Программирование на Hasell

(для любопытных)

Функциональное программирование базируется на довольно сложной математической теории функций. Но основная его идея состоит в том, что в функциональной программе должны содержаться одни только функции и ничего более. При этом функции должны быть «чистыми». Чистой называется такая функция, которая при заданных значениях своих аргументов возвращает всегда один и тот же результат. Иногда в теле функции могут создаваться локальные переменные, но они играют вспомогательную роль для хранения промежуточных данных, при выходе из функции уничтожаются и на чистоту функций не влияют. «Не чистыми» называются функции, имеющие «побочный эффект», который состоит в том, что функции что-то изменяют в самом компьютере или во внешнем мире, или поведение функции зависит от каких-то внешних факторов. Пусть, например, в теле функции используется какая-то глобальная переменная, внешняя по отношению к функции. Это означает, что результат, возвращаемый функцией зависит не только от переданных ей аргументов, но также и от значения посторонней глобальной переменной. Такая функция будет не чистой. К не чистым относятся и функции по обмену информацией с клавиатурой, экраном, печатающим устройством, с файлами. Поскольку без обмена информацией с внешним миром программы не имеют смысла, то это означает, что по-настоящему чистых функциональных программ не существует. Поэтому в дальнейшем будем считать, что функциональным называется такой стиль программирования, при котором в основном программа содержит только чистые функции, а функции с побочными эффектами являются исключением и применяются только в случае крайней необходимости. Я не буду здесь обсуждать преимущества чистого программного кода, эти преимущества весьма существенны, а познакомиться с ними можно практически в любой солидной книге по функциональному программированию.

Существует огромное количество книг по функциональному языку программирования Haskell. В своё время идеи Haskell были новым словом в программировании и вызывали огромный интерес. Теперь

существуют и другие, более продвинутые языки, позволяющие программировать в функциональном стиле, но Haskell по-прежнему занимает главное место в этой области и знакомство хотя бы с основами этого языка совершенно необходимо для любого программиста. Большинство книг по Haskell, да и по другим языкам тоже, или слишком перегружены подробностями и теоретическими рассуждениями, при начальном знакомстве бесполезными, или дают слишком краткое изложение, не позволяющее познакомиться с языком без привлечения дополнительной информации. Попробуем здесь преодолеть эти недостатки и изложить базовые понятия о Haskell в максимально простой форме, избегая общих рассуждений и сведений из теории о функциональном программировании. В основном будем полагаться на примеры, стараясь представить их в максимально простом и понятном виде.

1.Начало работы

Рабочая среда Haskell

Существует две основные реализации Haskell: ***Hugs*** - это интерпретатор, который в основном используется для обучения и ***GHC*** – компилятор, более подходящий для реальных приложений. GHC компилирует программу в машинный код, поддерживает параллельное выполнение и предоставляет полезные инструменты анализа производительности и отладки. По этим причинам здесь мы будем использовать GHC. GHC имеет три основных компонента.

- ***ghc*** - компилятор, который генерирует рабочий файл .exe.
- ***ghci*** - интерактивный интерпретатор и отладчик
- ***runghc*** - программа для запуска программ Haskell в виде скриптов, без необходимости их предварительной компиляции.

Пока, на первом этапе, мы будем использовать в основном ***ghci***, который можно запустить двумя способами: в командной строке или в специальном окне, предоставляющем некоторые дополнительные удобства. Для вызова интерпретатора надо в командной строке напечатать ***ghci***, появится сообщение о версии установленного на вашей машине интерпретатора и приглашение в виде:

Prelude>

Prelude – это название стандартной библиотеки и его вывод означает, что библиотека загружена и может использоваться. При загрузке других библиотек их названия появятся дополнительно, загромождая строку ввода. Чтобы избежать этого, можно приглашению дать другой вид по своему выбору. Создадим максимально краткое приглашение, например в виде ***h>***. Для этого надо выполнить команду:

Prelude> :set prompt h>

Теперь приглашение будет:

h>

Попутно мы увидели, что команды в ghci начинаются с двоеточия.

После приглашения можно вводить команды или код Haskell, который будет немедленно выполняться. Можно сказать, что ghci очень похож, например, на интерпретатор языка Ruby – ***irb***. Для вызова ghci в специальном окне, проще всего перейти в папку winghci и запустить там рабочий файл ***winghci***. В появившемся окне можно делать всё то же, что и в командной строке, только вверху окна имеется меню, которое делает работу удобнее и производительнее.

Интерпретатор ghci позволяет выполнять большой набор команд, полный список которых можно получить командой:

h> ***:?***

Например, команда ***:q*** завершает работу ghci. Загрузить посторонний модуль можно командой ***:module***, например:

h> ***:module + Data.Ratio***

Названия команд в ghci можно сокращать:

h> ***:m +Data.Ratio***

Модуль ***Data.Ratio*** позволяет работать с рациональными дробями:

h> ***(17%3) / (2%5)*** → 85 % 6

Мы выполнили деление одной рациональной дроби на другую. Как видим, для написания рациональной дроби применяется знак ***%***. Круглые скобки нужны для установления старшинства операций (об этом позже). Здесь и везде далее стрелкой → я буду заменять слова «получим», «будет выведено» и тому подобное. Сам ghci этой стрелки не выводит. При копировании какого-то кода для выполнения, надо стрелку и последующий текст убрать. Не буду также далее писать приглашение ***h>***.

Интерпретатор позволяет выполнять арифметические операторы в инфиксной форме, как в этом примере. Но это только синтаксический сахар, поскольку на самом деле Haskell может выполнять только функции, и ничего другого. Поэтому и арифметические операторы можно записывать в виде функций. Например, вместо

2 * 3 → 6

можно написать:

(*) 2 3 → 6

Это надо понимать, как вызов функции ***(*)*** (скобки нужны только для обозначений операторов, подобных этому) с аргументами ***2*** и ***3***.

И наоборот, любую функцию двух переменных можно применить, как инфиксный оператор. Для этого надо идентификатор функции заключить между апострофами (`). Создадим какую-нибудь функцию двух переменных, например:

f x y = (x + y)/(x - y)

Вызовем её, как функцию:

f 4 2 → 3

А теперь, как инфиксную:

4 `f` 2 → 3.0

При определении функций (а также и переменных) в ghci перед их идентификатором надо ставить слово ***let***:

let f x y = (x + y)/(x - y)

Дальше мы с этим разберёмся.

Если в имени функции использованы только символы (не буквы и не цифры), то она автоматически считается инфиксным оператором. При определении её имя надо заключать в скобки:

(%#@) x y = x^2 + y^2

2 %#@ 3 → 13

Операторы сравнения в Haskell обозначаются также, как и в других языках (**==, >, <, <=, >=**), кроме оператора «не равно» - вместо ***!=*** принято ***/=***

0.3 /= 23.2 → True

Булевы значения ***True*** и ***False*** пишутся с большой буквы.

Кроме того, вместо логического отрицания ! в других языках здесь используется слово ***not***:

not False → True

Для возведения в целую степень принят оператор ^:

2 ^ 3 → 8
2.5 ^ 3 → 15.625

При этом оператор ^ также на самом деле функция:
(^) 2.5 3 → 15.625

а для дробной степени применяется ********:
16 ** 0.5 → 4.0
(**) 16 0.5 → 4.0

Имеется полный набор элементарных математических функций. Кроме функции натурального логарифма ***log*** имеется ещё функция логарифма с заданным основанием ***logBase***:
logBase 3 5 → 1.464973520717927

Здесь первый аргумент — основание логарифма.

С помощью команды ***:info*** можно получить указатель старшинства для базовых операторов в виде числа от ***1*** до ***9***:
:info * → infixl 7 *

Это значит, что оператор ******* имеет уровень старшинства ***7***, а слово ***infixl*** означает, что оператор инфиксный и имеет левостороннюю ассоциативность (выполняется слева направо). Для правосторонних операторов, например ^, получим ***infixr***:
:info ^ → infixr 8 ^

Операции более высокого старшинства (приоритета) выполняются раньше. Приоритет всегда можно изменить с помощью круглых скобок:
2 * 3 + 4 → 10
2 * (3 + 4) → 14

ghci имеет одну встроенную константу:
pi → 3.141592653589793

Для других констант приходится вводить локальные переменные с помощью служебного слова ***let*** (так называемое, let-связывание). Например:
let e = exp 1
e → 2.718281828459045

Здесь мы применили библиотечную функцию ***exp***. Как видим, аргумент функции не надо заключать в скобки, хотя это и допускается (не всегда):
let e = exp(1)

Если аргументов несколько, то они разделяются пробелами (не запятыми):
compare 2 3 → LT

Функция ***compare*** сравнивает свои два аргумента по величине (***LT*** — меньше).

Синтаксис, используемый для кода, выполняемого в ghci, иногда отличается от синтаксиса в программах, выполняемых в виде скриптов.

Списки

Теперь, несколько забегая вперёд, приведём некоторые сведения по работе со списками. Это нам потребуется для примеров, рассматриваемых далее. Введём какой-нибудь список в ghci:
[1, 2, 3] → [1,2,3]

Интерпретатор просто возвращает введённый список. Элементы списка могут быть любыми, но только обязательно все одного типа. Пустой список имеет вид ***[]***. Для элементов, представленных числами натурального ряда, можно использовать диапазон:
[1..10] → [1,2,3,4,5,6,7,8,9,10]

Список включает обе конечные точки диапазона. Можно указать шаг в диапазоне просто указав два первых элемента:
[3, 7..25] → [3,7,11,15,19,23]

Числа могут быть и с плавающей запятой:
[4.3, 4.8..6.2] → [4.3,4.8,5.3,5.8,6.3]

Верхняя граница оказалась завышенной. Перечисления для чисел с плавающей запятой иногда могут выдавать сюрпризы, так что ими не стоит злоупотреблять. Числа могут и убывать:
[10,9..1] → [10,9,8,7,6,5,4,3,2,1]

Если не указать верхнюю границу диапазона, интерпретатор попытается создать бесконечный список и потребуется аварийная остановка программы. Далее мы увидим, что бесконечные списки в некоторых ситуациях окажутся полезными.

Оператор конкатенации обозначается ++
["one", "two", "three"] ++ ["Peter", "Anna"] →
["one","two","three","Peter","Anna"]

Для добавления только одного элемента в начало списка применяется оператор (***:***)

77:[1, 2, 3] → [77,1,2,3]

Дальше мы увидим, что этот оператор необычайно полезен.

Функция ***length*** определяет размер списка:

let s = ["one","two","three","Peter","Anna"]
length s → 5

Строки и символы

Текстовая строка заключается в двойные кавычки:

"Number 5 is odd" → "Number 5 is odd"

Escape-символы и правила экранирования Haskell те же, что и во всех других языках. Так, комбинация ***\n*** означает перевод строки:

putStrLn "Heres a newline \n See?" →
Heres a newline
See?

Функция ***putStrLn*** выводит строку с учётом escape-символов. Обычная функция ***print*** выводит текст «как есть».

print "Heres a newline \n See?" → "Heres a newline \n See?"

