Функциональное программирование

Денис Москвин

31 августа 2020 г.

Содержание

1 Сравнение императивного и функционального программирования

Джон Бэкус в 1977 году при вручении ему премии Тьюринга выступил с лекцией «Можно ли освободить программирование от стиля фон Неймана? Функциональный стиль и соответствующая алгебра программ»[2]. В этой лекции он провел сравнение вычислительных моделей языков программирования, выделив в качестве основных модели фон Неймана и аппликативные модели. В современных терминах это соответствует императивной и функциональной парадигмам программирования.

Многие критиковали этот текст за излишнюю полемичность, однако в нем присутствует целый ряд идей, до сих пор не потерявших актуальности. Сравним императивное и функциональное программирование, частично следуя наблюдениям Бэкуса.

При императивном программировании вычисление описывается в терминах *инструкций*, изменяющих состояние *вычислителя*. В императивных программах:

- инструкции исполняются *последовательно*: `C1; C2; C3`;
- состояние изменяется инструкциями *присваивания* значений *изменяемым переменным*: `v := value`;
- есть механизмы *условного исполнения*: инструкции `if, switch`;
- группы инструкций можно повторять с помощью *циклов*: инструкции `while, for`;
- типы данных описываются с оглядкой на их физическое представление в памяти.

Фактически этот подход к программированию базируется на копировании низкоуровневой архитектуры компьютера: изменяемые переменные имитируют ячейки памяти, а высокоуровневые инструкции представляют собой последовательности низкоуровневых.

Приведем пример программы, вычисляющей факториал числа для императивного, фон-неймановского языка

```
long factorial (int n) {
   long res = 1;
   for (int i = n; i > 0; i--)
      res = res * i;
   return res;
}
```

Выполнение такой программы представляет собой переход вычислителя из начального состояния в конечное с помощью содержащихся в ней последовательных инструкций. Повторяющиеся действия оформлены с использованием цикла. Часть конечного состояния (память, доступная по адресу, на который ссылается локальная переменная `res`) интерпретируется как результат вычислений.

Перейдем теперь к описанию модели вычислений для функциональной парадигмы. Функциональная программа рассматривается не как последовательность инструкций, а как *выражение*. Её выполнение — это вычисление (*редукция*) этого выражения. Редукция осуществляется пошагово, каждый шаг представляет собой использование определения функции: ее вхождение в выражение заменяется на ее тело.

Вот пример функции, вычисляющая факториал числа на функциональном языке

```
factorial n = if n == 0 then 1 else n * factorial (n - 1)
```

Слева от знака равенства находится имя функции `factorial` и имя ее формального параметра `n`. Справа находится *тело функции* — выражение, которое может содержать формальный параметр и, возможно, определяемую функцию. В последнем случае функцию называют *рекурсивной*; наш `factorial` именно таков.

Рассмотрим выражение `factorial 3` представляющее собой применение функции к аргументу 3. Это выражение, которое допускает редукцию. Редукция осуществляется посредством *подстановки* определений функций в местах их «вызова» с заменой формальных параметров на фактические. Вот цепочка преобразований (шагов редукции), которые осуществляет вычислитель в этом случае:

```
factorial 3
 → if 3 == 0 then 1 else 3 * factorial (3 - 1)
 → ...
 → 3 * factorial (3 - 1)
 → 3 * if (3 - 1) == 0 then 1 else (3 - 1) * factorial ((3 - 1) - 1)
 → ...
 → 3 * 2 * 1 * 1
 → ...
 → 6
```

В этом примере подразумевается, что помимо определения функции `factorial` имеются подобные определения для стандартных арифметических и логических операторов и конструкции `if...then...else...`. Скажем, на втором шаге как раз осуществляется подстановка определения последней конструкции, редукция в котором в свою очередь потребует редукции применения к своим аргументам оператора сравнения чисел не равенство в выражении `3 == 0`.

Приведенная реализация функциональной модели вычислений представляет собой переписывание текста программы по определенным правилам. Очевидно, что такой подход весьма неэффективен. Существует гораздо более эффективные реализации этой модели, например, вычисление в контекстах или редукция на графах[1].

«Функциональная» модель вычислений приводит к целому ряду весьма удивительных последствий. Действительно, в ней полностью отсутствует ключевая для императивного программирования идея изменяемого состояния! Отсутствие состояния приводит к отсутствию понятий *изменяемых* переменных и присваивания им значений. Переменные остаются, но лишь как имена, вместо присваиваний имеет место однократное *связывание* этих имен с выражениями.

Отсутствие изменяемого состояния приводит и к отсутствию традиционных циклов. Действительно, если изменения состояния не доступно программисту, то нет возможности различать итерации цикла и управлять выходом из него. Заменой циклов служит рекурсия при определении функций. Приведенный выше пример функции `factorial` иллюстрирует идею повторной подстановки одного и того же рекурсивного определения функции при изменяющемся значении аргумента.

Еще одной отличительной чертой функционального программирования служит работа по преимуществу с *чистыми* функциями, то есть такими возвращаемое значение которых зависит только от значений параметров. Хотя для взаимодействия с операционной системой и пользователем приходится иметь дело с вычислениями, обладающими побочными эффектами, функциональные языки стремятся к изоляции подобных действий теми или иными языковыми средствами.

Функциональное программирование ограничивает доступный программисту базовый инструментарий. Однако эти ограничения не бесполезны, они существенно облегчают формальный анализ программ и манипулирование ими с использованием формальных методов. Это, в частности, эффективное доказательство свойства программ и, с другой стороны, возможность алгебраически выводить программы из их спецификации (заранее заданных свойств). Следует также отметить доступность широкого спектра высокоуровневых оптимизаций, базирующейся на анализе эквивалентных преобразований программ.

Многие функциональные языки имеют мощную систему типов, которая вместе с тем необременительна, благодаря эффективным алгоритмам вывода типов. Стоит так же отметить регулярный и лаконичный синтаксис, характерный для этих языков.

2 Haskell: общие сведения и установка

Haskell — *чистый* функциональный язык программирования с «*ленивой*» семантикой и полиморфной *статической* типизацией. Язык назван в честь американского логика и математика Хаскелла Брукса Карри. Сайт языка https://www.haskell.org/ является авторитетным источником полезной и релевантной информации.

Первая реализация языка Haskell была выпущена в 1990 году. Первый стандарт языка появился в 1998. Текущий стандарт — Haskell Report 2010 [6]. Стандарт определяет многие языковые конструкции в терминах трансляции в так называемый Haskell Kernel — синтаксически простой, но многословный функциональный язык.

Неофициальный девиз языка:*Avoid Success at All Costs!*. Расстановка логических скобок при синтаксическом разборе этого девиза порождает два возможных прочтения: шуточное и вдохновляющее на свершения.

Хотя исторически имелось несколько реализаций языка Haskell, в настоящее время осталась единственная The Glasgow Haskell Compiler, сокращенно GHC [9]. Последние версии компилятора 8.6.5/8.8.4/8.10.2.

Компилятор GHC поставляется вместе с интерактивным окружением GHCi, позволяющем вычислять выражения и запускать программы в режиме интерпретатора.

Помимо этого в поставку GHC входят инструменты для отладки и профилирования, а так же набор полезных библиотек. Термин «стандартная библиотека» понимается в двух смыслах. В узком смысле имеются в виду библиотеки, описанные в Haskell Report. В широком смысле — библиотеки, поставляемые с GHC.

Рекомендуемый механизм установки зависит от операционной системы. Для Windows это Haskell Platform, для Linux и MacOS — ghcup. Для нашего курса настоятельно рекомендуется использовать версию GHC не древнее 8.0.

Имеются две конкурирующие системы упаковки библиотек в пакеты и их дистрибуция: Cabal и Stack. Соответствующие хранилища пакетов называются Hackage и Stackage. Самые популярные инструменты поиска по документации Hoogle и Hayoo.

В поставку GHC не входит редактор исходного кода. Вы можете пользоваться любым удобным для вас текстовым редактором. Желательно, конечно, чтобы он был предназначен для редактирования кода программ и имел подсветку синтаксиса для языка Haskell.

По возможности стоит выполнить еще одну полезную настройку редактора: автоматическую замену символа табуляции на пробелы. Дело в том, что в Haskell размер отступов имеет важное значение для группировки частей программ, а настройки вашего редактора относительно визуального сдвига при табуляции могут не совпадать с представлениями компилятора о размере этого сдвига. Это может привести к неожиданным ошибкам компиляции. Если вы все же хотите активно использовать символ табуляции, а ваш редактор не поддерживает его замену на пробелы, настоятельно рекомендуем самостоятельно изучить принятые в Haskell правила его трансляции.

Существует обширная литература по языку Haskell. В качестве простого базового учебника мы рекомендуем [8, 3]. Также полезными для изучения представляются книги [7, 4, 5].

Итак все готово, чтобы написать первую программу. Создаём с помощью текстового редактора файл `Hello.hs` содержащий:

```
main = putStrLn "Hello, world!"
```

Затем запускаем интерпретатор GHCi, загружаем файл и вызываем определенную в нем функцию `main`

```
$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :load Hello
[1 of 1] Compiling Main             ( Hello.hs, interpreted )
Ok, one module loaded.
*Main> main
Hello, world!
*Main>
```

`Prelude` — стандартный модуль языка Haskell, всегда подгружаемый по умолчанию. Он содержит определения стандартных типов данных и основных функций для работы с

ними.

3 Основы программирования на Haskell

3.1 Выражения

Базовыми «строительными кирпичиками» для выражений служат *литералы*: символьные (`'a'`, `'X'`,`'\''`); строковые (`"Hello, world!\n"`); целочисленные (`42`) и с плавающей точкой (`3.14159`). Сюда же можно отнести и булевы значения (`True` и `False`), хотя на самом деле они определены через пользовательский тип данных.

Библиотека Prelude предоставляет стандартные операции над стандартными типами данных, позволяющие строить выражения: функции, операторы и конструкторы данных. В свою очередь интерпретатор позволяет вычислять их значения.

```
GHCi> 13 * 3 + 3
42
GHCi>  "Hello " ++ 'w' : "orld!"
"Hello world!"
GHCi> not False
True
GHCi> exp 1
2.718281828459045
GHCi> reverse "Hello"
"olleH"
```

Три последних примера иллюстрируют тот факт, что при вызове функций скобки не требуются.

3.1.1 Кортежи и списки

Существуют встроенные контейнеры: кортежи и списки. Мы строим их значения, используя соответствующие *конструкторы данных*.

Кортежи (tuples) хранят значения произвольных типов, разделенные запятыми и заключенные в круглые скобки, например

```
(42,"Hello world!")

(True,'c',4)
```

Минимальный допустимый размер кортежа равен 2. Стандарт говорит, что максимальный размер должен быть не меньше 15, в GHC — 62. Кортежи разных размеров относятся к разным типам, никаких неявных преобразований между ними нет.

Списки (lists) отличаются от кортежей тем, что тип их элементов должен быть одним и тем же. Для списков используют прямоугольные скобки[1]

```
[1,2,3]

['H','e','l','l','o']
```

Длина списка неограниченна, в том числе имеются одноэлементные списки, например `[42]`, и пустой список `[]`. В отличие от кортежа, тип списка не зависит от его длины и полностью определяется типом элементов.

Отметим, что на самом деле стандартные строковые литералы являются синтаксическим сахаром для списка символов:

```
GHCi> ['H','e','l','l','o']
"Hello"
```

Такой тип строк, принятый в стандартной библиотеке, не очень эффективен. В прикладных программах часто используют более эффективную реализацию, доступную через модуль `Data.Text`.

3.2 Объявления: связывание

Выражения (expressions) являются инструментом программирования в малом масштабе. Более крупными строительными блоками программы служат *объявления* (declarations). Наиболее часто встречающимися объявлениями служат *связывания* (binding).

3.2.1 Связывание переменных

Связывания осуществляются с помощью знака равенства. Простейший тип связывания — это *связывание переменной*. Имя переменной слева от знака равенства связывается с выражением, стоящим справа

```
x = 42
aBC = let z = x + y in z ^ 2
y = 7 + 3
```

Связывание может быть *глобальным* и *локальным*. В приведенном примере переменные `x`, `y` и `aBC` связаны глобально, на верхнем уровне, а переменная `z` — локально.

Первый символ имени переменной должен быть в нижнем регистре. Верхний регистр первого символа зарезервирован за именами конструкторов типов и данных, а также за именами модулей.

[1] Отметим, что в отличие от кортежа, где круглые скобки с запятыми представляют собой настоящий конструктор данных, для списка синтаксис с квадратными скобками — всего лишь синтаксический сахар.

Отметим, что связанные переменные часто называют *константами*, поскольку их значение раз и навсегда задается значением выражения в правой части. Еще иногда про связывание переменных говорят, что оно задает функцию нуля аргументов (*нульарную функцию)*. А вот чего не стоит делать, это называть связывание переменной присваиванием значения. Это совершенно другой механизм.

В интерпретоаторе GHCi тоже допустимо связывание

```
GHCi> answer = 39 + 3
```

Отметим, что связывание само по себе не форсирует никаких вычислений. Чтобы запустить их в интерпретаторе нужно обратиться к связанному имени

```
GHCi> answer
42
```

3.2.2 Связывание образцов

Связывание переменной является частным случаем более общей конструкции — *связывания образцов* (pattern binding). Примеры иллюстрируют это понятие:

```
GHCi> (x,y) = ('A',21*2)
GHCi> [u,v,w] = reverse "Hi!"
```

Конструкция в левой части равенства называется *образцом* (pattern). Она представляет собой конструктор (двухэлементного кортежа в первом примере и трехэлементного списка во втором), в котором вместо значений подставлены связываемые переменные.

Если мы обращаемся к одной из связанных переменных, то выражение справа вычисляется, и, если оно имеет ту же структуру, что и образец, то возвращается соответствующее значение.

```
GHCi> x
'A'
GHCi> y
42
GHCi> w
'H'
```

Если же структура, полученная в результате вычислений, не соответствует образцу, происходит аварийное завершение вычислений при попытке их форсировать[2].

```
GHCi> [p,q,r] = reverse "Hello!"
GHCi> q
*** Exception: Non-exhaustive patterns in [p, q, r]
```

[2] Напомним, что сам факт связывания никаких вычислений не запускает.

Механизм сопоставления с образцом является одним из ключевых при программировании на Haskell, при изучении пользовательских типов данных мы будем говорить о нем более подробно.

3.2.3 Функциональное связывание

Равенство может задавать функцию, такое связывание называют *функциональным* (function binding). Вот пример объявления, определяющего функцию

```
foo x y = 10 * x + y
```

После переменной `foo`, именующей функцию, следуют переменные `x` и `y`, именующие формальные параметры этой функции. На них допустимо ссылаться в теле этой функции — выражении, стоящем справа от знака равенства.

В приведенном примере идентификатор `foo` связывается глобально, а `x` и `y` — локально.

Применение функции `foo` записывается так же, как и левая часть ее определения, при этом вместо формальных параметров подставляются фактические значения или выражения:

```
GHCi> foo 1 3
13
GHCi> fortyTwo = foo 2 22
GHCi> fortyTwo
42
```

В процессе вычисления выражения, представляющего собой применение функции `foo 1 3`, формальные параметры `x` и `y` связываются с фактически переданными выражениями `1` и `3` соответственно. О таком связывании говорят как о *динамическом*, в отличии от обсуждаемого здесь *статического* связывания. Обычно род связывания, статический или динамический, ясен из контекста.

Отметим, что ни при определении ни при применении функций не используются круглые скобки вокруг аргументов. Запятая в качестве разделителя между аргументами тоже не используется. То есть, например, выражение `a b c d` представляет собой вызов функции с именем `a` на трех аргументах `b`, `c` и `d`.

Круглые скобки используются для другой цели – группировки частей выражения. Из выражения `f (g x y) (h z)`, например, сразу можно сделать вывод, что функции `f` и `g` бинарные, а `h` — унарная[3].

Допустимо использовать лямбда-выражения для определения функций.

```
foo  x y = 10 * x + y          -- комбинАторный стиль
foo' x   = \y -> 10 * x + y    -- смешанный стиль
foo''    = \x y -> 10 * x + y -- лямбда-стиль
```

[3]Более точно: если приведенное выражение прошло проверку типов, то определения всех пяти идентификаторов доступны и их арность не меньше 2 для `f` и `g` и 1 для `h`.

Все три приведенные определения эквивалентны.

