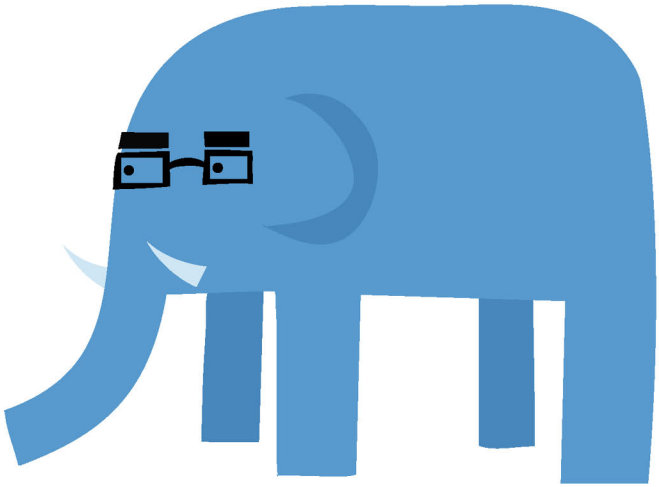
Изучай Haskell во имя добра!

Для начинающих



Миран Липовача

no starch press

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

УДК 004.432.42Haskell
ББК 32.973.28-018.1
Л61

Л61 **Миран Липовача**

Изучай Haskell во имя добра! / Пер. с англ. Леушина Д., Синицына А., Арсанукаева Я.– М.: ДМК Пресс, 2012. – 490 с.: ил.

ISBN 978-5-94074-749-9

На взгляд автора, сущность программирования заключается в решении проблем. Программист всегда думает о проблеме и возможных решениях – либо пишет код для выражения этих решений.

Язык Haskell имеет множество впечатляющих возможностей, но главное его свойство в том, что меняется не только способ написания кода, но и сам способ размышления о проблемах и возможных решениях. Этим Haskell действительно отличается от большинства языков программирования. С его помощью мир можно представить и описать нестандартным образом. И поскольку Haskell предлагает совершенно новые способы размышления о проблемах, изучение этого языка может изменить и стиль программирования на всех прочих.

Ещё одно необычное свойство Haskell состоит в том, что в этом языке придаётся особое значение рассуждениям о типах данных. Как следствие, вы помещаете больше внимания и меньше кода в ваши программы.

Вне зависимости от того, в каком направлении вы намерены двигаться, путешествуя в мире программирования, небольшой заход в страну Haskell себя оправдает. А если вы решите там остаться, то наверняка найдёте чем заняться и чему поучиться!

Эта книга поможет многим читателям найти свой путь к Haskell.

УДК 004.432.42Haskell
ББК 32.973.28-018.1

ISBN 978-1-59327-283-8 (англ.)
ISBN 978-5-94074-749-9

Оглавление

Введение

Перед вами книга «Изучай Haskell во имя добра!» И раз уж вы взялись за её чтение, есть шанс, что вы хотите изучить язык Haskell. В таком случае вы на правильном пути – но прежде чем продолжить его, давайте поговорим о самом учебнике.

Я решился написать это руководство потому, что захотел упорядочить свои собственные знания о Haskell, а также потому, что надеюсь помочь другим людям в освоении этого языка. В сети Интернет уже предостаточно литературы по данной теме, и когда я сам проходил период ученичества, то использовал самые разные ресурсы.

Чтобы поподробнее ознакомиться с Haskell, я читал многочисленные справочники и статьи, в которых описывались различные аспекты при помощи различных методов. Затем я собрал воедино все эти разрозненные сведения и положил их в основу собственной книги. Так что этот учебник представляет собой попытку создать ещё один полезный ресурс для изучения языка Haskell – и есть вероятность, что вы найдёте здесь именно то, что вам нужно!

Эта книга рассчитана на людей, которые уже имеют опыт работы с императивными языками программирования (C++, Java, Python...), а теперь хотели бы попробовать Haskell. Впрочем, бьюсь об заклад, что даже если вы не обладаете солидным опытом программирования, с вашей природной смекалкой вы легко освоите Haskell, пользуясь этим учебником!