Кроме ***putStrLn*** есть также и функция ***putStr***, которая не выполняет перевод строки. Обе эти функции принимают только строковые аргументы, для перевода других типов в строки следует пользоваться функцией ***show***:

putStr (show 777) → 777

На самом деле, текстовая строка — это просто список отдельных символов, которые записываются в одинарных кавычках:

let a = ['l', 'o', 't', 's', ' ', 'o', 'f', ' ', 'w', 'o', 'r', 'k']
a → "lots of work"

При этом интерпретатор возвращает обычную строку. Пустая строка записывается ***""*** и это синоним для ***[]***:

"" == [] → True

Для строк применимы операторы ++ и ***:***

"foo" ++ "bar" → "foobar"
'a':"bc" → "abc"

В начало строки можно добавить только символ, и они всегда записываются в одинарных кавычках.

Немного о типах

Имена переменных в Haskell записываются с маленькой буквы, а имена типов — с большой. Команда ***:set*** позволяет изменить некоторое поведение ghci, принятое по умолчанию. Например, команда ***:set +t*** заставляет выводить информацию о типе. Выполним эту команду:

:set +t

А теперь напечатаем текст:

"Hello" → "Hello" it :: [Char]

Интерпретатор сообщает, что некая переменная ***it*** имеет тип ***[Char]***, то-есть, это список с элементами типа ***Char***. Здесь ***it*** вспомогательная переменная, в которой сохраняется введённый текст. Эта переменная удобна тем, что её можно использовать при дальнейших действиях:

it ++ " World!" → "Hello World!"

Haskell позволяет использовать и тип ***String***, который просто синоним для типа ***[Char]***:

("hello" :: String) == ("hello" :: [Char]) → True

Посмотрим, какой тип имеет рациональная дробь:

5%7 → 5 % 7 it :: Ratio Integer

Целые числа в Haskell не ограничены по величине и имеют тип Integer:

5^29 → 186264514923095703125 it :: Integer

Есть также и обычный тип Int (ограниченный по величине).

Команда ***:unset*** отключает введённые ранее команды, например:

:unset +t

отключает вывод типов.

Информацию о типе можно также получить с помощью команды ***:type***:

:type "Hello" → "Hello" :: [Char]

let x = 0.78

:type x → x :: Double (можно ***:t x*** → x :: Double)

Посмотрим, какой тип имеет список с элементами — строками:

let s = ["one", "two", "three"]

:type s → s :: [[Char]]

Значит, этот список состоит из списков с элементами типа ***Char***.

Кроме типа ***Double*** Haskell имеет также и тип ***Float***, но нет особой нужды его использовать. Объявить и инициировать переменную (локальную) нужного типа можно так:
let x = 9 :: Int

2.Типы и функции

Общие замечания

Haskell имеет строгую статическую систему типов, но, как и другие языки, умеет выводить (derive) тип из контекста, так что принудительное задание типа не всегда необходимо. Haskell имеет некоторую поддержку программирования и с динамическими типами, хотя это не так просто, как в языках с динамической системой типов. Как уже могли видеть, при задании типа, тип от значения отделяется знаком ***::***
'***w' :: Char*** → 'w'
[0.5, 2.1, 77.0] :: [Double] → [0.5,2.1,77.0]

При вызове функции надо написать её имя и через пробелы перечислить аргументы:
even 8 → True
compare 7 2 → GT

Функции имеют более высокий приоритет, чем операторы, поэтому можно писать:
compare 7 2 == GT → True

Хотя скобки всегда можно поставить для читабельности:
(compare 7 2) == GT → True

Иногда, конечно, скобки необходимы:
compare (sqrt 3) (sqrt 6) → LT

Списки и кортежи

В Haskell составные данные (коллекции) представлены только двумя видами: списки (list) и кортежи (tuple). Со списками мы уже познакомились, добавим здесь ещё некоторые сведения о них.

Не существует возможности извлекать элементы из списка по индексу, можно только получить первый элемент с помощью функции ***head***:
s = ["aa", "bb", "cc", "dd"]

head s → "aa"

Обычно первый элемент называют головой списка (head). Функция ***tail*** позволяет получит список без первого элемента:

tail s → ["bb","cc","dd"]

Такой список называется хвостом (tail). На первый взгляд кажется удивительным, что с помощью только этих двух функций в функциональном программировании удаётся, как мы увидим далее, делать со списками всё, что угодно. Поскольку строки это тоже списки, то head и tail можно применять и к строкам:

head "hello" → 'h'

tail "hello" → "ello"

Есть ещё функция ***last***, которая позволяет извлечь последний элемент списка:

last ["bb","cc","dd"] → "dd"

last "hello" → 'o'

Эта функция играет не столь большую роль, как две предыдущих.

Кортеж это элементы любых типов, разделённые запятыми и заключённые в круглые скобки:

let t = (77, True, "Hello", 0.75)

Посмотрим на тип такого кортежа:

:type t → t :: (Integer, Bool, [Char], Double)

Пустой кортеж будет ***()***:

let t = ()

Посмотрим на его тип:

:type t → t :: ()

На самом деле это отдельный тип (), имеющий одно значение, тоже (), потом он нам потребуется. В Haskell не используются кортежи с одним элементом. Чаще всего применяются кортежи с двумя элементами, и эти кортежи будем для краткости называть парами.

Функции над списками и кортежами

Кроме функций ***head*** и ***tail*** есть ещё одна пара функций, работающих со списками: ***tak***е и ***drop***. Эти функции принимают два аргумента, а их работу покажем на примере:

take 2 [1,2,3,4,5] → [1,2]

drop 3 [1,2,3,4,5] → [4,5]

Значит, функция ***take*** позволяет извлечь из списка заданное количество начальных элементов, представив их тоже в форме списка, а функция ***drop*** позволяет извлечь элементы, начиная от заданного по номеру (считая от нуля) и до конца списка.

drop 3 ['a', 'b', 'c', 'd', 'e', 'f'] → "def"

Результат также имеет вид списка, ну, а в последнем примере этот список имеет вид текста.

Функция ***reverse*** меняет порядок элементов в списке на противоположный:

reverse [1,2,3,4,5] → [5,4,3,2,1]

Попробуем извлечь предпоследний элемент списка:

head (tail (reverse [1,2,3,4,5])) → 4

Для кортежей имеется две функции, применяемые исключительно только для пар. Функция ***fst*** извлекает первый элемент пары, а функция ***snd*** — второй:

fst (5, "hello") → 5

snd (5, "hello") → "hello"

Кортеж можно использовать, например, для нахождения суммы чисел:

f (x,y) = x + y

f (2,5) → 7

При создании функции в ghci надо, как и для переменных, использовать слово ***let***:

let f (x,y) = x + y

Здесь единственным аргументом функции ***f*** является кортеж. Можно и для большего числа слагаемых:

f (x,y,z) = x + y +z

f (2,3,5) → 10

Типы функций и чистота

В Haskell всё имеет тип и функции тоже. Посмотрим на тип какой-нибудь функции, например ***lines***. Эта функция разбивает строку по управляющему символу ***\n*** и возвращает список:

lines "the quick\nbrown fox\njumps" →
["the quick","brown fox","jumps"]

Определим тип:

:type lines → lines :: String -> [String]

Таким образом тип показывает, что функция принимает одну строку и возвращает список строк. Обычно подобную информацию называют сигнатурой функции. Следовательно, тип функции в Haskell является её сигнатурой. Как увидим далее, сигнатура функций в Haskell играет очень важную роль.

Посмотрим ещё на тип (сигнатуру) какой-нибудь функции с побочным эффектом, например функции ***putStrLn***, которая выводит на терминал строки:

:type putStrLn → putStrLn :: String -> IO ()

Здесь ***IO*** название библиотеки средств ввода — вывода. Сигнатура говорит нам, что функция принимает строку и передаёт её средствам ввода — вывода, а возвращает знакомый уже нам тип ***()***, который можно считать аналогом типу ***void*** в других языках. На самом деле функция ничего не возвращает. Значит, если возвращаемый тип начинается с ***IO***, то это функция с побочным эффектом.

Каррирование функций

Всякая функция с несколькими аргументами может быть приведена к функции, число аргументов у которой на единицу меньше. Эта операция называется каррированием. Она позволяет любую функцию со многими аргументами свести в итоге к функции с одним аргументом. Определим какую-нибудь простейшую функцию с двумя аргументами, например:

```
f :: Int -> Int -> Int
f x y = x + y;
```

Если задать конкретное значение одного аргумента, то получим функцию с одним аргументом. Этой функции можно дать новое имя:

```
g  = f 2;
```

Теперь функцию ***g*** можно вызвать:

g 3 → 5

Посмотрим на тип функции ***g***:

:type g → g :: Int -> Int

Можно не вводить новое имя, а вызвать каррированную функцию так:

(f 2) 3 → 5

:type f 2 → f 2 :: Int -> Int

Можно определить тип и для функции с конкретными значениями всех аргументов:
:type f 2 3 → f 2 3 :: Int

Значит ***f 2 3*** это тоже функция, не имеющая аргументов и возвращающая тип ***Int***.

Композиция функций

В Haskell можно применять композицию функций в обычном стиле:

```
f x = x + 2
g y = y * y
t z = f (g z)
t 2 → 6
```

Есть также синтаксический сахар, позволяющий для композиции функций применять оператор точка (.):

```
f x = x + 2
g y = y * y
h z = (f . g) z
h 2 → 6
```

Если аргументы и результаты функций имеют разные типы, то композиция этих функций возможна только тогда, когда внутренняя вложенная функция возвращает результат того же типа, какой у аргумента внешней функции:

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

Здесь буквами ***a***, ***b*** и ***c*** обозначены параметры типов. Если внешняя функция ***f*** имеет аргумент типа ***b***, то результат вложенной функции ***g*** должен иметь этот же тип ***b***.

Исходные файлы Haskell

Возможности ввода программного кода в интерпретаторе ограничены, поэтому программы приходится помещать в файлы. Создадим простейшую программу, например такую, которая выполняет возведение в степень (аналог оператора **). Для этого в каком-нибудь текстовом редакторе напишем текст:

```
f x y = x ** y
```

и поместим этот текст в файл ***prog.hs***, расположенный там, где нравится. Это и будет наша исходная программа.

Замечание: считаю, что в простейших программах для обучения надо стремиться к максимальной краткости кода. Поэтому я не буду применять длинных идентификаторов, а обозначать названия функций, переменных и т. д. одной-двумя буквами: функции буду чаще всего обозначать буквой ***f***, переменные — буквами ***x, y, z*** и так далее. Только в реальных программах имеет смысл использовать содержательные имена, да и то не всегда. Считаю также, что в коротких примерах в книге нет смысла применять комментарии, всё, что надо можно объяснить в тексте. Обычно на эту мою рекомендацию возражают в том смысле, что мол надо приучать читателя к содержательным идентификаторам и к комментариям. Но ведь к этому давно уже все приучены и всюду можно встретить идентификаторы из 10-15 букв, а то и того больше. По-моему также не следует долго хранить все эти наши короткие программы на диске, а если захотелось их повторно выполнить, то проще всего скопировать текст из руководства. По этой причине я для всех примеров буду применять одно и то же название файла ***prog.hs***, помещая в него разные программы.

Для того, чтобы не возиться с полными именами файлов, лучше всего в командной строке перейти в ту папку, где расположен файл ***prog.hs*** и после этого вызвать ***ghci***. Загрузить файл в интерпретатор можно командой

:load prog.hs

Или, используя сокращение:

:l prog.hs (можно опускать расширение имени: ***:l prog***)

Если всё нормально, получим ответ:

Ok, modules loaded: Main.