Функция `foo'` хорошо подходит для иллюстрации идеи *частичного применения*. Эту функцию можно рассматривать как функцию двух аргументов, возвращающую число, но так же и как функцию одного аргумента, возвращающую функцию одного аргумента. Поэтому ее можно применить к одному аргументу, следующее определение вполне валидно

```
bar' = foo' 2 -- = \y -> 10 * 2 + y = \y -> 20 + y
```

Поскольку `bar'` — функция одного аргумента, ее можно вызвать

```
GHCi> bar' 22
42
```

Частичное применение работает и для двух других версий `foo`:

```
bar''  = foo'' 2
bar    = foo 2
```

с тем же самым результатом

```
GHCi> bar'' 22
42
GHCi> bar 22
42
```

Теперь понятно, что связывание переменной

```
fortyTwo' = (foo 2) 22
```

полностью эквивалентно `fortyTwo = foo 2 22`. Правые части этих двух связываний неразличимы. Про этот факт иногда говорят, что применение функции к аргументам ассоциативно влево: `f x y z` — просто сокращенная запись для `((f x) y) z`.

Функции нескольких аргументов, допускающие передачу этих аргументов последовательно, по одному, называют *каррированными* в честь Хаскелла Карри. В Haskell все функции таковы, что не удивительно, если вспомнить, в чью честь этот язык назван.

Каждая из лямбд в приведенных выше `foo'` и `foo''` задает так называемую *лексическую область видимости* для связываемой переменной. Глобальная область видимости одна, ее еще называют областью видимости уровня модуля. Локальных областей видимостей может быть несколько, они могут задаваться не только лямбдами, но и другими конструкциями языка.

В одной лексической области видимости связывание некоторой переменной может происходить не более одного раза.

```
z = 1            -- ok, связали
z = 2            -- ошибка, повторное глобальное связывание
q q = \q -> q    -- ok, но...
```

В последнем примере переменная q связывается трижды: как имя функции в глобальной области видимости, как имя параметра этой функции и как аргумент лямбды. Эти три области видимости вложены друг в друга. Переменная q, составляющая тело функции, связывается с аргументом лямбды, про остальные связывания говорят, что они «затенены» (shadowed). Можно проверить это в интерпретаторе

```
GHCi> q 13 42
42
```

Отметим, что первые два из следующих определений недопустимы, поскольку в них в одной и той же области видимости происходит попытка использовать для связывания одно и то же имя:

```
p p p = p              -- ошибка
p = \p p -> p          -- ошибка
p = \p -> (\p -> p)    -- ok
```

Следует отметить, что в интерпретаторе повторное связывание одной переменной на глобальном уровне допустимо:

```
GHCi> z = "Hello"
GHCi> z = 2
GHCi> z
2
```

При определении функции мы можем в качестве формальных параметров использовать не только переменные, но и любые образцы. Например, следующие функции просто невозможно было бы написать без этого механизма:

```
fst (x,y) = x
snd (x,y) = y
```

Эти функции весьма полезны и, поэтому, определены в стандартной библиотеке. При вызове происходит подстановка фактических значений вместо формальных параметров-переменных, использованных внутри образца.

```
GHCi> fst ("Hello",2128506)
"Hello"
GHCi> snd ("Hello",2128506)
2128506
```

Допускается вложенность образцов произвольной глубины:

```
GHCi> fstOfSnd (x,(y,z)) = y
GHCi> fstOfSnd ('z',(33,True))
33
```

Определение функции может быть дано в так называемом *бесточечный* (pointfree) стиле. Ниже приведены три способа определения логарифма по основанию 2 через библиотечную `logBase`:

```
lg x = logBase 2 x          -- комбинаторное определение

lg'  = \x -> logBase 2 x    -- определения через лямбды

lg'' = logBase 2            -- определения в бесточечном стиле
```

Последний способ, в котором имя связывается с частично примененной функцией, называется бесточечным, поскольку в определении отсутствует точка применения функции — ее аргумент. Отметим, что переход от второго равенства к третьему — ни что иное, как известная в λ-исчислении η-редукция.

3.3 Отступы

Часто программа оказывается довольно длинной и не помещается в одну строку, приходится переносить ее на следующие строки. Поскольку программа на Haskell состоит из объявлений, полезно договориться о том, где то место, в котором текущее объявление закончилось и началось следующее. В Haskell это делается с помощью *отступов*, которые отсчитываются в пробелах. Принято следующее правило: ***при переносе кода объявления на следующую строку отступ должен быть больше, чем отступ начала этого объявления***.

Если при переносе отступ оказался равен отступу начала объявления — значит началось новое объявление той же самой области видимости. Если отступ оказался меньше отступа начала объявления — то это начало объявления более широкой области видимости[4]. Естественно, что, если в месте такого переноса новое объявление недопустимо, то мы получим ошибку компиляции. Подробности см. Haslell 2010 Report[6], описание в разделе 2.7 и точную трансляцию в разделе разделе 10.3.

Вот, например, функции, возвращающие пару корней квадратного уравнения $ax^2 + bx + c = 0$ и их число:

```
roots a b c =
  (               -- начало пары
    (- b - sqrt (b ^ 2 - 4 * a * c)) / (2 * a), -- первый элемент
    (- b + sqrt (b ^ 2 - 4 * a * c)) / (2 * a)  -- второй элемент
```

[4]Мы столкнемся с такими ситуациями, когда будем изучать конструкции `where...`, `let...in...`, `case...of...` и `do...`.

```
  )               -- конец пары
nRoots a b c =    -- начало нового (глобального) объявления
  if b ^ 2 - 4 * a * c > 0
  then 2
  else if b ^ 2 - 4 * a * c == 0
  then 1
  else 0
```

Обратим внимание, что отступы всех строк, составляющих тело функций, могут увеличиваться и уменьшаться произвольным образом. Главное, чтобы они были больше нуля: первое объявление начинается идентификатором `roots` с нулевым отступом; второе начинается, когда отступ вновь становится нулевым.

Смысл приведенных выше функций не требует особых пояснений. Приведем, тем не менее, пример: для $x^2 - x - 6 = 0$ получим

```
GHCi> roots 1 (-1) (-6)
(-2.0,3.0)
GHCi> nRoots 1 (-5) 6
2
```

3.4 Рекурсия

Основным инструментом кодирования повторяющихся вычислений в функциональных языках служит рекурсия. Рекурсивное определение содержит имя определяемой функции в ее теле. Корректная реализация рекурсии должна содержать достижимое терминирующее условие.

```
factorial0 n = if n > 1
               then n * factorial0 (n - 1)
               else 1

factorial1 n = if n == 0
               then 1
               else n * factorial1 (n - 1)
```

При сравнении поведения двух этих реализации на отрицательных входных данных мы видим отличие в поведении вне области определения факториала. Первая версия говорит нам, что факториал любого отрицательного числа равен 1, вторая зависает. Первая дает нам частично неправильные результаты, вторая ведет себя крайне плохо на некорректных входных данных.

Прежде чем обсудить возможные исправления этой проблемы, следует познакомится еще с одним понятием.

3.5 Ошибки времени исполнения

В Haskell имеется специальное значение $\bot$ (*основание*, *дно*, bottom), маркирующее ошибку времени исполнения. Это либо «зависаюшее» (никогда не завершающееся или *расходящееся*) вычисление, либо аварийно завершившееся. Рекурсия позволяет легко закодировать «зависающее» вычисление

```
bot = 1 + bot
```

Более того, так будет вести себя любая рекурсия, если мы не озаботились наличием в ней терминирующего условия и его достижимостью при вычислении. Имейте это в виду, когда реализуете рекурсивные функции: рекурсия мощный, но опасный инструмент!

Библиотечная константа `undefined` представляет собой другой пример «реализации» $\bot$. При любой попытке ее вычислить происходит немедленное аварийное завершение программы с выводом сообщения `"Prelude.undefined"` в диагностический поток:

```
GHCi> undefined
*** Exception: Prelude.undefined
```

В традиционных языках программирования особой пользы от такой константы нет, однако позже мы поймем, почему ее включили в стандартную библиотеку Haskell.

«Зависаюшее» вычисление, вроде функции `bot`, не выполняет никакой видимой полезной работы. Однако Haskell позволяет определять никогда не завершающиеся вычисления, возвращающие полезные частичные результаты. Например,

```
fortyTwos = 42 : fortyTwos
```

на первый взгляд устроена так же как `bot`. Оператор `(:)` добавляет значение, переданное ему в качестве левого аргумента, в голову списка, переданного в качестве правого:

```
GHCi> 3 : [4,5,6]
[3,4,5,6]
```

Так же как `bot` описывает вычисление, стремящееся получить результат неограниченным прибавлением 1, `fortyTwos` задает расходящееся вычисление, конструирующее бесконечный список из чисел 42. Однако, в отличие от `bot`, вызов которой в GHCi ничего не выводит,

```
GHCi> bot
Interrupted.
```

вызов `fortyTwos` демонстрирует доступность промежуточных результатов:

```
GHCi> fortyTwos
[42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,
42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,4
2,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42
,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,
42,42,42,42,42,42,42,42,42,42,42,42,42,42,42,Interrupted.
```

Haskell обеспечивает возможность воспользоваться этой доступностью: в случае подобной *продуктивной* расходимости мы можем остановить неограниченные рекурсивные вызовы способами, внешними по отношению к рекурсивно определенной функции:

```
GHCi> take 5 fortyTwos
[42,42,42,42,42]
```

Точный механизм техники, позволяющей реализовывать функции, подобные `take`, мы обсудим позже.

Теперь мы можем написать более «аккуратную» версию факториала

```
factorial n =
  if n < 0
  then error "factorial: negative argument"
  else if n > 1
       then n * factorial (n-1)
       else 1
```

Использованная здесь библиотечная функция `error` — это гибкая версия `undefined` с настраеваемым сообщением об ошибке:

```
GHCi> factorial (-3)
*** Exception: factorial: negative argument
```

Еще более аккуратным был бы подход с перехватываемыми исключениями, позже мы его изучим.

3.6 Ленивые и энергичные вычисления

Мы уже говорили, что ФП характеризуется подстановочной моделью вычислений. Вычислитель находит в выражении вхождения вызовов функций, снабженные достаточным[5] количеством аргументов, и заменяет их на тело этой функции из ее определения, подставляя при этом фактические аргументы вместо формальных. Подобные вхождения называют *редексами* (redex, от reducible expression), процесс их вычисления (редукции) — сокращением.

[5] То есть соответствующим арности этой функции.

Возникает вопрос, что делать пошаговому вычислителю, если в вычисляемом выражении таких редексов несколько? Например, для нашей `foo x y = 10 * x + y`, выражение `foo (foo 1 2) 3` может быть редуцировано двумя разными способами

```
foo (foo 1 2) 3
 ⟶ 10 * (foo 1 2) + 3
 ⟶ 10 * (10 * 1 + 2) + 3
 ⟶ ...
 ⟶ 123
```

```
foo (foo 1 2) 3
 ⟶ foo (10 * 1 + 2) 3
 ⟶ ...
 ⟶ foo 12 3
 ⟶ 10 * 12 + 3
 ⟶ ...
 ⟶ 123
```

В первом случае происходит сокращение внешнего редекса, во втором — внутреннего. Первый способ соответствует *ленивой* (lazy) модели вычислений: аргументы передаются в функцию без предварительного вычисления. Именно этот способ принят в Haskell. Альтернативная, более традиционная модель называется *энергичной* (eager). Она принята, например, в OCaml и многих других функциональных языках. Иногда вместо терминов энергичный/ленивый используют понятия строгий/нестрогий.

Наша подстановочная вычислительная модель обладает замечательным свойством: результат вычислений не зависит от порядка сокращений редексов. Это свойство называют *конфлюэнтностью*. Иногда используют термины *свойство ромба* или *черч-россеровость*. Это очень важное свойство, поскольку позволяет оптимизатору переключаться между разными стратегиями вычислений.

В стандартной библиотеке есть очень простая функция двух аргументов, которая возвращает значение своего первого аргумента, а второй полностью игнорирует:

```
const x y = x
```

Ленивая модель вычислений позволяет элиминировать расходимости: если результат расходящегося вычисления не нужен, это вычисление не вызывается. Вот пример совместной работы `undefined` и `const`

```
GHCi> const 42 undefined
42
```

С помощью оператора `($!)` можно перейти от ленивой модели вычислений к энергичной. При этом легко обнаружить, что расходимость перестает элиминироваться:

```
GHCi> const 42 $! undefined
*** Exception: Prelude.undefined
```

По этой причине понятие конфлюэнтности нуждается в уточнении. Впрочем это уточнение довольно естественно: если обе стратегии не расходятся, то результат всегда один.

3.7 Аккумуляторы

Полезной техникой программирования является обеспечение рекурсивной функции дополнительным параметром, который используется для передачи между шагами редукции промежуточных значений. Такие параметры называют *аккумуляторами*, поскольку их часто используют для накопления результатов вычислений.

Вот например реализация факториала, использующая аккумулятор во вспомогательной рекурсивной функции с дополнительным аргументом:

```
factorial' n = helper 1 n
helper acc n = if n > 1
               then helper (acc * n) (n - 1)
               else acc
```

Второй параметр функции `helper` используется для контроля за наступлением терминирующего условия, а в первом накапливается искомое произведение.

Разумное использование аккумулятором часто позволяет сохранить линейную по числу рекурсивных вызовов сложность, даже в тех случаях, когда прямая реализация алгоритма ведет себя экспоненциально.

3.8 `where`, `let`, предохранители

В Haskell'е имеются вспомогательные инструменты, обеспечивающие возможность удобного локального связывания подвыражений с именами. Это конструкция `where...` и выражение `let...in...`.

Конструкция `where` следует за выражением, в котором предполагается использование новых идентификаторов. Вот, например, версия функции `roots`, в которой общие подвыражения вынесены и поименованы:

```
roots' a b c = ((- b - sd) / denom, (- b + sd) / denom)
  where {sd = sqrt discr; discr = b ^ 2 - 4 * a * c; denom = 2 * a}
```

Вместо синтаксиса с фигурными скобками и разделителями в виде точки с запятой можно использовать *правило отступа* (layout rule):

```
roots'' a b c = ((- b - sd) / denom, (- b + sd) / denom)
  where sd = sqrt discr
        discr = b ^ 2 - 4 * a * c
        denom = 2 * a
```

Главным требованием правила отступа служит одинаковый отступ для начала каждого объявления, задающего локальное связывание.

Конструкция `where...` допускает связывание не только констант, но и полноценных функций. Следующая реализация факториала с аккумулятором позволяет убрать вспомогательную функцию из глобальной области видимости

```
factorial'' m = helper 1 m where
  helper acc n =
    if n > 1
    then helper (acc * n) (n - 1)
    else acc
```

Выражение `let...in...` отличается от `where...` порядком следования блоков, в которых новые имена связываются и используются.

```
roots''' a b c =
  let sd = sqrt discr
      discr = b ^ 2 - 4 * a * c
      denom = 2 * a
  in ((- b - sd) / denom, (- b + sd) / denom)

factorial''' m =
  let helper acc n =
        if n > 1
        then helper (acc * n) (n - 1)
        else acc
  in helper 1 m
```

Имеется удобная синтаксическая конструкция, позволяющая выбирать тело функции на основе анализа логических условий. Это так называемые *охранные выражения* или *предохранители* (guards). Предохранители просматриваются сверху вниз до первого истинного.

```
factorial'''' m = helper 1 m where
  helper acc n
    | n > 1     = helper (acc * n) (n - 1)
    | otherwise = acc

factorial''''' m =
  let helper acc n
        | n > 1     = helper (acc * n) (n - 1)
        | otherwise = acc
  in helper 1 m
```

Если все предохранители принимают значение ложь, то программа аварийно завершается. Поэтому рекомендуется иметь последний предохранитель безусловно истинным; для этого используют константу `otherwise`.

Конструкция `where...` может быть общей для предохранителей

```
nRoots' a b c
  | d > 0  = 2
  | d == 0 = 1
  | d < 0  = 0
  where d = b ^ 2 - 4 * a * c
```

Это делает `where...` специальной частью синтаксиса объявления функционального связывания. За это мы платим невозможностью использовать конструкцию `where...` в качестве полноценного выражения, в отличие от `let...in...`:

```
GHCi> sqrt (let x = 5 in 9 * x + 4)
7.0
GHCi> sqrt (9 * x + 4 where x = 5)
<interactive>: error: parse error on input `where'
```

3.9 Система модулей

Мы уже обсуждали структуру программ на Haskell. На самом «мелком» уровне мы используем выражения; более крупными строительными блоками служат объявления; и, наконец, на самом верхнем уровне программа состоит из набора *модулей*.