Моей первой реакцией на Haskell было ощущение, что язык какой-то уж слишком чудной. Но после преодоления начального барьера всё пошло как по маслу. Даже если на первый взгляд Haskell кажется вам странным, не сдавайтесь! Освоение этого языка похоже на изучение программирования «с нуля» – и это очень занимательно, потому что вы начинаете мыслить совершенно иначе...

ПРИМЕЧАНИЕ. *IRC-канал* `#haskell` *на Freenode Network – отличный ресурс для тех, кто испытывает затруднения в обучении и хочет задать вопросы по какой-либо теме. Люди там чрезвычайно приятные, вежливые и с радостью помогают новичкам.*

Так что же такое Haskell?

Язык Haskell – это *чисто функциональный* язык программирования. В *императивных* языках результат достигается при передаче компьютеру последовательности команд, которые он затем выполняет. При этом компьютер может изменять своё состояние. Например, мы устанавливаем переменную `a` равной 5, производим какое-либо действие, а затем меняем её значение... Кроме того, у нас есть управляющие инструкции, позволяющие повторять несколько раз определённые действия, такие как циклы `for` и `while`. В чисто функциональных языках вы не говорите компьютеру, *как* делать те или иные вещи, – скорее вы говорите, что представляет собой ваша проблема.

Факториал числа – это произведение целых чисел от 1 до данного числа; сумма списка чисел – это первое число плюс сумма всех остальных чисел, и так далее. Вы можете выразить обе эти операции в виде *функций*. В функциональной программе нельзя присвоить переменной сначала одно значение, а затем какое-то другое. Если вы решили, что `a` будет равняться 5, то потом уже не сможете просто передумать и заменить значение на что-либо ещё. В конце концов, вы же сами сказали, что `a` равно 5! Вы что, врун какой-нибудь?

В чисто функциональных языках у функций *отсутствуют побочные эффекты*. Функция может сделать только одно: рассчитать что-нибудь и возвратить это как результат. Поначалу такое ограничение смущает, но в действительности оно имеет приятные последствия: если функция вызывается дважды с одними и теми же параметрами, это гарантирует, что оба раза вернётся одинаковый результат. Это свойство называется *ссылочной прозрачностью*. Оно позволяет

программисту легко установить (и даже доказать), что функция корректна, а также строить более сложные функции, объединяя простые друг с другом.

Haskell – *ленивый* язык. Это означает, что он не будет выполнять функции и производить вычисления, пока это действительно вам не потребовалось для вывода результата (если иное не указано явно). Подобное поведение возможно как раз благодаря ссылочной прозрачности. Если вы знаете, что результат функции зависит только от переданных ей параметров, неважно, в какой именно момент вы её вызываете. Haskell, будучи ленивым языком, пользуется этой возможностью и откладывает вычисления на то время, на какое это вообще возможно. Как только вы захотите отобразить результаты, Haskell проделает минимум вычислений, достаточных для их отображения. Ленивость также позволяет создавать бесконечные структуры данных, потому что реально вычислять требуется только ту часть структуры данных, которую необходимо отобразить.

Предположим, что у нас есть неизменяемый список чисел `xs = [1,2,3,4,5,6,7]` и функция `doubleMe` («УдвойМеня»), которая умножает каждый элемент на 2 и затем возвращает новый список. Если мы захотим умножить наш список на 8 в императивных языках, то сделаем так:

```
doubleMe(doubleMe(doubleMe(xs)))
```

При вызове, вероятно, будет получен список, а затем создана и возвращена копия. Затем список будет получен ещё два раза – с возвращением результата. В ленивых языках программирования вызов `doubleMe` со списком без форсирования получения результата означает, что программа скажет вам что-то вроде: «Да-да, я сделаю это позже!». Но когда вы захотите увидеть результат, то первая функция `doubleMe` скажет второй, что ей требуется результат, и немедленно! Вторая функция передаст это третьей, и та неохотно вернёт удвоенную 1, то есть 2.