Позже разберёмся, зачем тут появилось слово Main, пока не будем обращать на него внимание. Отмечу только, что эту программу prog.hs нельзя скомпилировать в рабочий файл, компилируемая программа должна иметь функцию ***main***, определяющую точку входа. Теперь мы можем только выполнять на интерпретаторе функции, определённые в программе. Интерпретатор сам выводит на экран возвращаемый функцией результат. В частности, мы можем вызвать функцию ***f***:

f 25 0.5 → 5.0

Посмотрим на тип функции ***f***:

:type f → f :: Floating a => a -> a -> a

Текст ***a -> a -> a*** означает, что функция ***f*** принимает два аргумента типа ***a*** и возвращает результат также типа ***a***. Выше говорилось, что идентификатор типа должен начинаться с заглавной буквы, а здесь он обозначен строчной буквой ***a***. Дело в том, что в данном случае под

типом ***a*** понимается любой (неопределённый) тип, который называют обычно «параметром типа». В Haskell действует правило: у конкретного типа идентификатор должен начинаться с заглавной буквы, а у параметра типа — с маленькой. По традиции принято неопределённые типы обозначать буквами алфавита: ***a, b, c…***, хотя это совсем не обязательно. Текст ***Floating a*** обычно называют «контекстом объявления типа», или просто «контекстом», и он указывает, что в данном случае тип ***a*** не совсем произвольный, а должен принадлежать классу ***Floating***, в котором определены методы для чисел с плавающей запятой. С классами мы познакомимся позже. Контекст объявления типа, который отделяется от самой сигнатуры стрелкой =>, накладывает на параметры типа некоторые ограничения. Фактически, в данном случае произвольный тип ***a*** ограничен только типами ***Double*** и ***Float***.

Сигнатуру функции можно задать принудительно, поместив её в начало файла:

```
f :: Double -> Double -> Double
f x y = x ** y
```

Теперь не требуется контекст объявления типа, поскольку объявлен конкретный тип.

Хотя Haskell умеет автоматически выводить тип (сигнатуру) функции, принудительное задание сигнатуры это хороший стиль программирования. Более того, дальше мы увидим, что часто обойтись без сигнатуры просто невозможно.

Кстати, наша программа использует оператор **, на самом деле библиотечную функцию. Возникает вопрос: если функция используют сторонние функции, то есть ли гарантия отсутствия побочных эффектов? По-видимому нет. Чистота будет обеспечена только в том случае, если используемая функция сама не имеет побочных эффектов. Здесь, функция (**) удовлетворяет этому требованию.

Проанализируем тип ещё одной стандартной функции — ***show***. Эта фукция любой свой аргумент трансформирует в строку:

show 2.77 → "2.77"

show True → "True"

Посмотрим на тип:

:type show → show :: Show a => a -> String

Значит, show принимает аргумент произвольного типа, а возвращает всегда строку. Интересно, что здесь появился тип String, хотя на самом деле, как мы уже видели, это то же, что и тип [Char]. Контекст ***Show a*** указывает, что тип аргумента должен принадлежать классу ***Show***, в котором есть методы для трансформации разных типов в строку. С базовыми типами тут всё ясно, ошибки могут возникать при использовании типов «собственного производства».

Интересно будет также посмотреть на тип функции ***fst***:
:type fst → fst :: (a, b) -> a

Эта функция принимает пару элементы которой в общем случае могут иметь разный тип, поэтому параметры типа элементов обозначены ***a*** и ***b***. Результат функции — первый элемент, значит его параметр типа должен быть ***a***. В частном случае оба элемента пары могут иметь один и тот же тип и тогда оба параметра типа будут представлять один реальный тип. Значит, сигнатура всегда имеет в виду не какой-то частный, а общий случай.

Приведу ещё пример компилируемой программы, например для вычисления суммы элементов списка:

```
f :: [Int] -> Int
main = print(f [1,2,3,4,5])
f m = sum m
```

Функция ***main*** определяет точку входа в программу. В нашем случае она вызывает функцию ***f***, передавая ей список из целых чисел. При этом порядок расположения функций ***main*** и ***f*** не имеет значения. Функция ***f*** использует стандартную функцию ***sum***. Если этот код поместить в файл ***prog.hs***, то команда

```
runghc prog
```

выполнит программу и выдаст результат ***15***. Ну, а команда

```
ghc prog
```

создаёт рабочий файл ***prog.exe***, который можно запустить командой ***prog***. Если использовать полные пути для файлов, то эти файлы могут располагаться где угодно.

Условный оператор if

В общем виде ***if*** можно характеризовать так:
if условие ***then*** выражение1 ***else*** выражение2

Если условие даёт True, выполняется ***выражение1***, если ***False*** – ***выражение2***. Оба выражения должны давать результат одного и того же типа. Отсутствие ветви ***else*** недопустимо. Поскольку ***if*** возвращает результат всегда одного типа, то на самом деле этот оператор в целом можно считать выражением. Создадим опять программу в файле prog.hs:

```
f :: Int -> [Char]
f x = if mod x 2 == 1 then "odd x" else "even x"
```

Функция ***mod x y*** возвращает остаток от деления целых чисел ***x*** на ***y***. Загружаем файл:

```
:load prog.hs
```

и вызов функции ***f***:

f 3 → "odd x"

f 8 → "even x"

Имеется синтаксический сахар, который позволяет не писать явно слова ***if*** и ***else***:

```
f :: Int -> [Char]
f x |mod x 2 == 1 = "odd x"
    |mod x 2 /= 1 = "even x"
```

В этом варианте просто перечисляются альтернативные условия, предваряемые вертикальной чертой. Для читабельности можно поставить круглые скобки:

```
f x |(mod x 2 == 1) = "odd x"
    |(mod x 2 /= 1) = "even x"
```

Здесь надо соблюдать правила форматирования кода — вертикальная черта во всех строках должна находиться в одной и той же позиции. Дальше это обсудим отдельно.

Можно применять, например, такие трюки:

```
f x = x > 0
```

Эта функция определяет, является ли число положительным:

f 2 → True

f 0 → False

f (-2) → False

Рекурсия

Haskell не имеет операторов цикла вроде for или while, циклы создаются исключительно с помощью рекурсии. Самый популярный пример рекурсии это вычисление факториала числа. Программу легко написать с использованием ***if***:

```
f :: Int -> Int
f n = if n == 0 then 1 else n * f (n-1)
```

Нетрудно сообразить, что эта функция будет выполнять последовательное умножение: ***n * (n-1) * (n-2) * … * 1***. Когда ***n*** станет равна нулю, выполниться первая ветвь оператора ***if*** и цикл остановится.

Посмотрим на ещё один пример применения ***if***:

```
f :: Int -> [a] -> [a]
f n s = if n <= 0 || null s
    then s
    else f (n - 1) (tail s)
```

Здесь функция ***f*** возвращает последние элементы списка начиная с заданного номера (аналог функции drop).

f 3 [1,2,3,4,5] → [4,5]

В условном выражении ***n <= 0 || null s*** использована логическая операция || (или). Функция ***null*** выдаёт ***True***, если список пуст. Значит, цикл заканчивается, если ***n <= 0*** или список равен ***[]***. В Haskell в операции || второе условие не проверяется, если первое даёт True. Подобное поведение называют «ленивым». В Haskell все вычисления являются ленивыми, позже мы рассмотрим это важное качество подробнее. Пример показывает также, что выражения можно располагать на нескольких строках, надо только делать перенос на новую строку там, где не нарушается смысл выражения.

Пусть задан список, элементы которого представлены парами чисел. Требуется найти сумму разностей этих чисел:

```
f [] = 0
f((x,y) : xs) = (x - y) + f xs
```

f [(3,2), (5,3), (8,4)] → 7

Попробуйте разобраться самостоятельно, что тут происходит.

Сопоставление с образцом.

Как и другие языки, Haskell позволяет применять приём, именуемый «сопоставление с образцом». Обычно он используется в операторах ***case*** и Haskell тоже имеет этот оператор. Но при программировании функций можно обойтись без ***case***, используя «кусочное» задание функции. Вот как это можно сделать для факториала:

```
f :: Int -> Int
f 0 = 1
f n = n * f (n-1)
```

Вызываем функцию:

f 5 → 120

Это выглядит так, как будто мы функцию ***f*** определили дважды. На самом деле здесь аргумент выполняет роль образца по которому происходит вызов первого или второго определения функции. Аргумент ***0*** (в данном случае) обычно называют базой рекурсии — по нему происходит выход из цикла. Рекурсии такого вида имеют один недостаток: все сомножители произведения ***n, (n-1), (n-2),…,1*** сначала сохраняются в памяти и только в конце перемножаются. Это приводит к необоснованному расходу памяти. Преодолевается этот недостаток применением аккумулятора:

```
f :: Int -> Int -> Int
f 0 a = a
f n a = f (n-1) (n * a)
```

Здесь переменная ***a*** — аккумулятор, в котором накапливается произведение без расходования памяти. При вызове функции надо задать начальное значение ***a***, в данном случае оно равно единице:

f 5 1 → 120

Чтобы не вводить значение для ***a*** можно ввести ещё одну функцию:

```
fac :: Int -> Int
fac n = f n 1
f 0 a = a
f n a = f (n-1) (n * a)
```

Теперь мы вызываем функцию ***fac***, которая сама вызывает функцию ***f***, передавая ей начальное значение аккумулятора.

fac 5 → 120

Теперь посмотрим, как сопоставление с образцом применяется в операторе выбора case. Сначала простейший пример:

```
f :: Int -> [Char]
f x = case x of
      0 -> "null"
      1 -> "one"
      otherwise -> "foo"
```

Значит, оператор выбора использует ключевые слова ***case – of***. Аргумент ***x*** сопоставляется с образцами слева от стрелки -> , слово otherwise означает другой, иной и с этим образцом сопоставление всегда даёт True. Слово otherwise можно также заменить на знак _ , (обычно называют «знакозаменитель» или placeholder). Примеры вызова функции:

f 0 → "null"

f 77 → "foo"

Надо иметь ввиду, что все строки сопоставлений могут начинаться с любой, но одной и той же позиции, то-есть в Haskell форматирование текста играет важную роль. Позже это обсудим подробнее.

Теперь напишем программу вычисления факториала с применением case-of:

```
f :: Int -> Int
f n = case n of
      0 -> 1
      _ -> n * f (n-1)
```

Не слишком велика разница с кусочным заданием функции. Здесь тоже можно ввести аккумулятор:

```
 f :: Int -> Int -> Int
f n a = case n of
      0 -> a
      _ -> f (n-1) (n * a)
```

Вызываем функцию: ***f 5 1*** → 120

Для тренировки составим ещё программу, вычисляющую факториалы для диапазона чисел:

```
f :: Int -> Int
f n |n == 0 = 1
   |n > 0 = n * f (n-1)
```

```
fac :: Int -> [Int]
fac n |n <= 0 = []
      |n > 0 = fac (n-1) ++ [f n]
```

Вызываем функцию:

fac 5 → [1,2,6,24,120]

fac (-5) → []

Ещё пара характерных примеров.

