Язык программирования Haskell

→ *Учимся быть ленивыми* →

Разделы

Введение

Язык программирования Haskell является чисто функциональным языком программирования общего назначения с ленивыми вычислениями. Также у этого языка сильная статическая типизация. В будущем мы поговорим почему язык является чисто функциональным и почему его называют ленивым.

Haskell назван в честь американского математика Хаскелла Карри. Относится к языкам семейства ML и испытал большое влияние языка программирования Miranda.

Данная книга ориентирована как для новичков в программировании, так и для тех, кто уже знает один или несколько языков программирования. Книга будет вам предоставлять примеры готовых программ и функций, которые уже были проверены на работоспособность. Также в этой книге мы не будем пытаться понять все тонкости языка, так как книга, в основном, создана для понимания функциональной парадигмы, чем самого языка программирования.

Помимо этого, я буду предоставлять программы и функции на других языках программирования (Си, Python), чтобы можно было сравнивать с языком Haskell.

Стоит также упомянуть, что Haskell является достаточно лёгким языком программирования, на уровне Python или Ruby, но многих отпугивает сама концепция чистого функционального программирования, при которой не используются переменные и циклы. В данной же книге я попытаюсь показать, что функциональное программирование это вовсе не страшный зверь, а достаточно красивая и интересная парадигма.

Автор книги: Коваленко Г. А.

Вводный урок

И начнём мы пожалуй со старой и доброй традиции времён книги K&R, а именно с вывода сообщения “hello, world”. В Haskell это делается достаточно просто:

```
main :: IO()
main = putStrLn "hello, world"
```

Сохраните этот файл под именем Main.hs. Расширение .hs указывает на исходный файл языка программирования Haskell.

Всё, мы написали нашу первую программу. И чтобы её запустить, нам нужно воспользоваться либо компилятором, либо интерпретатором.
В этой книге будет использоваться компилятор ghc:
https://www.haskell.org/ghc/

И чтобы скомпилировать нашу программу, нам нужно ввести следующую команду в терминале:

```
ghc Main.hs
```

Эта команда предполагает, что у вас компилятор и файл находятся в одной директории. И после компиляции у вас создастся исполняемый файл с именем Main. При его запуске мы увидим следующий результат:

```
hello, world
```

В последующем я буду изображать вывод результата программы символом ‘>’ в терминале.

А теперь давайте рассмотрим более подробно код, который мы написали.

Main – это главная функция, с которой начинается любая программа в языке Haskell. После двух двоеточий идёт тип данных функции. В нашем случае тип данных функции IO() или иными словами Input-Output (Ввод-Вывод). Этот тип данных говорит нам, что функция способна принимать и выводить какие-либо данные.

```
1. main :: IO()
```

Во второй строке уже происходят действия функции main. После знака ‘равно’ мы используем функцию putStrLn, которая способна выводить какую-либо строку в терминал. После функции putStrLn мы указываем нужную нам строку и в итоге получаем необходимый результат.

```
2. main = putStrLn "hello, world"
```

Также стоит упомянуть, что тип данных функций необязательно писать и его можно просто опустить, в итоге Haskell сам поймёт какие типы данных нужно подставлять в тип функции. Тем не менее, это считается плохим тоном и впоследствии у нас могут возникать сложности в анализе каких-либо функций. Поэтому в книге мы будем всегда указывать типы данных.

Пример “hello, world” на языке программирования Си:

```
#include <stdio.h>
int main (void) {
        printf("hello, world\n");
        return 0;
}
```

Пример “hello, world” на языке программирования Python:

```
print("hello, world")
```

Функции

“Сложность мыслить функционально возникает вследствие привычки мыслить императивно.”

Никаких переменных, никаких циклов, никаких присваиваний в Haskell нет. Он оперирует, в большей степени, математическими понятиями, чем терминологией языков программирования.

Функции в Haskell – это чистые функции или иными словами математические. Помимо чистых функций есть также и нечистые функции, которые чаще всего встречаются в императивных языках программирования, подобия Си.

Определение чистых функций сводится к двум свойствам, которые стоит соблюдать для их создания. Если наблюдается только одно свойство из двух, то функция не является чистой.

1. Функция обязана быть детерминированной.

Другими словами, её выходные данные, должны полностью зависеть от входных.

2. Функция не должна иметь побочных эффектов.

Функции с побочными эффектами способны производить операции ввода-вывода, изменять значения глобальных переменных или изменять значения переданных ей аргументов по адресу.

Стоит сказать, что не каждая функция с побочным эффектом будет являться недетерминированной.
Но также и не факт, что недетерминированная функция, будет являться функцией с побочным эффектом.

```
int func (int x) {
	printf("hello, world\n");
	return x * x;
}
```

В данном случае, функция является детерминированной, так как возвращаемое значение зависит полностью от принимаемого аргумента x. Тем не менее, эта функция имеет побочные эффекты, потому что используется функция printf, способная выводить какие-либо данные в терминал, тем самым работая с операционной системой.

```
int global_x = 1;
int func (int x) {
	return global_x + x * x;
}
```

В этом примере, функция не является детерминированной, так как результат функции, зависит не только от переданных ей аргументов. Но эта функция не имеет побочных эффектов, потому что она способна лишь читать глобальную переменную, но изменять её значение или производить операции ввода-вывода она не в состоянии.

А теперь давайте приступать к Haskell’ю и попробуем создать очень простую функцию, которая будет возводить наше число в квадрат.

```
square :: Int -> Int
square x = x ^ 2 -- Возведение числа x в квадрат.

main :: IO()
main = print $ square 5
```

И как итог, мы получим следующий результат: 25
Теперь же давайте рассмотрим этот код более подробно.

Тип данных функции square говорит о том, что принимается в качестве первого аргумента значение типа данных Int (целочисленное значение) и возвращается тип данных Int как результат работы этой функции.
1. square :: Int -> Int

Функция square принимает в качестве аргумента значение x. После знака равенства указывается возвращаемое значение (x ^ 2), где крышечка '^' является оператором возведения в степень. При передаче нашего числа 5, под значение x подставляется это число и в итоге мы получаем (5 ^ 2). Далее это выражение возвращается в функцию main и функция main проделывает арифметическую операцию, и выводит при помощи функции print результат 25.

Также в этой строке мы видим однострочный комментарий, который начинается с двух дефисов '--'. После дефисов можно писать любые символы и они будут игнорироваться компилятором и интерпретатором.
2. square x = x ^ 2 -- Возведение числа x в квадрат.

Функция main использует функцию square передавая в неё число 5 и принимая число 25, далее идёт оператор '$', который избавляет нас от написания скобок. Без него этот код выглядил бы следующим образом: main = print (square 5). [Haskell программисты не слишком любят скобки и поэтому пытаются от них избавляться.]
Далее идёт функция print, которая и выводит наш полученный результат. Отличие print от функции putStrLn в том, что putStrLn способна выводить только строки, в то время как print способна выводить любые типы данных, которые привязаны к классу типов Show.
4. main :: IO()
5. main = print $ square 5

Люди, которые знают программирование, скорее уже заметили недочёт этой программы. А именно: "что будет, если на вход в функцию square поступит нецелое число, допустим 5.5?". Ответ: выведится ошибка.
И соответственно поступает сразу логический вопрос, как можно избежать этого? Есть два решения, чтобы устранить этот недочёт.

Первое – это изменить тип Int, на Float или Double:
```
square :: Float -> Float
```

[Float – вещественное число одинарной точности]
[Double – вещественное число двойной точности]

Второе – это изменить тип Int, на какой-либо тип (a).
```
square :: Num a => a -> a
```

Здесь предполагается, что нет точного определения у этого типа, но что точно нам известно – это то, что аргумент является числом, поэтому следует в типе данных указать класс Num и присвоить к этому классу тип ‘a’. Подобные типы ‘a’ называют полиморфными или переменными типов.

Теперь давайте попробуем реализовать функцию add, которая просто будет складывать два числа:

```
add :: Num a => a -> a -> a
add x y = x + y

main :: IO()
main = print $ add 5 6
```

Получим следующий результат:
```
> 11
```

Функция add принимает 2 аргумента (a) и возвращает значение (a). Также (a) относится к классу типов Num.
1. `add :: Num a => a -> a -> a`

Функция add принимает два аргумента (x, y) и складывает их при помощи функции (+).
2. `add x y = x + y`

Далее идёт передача двух чисел в функцию add и вывод результата при помощи функции print.
5. main = print $ add 5 6

У функций, которые принимают два аргумента, есть две формы записи, в языке программирования Haskell: префиксная и инфиксная.

Если функция обозначается при помощи символов алфавита, тогда по умолчанию используется префиксная форма записи (add x y).

Если функция обозначается при помощи иных символов, тогда используется инфиксная форма записи (x + y).

Таким способом нашу функцию можно немного изменить:

```
---------------------------------------------------------------------------------
add :: Num a => a -> a -> a
add x y = (+) x y

main :: IO()
main = print $ 5 `add` 6
---------------------------------------------------------------------------------
```

Получим тот же результат:
> 11

В этом примере мы указываем для функции (+) префиксную форму записи ((+) x y).
2. add x y = (+) x y

В этом же примере мы указываем для функции add инфиксную форму записи (5 `add` 6), где используются обратные кавычки.
5. main = print $ 5 `add` 6

[О типах данных рассказано более подробно в следующем разделе.]

1. *Реализация функции* ($):

```
-- ($$) = ($)

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

---

2. *Реализация функции* (^):

```
-- (^^^) = (^)

(^^^) :: Num a => a -> Int -> a
(^^^) _ 0 = 1
(^^^) _ y | y < 0 = error "degree < 0"
(^^^) x y = x * (^^^) x (y – 1)
```

Реализация функции square на языке программирования Си:

```
double square (double x) {
        return x * x;
}
```

Реализация функции square на языке программирования Python:

```
square = lambda x: x ** 2
```

Типы данных

Стандартные типы данных в Haskell:

Bool – тип данных, которые содержит лишь два значения True и False.

Int – тип данных, который представляет целые числа с ограниченным диапозоном значений.

Integer – тип данных, который представляет целые числа с неограниченным диапозоном значений.

Float – тип данных, который представляет вещественные числа с одинарной точностью.

Double – тип данных, который представляет вещественные числа с двойной точностью.

Char – тип данных, который представляет какой-либо символ. Символ обознается при помощи одинарных кавычек: 'a'.

String – тип данных, который является синонимом типа данных [Char] (списка символов) и представляется собой строку. Строка обозначается при помощи двойных кавычек: "hello, world".

Для того, чтобы проверить диапозон типов подобия Int, можно применять функции minBound и maxBound:

```
---------------------------------------------------------------------------------
main :: IO()
main = do
        print (minBound :: Int)       -- -9223372036854775808
        print (maxBound :: Int)       -- 9223372036854775807
---------------------------------------------------------------------------------
```

[Ключевое слово ‘do’ помогает нам писать код в стиле императивного программирования. Более подробно об этом говорится в разделе *Нечистые функции*.]

Помимо указания обычных типов данных, можно также указывать и классы типов в зависимости от того, что делает наша функция. Основные классы типов, которые чаще всего встречаются:

Eq – класс, который использует функции (==), (/=) [Функции сравнения], где (==) - равно, а (/=) - не равно.

Ord – класс, который наследует класс Eq и добавляет помимо этого функции (<), (<=), (>=), (>) [Функции сравнения], где (<) - меньше, (<=) - меньше или равно, (>=) - больше или равно, (>) - больше.

Num – класс, который использует функции (-), (+), (*), (^) [Функции арифметических операций], где (-) - вычитание, (+) - сложение, (*) - умножение, (^) - возведение в степень.

Fractional – класс, который наследует класс Num и добавляет помимо этого, функцию (/) [Функция арифметических операций], где (/) - деление.

Show – класс, который использует функцию show, где show – это перевод какого-либо типа данных в строку.

Read – класс, который использует функцию read, где read – это перевод строки в какой-либо тип данных.

Стоит сказать, что не всегда нужно будет указывать класс какого-либо типа. В некоторых функциях не имеет значения что поступает, будь то число или символ, или какой-либо список.

Некоторые функции могут являться функциями высшего порядка (ФВП). Определение таких функций сводится к тому, что они могут принимать или возвращать другие функции.

Тип данных такой функции выглядит примерно следующим образом: `func :: (a -> b) -> [a] -> [b]`.

Здесь поступает два аргумента – функция (a -> b) и список [a]. Возвращается список [b].

[Также Haskell позволяет создавать собственные типы данных и классы типов. Об этом говорится в разделах *Создание типов данных* и *Создание классов*.]

Ветвления

Способов разветвления кода в языке Haskell достаточно много.
И давайте поставим перед собой простую задачу:

- при вводе числа 1, будет возвращаться строка “hello”
- при вводе числа 2, будет возвращаться строка “world”
- при вводе иного числа, будет возвращаться строка “undefined”

Тип данных функции во всех случаях будет одинаковым:

```
func :: Int -> String
```

Также везде будет функция main следующего вида и во всех случаях у нас будет выводится слово "world":

```
main :: IO()
main = print $ func 2
```

Везде, где ветка условия переходит на следующую строку, нужно ставить отступы (аналогично языку программирования Python).

1. *if then else*

```
func x = if x == 1
        then "hello"
        else if x == 2
                then "world"
                else "undefined"
```

Если вы до этого изучали какие-либо языки программирования, то вам эта конструкция должна быть знакомой.

Читается всё это следующим образом:
Если x равняется единице, тогда вернуть “hello”, иначе если x равняется двум, тогда вернуть “world”, иначе вернуть “undefined”.

Пример этой конструкции на языке программирования Си:

```
char *func (int x) {
	if (x == 1)
		return "hello";
	else if (x == 2)
		return "world";
	else
		return "undefined";
}
```

2. *case of*

```
func x = case x of
	1 -> "hello"
	2 -> "world"
	otherwise -> "undefined"
```

Конструкция case of тоже должна быть знакомой массе программистов, так как она напоминает оператор switch в Си подобных языках.

Пример этой конструкции на языке программирования Си:

```
char *func (int x) {
	switch (x) {
		case 1: return "hello";
		case 2: return "world";
		default: return "undefined";
	}
}
```

Читается эта конструкция следующим образом:

Взять число x и сравнить его:
с единицей, и если число x равно единице, тогда вернуть “hello”.
с двойкой, и если число x равно двойке, тогда вернуть “world”.
Иначе, если ничего выше не нашлось, вернуть “undefined”.

Функция otherwise должна стоять в самом конце ветки условия, потому что она всегда возвращает значение True:

```
otherwise :: Bool
otherwise = True
```

3. *function argument*

```
func 1 = "hello"
func 2 = "world"
func _ = "undefined"
```

В языке Haskell функция сразу способна определять переданные ей аргументы. Это называется сопоставление с образцом.

Таким образом, если функция получила в качестве аргумента число 1, тогда она вернёт “hello”. Если получила число 2, тогда она вернёт “world”. Иначе, вернёт “undefined”. Символ нижнего подчёркивания ‘_’ означает, что неважно какой был передан аргумент в эту функцию. В нашем же случае нижний знак подчёркивания играет роль функции otherwise.

4. *function argument and condition*