Вторая получит и вернёт первой функции результат – 4. Первая увидит результат и выдаст вам 8. Так что потребуется только один

проход по списку, и он будет выполнен только тогда, когда действительно окажется необходим.

Язык Haskell – *статически типизированный* язык. Когда вы компилируете вашу программу, то компилятор знает, какой кусок кода – число, какой – строка и т. д. Это означает, что множество возможных ошибок будет обнаружено во время компиляции. Если, скажем, вы захотите сложить вместе число и строку, то компилятор вам «пожалуется».

В Haskell есть очень хорошая система типов, которая умеет автоматически делать вывод типов. Это означает, что вам не нужно описывать тип в каждом куске кода, потому что система типов может вычислить это сама. Если, скажем, `a = 5 + 4`, то вам нет необходимости говорить, что `a` – число, так как это может быть выведено автоматически. Вывод типов делает ваш код более универсальным. Если функция принимает два параметра и складывает их, а тип параметров не задан явно, то функция будет работать с любыми двумя параметрами, которые ведут себя как числа.

Haskell – *ясный* и *выразительный* язык, потому что он использует множество высокоуровневых идей; программы обычно короче, чем их императивные эквиваленты, их легче сопровождать, в них меньше ошибок.

Язык Haskell был придуман несколькими по-настоящему умными ребятами (с диссертациями). Работа по его созданию началась в 1987 году, когда комитет исследователей задался целью изобрести язык, который станет настоящей сенсацией. В 1999 году было опубликовано описание языка (Haskell Report), ознаменовавшее появление первой официальной его версии.

Что понадобится для изучения языка

Если коротко, то для начала понадобятся текстовый редактор и компилятор Haskell. Вероятно, у вас уже установлен любимый

редактор, так что не будем заострять на этом внимание. На сегодняшний день самым популярным компилятором Haskell является GHC (Glasgow Haskell Compiler), который мы и будем использовать в примерах ниже. Проще всего обзавестись им, скачав Haskell Platform, которая включает, помимо прочего, ещё и массу полезных библиотек. Для получения Haskell Platform нужно пойти на сайт http://hackage.haskell.org/platform/ и далее следовать инструкциям по вашей операционной системе.

GHC умеет компилировать сценарии на языке Haskell (обычно это файлы с расширением *.hs*), а также имеет интерактивный режим работы, в котором можно загрузить функции из файлов сценариев, вызвать их и тут же получить результаты. Во время обучения такой подход намного проще и эффективнее, чем перекомпиляция сценария при каждом его изменении, а затем ещё и запуск исполняемого файла.

Как только вы установите Haskell Platform, откройте новое окно терминала – если, конечно, используете Linux или Mac OS X. Если же у вас установлена Windows, запустите интерпретатор командной строки (*cmd.exe*). Далее введите `ghci` и нажмите **Enter**. Если ваша система не найдёт программу GHCi, попробуйте перезагрузить компьютер.

Если вы определили несколько функций в сценарии, скажем, *myfunctions.hs*, то их можно загрузить в GHCi, напечатав команду `:l myfunctions`. Нужно только убедиться, что файл *myfunctions.hs* находится в том же каталоге, из которого вы запустили GHCi.

Если вы изменили *hs*-сценарий, введите в интерактивном режиме `:l myfunctions`, чтобы загрузить его заново. Можно также перегрузить загруженный ранее сценарий с помощью команды `:r`. Обычно я поступаю следующим образом: определяю несколько функций в *hs*-файле, загружаю его в GHCi, экспериментирую с функциями, изменяю файл, перезагружаю его и затем всё повторяю. Собственно, именно этим мы с вами и займёмся.

Благодарности

Благодарю всех, кто присылал мне свои замечания, предложения и слова поддержки. Также благодарю Кита, Сэма и Мэрилин, которые помогли мне отшлифовать мастерство писателя.

1

НА СТАРТ, ВНИМАНИЕ, МАРШ!

Отлично, давайте начнём! Если вы принципиально не читаете предисловий к книгам, в данном случае вам всё же придётся вернуться назад и заглянуть в заключительную часть введения: именно там рассказано, что вам потребуется для изучения данного руководства и для загрузки программ.