Для объявления модуля мы используем ключевое слово `module`, после которого идет имя модуля (начинающееся с символа в верхнем регистре):

```
module Lesson1 where {
bot = 1 + bot;          -- variable binding
(x,y) = ('A',21*2);     -- pattern binding, x and y are global
foo x y = 10 * x + y;   -- function binding, x and y are local
}
```

Модуль представляет собой множество объявлений. На практике все используют эквивалентный синтаксис с отступами:

```
module Lesson1 where
bot = 1 + bot           -- variable binding
(x,y) = ('A',21*2)      -- pattern binding, x and y are global
foo x y = 10 * x + y    -- function binding, x and y are local
```

Отметим, что обычно для объявлений в глобальной области видимости используют нулевые отступы, хотя по стандарту это не обязательно.

GHC требует, чтобы каждый модуль был реализован в отдельном файле, и имя файла совпадало с именем модуля[6], хотя стандарт не накладывает таких ограничений, оставляя это во власти разработчиков компилятора.

Модули позволяют задавать пространствами имён и обеспечивают инкапсуляцию через списки экспорта и импорта. Если мы хотим использовать в модуле `A` имена из модуля `B` мы должны импортировать этот модуль:

```
module A where
import B
foo = qux quux
bar = ...
baz = ...
```

Объявления импорта должны быть первыми объявлениями модуля. В этом примере подразумевается, что `qux` и `quux` экспортируются из модуля `B`. По умолчанию экспортируются все имена, определенные в модуле глобально, на верхнем уровне. В частности, если кто-то будет импортировать наш модуль `A`, то ему станут доступны `foo`, `bar` и `bas`. Такое поведение может быть изменено с помощью *списка экспорта* (export list):

```
module A (foo, bar) where
import B
foo = qux quux
bar = ...
baz = ...
```

Здесь мы явно задали публичный интерфейс нашего модуля, перечислив список `(foo, bar)` экспортируемых из него имен. Функция `baz` при этом не экспортируется, оставаясь деталью реализации.

Существует также *списка импорта* (import list). Импортируя модуль, мы явно можем указать в скобках все те имена из него, которые нам потребуются:

```
module A where
import B (qux,quux)
foo = qux quux
bar = ...
baz = ...
```

Указание исчерпывающего списка импорта считается хорошим стилем программирования, поскольку оно позволяет не засорять пространство доступных имен и ускоряет компиляцию.

Одной из проблем при программировании служит ситуация *конфликта имен*: одно и то же имя может быть определено в разных модулях. Если мы импортируем оба, то возникает неоднозначность:

[6] Имеются некоторые незначительные послабления.

```
GHCi> map (\x -> x + 2) [1,2,3]
[3,4,5]
GHCi> import Data.Map
GHCi> map (\x -> x + 2) [1,2,3]
<interactive>: error:
    Ambiguous occurrence `map'
    It could refer to either `Prelude.map',
                             or `Data.Map.map',
```

Конфликты имён разрешаются через полные имена: они включают не только имя функции, но и имя модуля, в котором эта функция объявлена: `Prelude.map` и `Data.Map.map` уже не конфликтуют. Разделителем служит точка, пробелы вокруг нее не допускаются.

```
GHCi> Prelude.map (\x -> x + 2) [1,2,3]
[3,4,5]
```

Если подобный конфликт имен является массовым, можно использовать *квалифицированный импорт*:

```
import qualified B (qux,quux)
foo = B.qux B.quux
```

Теперь мы застрахованы от конфликтов имен, поскольку не можем использовать имена `qux` и `quux` из модуля `B` без явного указания имени модуля. Такое («неквалифицированное») использование приведет к ошибке компиляции независимо от того, возникает конфликт имен или нет.

Часто при импорте модулей с длинными именами удобно использовать *псевдонимы*:

```
GHCi> import Data.Set as S
GHCi> S.map (\x -> x + 2) (S.fromList [1,2,3])
fromList [3,4,5]
```

Во многих примерах мы импортировали модули в интерпретаторе GHCi, однако точно те же правила действуют при импорте в файлах исходного кода.

3.10 Операторы и сечения

Оператор в Haskell — это комбинация одного или более символов из следующего списка

```
! # $ % & * + . / < > ? @ ^ | - ~      = \        :
```

Все операторы *бинарные* и *инфиксные*, то есть выражение `(u <*?*> v)` представляет собой применение оператора `<*?*>` к двум аргументам `u` и `v`.

Исключением служит унарный префиксный минус, который всегда ссылается на `Prelude.negate`, то есть выражение `(- x)` транслируется в `negate x`. Еще одним специальным случаем служит символ `:` (двоеточие). Если он является первым символом оператора, то такой оператор должен быть инфиксным конструктором данных и объявляться как часть определения типа данных.

В отличие от ряда других языков, где набор операторов фиксирован, в Haskell их можно задать сколь угодно много.

Определение оператора похоже на определение функции, но идентификатор оператора в левой части определения используется инфиксно. Например, определив оператор `*+*` для суммы квадратов

```
a *+* b = a ^ 2 + b ^ 2
```

мы можем использовать его

```
GHCi> 3 *+* 4
25
```

Операторы могут определяться и использоваться в префиксном (функциональном) стиле. Для этого имя оператора нужно заключить в круглые скобки. Например, определим префиксно оператор для суммы кубов

```
(*+*+*) a b = a ^ 3 + b ^ 3
```

Мы можем его (как и любой другой оператор) использовать и в инфиксном и в префиксном стиле

```
GHCi> (*+*+*) 2 3
35
GHCi> 2 *+*+* 3
35
GHCi> (*+*) 4 3
25
```

Встроенные операторы, конечно же, не исключение

```
GHCi> 4 + 3
7
GHCi> (+) 4 3
7
GHCi> (-) 4 3
1
```

Функции, в свою очередь, могут определяться и использоваться в инфиксном (операторном) стиле. Для этого мы должны заключить имя функции в обратные кавычки:

```
x `plusminus` y = (x + y, x - y)
```

Любую функцию можно вызывать как в обычном функциональном так и операторном стиле, независимо от способа ее определения:

```
GHCi> plusminus 4 3
(7,1)
GHCi> 4 `plusminus` 3
(7,1)
GHCi> elem 3 [1,2,3]
True
GHCi> 5 `elem` [1,2,3]
False
```

3.10.1 Приоритет и ассоциативность

Программирование с использованием операторов требует определенных договоренностей, касающихся их *приоритета* и *ассоциативности*. Действительно, знание, что результат вычисления выражения

```
1 + 2 * 3
```

равен 7, следует из предварительной договоренности, что умножение имеет более высокий приоритет нежели сложение. Если бы приоритеты были бы противоположными, результат равнялся бы 9.

Есть еще одна проблема: как поступать с конструкциями, в которых операторы имеют одинаковый приоритет, например

```
1 - 3 - 2

2 ^ 3 ^ 2
```

В первом случае стандартная среди математиков договоренность об ассоциативности заключается в том, что оператор ассоциативен влево, а во втором — вправо:

```
1 - 3 - 2 == (1 - 3) - 2 -- == -4

2 ^ 3 ^ 2 == 2 ^ (3 ^ 2) -- == 512
```

В Haskell приоритет и ассоциативность задают с помощью *объявлений фиксности* (fixity declaration) `infixl`, `infixr` или `infix`. Например:

```
infixl 6 +++, *****
```

Теперь введённые нами операторы левоассоциативны и имеют тот же приоритет (равный 6), что и обычный оператор сложения.

Объявление `infix` используется, когда мы не хотим неявно подразумевать скобки в цепочках операторов. Например, для операторов сравнения ассоциативность не задана:

```
GHCi> True > False
True
GHCi> True > False > False
error: Precedence parsing error
     cannot mix `>' [infix 4] and `>' [infix 4] in the same infix expression
```

Диапазон изменения приоритетов — от 0 до 9, чем выше приоритет оператора, тем «теснее» он связывает свои аргументы. Если приоритет оператора не задан, то по умолчанию считается, что он `infixl 9`.

Попробуйте, имея в виду данные выше объявления, расставить скобки и вычислить выражения

```
1 *+* 2 + 3

3 + 1 *+* 2 * 3
```

Функциям тоже можно задавать приоритет (и ассоциативность), это делается с помощью тех же объявлений. Например,

```
infix 5 `plusminus`
```

Поскольку приоритет 5 ниже приоритета сложения (6) и умножения (7), в следующем примере арифметические операции связываются сильнее, чем от `` `plusminus` ``:

```
GHCi> 5 + 3 `plusminus` 6 * 2
(20,-4)
```

3.10.2 Стандартные библиотечные операторы

В Haskell довольно большое число операторов описано в стандарте языка; еще большее их число реализовано в стандартных библиотеках, поставляющихся с GHC.

Ниже приведены приоритет операторов и функций из Haskell Report.

```
infixl 9  !!
infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`
```

```
infixl 6  +, -
infixr 5  ++, :
infix  4  ==, /=, <, <=, >=, >, `elem`, `notElem`
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, `seq`
```

Если вы хотите выяснить приоритет какого-то оператора, то это можно сделать в GHCi, набрав `:info`

```
GHCi> :info &&
(&&) :: Bool -> Bool -> Bool    -- Defined in `GHC.Classes'
infixr 3 &&
```

Обычное применение функции к аргументу `f x` тоже можно рассматривать как оператор, обозначаемый символом пробел. Считается, что этот оператор имеет наивысший (10) приоритет. То есть применение связывает аргументы сильнее любого оператора, что позволяет однозначно разбирать конструкции вида

```
f x y <*?*> g z
```

независимо от приоритета оператора `<*?*>`.

Оператор `($)` задаёт применение функции к аргументу, фактически делая явным упомянутый выше «оператор» пробел:

```
infixr 0 $
f $ x = f x
```

На первый взгляд его смысл не очевиден. Действительно, зачем заменять пробел на другой значок? Однако тот факт, что его приоритет установлен наименьшим из возможных, позволяет использовать его для элиминации избыточных скобок[7]

```
f (g x) ≡ f $ g x

f (g x (h y)) ≡ f $ g x (h y) ≡ f $ g x $ h y
```

Из второго примера ясна причина правоассоциативности этого оператора. Также оператор `($)` используют для передачи применения в функции высших порядков.

Еще одним важным и широко используемым оператором в Haskell является оператор композиции функций `(.)`

[7]Метаоператор ≡ мы будем использовать для обозначения операционной эквивалентности двух выражений: два выражения равны, если их значения совпадают. Если выражения представляют собой функции, то речь идет о поточечном равенстве: две функции равны, если на любых наборах переданных им аргументов результат вычислений один и тот же.

```
infixr 9  .
f . g = \ x -> f (g x)
```

Этот оператор принимает две функции, и возвращает функцию, получающуюся последовательным применением функций-аргументов, иными словами, их композицию. Например, выражение `(^2) . (+5)` — это функция, прибавляющая 5 к своему аргументу, а затем возводящая результат в квадрат:

```
GHCi> (^2) . (+5) $ 1
36
GHCi> (^2) . (+5) $ 2
49
```

3.10.3 Сечения операторов

Операторы на самом деле просто функции и, поэтому, допускают частичное применение. Есть дополнительный синтаксический сахар, обеспечивающий возможность записать частичное применение оператора как к левому, так и к правому своему аргументу.

Левое сечение (left section) оператора `(**+)` записывается так `(2 **+)` и эквивалентно обычному частичному применению этого оператора, приведенного к функциональному стилю:

```
(2 **+) ≡ (**+) 2 ≡ \y -> 2 **+ y
```

Правое сечение (right section) связывает правый аргумент оператора, то есть второй, если перевести оператор в функциональный стиль. Записать это в виде частичного применения невозможно, в виде лямбды это выглядит так

```
(**+ 3) ≡ \x -> x **+ 3
```

Скобки являются неотъемлемой частью синтаксиса сечений, их наличие обязательно.

3.10.4 Сечения кортежей

Язык Haskell определен в стандарте Haskell Report. Однако имеется множество полезных расширений, не описанных в стандарте, но активно используемых в GHC. Среди них возможность задавать *сечения кортежей* (tuple sections). Название этого расширения `TupleSections`. Для того, чтобы использовать его в интерпретаторе, нужно подключить его с помощью команды GHCi `set`, добавив `-X` перед именем подключаемого расширения

```
GHCi> :set -XTupleSections
GHCi> pairSection = (,'x')
GHCi> pairSection "Hello"
```

```
("Hello",'x')
GHCi> pairSection 42
(42,'x')
GHCi> tripleSection = ("ABC",,33)
GHCi> tripleSection True
("ABC",True,33)
```

Если мы хотим использовать сечения кортежей не только в интерпретаторе, но и в файле исходного кода, то нужно поместить прагму в самое начало файла, до ключевого слова `module`

```
{-# LANGUAGE TupleSections #-}
module MyModule where
...
```

4 Базовые типы

До сих пор мы пользовались языком Haskell как совершенно бестиповым. Однако на самом деле он имеет мощную статическую систему типов. В то же время эта система совершенно ненавязчива, Haskell умеет выводить типы всех выражений и связанных в объявлениях переменных безо всякого участия программиста. При этом если выражение построено с нарушением правил типизации, система проверки типов (тайпчекер) сообщит об ошибке:

```
GHCi> not 'z'
<interactive>: error:
    * Couldn't match expected type `Bool' with actual type `Char'
    * In the first argument of `not', namely 'z'
      In the expression: not 'z'
      In an equation for `it': it = not 'z'
```

Тайпчекер здесь сообщает нам, что использование функции `not` подразумевает передачу булева значения в качестве аргумента, а фактически осуществляется попытка передачи символа. Поскольку выражение не прошло тайпчекер, система времени исполнения отказывается вычислять его значение.

Типы языка Haskell образуются при помощи конструкторов из базовых встроенных типов. Вот эти базовые типы:

- `Bool` — булево значение;
- `Char` — символ Юникода;
- `Int`, `Integer` — целые числа фиксированной[8] и неограниченной разрядности;

[8] Для GHC совпадает с разрядностью платформы; на современных компьютерах 64 бита.

- `Float`, `Double` — числа с плавающей точкой одинарной и двойной точности;
- `type1 -> type2` — тип функции с аргументом типа `type1` и возвращаемым значением типа `type2`;
- `(type1, type2, ..., typeN)` — тип кортежа, `N > 1`;
- `()` — единичный тип, с одной константой `()`;
- `[type1]` — тип списка с элементами типа `type1`.

Имя конкретного типа, как встроенного так и пользовательского, должно начинаться с символа в верхнем регистре[9]. Оператор `(::)` используется для связывания выражения и его типа. В GHCi для выяснения типа выражения используют команду `:type`[10].

```
GHCi> :type True
True :: Bool
GHCi> :type 'z'
'z' :: Char
GHCi> :type "Hello"
"Hello" :: [Char]
GHCi> :type ("Hello",False)
("Hello",False) :: ([Char], Bool)
GHCi> :type not
not :: Bool -> Bool
```

Теперь видна причина, по которой тайпчекер отверг вызов `not 'z'`: функция ожидает аргумента типа `Bool`, а передается аргумент типа `Char`. Именно об этом и говорится в сообщении об ошибке.

Хотя Haskell умеет выводить типы, программист может явно указывать желаемый тип выражения

```
GHCi> :t (sin pi :: Double)
(sin pi :: Double) :: Double
GHCi> :t (42 :: Integer)
(42 :: Integer) :: Integer
GHCi> :t (42 :: Double)
(42 :: Double) :: Double
```

Конечно мы не можем указывать здесь произвольные типы: вызов

[9] Чуть позже мы познакомимся с переменными типа, для которых регистр первого символа такой же как и для обычных переменных — нижний.

[10] Эту команду интерпретатора можно сокращать до первого символа, набирая `:t`. Это верно и для ряда других широко используемых команд.

```
GHCi> :t (42 :: Char)
```

приведет к ошибке. Несколько позже мы обсудим механизм, обеспечивающий полиморфное поведение числовых литералов.