```
func x | x == 1 = "hello"
func x | x == 2 = "world"
func _ = "undefined"
```

Также функция способна принимать аргумент и сразу его сранивать при помощи других функция подобия (==), как в примере *№ 4*.

5. *guard*

```
func x
        | x == 1 = "hello"
        | x == 2 = "world"
        | otherwise = "undefined"
```

Пятый пример является фактически примером *№ 4*, с тем лишь отличием, что условия находятся в одном блоке.

6. *guard if*
[Чтобы использовать эту конструкцию ветвления, надо подключить расширение MultiWayIf в начале файла .hs]

```
{-# LANGUAGE MultiWayIf #-}

{-
	Пример многострочного
	комментария
-}

func x = if
	| x == 1 -> "hello"
	| x == 2 -> "world"
	| otherwise -> "undefined"
```

Шестая конструкция во многом схожа с примером *№ 5*. Отличия лишь заключены в том, что нужно подключить расширение, а также добавить ‘= if’ и изменить знаки равенства ‘=’ на стрелочки ‘->’.

Также здесь показан комментарий вида ‘{--}’. В отличии от однострочного комментария ‘--’ может располагаться на нескольких строках. Является многострочным комментарием.

В конструкциях [if then else], [case of], [guard], [guard if] оступы необходимы при переходе на новую строку, аналогично языку программирования Python.

Конструкции let и where

Без переменных обходиться можно, но без именованных констант достаточно сложно, иначе наш код будет являться сборником магических чисел и значений. Чтобы этого избежать - применяют две конструкции: let и where.

Часто новички, изучающие язык Haskell, считают, что конструкции let и where позволяют нам создавать переменные. Но отнюдь это не так. То что они считают переменными, является константами. Если же идти вглубь языка, то подобные константы будут являться функциями, которые просто возвращают константные значения.

И давайте промоделируем ситуацию. Допустим мы просматриваем код программиста, в котором есть функция под названием ‘func’ и в неё передаётся какое-то магическое число. Чтобы понять чем является это число – нужно будет проанализировать саму функцию func. Но если заместо магического числа будет находиться именованная константа, то и работы по анализу функции будет в несколько раз меньше.

Фактически конструкции let и where выполняют одну и ту же задачу. Отличия заключены лишь в позиции, где они находятся, а также в том, что конструкция where видит все значения в пределах этой функции, а конструкция let видит только указанный блок. Конструкция let находится перед использованием именованной константы, тогда как конструкция where находится после использования именованной константы.

И вот пример как с конструкцией let, так и с конструкцией where, которые выполняют одно и то же действие.

Допустим у нас есть функция func, которая принимает значение типа Int и возвращает значение типа String:

```
func :: Int -> String
func x = case x of
        1960 -> "1-9-6-0"
        1970 -> "1-9-7-0"
        1980 -> "1-9-8-0"
        1990 -> "1-9-9-0"
        otherwise -> "nothing"
```

Тип данных функции main остаётся прежним: main :: IO().

1. Пример с *let*:

```
main = let year = 1970 in print $ func year
```

2. Пример с *where*:

```
main = print $ func year where year = 1970
```

Результат программы:

```
> "1-9-7-0"
```

Какую конструкцию использовать – это дело вкуса. Но также мы можем использовать эти две конструкции вместе, что часто не рекомендуется делать из-за сложности чтения кода. Тем не менее, иногда в некоторых задачах может быть полезно это свойство. Как пример – какая-либо формула из физики. Я взял в качестве примера – нахождение пути:

```
main :: IO()
main =
        let s = v * t
        in print s
        where
                v = 40
                t = 10
```

Также я упоминал, что в языке Haskell именованные константы – это функции, возвращающие константные значения. И выше описанный пример можно расписать так:

```
main :: IO()
main =
	let
		s :: Num a => a
		s = v * t
	in
		print s
	where
		v :: Num a => a
		v = 40

		t :: Num a => a
		t = 10
```

Функции s, v и t ничего не принимают, но возвращают какие-либо значения. В данном случае функция v ничего не принимает, но возвращает значение 40, функция t также ничего не принимает и возвращает значение 10, и функция s ничего не принимает и возвращает результат умножения v на t.

Указание типов функций в конструкциях let и where часто опускается, так как зачастую нет особого смысла указывать явно тип, где функция просто возвращает результат, не принимая никаких аргументов.

Списки

Списки в языке программирования Haskell являются множеством элементов одного типа данных и не являются индексируемыми.
И давайте посмотрим пример какого-либо списка:

```
---------------------------------------------------------------------------------
main :: IO()
main = putStrLn "hello, world"
---------------------------------------------------------------------------------
```

Как не странно, но строка является списком, а именно списком символов, так как тип данных String – это синоним типа данных [Char] и везде где есть String, можно без проблем подставить [Char].

Этот код можно было расписать иначе и результат останется тем же:
```
main = putStrLn ['h','e','l','l','o',',',' ','w','o','r','l','d']
```

В этом случае язык Haskell схож с языком Си, где как таковых строк нет.

Также не стоит забывать о том, что одинарные кавычки – это символы Char, а двойные кавычки – это строки [Char].

Теперь давайте посмотрим на различные действия, которые мы можем делать со списками. Допустим перед нами стоит несколько простых задач:

1. *Вывести список чисел от 1 до 10.*

Первый способ решения:
```
main = print [1,2,3,4,5,6,7,8,9,10]
```

Второй способ решения:
```
main = print [1..10]
```

В данном случае две точки (..) заполнят промежуток числами от единицы до десяти.

```
> [1,2,3,4,5,6,7,8,9,10]
```

2. *Вывести все нечётные числа от 1 до 10.*

Первый способ решения:

```
main = print [1,3,5,7,9]
```

Второй способ решения:

```
main = print [1,3..10]
```

В данном случае Haskell проанализирует шаг между единицей и тройкой, и в последующем будет применён этот шаг.

```
> [1,3,5,7,9]
```

3. *Вывести все числа от 10 до 1.*

Первый способ решения:

```
main = print [10,9,8,7,6,5,4,3,2,1]
```

Второй способ решения:

```
main = print [10,9..1]
```

В данном случае нам необходимо указать шаг от 10 до 9, так как Haskell работает без шага только на возрастание.

```
> [10,9,8,7,6,5,4,3,2,1]
```

4. *Вывести все числа кратные десяти на промежутке от 10 до 100.*

Первый способ решения:

```
main = print [10,20,30,40,50,60,70,80,90,100]
```

Второй способ решения:

```
main = print [10,20..100]
```

```
> [10,20,30,40,50,60,70,80,90,100]
```

5. *Вывести весь английский алфавит.*

Первый способ решения:

```
main = print
['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']
```

Второй способ решения:

```
main = print "abcdefghijklmnopqrstuvwxyz"
```

Третий способ решения:

```
main = print ['a'..'z']
```

У каждого символа есть своё число, у символа ‘a’ - это 97, ‘b’ - это 98 .. ‘z’ - это 122.

```
> "abcdefghijklmnopqrstuvwxyz"
```

6. *Вывести числа от 0 до 3 с шагом 0.5.*

Первый способ решения:

```
main = print [0.5,1,1.5,2,2.5,3]
```

Второй способ решения:

```
main = print [0.5,1..3]
```

```
> [0.5,1.0,1.5,2.0,2.5,3.0]
```

7. *Вывести числа от 0 до -10.*

Первый способ решения:

```
main = print [0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]
```

Второй способ решения:
```
main = print [0,-1..(-10)]
```

Мы поставили скобки, так чтобы функция минус (-) относилась только к числу 10.

```
> [0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]
```

8. *Вывести числа от 1 до бесконечности.*
```
main = print [1..]
```

Как итог мы получим бесконечную программу, которая будет печатать нам числа в порядке возрастания

```
> [1,2,3,4,5,6,7,8,9, . . .
```

Если нам нужно получить несколько значений из бесконечного списка, то можем воспользоваться функцией take:
```
main = print $ take 10 [1..]
```

```
> [1,2,3,4,5,6,7,8,9,10]
```

Также стоит сказать, что запись подобия: `main = print [1,2,3,4,5]` является эквивалентом записи: `main = print $ 1:2:3:4:5:[]`

Знак двоеточия (:) добавляет значение слева в начало списка. Результат вывода функции print в обоих случаях будет равен следующему: [1,2,3,4,5].

С задачками мы на этом пожалуй и остановимся. Надеюсь вы поняли как можно в языке Haskell заполнять список какими-либо данными.

Скорее всего у вас возник вопрос: “а как можно получить определённый элемент этого списка?”. И всё достаточно просто:
```
main = print $ ['a'..'z'] !! 2
```

В итоге мы получим символ ‘c’, так как индексирование начинается с нуля. Пару страниц назад я говорил, что списки в языке Haskell не индексируются и я не врал.

Функция (!!) является рекурсивной и принимает в качестве аргумента какой-либо список и индекс. В нашем случае список был ['a'..'z'], а индекс 2.

Далее функция (!!) каждый раз проверяет индекс на значение 0 и если индекс не равен нулю, тогда вычесть из индекса единицу, из списка убрать голову (head), а в рекурсивную функцию (!!) положить хвост (tail) списка.

Рекурсия закончится тогда, когда индекс будет равен нулю и как итог мы получим голову (head) оставшегося списка.

[Более подробно мы разберём рекурсию в разделе *Рекурсия*.]

Также в Haskell есть такая конструкция как генераторы списков, они достаточно удобны и гибки в использовании:

```
main = print [x^2 | x <- [1..10], x <= 5]

> [1,4,9,16,25]
```

В данном выше примере делается следующее:

Сначала выполняется блок (x <- [1..10]). x – это значение, которое берётся из списка [1..10].

Далее выполняется блок условия (x <= 5). Если x меньше или равен пяти, тогда выполнять действие.

И в последнюю очередь выполняется блок действия (x^2). В данном случае берётся значение x из списка [1..10] и возводится в квадрат.

Новый список дополняется новыми элементами до тех пор, пока старый список не будет пустым.
Значения же в новом списке зависят от условия.

Также может и не быть условий в генераторе:

```
main = print [x^2 | x <- [1..10]]
```

```
> [1,4,9,16,25,36,49,64,81,100]
```

Генератор списков может иметь ещё и следующий вид:

```
main = print [x*y | x <- [4..6], y <- [1..3]]
```

```
> [4,8,12,5,10,15,6,12,18]
```

Алгоритм генерации подобного списка:
[(4*1), (4*2), (4*3), (5*1), (5*2), (5*3), (6*1), (6*2), (6*3)]

К этому генератору также можно легко добавить условие:

```
main = print [x*y | x <- [4..6], y <- [1..3], x == 5]
```

```
> [5,10,15]
```

Мы можем делать даже так:

```
main = print [if mod x 2 == 0 then "hello" else "world" | x <- [1..5]]
```

```
> ["world","hello","world","hello","world"]
```

Функция mod возвращает остаток от деления. В данном случае, если остаток деления числа x на число 2 будет равен нулю, тогда добавить в список строку “hello”, иначе “world”.

3. *Реализация функции head*:

```
-- head' = head

head' :: [a] -> a
head' [] = undefined
head' (x:_) = x
```

---

4. *Реализация функции tail*:

```
-- tail' = tail

tail' :: [a] -> [a]
tail' [] = []
tail' (_:xs) = xs
```

---

5. *Реализация функции take*:

```
-- take' = take

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

---

6. *Реализация функции* (!!):

```
-- (!!!) = (!!)

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

Реализация функции head на языке программирования Python:

```
head = lambda x: x[0]
```

Реализация функции tail на языке программирования Python:

```
tail = lambda x: x[1:]
```

Кортежи

Кортежи в языке программирования Haskell являются набором элементов всеразличных типов данных, на основе которых создаётся новый тип данных.

Используются кортежи реже, чем списки, тем не менее иногда они бывают полезными, допустим, когда нужно передавать или возвращать разные типы данных.

Простой пример, при котором функция принимает строку и число, и возвращает кортеж типа (String, Int):

```
---------------------------------------------------------------------------------------
func :: String -> Int -> (String, Int)
func x y = (x, y)

main :: IO()
main = print $ func "hello" 571
---------------------------------------------------------------------------------------

> ("hello",571)
```

Также давайте реализуем стандартные функции языка, которые помогают нам вычерпывать из кортежа нужные элементы.

7. *Реализация функции fst*:

```
---------------------------------------------------------------------------------------
-- fst' = fst

fst' :: (a, b) -> a
fst' (x,_) = x
---------------------------------------------------------------------------------------
```

```
main :: IO()
main = print $ fst ("hello",571)
```

```
> "hello"
```

8. *Реализация функции snd*:

```
-- snd' = snd

snd' :: (a, b) -> b
snd' (_,y) = y
```

```
main :: IO()
main = print $ snd ("hello",571)
```

```
> 571
```

Часто кортежи встречаются в подобном виде:
[(1, "First"), (2, "Second"), (3, "Third")]

И наша текущая цель – это получить список всех вторых элементов кортежей заданного списка.

Простой способ - это воспользоваться генератором списков:

```
main :: IO()
main = print [b | (_,b) <- list]
       where list = [(1, "First"), (2, "Second"), (3, "Third")]
```

```
> ["First","Second","Third"]
```

Второй способ – это воспользоваться рекурсией:

```
-------------------------------------------------------------------------------
get_snd :: [(a,b)] -> [b]
get_snd [] = []
get_snd (x:xs) = snd x : get_snd xs

main :: IO()
main = print $ get_snd list
        where list = [(1, "First"), (2, "Second"), (3, "Third")]
-------------------------------------------------------------------------------

> ["First","Second","Third"]
```

Рекурсия

“Чтобы понять рекурсию, нужно сначала понять рекурсию.”

Чаще всего в рекурсиях идёт работа со списками, но иногда бывают функции, где нужно производить работу с числами. И к таким функциям чаще всего относят нахождение чисел факториала и фибоначчи.

Эти две функции я взял неспроста. Люди, которые уже знают программирование, скорее всего реализовывали эти функции и тем самым легко могут сравнить синтаксис и понять что к чему.

Начнём пожалуй с функции нахождения факториала:

```
-------------------------------------------------------------------------------
-- 1 * 2 * 3 * 4 * 5 * 6 = 720

factorial :: Integer -> Integer
factorial x | x < 2 = 1
factorial x = (*) x $ factorial $ x - 1

main :: IO()
main = print $ factorial 6
-------------------------------------------------------------------------------

> 720
```

Здесь всё предельно просто:

Если передаваемый аргумент (x) будет меньше двух, тогда прекратить рекурсию и вернуть число 1.

Иначе, если аргумент больше или равен двум, тогда работать по следующей формуле: x * f (x-1).

Полная последовательность работы этой функции с аргументом 6:

```
factorial 6 ->
   6 * factorial (6 - 1)  ->
      5 * factorial (5 - 1)  ->
         4 * factorial (4 - 1)  ->
            3 * factorial (3 - 1)  ->
               2 * factorial (2 - 1) ->
                     factorial 1 = 1     -- на этом этапе factorial прекращает
                                         -- рекурсию и возвращает число 1, т.к.
                                         -- число 1 меньше двух.
```

Если мы отбросим рекурсивную часть, то получим следующее:

```
6 *
   5 *
      4 *
         3 *
            2 *
               1
> 720
```

Сначала рекурсивные вычисления идут сверху-вниз, но когда функция прекращает рекурсию, вычисления начинают идти в обратную сторону, с той позиции, где рекурсия прекратила свою работу.