Первое, что мы сделаем, – запустим компилятор GHC в интерактивном режиме и вызовем несколько функций, чтобы «прочувствовать» язык Haskell – пока ещё в самых общих чертах. Откройте консоль и наберите `ghci`. Вы увидите примерно такое приветствие:

```
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

Поздравляю – вы в GHCi!

ПРИМЕЧАНИЕ. *Приглашение консоли ввода – `Prelude>`, но поскольку оно может меняться в процессе работы, мы будем использовать просто `ghci>`. Если вы захотите, чтобы у вас было такое же приглашение, выполните команду `:set prompt "ghci> "`.*

Немного школьной арифметики:

```
ghci> 2 + 15
17
ghci> 49 * 100
```

```
4900
ghci> 1892 - 1472
420
ghci> 5 / 2
2.5
```

Код говорит сам за себя. Также в одной строке мы можем использовать несколько операторов; при этом работает обычный порядок вычислений. Можно использовать и круглые скобки для облегчения читаемости кода или для изменения порядка вычислений:

```
ghci> (50 * 100) - 4999
1
ghci> 50 * 100 - 4999
1
ghci> 50 * (100 - 4999)
-244950
```

Здорово, правда? Чувствую, вы со мной не согласны, но немного терпения! Небольшая опасность кроется в использовании отрицательных чисел. Если нам захочется использовать отрицательные числа, то всегда лучше заключить их в скобки. Попытка выполнения `5 * -3` приведёт к ошибке, зато `5 * (-3)` сработает как надо.

Булева алгебра в Haskell столь же проста. Как и во многих других языках программирования, в Haskell имеется два логических значения `True` и `False`, для конъюнкции используется операция `&&` (логическое «И»), для дизъюнкции – операция `||` (логическое «ИЛИ»), для отрицания – операция `not`.

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True&&True)
False
```

Можно проверить два значения на равенство и неравенство с помощью операций `==` и `/=`, например:

```
ghci> 5 == 5
True
ghci> 1 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 4
True
ghci> "привет" == "привет"
True
```

А что насчёт `5 + лама` или `5 == True`? Если мы попробуем выполнить первый фрагмент, то получим большое и страшное сообщение об ошибке[1]!

```
No instance for (Num [Char])
arising from a use of `+' at <interactive>:1:0-9
Possible fix: add an instance declaration for (Num [Char])
In the expression: 5 + "лама"
In the definition of `it': it = 5 + "лама"
```

Та-ак! GHCi говорит нам, что `лама` не является числом, и непонятно, как это прибавить к 5. Даже если вместо `лама` подставить `четыре` или `4`, Haskell всё равно не будет считать это числом! Операция + ожидает, что аргументы слева и справа будут числовыми. Если же мы попытаемся посчитать `True == 5`, GHCi опять скажет нам, что типы не совпадают.

Несмотря на то что операция + производится только в отношении элементов, воспринимаемых как число, операция сравнения (==), напротив, применима к любой паре элементов, которые можно сравнить. Фокус заключается в том, что они должны быть одного типа. Вы не сможете сравнивать яблоки и апельсины. В подробностях мы это обсудим чуть позже.

ПРИМЕЧАНИЕ. *Запись `5 + 4.0` вполне допустима, потому что `5` может вести себя как целое число или как число с плавающей точкой. `4.0` не может выступать в роли целого числа, поэтому именно число `5` должно «подстроиться».*

[1] В современных версиях интерпретатора GHCi для печати результатов вычислений используется функция `show`, которая представляет кириллические символы соответствующими числовыми кодами Unicode. Поэтому в следующем листинге вместо строки "лама" будет фактически выведено "\1083\1072\1084\1072". В тексте книги для большей понятности кириллица в результатах оставлена без изменений. – *Прим. ред.*

Вызов функций

Возможно, вы этого пока не осознали, но все это время мы использовали функции. Например, операция `*` – это функция, которая принимает два числа и перемножает их. Как вы видели, мы вызываем её, вставляя символ `*` между числами. Это называется «*инфиксной* записью».