Программа, проверяющая отсортирован ли данный список:

```
f [] = True
f [_] = True
f (x : y: xs) = x < y && f (y:xs)
```

При этом пустой список и список с одним элементом мы тут относим к отсортированным.

f [1,2,3,4,5] → True

f [1,7,3,4,5] → False

Есть ещё такой синтаксический сахар:

```
f (x : r@(y:xs)) = x < y && f (r)
```

То-есть, с помощью знака **@** можно списку дать имя (в данном случае ***r***), что позволяет иногда сделать код компактнее.

Во втором примере определим максимальный и минимальный элемент в массиве. Результат в форме кортежа:

```
f [x] = (x,x)
f (x:xs) = ( if x > a then x else a, if x < b then x else b)
         where (a, b) = f xs
```

Два условных оператора if разделяются запятой. Здесь использован очень полезный оператор ***where***, который, как увидим далее, применяется в Haskell очень часто. Смысл его ясен из названия (where переводится «где», «который»).

F [3,1,7,4] → (7,1)

Не так просто сходу понять, как работает эта программа.

3.Определение типов и функции

Определение нового типа данных

Haskell позволяет создавать новые (пользовательские) типы данных с более сложной структурой, чем базовые. Для этого

применяется ключевое слово ***data***, после которого располагается идентификатор нового типа, начинающийся с заглавной буквы, как и у базовых типов. Имя типа принято называть конструктором типа. Создадим хотя бы примитивный пример типа ***Person***, позволяющего хранить анкетные данные. Теперь уместно применить содержательные идентификаторы, иначе новый тип не будет иметь смысла.

```
data Person = Form String Int [String]
        deriving (Show)
p1 = Form "Peter" 25 ["Moskva", "Arbat", "12"]
p2 = Form "Natasha" 18 ["Vologda", "Lenina", "132"]
```

Итак, наш новый тип имеет название ***Person***. Слово после знака равенства (у нас ***Form***), тоже с заглавной буквы, называют именем конструктора значения. Дальше идут так называемые компоненты типа, подобные полям класса в других языках. Будем дальше называть их тоже полями. В конструкторе указываются только типы этих полей. В этом примере поле с типом ***String*** предназначено для хранения имени, с типом ***Int*** — возраста, а в списке ***[String]*** будем хранить адрес. Директива ***deriving (Show)*** задаёт принадлежность нового типа классу ***Show*** для того, чтобы экземпляры типа можно было трансформировать в строку для вывода.

Полученный тип позволяет создавать переменные этого типа с конкретным содержанием полей. Эти переменные похожи на экземпляры класса в других языках. Мы создали два таких экземпляра с идентификаторами ***p1*** и ***p2***. После загрузки программы в ghci можно посмотреть значения ***p1*** и ***p2***:

```
p1 → Form "Peter" 25 ["Moskva","Arbat","12"]
p2 → Form "Natasha" 18 ["Vologda","Lenina","132"]
```

Убедимся, что они имеют тип ***Person***:

```
:type p1 → p1 :: Person
```

Более подробные сведения о типе позволяет получить команда ***:info***

```
:info Person →
data Person = Form String Int [String]  -- Defined at prog.hs:1:6
instance Show Person -- Defined at prog.hs:2:25
```

Посмотрим на сигнатуру конструктора ***Form***:

```
:type Form → Form :: String -> Int -> [String] -> Person
```

В примере использованы разные имена для конструктора типа - ***Person*** и для конструктора значений — ***Form***. Но на практике удобнее использовать для обоих конструкторов одно и то же имя; из контекста всегда ясно, который из конструкторов имеется ввиду в конкретном случае. Имеется также возможность предварительно задать идентификаторы для полей типа, используя ключевое слово ***type***. В итоге будем иметь:

```
type Name = String
type Age = Int
type Address = [String]
data Person = Person Name Age Address
        deriving (Show)
p1 = Person "Peter" 25 ["Moskva", "Arbat", "12"]
p2 = Person "Natasha" 18 ["Vologda", "Lenina", "132"]
```

Теперь команда ***:type Person*** выдаёт сигнатуру конструктора значений:

```
:type Person → Person :: String -> Int -> [String] -> Person
```

Вводимые с помощью ключевого слова ***type*** имена фактически являются синонимами расположенных справа от знака равенства типов.

Новые переменные типа Person можно создавать и в интерпретаторе:

```
let p3 = Person "Anna" 20 ["Rostov", "Sadovaja", "150"]
```

Тип Person можно использовать при создании нового типа. Например, создадим тип ***Employee***, добавив к данным типа Person поле ***Profession*** типа ***String***. Это можно сделать, например так:

```
type Profession = String
data Employee = Employee  Person Profession
          deriving (Show)
e1 =  Employee p1 "doctor"
```

Прочитаем переменную ***e1*** в интерпретаторе:

```
e1 → Employee (Person "Peter" 25 ["Moskva","Arbat","12"]) "doctor"
```

Посмотрим на тип экземпляра e1:

```
:type e1 → e1 :: Employee
```

и на сигнатуру Employee:

```
:type Employee → Employee :: Person -> Profession -> Employee
```

Перечисления (enum)

Перечисления создаются с тем же ключевым словом ***data*** и тоже представляют новый тип. Например:

```
data Roygbiv = Red
             | Orange
             | Blue
             | Green
               deriving (Eq, Ord, Show)
```

Вертикальной чертой обозначаются альтернативы полей; новый тип принадлежит классам ***Eq, Ord*** и ***Show***. Принадлежность к классу ***Eq*** означает возможность проверки на равенство == и /=, а дочерний для ***Eq*** класс ***Ord*** содержит методы сравнения >, <, >=, <=:

```
Red == Blue → False
Red == Red → True
Red > Green → False
Blue > Red → True
```

Проверим тип:

```
:type Red → Red :: Roygbiv
```

Один конструктор типа может содержать несколько альтернативных конструкторов значений, перечисленных через прямую черту |:

```
type V = (Double, Double)
type R = Double
data Shape = Circle V R
           | Polygon [V]
      deriving (Show)
s1 = Circle (2.3, 7.8) 1.3
s2 = Polygon [(0, 0), (2.5, 0), (2.5, 3), (0, 3)]
```

Конструктор типа Shape может использовать любой из двух конструкторов значений: ***Circle*** или ***Polygon***, создавая переменные, представляющие окружность или многоугольник. Окружность задаётся двумя координатами центра окружности (точка на плоскости) и радиусом, а многоугольник — координатами вершин многоугольника. Для представления координат точек использован кортеж ***V***. Переменная ***s1*** содержит параметры для конкретной окружности, а переменная ***s2*** – для прямоугольника:

s1 → Circle (2.3,7.8) 1.3
s2 → Polygon [(0.0,0.0),(2.5,0.0),(2.5,3.0),(0.0,3.0)]

Обе эти переменные имеют один и тот же тип ***Shape***, хотя представляют совершенно разные вещи.
:type s1 → s1 :: Shape
:type s2 → s2 :: Shape

Функции, работающие со списками

Для функций, использующих списки, обычно применяется сопоставление с образцом (pattern matching). Запрограммируем для начала вычисление суммы элементов списка:

```
f:: [Int] -> Int
f [] = 0
f s = head s + f (tail s)
```

Рекурсивный цикл заканчивается, когда список становится пуст.
f [1,2,3,4,5] → 15

(Для вычисления суммы списка есть библиотечная функция ***sum***)

Имеется синтаксический сахар для функций ***head*** и ***tail***, позволяющий сделать код более кратким и выразительным. Ранее мы отмечали, что для вставки элемента в начало списка применяется оператор, обозначаемый двоеточием:
9:[1,2,3] → [9,1,2,3]

Этот оператор позволяет также разделить любой список на голову и хвост:

```
let a:b = [1,2,3,4,5]
```

a → 1
b → [2,3,4,5]

По традиции в Haskell принято обозначать голову буквой ***x***, а хвост — буквами ***xs***. Используя этот «сахар» предыдущий пример можно представить так:

```
f:: [Int] -> Int
f [] = 0
f (x:xs) = x + f xs
```

Посмотрим на более содержательный пример — создадим функцию, позволяющую извлекать элементы списка по индексу:

```
f :: [Int] -> Int -> Int
```

f (x:xs) 0 = x
f (x:xs) n = f xs (n-1)

Применим эту функцию:

f [5,1,7,9,4,12] 3 → 9

Ещё один, более замысловатый пример. Пусть требуется извлечь из списка все элементы, представленные чётными числами:

```
f :: [Int] -> Int-> [Int]
f s 0 = if even (last s) then [last s] else []
f (x:xs) n = if even x then [x] ++ f xs (n-1) else f xs (n-1)
res :: [Int] -> [Int]
res s = f s ((length s)-1)
```

Пришлось ввести две функции ***f*** и ***res*** — выше мы уже применяли этот приём при вычислении факториалов для диапазона.

res [2,3,6,4,7,9,8] → [2,6,4,8]

Любую программу на Haskell можно составить многими способами.

Составим список из всех чётных чисел для заданного числа:

```
f n = if n == 0 then []
      else case even n of
           True -> f (n - 2) ++ [n]
           otherwise -> f (n-3) ++ [n - 1]
```

f 7 → [2,4,6]

f 12 → [2,4,6,8,10,12]

Создадим функцию принимающую список чисел и возвращающую список их кубов:

```
f [] = []
f (x:xs) = x ^ 3 : f xs
```

f [1,2,3,4,5] → [1,8,27,64,125]

Для заданного числа ***n*** выдать ***[1,2,3...n]***:

```
f n =
      let g m = if m == n then [m] else m : g (m + 1)
      in g 1
```

f 5 → [1,2,3,4,5]

Здесь локальная функция ***g*** использует переменную ***n***, хотя явно она туда не передаётся. В примере использован оператор ***in*** столь же полезный, как и ***where***.

Работа с кортежами

Написать функцию, извлекающую из кортежа элементы по их идентификаторам очень просто:

f (a,b,x:xs,c) = (b, xs)

f (0.2, "hello", [1,2,3,4,5], 77) → ("hello",[2,3,4,5])

Посмотрим, что за тип у этой функции:

:type f → f :: (t, t3, [t2], t1) -> (t3, [t2])

Для параметров типа вместо букв a, b, c… компилятор применил букву t с числовым номером, видимо намекая на то, что имеем дело с кортежем — tuple. Интересно, что номера эти расположены не по порядку. Как видим, для элемента в виде списка извлекать можно голову ***x*** и хвост ***xs***. Поскольку список можно представить, например, в таком виде: ***x1:x2:x3:xs***, то можно делать и такие вещи;

f (a,b,x1:x2:x3:xs,c) = (b, x3, xs)

f (0.2, "hello", [1,2,3,4,5], 77) → ("hello",3,[4,5])

Если аргумент функции представлен кортежем, в котором есть элементы с конкретными значениями, то эти значения надо просто повторить при вызове функции:

f (True, a, x:xs, 5) = (a, x)

f(True, "hello", [1,2,3,4], 5) → ("hello",1)

Если вместо заданных указать другие значения, будет ошибка.

f(True, "hello", [1,2,3,4], 77) — так делать нельзя.