```
6 * (5 * (4 * (3 * (2 * 1))))
```

Поэтому верная последовательность действий функции factorial будет такой: [1 * 2 * 3 * 4 * 5 * 6], но не такой: [6 * 5 * 4 * 3 * 2 * 1].

Реализация функции factorial на языке программирования Си:

```
unsigned long factorial (unsigned int x) {
        unsigned long num = 1;
        while (x > 1)
                num *= x--;
        return num;
}
```

Реализация функции factorial на языке программирования Python:

```
def factorial (x):
        if x < 2: return 1
        else: return x * factorial(x-1)
```

Функция нахождения числа фибоначчи по номеру:

```
-------------------------------------------------------------------------------
-- N  :  1  2  3  4  5  6   7   8   9  10  11  ...
-- F  :  1  1  2  3  5  8  13  21  34  55  89  …

fibonacci :: Integer -> Integer
fibonacci x | x < 3 = 1
fibonacci x = fibonacci (x-1) + fibonacci (x-2)

main :: IO()
main = print $ fibonacci 6
-------------------------------------------------------------------------------

> 8
```

Функция fibonacci также является рекурсивной, но она вызывает себя два раза, вместо одного. Это может показаться достаточно сложным, но на самом деле здесь нет ничего страшного:

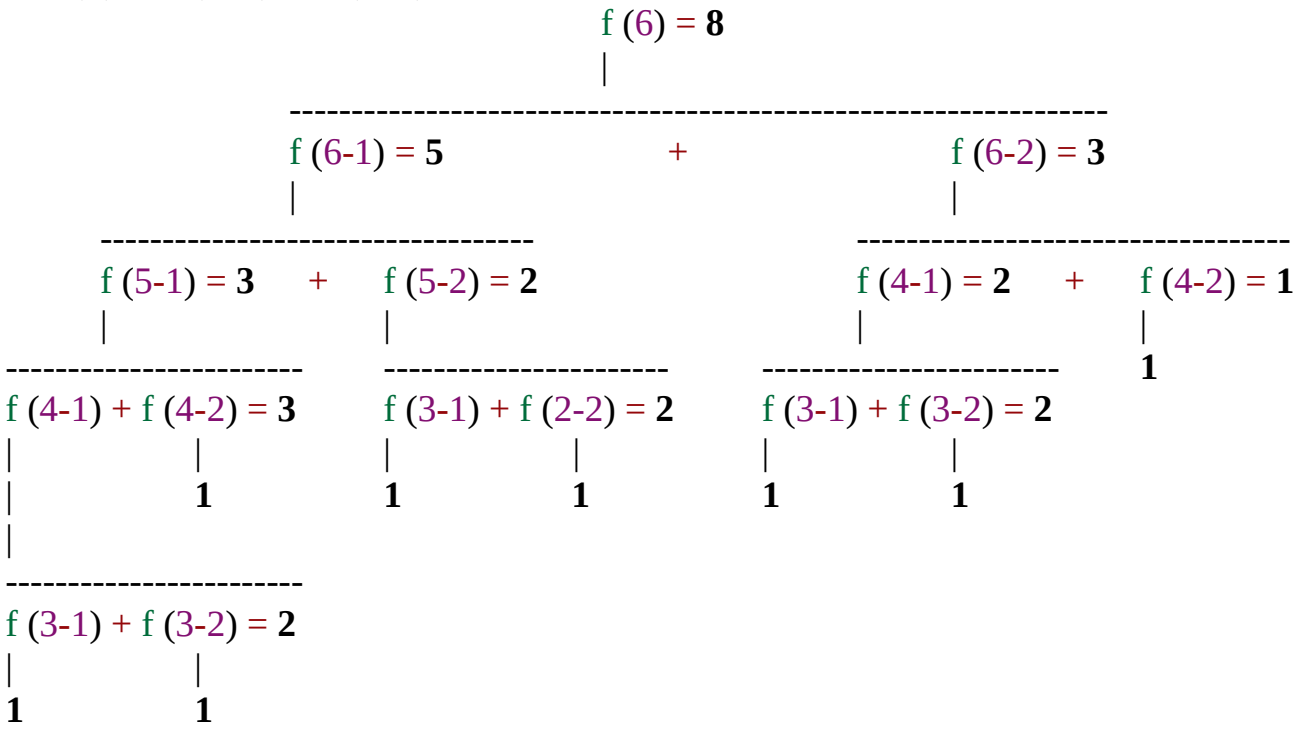
Если передаваемый аргумент меньше трёх, тогда прекратить рекурсию и вернуть число 1.

Иначе, если аргумент больше или равен трём, тогда работать по следующей формуле: f (x-1) + f(x-2)

Вся суть этой рекурсии заключена в разделении функции на две части.

Полная последовательность работы функции fibonacci с аргументом 6:

-- f = fibonacci
-- f (x) = 1, x < 3
-- f (x) = f (x-1) + f (x-2)



Реализация функции fibonacci на языке программирования Си:

```
unsigned long fibonacci (unsigned int z) {
	unsigned long x, y;
	x = y = 1;
	while (z --> 1)
		x ^= y, y ^= x, x = y + (x ^ y);
	return y;
}
```

Реализация функции fibonacci на языке программирования Python:

```
def fibonacci (x):
	if x < 3: return 1
	else: return fibonacci(x-1) + fibonacci(x-2)
```

Функция нахождения номера фибоначчи по числу:

```
-------------------------------------------------------------------------------
-- N  :  1  2  3  4  5  6   7   8   9  10  11  ...
-- F  :  1  1  2  3  5  8  13  21  34  55  89  …

fibonacci :: Integer -> Integer
fibonacci x | x < 3 = 1
fibonacci x = fibonacci (x-1) + fibonacci (x-2)

findFib :: Integer -> Integer
findFib x = func 1
        where func n
                | x == fibonacci n = n
                | n > 20 = -1
                | otherwise = func $ n + 1

main :: IO()
main = print $ findFib 34
-------------------------------------------------------------------------------

> 9
```

Функция findFib использует внутреннюю рекурсивную функцию func передавая в неё константу 1. Результатом функции findFib является результат функции func.

Смысл внутренней функции func заключён в счётчике (n), а также в том, что она способна видеть аргументы функции findFib (x).

Сама функция func является достаточно лёгкой рекурсивной функцией, которая способна сравнивать два получившихся числа. Её словесный алгоритм звучит следующим образом:
- Принять на вход число-счётчик n.
- Если аргумент функции findFib равен числу от функции fibonacci(n), тогда вернуть число n.

- Если число n больше константы 20, тогда вернуть -1 как результат того, что нет ответа в диапозоне от 1 до 20.
- Иначе, увеличить счётчик на единицу и передать его рекурсивно в функцию func.

Реализация функции findFib на языке программирования Си:

```
unsigned int findFib (const unsigned long x) {
	unsigned int i;
	for (i = 1; i <= 20; i++)
		if (x == fibonacci(i))
			return i;
	return -1;
}
```

Реализация функции findFib на языке программирования Python:

```
def findFib (x):
	def func (n):
		if x == fibonacci(n): return n
		elif n > 20: return -1
		else: return func(n+1)
	return func(1)
```

А теперь попробуем создать рекурсивную функцию, которая будет принимать список чисел и возвращать список чисел возведённых в квадрат.

По-факту, это будет бесполезная функция. Но она достаточно хорошо может показать как рекурсия способна функционировать со списками.

```
list_square :: Num a => [a] -> [a]
list_square [] = []
list_square (x:xs) = x^2 : list_square xs
```

```
main :: IO()
main = print $ list_square [1..5]
```

Получим мы следующий результат:
> [1,4,9,16,25]

Функция принимает список с каким-либо числовым типом данных (a) и возвращает нам список с таким же числовым типом данных (a).
1. list_square :: Num a => [a] -> [a]

Вторая строка – это выход из рекурсии. Если список будет пустым – тогда вернуть пустой список. Изначально это покажется бессмысленным, но вся суть этой рекурсивной функции заключена в третьей строке.
2. list_square [] = []

В этой строке и происходят основные действия, которые мы сейчас разберём.
3. list_square (x:xs) = x^2 : list_square xs

(x:xs) – полный список, который мы передали в функцию list_square. Список в данном случае разделяется на части при помощи двоеточия. Это выглядит следующим образом:

1:[2,3,4,5] - равносильно обычному списку [1,2,3,4,5].

x – голова (head) списка. Самый первый элемент списка.

xs – хвост (tail) списка. Весь список без головы (без первого элемента).

С левой частью мы разобрались. Теперь давайте посмотрим алгоритм правой части:

- берётся голова списка (x) и возводится во вторую степень: x^2
- далее полученный результат, от возведённого числа в степень (x^2), помещается обратно в список при помощи двоеточия.
- и в функцию list_square передаётся хвост списка: list_square xs

В словесном алгоритме может быть много непонятных вещей, так что воссоздадим полную последовательность работы данной рекурсии:

На вход в функцию list_square поступает список вида [1,2,3,4,5].

```
list_square [1,2,3,4,5] -> 1:[2,3,4,5]
1^2:
   list_square [2,3,4,5] -> 2:[3,4,5]
   2^2:
      list_square [3,4,5] -> 3:[4,5]
      3^2:
         list_square [4,5] -> 4:[5]
         4^2:
            list_square [5] -> 5:[]
            5^2:
               list_square [] = []     -- на этом этапе list_square прекращает
                                       -- рекурсию и возвращает пустой список.
```

Если мы отбросим рекурсию функции, то получим следующий вид:

```
(1^2) : (2^2) : (3^2) : (4^2) : (5^2) : [] =
= [(1^2), (2^2), (3^2), (4^2), (5^2)] =
= [1, 4, 9, 16, 25]
```

Большинство рекурсивных функций, которые работают со списками, строятся по такому же алгоритму, который мы разобрали.

Реализация функции list_square на языке программирования Си:

```
void list_square (double array[], int length) {
        while (--length >= 0)
                array[length] *= array[length];
}
```

Реализация функции list_square на языке программирования Python:

```
def list_square (List):
        if not List: return []
        else: return [head(List)**2] + list_square(tail(List))
```

Теперь давайте по этому алгоритму попробуем воссоздать стандартную функцию map, которая изначально есть в языке Haskell.

Эта функция является функцией высшего порядка и принимает в качестве аргумента функцию и список.

9. *Реализация функции map*:

```
-------------------------------------------------------------------------------
-- map' = map

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


-------------------------------------------------------------------------------
main :: IO()
main = print $ map (^2) [1..5]
-------------------------------------------------------------------------------
```

```
> [1,4,9,16,25]
```

Функция map принимает в качестве аргумента функцию (a -> b) и список [a]. Далее через функцию (a -> b) проходит список [a] и возвращается список [b].

1. `map' :: (a -> b) -> [a] -> [b]`

Также стоит сказать, что можно было бы использовать только типы данных вида (a), без использования типа (b), но у нас сразу начнут возникать потенциальные ошибки, так как функция map способна принимать один тип, а возвращать другой: `main = print $ map show [1..5]`

```
> ["1","2","3","4","5"]
```

Функция show принимает какой-либо тип данных и этот тип данных возвращает в виде строки. Если бы у нас были только типы данных вида (a), тогда у нас бы вывелась ошибка компиляции.

Вторая строка является выходом из рекурсии. Первый аргумент не учитывается, а учитывается только список.
2. `map' _ [] = []`

Третья строка является основной, в ней и происходит рекурсия по тому же самому алгоритму, который мы рассматривали ранее, только вместо (^2) можно подставлять любую функцию, которая будет работать с одним значением.
3. `map' f (x:xs) = f x : map' f xs`

В качестве аргументов в функцию map передаётся функция (^2) и список [1..5]. Но скорее всего вас удивит тот факт, почему функция подобия (^2) не вызывает никаких ошибок, ведь у этой функции нет второго аргумента (числа), который ей необходим.
И это явление называется сечением функции.
[Более подробно об это говорится в разделе *Каррирование*.]
5. `main = print $ map' (^2) [1..5]`

Мы передаём в качестве аргумента неполную функцию вида (^2).
Эта функция возводит элементы списка в квадрат: [1,4,9,16,25].
Но, если мы переставим функцию ^ в правую часть = (2^), то в итоге функция map будет выдавать нам степени двойки: [2,4,8,16,32].

Это работает потому что функция (^) принимает инфиксную форму записи, а не префиксную, тем самым значение зависит от позиции функции.

Чтобы сделать нашу функцию (^) в префиксной форме, нужно поместить знак ^ в отдельную скобку, тем самым получим следующее выражение: ((^) 2). Но здесь мы будем всегда получать степени двойки.

Чтобы получать возведённые в квадрат элементы списка с префиксной формой, нам нужно будет воспользоваться лямбда-функцией:

```
main = print $ map' (\x -> (^) x 2) [1..5]
```

Знак (\) - означает лямбда-функцию, после этого знака идут аргументы, подобия x, далее пишется стрелка (->) и после неё то, что возвращает нам лямбда-функция.

Как итог мы получаем новую функцию:
- которая берёт какое-либо значение x.
- которая содержит в себе функцию возведения в степень (^)
- которая будет передаваться в функцию map в качестве аргумента.

Реализация функции map на языке программирования Си:

```
void map (double (*func) (double), double array[], int length) {
        while (--length >= 0)
                array[length] = (*func) (array[length]);
}
```

Реализация функции map на языке программирования Python:

```
def map (func, List) -> list:
        if not List: return []
        else: return [func(head(List))] + map(func, tail(List))
```

Также иногда возникают ситуации, когда нужно использовать в одной рекурсии весь список, хвост списка и голову. Таким примером может служить функция dropWhile, которая отбрасывает элементы до тех пор, пока условие не будет истинным.

10. *Реализация функции dropWhile*:

```
-- dropWhile' = dropWhile

dropWhile' :: (a -> Bool) -> [a] -> [a]
dropWhile' _ [] = []
dropWhile' f (x:xs)
        | f x = dropWhile' f xs
        | otherwise = x:xs
```

Здесь есть такая конструкция как otherwise = x:xs при условии того, что функция f (x) вернула False. Эту функцию можно было написать немного иначе, при помощи образца (@):

```
dropWhile' :: (a -> Bool) -> [a] -> [a]
dropWhile' _ [] = []
dropWhile' f qq@(x:xs)
        | f x = dropWhile' f xs
        | otherwise = qq

main :: IO()
main = print $ dropWhile' (< 4) [1..10]
```

```
> [4,5,6,7,8,9,10]
```

Реализация функции dropWhile на языке программирования Си:

```
unsigned int dropWhile (
        _Bool (*func) (double), double array[], const unsigned int length
) {
        unsigned int position = 0;
        while (position < length && (*func) (array[position++]));
        return position - 1;
}
```

Реализация функции dropWhile на языке программирования Python:

```
def dropWhile (func, List):
        if not List: return []
        elif func(head(List)): return dropWhile(func, tail(List))
        else: return List
```

Язык программирования Haskell также позволяет создавать список функций и использовать этот список в качестве аргументов другой функции. И вот пример, как это можно сделать:

```
func :: [a -> a] -> a -> [a]
func [] _ = []
func (f:fs) x = f x : func fs x

main :: IO()
main = print $ func [(^2), (*2), (/2)] 4
```

```
> [16.0,8.0,2.0]
```

В данном случае тип [a -> a] обозначает список функций, которые принимают значение типа a и возвращают значение этого же типа.

Создание типов данных

В языке программирования Haskell существует три ключевых слова для создания типов данных:

- data – создаёт новый тип данных с возможностью безграничного добавления значений этого типа.

И самым простым примером использования ключевого слова data – является создание типа данных с двумя значениями.
Назовём наш тип данных Nums, а его значения: One и Two, а также создадим функцию, которая просто будет переводить наш тип данных в число 1 или 2:

```
-------------------------------------------------------------------------------
data Nums = One | Two

check :: Nums -> Int
check One = 1
check Two = 2

main :: IO()
main = print $ check One
-------------------------------------------------------------------------------

> 1
```

Вот такая у нас получилась простая программа. При помощи знака (|) можно ещё добавлять всеразличные значения и их может быть столько, сколько вам необходимо.

Но здесь есть достаточно значительная проблема, а именно, в этом примере мы не можем вывести наш тип данных напрямую без использования функций подобия check.

Это можно исправить двумя способами:

1. Привязать значения типа данных к аргументам функции show и указать что должно возвращаться. В этом нам поможет ключевое слово instance.

