглядеть так:

```
std::string checkLocalhost(const std::string& ip) {
    if(ip == "127.0.0.1" || ip == "0.0.0.0" || ip == "::1") {
        return "It's a localhost!";     //        ...
    }

    return "No, it's not a localhost."  // ...       .
}
```

В Haskell не существует аналога инструкции `return`, поскольку единственной «точкой возврата» из функции является её конечный результат. Звучит необычно, поясню примером.

Как мы знаем, применение функции - это выражение. А так как оперирует оно с неизменными данными, мы всегда можем заменить выражение своим конечным результатом. И ничего не сломается. Например, выражение:

```
checkLocalhost "127.0.0.1"
```

может быть заменено

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому

в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть

указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть

указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть

указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Stackage

В предыдущей главе мы познакомились с Hackage, репозиторием Haskell-пакетов. Теперь же настало время познакомится со “стабильным Hackage”.

Проблема

В станодавние времена существовала в мире Haskell большая проблема, связанная с пакетами, и называлась она “dependency hell”. Вот в чём её суть.

Как вы уже поняли, каждый из Haskell-пакетов тоже имеет ряд зависимостей от неких других пакетов, и эти зависимости перечислены в его `.cabal`-файле. Зависимость может быть указана “версионной вилкой”, например:

```
http-client  >= 0.3 && < 0.5
```

Это означает, что данный пакет зависит от пакета `http-client` любой версии, входящей в промежуток от `0.3` включительно до `0.5`. Однако зависимость может быть указана точно:

```
http-client  == 0.3.6.1
```

В этом случае нам нужна версия `0.3.6.1` и никакая другая. Вот тут-то нас и поджидает ад.

Допустим, наш проект зависит от двух пакетов, `A` и `B`, каждый из которых в свою очередь зависит от третьего пакета `C`. Но к сожалению, звёзды оказались неблагосклонны к нашему проекту, и выяснилось, что пакет `A` зависит от пакета `C` версии `1.8.1`, в то время как пакет `B` - от пакета `C` версии `2.0.0`... Эту проблему пытались решить несколькими способами, но самым удобным оказался способ под названием `Stackage`.

Что это такое?

Название Stackage происходит от слияния слов “Stable” и “Hackage”. Идея в том, чтобы предоставить разработчику надёжный набор пакетов, на который гарантированно можно положиться.

В основе этой идеи лежит LTS Haskell snapshot, или LTS-снимок (от Long-Term Support). LTS-снимок представляет собой большой список пакетов жёстко заданных версий. И подобраны эти версии таким образом, чтобы все пакеты были версионно совместимы друг с другом. Гарантированно и железобетонно. То есть, возвращаясь к нашему примеру, если в LTS-снимке перечислены пакеты `A` и `B`, то их версии подобраны так, чтобы они зависели от одной и той же версии пакета `C`. В этом случае у нас никогда не возникнет никаких версионных коллизий.

Откроем файл `stack.yaml` в корне нашего проекта и найдём там строчку вида:

```
resolver: lts-5.2
```

В этой строке мы задаём номер LTS-снимка, используемого в данном проекте. Да, у каждого снимка есть собственный номер, или, если хотите, версия. В данном случае мы объявляем, что наш проект “живёт” в рамках LTS-снимка под номером `5.2`. Это означает, что если наш проект зависит от пакета `text`, то не от абы какой версии `text`, но именно от версии `1.2.2.0`, ведь именно эта версия упомянута в данном LTS-снимке.

Кстати, содержимое снимка `5.2` можно посмотреть здесь. В конце адреса меняем номер - и видим содержимое другого снимка.

Неизменность

LTS-снимки хороши и тем, что неизменны. В упомянутом снимке `5.2` перечислены 1766 пакетов конкретных версий, и так оно будет всегда. Следовательно, если члены команды Haskell-разработчиков, работающие на одним и тем же проектом, используют один и тот же снимок - у всех у них всё будет компилироваться. Сегодня, завтра и через пять лет. Создатели Stackage провозгласили это одним из девизов данной идеи: “То, что работает сегодня, должно работать и завтра”.

Утилита stack?

Да, совершенно верно: уже знакомая нам утилита `stack` является частью проекта Stackage, оттого и схожесть в названии. Поэтому, используя `stack`, вы уже используете какой-то LTS-снимок.

А если не из репозитория?

В самом деле, представим себе такую ситуацию. Некий разработчик сделал полезную Haskell-библиотеку, но не стал включать её в единый репозиторий, а вместо этого поселил её в своём GitHub-профиле. А нам эта библиотека очень уж приглянулась. Как же включить её в проект? Открываем `stack.yaml` и пишем:

```
- location:
    git: https://github.com/user/project.git
    commit: 018919f43854b1d6f30d4270f5471db807ac1a41
```

Думаю, тут всё понятно без комментариев: пакет будет вытянут из данного репозитория (с использованием указанного коммита), собран и включён в наш проект. И не забудьте включить имя данного пакета в списке зависимостей в `.cabal`-файле.

Ещё почитать

Я написал небольшую обзорную статью, посвящённую `stack` и работе с LTS-снимками.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.

Неизменность данных

Как было упомянуто ранее, одной из фундаментальных черт языка Haskell является отсутствие оператора присваивания.

Знаете, услышав о котором впервые, я не поверил своим ушам. Как это можно программировать без оператора присваивания?! А как же мы будем изменять состояние наших переменных?

Чтобы разобраться, рассмотрим такую строку:

```
a = 123
```

В императивном языке такая инструкция означает присваивание. В этом случае мы приказываем: “Возьми совокупность байтов, соответствующую значению 123, и замени ею ту совокупность байтов, которая хранилась в переменной `a` до этого”. Таким образом, происходит перезапись старого значения новым.

Однако в чисто функциональном языке такая инструкция означает то же, что она означает в математике, а именно равенство. В этом случае мы объявляем: “Значение `a` равно 123”.

Вы спросите, в чём разница? Мы ведь в любом случае получаем переменную `a`, равную 123. А разница в том, что присваивание значения перменной может происходить множество раз, в то время как объявление равенства может быть указано лишь единожды. И если мы объявили, что значение `a` равно 123 - так оно и будет до скончания времён, и ничто не сможет этого изменить. Именно поэтому в языке Haskell нет ни понятия “переменная”, ни ключевого слова `const`, ведь все значения в этом языке константны по своей сути.

Вы спросите, как же мы сможем добавить элемент в какой-нибудь список, если у нас всё константное? Ответ: никак. Мы не можем *изменить* имеющееся значение, мы можем лишь создать на его основе *новое* значение. О памяти не беспокойтесь: выделена она будет автоматически, равно как и уничтожена, ответственность за это ляжет на встроенный в Haskell сборщик мусора.

Вскоре вы убедитесь, что без оператора присваивания можно прекрасно обходиться.