Типы списка, кортежа и функции можно записывать не только в определенной выше миксфиксной[11] и инфиксной нотации, но и в префиксной

```
GHCi> [1,2,3] :: [Double]
[1.0,2.0,3.0]
GHCi> [1,2,3] :: [] Double
[1.0,2.0,3.0]

GHCi> (1,'z',True) :: (Int,Char,Bool)
(1,'z',True)
GHCi> (1,'z',True) :: (,,) Int Char Bool
(1,'z',True)
GHCi> (,,) 1 'z' True :: (,,) Int Char Bool
(1,'z',True)
GHCi> :t (,,) 1 'z' True :: (,,) Int Char Bool
(,,) 1 'z' True :: (,,) Int Char Bool :: (Int, Char, Bool)

GHCi> :t (\x -> x) :: Char -> Char
(\x -> x) :: Char -> Char :: Char -> Char
GHCi> :t (\x -> x) :: (->) Char Char
(\x -> x) :: (->) Char Char :: Char -> Char
```

Типы, записанные в префиксной нотации, являются полными синонимами обычных инфиксных и миксфиксных вариантов. Префиксная нотация полезна, поскольку позволяет применять для встроенных типов многие механизмы доступные для пользовательских, которые определяются префиксно. Например, с помощью префиксной нотации удобно задавать частичное применение типов.

4.1 Устройство и использование типов

Основным способом объявлять *пользовательские типы данных* (user-defined datatypes declaration) является объявление `data`. Например, стандартный булев тип представляет собой перечисление (enumeration), определяемое следующим образом

```
data Bool = True | False
```

Присутствующее в левой части равенства идентификатор `Bool` называется *конструктором типа*, а присутствующие справа `True` и `False` называются *конструкторами дан-*

[11] К миксфиксной относят любые операторные нотации, выходящие за рамки стандартной классификации префиксный-инфиксный-постфиксный.

ных. Имена пользовательских конструкторов должны начинаться с символа в верхнем регистре. Вертикальная черта

Каковы методы работы с подобными типами данных? При определении функции мы можем в качестве формальных аргументов использовать не только переменные, но и образцы. Мы видели это раньше на примере функций над встроенным типом кортежа, например `fst (x,y) = x`.

На самом деле *образец* с технической точки зрения — это конструктор данных, расположенный не в правой части связывания, а в левой. Поскольку у типа `Bool` два взаимоисключающих конструктора данных, то для определения полновесной функции над этим типом нужно задать два равенства:

```
not       :: Bool -> Bool
not True  = False
not False = True
```

При вызове функции происходит *сопоставление с образцом*: выбирается то равенство, которое соответствует фактически переданному аргументу. Если в качестве аргумента передается не значение, а выражение, то, несмотря на ленивую природу языка, вычисление этого выражения форсируется. Это оправдано, поскольку иначе невозможно выбрать подходящее равенство, а значит невозможно продолжить вычисление. Именно этот механизм обеспечивает расходимость функции

```
bot :: Bool
bot = not bot
```

Отметим, что мы начали сопровождать определения функций глобальными *объявлениями сигнатуры типа*[12] (type signature declaration): `not :: Bool -> Bool`, `bot :: Bool`. Это необязательно (тайпчекер умеет выводить эти типы), но приветствуется. Такие объявления работают как верифицируемая тайпчекером документация.

4.1.1 Тип функции нескольких переменных

Каков тип можно приписать функции нескольких переменных? Вспомним функцию двух переменных

```
foo x y = 10 * x + y
```

Мы уже обсуждали, что ее можно интерпретировать как функцию одной переменной `x`, возвращающей функцию одной переменной `y`[13]. Это позволяет осуществить частичное применение: `foo 4` — это функция одного аргумента, прибавляющая к нему число 40. Отсюда естественным образом получаем, что следующий тип допустим для `foo`

[12] Обычно просто говорят – объявление типа.

[13] Напомним совершенно эквивалентную `foo' x = \y -> 10 * x + y`.

```
foo :: Int -> (Int -> Int)
```

Отметим, что функциональная стрелка в определении типа `(->)` представляет собой бинарный оператор. Хотя это оператор не над выражениями языка, а над типами, разработчиками языка для него задана правая ассоциативность. Поэтому другая, более компактная и общепринятая запись объявления типа `foo` такова

```
foo :: Int -> Int -> Int
```

Отметим, что правая ассоциативность функциональной стрелки на уровне типов хорошо согласуется с левой ассоциативностью оператора применения на уровне выражений `foo 4 2 == (foo 4) 2`. Действительно

```
GHCi> :t foo 4
foo 4 :: Int -> Int
GHCi> bar = foo 4
GHCi> :t bar
bar :: Int -> Int
GHCi> bar 2
42
```

Если тип имеет несколько конструкторов данных, то функция нескольких переменных для него задается путем сопоставления с нужным для реализации этой функции числом образцов:

```
(&&)           :: Bool -> Bool -> Bool
(&&) True True = True
(&&) x    y    = False
```

Полный перебор всех возможных конструкторов дал бы 4 уравнения. Это была бы допустимая, но не самая компактная реализация.

Мы видим, что допустимо вперемешку использовать в качестве формальных аргументов функций как переменные, так и конструкторы. Образцами называется и то и другое. При этом сопоставление с переменной всегда удачно.

В приведенной выше реализации второе уравнение содержит переменные `x` и `y` в левой части объявления функции. При необходимости мы могли бы использовать их при реализации правой части. Но в данном случае правая часть в них не нуждается, поэтому, чтобы не тратить впустую имена, можно использовать символ подчеркивания, называемый *джокер* (wildcard):

```
(&&)           :: Bool -> Bool -> Bool
(&&) True True = True
(&&) _    _    = False
```

4.1.2 Параметрический полиморфизм

Многие из рассмотренных нами функций были определены для конкретных типов данных. Однако некоторые функции не накладывают на типы своих аргументов совершенно никаких ограничений. Определим две подобные функции[14]

```
i x = x

k x1 x2 = x1
```

Мы можем вызывать их на любых аргументах:

```
GHCi> i 42
42
GHCi> i "Hello"
"Hello"
GHCi> k 'z' True
'z'
GHCi> k 'z' undefined
'z'
```

Такие функции называются *полиморфными*. Для описания типа их аргументов используют не имена конкретных типов, а *переменные типа*:

```
GHCi> :t i
i :: p -> p
GHCi> :t k
k :: p1 -> p2 -> p1
```

Используемый здесь род полиморфизма называют *параметрическим*, поскольку переменные типа (`p`, `p1`, `p2` в примерах выше) параметризуют типовое выражение.

Применение полиморфной функции к значению конкретного типа приводит к снятию полиморфизма. При этом тип получившегося выражения определяется так: все вхождения конкретизируемого параметра замещаются на конкретный тип фактически переданного аргумента:

```
GHCi> :t i True
i True :: Bool
GHCi> :t i 'w'
i 'w' :: Char
GHCi> :t k 'z' False
k 'z' False :: Char
```

[14] Функции с такими определениями присутствуют в стандартной библиотеке под именами `id` и `const`. Приведенные тут «однобуквенные» имена взяты из комбинаторной логики, где эти функции входят в набор канонических комбинаторов под стандартными названиями **I** и **K**.

В первых двух примерах в типе `i :: p -> p` переменная типа `p` заменяется на `Bool` и `Char` соответственно. В третьем примере в типе `k :: p1 -> p2 -> p1` происходит замена `p1` на `Char`, а `p2` на `Bool`.

При частичном применении полиморфизм снимается частично

```
GHCi> :t k 'z'
k 'z' :: p2 -> Char
GHCi> :t k "ABC"
k "ABC" :: p2 -> [Char]
```

Однако полиморфизм может сохраниться и при полном применении, если фактически переданный аргумент сам является полиморфным

```
GHCi> :t i i
i i :: p -> p
```

В этом выражении типы у двух вхождений `i` разные. Правое вхождение имеет стандартный полиморфный тип `i :: p -> p`. Левое вхождение «подстраивается» по типу под правое. Поскольку у функции тип формального и фактического аргумента должны совпадать, то тип левого вхождения `i :: (p -> p) -> (p -> p)`, что эквивалентно `i :: (p -> p) -> p -> p` благодаря правой ассоциативности функциональной стрелки. Применяя одно к другому получаем результат, наблюдаемый выше в сессии GHCi.

Более полное представление об работе с полиморфными типами можно получить, заглянув в реализацию системы вывода типов GHC. К счастью для знакомства с некоторыми подробностям реализации нет необходимости изучать исходный код тайпчекера. Для этого достаточно подключить два расширения. Первое `fprint-explicit-foralls` является расширением интерпретатора GHCi. Оно позволяет увидеть как на самом деле устроены полиморфные функции:

```
GHCi> :set -fprint-explicit-foralls
GHCi> :t i
i :: forall {p}. p -> p
GHCi> :t k
k :: forall {p1} {p2}. p1 -> p2 -> p1
```

Мы видим, что на самом деле переменные типа находятся под квантором общности. Например, тип `i` читается так: «для любого типа `p` это функция из `p` в `p`».

Такой подход делает `i` функцией не одной переменной, а двух. Мы можем отдельно управлять конкретизацией полиморфной функции, и отдельно передавать ей параметр конкретного типа. Для этого нужно подключить еще одно расширение[15]

[15] Это расширение требует явного указания сигнатуры используемых типов. Поэтому для того, чтобы примеры ниже работали, следует добавить объявления `i :: p -> p` и `k :: p1 -> p2 -> p1` в файл исходного кода и перезагрузить модуль.

```
GHCi> :set -XTypeApplications
```

Теперь мы можем применять выражение не только к выражению, но и выражение к типу (отсюда название расширения). Чтобы отличать этот род применения от обычного, перед именем типа используется специальный символ `@`. Например, применить комбинатор `i` к типу `Bool`

```
GHCi> :t i @Bool
i @Bool :: Bool -> Bool
```

Такое применение допустимо, только если выражение имеет полиморфный тип

```
GHCi> :t k @Bool
k @Bool :: forall {p2}. Bool -> p2 -> Bool
GHCi> :t k @Bool @Char
k @Bool @Char :: Bool -> Char -> Bool
GHCi> :t i @Bool @Char
<interactive>: error:
    * Cannot apply expression of type `Bool -> Bool'
      to a visible type argument `Char'
    * In the expression: i @Bool @Char
```

Мономорфизированные выражения можно использовать так же как обычные

```
GHCi> i @Int 42
42
GHCi> k @Bool @Char True 'z'
True
GHCi> k @Bool @Char True False

<interactive>: error:
    * Couldn't match expected type `Char' with actual type `Bool'
    * In the fourth argument of `k', namely `False'
      In the expression: k @Bool @Char True False
      In an equation for `it': it = k @Bool @Char True False
```

Отметим, что типовые параметры типов являются неявными. Если их опустить, то они будут выведены автоматически:

```
GHCi>  k @Char 'x' True
'x'
GHCi>  k @Char 'x' 42
'x'
GHCi> :t k @Char 'x'
k @Char 'x' :: forall {p2}. p2 -> Char
```

4.1.3 Специальный полиморфизм

Помимо описанного выше параметрического полиморфизма в Haskell есть еще и *специальный полиморфизм* (ad hoc polymorphism). Определим аналог `foo`, не указывая сигнатуру типа

```
GHCi> bas x y = 10 * x + y
```

Нетрудно увидеть, что эта функция имеет полиморфное поведение

```
GHCi> bas (2 :: Int) (3 :: Int)
23
GHCi> bas (2 :: Double) (3 :: Double)
23.0
```

Каков тип этой функции?

```
GHCi> :t bas
bas :: Num a => a -> a -> a
```

Это читается так: в *контексте* `Num a` функция имеет полиморфный тип `a -> a -> a`.

Контекст `Num a` накладывает на тип `a` ограничения: для него должны быть определены операторы сложения, умножения и т.п. Тип `bas` выводится тайпчекером из типа операций над аргументами `x` и `y` этой функции:

```
GHCi> :t (+)
(+) :: Num a => a -> a -> a
GHCi> :t (*)
(*) :: Num a => a -> a -> a
```

Механизмом описания набора ограничений, формирующих контекст, служат *классы типов* (type classes), мы изучим их позже. Заметим лишь, что в стандартной библиотеке определен класс типов `Num`, описывающий стандартный интерфейс для любого числового типа

```
class  Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
```

Отметим, что в Haskell числовые литералы полиморфны

```
GHCi> :t 42
42 :: Num p => p
```

и, поэтому, принадлежат любому типу данных, выставляющему интерфейс `Num`.

Чтобы обеспечить конкретный тип данных некоторым интерфейсом, нужно реализовать *представителя* (instance) соответствующего класса типов (имплементировать интерфейс). Для всех стандартных числовых типов реализованы представители класса типов `Num`. Именно поэтому мы можем осуществлять вызовы

```
GHCi> bas (2 :: Int) (3 :: Int)
23
GHCi> bas (2 :: Double) (3 :: Double)
23.0
GHCi>  bas (2 :: Integer) (3 :: Integer)
23
GHCi>  bas (2 :: Rational) (3 :: Rational)
23 % 1
```

Однако представители `Num` определены не для всех типов:

```
GHCi> bas 'y' 'z'
<interactive> error:
   * No instance for (Num Char) arising from a use of `bas'
```

В отличие от параметрического полиморфизма, где одна реализация обслуживает все бесконечное семейство допустимых типов, в случае специального полиморфизма каждый представитель конкретного класса типов должен быть реализован явно.

Перечислим на примерах функций наиболее часто используемые стандартные классов типов. Среди них класс `Eq`, отвечающий за сравнение на равенство

```
GHCi> :t (==)
(==) :: Eq a => a -> a -> Bool
GHCi> :t (/=)
(/=) :: Eq a => a -> a -> Bool
```

класс `Ord`, отвечающий за порядковые сравнения

```
GHCi> :t (<)
(<) :: Ord a => a -> a -> Bool
GHCi> :t (<=)
(<=) :: Ord a => a -> a -> Bool
```

класс `Enum`, отвечающий за итерации

```
GHCi> :t succ
succ :: Enum a => a -> a
GHCi> :t pred
pred :: Enum a => a -> a
```

классы Show и Read, отвечающие соответственно за сереализацию в строковое представление и десереализацию

```
GHCi> :t show
show :: Show a => a -> String
GHCi> :t read
read :: Read a => String -> a
```

Кроме того имеется целая иерархия числовых интерфейсов; вот некоторые примеры:

```
GHCi> :t div
div :: Integral a => a -> a -> a
GHCi> :t (/)
(/) :: Fractional a => a -> a -> a
GHCi> :t 3.14
3.14 :: Fractional p => p
GHCi> :t sin
sin :: Floating a => a -> a
GHCi> :t pi
pi :: Floating a => a
```

Все числовые типы являются представителями класса типов Num. Типы данных Int и Integer являются представителями класса типов Integral, в котором определено целочисленное деление. Типы данных Float и Double являются представителями классов Fractional и Floating, а тип данных дробей Rational — только Fractional.

4.1.4 Функции высших порядков и их типы

Функция высшего порядка (ФВП, high order function, HOF) — это функция, аргументом которой является функция. Определение в Википедии содержит продолжение «. . . или возвращаемым значением которой служит функция». Однако для языков с каррированными функциями такое определение не очень подходит, поскольку при этом любая функция более чем одного аргумента является ФВП. Например, оператор сложения чисел.

Тип функции высшего порядка должен содержать функциональный (стрелочный) тип в качестве аргумента. Простейший пример функции высшего порядка — оператор применения:

```
infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

Его левым аргументом служит функция, частичные применения через сечения это иллюстрируют

```
GHCi> :t (cos $)
(cos $) :: Floating b => b -> b
GHCi> :t ($ pi)
($ pi) :: Floating a => (a -> b) -> b
GHCi> cos $ pi
-1.0
```

Еще одна функция высшего порядка — это оператор композиции функций

```
infixr 9  .
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \ x -> f (g x)
```

Тип оператора композиции полностью соответствует ее определению: два его функциональных аргументам (`f` и `g`) полиморфны и по типу аргумента и по типу возвращаемого значения. Единственное ограничение - тип аргумента функции `f` (тип `b`) должен совпадать с возвращаемым типом `g`. Это ограничение отражено в типах:

```
f :: b -> c
g :: a -> b
```

что дает `f . g :: a -> c`. Отметим, что на оператор композиции мы можем смотреть двояко: как на функцию двух аргументов, возвращающую функцию — композицию этих аргументов, или как на функцию трех аргументов, вычисляющую значение этой композиции на третьем аргументе:

```
GHCi> :t show
show :: Show a => a -> String
GHCi> :t (+5)
(+5) :: Num a => a -> a
GHCi> :t show . (+5)
show . (+5) :: (Show a, Num a) => a -> String
GHCi>  (show . (+5)) 12
"17"
```

Приведем еще несколько примеров стандартных ФВП. В модуле `Data.Tuple` определены функции `curry` и `uncurry`, осуществляющие каррирование и обратную процедуру. Функция `curry` принимает функцию над парой, а возвращает функцию, аргументы в которую передаются последовательно:

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f = \ x y -> f (x,y)
```

Функция `uncurry` выполняет обратную процедуру, превращая обычную[16] функцию двух аргументов в функцию над парой

[16] В Haskell это значит каррированную!

```
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f = \ p -> f (fst p) (snd p)
```

Например, для наших `foo :: Int -> Int -> Int` и `bas :: Num a => a -> a -> a` можно построить раскаррированные версии:

```
GHCi> :t uncurry foo
uncurry foo :: (Int, Int) -> Int
GHCi> :t uncurry bas
uncurry bas :: Num c => (c, c) -> c
GHCi> (uncurry bas) (2,3)
23
```

В последнем примере работает механизм `default`: при попытке вычисления полиморфных числовых выражений происходит попытка мономорфизации до `Integer`, при неудаче проверки типов следует попытка мономорфизации до `Double`.