```
data Nums = One | Two

instance Show Nums where
        show One = "One"
        show Two = "Two"

main :: IO()
main = print One
```

```
> One
```

2. добавить в создание типа данных следующую строку: deriving Show. В данном случае deriving автоматически привязывает наш тип к функции show. В итоге наш тип является наследуемым от класса Show:

```
data Nums = One | Two deriving Show

main :: IO()
main = print One
```

```
> One
```

Помимо класса Show, мы также можем добавлять и другие классы. Допустим добавим класс Eq, который поможет нам сравнивать значения типа Nums.

```
data Nums = One | Two deriving Eq

main :: IO()
main = print $ One == Two
```

```
> False
```

Этот же пример с ключевым словом instance:

```
data Nums = One | Two

instance Eq Nums where
        (==) One One    = True
        (==) Two Two    = True
        (==) _ _        = False

main :: IO()
main = print $ One == Two
```

Мы также можем сделать значения нашего типа перечисляемыми при помощи класса Enum:

```
data Nums = One | Two | Three | Four | Five deriving (Show, Enum)

main :: IO()
main = print $ [x | x <- [One ..]]
```

```
> [One,Two,Three,Four,Five]
```

[После значения One, в списке, должен стоять пробел и только после этого идти две точки]

Также ключевое слово data может содержать в своём значении несколько различных типов:

```
-------------------------------------------------------------------------------
data IPv4 = IPv4 String Int

get :: IPv4 -> String
get (IPv4 ip port) = "IPv4: " ++ ip ++ ". Port: " ++ show port

main :: IO()
main = print $ get $ IPv4 "127.0.0.1" 8080
-------------------------------------------------------------------------------

> "IPv4: 127.0.0.1. Port: 8080"
```

Может также показаться странным, что тип данных IPv4 также является и внутренней составляющей его значения IPv4 String Int.
IPv4 – в значении это конструктор, a String и Int – это значения других типов.

[Такая конструкция чаще всего используется для того, чтобы написанный код можно было легче воспринимать.]

Тем самым у нас появляется возможность создавать сложные типы, которые в своём значении могут содержать другие типы данных.
Такие конструкции называют алгебраическими типами данных (АТД).

Функция (++) конкатенирует (склеивает) два списка в один.

Написанную выше программу можно немного дополнить:

```
-------------------------------------------------------------------------------
data Address = IPv4 String Int | IPv6 String Int

get :: Address -> String
get (IPv4 ip port) = "IPv4: " ++ ip ++ ". Port " ++ show port
get (IPv6 ip port) = "IPv6: " ++ ip ++ ". Port " ++ show port

main :: IO()
main = print $ get $ IPv6 "fe80:0:0:0:200:f8ff:fe21:67cf" 8080
-------------------------------------------------------------------------------

> "IPv6: fe80:0:0:0:200:f8ff:fe21:67cf. Port 8080"
```

И теперь у нас появляется возможность указывать не только IPv4 адрес, но также и IPv6 соответственно.

11. *Реализация функции* (++):

```
-------------------------------------------------------------------------------
-- (+++) = (++)

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

Также у ключевого слова data есть ещё одна достаточно хорошая особенность, а именно создание меток полей.

Допустим мы работаем в оффисе и у каждого сотрудника есть свой ID, Имя-Фамилия, Телефон, Email и есть ли у него дома кот. Всё это мы можем объединить в один тип данных следующим образом:

```
-------------------------------------------------------------------------------
data Person = Person {
        id_person  :: Int,
        name       :: String,
        phone      :: String,
        email      :: String,
        cat        :: Bool
}

main :: IO()
main = print $ name some_person
        where
                some_person :: Person
                some_person = Person {
                        id_person  = 1,
                        name       = "Alex Adamson",
                        phone      = "9-876-543-21-00",
                        email      = "example@email.com",
                        cat        = True
                }
-------------------------------------------------------------------------------

> "Alex Adamson"
```

В данном случае в типе данных Person находится конструктор Person и указывается пять функций (метки поля), которые что-либо возвращают: допустим функция id_person возвращает значение Int, функция name возвращает значение String и тд.

Далее у нас создаётся функция some_person, которая будет возвращать тип данных Person. В эту функцию мы передаём все значения, которые должен содержать тип Person по меткам.

Если мы не укажем какого-либо поля с меткой, ошибки компиляции у нас не будет, но компилятор выведит предупреждение, что тип не является полным и тем самым у нас может возникнуть потенциальная ошибка.

И последним действием мы выводим имя и фамилию сотрудника. Чтобы вывести результат – нам нужно сначала указать метку поля (name), а потом функцию, которая возвращает тип Person (some_person).

Если мы сделаем тип Person наследником класса Show, тогда мы сможем вывести сразу всю информацию с функции some_person. Также мы можем немного упростить запись меток поля:

```
data Person = Person {
        id_person  :: Int,
        name       ,
        phone      ,
        email      :: String,
        cat        :: Bool
} deriving Show

main :: IO()
main = print some_person
        where
                some_person :: Person
                some_person = Person {
                        id_person  = 1,
                        name       = "Alex Adamson",
                        phone      = "9-876-543-21-00",
                        email      = "example@email.com",
                        cat        = True
                }
```

```
> Person {id_person = 1, name = "Alex Adamson", phone = "9-876-543-21-00", email = "example@email.com", cat = True}
```

Помимо всего прочего, data тип может иметь свои параметры:

```
data All a = All a deriving Show

main :: IO()
main = print $ All "hello, world"
```

```
> All "hello, world"
```

В данном примере, тип All принимает аргумент, который может быть любого типа данных.

-type – создаёт синоним уже существующего типа данных, как это сделано в типе String: type String = [Char].

```
type Email = String
type Message = String

hello :: Email -> Message
hello xs = "Hello, " ++ xs

main :: IO()
main = print $ hello "example@email.com"
```

```
> "Hello, example@email.com"
```

-newtype – создаёт новый тип данных с конструктором и другим типом.

```
--------------------------------------------------------------------------------
newtype IPv4 = IPv4 String

getIP :: IPv4 -> String
getIP (IPv4 ip) = "IPv4: " ++ ip

main :: IO()
main = print $ getIP $ IPv4 "127.0.0.1"
--------------------------------------------------------------------------------

> "IPv4: 127.0.0.1"
```

И это впринципе всё что может newtype. Смысл его использования сводится к оптимизации программы, так как data создаёт свои собственные типы данных, в то время как newtype использует уже существующий тип данных с конструктором.

Теперь немного отвлечёмся от теории и попробуем погрузиться вглубь языка Haskell. Представим такую ситуацию – перед нами есть тот же самый язык программирования Haskell, но в нём отсутствуют все типы данных и функции, за исключением функции print. Кажется что в таком случае мы сможем сделать только программу с выводом строки “hello, world” и ничего более, но на самом деле Haskell является очень гибким языком программирования и в нём изначально “вшито” очень мало конструкций, что позволяет даже при минимальных средствах создать полностью рабочую программу.

Первое что мы сделаем – это импортируем из модуля Prelude только функцию print, тип IO() и класс Show, всё остальное же будет игнорировано, тем самым мы не сможем воспользоваться уже какой-нибудь готовой функцией и не возникнет ошибки дубликатов функций.

```
--------------------------------------------------------------------------------
-- [модуль Prelude всегда автоматически подключался к нашим
-- программам]
import Prelude (IO(), print, Show)
```

```
-- Начнём нашу программу с типа данных Bool:
data Bool = False | True  deriving (Show)

-- Теперь реализуем функцию ($):
($) :: (a -> b) -> a -> b
($) f x = f x

-- Можем также реализовать функции (==) и (/=):
(==) :: Bool -> Bool -> Bool
(==) True True = True
(==) False False = True
(==) _ _ = False

-- main :: IO() -- проверка работоспособности функции (==).
-- main = print $ (==) True False

(/=) :: Bool -> Bool -> Bool
(/=) True False = True
(/=) False True = True
(/=) _ _ = False

-- main :: IO() -- проверка работоспособности функции (/=).
-- main = print $ (/=) True False

-- Воссоздадим функции && (AND) и || (OR), а также NOT:
-- AND:
(&&) :: Bool -> Bool -> Bool
(&&) True x = x
(&&) False _ = False

-- main :: IO() -- проверка работоспособности функции AND.
-- main = print $ (&&) True False
```

```
-- OR:
(||) :: Bool -> Bool -> Bool
(||) False x = x
(||) True _ = True

-- main :: IO() -- проверка работоспособности функции OR.
-- main = print $ (||) True False

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

-- main :: IO() -- проверка работоспособности функции NOT.
-- main = print $ not True

-- Имея функции AND, OR и NOT – мы сможем реализовать XOR:
-- XOR:

xor :: Bool -> Bool -> Bool
xor x y = (not x && y) || (x && not y)

-- main :: IO() -- проверка работоспособности функции XOR.
-- main = print $ xor True False

-- Теперь давайте создадим ветвление (if then else):
if_then_else :: Bool -> a -> a -> a
if_then_else True  f _ = f
if_then_else False _ s = s

-- А теперь попробуем создать наши первые рекурсивные функции and и
-- or, которые содержат список типов Bool.
```

```haskell
-- and:
and :: [Bool] -> Bool
and [] = True
and [x] = x
and (x:y:xs) = if_then_else (not x && y) (False) $ and (y:xs)

-- main :: IO() -- проверка работоспособности функции and.
-- main = print $ and [True, True, False, True]

-- or:
or :: [Bool] -> Bool
or [] = False
or [x] = x
or (x:y:xs) = if_then_else (x || y) (True) $ or (y:xs)

-- main :: IO() -- проверка работоспособности функции or.
-- main = print $ or [True, True, False, True]
```

Таким образом, мы сделали достаточно много рабочих функций полностью с нуля. Как раз в этом и заключена удивительная особенность языка программирования Haskell.

Каррирование

Теперь давайте попробуем ещё немного углубиться в язык Haskell и постигнем небольшой дзен, а именно каррирование.

[Каррирование получило такое наименование, как и сам язык программирования Haskell, по имени Хаскелла Карри]

Вспомним нашу функцию add, которая как я говорил, принимает два аргумента (x и y), но я отчасти соврал, потому что все функции в языке Haskell принимают всегда один аргумент.

Возьмём опять же тип функции add:
```
add :: Num a => a -> a -> a
```

Haskell же воспринимает тип этой функции немного иным образом:
```
add :: Num a => a -> (a -> a)
```

Это можно описать таким способом: функция add принимает в качестве аргумента значение a и возвращает функцию (a -> a), в которой в свою очередь принимается значение типа a и возвращается значение типа a.

Будем использовать в примере ту же самую функцию add, но немного её изменим добавив вместо аргументов лямбда-функции, так как лучше всего изобразить каррирование помогают именно они.

```
add :: Num a => a -> (a -> a)
add = \x y -> x + y

main :: IO()
main = print $ add 5 6
```

```
> 11
```

Теперь попробуем передавать в функцию add по одному аргументу.

```
add :: Num a => a -> (a -> a)
add = \x -> \y -> x + y

main :: IO()
main = print func'
        where
                func :: Num a => a -> a
                func = add 5

                func' :: Num a => a
                func' = func 6
```

Результат программы останется тем же:

```
> 11
```

А сейчас посмотрим внутреннее строение функции func, которая берёт функцию add лишь с одним аргументом 5: `func = add 5`.

```
func :: Num a => a -> a
func = \y -> 5 + y

main :: IO()
main = print func'
        where
                func' :: Num a => a
                func' = func 6
```

```
> 11
```

И в конце концов мы получаем функцию func', которая берёт функцию func и передаёт ей аргумент 6: func' = func 6. И впоследствии производится функция (+) с аргументами 5 и 6.

```
func' :: Num a => a
func' = 5 + 6

main :: IO()
main = print func'
```

```
> 11
```

Таким образом, можно сделать вывод о том, что в Haskell любые функции, в которых больше одного аргумента, являются функциями высшего порядка (ФВП), так как способны возвращать другую функцию.

Также при помощи каррирования становится возможным частичное применение функций и тем самым бесточечная нотация, а также создание сечений.

Частичным, называют применение функции, когда передаётся аргументов меньше, чем ожидается.

Пример частичного применения функции:

```
main :: IO()
main = print $ square [1..5]
        where
                square :: Num a => [a] -> [a]
                square = map (\x -> x^2)
```

```
> [1,4,9,16,25]
```

В данном случае функция map ожидает в качестве второго аргумента – список.

Сечением, называют использование функции с фиксированным аргументом, подобия (^2).

Пример сечения функции:

```
-------------------------------------------------------------------------------
main :: IO()
main = print $ (^2) 5
-------------------------------------------------------------------------------

> 25
```

Бесточечной нотацией, называют определение функции, при которой опускаются её аргументы.