Обычно функции являются префиксными, поэтому в дальнейшем мы не будем явно указывать, что функция имеет префиксную форму – это будет подразумеваться. В большинстве императивных языков функции вызываются указанием имени функции, а затем её аргументов (как правило, разделенных запятыми) в скобках. В языке Haskell функции вызываются указанием имени функции и – через пробел – параметров, также разделенных пробелами. Для начала попробуем вызвать одну из самых скучных функций языка:

```
ghci> succ 8
9
```

Функция `succ` принимает на вход любое значение, которое может иметь последующее значение, после чего возвращает именно последующее значение. Как вы видите, мы отделяем имя функции от параметра пробелом. Вызывать функции с несколькими параметрами не менее просто.

Функции `min` и `max` принимают по два аргумента, которые можно сравнивать (как и числа!), и возвращают большее или меньшее из значений:

```
ghci> min 9 10
9
ghci> min 3.4 3.2
3.2
ghci> max 100 101
101
```

Операция применения функции (то есть вызов функции с указанием списка параметров через пробел) имеет наивысший приоритет. Для нас это значит, что следующие два выражения эквивалентны:

```
ghci> succ 9 + max 5 4 + 1
16
ghci> (succ 9) + (max 5 4) + 1
16
```

Однако если мы хотим получить значение, следующее за произведением чисел 9 и 10, мы не можем написать `succ 9 * 10`, потому что это даст значение, следующее за 9 (т. е. 10), умноженное на 10, т. е. 100. Следует написать `succ (9 * 10)`, чтобы получить 91.

Если функция принимает ровно два параметра, мы также можем вызвать её в инфиксной форме, заключив её имя в обратные апострофы. Например, функция `div` принимает два целых числа и выполняет их целочисленное деление:

```
ghci> div 92 10
9
```

Но если мы вызываем её таким образом, то может возникнуть неразбериха с тем, какое из чисел делимое, а какое делитель. Поэтому можно вызвать функцию в инфиксной форме, что, как оказывается, гораздо понятнее[2]:

```
ghci> 92 `div` 10
9
```

Многие люди, перешедшие на Haskell с императивных языков, придерживаются мнения, что применение функции должно обозначаться скобками. Например, в языке C используются скобки для вызова функций вроде `foo()`, `bar(1)` или `baz(3, xa-xa)`. Однако, как мы уже отмечали, для применения функций в Haskell предусмотрены пробелы. Поэтому вызов соответствующих функций производится следующим образом: `foo`, `bar 1` и `baz 3 xa-xa`. Так что если вы увидите выражение вроде `bar (bar 3)`, это не значит, что `bar` вызывается с параметрами `bar` и `3`. Это значит, что мы сначала вызываем функцию `bar` с параметром `3`, чтобы получить некоторое число, а затем опять вызываем `bar` с этим числом в качестве параметра. В языке C это выглядело бы так: “`bar(bar(3))`”.

[2] На самом деле любую функцию, число параметров которой больше одного, можно записать в инфиксной форме, заключив её имя в обратные апострофы и поместив её в таком виде ровно между первым и вторым аргументом. – *Прим. ред.*

Функции: первые шаги

Определяются функции точно так же, как и вызываются. За именем функции следуют параметры[3], разделенные пробелами. Но при определении функции есть ещё символ `=`, а за ним – описание того, что функция делает. В качестве примера напишем простую функцию, принимающую число и умножающую его на 2. Откройте свой любимый текстовый редактор и наберите в нём:

```
doubleMe x = x + x
```

Сохраните этот файл, например, под именем *baby.hs*. Затем перейдите в каталог, в котором вы его сохранили, и запустите оттуда GHCi. В GHCi выполните команду `:l baby`. Теперь наш сценарий загружен, и можно поупражняться с функцией, которую мы определили:

```
ghci> :l baby
[1 of 1] Compiling Main             ( baby.hs, interpreted )
Ok, modules loaded: Main.
ghci> doubleMe 9
18
ghci> doubleMe 8.3
16.6
```