Еще один пример ФВП — функция, меняющаяя местами аргументы переданной ей функции:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f = \ y x -> f x y
```

Канонический пример использования `flip`

```
GHCi> flip (++) "Hello " "world!"
"world!Hello "
```

Вот еще одна библиотечная ФВП (из `Data.Function`), реализация которой использует `flip`:

```
infixl 1 &
(&) :: a -> (a -> b) -> b
(&) = flip ($)
```

Этот оператор работает как `$`, но с развернутым в противоположную сторону конвейером вычислений:

```
GHCi> (+12) $ (*10) $ 3
42
GHCi> 3 & (*10) & (+12)
42
```

4.2 Ленивость и форсирование

4.2.1 Тип ⊥

Сколько значений у типа `Bool`? На первый взгляд два — `True` и `False`, в соответствии с определением:

```
data Bool = True | False
```

Но это не так! Вспомним выражение `bot :: Bool`, определённое рекурсивно

```
bot :: Bool
bot = not bot
```

Его значение — не `True` и не `False`, это не завершающееся вычисление. В статической семантике языка Haskell ему приписывается значение ⊥ (напомним, что этот символ читается как основание или дно).

Ничего специфически булева в ⊥ нет, можно написать расходящиеся вычисления для любого типа. ⊥ — значение, разделяемое всеми типами:

```
⊥ :: forall {a}. a
```

Ошибкам, приводящим к аварийному завершению программы, тоже приписывается это значение. Аварийное прерывание может произойти в вычислении любого типа, поэтому

```
GHCi> :t undefined
undefined :: forall {a}. a
```

Эта «наивысшая степень полиморфизма» `undefined` приводит к следующему интересному наблюдению: если какое-то подвыражение проходящего проверку типов выражения заменить на `undefined`, то новое выражение тоже пройдет проверку типов. Поэтому `undefined` часто используют вместо еще не написанных частей программы.

4.2.2 ⊥ и функции: строгость и тотальность

Ленивую модель мы обсуждали ранее. Напомним, что при вычислениях в рамках этой модели сокращается внешний редекс. Это приводит к тому, что аргументы передаются в функцию невычисленными. Например, для

```
ignore x = 42
```

следующие вызовы не расходится

```
GHCi> ignore bot
42
GHCi> ignore undefined
42
```

Такие функции как `ignore`, игнорирующие значение своего аргумента, называются *нестрогими* (non-strict) по этому аргументу. *Строгая* (strict) функция, наоборот, определяется как расходящаяся на расходящемся аргументе, то есть ее характеристическое свойство

```
f ⊥ = ⊥
```

Классификация по шкале строгий-нестрогий связана с тем, как функция реагирует на переданную в качестве аргумента расходимость. Есть еще два важных понятия, описывающих может ли функция порождать расходимость или нет. Если функция завершается при всех нерасходящихся значениях аргументов, ее называют *тотальной* (total). Если же имеются нерасходящиеся значения аргументов, на которых функция расходится, то ее называют *частичной* (partial).

4.2.3 Ленивая модель: механизм разделения

Мы видели, что преимуществом ленивой модели по сравнению с энергичной служит лучшая определенность: не нужные для вычисления ответа расходимости эффективно элиминируются. Однако наивная реализация ленивой модели может приводить к неэффективности. Рассмотрим, например, функцию

```
fun x = (x+5,2*x)
```

Если применить нашу подстановочную модель для вычисления выражения `fun (3+1)`, то можно заметить, что сумма `(3+1)` будет вычисляться 2 раза (синий шаг редукции):

```
fun (3+1) → ((3+1)+5,2*(3+1))
          → (4+5,    2*(3+1))
          → (9,      2*(3+1))
          → (9,      2*4)
          → ...
```

Это может быть проблемой, если вместо суммы `(3+1)` у нас будет передаваться выражение, требующее долгого вычисления. Такая неэффективность — плата за передачу в функцию невычесленных аргументов.

Решением служит так называемый механизм *разделения*. Он позволяет не подставлять выражения вроде `(3+1)` напрямую, а избегать дублирования, обеспечивая передачу ссылки. Механизм разделения может быть реализован по-разному, например, через вычисления в контекстах или редукцию на графах. Первый способ выглядит так: вводится *контекст вычисления*, в котором повторяющиеся вхождения связываются с вспомогательным именем:

```
fun y = let x = y in (x+5,2*x)
```

Вычисления в контексте форсируются, когда в их результате возникает необходимость:

```
fun (3+1) → let x=3+1 in (x+5,2*x)
          → let x=4   in (x+5,2*x)
          → let x=4   in (4+5,2*x)
          → let x=4   in (9,  2*x)
          → let x=4   in (9,  2*4)
          → ...
```

Повторное обращение происходит к уже вычисленному значению в контексте.

4.2.4 Форсирование вычислений

В Haskell имеется возможность локально перейти к энергичной модели вычислений или, иными словами, форсировать вычисления по требованию программиста. Для этого используют специальный комбинатор `seq :: a -> b -> b`. Он может быть описан следующей парой уравнений

```
seq ⊥ b = ⊥
seq a b = b, если a ≠ ⊥
```

Отметим, что это описание поведения, а не реализация через механизм сопоставления с образцом ($\bot$ не является образцом). Это встроенный комбинатор, позволяющий нарушить ленивую семантику языка. Вычисление его первого аргумента происходит без необходимости, форсируется.

Использование `seq` потворствует распространению $\bot$, интересуясь значением своего первого аргумента

```
GHCi> seq undefined 42
*** Exception: Prelude.undefined
GHCi> seq (id undefined) 42
*** Exception: Prelude.undefined
```

Однако `seq` не форсирует вычисления для всех входящих в выражение редексов. Если у нас на верхнем уровне находится конструкция, отличная от редекса, то `seq` останавливается. К таким конструкциям относятся конструкторы данных, лямбда-абстракции и частично примененные функции. Эти конструкции, являясь «значениями», обеспечивают барьер для распространения $\bot$

```
GHCi> seq (undefined,undefined) 42
42
GHCi> seq (\x -> undefined) 42
42
GHCi> seq ((+) undefined) 42
42
```

Иногда упомянутые выше «не редексы» (конструкторы данных, лямбда-абстракции и частично примененные функции) объединяют одним термином – их называют *слабой головной нормальной формой* (weak head normal form, WHNF). Вводя это определение, можно перейти к короткой формулировке: `seq` форсирует вычисления до WHNF.

4.2.5 Использование `seq`

Через `seq` определяется оператор энергичного применения (с вызовом-по-значению)

```
infixr 0 $!
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

Как уже отмечалось ранее переход к энергичной модели приводит к «худшей определенности»

```
GHCi> ignore undefined
42
GHCi> ignore $! undefined
*** Exception: Prelude.undefined
```

Приведем пример использования энергичного применения. Вспомним факториал с аккумулирующим параметром

```
factorial n = helper 1 n where
  helper acc k | k > 1 = helper (acc * k) (k - 1)
               | otherwise = acc
```

Из-за ленивости к моменту срабатывания `otherwise` параметр `acc` будет содержать недовычисленое выражение вида

```
(...((1 * n) * (n - 1)) * (n - 2) * ... * 2)
```

Оптимизатор GHC обычно справляется, имея встроенный *анализатор строгости*. Но можно, не полагаясь на него, написать

```
factorial n = helper 1 n where
  helper acc k | k > 1 = (helper $! acc * k) (k - 1)
               | otherwise = acc
```

5 Алгебраические типы данных и сопоставление с образцом

Вспомним введенные ранее понятия образца (pattern) и сопоставления с образцом (pattern matching). Рассмотрим, например, библиотечную функцию, переставляющую элементы

пары

```
swap :: (a,b) -> (b,a)     -- Data.Tuple
swap (x,y) = (y,x)
```

Конструкция `(x,y)` в левой части определения функции представляет собой образец. Синтаксически он практически идентичен конструктору данных `(y,x)` в правой части. Действительно, конструкторы данных — это основной строительный блок для образцов. Однако, если применение конструкторов по прямому назначению *конструирует* данные, в образцах конструкторы используются для *деконструкции* переданных аргументов и извлечения их структурных составных частей.

При вызове

```
GHCi> swap (5+2,True)
(True,7)
```

происходит сопоставление с образцом:

- проверяется, что конструктор `(,)` — подходящий (для пары это тривиально);
- переменные `x` и `y` связываются с выражениями `5+2` и `True`;
- осуществляется подстановка выражений вместо переменных в теле функции `swap`.

До сих пор мы в основном имели дело со встроенными типами данных. Пользовательские типы данных в Haskell строятся из базовых с помощью трех операциий над типами: суммы, произведения и возведения в степень.

5.1 Тип суммы

Типы данных подобные

```
data Bool = True | False
```

называются *перечислениями*. Они представляют собой набор конструкторов, соединенных оператором `|`. Для того, чтобы сконструировать значение перечисления нужно выбрать один конструктор, поэтому оператор `|` читается как «или». Типы построенные с помощью этого оператора называют *типами суммы*.

В Prelude, помимо `Bool`, доступно еще одно полезное перечисление с тремя конструкторами данных

```
data Ordering = LT | EQ | GT
```

Конечно же мы можем объявлять пользовательские типы данных самостоятельно, используя ключевое слово `data`. Вот пример пользовательского перечисления с четырьмя конструкторами данных

```
data CardinalDirection = North | East | South | West
```

Конструкторы данных имеют тип `CardinalDirection`:

```
GHCi> dir = North
GHCi> :t dir
dir :: CardinalDirection
```

Однако попытка вывести значение `dir` приведет к ошибке:

```
GHCi> dir
 error: No instance for (Show CardinalDirection)
        arising from a use of `print'
    In a stmt of an interactive GHCi command: print it
```

Дело в том, что в поток вывода выводится строковое представление для значения типа данных, а как оно должно быть организовано зависит от намерений программиста. За перевод типа данных в строковое представление отвечает класс типов `Show`. Если мы хотим обеспечить такой перевод для пользовательского типа данных `CardinalDirection`, необходимо реализовать представителя класс типов `Show` для нашего типа. Однако, если нас устраивает простой вывод имени конструктора, можно попросить компилятор сгенерировать этого представителя автоматически, используя механизм *производных представителей* (deriving instances). Это очень просто сделать, достаточно внести небольшие изменения в объявление типа

```
data CardinalDirection = North | East | South | West deriving Show
```

Механизм производных представителей работает для целого ряда библиотечных классов типов.

Поскольку перечисления (и, шире, типы суммы) имеют несколько конструкторов, сопоставление с образцом должно порождать ветвление. Оно достигается определением функции через набор равенств — по равенству на каждый интересный в контексте реализации данной функции образец.

```
hasPole :: CardinalDirection -> Bool
hasPole North = True
hasPole South = True
hasPole _     = False
```

Сопоставление с образцом происходит сверху вниз, до первого удачного

```
GHCi> hasPole South
True
GHCi> hasPole West
False
```

Отметим, что порядок перечисления образцов важен — если в реализации `hasPole` переместить последнее уравнение вверх, результаты будут другими.

Встроенные типы данных ведут себя так, как будто они определены как перечисления:

```
data Char = '\NUL' | ... | 'a' | 'b' | 'c' | 'd' | ...
          | '\1114111'

data Int = -9223372036854775808 | ...
         | -2 | -1 | 0 | 1 | 2 | ...
         | 9223372036854775807

data Integer = ... | -2 | -1 | 0 | 1 | 2 | ...
```

Это позволяет использовать соответствующие литералы как образцы

```
isAnswer :: Integer -> Bool
isAnswer 42 = True
isAnswer _  = False
```

5.2 Семантика сопоставления с образцом

Поскольку определение функции может содержать

- несколько равенств;
- несколько аргументов-образцов в каждом равенстве;
- вложенные подобразцы в каждом образце

требуются объемные правила сопоставления. Общий принцип, однако, описать довольно просто: **сопоставление с образцом происходит сверху-вниз, затем слева-направо**. Результат сопоставления с образцом может быть

- успешным (succeed);
- неудачным (fail);
- расходящимся (diverge).

Успех означает, что мы можем переходить к правой части текущего равенства. Разница между неудачей и расходимостью такова: при неудаче мы просто переходим к следующему равенства, а расходимость аварийно завершает работу.

Образцам помимо конструкторов называют также переменные и подчеркивание[17]. Последние два вида образцов относят к *неопровержимым* (irrefutable). Неопровержимы те образцы, сосоставление с которыми всегда успешно. Образец, сопоставление с

[17] Это удобно терминологически — формальные аргументы функции всегда являются образцами.

которым может завершиться отличным от успеха исходом называют *опровержимым* (refutable).

Посмотрим как работают приведенные правила на примере функции

```
bar (1, 2) = 3
bar (0, _) = 5
```

Вызов `bar (1, 2)` приводит к успеху на первом же равенстве. Вызов `bar (0, 7)` неудачен на первом равенстве, и успешен на втором. Вызов `bar (2, 1)` — две неудачи и, как следствие, расходимость.

Вызов `bar (1, 5-3)` будет успешен на первом равенстве. Однако для того, чтобы удостовериться в этом, системе времени исполнения придется форсировать вычисление аргумента. Это добавляет в ленивый язык некоторую (необходимую) степень энергичности. Однако вычисление форсируется только до слабой головной нормальной формы (WHNF): достаточно получить на верхнем уровне некоторый конструктор данных. Все, что находится за его барьером, остается невычисленным.

Вызов `bar (1, undefined)` приводит к расходимости в первом же образце, причина этого ясна из предыдущего абзаца.

Однако вызов `bar (0, undefined)` возвращает 5, то есть он неудачен (но не расходится!) в первом образце и успешен во втором. Здесь срабатывает правило «слева направо»: в первом равенстве неудача сопоставления фактического 0 с ожидаемой 1 приводит к тому, что все сопоставление объявляется неудачным, дело просто не доходит до потенциальной расходимости.

5.3 Тип произведения

Декартово *произведение* нескольких типов — это тип, в котором лежат значения всех сомножителей. Простейшие произведения — это встроенные кортежи (пары, тройки и т.д.). Однако работа с кортежами не очень удобна, часто хочется иметь конструктор данных, название которого отражает логику предметной области. Пусть мы, например, хотим работать с точками на координатной плоскости, и использовать встроенный тип `Double` для хранения координат. В Haskell для этих целей мы можем использовать тип-произведение двух значений типа `Double` с одним именованным конструктором данных:

```
data PointDouble = PtD Double Double
  deriving Show
```

Отметим, что имя типа данных (в нашем примере `PointDouble`) может не совпадать с именем конструктора данных (в нашем примере `PtD`), хотя в случае, когда конструктор данных один, их обычно делают одинаковыми.

Конструктор данных имеет тип функции, арность которой равна количеству сомножителей тип-произведения. Как и любая функция, конструктор может применяться частично

```
GHCi> :t PtD
PtD :: Double -> Double -> PointDouble
GHCi> :t PtD 3
PtD 3 :: Double -> PointDouble
GHCi> :t PtD 3 5
PtD 3 5 :: PointDouble
```

Однако конструктор данных, в отличие от функции, не имеет никакого кодируемого пользователем вычислительного поведения. Будучи снабженным требуемым количеством фактических аргументов, он просто порождает значение целевого типа, выступающее в роли хранилища для выражений-аргументов.