Пример без использования бесточечной нотации:

```
sum' :: Num a => [a] -> a
sum' xs = foldr (+) 0 xs
```

Пример с использованием бесточечной нотации:

```
sum' :: Num a => [a] -> a
sum' = foldr (+) 0

-------------------------------------------------------------------------------
main :: IO()
main = print $ sum [1,2,3,4,5]
-------------------------------------------------------------------------------

> 15
```

12. *Реализация функции foldr*:

```
-- foldr' = foldr

foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' _ y [] = y
foldr' f y (x:xs) = f x $ foldr' f y xs
```

Реализация функции foldr на языке программирования Python:

```
def foldr (func, elem, List):
	if not List: return elem
	else: return func(head(List), foldr(func, elem, tail(List)))
```

Реализация функции foldr на языке программирования Си:

```
double foldr (
	double (*func) (double, double), double num,
	double array[], int length
) {
	while (--length >= 0)
		num = (*func) (num, array[length]);
	return num;
}
```

Также в статьях и книгах может встречаться такое слово как композиция. Это применение функции к результату вычисления другой функции. За композицию отвечает функция точки (.).

Пример использования композиции:

```
main :: IO()
main = print $ func [1..10]
	where func = reverse . filter even
```

```
> [10,8,6,4,2]
```

В данном примере сначала используется функция filter, которая принимает какую-либо функцию и список. Функция even определяет является ли элемент списка чётным. На основе этого функция filter создаёт список только чётных чисел. Далее производит работу функция reverse, которая просто переворачивает наш список.

13. *Реализация функции* (.):

```
-------------------------------------------------------------------------------
-- (...) = (.)

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

14. *Реализация функции reverse*:

```
-------------------------------------------------------------------------------
-- reverse' = reverse

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

Реализация функции reverse на языке программирования Си:

```
void reverse (double s[], const unsigned int length) {
        double temp;
        unsigned int left, right;
        for (left = 0, right = length-1; left < right; left++, right--)
                temp = s[left], s[left] = s[right], s[right] = temp;
}
```

Реализация функции reverse на языке программирования Python:

```
def reverse (List):
	if not List: return []
	else: return reverse(tail(List)) + [head(List)]
```

15. *Реализация функции filter*:

```
-- filter' = filter

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

Реализация функции filter на языке программирования Python:

```
def filter (func, List) -> list:
	if not List: return []
	elif func(head(List)): return [head(List)] + filter(func, tail(List))
	else: return filter(func, tail(List))
```

Реализация функции filter на языке программирования Си:

```
unsigned int filter (
	_Bool (*func) (double), double array[],
	double result[], const unsigned int length
) {
	unsigned int pos = 0;
	double *p = array;
	while (p < array + length) {
		if ((*func) (*p))
			result[pos++] = *p;
		++p;
	}
	return pos;
}
```

16. *Реализация функции even*:

```
-- even' = even

even' :: Integral a => a -> Bool
even' x | mod x 2 == 0 = True
even' _ = False
```

Реализация функции even на языке программирования Си:

```
_Bool even (int x) {
	if ( !(x % 2) ) return 1;
	return 0;
}
```

Реализация функции even на языке программирования Python:

```
even = lambda x: True if not x % 2 else False
```

Лень

“Лень является отличным качеством для тех, кому не лень, эту самую лень, использовать во благо лени.”

Ленивые вычисления являются фундаментом, на котором находится язык программирования Haskell. Их также называют отложенными вычислениями.

Помимо отложенных вычислений, есть также и строгие вычисления, которые присущи большинству языков программирования.

Суть стратегии ленивых вычислениях сводится к тому, что сами вычисления откладываются до тех пор, пока не понадобится их результат. Это свойство может быть как плюсом, так и минусом.

Допустим у нас есть функция add, которая складывает два аргумента и мы в неё передаём определённые выражения:
```
add (5 + 3, 9 + 11)
```

И вот как будут обстоять дела в строгих вычислениях:
```
add (5 + 3, 9 + 11) -> (8, 20) ->
add: 8 + 20 ->
print 28
```

А вот так это будет происходить в ленивых вычислениях:
```
add (5 + 3, 9 + 11) ->
add: (5 + 3) + (9 + 11) ->
print ((5 + 3) + (9 + 11)) -> (8 + 20) -> 28
```

Также, ленивые вычисления способны проверять аргументы таким способом, чтобы было меньше действий.

В данном случае ленивые вычисления будут ориентироваться по первому аргументу и не будут никак пытаться вычислять второй аргумент:

```
(||) :: Bool -> Bool -> Bool
(||) False x = x
(||) True _ = True
```

В Haskell также используется мемоизация. Это свойство, при котором сохраняется результат выполнения функции для предотвращения повторных вычислений. Как пример:

```
func :: Integral a => a -> a
func x = x + x * x ^ x

main :: IO()
main = print $ func $ 2 + 3
```

```
> 15630
```

И соответственно вместо такого варианта вычисления:
`func (2 + 3) = (2 + 3) + (2 + 3) * (2 + 3) ^ (2 + 3)`

Мы получим следующий вариант, который использует Haskell:
`func (2 + 3) = 5 + 5 * 5 ^ 5`

Теперь вспомним наш бесконечный список: [1..]
По-факту он и вправду является бесконечным, но он не вычисляется до тех пор, пока мы не потребуем его данных. И в зависимости от того, сколько мы их потребуем, столько мы и получим, ни больше ни меньше.

Если мы сделаем таким образом: main = print [1..], то получим бесконечную последовательность чисел и тем самым бесконечную программу.

```
> [1,2,3,4,5,6,7,8,9, . . .
```

Но если мы попробуем взять только часть этого списка (5 элементов): main = print $ take 5 [1..], то мы и получим только пять элементов этого списка, а всё что идёт после пятого элемента просто не вычисляется.

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

По подобной же причине списки не являются индексируемыми, потому что, изначально не ясно, сколько элементов списка потребуется.

Также, наглядным примером ленивости языка может послужить функция foldl:

```
-------------------------------------------------------------------------------
main :: IO()
main = print $ foldl (+) 0 [1,2,3]
-------------------------------------------------------------------------------
```

```
> 6
```

17. *Реализация функции foldl*:

```
-------------------------------------------------------------------------------
-- foldl' = foldl

foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' _ y [] = y
foldl' f y (x:xs) = foldl' f (f y x) xs
-------------------------------------------------------------------------------
```

Алгоритм работы функции foldl:

```
foldl (+) 0 [1,2,3] -> 1:[2,3]
   foldl (+) (0 + 1) [2,3] -> 2:[3]
      foldl (+) ((0 + 1) + 2) [3] -> 3:[]
         foldl (+) (((0 + 1) + 2) + 3) [] -> []

         (((0 + 1) + 2) + 3) <-
      ((1 + 2) + 3) <-
   (3 + 3) <-
6 <-
```

Ранее мы рассматривали схожую функцию под названием foldr. Отличия foldl и foldr заключены в способе достижения результата.

Алгоритм работы функции foldl уже рассмотрели, а пример работы алгоритма функции foldr ранее не рассматривали. Мы только реализовывали саму эту функцию (стр. 55).

Алгоритм работы функции foldr:

```
foldr (+) 0 [1,2,3] -> 1:[2,3]
   1 + (foldr (+) 0 [2,3]) -> 2:[3]
      2 + (foldr (+) 0 [3] ) -> 3:[]
         3 + (foldr (+) 0 []) -> [] = 0

         (1 + (2 + (3 + 0))) <-
      (1 + (2 + 3)) <-
   (1 + 5) <-
6 <-
```

Если посмотреть на последовательность работы foldr, то будет это выглядить следующим образом:
(1 + (2 + (3 + 0))) -- Right

В то время как в foldl это выглядит так:
(((0 + 1) + 2) + 3) -- Left

Реализация функции foldl на языке программирования Си:

```
double foldl (
        double (*func) (double, double), double num,
        double array[], const unsigned int length
) {
        double *p = array;
        while (p < array + length)
                num = (*func) (num, *p++);
        return num;
}
```

Реализация функции foldl на языке программирования Python:

```
def foldl (func, elem, List):
        if not List: return elem
        else: return foldl(func, func(elem, head(List)), tail(List))
```

Вот ещё пример ленивого вычисления, при котором вычисление никогда не произойдёт: `main = let f = sum [1..] in print "hello, world"`,
так как результат функции f нигде не используется. Такое выражение называют задумкой или Thunk.

Кажется, что есть всего два состояния ленивых вычислений: вычисляется, при условии того, когда потребуется результат (Normal Form) или вообще не вычисляется (Thunk)

Но на самом деле есть промежуточное состояние, при котором вычисления происходят, но выполняются не полностью (Weak Head Normal Form).

Классическим примером этого служит функция length:

```
main = print $ length ['a'..'d']
```

Её работа происходит следующим образом:
```
length [Thunk, Thunk, Thunk, Thunk] ->
1 + length [Thunk, Thunk, Thunk] ->
1 + (1 + length [Thunk, Thunk]) ->
1 + (1 + (1 + length [Thunk])) ->
1 + (1 + (1 + (1 + 0))) <-
1 + (1 + (1 + 1)) <-
1 + (1 + 2) <-
1 + 3 <-
4 <-
```

Функции length совершенно незачем знать, что содержит сам список и поэтому не происходит никаких вычислений связанных с ним.
Стоит также упомянуть, что Thunk's всёравно занимают определённую память в программе.

Плюсы ленивых вычислений сводятся к тому, что мы получаем только то, что нам действительно необходимо, всё остальное же игнорируется. За счёт этого мы *Можем* получить достаточно высокую производительность.

Помимо этого ленивые вычисления *Могут* приводить к экономии оперативной памяти, где допустим нужно выполнить лишь 50% необходимой нам работы, вместо 100%.

18. *Реализация функции length*:
```
--------------------------------------------------------------------------------
-- length' = length

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

И вот последний пример ленивости языка Haskell:

```
---------------------------------------------------------------------------
main :: IO()
main = print $ sum $ takeWhile (<10000) $ filter odd $ map (^2) [1..]
---------------------------------------------------------------------------
```

```
> 166650
```

Подобную конструкцию невозможно было бы реализовать в языках программирования со строгими вычислениями, потому что они бы пытались возвести в квадрат все элементы бесконечного списка.

19. *Реализация функции takeWhile*:

```
---------------------------------------------------------------------------
-- takeWhile' = takeWhile

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

Реализация функции takeWhile на языке программирования Си:

```
unsigned int takeWhile (
        _Bool (*func) (double), double array[], const unsigned int length
) {
        unsigned int position = 0;
        while (position < length && (*func) (array[position++]));
        return position - 1;
}
```

Реализация функции takeWhile на языке программирования Python:

```
def takeWhile (func, List):
        if not List: return []
        elif func(head(List)): return [head(List)] + takeWhile(func, tail(List))
        else: return []
```

20. *Реализация функции odd*:

```
-------------------------------------------------------------------------------
-- odd' = odd

odd' :: Integral a => a -> Bool
odd' x | mod x 2 /= 0 = True
odd' _ = False
-------------------------------------------------------------------------------
```

Реализация функции odd на языке программирования Си:

```
_Bool odd (int x) {
        if (x % 2) return 1;
        return 0;
}
```

Реализация функции odd на языке программирования Python:

```
odd = lambda x: True if x % 2 else False
```

А теперь немного о минусах ленивых вычислений.

Основным минусом отложенных вычислений является накопление выражений, как мы видели ранее: 1 + (1 + (1 + (1 + (...)))). Подобную ситуацию называют утечкой пространства (space leak), если она расходует в разы больше памяти, чем ожидается.

Допустим прежде чем вычислить сумму всех элементов списка – ленивые вычисления накапливают выражения. Сами вычисления начинаются только тогда, когда рекурсия останавливается и функция возвращает некий результат.

21. *Реализация функции sum*:

```
-- sum' = sum

sum' :: [Integer] -> Integer
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

При малых списках это не имеет особого значения, но когда списки становятся огромными, тогда наша программа начинает очень быстро и в больших объёмах жрать память.

И соответственно возникает вопрос, как можно избавиться от такого значительного минуса. В данном случае всё сводится к использованию строгих вычислений, так как ленивые здесь крайне неэффективны.

Самым простым способом борьбы с ленью в нашем случае это – подключить расширение компилятора BangPatterns и немного изменить нашу функцию sum:

```
{-# LANGUAGE BangPatterns #-}

sum' :: [Integer] -> Integer
sum' = func 0
        where
                func !n [] = n
                func !n (x:xs) = func (x + n) xs
main :: IO()
main = print $ sum' [1..10^8]
```

```
> 5000000050000000
```

Восклицательный знак (!) указывает на строгость вычисления аргумента. Если вы попытаетесь использовать альтернативную версию функции sum с ленивыми вычислениями, то вычисления займут достаточно много времени, а памяти сожрётся очень много.

Вывод всей этой истории таков, что нужно понимать, где и когда использовать необходимые нам вычисления. В некоторых ситуациях ленивые вычисления могут повысить производительность программы, но в некоторых наоборот замедлить. Это высказывание также относится и к строгим вычислениям.

Реализация функции sum на языке программирования Си:

```
double sum (double array[], int length) {
	double sum_value = 0;
	while (--length >= 0)
		sum_value += array[length];
	return sum_value;
}
```

Реализация функции sum на языке программирования Python:

```
def sum (List):
	if not List: return 0
	else: return head(List) + sum(tail(List))
```

Рекурсивные типы данных

Язык программирования Haskell помимо создания обычных типов данных, позволяет нам создавать рекурсивные типы данных.

Вот простой пример рекурсивного типа данных:

```
---------------------------------------------------------------------------------------
data List a = Empty | Cons a (List a) deriving Show

main :: IO()
main = print $ Cons 5 (Cons 6 (Cons 7 Empty))
---------------------------------------------------------------------------------------
```

```
> Cons 5 (Cons 6 (Cons 7 Empty))
```

Рекурсивным типам данных необходим сигнал их завершения. В данном случае, значение Empty является таким сигналом.

Теперь давайте попробуем немного изменить пример выше, переписав весь механизм работы в функцию, принимающую список элементов:

```
---------------------------------------------------------------------------------------
data List a = Empty | Cons a (List a) deriving Show

func :: [a] -> List a
func [] = Empty
func (x:xs) = Cons x (func xs)

main :: IO()
main = print $ func [5,6,7]
---------------------------------------------------------------------------------------
```

```
> Cons 5 (Cons 6 (Cons 7 Empty))
```

Чтобы легче понять рекурсивные типы данных, следует вспомнить списки и как в них добавляются элементы: 1:2:3:4:5:[]. В данном примере изменим значение Cons на следующее :-: .

```
---------------------------------------------------------------------------------
data List a = Empty | a :-: (List a) deriving Show

func :: [a] -> List a
func [] = Empty
func (x:xs) = x :-: (func xs)

main :: IO()
main = print $ func [1,2,3,4,5]
---------------------------------------------------------------------------------

> 1 :-: (2 :-: (3 :-: (4 :-: (5 :-: Empty))))
```

И как вы видите, сходства со списками имеются.