Этот же приём можно применить для извлечения данных из экземпляров составных пользовательских типов. Возьмём, например созданный ранее нами тип Person:

```
type Name = String
type Age = Int
type Address = [String]
data Person = Person Name Age Address
        deriving (Show)
p1 = Person "Peter" 25 ["Moskva", "Arbat", "12"]
f1 (Person a b c) = a
f2 (Person a b c) = b
f3 (Person a b c) = c
```

Выполняем извлечение:

f1 p1 → "Peter"

f2 p1 → 25
f3 p1 → ["Moskva","Arbat","12"]

Как видим, в кортеже — аргументе надо применить конструктор значений (Person). Посмотрим на типы этих функций:
:type f1 → f1 :: Person -> Name
:type f2 → f2 :: Person -> Age
:type f3 → f3 :: Person -> Address

При извлечении элементов этим способом идентификаторы не нужных элементов можно заменить на placeholder (иногда ещё называют wild card):

```
f1 (Person a _ _) = a
f2 (Person _ b _) = b
f3 (Person _ _ c) = c
```

Создание подобных функций для извлечения данных несколько громоздко и запутанно. К счастью, Haskell позволяет использовать так называемый ***Record syntax*** (синтаксис записи). В этом случае функции доступа к элементам создаются одновременно с созданием самого типа, а идентификаторами функций являются сами названия полей. Снова создадим тип Person, применяя этот Record syntax. Теперь это будет выглядеть примерно так:

```
data Person = Person {
        name :: String,
        age :: Int,
        address :: [String]
        } deriving (Show)
p1 = Person "Peter" 25 ["Moskva", "Arbat", "12"]
```

Введённые нами названия полей теперь можно использовать, как функции, а в качестве аргумента к ним надо передать идентификатор экземпляра типа:
name p1 → "Peter"
age p1 → 25
address p1 → ["Moskva","Arbat","12"]

Посмотрим на тип этих функций:
:type name → name :: Person -> String
:type age → age :: Person -> Int
:type address → address :: Person -> [String]

При этом мы обошлись без создания типов ***Name, Age, Address***.

Сообщения об ошибках

Стандартная функция ***error*** принимает строку, которая и будет выведена на терминал при запуске функции:

```
f (x:xs) = if null xs
           then error "list too short"
           else head xs
```

f [5,2,7,9] → 2

f [5] → *** Exception: list too short

Если посмотреть на сигнатуру функции ***f***, то увидим, что результат имеет тот же тип, что и у элементов списка:

:type f → f :: [a] -> a

Мы уже знаем, что обе ветви оператора ***if*** должны иметь одинаковый тип, а у нас первая ветвь выводит текст, а вторая — элемент списка. Посмотрим на сигнатуру функции error:

:type error → error :: [Char] -> a

Получается, что error возвращает результат произвольного типа. Дело в том, что тип этого результата автоматически подстраивается под тип результата второй ветви и это устраняет противоречие.

Кстати, с использованием placeholder нашу функцию можно представить в такой лаконичной форме:

```
f (_:x:_) = x
f _       = error "list too short"
```

Здесь ***f _*** надо понимать, как вызов функции ***f*** с любым аргументом.

Кроме выдачи текста, функция ***error*** выполняет аварийный выход из программы. Естественно, что нам иногда хотелось бы продолжить работу программы с учётом сложившейся ситуации. Значит, нам надо иметь возможность получить одинаковый тип результата в обоих ветвях оператора ***if*** каким-то другим способом. Это можно сделать с использованием предлагаемого Haskell особого типа ***Maube***. Этот тип представляет такие переменные, которые могут иметь значение или не иметь его вовсе, например, это может быть пустая клетка в базе данных. Для представления существующего значения используется ключевое слово ***Just***, после которого может быть значение любого типа, например:

let x = Just 2.78
x → Just 2.78
let y = Just "Hello"
y → Just "Hello

Отсутствующее значение обозначается словом ***Nothing***:

let z = Nothing
z → Nothing

Все эти переменные имеют тип Maube:

:type x → x :: Maybe Double
:type y → y :: Maybe [Char]
:type z → z :: Maybe a

Буквой ***a*** обозначен произвольный тип. С использованием типа Maube можем написать:

```
f (_:x:_) = Just x
f _       = Nothing
```

Вызывая эту функцию, получим такие результаты:

f [6,3,9,1] → Just 3
f [8] → Nothing

Теперь мы можем отреагировать на результат Nothing, предприняв нужные действия и продолжить работу программы, например, так:

```
f (_:x:_) = Just x
f _       = Nothing
fi :: [Int] -> Int
fi y = if (f y) == Nothing then 0 else head (tail y)
```

Здесь мы отсутствующий результат заменяем на ноль:

fi [1,2,3] → 2
fi [8] → 0

Для преобразования типа Maube в обычный тип нужна специальная функция, например, такая:

```
r (Just a) = a
```

Тогда мы можем написать:

```
r (Just a) = a
f (_:x:_) = Just x
f _       = Nothing
fi :: [Int] -> Int
fi y = if (f y) == Nothing then 0 else r (f y)
```

Для трансформации типа Maube к типу String подходит такая функция:

```
f :: Maybe [Char] -> [Char]
f x =  case x of
        Nothing -> ""
        Just n  -> n
```

А для чисел (Int или Double) такая:

```
f :: Num a => Maybe a -> a
f x =  case x of
        Nothing -> 0
        Just n  -> n
```

Хеш (Hash)

В Haskell не используется структурный тип Hash (ассоциированный список, словарь, Map) в том виде, как это существует в других языках программирования. Вместо этого применяется список с элементами в виде кортежей — пар, например:

```
let h = [(1, "one"), (2, "two"), (3, "three"), (4, "four")]
```

Первые элементы пар могут рассматриваться, как ключи (***key***) , а вторые — как значения (***value***) хеша. Тип как ключей, так и значений может быть любым. Для извлечения значения по ключу имеется встроенная функция ***lookup***, посмотрим на её тип:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Значит, ключ должен принадлежать классу ***Eq***, то-есть должен допускать операции сравнения. Функция lookup принимает два аргумента: первый — значение ключа, второй — ассоциированный список и возвращает значение типа Maybe:

lookup 2 h → Just "two"

При попытке извлечь значение при несуществующем ключе получим ***Nothing***:

lookup 7 h → Nothing

Как обычно, при извлечении значения из типа Maybe приходится применять специальную функцию, например такую, как мы уже применяли ранее:

```
r (Just a) = a
```

Теперь можно сделать например так:

```
fi :: Eq a => a -> [(a,String)] -> String
```

```
h = [(1, "one"), (2, "two"), (3, "three"), (4, "four")]
r (Just a) = a
fi x h = if (lookup x h) == Nothing then "No" else r (lookup x h)
```

Попробуем применить:

fi 3 h → "three"
fi 7 h → "No"

Впрочем, можно придумать и другие варианты. Функция ***lookup*** совсем несложная, мы можем сами написать её аналог:

```
f :: Eq a => a -> [(a, b)] -> Maybe b
f _ [] = Nothing
f x ((a,b):xs) =
   if x == a
     then Just b
     else f x xs
```

Протестируем:

```
let h = [(1, "one"), (2, "two"), (3, "three"), (4, "four")]
```

f 4 h → Just "four"
f 7 h → Nothing

Использование локальных переменных

При необходимости в теле функции можно создавать локальные переменные. Иногда это позволяет избежать повторений кода, сделать программу более читабельной и так далее. Ранее я уже упоминал, что для определения локальной переменной применяется слово ***let***, а слово ***in*** указывает, где заканчивается блок для let и начинается функция, в которой локальные переменные используются..
Посмотрим на примере:

```
f :: Int -> Int -> Maybe Int
f x y = let r    = 100
            n = y - x
      in if y < r then Nothing else Just n
```

Этому примитивному примеру можно приписать какое-нибудь смысловое содержание. Например, ***y*** можно рассматривать, как размер вклада в банке, ***x*** – снимаемая с вклада сумма, а ***r*** – страховой резерв вклада, который нельзя уменьшить. Функция возвращает остаток вклада.

f 25 120 → Just 95
f 25 80 → Nothing

В примере с одним словом ***let*** определены две переменные ***r*** и ***n***. При этом надо иметь ввиду, что иногда в программах на Haskell отступы не произвольны, а должны соответствовать неким требованиям. В данном случае идентификатор ***n*** должен находиться строго под идентификатором ***r***. Можно также повторять слово ***in***, создавая цепочку, и тогда расположение отступов не имеет значения:

```
f x y = let r = 100
   in let n = y - x
        in if y < r then Nothing else Just n
```

Иногда внутри блока, в котором используется локальная переменная, может быть создана ещё одна локальная переменная. Ничто не препятствует тому, чтобы вложенная переменная имела тот же идентификатор, что и внешняя:

```
f = let x = 25
     in ((let x = "Hello" in x ++ " world"), x)
```

Здесь объявлены две разные переменные, обозначенные одной и той же буквой ***x***. «Внутренняя» переменная временно маскирует «внешнюю», никак на неё не влияя.

f → ("Hello world",25)

В следующем примере локальная переменная обозначена той же буквой ***a***, что и аргумент функции ***f***.

```
f a = let a = "Hello, " in a ++ "Marta"
```

Пример этот искусственный, поскольку здесь аргумент функции никак не используется, однако компилятор не фиксирует ошибку, а функции при вызове можно передать значение любого типа:

f 12 → "Hello, Marta"
f "hhhhh" → "Hello, Marta"

Это отражено и в сигнатуре функции:

:type f → f :: t -> [Char]

Есть ещё и другой способ объявления локальной переменной, при котором вместо ***let*** применяется слово ***where***. Применяемый при этом синтаксис покажем на том же примере, что был рассмотрен ранее:

```
f x y = if y < r then Nothing else Just n
   where r    = 100
         n = y — x
```

Здесь опять надо выполнять требование об отступах — при объявлении двух или более переменных при одном ***where*** их идентификаторы должны строго располагаться в одной и той же позиции.

Точно также, как локальные переменные, можно объявлять и локальные функции, например:

```
f :: String -> [Int] -> [String]
f w m = map g m
   where g 0 = "no " ++ w ++ "s"
         g 1 = "one " ++ w
         g n = show n ++ " " ++ w ++ "s"
```

Здесь использована функция ***map*** (подобные функции обычно называют итераторами), которая принимает два аргумента: функцию ***g*** и список ***m***. Итератор ***map*** применяет функцию к каждому элементу списка, возвращая результат в виде нового списка. При создании локальной функции ***g*** использован обычный приём сопоставления с образцом.

f "apple" [0,1,2,3] → ["no apples","one apple","2 apples","3 apples"]

Напоминаю, что функция show трансформирует свой аргумент к типу String. Кроме ***map*** Haskell имеет и другие итераторы.

В своей программе в любом месте мы можем объявить переменную традиционным способом, например:

```
x = 0.875
```

Но компилятор буде рассматривать ***x***, как функцию без аргументов, возвращающую число ***0.875***. В этом легко убедиться, посмотрев на тип:

:type x → x :: Double

Значит, в программе надо и использовать ***x***, как функцию.