Сопоставление с образцом для типа-произведения решает обратную задачу — извлечение выражений из этого хранилища и связывания их с именами для дальнейшего использования. Вот, например, функция, которая по переданным координатам двух точек ищет координаты середины соединяющего эти точки отрезка:

```
midPointDouble :: PointDouble -> PointDouble -> PointDouble
midPointDouble (PtD x1 y1) (PtD x2 y2) =
   PtD ((x1 + x2) / 2) ((y1 + y2) / 2)
```

Координаты точек-аргументов функции связываются в образце с переменными и обрабатываются в правой части для получения координат середины отрезка:

```
GHCi> midPointDouble (PtD 3 5) (PtD 9 8)
PtD 6.0 6.5
```

5.3.1 Полиморфные типы

Тип данных `PointDouble` был создан в представлении, что координаты точки имеют тип `Double`. Однако могут возникнуть ситуации, когда удобнее иметь координаты типа `Float` или вообще целочисленные, например, `Int`.

Универсальным решением будет сделать тип точки полиморфным по типу координат. Для этого нужно параметризовать тип точки ти́повым параметром:

```
data Point a = Pt a a
  deriving Show
```

Теперь тип конструктора данных полиморфен

```
GHCi> :type Pt
Pt :: a -> a -> Point a
```

и мы можем создавать точки с разными типами координат

```
GHCi> pDbl = Pt (3::Double) (5::Double)
GHCi> pDbl
Pt 3.0 5.0
GHCi> pInt = Pt (3::Int) (5::Int)
GHCi> pInt
Pt 3 5
GHCi> :t pDbl
pDbl :: Point Double
GHCi> :t pInt
pInt :: Point Int
```

Сам по себе конструктор типа `Point` становится теперь не обычным типом, а функцией над типами (чаще говорят *оператор над типами*, хотя это не оператор в смысле Haskell). Конкретный тип, напирмер, `Point Int` получается применением конструктора типа к обычному типу; в нашем случае к `Int`. Обратим внимание, что здесь используется тот же минималистичный синтаксис применения, что и при построении выражений, хотя речь идет о построении типов!

Конструктор типа `Point` имеет один ти́повый параметр `a`. Это значит, что его применения `Point Int` или `Point Double` допустимы, а, например, `Point Int Double` нет. Очевидно, что конструкция `Point Point` тоже должна быть отвергнута: хотя число параметров здесь подходящее, *тип* параметра не тот что нужен. Постойте-ка, мы заговорили о типе типа! Для статического контроля подобного рода в Haskell имеется система типов над системой типов, которая для избежания путаницы называется системой *кайндов*.

Система кайндов структурно повторяет систему типов Haskell, однако базовый кайнд всего один — `*`. Все простые типы имеют этот кайнд. К простым относятся все типы, которые нами использовались справа в отношении типизации `::`, в том числе и стрелочные. Этот же оператор `::` используется для отношения «кайндизации» в системе кайндов. Ровно так же как в системе типов утверждение `42 :: Int` читалось как «`42` — это целое», в системе кайндов утверждение `Int :: *` читается как «`Int` — это (простой) тип». Утверждение `(+5) :: Int -> Int`, читающееся как «`(+5)` — это функция с одним параметром из `Int` в `Int`», в системе кайндов имеет аналог `Point :: * -> *`, который читается как «`Point` — это однопараметрический оператор из простого типа в простой тип».

GHCi позволяет анализировать кайнды как пользовательских типов

```
GHCi> :kind Point
Point :: * -> *
GHCi> :kind Point Int
Point Int :: *
```

так и встроенных

```
GHCi> :k []
[] :: * -> *
GHCi> :k [] Int
[] Int :: *
GHCi> :k [Int]
[Int] :: *

GHCi> :k (,)
(,) :: * -> * -> *
GHCi> :k (,) Char
(,) Char :: * -> *
GHCi> :k (,) Char Bool
(,) Char Bool :: *
GHCi> :k (Char,Bool)
(Char,Bool) :: *

GHCi> :k Int -> Int -> Int
Int -> Int -> Int :: *
```

5.3.2 Полиморфные функции над полиморфными типами

Полиморфные типы данных полиморфны параметрически, то есть на типовый параметр `a` в типе `Point a` невозможно наложить ad hoc ограничения[18]. Однако при реализации функций над полиморфными типами подобные ограничения допустимы. Например, в общем случае невозможно найти середину отрезка, если координаты вершин целочисленны. Поэтому в полиморфном обобщении подобной функции возникает контекст `Fractional`:

```
midPoint :: Fractional a => Point a -> Point a -> Point a
midPoint (Pt x1 y1) (Pt x2 y2) =
  Pt ((x1 + x2) / 2) ((y1 + y2) / 2)
```

Контекст `Fractional` нет необходимости писать руками, он выводится автоматически из использования в теле функции оператора `(/)`. Если бы в теле использовался только оператор сложения, мы бы имели более общий контекст `Num`.

```
GHCi> :t midPoint (Pt 3 5) (Pt 9 8)
midPoint (Pt 3 5) (Pt 9 8) :: Fractional a => Point a
GHCi> midPoint (Pt 3 5) (Pt 9 8)
Pt 6.0 6.5
```

[18] Отметим, что в старых версиях Haskell такое было возможно.

5.3.3 Стандартные алгебраические типы

В стандартной библиотеке помимо кортежей и списков имеется несколько полезных полиморфных типов общего назначения.

Тип `Maybe a` позволяет добавить к произвольному типу `a` дополнительное необязательное значение

```
data  Maybe a  =  Nothing | Just a
```

Конструктор `Just` оборачивает «регулярное» значение, а `Nothing` играет роль дополнительного элемента. Для извлечения значения можно использовать сопоставление с образцом или библиотечную функцию-элиминатор[19] (из модуля `Data.Maybe`)

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

Эта функция позволяет по переданному значению типа `Maybe a` получить значение произвольного типа `b`, если у нас имеется обработчик типа `a -> b` для «регулярных» значений и значение типа `b` для случая `Nothing`.

Примером использования типа `Maybe` может служить библиотечная функция поиска элемента списка, удовлетворяющего предикату:

```
find :: (a -> Bool) -> [a] -> Maybe a
```

Если элемент, удовлетворяющий предикату, отсутствует, возвращается `Nothing`:

```
GHCi> find (>10) [7,5,12,16]
Just 12
GHCi> find (>20) [7,5,12,16]
Nothing
```

Тип `Either a b` хранит одно значение из двух, являясь типом-суммой двух произвольных типов

```
data  Either a b  =  Left a | Right b
```

Этот тип является стандартным библиотечным способом построить сумму типов, взяв два существующих типа. В этом он похож на стандартный двухэлементный кортеж (пару), которая обеспечивает стандартный способ построить произведение двух типов.

В модуле `Data.Either` определен стандартный элиминатор для типа `Either`[20]

[19] *Элиминатором* называется функция «обратная» конструктору в том смысле, что она позволяет, взяв значение некоторого типа, «использовать» его функциональность, желательно наиболее общим образом.

[20] Отметим дуальность типов `Either` и `(,)`. У первого два конструктора и один элиминатор, а у второго один конструктор и два элиминатора `fst` и `snd`.

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
```

С помощью `Either` обычно описывают вычисления, в которых возможны проблемные результаты. Конструктор данных `Right` используют для упаковки успешных результатов, а `Left` — для реакции на неудачу.

```
(/?) :: (Eq a, Fractional a) => a -> a -> Either String a
x /? 0 = Left "(/?) error: division by zero"
x /? y = Right (x / y)
```

```
GHCi> (6 /? 2)
Right 3.0
GHCi> (6 /? 0)
Left "(/?) error: division by zero"
GHCi> either show show (6 /? 2)
"3.0"
GHCi> either show show (6 /? 0)
"\"(/?) error: division by zero\""
```

На основе типа `Either` в Haskell построен стандартный монадический механизм обработки исключений.

И `Maybe` и `Either` представляют собой суммы типов. При этом слагаемые в трех случаях из четырех можно рассматривать как типы-произведения из одного сомножителя, а в четвертом (`Nothing`) — из нуля сомножителей.

5.4 Экспоненциальные типы

Экспоненциальный тип — это тип функции. Такое название происходит из наблюдения за мощностями типов, то есть количеством их элементов. У типа `Bool` мощность равна 2, у типа `Ordering` — 3. У типа их суммы `Either Bool Ordering` мощность равна $2+3=5$. У типа их произведения `(Bool,Ordering)` имеется $2*3=6$ различных элементов. Если посчитать число различных[21] функций типа `Bool -> Ordering`, то окажется, что их $3^2=9$. Если же посчитать число различных функций типа `Ordering -> Bool`, то окажется, что их $2^3=8$. Иными словами, на вполне разумных основаниях вместо стандартного встроенного типа функции `a -> b` можно было бы использовать нотацию `b ^ a`.

Экспоненциальный (или функциональный) тип — это полноценный тип языка, его можно использовать всюду, где мы можем писать произвольный тип. В частности, мы можем использовать его в качестве типа аргумента конструктора данных:

[21] Две функции считаются различными, если существует значение аргумента, на котором они дают разные результаты.

```
data Endom a = Endom (a -> a)
```

Этот тип представляет собой контейнер для хранения функции, у которой область определения и множество значений совпадают[22]. Можно написать элиминатор для извлечения функции из такого контейнера:

```
appEndom :: Endom a -> a -> a
appEndom (Endom f) = f
```

Извлечение функции происходит при частичном применении элиминатора

```
GHCi> e = Endom (\n -> 2 * n + 3)
GHCi> :t e
e :: Num a => Endom a
GHCi> :t appEndom e
appEndom e :: Num a => a -> a
```

А полное применение `appEndom` дает извлечение-а-затем-применение[23]

```
GHCi> e `appEndom` 5
13
```

У экспоненциального типа есть оно отличие от типов суммы и произведения. Мы не можем «разобрать» такой тип с помощью сопоставления с образцом, переменная является единственным подходящим образцом для функционального аргумента.

5.5 Рекурсивные типы

При объявлении типов можно использовать рекурсию. Это значит, что допустимо указывать объявляемый тип в качестве аргумента конструктора данных[24]:

```
GHCi> data Nat = Zero | Suc Nat deriving Show
```

Конструктор `Suc` при этом оказывается эндоморфизмом над типом `Nat`

```
GHCi> :t Zero
Zero :: Nat
GHCi> :t Suc
Suc :: Nat -> Nat
```

Это позволяет строить неограниченное число обитателей данного типа:

[22]Название типа происходит от слова *эндоморфизм*, которым подобные функции обозначают математики. В стандартной библиотеке имеется тип `Endo` почти эквивалентный приведенному здесь, но объявленный не через `data`, а через другой механизм (`newtype`) для большей эффективности.

[23]Отсюда название `appEndom`: **app**lication of **endom**orphism — применение эндоморфизма.

[24]Пример ко всему прочему иллюстрирует возможность объявления `data` в GHCi.

```
GHCi> one = Suc Zero
GHCi> two = Suc (Suc Zero)
GHCi> three = Suc two
GHCi> four = Suc three
...
```

Это так называемые числа Пеано. Их можно рассматривать как способ кодирования натуральных чисел в унарной системе исчисления. Поскольку тип `Nat` это тип суммы, тотальные функции над ним, использующие опровержимые образцы, определяются несколькими равенствами[25]

```
GHCi> {pred (Suc n) = n; pred Zero = Zero}
GHCi> pred two
Suc Zero
```

Другим примером рекурсивного типа служит список. Хотя встроенный список имеет специальный синтаксис, мы можем определить полностью эквивалентный ему пользовательский тип списка

```
data List a = Nil | Cons a (List a)
  deriving Show
```

Конструкторы данных типа `List` имеют типы

```
Nil :: List a
Cons :: a -> List a -> List a
```

Пользовательские функции над рекурсивными типами данных удобно писать через сопоставление с образцом и рекурсию

```
len :: List a -> Int
len  Nil         =  0
len (Cons _ xs)  =  1 + len xs
```

Рекурсия в типах позволяет строить конструкции произвольной степени вложенности, а рекурсия в функциях дает возможность эти конструкции обрабатывать:

```
GHCi> myList = Cons 'a' (Cons 'b' (Cons 'c' Nil))
GHCi> len myList
3
```

[25] Хотя тип `Nat` структурно похож на числа Черча, вычисление предессора здесь намного эффективнее, благодаря наличию механизма сопоставления с образцом.

6 Стандартные списки и работа с ними

Списки являются основным контейнерным типом в функциональных языках. Стандартные списки являются встроенным типом данных со специальным синтаксисом. Список это тип-сумма с двумя конструкторами данных. Первый — это конструктор пустого списка

```
[] :: [a]
```

Второй конструктор добавляет элемент в голову уже существующего списка:

```
infixr 5 :
(:) :: a -> [a] -> [a]
```

Этот конструктор представляет собой оператор. В Haskell конструкторы-операторы допустимы, они называются *инфиксными конструкторами данных*. Однако на их имена накладывается специальное ограничение: они должны начинаться с символа двоеточия. Конструктор списка удовлетворяет этому требованию.

Синтаксис для списка с квадратными скобками это синтаксический сахар для применения конструкторов. Правила трансляции ясны из следующего примера:

```
[1,2,3] ≡ 1:2:3:[]
```

Поскольку конструктор `(:)` правоассоциативен

```
1:2:3:[] ≡ 1:(2:(3:[]))
```

6.1 Стандартные функции над списками

Примеры определения функций над списком

```
head        :: [a] -> a
head (x:xs) =  x
head []     =  error "Prelude.head: empty list"

tail         :: [a] -> [a]
tail (x:xs)  =  xs
tail []      =  error "Prelude.tail: empty list"
```

Многие функции над списками используют сопоставление с двумя образцами: один для пустого списка `[]`, другой для непустого `(x:xs)`. При этом переменная `x` связывается с головой (первым элементом) непустого списка, а переменная `xs` — с его хвостом (остатком после отделения первого элемента). Две приведенные выше функции просто возвращают результаты этих связываний.

```
GHCi> head [1,2,3,4]
1
GHCi> tail "ABCD"
"BCD"
```

Функции `head` и `tail` являются частичными. В современном Haskell использовать подобные функции не рекомендуется, не смотря на то, что они входят в стандартную библиотеку.

Еще одним важнейшим оператором для списков служит оператор бинарной конкатенации: он делает из двух списков один, присоединяя первый к началу второго.

```
infixr 5 ++

(++)          :: [a] -> [a] -> [a]
[]     ++ ys  =  ys
(x:xs) ++ ys  =  x : xs ++ ys
```

Из реализации оператора видно, что его сложность (число рекурсивных вызовов до наступления терминирующего условия) линейно зависит от размера первого списка и не зависит от второго.

В модуле `Data.List` имеется много полезных функций для работы со списками. Наиболее употребительные из них экспортируются в `Prelude`. Приведем некоторые из них.

Функция, возвращающая количество элементов в списке[26]

```
length    :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

```
GHCi> length "Hello world!"
12
```

Функция, обеспечивающая конкатенацию списка списков в список.

```
concat          :: [[a]] -> [a]
concat []       = []
concat (xs:xss) = xs ++ concat xss
```

Она естественным образом реализуется через бинарную конкатенацию.

```
GHCi> concat ["Hello ","world","!"]
"Hello world!"
```

[26]В современном Haskell эта функция (и ряд других, приведенных ниже) имеет более общую сигнатуру и реализацию, позволяющую обслуживать широкий класс контейнерных типов.

6.1.1 Поиск элементов

Функция, проверяющая наличие элемента в списке

```
infix  4 `elem`
elem  :: (Eq a) => a -> [a] -> Bool
elem _ []     = False
elem x (y:ys) = x == y || elem x ys
```

Часто используется инфиксно

```
GHCi> 'e' `elem` "Hello"
True
GHCi> 'i' `elem` "Hello"
False
```

Функция, осуществляющая поиск значения с заданным ключом в ассоциативном списке (то есть списке пар ключ-значение)

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup _   []          = Nothing
lookup key ((k,v):kvs)
  | key == k           = Just v
  | otherwise          = lookup key kvs
```

Возвращаемым значением служит тип `Maybe`, поскольку поиск может завершится неудачей

```
GHCi> lookup 2 [(1,"Hello"),(3,"world")]
Nothing
GHCi> lookup 3 [(1,"Hello"),(3,"world")]
Just "world"
```

Оператор, возвращающий элемент списка с заданным индексом

```
infixl 9  !!
(!!) :: [a] -> Int -> a
xs     !! n | n < 0 =  error "Prelude.!!: negative index"
[]     !! _         =  error "Prelude.!!: index too large"
(x:_)  !! 0         =  x
(_:xs) !! n         =  xs !! (n-1)
```

Левая ассоциативность позволяет удобно обслуживать вложенные списки:

```
GHCi> ["Hello","world"] !! 0 !! 1
'e'
```

```
GHCi> ["Hello","world"] !! 1 !! 2
'r'
```

6.1.2 Выделение подсписков

Функция `take` получает целое число `n` и список и возвращает первые `n` элементов списка. Если элементов меньше, чем `n`, возвращается сколько есть. Если `n` не положительно, возвращается пустой список.