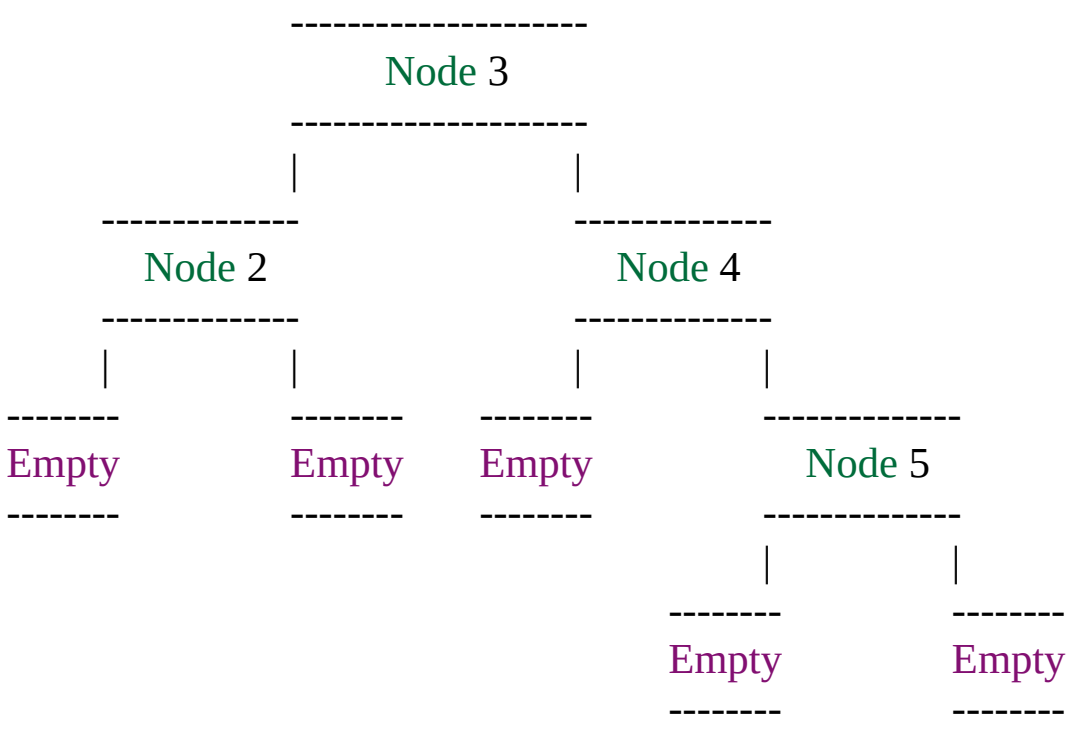
Отличным применением рекурсивных типов данных, является создание деревьев:

```
---------------------------------------------------------------------------------
data Tree a = Empty | Node a (Tree a) (Tree a) deriving Show

tree :: Ord a => a -> Tree a -> Tree a
tree x Empty = Node x Empty Empty
tree x (Node y left right)
        | x == y = Node x left right
        | x <   y = Node y (tree x left) right
        | x >   y = Node y left (tree x right)

main :: IO()
main = print $ foldr tree Empty [5,4,2,3]
---------------------------------------------------------------------------------
```

> Node 3 (Node 2 Empty Empty) (Node 4 Empty (Node 5 Empty Empty))

Новичкам в программировании достаточно сложно понять как работают и функционируют деревья. Тем не менее, я попробую как можно детальнее показать их работу, на примере пошагового алгоритма.

Вот так выглядит наше дерево со списком [5,4,2,3]:



Пошаговый алгоритм работы дерева со списком [5,4,2,3]:

```
foldr tree Empty [5,4,2,3] -> 5:[4,2,3]
   tree 5 (foldr tree Empty [4,2,3]) -> 4:[2,3]
      tree 4 (foldr tree Empty [2,3]) -> 2:[3]
         tree 2 (foldr tree Empty [3]) -> 3:[]
            tree 3 (foldr tree Empty []) -> [] = Empty

            tree 3 Empty = Node 3 Empty Empty

         tree 2 (Node 3 Empty Empty) = Node 3 (tree 2 Empty) Empty =
                  = Node 3 (Node 2 Empty Empty) Empty

      tree 4 (Node 3 (Node 2 Empty Empty) Empty) =
            = Node 3 (Node 2 Empty Empty) (tree 4 Empty) =
            = Node 3 (Node 2 Empty Empty) (Node 4 Empty Empty)

   tree 5 (Node 3 (Node 2 Empty Empty) (Node 4 Empty Empty)) =
      = Node 3 (Node 2 Empty Empty) (Node 4 Empty (tree 5 Empty)) =
      = Node 3 (Node 2 Empty Empty) (Node 4 Empty (Node 5 Empty Empty))
```

Мы также можем создать функцию, которая будет пытаться находить указанный элемент в дереве:

```
treeElem :: Ord a => a -> Tree a -> Bool
treeElem x Empty = False
treeElem x (Node a left right)
      | x == a = True
      | x <  a = treeElem x left
      | x >  a = treeElem x right

main :: IO()
main = print $ treeElem 4 (Node 3 (Node 2 Empty Empty) (Node 4 Empty
(Node 5 Empty Empty)))
```

```
> True
```

В данном же случае алгоритм функции достаточно простой:

```
treeElem 4 (Node 3 (Node 2 Empty Empty) (Node 4 Empty (Node 5 Empty
Empty)))  ->  [ 4 > 3 ]  ->
treeElem 4 (Node 4 Empty (Node 5 Empty Empty)) ->  [ 4 == 4 ]  ->
= True
```

Модули

В языке программирования Haskell есть библиотеки, которые состоят из модулей. В этих модулях находятся какие-либо функции, типы данных, классы типов и тп.

Библиотеки и непосредственно модули созданы для того, чтобы программисты каждый раз не писали то, что уже давно было написано до них.

Также модули помогают нам разграничивать код на определённые участки, тем самым делая его более читаемым.

Есть в Haskell и стандартный модуль, который импортируется в программу неявно. Это модуль Prelude. В нём находятся все те функции, типы данных и классы типов, которые мы ранее использовали. Данный модуль можно явно импортировать в программу: import Prelude

Если вы поставите просто пустые скобки, тогда модуль Prelude не будет импортирован: import Prelude ()

Чтобы из данного модуля импортировать только определённые конструкции, следует поставить скобки и в них написать то, что будет использоваться:

```
-------------------------------------------------------------------------------
import Prelude (IO(), print)

main :: IO()
main = print "hello, world"
-------------------------------------------------------------------------------
```

Существует также полностью противоположный способ импортирования некоторых элементов модуля, при помощи ключевого слова hiding:

```
import Prelude hiding (getLine, putStrLn)

main :: IO()
main = print "hello, world"
```

В данном случае, функции getLine и putStrLn не будут импортированы. Все остальные же функции, из этого модуля, успешно можно будет использовать.

Также допускается запись явного указания модуля:

```
import Prelude (IO(), print)

main :: Prelude.IO()
main = Prelude.print "hello, world"
```

Часто в импортировании модулей используют ключевое слово qualified, которое помогает явно указывать на модуль, используемый в определённой части кода:

```
import qualified Prelude (IO(), print)

main :: Prelude.IO()
main = Prelude.print "hello, world"
```

Отличие обычного метода от метода с ключевым словом qualified, заключено в том, что qualified требует обязательного и явного указания модуля, иначе мы получим ошибку компиляции программы.

Название у модуля может быть большое, а возможно он и вовсе находится в какой-нибудь библиотеке. Тем самым, нужно будет указывать весь путь, вплоть до этого модуля. Нельзя сказать что это удобно, поэтому часто используется ещё одно ключевое слово - as:

```
import qualified Prelude as P (IO(), print)

main :: P.IO()
main = P.print "hello, world"
```

После ключевого слова as может стоять любое название, главное, чтобы оно начиналось с заглавной буквы.

С импортированием модулей мы разобрались, теперь же, настало время писать собственные модули.

Наш модуль назовём как Func, который будет просто содержать функции подобия факториала и фибоначчи.

Первое, что мы сделаем – это создадим файл с именем Func.hs. В отличии от основного файла, название модуля обязательно должно начинаться с заглавной буквы.

Теперь в этот файл запишем следующее:

```
module Func where

factorial :: Integer -> Integer
factorial x | x < 2 = 1
factorial x = (*) x $ factorial $ x - 1
```

```
fibonacci :: Integer -> Integer
fibonacci x | x < 3 = 1
fibonacci x = fibonacci (x-1) + fibonacci (x-2)
```

Первая строка говорит, что мы импортируем всё содержимое этого модуля. Но что, если в нашей программе, есть функции, которые являются промежуточными и созданы только для реализации импортируемых функций?

В таком случае, мы можем указать, какие функции следует импортировать: `module Func (factorial, fibonacci) where`

Содержание файла Func.hs:

```
module Func (factorial, fibonacci) where

factorial :: Integer -> Integer
factorial x | x < 2 = 1
factorial x = (*) x $ factorial $ x - 1

fibonacci :: Integer -> Integer
fibonacci x | x < 3 = 1
fibonacci x = fibonacci (x-1) + fibonacci (x-2)
```

Содержание файла Main.hs:

```
import qualified Func as F (factorial)

main :: IO()
main = print $ F.factorial 6
```

```
> 720
```

В данном случае файлы Main.hs и Func.hs должны находиться в одной директории для успешной компиляции. Как итог мы получим следующий результат программы: 720.

Но что произойдёт в том случае, если модуль Func.hs будет находится в какой-нибудь другой директории? Будет конечно же ошибка компиляции.

Представим такую ситуацию. Мы создали новую директорию под названием Lib, которая находится рядом с файлами Main.hs и Func.hs. Далее мы переместили модуль Func.hs в эту самую директорию Lib.

И чтобы, в данном случае, импортировать модуль Func, нам нужно будет указать директорию Lib: `import qualified Lib.Func as F (factorial)`

А в модуле Func нам также необходимо указать эту самую директорию Lib: `module Lib.Func (factorial, fibonacci) where`

И в завершение раздела стоит сказать, что наш основной файл, также является модулем и его нужно также указывать. Хоть Haskell и сам может понимать, какой файл является основным, а какой относится к библиотекам, тем не менее по стандарту Haskell 2010 следует это явно указывать:

```
-------------------------------------------------------------------------------
module Main where

import qualified Func as F (factorial)

main :: IO()
main = print $ F.factorial 6
-------------------------------------------------------------------------------
```

Создание классов

Классы типов, в языке программирования Haskell, созданы для объединенения всеразличные типов данных под общие функции.

Как пример, давайте создадим класс типов, который будет содержать в качестве аргумента какую-либо страну. Назовём этот класс Country и в нём будут размещены две функции (два метода): location и square:

```
-------------------------------------------------------------------------------
class Country a where
        location :: a -> String
        square :: a -> Int
-------------------------------------------------------------------------------
```

Далее создадим два типа двух стран: Poland и Canada:

```
-------------------------------------------------------------------------------
data Poland = Poland
data Canada = Canada
-------------------------------------------------------------------------------
```

Привяжем эти типы данных к классу Country:

```
-------------------------------------------------------------------------------
instance Country Poland where
        location Poland = "Europe"
        square Poland = 312679

instance Country Canada where
        location Canada = "North America"
        square Canada =  9984670
-------------------------------------------------------------------------------
```

Теперь мы можем создать функцию, в качестве аргументов которой принимаются значения типов привязанных к классу Country.

```
info :: Country a => a -> (String, Int)
info x = (location x, square x)
```

И как итог, попробуем вывести информацию о стране Canada:

```
main :: IO()
main = print $ info Canada
```

```
> ("North America",9984670)
```

Мы также можем создать класс, который будет являться наследником другого класса. В качестве примера создадим новый класс, под названием Europe, который будет являться наследником класса Country:

```
class Country a => Europe a where
        neighbors :: a -> [String]
```

А теперь привяжем к новому классу тип Poland:

```
instance Europe Poland where
        neighbors Poland = ["Belarus", "German"]
```

И на основе этого, можно легко создать функцию, которая работает с типами класса Europe:

```
--------------------------------------------------------------------------------
info_europe :: Europe a => a -> (String, Int, [String])
info_europe x = (location x, square x, neighbors x)

main :: IO()
main = print $ info_europe Poland
--------------------------------------------------------------------------------
```

```
> ("Europe",312679,["Belarus","German"])
```

Суть создания классов наследников в Haskell сводится к уменьшению возможных вариантов типов данных, которые может принимать функция.

Нечистые функции

До этого раздела мы писали код преимущественно чистыми функциями и уделяли внимание только им. Тем не менее, иногда возникает потребность написания кода с нечистыми функциями.

Как мы говорили ранее, чистые функции имеют 2 свойства при которых они вправе считать себя чистыми.

И мы в каждой программе использовали как минимум одну нечистую функцию, которой являлась функция main с типов IO(): main :: IO()

[IO является одной из монад в языке программирования Haskell]

Скобки после IO говорят о том, что функция возвращает нам пустой кортеж. И формально это одно из определений чистой функции, при которой она обязана что-либо возвращать. Но данная функция не является чистой, так как она имеет побочные эффекты.

В языке программирования Haskell также присутствует ключевое слово do, которое позволяет нам использовать императивный стиль написания программ.

```
main :: IO()
main = do
        print "hello"
        print "world"
```

```
> "hello"
> "world"
```

Тем не менее do-нотация является просто синтаксическим сахаром.

Как пример, в данном случае можно было не использовать ключевое слово do, а воспользоваться функцией then (>>) и результат останется тем же. Эта функция просто выполняет последовательно действия:

```
---------------------------------------------------------------------------------
main :: IO()
main = print "hello" >> print "world"
---------------------------------------------------------------------------------
```

Помимо всего прочего, мы можем вводить какие-либо значения с терминала и использовать их в своей программе. Как пример, в этом нам может помочь функция getLine:

```
---------------------------------------------------------------------------------
main :: IO()
main = do
        x <- getLine -- я ввожу строку “hello, world”
        print x
---------------------------------------------------------------------------------

> "hello, world"
```

Стрелка влево (<-) способна создавать идентификаторы и использовать их в нашей программе.

Функция getLine считывает наше сообщение из терминала и после того как мы нажмём Enter наша строка поместится в идентификатор x.

Этот пример также можно было расписать иначе, без использования ключевого слова do и (<-), но с использованием функции bind (>>=):

```
---------------------------------------------------------------------------------
main :: IO()
main = getLine >>= \x -> print x
---------------------------------------------------------------------------------
```

В этом примере можно исключить лямбда-функцию x и результат останется тем же, при условии того, что вводим одни и те же значения:

```
---------------------------------------------------------------------------
main :: IO()
main = getLine >>=  print
---------------------------------------------------------------------------
```

Также существует функция (=<<) аналогичная функции bind (>>=), только её аргументы переставлены местами:

```
---------------------------------------------------------------------------
main :: IO()
main = print =<< getLine
---------------------------------------------------------------------------
```

Теперь зная это попробуем сложить два числа при помощи ввода строки из терминала:

```
---------------------------------------------------------------------------
main :: IO()
main = do
        x <- getLine >>= \x -> return (read x :: Double) -- ввожу 4.3
        y <- getLine >>= \x -> return (read x :: Double) -- ввожу 10.4
        print $ x + y
---------------------------------------------------------------------------

> 14.7
```

Функция return в Haskell’е, не имеет никакого отношения к ключевому слову return в других языках программирования. В данном случае, она просто преобразовывает тип данных T в тип данных IO T.