Анонимные (безымянные) функции

Как и другие языки, Haskell позволяет использовать анонимные функции. Синтаксис продемонстрируем на примере:

```
\x -> x * x
```

Здесь функция возводит свой аргумент в квадрат. Например, эту анонимную функцию можно применить для итератора ***map***. Выполним в интерпретаторе ghci:

map (\x -> x * x) [1,2,3,4,5] → [1,4,9,16,25]

Кроме map есть ещё и итератор ***flip map***, у которого аргументы переставлены местами (слово flip переводится перевёрнутый):

flip map [1,2,3,4,5] (\x -> x * x) → [1,4,9,16,25]

Анонимной функции всегда можно присвоить имя:

```
f = \x -> x * x;
f 5 → 25
```

Анонимные функции могут иметь и больше одного аргумента:

```
f = \ x y -> x + y
f 2 3 → 5
```

В ghci анонимную функцию можно создать и вызвать, например, так:

```
(\ x y -> x + y) 2 3 → 5
```

Ещё о правилах для отступов в программах

Многие современные языки не накладывают никаких ограничений на расположение кода в программе, а Haskell, наоборот, хотя и не повсеместно, но требует соблюдения правил форматирования. Назовём хотя бы основные из этих правил.

При объявлении функции её идентификатор можно разместить в любой позиции в строке, но если будут ещё и другие объявления функций, строго требуется чтобы они располагались в той же позиции, что и предыдущая иначе компилятор выдаст ошибку.

```
g :: Double -> Double
g x = x * x
f :: Double -> Double -> Double
f a b = a + g b
```

Сигнатура функции тоже должна находиться в той же позиции.

```
f 2.0 3 → 11.0
```

Недопустим, например, такой выриант:

```
g :: Double -> Double
g x = x * x
   f :: Double -> Double -> Double
   f a b = a + g b
```

Как уже говорилось, идентификаторы локальных переменных в одном блоке ***let*** должны располагаться в одной позиции. Однако, это

не касается расположения разных блоков ***let***. Где угодно могут стоять и блоки ***in***:

```
f = let b = 2
        c = True
in let a = b
           in (a, c)
```

В таком варианте всё будет работать:

f → (2,True)

В переключателе ***case*** образцы для сопоставления тоже должны стоять в одной позиции:

```
f x y =
   case y of
    Nothing -> x
    Just v -> v
```

Здесь показан ещё один вариант перевода из типа Maube в обычный тип.

f 0 (Just 23.8) → 23.8

f 0 Nothing → 0

Все эти ограничения основаны на том, что если требуемое расположение не выполняется, то следующая строка всегда рассматривается, как продолжение предыдущей. Авторы языка считают эти ограничения достоинством, поскольку улучшают порядок и читабельность. Мне это кажется сомнительным. К счастью, ограничения можно снять, заключая блоки в фигурные скобки и разделяя выражения точками с запятой:

```
f = let a = 1
        b = 2
        c = 3
       in a + b + c

g = let { a = 1;  b = 2;
       c = 3 }
     in a + b + c
```

Оба варианта в этом примере абсолютно идентичны:

f → 6

g → 6

Охрана (Guard) при сопоставлении с образцом

Для проверки дополнительных условий при сопоставлении с образцом можно применять такой синтаксис:

```
f x y
    | x <= 0 = Nothing
    | x > r = Nothing
    | otherwise = Just n
  where r    = 100
        n = y — x
```

Как обычно, при задании локальных переменных с помощью where идентификаторы ***r*** и ***n*** должны находиться в одной позиции.

f 20 80 → Just 60
f 0 100 → Nothing
f ***120 300*** → Nothing

Здесь можно было бы применить и уже знакомый нам приём при кусочном задании функции:

```
f x _ |x <= 0 = Nothing
f x _ |x > 100= Nothing
f x y = Just (y — x)
```

Мне такой вариант больше нравится.

Приведём ещё рассмотренный нами ранее аналог функции ***drop***, теперь написанный с использованием *Guard:*

```
f n xs | n <= 0 = xs
f _ []         = []
f n (_:xs)     = f (n - 1) xs
```

Проверим:

f 3 [1,2,3,4,5,6,7] → [4,5,6,7]

4.Классы типов

Определение класса типов

В общей форме это определение можно сформулировать так: класс типов определяет набор функций, которые могут иметь разные

реализации в зависимости от типа аргументов, получаемых этими функциями. Обычно от подобных общих формулировок мало толку, будем разбираться на конкретных примерах.

Ранее мы уже встречали применение классов типов, они присутствуют иногда в сигнатурах функций, например мы уже видели такую сигнатуру:

f :: Floating a => a -> a -> a

Это значит, что функция ***f*** принимает два аргумента произвольного, но принадлежащего классу ***Floating*** типа, а само это ограничение называется контекстом объявления типа (или просто контекстом). Создадим для примера такую функцию:

f x y = if x == y then "x == y" else "x /= y"

И посмотрим на её тип:

f :: Eq a => a -> a -> [Char]

Здесь контекст требует, чтобы произвольный тип ***a*** принадлежал классу ***Eq***, в котором определён тип, допускающий сравнение на равенство (==).

Внешне классы типа похожи на классы в объектно ориентированном программировании и объявляются они также со словом ***class***, после которого идёт название класса, а затем перечень функций. Как увидим далее, можно создавать также и экземпляры классов типов. Но фактически классы типов это совсем не то, что обычные классы в ООП. Попробуем теперь объявить класс типа, вроде упомянутого выше класса ***Eq***. Назовём наш класс, например, MyEq:

```
class MyEq a where
    f :: a -> a -> Bool
```

Это значит, что класс типа ***MyEq*** принимает тип ***a*** и имеет одну функцию, принимающую два аргумента этого типа и возвращающую результат типа ***Bool***. Можно этот код загрузить в ghci и посмотреть на тип функции ***f***:

f :: MyEq a => a -> a -> Bool

Как видим, контекст требует, чтобы тип ***a*** принадлежал классу ***MyEq***. Посмотрим теперь на следующий код:

```
data Color = Red
           | Blue
           | Green
```

```
class MyEq a where
   f :: a -> a -> Bool
instance MyEq Color where
   f Red Red = True
   f Green Green = True
   f Blue Blue = True
   f _ _ = False
```

Здесь мы создали новый тип ***Color*** с помощью enum (перечисления) и создали экземпляр (***instance***) класса ***MyEq***, в котором определили конкретный вид функции ***f***. Теперь мы можем с помощью этой функции сравнивать на равенство значения типа ***Color***:

f Red Red → True

f Red Blue → False

На самом деле в конкретных программах создание типа, объявление класса типов и создание экземпляра находятся в разных модулях и, соответственно, в отдельных файлах. Там, где нужно модули загружаются директивой ***import***. Но мы здесь и дальше будем для удобства всё помещать в один файл.

Мы даже можем оператор сравнения сделать в инфиксной форме, надо только использовать для этого не буквы, а специальные символы. Например, мы можем обозначить этот оператор парой знаков ##:

```
class MyEq a where
   (##) :: a -> a -> Bool
instance MyEq Color where
   Red ## Red = True
   Green ## Green = True
   Blue ## Blue = True
   _ ## _ = False
```

Теперь сравнение будет выглядеть так:

Red ## Red → True

Red ## Green → False

Конечно, в этом примере в экземпляре класса мы просто перебрали все возможные значения полей типа Color и вообще-то могли бы обойтись без всяких классов типов, просто объявив функцию f или (##). Теперь посмотрим на ещё один более показательный пример. Пусть мы хотим получить такой тип классов, который позволил бы

создать функцию, определяющую, будут ли два значения соседями. При этом соседями будем считать числа, абсолютная разность которых не превышает единицы, или буквы, находящиеся рядом в алфавите. Назовём этот класс типов словом ***Sosed***:

```
import Data.Char(ord)
class Sosed a where
        f :: a -> a -> Bool
        g :: a -> a -> Bool
        f x y = (g x y) || (g y x)
instance Sosed Integer where
        g x y = ((x - y) == 1)
instance Sosed Char where
        g x y = ((ord(x) - ord(y)) == 1)
```

Убедимся, что функция ***f*** работает для чисел:

f 1 2 → True

f 3 1 → False

И для букв:

f 'b' 'a' → True

f 'a' 'c' → False

Здесь функция ***ord*** из импортированного модуля ***Char*** возвращает цифровое представление для букв. Создав два экземпляра класса Sosed, мы сделали функцию f универсальной — она работает и для чисел и для букв. Чтобы не применять здесь функцию ***abs*** мы ввели функцию ***g*** и такое определение для **f**:

```
f x y = (g x y) || (g y x)
```

Встроенные классы типов

Haskell имеет несколько встроенных (стандартных) классов типа, например ***Eq, Ord, Show, Read*** и другие. Встроенные классы созданы так, что позволяют с помощью директивы ***deriving*** присоединять разные типы к этим классам и пользоваться их функциями. Сделаем это для нашего типа ***Color*** и класса ***Eq***, содержащего функции проверки на равенство (==) и на неравенство (/=):

```
data Color = Red
           | Orange
           | Blue
```

```
        | Green
        deriving (Eq)
f :: Color -> Color -> Bool
f x y = x == y
g x y = x /= y
```

Применив стандартный класс, мы избавились от необходимости создавать экземпляр и функцию для сравнения всех полей перечисления Color, как это мы делали ранее.

f Red Red → True

g Red Blue → True

Применим теперь этот приём к созданному ранее типу ***Person*** для проверки на равенство по какому-нибудь полю этого типа. Например мы можем проверять, являются ли персоны ровесниками, используя поле ***Age***:

```
type Name = String
type Age = Int
type Address = [String]
data Person = Person Name Age Address
        deriving (Eq, Show)
g :: Person -> Age
g (Person _ t _) = t
f :: Person -> Person -> Bool
f x y = (g x) == ( g y)
p1 = Person "Peter" 25 ["Moskva", "Arbat", "12"]
p2 = Person "Natasha" 25 ["Vologda", "Lenina", "132"]
p3 = Person "Anna" 18 ["Moskva", "Arbat", "12"]
```

Теперь мы можем применять функцию ***f*** к экземплярам типа ***Person***:

f p1 p2 → True

f p1 p3 → False

Можно выполнять сравнение и по полю ***Address*** изменив следующие строки кода:

```
g :: Person -> Address
g (Person _ _ t) = t
```

Проверим, живут ли персоны в одно и той же квартире:

f p1 p2 → False

f p1 p3 → True

Класс ***Ord*** аналогичен классу ***Eq*** и содержит инфиксные функции **>, <, >=, <=**.

Очень полезен и часто используется класс типов ***Show***. Этот класс имеет функцию ***show***, которая позволяет преобразовывать значения разных типов, включая пользовательские, к типу ***String***. Посмотрим в интерпретаторе:

show 9 → "9"

show [1,2,3] → "[1,2,3]"

Оператор ***putStrLn*** выводит результат функции ***show*** без кавычек:

putStrLn (show [1,2,3]) → [1,2,3]

а оператор ***print*** – с кавычками:

print (show [1,2,3]) → "[1,2,3]"

Ранее мы уже присоединяли созданный нами тип Person к классу ***Show*** с помощью директивы ***deriving***. После этого мы можем выводить на терминал экземпляры своего типа, используя функцию ***show***:

putStrLn (show p1) → Person "Peter" 25 ["Moskva","Arbat","12"]

Класс ***Read*** по существу противоположен классу **Show**. Функция ***read*** класса ***Read*** принимает строку и возвращает результат нужного нам типа.

:type read → read :: Read a => String -> a

Конкретно это делается следующим образом:

```
let x = "25.078"
let y = (read x) :: Double
print (y * 2) → 50.156
```

Значит, при использовании ***read*** надо указывать тип, к которому требуется преобразовать строку.