```
take                    :: Int -> [a] -> [a]
take n _      | n <= 0  =  []
take _ []               =  []
take n (x:xs)           =  x : take (n-1) xs
```

```
GHCi> take 3 "ABCDE"
"ABC"
GHCi> take 10 "ABCDE"
"ABCDE"
```

Функция `drop` «дуальна» к `take`: она отбрасывает первые `n` элементов, возвращая то, что осталось.

```
drop                    :: Int -> [a] -> [a]
drop n xs     | n <= 0  =  xs
drop _ []               =  []
drop n (_:xs)           =  drop (n-1) xs
```

6.1.3 Функции высших порядков над списками

В библиотеке `Data.List` много функций высших порядков. У следующих двух функций первый аргумент — функция типа `a -> Bool`, то есть унарный предикат.

```
filter          :: (a -> Bool) -> [a] -> [a]
filter _ []     =  []
filter p (x:xs)
  | p x         =  x : filter p xs
  | otherwise   =  filter p xs

takeWhile       :: (a -> Bool) -> [a] -> [a]
takeWhile _ []  =  []
takeWhile p (x:xs)
  | p x         =  x : takeWhile p xs
  | otherwise   =  []
```

Функция `filter` возвращает (сохраняя порядок) все те элементы списка, которые удовлетворяют предикату. Функция `takeWhile` возвращает наибольший префикс списка, все элементы которого удовлетворяют предикату.

```
GHCi> filter (<50) [2,12,85,0,6]
[2,12,0,6]
GHCi> takeWhile (<50) [2,12,85,0,6]
[2,12]
GHCi> dropWhile (<50) [2,12,85,0,6]
[85,0,6]
```

У функции высшего порядка `map` функциональный аргумент — произвольная функция:

```
map          :: (a -> b) -> [a] -> [b]
map _ []     =  []
map f (x:xs) =  f x : map f xs
```

Функция `map` обрабатывает каждый элемент списка переданной функцией-обработчиком, формируя список результатов той же длины, но, возможно, другого типа.

```
GHCi> map (+1) [1,2,3]
[2,3,4]
GHCi> map length ["Good","bye","world"]
[4,3,5]
GHCi> map (^2) . map length $ ["Good","bye","world"]
[16,9,25]
```

6.1.4 Семейства `zip` и `zipWith`

Имеется набор стандартных функции, позволяющих соединить несколько списков в один. Мы уже обсуждали конкатенацию, когда один из списков присоединялся в голову другого. Другим способом соединения служит «зиппирование», от английского zipper (застежка-молния). Зубцы с двух сторон такой застежки сцепляются последовательно: первый с первым, второй со вторым и т.д. Именно так обрабатываются и элементы двух списков в функции `zip`: из элементов с одинаковыми индексами образуется пара, помещаемая в результирующий список с тем же индексом:

```
zip :: [a] -> [b] -> [(a,b)]
zip []     _      = []
zip _      []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

На самом деле имеется целое семейство функций, подобных `zip`: сама `zip`, `zip3`, `zip4` и т.д. Результирующий список образуется конструированием кортежа подходящего размера из элементов списков-аргументов:

```
zip3 :: [a] -> [b] -> [c] -> [(a,b,c)]

zip4 :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]

...
```

Функция `zip` задает стандартный для Haskell способ явного индексирования:

```
GHCi> zip [1,2,3,4] "ABC"
[(1,'A'),(2,'B'),(3,'C')]
```

Отметим, что длина самого короткого из списков определяет длину результата. Это означает, что при «зиппировании» мы можем потерять часть информации — в приведенном примере это число 4.

Имеется семейство `unzip`, «обратное» `zip`:

```
unzip :: [(a,b)] -> ([a],[b])

unzip3 :: [(a,b,c)] -> ([a],[b],[c])

...
```

Более общим способом соединения двух списков служит замена в функциях семейства `zip` конструктора данных пары (тройки, четверки и т.п.) на произвольную функцию нужного числа аргументов, предоставляеную пользователем. Это порождает семейство функции высших порядков `zipWith`

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ []     _      = []
zipWith _ _      []     = []
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs

zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
...
```

Все перечисленные семейства содержат функции вплоть до 7 аргументов.

Задача 1. Верно ли, что `uncurry zip . unzip` эквивалентна `id` на своей области определения?

Задача 2. Верно ли, что `unzip . uncurry zip` эквивалентна `id` на своей области определения?

6.2 Способы генерации списков

Если определить рекурсивную функцию без терминирующего условия, то такая рекурсия во время исполнения приведет к расходимости.

```
GHCi> ones = 1 : ones
GHCi> :type ones
ones :: Num a => [a]
GHCi> ones
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,Interrupted.
GHCi> numsFrom n = n : numsFrom (n+1)
GHCi> :t numsFrom
numsFrom :: Num t => t -> [t]
GHCi> numsFrom 3
[3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,Interrupted.
```

Однако, если, как в приведенных выше примерах, в процессе рекурсии мы конструируем некоторую структуру данных рекурсивного типа, то возникающая расходимость является продуктивной. Это значит, что мы можем терминировать ее внешними средствами, пользуясь тем, что сопоставление с образцом форсирует вычисления до слабой головной нормальной формы, то есть до конструктора на верхнем уровне.

```
GHCi> take 2 ones
[1,1]
GHCi> take 4 (numsFrom 3)
[3,4,5,6]
```

Каков механизм этой «внешней терминации»? Вспомнив реализацию

```
take  :: Int -> [a] -> [a]
take n _      | n <= 0  = []
take _ []               = []
take n (x:xs)           = x : take (n-1) xs
```

приведем шаги ленивого вычислителя:

```
take 2 ones ⟶ -- (1)
take 2 (1 : ones) ⟶ -- (2)
1 : take (2-1) ones) ⟶ -- (3)
1 : take 1 (1 : ones) ⟶ -- (4)
1 : 1 : take (1-1) ones) ⟶ -- (5)
1 : 1 : take 0 ones) ⟶ -- (6)
1 : 1 : []
```

- (1) сопоставление с образцом форсирует вычисление `ones` до WHNF;

- (2) подходит последнее уравнение в определении `take`, используем его;
- (3) сопоставление с образцом форсирует вычисление до WHNF обоих аргументов `take`: первого, чтобы отвергнуть первое из уравнений в определении `take`, второго — чтобы отвергнуть второе;
- (4) используем последнее уравнение в определении `take`;
- (5) сопоставление с образцом форсирует вычисление до WHNF первого аргумента `take`;
- (6) подходит первое уравнение в определении `take`, используем его.

Таким образом мы можем эффективно работать с «бесконечными» списками. Более того, мы можем декларативно описывать преобразования бесконечных списков, порождая новые (бесконечные) списки. Функции работы со списками из `Data.List` реализованы так, чтобы способ обработки бесконечных списков ничем не отличается от конечных:

```
GHCi> squares = map (^2) (numsFrom 0)
GHCi> takeWhile (<=100) squares
[0,1,4,9,16,25,36,49,64,81,100]
```

Каноническим примером рекурсивного построения бесконечного списка служит бесконечный список чисел Фибоначчи

```
GHCi> fibs = 0 : 1 : zipWith (+) fibs (drop 1 fibs)
GHCi> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

Имеется библиотечный аналог нашей пользовательской функции `numsFrom`. Он носит название `enumFrom` и имеет более общий тип, позволяющий работать не только с числами, но и с любыми перечислениями, то есть любыми представителями класса типов `Enum`.

```
GHCi> :t enumFrom
enumFrom :: Enum a => a -> [a]
GHCi> enumFrom False
[False,True]
GHCi> enumFrom ((maxBound::Int) - 2)
[9223372036854775805,9223372036854775806,9223372036854775807]
```

Заметим, что когда тип перечисления является ограниченным (то есть представителем класса типов `Bounded`), то список аккуратно терминируется. Для неограниченного типа `Integer` функция `enumFrom` порождает бесконечный список.

Для функции `enumFrom` имеется удобный синтаксический сахар, задаваемый следующим правилом трансляции

```
[e1..] ≡ enumFrom e1
```

То есть предыдущие примеры могут быть записаны так

```
GHCi> [False ..]
[False,True]
GHCi> [(maxBound::Int) - 2 ..]
[9223372036854775805,9223372036854775806,9223372036854775807]
```

Класс типов `Enum` задает еще ряд полезных *арифметических последовательностей*, которые транслируются в функции, подобные `enumFrom`:

```
GHCi> [1..10]
[1,2,3,4,5,6,7,8,9,10]
GHCi> ['A'..'z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz"
GHCi> [1,3..17]
[1,3,5,7,9,11,13,15,17]
GHCi> [1,3..]
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,Interrupted.
```

Для формирования «нелинейных» последовательностей имеется другая техника, носящая название *выделение списка* (list comprehension)[27]

```
GHCi> digits = [0..9]
GHCi> [ x^2 | x <- digits ]
[0,1,4,9,16,25,36,49,64,81]
```

Часть справа от вертикальной черты носит название генератора: элементы, связываемые с переменной `x` пробегают по всему списку `digits`. Слева от вертикальной черты находится выражение, в котором можно использовать `x`[28]. При наличии нескольких генераторах чаще обновляется тот, что правее:

```
GHCi> [ [x,y] | x <- "ABC", y <- "de" ]
["Ad","Ae","Bd","Be","Cd","Ce"]
```

Генераторы могут ссылаться на значения из предыдущих генераторов; кроме того можно использовать предикаты над этими значениями для фильтрации результатов. Вот например, выделение пифагоровых троек (сторон прямоугольных треугольников с целыми длинами):

[27] Название и его перевод восходят к аксиоматической теории множеств Цермело-Френкеля, в которой имеется аксиома выделения (axiom of comprehension).

[28] Можно провести аналогию с математической нотацией $\{x^2 \mid x \in \texttt{digits}\}$, читая символ `<-` как «принадлежит», а не «пробегает». Впрочем, аналогия не совсем полная: в отличие от множеств для списков порядок элементов существенен.

```
GHCi> ls = [1..19]
GHCi> [ (x,y,z) | x <- ls, y <- [1..x], z <- ls, x^2 + y^2 == z^2 ]
[(4,3,5),(8,6,10),(12,5,13),(12,9,15),(15,8,17)]
```

Для фильтрации можно использовать сопоставление с образцом в генераторе:

```
GHCi> tst = [Just 2,Nothing,Just 3]
GHCi> [ x | Just x <- tst ]
[2,3]
```

Хотя допустим всего один образец, неудачное сопоставление с ним не влечет расходимости: элемент просто не включается в результирующий список.

7 Образцы: дополнительные сведения

Все языковые конструкции содержащие образцы при трансляции в Kernel превращаются в выражение `case ... of ...`. Например, функция

```
head (x:_)    =  x
head []       =  error "head: empty list"
```

транслируется в Kernel следующим образом

```
head'' xs = case xs of
  (x:_) -> x
  []    -> error "head'': empty list"
```

Общее правило трансляции сопоставления с образцом таково: функция

```
f p11 ... p1k -> e1
...
f pn1 ... pnk -> en
```

транслируется в Kernel следующим образом

```
f x1 ... xk = case (x1,...,xk)
  (p11,...,p1k) -> e1
  ...
  (pn1,...,pnk) -> en
```

Здесь (и в целом в правилах трансляции) p_{ij} — метапеременные, маркирующие образцы, а x_j — переменные. Про последние здесь оговаривается, что они *свежие*, то есть отличные от всех других переменных в программе.

Поскольку `case ... of ...` — выражение, его можно использовать в любом месте кода:

```
GHCi> f v = (case v of Just x -> x; Nothing -> 2)^2
GHCi> f (Just 5)
25
GHCi> f Nothing
4
```

Такое поведение полезно, когда возникает необходимость выполнить сопоставление с образцом некоторой переменной в процессе реализации тела функции.

7.1 Специальные виды образцов

7.1.1 As-образец

В некоторых ситуациях мы имеем в левой части определения функции сложный образец с глубоким вложением. При этом в правой части нам может потребоваться собрать ту же самую структуру обратно. Например, в определении функции

```
dupFirst        :: [a] -> [a]
dupFirst (x:xs) = x:x:xs
```

мы справа собираем список-аргумент функции `x:xs`, разобранный слева. Можем присвоить локальный псевдоним `ys` образцу `x:xs`, используя затем этот псевдоним в правой части определения

```
dupFirst'           :: [a] -> [a]
dupFirst' ys@(x:xs) = x:ys
```

Такой образец носит название *as-образца*.

7.1.2 Ленивые образцы

Мы уже знакомились с концепцией неопровержимых образцов — то есть таких, сопоставление с которыми всегда успешно. К ним относятся подчеркивание (`_`) и формальные параметры-переменные. Существует еще один класс неопровержимых образцов — *ленивые* образцы (lazy patterns).

Ленивый образец, это любой обычный образец, которому предпослана тильда (`~`). Сопоставление с таким образцом всегда проходит успешно, а динамическое связывание откладывается до момента использования в правой части.

Рассмотрим оператор

```
(***) :: (a -> c) -> (b -> d) -> (a, b) -> (c, d)
(***) f g ~(x,y) = (f x,g y)
```

который принимает две функции обработчика и пару. Функции обработчики применяются к элементам пары: первая к первому, а вторая ко второму. Применяя этот оператор

к каким-то двум функциональным аргументам получим функцию из пары в пару. Если бы образец пары в определении оператора не был бы ленив, такая функция была бы строгой: при передаче в нее расходимости любого рода потребовалось бы сопоставление с конструктором пары, приводящее к расходимости результата.

Однако ленивый образец форсирует сопоставление, только если хотя бы один из элементов пары `(x,y)` фактически потребовался бы при вычислении правой части функции. Поэтому

```
GHCi> (const 1 *** const 2) undefined
(1,2)
```

Про ленивые образцы говорят, что они повышают степень определенности функции (или расширяют ее область определения), элиминируя ряд потенциальных расходимостей.

7.2 Места использования образцов

Мы уже видели, что образцы могут использоваться на глобальном уровне, в левой части объявления функции и в выражении `case ... of ...`. Однако список мест, где они допустимы, не ограничивается указанным набором. Произвольные образцы можно использовать также в левой части `let`-выражений и слева от `->` в лямбда-абстракциях — там, где мы раньше использовали лишь переменные. Также имеется возможность осуществлять сопоставление с образцом в предохранителях.

7.2.1 Образцы в `let`-выражениях

В `let`-выражениях (и конструкциях `where`) можно связывать не только переменные, но и произвольные образцы. Вот примеры, иллюстрирующие эту возможность

```
doubleTail ys = let x:xs = ys
  in xs ++ xs

doubleTail' ys = xs ++ xs
  where x:xs = ys
```

В интерпретаторе

```
GHCi> doubleTail "ABC"
"BCBC"
GHCi> doubleTail' "ABC"
"BCBC"
```

Общие правила трансляции `let` в Kernel достаточно сложны[29], однако для нерекурсивного `let` с одиночным связыванием все достаточно просто:

[29] См. раздел 3.12 Haskell Report для полного набора этих правил.

```
let p = e1 in e0 ≡ case e1 of ~p -> e0
```

Обратите внимание на маркер ленивости. Он приводит к тому, что сопоставление с образцом в `let`-выражениях неопровержимо; следующие примеры демонстрируют это:

```
GHCi> let x:xs = [] in 42
42
GHCi> let x:xs = undefined in 42
42
GHCi> let x:xs = [] in x
*** Exception: Non-exhaustive patterns in x : xs
GHCi> let x:xs = undefined in x
*** Exception: Prelude.undefined
GHCi> answer ys = 42 where x:xs = ys
GHCi> answer []
42
```

Последний вызов показывает, что образцы в конструкции `where` тоже ленивы.

С помощью образцов в `let` можно написать достаточно ленивую реализацию оператора `(***)` из предыдущего раздела, не пользуясь явно ленивыми образцами:

```
(***) :: (a -> c) -> (b -> d) -> (a, b) -> (c, d)
(***) f g p = let (x,y) = p in (f x,g y)
```

7.2.2 Образцы в лямбда-абстракциях

В лямбда-абстракциях тоже допустимо использование не только переменных, но и образцов. Вот пример реализации функции, возвращающей голову списка

```
head''' = \(x:_) -> x
```

Недостаток этой реализации в том, что обработать можно только один образец на один аргумент лямбды[30].