И с этого момента надо поподробнее.

Функции подобия getLine возвращают не просто тип String, а возвращают IO String.

Стрелки (<-), (=<<), (>>=) способны преобразовывать тип IO String уже непосредственно в тип String.

Функция же подобия return наоборот способна преобразовывать тип данных String в тип IO String.

Таким образом, мы не можем напрямую к идентификатору присвоить какое-либо значение, нужно будет использовать функцию return:

```
---------------------------------------------------------------------------
main :: IO()
main = do
        x <- return 5
        x <- return $ x + 10
        print x
---------------------------------------------------------------------------

> 15
```

Может показаться что идентификатор x является переменной, потому что присутствует способность как хранить, так и изменять значение. Но на самом деле это не переменная. Это аргумент лямбда-функции.
Вот пример, этой же программы без синтаксического сахара:

```
---------------------------------------------------------------------------
main :: IO()
main = return 5 >>= \x -> return (x + 10) >>= \x -> print x
---------------------------------------------------------------------------
```

Зная всё это, мы можем немного переделать нашу программу, со сложением двух чисел, следующим образом, добавив в неё новую функцию input_double:

```
-------------------------------------------------------------------------------
input_double :: String -> IO Double
input_double m = putStrLn m >> getLine >>= return . read

main :: IO()
main = do
        x <- input_double "Number X: " -- ввожу 4.3
        y <- input_double "Number Y: " -- ввожу 10
        putStr "Result: " >> print (x + y)
-------------------------------------------------------------------------------

> Number X:
> 4.3
> Number Y:
> 10
> Result: 14.3
```

Ввод выглядит немного не красивым, так как нужно каждый раз вводить число с новой строки. Это можно изменить при помощи функции putStr.

Но здесь всё не так просто, потому что сначала будет срабатывать функция getLine, а только потом выведутся все строки с результатом:

```
> 4.3
> 10
> Number X: Number Y: Result: 14.3
```

Это можно исправить отключением буферизации. В этом нам поможет модуль System.IO:

```
-------------------------------------------------------------------------------
import System.IO (hSetBuffering, stdout, BufferMode(NoBuffering))

input_double :: String -> IO Double
input_double m = putStr m >> getLine >>= return . read
```

```
main :: IO()
main = do
        hSetBuffering stdout NoBuffering
        x <- input_double "Number X: "
        y <- input_double "Number Y: "
        putStr "Result: " >> print (x + y)
```

```
> Number X: 4.3
> Number Y: 10
> Result: 14.3
```

Помимо функции main мы также можем создавать и другие нечистые функции с типом IO() и также в них использовать do-нотацию:

```
print_hello :: IO()
print_hello = do
        print "hello"
        print "world"

main :: IO()
main = print_hello
```

```
> "hello"
> "world"
```

Подобные нечистые функции также способны быть рекурсивными и принимать в качестве аргументов какие-либо значения, в том числе и значения типа IO():

```
replicate_IO :: Int -> IO() -> IO()
replicate_IO 0 _ = return ()
replicate_IO n act = act >> replicate_IO (n - 1) act
```

```
main :: IO()
main = replicate_IO 5 (print "hello, world")
```

```
> "hello, world"
> "hello, world"
> "hello, world"
> "hello, world"
> "hello, world"
```

В данном случае return останавливает рекурсию и возвращает пустой кортеж, как того и требует тип IO().

Также, просматривая примеры всеразличных программ, можете встретить следующую конструкцию с ключевым словом let, но без связки с in:

```
main :: IO()
main = do
        let x = 5
        let x = 10
        print x
```

Может опять же показаться, что x – это некая переменная. Но если убрать синтаксический сахар, то это будет выглядить следующим образом:

```
main :: IO()
main = let x = 5 in let x = 10 in print x
```

Как итог, стоит сказать, что не надо весь код делать в императивном стиле. Хоть Haskell и даёт нам такую возможность, но её следует использовать только тогда, когда потребуется.

Во всех же остальных случаях стоит использовать только чистые функции, так как они обладают следующим списком плюсов:

- Предсказуемость. Чистая функция будет всегда функционировать одинакого, вне зависимости от того, в каком окружении она работает.

- Лёгкость тестирования программы. Если чистая функция один раз прошла проверку на работоспособность, то со 100% гарантией, она будет работать без ошибок в любых программах.

- Параллельность. Порядок выполнения, для чистых функций, не имеет никакого значения и потому, возникает лёгкость запустить программу с одновременным выполнением функций. Как пример, компилятор ghc способен автоматически распараллеливать выполнение программы.

Ввод и вывод

Язык программирования Haskell использует монаду IO для интеграции операций ввода-вывода в чисто функциональном контексте.

Типы функций вывода:

```
putChar    :: Char -> IO()
putStr     :: String -> IO()
putStrLn   :: String -> IO()
print      :: Show a => a -> IO()
```

Мы уже знаем все функции вывода, за исключением функции putChar:

```
main :: IO()
main = putChar 'A'
```

```
> A
```

Типы функций ввода:

```
getChar       :: IO Char
getLine       :: IO String
getContents   :: IO String
interact      :: (String -> String) -> IO()
readIO        :: Read a => String -> IO a
readLn        :: Read a => IO a
```

Сейчас мы знаем только функцию getLine, все остальные же функции ранее не рассматривали.

Начнём пожалуй по порядку и посмотрим на функцию getChar:

```
---------------------------------------------------------------------------
main :: IO()
main = getChar >>= print -- ввожу строку “ABC”
---------------------------------------------------------------------------

> 'A'
```

Как итог мы получим только первый символ ввода, а все остальные символы будут помещены в буффер. Это можно проверить функцией getLine:

```
---------------------------------------------------------------------------
main :: IO()
main = do
        getChar >>= print -- ввожу строку “ABC”
        getLine >>= print
---------------------------------------------------------------------------

> 'A'
> "BC"
```

Теперь немного о функции getContents. Данная функция схожа с функцией getLine, которая также возвращает строку. Но у функции getContents нет как такового символа выхода, в то время как у getLine есть символ выхода - ‘\n’ (новая строка).

И давайте попробуем реализовать функцию getLine при помощи функции getContents:

```
---------------------------------------------------------------------------
notEndOfString :: Char -> Bool
notEndOfString x = if x /= '\n' then True else False
```

```
getLine' :: IO String
getLine' = getContents >>= return . takeWhile notEndOfString

main :: IO()
main = getLine' >>= print -- ввожу строку “ABC”
```

```
> "ABC"
```

Настала очередь функции interact. Данная функция позволяет нам не только вводить данные, но и выводить их. Создадим программу, которая будет переворачивать вводимую строку:

```
notEndOfString :: Char -> Bool
notEndOfString x = if x /= '\n' then True else False

reverseLine :: String -> String
reverseLine = reverse . takeWhile notEndOfString

main :: IO()
main = interact reverseLine -- ввожу “hello, world”
```

```
> dlrow ,olleh
```

На этапе с функцией readIO, вспомним нашу простенькую программу, которая считывает два числа с потока ввода и выдаёт результат их сложения. В этой программе можно немного изменить функцию input_double и работоспособность программы останется прежней:

```
input_double :: String -> IO Double
input_double m = putStr m >> getLine >>= readIO
```

Вкратце readIO – это просто смесь двух функций: read и return.

И завершающей функцией стандартного ввода будет функция readLn. Эта функция тоже является смесью двух функций, но на этот раз уже: readIO и getLine.

Как итог, наша функция input_double может быть улучшена:

```
-------------------------------------------------------------------------------
input_double :: String -> IO Double
input_double m = putStr m >> readLn
-------------------------------------------------------------------------------
```

Теперь настало время разобрать ещё три функции, которые помогут нам работать с файлами:

```
-------------------------------------------------------------------------------
writeFile    :: FilePath -> String -> IO()
appendFile :: FilePath -> String -> IO()
readFile     :: FilePath -> IO String
-------------------------------------------------------------------------------
```

Тип FilePath просто является синонимом типа данных String:
`type FilePath = String`

Начнём мы пожалуй с создания текстового файла, в котором будет храниться строка “hello”.

```
-------------------------------------------------------------------------------
main :: IO()
main = writeFile "file.txt" "hello"
-------------------------------------------------------------------------------
```

После того, как мы запустим нашу программу, создастся файл с именем file.txt и содержанием “hello”. Данная функция перезаписывает файл.

Теперь воспользуемся функцией appendFile, которая добавляет в конец файла нашу строку, не перезаписывая тем самым содержимое:

```
---------------------------------------------------------------------------
main :: IO()
main = appendFile "file.txt" ", world"
---------------------------------------------------------------------------
```

При запуске нашей программы, содержимое файла дополнится строкой “, world”.

И последняя функция из нашего списка – это readFile (чтение файла):

```
---------------------------------------------------------------------------
main :: IO()
main = readFile "file.txt" >>= print
---------------------------------------------------------------------------

> "hello, world"
```

У всех этих функций есть один минус. Допустим, что произойдёт, если у нас нет прав создавать файл в определённой директории? Или что произойдёт, если мы попытаемся прочитать файл, которого не существует? Правильно, выведится ошибка, а работа программы сразу прекратится.

Чтобы предотвращать возможные ошибки следует использовать функцию catch вместе с типом IOException из модуля Control.Exception.

Пример чтения несуществующего файла без предотвращения возможных ошибок:

```
---------------------------------------------------------------------------
main :: IO()
main = (print =<< readFile "file.txt") >> print "end of file"
---------------------------------------------------------------------------
```

```
> main: file.txt: openFile: does not exist (No such file or directory)
```

В итоге мы не получаем последующую работу программы и прерываем выполнение на моменте ошибки.

Теперь пример чтения несуществующего файла с предотвращением возможных ошибок:

```
-------------------------------------------------------------------------------
import Control.Exception (catch, IOException)

try_open :: (FilePath -> IO String) -> FilePath -> IO String
try_open func path = catch (func path) maybe_err
        where
                maybe_err :: IOException -> IO String
                maybe_err err = return $ show err -- message

main :: IO()
main = (print =<< try_open readFile "file.txt") >> print "end of file"
-------------------------------------------------------------------------------

> "main: file.txt: openFile: does not exist (No such file or directory)"
> "end of file"
```

Строку, с возвращением ошибки, в функции maybe_err, можно изменить на своё сообщение об ошибке: `maybe_err err = return $ "error: read file"`

Мы также можем читать аргументы, которые передаются программе из терминала. В этом нам поможет функция getArgs из модуля System.Environment:

```
-------------------------------------------------------------------------------
import System.Environment (getArgs)

main :: IO()
main = getArgs >>= print
-------------------------------------------------------------------------------
```

Если мы запустим нашу программу со следующими аргументами в терминале: [./main hello world], то получим список из двух строк:

```
> ["hello","world"]
```

Помимо всего прочего, мы можем создавать директории, читать их содержимое и удалять их, а также выполнять команды из под операционной системы. В этом нам помогут такие модули как: System.Directory и System.Process.

Начнём с некоторых функций из модуля System.Directory:

```
-------------------------------------------------------------------------------
createDirectory :: FilePath -> IO()
renameDirectory :: FilePath -> FilePath -> IO()
removeDirectory :: FilePath -> IO()

renameFile :: FilePath -> FilePath -> IO()
removeFile :: FilePath -> IO()

doesDirectoryExist :: FilePath -> IO Bool
doesFileExist :: FilePath -> IO Bool

getDirectoryContents :: FilePath -> IO [FilePath]
getCurrentDirectory :: FilePath -> IO FilePath
getPermissions :: FilePath -> IO Permissions
-------------------------------------------------------------------------------
```

Первое что мы сделаем – это подключим модуль System.Directory:

```
import System.Directory
```

И пойдём по порядку, начиная с функции createDirectory.
Здесь всё элементарно:

```
main :: IO()
main = createDirectory "MyDir"
```

В итоге, у нас создастся директория с именем MyDir.

Далее идёт функция renameDirectory:

```
main :: IO()
main = renameDirectory "MyDir" "NewName_forMyDir"
```

В качестве первого аргумента – имя текущей директории, в качестве второго аргумента – новое имя директории.

И теперь удалим эту директорию:

```
main :: IO()
main = removeDirectory "NewName_forMyDir"
```

Также существуют аналогичные функции для работы с файлами: renameFile и removeFile. Принцип работы тот же, только в качестве аргумента уже имя файла, а не директории.

Иногда бывает полезным узнать, существует ли директория или файл, которые нам необходимы. В этом нам могут помочь две функции: doesDirectoryExist и doesFileExist. Работают они одинакого, но одна функция расчитана для директорий, другая для файлов. Обе функции возвращают значение типа данных Bool. Пример:

```
main :: IO()
main = doesDirectoryExist "MyDir" >>= print
```

```
> False
```

При работе с операционной системой, часто возникает потребность узнать содержимое определённой директории. И в этом поможет функция getDirectoryContents. Принимает она какую-либо директорию в виде строки и возвращает нам список строк элементов этой директории.

```
main :: IO()
main = getDirectoryContents "/home/user/Templates/Haskell" >>= print
```

```
> [".","..","main.hi","main.hs","main.o","main"]
```

Иногда нужно узнать, в какой директории запущена сама программа. И на помощь придёт функция getCurrentDirectory.

```
main :: IO()
main = getCurrentDirectory >>= print
```

```
> "/home/user/Templates/Haskell"
```

Ну и последняя функция из списка – это функция getPermissions. Данная функция предоставляет информацию об уровне доступа к файлу или директории.

```
---------------------------------------------------------------------------
main :: IO()
main = getPermissions "/home/user/" >>= print
---------------------------------------------------------------------------
```

```
> Permissions {readable = True, writable = True, executable = False,
searchable = True}
```

Чтобы получить конкретное значение, мы должны просто указать функцию типа данных Permissions, которая что-либо возвращает:

```
---------------------------------------------------------------------------
main :: IO()
main = getPermissions "/home/user/" >>= \xs -> print $ readable xs
---------------------------------------------------------------------------
```

```
> True
```

И некоторые функции из модуля System.Process:

```
---------------------------------------------------------------------------
callCommand :: String -> IO()
system :: String -> IO ExitCode
readProcessWithExitCode :: FilePath -> [String] -> String ->
                                                IO (ExitCode, String, String)
---------------------------------------------------------------------------
```

Чтобы использовать эти функции, нам конечно же необходимо подключить нужный модуль:

```
-------------------------------------------------------------------------------
import System.Process
-------------------------------------------------------------------------------
```

Самой простой функцией из этого списка, является функция callCommand. Она просто выполняет указанную команду в операционной системе.

```
-------------------------------------------------------------------------------
main :: IO()
main = callCommand "touch file.txt"
-------------------------------------------------------------------------------
```

Как итог, в операционных системах семейства Unix, создастся файл с именем file.txt.