Поскольку операция + применима как к целым числам, так и к числам с плавающей точкой (на самом деле – ко всему, что может быть воспринято как число), наша функция одинаково хорошо работает с любыми числами. А теперь давайте напишем функцию, которая принимает два числа, умножает каждое на два и складывает их друг с другом. Допишите следующий код в файл *baby.hs*:

```
doubleUs x y = x*2 + y*2
```

ПРИМЕЧАНИЕ. *Функции в языке Haskell могут быть определены в любом порядке. Поэтому совершенно неважно, в какой последовательности приведены функции в файле baby.hs.*

[3] На самом деле в определении функций они называются образцами, но об этом пойдёт речь далее. – *Прим. ред.*

Теперь сохраните файл и введите `:l baby` в GHCi, чтобы загрузить новую функцию. Результаты вполне предсказуемы:

```
ghci> doubleUs 4 9
26
ghci> doubleUs 2.3 34.2
73.0
ghci> doubleUs 28 88 + doubleMe 123
478
```

Вы можете вызывать свои собственные функции из других созданных вами же функций. Учитывая это, можно переопределить `doubleUs` следующим образом:

```
doubleUs x y = doubleMe x + doubleMe y
```

Это очень простой пример общего подхода, применяемого во всём языке – создание простых базовых функций, корректность которых очевидна, и построение более сложных конструкций на их основе.

Кроме прочего, подобный подход позволяет избежать дублирования кода. Например, представьте себе, что какие-то «математики» решили, будто 2 – это на самом деле 3, и вам нужно изменить свою программу. Тогда вы могли бы просто переопределить `doubleMe` как `x + x + x`, и поскольку `doubleUs` вызывает `doubleMe`, данная функция автоматически работала бы в странном мире, где 2 – это 3.

Теперь давайте напишем функцию, умножающую число на два, но только при условии, что это число меньше либо равно 100 (поскольку все прочие числа и так слишком большие!):

```
doubleSmallNumber x = if x > 100
                      then x
                      else x*2
```

Мы только что воспользовались условной конструкцией `if` в языке Haskell. Возможно, вы уже знакомы с условными операторами из других языков. Разница между условной конструкцией `if` в Haskell и операторами `if` из императивных языков заключается в том, что ветвь `else` в языке Haskell является обязательной. В императивных языках вы можете просто пропустить пару шагов, если

условие не выполняется, а в Haskell каждое выражение или функция должны что-то возвращать[4].

Можно было бы написать конструкцию `if` в одну строку, но я считаю, что это не так «читабельно». Ещё одна особенность условной конструкции в языке Haskell состоит в том, что она является выражением. *Выражение* – это код, возвращающий значение. `5` – это выражение, потому что возвращает 5; `4 + 8` – выражение, `x + y` – тоже выражение, потому что оно возвращает сумму *x* и *y*.

Поскольку ветвь `else` обязательна, конструкция `if` всегда что-нибудь вернёт, ибо является выражением. Если бы мы хотели добавить единицу к любому значению, получившемуся в результате выполнения нашей предыдущей функции, то могли бы написать её тело вот так:

```
doubleSmallNumber' x = (if x > 100 then x else x*2) + 1
```

Если опустить скобки, то единица будет добавляться только при условии, что `x` не больше 100. Обратите внимание на символ апострофа (`'`) в конце имени функции. Он не имеет специального значения в языке Haskell. Это допустимый символ для использования в имени функции.

Обычно мы используем символ прямого апострофа () для обозначения строгой (не ленивой) версии функции либо слегка модифицированной версии функции или переменной. Поскольку апостроф – допустимый символ в именах функций, мы можем определять такие функции:

```
conanO'Brien = "Это я, Конан О'Брайен!"
```

Здесь следует обратить внимание на две важные особенности. Во-первых, в названии функции мы не пишем имя `conan` с прописной буквы. Дело в том, что наименования функций не могут начинаться с прописной буквы – чуть позже мы разберёмся, почему. Во-вторых, данная функция не принимает никаких параметров.