5.Воод-вывод

Работа с терминалом и клавиатурой

Для вывода на терминал Haskell предлагает две функции: ***print*** и ***putStrLn*** (***putStr***), которые мы применяли уже достаточно широко. Посмотрим ещё раз на их сигнатуру:

:type print → print :: Show a => a -> IO ()

:type putStrLn → putStrLn :: String -> IO ()

Значит, ***print*** принимает аргумент произвольного типа, но принадлежащего классу типов ***Show*** и возвращает тип ***IO ()***. Мы уже отмечали, что ***()*** представляет пустой кортеж, а на самом деле это аналог типу Void в других языках, то-есть, print ничего не возвращает и применяется только для побочного эффекта. Кстати, если вызвать тип функции с заданным аргументом, то получим:

:type putStrLn "hello" → putStrLn "hello" :: IO ()

Позже нам это потребуется.

Основная функция для вывода в Haskell это ***putStrLn***, которая принимает только тип ***String***. Следовательно, очень часто ***putStrLn*** используется совместно с функцией ***show***. В передаваемом функции ***putStrLn*** тексте можно выполнять конкатенацию:

putStrLn ("Welcome to Haskell, " ++ "Vanja")

Можно опускать скобки, поставив знак ***$*** впереди:

putStrLn $ "Welcome to Haskell, " ++ "Vanja"

Нет возможности выполнять интерполяцию в тексте, приходится использовать функцию ***show*** и конкатенацию:

putStrLn $ "Result = " ++ show (2 * 9) ++ " years" → Result = 18 years

То же делает и print, только текст будет в кавычках:

print $ "Result = " ++ show (2 * 9) ++ " years" → "Result = 18 years"

Возвращаемое функциями вывода значение ***IO ()*** можно присвоить какой-нибудь переменной. Поскольку в Haskell применяются только ленивые вычисления, то никакого вывода не терминал не произойдёт:

let f = putStrLn "hello"

Вывод будет только когда будет действие (actions), например, вызов функции ***f***:

f → hello

Эти actions программируются в Haskell путём применения так называемого блока ***do***, но об этом чуть позже.

Для ввода с клавиатуры применяется директива ***getLine*** со специальным синтаксисом:

x <- getLine

При этом возникнет режим ожидания ввода с клавиатуры, например, мы можем напечатать:

Peter

Переменная ***x*** будет инициирована текстовым значением ***"Peter"***:

putStrLn x → Peter

Обратная стрелка (<-) здесь служит для преобразования типа. Дело в том, что функция ***getLine*** возвращает результат типа ***IO String***:
:type getLine → getLine :: IO String

Такой тип принадлежит так называемой монаде IO. О монадах немного поговорим позже. Тип IO String нельзя использовать, например, для функции print или PutStrLn, аргумент этих функций должен иметь тип String. Стрелка как раз и трансформирует тип к нужному:

x <- getLine

Вводим с клавиатуры:"hello"

:type x → x :: String

Теперь мы можем запрограммировать, например, такой диалог:

```
main = do
  putStrLn "Greetings!  What is your name?"
  x <- getLine
  putStrLn $ "Welcome to Haskell, " ++ x ++ "!"
```

Это полноценная программа, которую можно скомпилировать и получить рабочий файл ***.exe***. Как видим, в программе должна присутствовать традиционная функция ***main***, определяющая точку входа. Далее идёт упомянутый выше блок ***do***. Блок включает все строки от слова ***do*** до конца кода, содержащего строки, начинающиеся с одной и той же позиции. Напоминаю, что в Haskell расположение отступов в строках играет важную роль. Можно, конечно, вместо этого правила использовать фигурные скобки и точки с запятой. Тогда код можно располагать как угодно:

```
main = do
{putStrLn "Greetings!  What is your name?"; x <- getLine;
 putStrLn $ "Welcome to Haskell, " ++ x ++ "!"}
```

Из командной строки программу можно скомпилировать командой:

ghc --make prog.hs

Для запуска программы достаточно напечатать

prog → Greetings! What is your name?

Если на запрос ввода с клавиатуры ответим Boris Uvarov, то получим ответ:

Welcome to Haskell, Boris Uvarov!

Кстати, программу можно выполнить и из интерпретатора ghci, для чего файл надо загрузить как обычно и затем вызвать функцию ***main***.

Теперь посмотрим на пример, в котором чистая функция ***f*** используется внутри I/O actions:

```
f :: String -> String
f m = "Pleased to meet you, " ++ m ++ ".\n" ++
      "Your name contains " ++ x ++ " characters."
      where x = show (length m)
main :: IO ()
main = do
       putStrLn "Greetings once again.  What is your name?"
       s <- getLine
       let t = f s
       putStrLn t
```

Здесь функция ***f*** всегда возвращает один и тот же результат при одном и том же аргументе, она не имеет побочных эффектов, обладает обычной «ленивостью» и использует только стандартные функции (++), ***show*** и ***length***. В блоке ***do*** результат этой функции связывается с переменной ***t*** и только при actions происходит вывод на терминал. Это характерный пример, когда чистый код строго отделён от кода с побочным эффектом, что повышает надёжность программы. Именно такой подход считается в Haskell предпочтительным. Обратите внимание на тип функции ***main***, который показывает, что блок ***do*** всегда возвращает последнее вычисленное значение (в данном случае это результат функции ***putStrLn***).

Работа с файлами

Для записи в файл применяется функция ***writeFile***, принимающая два аргумента: название файла в кавычках и текст, который нужно записать:

```
writeFile "file1.txt" "Hello, Luisa!"
```

При записи старое содержимое файла стирается. Посмотрим на тип функции:

:type writeFile →writeFile :: FilePath -> String -> IO ()

Значит, здесь может использоваться полный путь файла. Для добавления текста в конец файла без уничтожения его содержимого применяется функция ***appendFile***:

```
appendFile "file1.txt" "Hello, Ljuda!"
```

Прочитать содержимое файла можно с помощью функции ***readFile***:
readFile "file1.txt" →"Hello, Luisa!Hello, Ljuda!"

Всё содержимое файла выводится на терминал одной строкой. Посмотрим на тип:
:type readFile → readFile :: FilePath -> IO String

Как видим, результат имеет тип ***IO String***, а не ***String***, и это означает, что его нельзя, например, применять как аргумент функции print или putStrLn. Как увидим далее, тип с приставкой ***IO String*** применяется в так называемых монадах, но о них позже. В модуле ***System.IO*** имеется целый набор функций, обеспечивающих всю работу по обмену информацией с файлами. Покажем их на примере, в котором считываются данные из файла ***file1.txt*** и записываются в файл ***file2.txt***:

```
import System.IO
main :: IO ()
main = do x <- openFile "file1.txt" ReadMode
          y <- openFile "file2.txt" WriteMode
          f x y
          hClose x
          hClose y
f :: Handle -> Handle -> IO ()
f x y = do z <- hIsEOF x
           if z then return () else do s <- hGetLine x
                                       hPutStrLn y s
                                       f x y
```

Итак, две строки:
x <- openFile "file1.txt" ReadMode
y <- openFile "file2.txt" WriteMode

объявляют две переменные ***x*** и ***y***, имеющие специальный тип ***Handle***, которые обеспечивают доступ к файлам по чтению и по записи соответственно. Кроме модификаторов ***ReadMode*** и ***WriteMode*** есть ещё ***ReadWriteMode*** и ***AppendMode*** . Смысл этих модификаторов ясен из их названий. Всю работу по обмену данными между файлами выполняет функция ***f***, принимающая ***x*** и ***y*** в качестве аргументов. Условие ***z <- hIsEOF x*** проверяет достижение конца файла. Если условие даёт ***True***, происходит выход из цикла с помощью функции ***return***. Иначе выполняется ещё один блок ***d***o, в

котором выражение ***s <- hGetLine x*** с помощью функции ***hGetLine*** считывает очередную строку из файла ***file1.txt***, а функция ***hPutStrLn*** записывает эту строку в файл ***file2.txt***. Затем функция ***f*** вызывается повторно. Строки ***hClose x*** и ***hClose y*** закрывают файлы. Снова обратите внимание на обязательные отступы в коде. В данном случае ***return*** это функция, возвращающая свой аргумент с приставкой ***IO*** для того, чтобы обе ветви оператора if – else возвращали один и тот же тип. У нас ***return ()*** возвращает ***IO ()***. Если написать ***return 9***, то получим на выходе ***IO 9***. Интересно посмотреть на тип функции ***openFile***. В интерпретаторе сначала надо загрузить модуль System.IO командой:

:module System.IO

Теперь вызываем тип:

:type openFile → openFile :: FilePath -> IOMode -> IO Handle

Здесь ***FilePath*** фактически синоним для ***String***.

В модуле System.Directory есть функция ***removeFile*** для удаления файлов. Эта функция принимает один аргумент — название удаляемого файла. Функция ***renameFile*** позволяет переименовать файл. Функция принимает два аргумента: старое и новое имя файла.

Если файл содержит текст, то можно использовать только функции ***readFile*** и ***writeFile***, которые выполняют все действия по открытию и закрытию файла, по чтению и записи текста. Например, запрограммируем чтение текста из одного файла и запись его в другой файл, при этом переведя весь текст в верхний регистр. Для перевода в верхний регистр используем функцию ***toUpper*** из модуля ***Data.Char***:

```
import Data.Char(toUpper)
main = do
    x <- readFile "file1.txt"
    writeFile "file2.txt" (map toUpper x)
```

Функция interact

Haskell имеет и ещё более простой способ для обмена информацией между устройствами, для этого применяется встроенная функция ***interact***. Этой функции надо передать свою функцию с типом ***String -> String***, которая может текст трансформировать нужным образом. Если текст надо передать без

изменения, можно применить функцию ***f m = m***; Вся программа будет иметь такой вид:

```
main = interact (f)
f :: String -> String
f m = m;
```

Информация о том, между какими устройствами должен быть выполнен обмен данными задаётся в команде на выполнение программы. Применим команду ***runghc***, которую мы упоминали ранее. Теперь она будет, например, такой:

```
runghc prog.hs < file1.txt > file2.txt
```

Значит, название файла, из которого считываются данные, надо записать в угловых скобках. Если при этом требуется перевести текст в верхний регистр, надо сделать так:

```
import Data.Char(toUpper)
main = interact (map toUpper)
```

Здесь использована краткая форма вызова функции ***map*** с использованием приёма каррирования, о котором говорилось выше. Чтобы показать более наглядно, как эта краткая форма применяется, приведём такой вспомогательный пример:

```
fi m = print(f m);
f :: [Int] -> [Int]
f = map g;
g :: Int -> Int
g t = t * t;
```

Выражение ***f = map g;*** каррирует функцию ***map***, создавая функцию одного переменного.

fi [1,2,3,4,5] →[1,4,9,16,25]

Чтобы вывести считанный из файла file1.txt текст на терминал, команда вызова программы должна быть такой:

```
runghc prog.hs < file1.txt
```

Если совсем не указывать устройств и вызвать программу так:

```
runghc prog.hs
```

то обмен по умолчанию будет между клавиатурой и терминалом, то-есть, программа будет ждать ввода, а напечатанная строка будет появляться на терминале в верхнем регистре.