Общее правило трансляции лямбды с образцами в Kernel выглядит довольно просто и естественно

```
\p1 ... pn -> e1 ≡ \x1 ... xn -> case (x1,...,xn) of (p1,...,pn) -> e1
```

Здесь, как обычно, p_i — метапеременные, маркирующие образцы, а x_i — свежие переменные.

[30] Имеется расширение `LambdaCase`, решающее эту проблему. Однако, на мой взгляд, получающиеся при этом лямбды выглядят слишком тяжеловесно.

Это правило обеспечивает лямбдам ту же функциональность, что и обычным функциям — любое сопоставление с образцом начинается после полного применения, частично примененная лямбда является WHNF.

Вот еще один пример образцов в лямбда выражении, на этот раз с функцией двух аргументов

```
GHCi> tst = \(Just x) (Right y) -> x + y
GHCi> :t tst
tst :: Num a1 => Maybe a1 -> Either a2 a1 -> a1
GHCi> tst (Just 5) (Right 7)
12
GHCi> tst (Just 5) (Left 7)
*** Exception: Non-exhaustive patterns in lambda
GHCi> tst Nothing (Right 7)
*** Exception: Non-exhaustive patterns in lambda
GHCi> tst Nothing `seq` 42
42
```

7.2.3 Охранные образцы

В Haskell 2010 синтаксис предохранителей был расширен по сравнению с предыдущим стандартом Haskell 98. В языке появилась возможность использовать образцы в предохранителе, так называемые *охранные образцы* (pattern guards). Рассмотрим пример

```
firstOdd :: [Integer] -> Integer
firstOdd xs
  | Just x <- find odd xs = x
  | otherwise             = 0
```

Если обычный предохранитель должен был иметь булев тип, то теперь он может возвращать значение любого типа (в примере `find odd xs` возвращает `Maybe Integer`), которое после этого сопоставляется с заданным образцом этого типа (в примере — `Just x`). Удачное сопоставление приводит к правой части определения функции, неудачное — к переходу к следующему предохранителю.

```
GHCi> firstOdd [2,3,4]
3
GHCi> firstOdd [2,4,6]
0
```

Охранные образцы могут присутствовать в предохранителях вперемешку с обычными булевыми выражениями:

```haskell
firstOddIsBig :: [Integer] -> Bool
firstOddIsBig xs
  | Just x <- find odd xs, x > 1000 = True
  | otherwise                       = False
```

В интерпретаторе

```
GHCi> firstOddIsBig [2,3,4,1001]
False
GHCi> firstOddIsBig [2,4,1001]
True
```

Охранные образцы — инструмент, чрезвычайно востребованный разработчиками компиляторов и тайпчекеров, которым приходится перебирать довольно большое число синтаксических конструкций и специальных случаев целевого языка. Охранные образцы, существовавшие ранее как расширение, были добавлены в стандарт языка по их просьбе.

8 Типы данных: дополнительные сведения

8.1 Синтаксис записей: метки полей

Для доступа к полям типа-произведения, например,

```haskell
data Point a = Pt a a
  deriving Show
```

приходится использовать специальные лямбды `\(Pt x _) -> x` или `\(Pt _ y) -> y`. Можно было бы вместо анонимных лямбд явно определить именованные функции доступа[31]

```haskell
ptx :: Point a -> a
ptx (Pt x _) = x

pty :: Point a -> a
pty (Pt _ y) = y
```

Для сложных типов с большим количеством полей это утомительная, хотя и несложная работа. Однако в Haskell есть возможность получить ровно ту же функциональность (и даже большую!), просто задав имена для полей типа-произведения. Это делается с помощью механизма *меток полей* (field labels) при определении типа

[31] Иногда для подобных функций используют математический термин *проекции* или термин из ООП *акцессоры*.

```
data Point a = Pt { ptX :: a, ptY :: a }
  deriving Show
```

Типы данных, поля которых снабжены метками, называют *записями* (records).

Метки имееют тип `Point a -> a` и работают точно так же, как и определенные вручную проекции

```
GHCi>  myPt = Pt 3 2
GHCi> ptX myPt
3
GHCi> ptx myPt
3
```

Имеется возможность инициализировать запись не только в стандартном синтаксисе вызова конструктора, но и через метки полей:

```
GHCi> myPt1 = Pt {ptY = 2, ptX = 3}
GHCi> myPt1
Pt {ptX = 3, ptY = 2}
```

Обратите внимание, что порядок полей при инициализации произволен. Можно даже инициализировать не все поля, правда при этом вы получите предупреждение при инициализации и ошибку при попытке доступа к неинициализированному полю

```
GHCi> myPt2 = Pt {ptX = 3}
 warning: [-Wmissing-fields] Fields of `Pt' not initialised: ptY
GHCi> ptX myPt2
3
GHCi> ptY myPt2
*** Exception: Missing field in record construction ptY
```

Помимо стандартное использования в качестве проекций

```
absP p = sqrt (ptX p ^ 2 + ptY p ^ 2)
```

метки полей имеют еще целый ряд стратегий использования. Можно связать их с переменными в образце, вот версия той же самой функции в этом синтаксисе

```
absP' Pt {ptX = x, ptY = y} = sqrt (x ^ 2 + y ^ 2)
```

Может возникнуть вопрос, а в чем преимущество такой техники по сравнению с использованием стандартных образцов

```
absP'' (Pt x y) = sqrt (x ^ 2 + y ^ 2)
```

Кажется, что `absP'` просто многословная версия `absP''`. Однако если полей десяток, а нам нужен доступ только к паре, то синтаксис меток легко позволяет ограничится этой парой, а обычное сопоставление с образцом потребует восьми подчеркиваний для неиспользуемых аргументов.

С помощью меток полей записи можно «обновлять»; вот как это выглядит синтаксически

```
GHCi> myPt3 = Pt {ptX = 7, ptY = 8}
GHCi> myPt3 {ptX = 42}
Pt {ptX = 42, ptY = 8}
```

Ясно, что это неразрушающее обновление: `myPt3 {ptX = 42}` это новая запись, разделяющая с `myPt3` общее поле `ptY`. При этом `myPt3` по-прежнему доступен и неизменен:

```
GHCi> myPt3
Pt {ptX = 7, ptY = 8}
```

Метки полей одного типа могут быть общими в нескольких конструкторах данных для типа суммы произведений:

```
data Homo = Known   { name :: String, male :: Bool }
          | Unknown {                 male :: Bool }
```

```
GHCi> john = Known "John" True
GHCi> stranger = Unknown False
GHCi> male john
True
GHCi> male stranger
False
```

Однако одинаковые метки полей для *разных типов* недопустимы, их область видимости — глобальная. Добавив

```
data Bad = Bad { male :: Bool }
```

мы получим ошибку компиляции: Multiple declarations of 'male'.

Решение о глобальной видимости меток полей, принятое при разработке языка Haskell, часто подвергается критике. Предлагается ряд расширений, решающих эту проблему, но пока они далеки от общепризнаности.

8.2 Синонимы и переименования типов

8.2.1 Синоним типа: объявление `type`

Ключевое слово `type` задаёт *синоним типа*. Классический пример синонима из стандартной библиотеки

```
type String = [Char]
```

Синонимы типов вводятся исключительно для удобства и играют чисто синтаксическую роль. Они позволяют заменять громоздкие типовые конструкции на компактные, скрывая ненужные детали. Синоним и исходный тип полностью взаимозаменяемы[32]:

```
GHCi> str = "Hello" :: String
GHCi> :t str
str :: String
GHCi> length str
5
```

Функция `length` ожидает списка в качестве аргумента, но `str` и есть список, хотя синоним скрывает этот факт.

Синонимы типа могут быть параметризованными:

```
GHCi> type EC = Either Char
GHCi> :kind EC
EC :: * -> *
```

Теперь `EC` выглядит как однопараметрический конструктор типа, и мы можем использовать его в таком качестве.

```
GHCi> type LEC b = [EC b]
GHCi> le = [Right 5, Left 'z'] :: LEC Int
GHCi> :t le
le :: LEC Int
```

Явное приписывание типа с использованием синонимов приводит к тому, что интерпретатор предъявляет именно синоним. Но это поведение не очень устойчиво к преобразованиям, требующим вывода типов

```
GHCi> :t map id le
map id le :: [EC Int]
```

Хотя `map id` имеет тип `[b] -> [b]`, то есть не меняет типа своего аргумента, синоним при такой обработке раскрывается в базовый тип.

[32] За исключением объявления представителя класса типов.

8.2.2 Переименования типа: объявление `newtype`

Ключевое слово `newtype` задаёт *новый тип* с единственным однопараметрическим конструктором, упаковывающий уже существующий тип:

```
newtype AgeNT = AgeNT Int
```

Общий синтаксис объявления `newtype` таков

```
newtype T u1 ... uk = N t
```

Здесь, u_i — метапеременные типа, `t` — упаковываемый тип данных, зависящий от u_i. Конструкторам типа `T` и данных `N` обычно дают одно и то же имя. В подавляющем большинстве случаев объявление `newtype` сопровождается меткой для единственного поля. То есть наш пример удобно переписать так

```
newtype AgeNT = AgeNT { getAgeNT :: Int }
```

В отличие от объявления синонима `type` переименование `newtype` действительно порождает новый тип данных, хотя внутри него лежит уже существующий тип. Приведение от внутреннего типа к упаковывающему всегда должно выполняться явно, для этого служит конструктор данных упаковывающего типа. Приведение в обратную сторону тоже легко обеспечить: если есть метка поля, то она выполняет эту работу; если ее нет, то можно воспользоваться сопоставлением с образцом.

```
GHCi> age = AgeNT 42
GHCi> :t age
age :: AgeNT
GHCi> age
AgeNT {getAgeNT = 42}
GHCi> getAgeNT age
42
```

Может показаться, что особого смысла во введении типа данных с помощью `newtype` нет. Ровно того же поведения мы могли бы добиться, объявив аналогичный тип данных с помощью `data`

```
data    AgeDT = AgeDT { getAgeDT :: Int }
```

Тем не менее между `AgeNT` и `AgeDT` есть важное операционное отличие. Тип данных `AgeDT` задает дополнительный уровень косвенности во время исполнения, в то время как `AgeNT` во время исполнения — это просто `Int`. Это приводит к большей эффективности и лучшей определенности `newtype`. Действительно, реализовав похожие функции

```
fNT (AgeNT n) = 42
fDT (AgeDT n) = 42
```

можно обнаружить следующую разницу в поведении

```
GHCi> fNT undefined
42
GHCi> fDT undefined
*** Exception: Prelude.undefined
```

Функция `fDT` расходится из-за попытки сопоставить во время исполнения расходимость с образцом `(AgeDT n)`. В противоположность этому у функции `fNT` образец `(AgeNT n)` в аргументе существует только статически. Во время исполнения аргумент `fNT` — это переменная `n` типа `Int`, поэтому необходимость форсировать вычисление `undefined` отсутствует.

Приведем несколько простых примеров упаковок `newtype` из стандартной библиотеки:

```
newtype All = All { getAll :: Bool }

newtype Any = Any { getAny :: Bool }

newtype Sum a = Sum { getSum :: a }

newtype Product a = Product { getProduct :: a }

newtype Identity a = Identity { runIdentity :: a }

newtype Endo a = Endo { appEndo :: a -> a }
```

Типы `All` и `Any` упаковывают булев тип для реализации представителя класса типов `Monoid` относительно конъюнкции и дизъюнкции соответственно. Типы `Sum` и `Product` нужны для тех же целей, но не для булева типа, а для числовых типов. Они наделяют числа интерфейсом моноида относительно сложения и умножения. Тип `Identity` — это универсальная упаковка для произвольного типа, а `Endo` — универсальная упаковка для произвольного эндоморфизма. Отметим, что `Sum`, `Product` и `Identity` структурно совершенно идентичны, но используются для разных целей.

8.3 Дополнительные техники

8.3.1 Фантомные типы

Фантомным называется тип данных, в конструкторе типа которого содержится типовая переменная, отсутствующая в правой части объявления. Например,

```
newtype Temperature a = Temperature Double
```

Фантомные типы позволяют на уровне типов хранить дополнительную информацию, используемую при тайпчекинге, но не занимающую места в памяти во время исполнения. Величине `Temperature 42` может быть приписано бесконечное множество разных фантомных типов, хотя во время исполнения это будет просто величина типа `Double`.

Haskell позволяет определять необитаемые или пустые типы данных, то есть такие типы, которые не имеют ни одного значения. Это очень просто сделать, вот два таких типа,

```
data Celsius
data Fahrenheit
```

Их смысл — в использовании вместо параметра `a` в конструкторе типа `Temperature`:

```
comfortTemperature :: Temperature Celsius
comfortTemperature = Temperature 23

c2f :: Temperature Celsius -> Temperature Fahrenheit
c2f (Temperature c) = Temperature (1.8 * c + 32)
```

Нам хотелось бы уметь выполнять над температурами арифметические операции. Этого можно добиться, подключив расширение `GeneralizedNewtypeDeriving`. Оно позволяет использовать механизм производных представителей для класса типов `Num`[33]. Нужно лишь изменить объявление нашего типа

```
newtype Temperature a = Temperature Double
  deriving (Num,Eq,Show)
```

Теперь можно пользоваться арифметикой:

```
GHCi> :t comfortTemperature
comfortTemperature :: Temperature Celsius
GHCi> comfortTemperature
Temperature 23.0
GHCi> comfortTemperature + Temperature 2
Temperature 25.0
```

При этом типы гарантируют, что арифметические операции допустимы только «внутри» каждого конкретного типа температуры

```
GHCi> c2f comfortTemperature
Temperature 73.4
GHCi> :t c2f comfortTemperature
c2f comfortTemperature :: Temperature Fahrenheit
```

[33]Если, конечно, упаковываемый в `newtype` тип сам является представителем `Num`. Для нашего внутреннего типа `Double` это так.

```
GHCi>
Temperature 0.0
GHCi> c2f comfortTemperature - comfortTemperature
 error: Couldn't match type `Celsius' with `Fahrenheit'
```

8.3.2 Форсирование строгости

Флаг строгости `!` в конструкторе данных позволяет форсировать вычисление соответствующего поля. Приведем в качестве примера тип данных комплексных чисел из стандартного модуля `Data.Complex`

```
infix 6 :+
data Complex a = !a :+ !a
```

Говорят, что конструктор данных *строг* по аргументу, которому предпослан флаг строгости. В частности, инфиксный конструктор комплексного числа `(:+)` строг по обоим своим аргументам.

Чтобы пронаблюдать эффект форсирования вычислений, сравним поведение стандартной пары и комплексного числа

```
GHCi> case (1,undefined) of (_,_) -> 42
42
GHCi> case 1 :+ undefined of _ :+ _ -> 42
*** Exception: Prelude.undefined
```

При этом вычисление полей форсируется, только когда форсируется вычисление родительской структуры, поэтому

```
GHCi> case 1 :+ undefined of _ -> 42
42
```

В рамках статической семантики (денотационно) значением `(1,undefined)` служит `(1,`$\perp$`)`, в то время как значением `1 :+ undefined` служит $\perp$.

Имеется расширение языка `BangPatterns`, которое позволяет форсировать вычисление при связывании в образцах:

```
GHCi> :set -XBangPatterns
GHCi> foo !x = True
GHCi> foo undefined
*** Exception: Prelude.undefined
```

Список литературы

[1] Харрисон П. Филд А. *Функциональное программирование.* М.: Мир, 1993.

[2] Бэкус Дж. *Можно ли освободить програмирование от стиля фон-Неймана? Функциональный стиль и соответствующая алгебра программ // Лекции лауреатов премии Тьюринга.* М.: Мир, 1993.

[3] Липовача М. *Изучай Haskell во имя добра!* ДМК Пресс, 2012.

[4] Уилл Курт. *Программируй на Haskell.* ДМК Пресс, 2019.

[5] Vitaly Bragilevsky. *Haskell in Depth.* Manning Publications, 2020.

[6] Simon Marlow (editor). Haskell 2010 language report. 2010.

[7] Will Kurt. *Get Programming with Haskell.* Manning Publications, 2018.

[8] Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide.* No Starch Press, 2011.

[9] Simon Marlow and Simon Peyton-Jones. The glasgow haskell compiler. In Amy Brown and Greg Wilson, editors, *The Architecture of Open Source Applications, Volume II.* Lulu Press, 2012.

Index