Когда функция не принимает аргументов, говорят, что это *константная* функция. Поскольку мы не можем изменить содержание имён (и функций) после того, как их определили, идентификатор

[4] Вообще говоря, конструкцию с `if` можно определить в виде функции:

```
if' :: Bool -> a -> a -> a
if' True x _ = x
if' False _ y = y
```

Конструкция введена в язык Haskell на уровне ключевого слова для того, чтобы минимизировать количество скобок в условных выражениях. – *Прим. ред.*

`conanO'Brien` и строка `"Это я, Конан О'Брайен!"` могут использоваться взаимозаменяемо.

Списки

Как и списки покупок в реальном мире, списки в языке Haskell очень полезны. В данном разделе мы рассмотрим основы работы со списками, генераторами списков и строками (которые также являются списками).

Списки в языке Haskell являются *гомогенными* структурами данных; это означает, что в них можно хранить элементы только одного типа. Можно иметь список целых или список символов, но нельзя получить список с целыми числами и символами одновременно.

Списки заключаются в квадратные скобки, а элементы разделяются запятыми:

```
ghci> let lostNumbers = [4,8,15,16,23,42]
ghci> lostNumbers
[4,8,15,16,23,42]
```

> **ПРИМЕЧАНИЕ.** *Можно использовать ключевое слово `let`, чтобы определить имя прямо в GHCi. Например, выполнение `let a = 1` из GHCi – эквивалент указания `a = 1` в скрипте с последующей загрузкой.*

Конкатенация

Объединение двух списков – стандартная задача. Она выполняется с помощью оператора ++[5].

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
ghci> "привет" ++ " " ++ "мир"
"привет мир"
```

[5] Следует отметить, что *операторами* называются двухместные инфиксные функции, имена которых состоят из служебных символов: `+`, `*`, `>>=` и т. д. – *Прим. ред.*

```
ghci> ['в','о'] ++ ['-'] ++  ['о','т']
"во-от"
```

ПРИМЕЧАНИЕ. *Строки в языке Haskell являются просто списками символов. Например, строка* `привет` *– это то же самое, что и список* `['п','р','и','в','е','т']`. *Благодаря этому для работы со строками можно использовать функции обработки символов, что очень удобно.*

Будьте осторожны при использовании оператора ++ с длинными строками. Если вы объединяете два списка (даже если в конец первого из них дописывается второй, состоящий из одного элемента, например `[1,2,3] ++ [4]`), то язык Haskell должен обойти весь список с левой стороны от ++. Это не проблема, когда обрабатываются небольшие списки, но добавление к списку из 50 000 000 элементов займет много времени. А вот если вы добавите что-нибудь в начало списка с помощью оператора : (также называемого «*cons*»), долго ждать не придётся.

```
ghci> 'В':"ОТ КОШКА"
"ВОТ КОШКА"
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

Обратите внимание, что оператор : принимает число и список чисел или символ и список символов, в то время как ++ принимает два списка. Даже если вы добавляете один элемент в конец списка с помощью оператора ++, следует заключить этот элемент в квадратные скобки, чтобы он стал списком:

```
ghci> [1,2,3,4] ++ [5]
[1,2,3,4,5]
```

Написать `[1,2,3,4] ++ 5` нельзя, потому что оба параметра оператора ++ должны быть списками, а `5` – это не список, а число.

Интересно, что `[1,2,3]` – это на самом деле синтаксический вариант `1:2:3:[]`. Список `[]` – пустой, и если мы добавим к его началу 3, получится `[3]`; если затем добавим в начало 2, получится `[2,3]` и т. д.

ПРИМЕЧАНИЕ. *Списки* `[]`, `[[]]` *и* `[[],[],[]]` *совершенно разные. Первый – это пустой список; второй – список, содержащий пустой список; третий – список, содержащий три пустых списка.*

Обращение к элементам списка

Если вы хотите извлечь элемент из списка по индексу, используйте оператор `!!`. Индексы начинаются с нуля.