6.Немного о монадах

Слово монада (monad) имеет очень широкое распространение в разных областях знаний: от названия простейшего организма в биологии до термина в оккультных науках. Чаще всего этим словом обозначают нечто трудно понятное в той или иной области знаний. Прижилось это слово и в программировании и прежде всего в языке Haskell. При программировании на Haskell, как впрочем и на других языках, вполне можно обойтись без всяких монад, но считается, что использование этих штук позволяет облегчить работу и сделать код более компактным. При этом в Haskell по крайней мере одну встроенную монаду под названием ***IO*** приходится использовать постоянно, поскольку на ней базируется рассмотренный выше ввод-вывод. Попробуем хотя бы поверхностно познакомиться с монадами на примере этой ***IO***.

Прежде всего можно отметить, что в Haskell монада это некий класс типов, имеющий такое определение:

```
class Monad m where
    (>>=) :: ma -> (a -> mb) -> mb
    (>>) :: ma -> mb -> mb
    return :: a -> ma
    fail :: String -> ma
```

Значит, в классе ***Monad*** определены четыре функции: (>>=), (>>), ***return*** и ***fail***. Функции (>>=) и (>>) инфиксные. Внешне в определении этого класса нет ничего необычного, кроме некой приставки ***m*** перед неопределёнными параметрами типа ***a*** и ***b***. Эта приставка, фиксируемая при создании экземпляров класса монад, как раз и определяет название конкретной монады. Если заменить ***m*** на ***IO***, то получим монаду ***IO***, определять которую не надо, поскольку она загружается автоматически из модуля Prelude. Для начала посмотрим, как работает функция (>>), сигнатура которой в монаде ***IO*** имеет вид:

```
(>>) :: IO a -> IO b -> IO b
```

Ранее мы уже знакомились с сигнатурой функции PutStrLn:

```
putStrLn :: String -> IO ()
```

Значит, результат, возвращаемый функцией ***putStrLn***, может быть использован для аргументов функции (>>), хотя на самом деле он и представляет пустой кортеж. Попробуем в интерпретаторе:

putStr "Hello, " >> putStrLn "world" → Hello, world

Получается, что (>>) просто выполняет оба операнда, правый и левый. Выведем тип, возвращаемый в данном случае:

:type putStr "Hello, " >> putStrLn "world" →

putStr "Hello, " >> putStrLn "world" :: IO ()

Поскольку тип этого результата ***IO ()***, то его опять можно использовать, как аргумент для другой (>>), например:

putStr "Hello, " >> putStr "world" >> putStrLn " monad" →

Hello, world monad

Итак, функция (>>) позволяет создавать цепочки, которые будут последовательно выполняться.

Посмотрим теперь, как работает (>>=) у которой в монаде ***IO*** такая сигнатура:

(>>=) :: IO a -> (a -> IO b) -> IO b

Левый операнд в данном случае может иметь тип ***IO String***. Как раз такой тип возвращает уже знакомая нам функция ***getLine***:

:type getLine → getLine :: IO String

Правым операндом должна быть функция с сигнатурой (***a -> IO b***), а это значит, что здесь можно опять использовать putStrLn:

getLine >>= putStrLn

Если мы на запрос функции ***getLine*** введём с клавиатуры какой-то текст, например ***Masha***, то и на выходе получим тот же текст ***Masha***. Получается, что функция (>>=) передаёт свой левый операнд уже без приставки ***IO*** функции, представленной правым операндом.

Усложним пример. Напишем такую программу:

```
f = getLine >>= g
  where g x = putStrLn $ "Hello, " ++ x
```

Напомню, что текст ***$ "Hello, " ++ x*** синтаксический сахар для ***("Hello, " ++ x)***.

Загружаем файл в ghci и вызываем функцию ***f***. Если теперь введём с клавиатуры например ***Natasha***, то получим:

Hello, Natasha

Это уже более интересный результат: (>>=) позволяет создавать функцию, которая умеет вводить текст с клавиатуры, делать над этим текстом какие-то манипуляции и выводить полученный результат на терминал. Функцию (>>=) тоже можно применять последовательно, создавая цепочки любой длины.

Чтобы понять, зачем монаде потребовалась функция ***return***, проанализируем такую программу:

```
f = return (g "Hello" "world") >>= putStrLn
    where g x y = x ++ ", " ++ y ++ "!"
```

f → Hello, world!

Программа вставляет в текст запятую, пробел и восклицательный знак и выводит результат. Мы не можем на вход (>>=) подать просто (***g "Hello" "world"***), так как функция ***g*** возвращает тип ***String***, а нам надо ***IO String***. Функция ***return*** как раз и добавляет это ***IO***, поскольку её сигнатура:

```
return :: Monad IO => a -> IO a
```

При этом тип ***a*** согласно контексту объявления типа ***Monad IO =>*** должен принадлежать классу ***Monad IO***. В данном случае это тип ***String***.

Наконец, функция ***fail*** нужна для обработки исключений, в монаде ***IO*** она может выдавать исключение ***IoError***. В другом контексте она способна на большее.

Посмотрим на пример программы, использующей уже знакомую нам do-нотацию:

```
main = do
    putStrLn "What's your name?"
    x <- getLine
    putStrLn $ "Hello, " ++ x ++ "!"
```

Оказывается, что do-нотация это всего лишь синтаксический сахар, а в действительности здесь используется монада. Можно полностью отказаться от do-нотаций, а вышеприведённый код заменить на такой:

```
main =
    putStrLn "What's your name?" >>
    getLine >>=
    \x -> putStrLn $ "Hello, " ++ x ++ "!"
```

Здесь последовательно применяются функции (>>) и (>>=). Применение анонимной функции делает код немного компактнее.

Ранее мы применяли функцию «обратная стрелка» (<-), которая тип IO String трансформирует в тип String. Фактически эта функция выполняет обратное действие по отношению к функции ***return*** в монаде IO. Все действия со списками основаны на монаде ***[]*** и в этой

монаде функция (<-) имеет совсем другой смысл. Посмотрим на пример:

```
f :: [Int] -> [Int]
f m = do
      x <- m
      [(x * x)]
```

f [2,5,3,7] → [4,25,9,49]

Значит, здесь (<-) выполняет роль итератора и очень похожа на функцию ***map***, при использовании которой этот пример будет иметь вид:

```
f :: [Int] -> [Int]
f m = map (\x -> x * x) m
```

Посмотрите на четыре функции, записанные разными способами, но выполняющие одни и те же действия над двумя списками:

```
f m n = do { x <- m; y <- n; return (x,y) }
f1 m n = [(x,y) | x <- m, y <- n]
f2 m n = m >>= \x -> n >>= \y -> return (x, y)
f3 m n = concat (map (\x -> concat (map (\y -> return (x, y)) n)) m)
```

f [1,2,3] [4,5] → [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

Все эти функции создают комбинации элементов двух списков, представленные в виде пар. Анализируя эти функции можно получить много полезной информации.

Приведу ещё один пример программы на эту тему. Эта программа определяет все сомножители заданного числа:

```
g :: Bool -> [a] -> [a]
g True  m = m
g False _  = []
f :: Int -> [(Int, Int)]
f n = do
      x <- [1..n]
      y <- [x..n]
      g (x * y == n) (return (x, y))
```

Результат получаем в виде списка кортежей, содержащих искомые сомножители:

f 100 → [(1,100),(2,50),(4,25),(5,20),(10,10)]

Анализируя аргументы, переданные функции ***g***, можно догадаться, что программа просто перебирает все возможные сомножители и

отбирает из них те, произведение которых равно заданному числу. Предлагаю читателю самому разобраться, как тут получается список в результате. Даю подсказку. Создадим такую функцию:

```
f :: int -> [int]
f x = return(x)
```

Убедимся, что эта функция возвращает список:

```
f 2 → [2]
```

Тип Maybe, с которым мы знакомились ранее, тоже представляет монаду с этим именем. Вот как выглядит экземпляр класса типа для этой монады:

```
instance Monad Maybe where
    Just x >>= k  =  k x
    Nothing >>= _ =  Nothing
    Just _ >> k   =  k
    Nothing >> _  =  Nothing
    return x      =  Just x
    fail _        =  Nothing
```

Нетрудно понять смысл этого текста.

Вообще тема монад огромна, можно сказать неисчерпаема. Существует много различных определений монад, часто противоречивых. Практически можно считать, что в Haskell монады это некие особые классы типов, имеющие какие-то методы с нетрадиционными возможностями. Соответственно, применение монад позволяет писать более компактные программы. При этом надо отметить, что эти программы приобретают совсем нечитабельный вид. Кстати, существуют даже языки программирования, например язык J, в которых почти не применяются слова и весь код состоит из одних только условных значков. Не знаю, как другим, но мне лично, такие программы не очень нравятся.

7. Регулярные выражения

При работе с регулярными выражениями надо импортировать модуль ***Text.Regex.Posix***. Для сопоставления текста с регулярным выражением применяется инфиксный оператор =~. Первым операндом (слева от =~) должен быть текст, а вторым — регулярное выражение. Оба операнда могут иметь тип ***String*** или ***ByteString*** (о

типе ByteString немного позже). Сопоставление может возвращать результат разных типов. Иногда в контексте программы тип результата может быть получен автоматически, но лучше его указывать явно в конце выражения через **::**. В частности, этот тип может быть ***Bool***:

import Text.Regex.Posix
f1 = "my left foot" =~ "foo" :: Bool
f2 = "your right hand" =~ "bar" :: Bool
f3 = "жили были дед да баба" =~ "(дед|баба)" :: Bool
f4 = "жили были дед да баба" =~ "(кот|баба)" :: Bool

Получим:

f1 → True
f2 → False
f3 → True
f4 → True

Значит, регулярное выражение может иметь сразу два значения и тогда они должны располагаться в круглых скобках и разделяться прямой чертой.

Можно указать тип ***Int***, который дает нам подсчет количества совпадений:

"Горные вершины спят во тьме ночной" =~ "ны" :: Int → 2

Если надо подсчитать количество каких-то букв в тексте (например, гласных), то эти буквы надо заключить в квадратные скобки:

"Горные вершины спят во тьме ночной" =~ "[аеиоуыюя]" :: Int → 11

```
data JValue = JString String
            | JNumber Double
            | JBool Bool
```

```
          | JNull
          | JObject [(String, JValue)]
          | JArray [JValue]
            deriving (Eq, Ord, Show)


JString "foo" → JString "foo"
JNumber 2.7 → JNumber 2.7
:type JBool True → JBool True :: Jvalue

getString :: JValue -> Maybe String
getString (JString s) = Just s
getString _           = Nothing

getString (JString "hello") → Just "hello"
getString (JNumber 3) → Nothing


m = [(1, "one"), (2, "two"), (3, "three"), (4, "four")]
myLookup :: Eq a => a -> [(a, b)] -> Maybe b
myLookup _ [] = Nothing
myLookup key ((k,v):xs) =
   if key == k
      then Just v
      else myLookup key xs
myLookup 4 m → Just "four"


import qualified Data.Map as Map
h =
   Map.insert 2 "two" .
   Map.insert 4 "four" .
   Map.insert 1 "one" .
   Map.insert 3 "three" $ Map.empty
h → fromList [(1,"one"),(2,"two"),(3,"three"),(4,"four")]
```