Вторая же функция, system, делает то же самое, но уже способна возвращать результат выполнения команды.

```
-------------------------------------------------------------------------------
main :: IO()
main = system "touch file.txt" >>= print
-------------------------------------------------------------------------------

> ExitSuccess
```

Если же я попытаюсь создать файл в директории, к которой нет доступа у обычного пользователя, то получим сообщение об ошибке:

```
-------------------------------------------------------------------------------
main :: IO()
main = system "touch /usr/file.txt" >>= print
-------------------------------------------------------------------------------

> ExitFailure 1
```

И последняя функция из нашего списка – это readProcessWithExitCode. Она возвращает кортеж трёх значений: успешность выполнения команды (как в system), результат выполнения программы (в отличии от callCommand и system, мы можем этот результат посмотреть в самой нашей программе) и сообщение об ошибке (которое мы должны были бы получить в терминале).

```
main :: IO()
main = readProcessWithExitCode "ls" [] "" >>= print
```

```
> (ExitSuccess,"main\nmain.hi\nmain.hs\nmain.o\n","")
```

В данном случае, команда завершила своё выполнение без ошибок – ExitSuccess и тем самым в третьем элементе кортежа список пустой. Второй же элемент кортежа – это результат выполнения команды.

Допустим, чтобы получить только результат команды, мы можем сделать следующее:

```
main :: IO()
main = readProcessWithExitCode "ls" [] "" >>= \(_,xs,_) -> print xs
```

```
> "main\nmain.hi\nmain.hs\nmain.o\n"
```

Но что если мы хотим получить список строк, а не полноценную строку из терминала? В этом нам может помочь обычная и стандартная функция words:

```
main :: IO()
main = readProcessWithExitCode "ls" [] "" >>= \(_,xs,_) -> print $ words xs
```

```
> ["main","main.hi","main.hs","main.o"]
```

22. *Реализация функции words*:

```
-------------------------------------------------------------------------------
-- words' = words

words' :: String -> [String]
words' [] = []
words' (x:xs) | isSpace x = words' xs
words' xs = takeWhile (not . isSpace) xs : (words' $ dropWhile (not . isSpace) xs)
-------------------------------------------------------------------------------
```

23. *Реализация функции isSpace*:

```
-------------------------------------------------------------------------------
-- isSpace' = isSpace

isSpace' :: Char -> Bool
isSpace' x = if x == ' ' || x == '\t' || x == '\n' then True else False
-------------------------------------------------------------------------------
```

Монады

“Не так страшен чёрт, как его малюют.”

Чтобы рассказать о всеразличных монадах и способах их применения уйдёт конечно не один десяток страниц. В данном же случае я постараюсь рассказать о монадах обобщённо и на практике, не пытаясь углубляться в их теорию.

И стоит начать этот раздел с вопроса. Что представляют собой монады? Монады это типы данных, которые относятся к классу Monad. В этом классе находятся свои конкретные функции (методы). Монады, которые относятся к этому классу, являются некой обёрткой для других типов данных.

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

[Для наглядности, заместо m, можете подставить IO]

И как вы видите, все эти функции мы уже разобрали в монаде IO, за исключением функции fail. Данную функцию достаточно редко используют и фактически она представляет собой функцию error.

В данном разделе будут приведены наиболее часто встречаемые монады, за исключением IO, так как эту монаду мы уже достаточно хорошо изучили в двух предыдущих разделах.

И чтож, давайте приступать к монадам. И начнём со списков.

Кажется, что мы уже достаточно хорошо изучили списки, но есть одна вещь, которую я не упоминал. Списки тоже являются монадой. И вот наглядный пример:

```
import qualified Data.Char as D (toLower)

main :: IO()
main = print $ "HELLO, WORLD" >>= func
        where func = \c -> return $ D.toLower c
```

```
> "hello, world"
```

Но для этой конструкции есть синтаксический сахар, который мы уже знаем. Это генераторы списков:

```
import qualified Data.Char as D (toLower)

main :: IO()
main = print $ [D.toLower x | x <- "HELLO, WORLD"]
```

Вот ещё один пример монады списка:

```
func :: [Int] -> [Int]
func xs = do
        x <- xs
        return $ x ^ 2

main :: IO()
main = print $ func [1..10]
```

```
> [1,4,9,16,25,36,49,64,81,100]
```

И пример с синтаксическим сахаром:

```
main :: IO()
main = print $ [x ^ 2 | x <- [1..10]]
```

Также стоит сказать, что сам тип подобия [a] тоже в какой-то степени является синтаксическим сахаром и выглядит на самом деле следующим образом: [] a. Как пример этого:

```
func :: [] Char -> [] Char
func x = x ++ ", world"

main :: IO()
main = print $ func "hello"
```

```
> "hello, world"
```

Вся суть монады [] списка заключена в том, что она работает со всеми значениями, которые находятся в ней.

Теперь поговорим о монаде Maybe, которая тоже очень часто используется в программах. Она помогает создавать исключения в чистых функциях.

Определение типа данных Maybe следующее:

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

Пример функции lookup, которая возвращает значение с типом Maybe:

```
-------------------------------------------------------------------------------
main :: IO()
main = print $ lookup 'c' [('a',1),('b',2),('c',3),('d',4)]
-------------------------------------------------------------------------------
```

```
> Just 3
```

24. *Реализация функции lookup*:

```
-------------------------------------------------------------------------------
-- lookup' = lookup

lookup' :: Eq a => a -> [(a,b)] -> Maybe b
lookup' _ [] = Nothing
lookup' x ((a,b):xs)
        | x == a = Just b
        | otherwise = lookup' x xs
-------------------------------------------------------------------------------
```

Смысл использования Maybe заключён в том, что мы не знаем конкретно возвращаемого значения функции. Поэтому, если элемент в списке не был найден, тогда вернуть значение Nothing, иначе вернуть Just с переданным элементом.

Помимо этого, тип Maybe привязан к классу Monad:

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

И чтобы удостовериться, что Maybe является одной из монад, можно привести следующий пример:

```
-------------------------------------------------------------------------------
-- Создадим две функции f и g, которые, как пример, просто не должны
-- принимать значения 0 и 100:

f :: Int -> Maybe Int
f 0 = Nothing
f x = Just x

g :: Int -> Maybe Int
g 100 = Nothing
g x = Just x

-- Далее задача такова: пропустить наше число последовательно через
-- эти две функции. В данном случае мы можем вовсе не использовать
-- функцию из класса Monad, а воспользоваться ветвлением:

h :: Int -> Maybe Int
h x = case f x of
        Just n -> g n
        Nothing -> Nothing

-- Но, если воспользуемся функцией класса Monad, тогда кода станет
-- меньше:

h' :: Int -> Maybe Int
h' x = f x >>= g

main :: IO()
main = (print $ h 1) >> (print $ h' 1)
-------------------------------------------------------------------------------

> Just 1
> Just 1
```

Вот ещё подобный пример:

```
increment :: Int -> Maybe Int
increment 0 = Nothing
increment x = Just $ x + 1

main :: IO()
main = do
        print $ Just 5 >>= increment
        print $ Just 0 >>= increment
        print $ Nothing >>= increment
```

```
> Just 6
> Nothing
> Nothing
```

Функцию increment можно было расписать немного иначе:

```
increment :: Int -> Maybe Int
increment 0 = fail ""
increment x = return $ x + 1
```

Существует также монада очень схожая по свойстам с монадой Maybe. Название этой монады – Either. Она подключается с модулем Control.Monad.Except.

И возьмём мы пожалуй тот же пример из монады Maybe:

```
-------------------------------------------------------------------------------
import Control.Monad.Except

lookup' :: Eq a => a -> [(a,b)] -> Either String b
lookup' _ [] = Left "Error"
lookup' x ((a,b):xs)
        | x == a = Right b
        | otherwise = lookup' x xs

main :: IO()
main = print $ lookup' 'c' [('a',1),('b',2),('c',3),('d',4)]
-------------------------------------------------------------------------------
```

```
> Right 3
```

Если мы укажем, допустим, символ ‘z’, тогда функция вернёт значение `Left "Error"`.

Также существует ещё три полезные монады: Writer, Reader и State. Начнём пожалуй с первой.

Данная монада полезна в записи логов.

Как пример, у нас есть функция square (возведение в квадрат). Нам нужно понимать сколько действий она совершила и какие данные на вход поступали. И вот пример подобной реализации:

```
-------------------------------------------------------------------------------
import Control.Monad.Writer

square :: Int -> Writer String Int
square x = tell (" Before:" ++ show x) >> (return $ x ^ 2)

main :: IO()
main = print $ runWriter $ square 2 >>= square >>= square
-------------------------------------------------------------------------------
```

```
> (256," Before:2 Before:4 Before:16")
```

Функция tell способна запаковывать наши данные в возвращаемое значение. Функция runWriter помогает нам вывести итоговый кортеж. Сам же тип Writer выглядит следующим образом:

```
-------------------------------------------------------------------------------
data Writer w a = Writer { runWriter :: (a, w) }
-------------------------------------------------------------------------------
```

Теперь давайте посмотрим на монаду Reader.
Монада Reader позволяет нам легко взаимодействовать с несколькими функциями. Тем не менее, для начала посмотрим на простенький пример, при котором функция будет просто возводить наше число в квадрат:

```
-------------------------------------------------------------------------------
import Control.Monad.Reader

square :: Reader Int Int
square = ask >>= return . (^2)

main :: IO()
main = print $ runReader square 5
-------------------------------------------------------------------------------
```

```
> 25
```

В данном примере, ключевым элементом является функция ask:
```
ask = Reader $ \x -> x
```

Она считывает наше вводимое значение: Reader Int.

Также мы видим функцию runReader. Она определена в типе Reader следующим образом:

```
-------------------------------------------------------------------------------
data Reader r a = Reader { runReader :: r -> a }
-------------------------------------------------------------------------------
```

Теперь же посмотрим на пример со взаимодействием нескольких функций:

```
-------------------------------------------------------------------------------
import Control.Monad.Reader

square :: Reader Int Int
square = ask >>= return . (^2)

plusFive :: Reader Int Int
plusFive = ask >>= return . (+5)

square_and_plusFive :: Reader Int [Int]
square_and_plusFive = do
        one <- square
        two <- plusFive
        return [one, two]

main :: IO()
main = print $ runReader square_and_plusFive 5
-------------------------------------------------------------------------------

> [25,10]
```

Ну и настал черёд монады State. Данная монада схожа с монадой Reader, но помимо чтения, она также может записывать:

```
-------------------------------------------------------------------------------
import Control.Monad.State

square :: State Int [Char]
square = get >>= (\x -> (put $ x ^ 2) >> (return $ show x))
```

```
main :: IO()
main = print $ runState square 5
```

```
> ("5",25)
```

Определение типа State:

```
data State s a = State { runState :: s -> (a, s) }
```

Функция get исполняет роль функции ask в монаде Reader, а функция put схожа с функцией tell в монаде Writer.

Функция put должна принимать такой же тип данных, которые подают в функцию square, в то время как возвращаемое значение, при помощи return, может быть любого типа.

Функторы

Типы данных, которые относятся к классу Functor представляют собой функторы. Класс Functor достаточно лёгкий и в нём содержится всего одна функция (метод):

```
class Functor f where
        fmap :: (a -> b) -> f a -> f b
```

Фактически, любая монада, является автоматически функтором. Посмотрим следующий пример:

```
main :: IO()
main = fmap reverse getLine >>= print -- ввожу строку “hello, world”
```

```
> "dlrow ,olleh"
```

Функция fmap работает точно также как функция map. Но всё дело в том, что map привязана только к спискам:

```
instance Functor [] where
        fmap = map
```

Для большей наглядности, вы можете подставить список [] заместо f и как итог, получите ту самую функцию map:

```
class Functor [] where
        fmap :: (a -> b) -> [] a -> [] b
```

Особенность функции fmap заключена в том, что она способна гибко работать со всеразличными типами данных, которые привязаны к классу Functor.

Как пример этого, можно привести всем нам знакомый тип данных Maybe:

```
instance Functor Maybe where
        fmap f (Just x) = Just (f x)
        fmap f Nothing =  Nothing
```

И если попробуем передать в функцию fmap значение типа Maybe, то действия произойдут только со значением типа, а не с конструктором:

```
main :: IO()
main = print $ fmap f (Just 5)
        where f = (^2)
```

```
> Just 25
```

На основе этого, попробуем создать полностью аналогичную ситуацию как с типом Maybe. Создадим тип под названием Action, далее его привяжем к классу Functor и укажем как нужно работать со значениями типа Action:

```
---------------------------------------------------------------------------------
data Action a = With a | None deriving Show

instance Functor Action where
        fmap f (With x) = With (f x)
        fmap f None = None

main :: IO()
main = print $ fmap f (With 5)
        where f = (^2)
---------------------------------------------------------------------------------
```

Всё, как итог, мы получим точно такой же результат: With 25.

У функции fmap есть инфиксная форма записи - (<$>):

```
---------------------------------------------------------------------------------
main :: IO()
main = print $ (^2) <$> (Just 5)
---------------------------------------------------------------------------------
```

```
> Just 25
```

Также функция fmap способна возвращать другую функцию. Как пример:

```
---------------------------------------------------------------------------------
main :: IO()
main = print $ f 10
        where f = fmap (+2) (+3)
---------------------------------------------------------------------------------
```

```
> 15
```

В данном случае происходить композиция двух функций.

Существуют ещё помимо обычных функторов - Аппликативные функторы. Данные функторы являются типами данных, которые относятся к классу Applicative.

```
class Functor f => Applicative f where
        pure :: a -> f a
        (<*>) :: f (a -> b) -> f a -> f b

        (*>) :: f a -> f b -> f b
        (*>) u v = pure (const id) <*> u <*> v

        (<*) :: f a -> f b -> f a
        (<*) u v = pure const <*> u <*> v
```

Суть таких функторов остаётся той же, но теперь и передаваемая функция f может относится к функторам.

Пример работы функции (<*>):

```
import Control.Applicative

main :: IO()
main = print $ Just (+3) <*> Just 2
```

```
> Just 5
```

Таким способом, мы можем производить действия с функцией, которая привязана к функтору. Как итог, получаем ожидаемый результат.

Вот ещё пара интересных примеров, которые можно сделать при помощи аппликативных функторов:

```
main :: IO()
main = print $ [(*2), (+3)] <*> [1, 2, 3]
```

```
> [2,4,6,4,5,6]
```

В данном случае, каждый элемент списка взаимодействует со списком функций. В итоге, получаем сначала список, пройденный через первую функцию, далее этот же список, пройденный через вторую функцию. В итоге происходит конкатенация списков в один.

```
main :: IO()
main = print $ (^) <$> Just 2 <*> Just 5
```

```
> Just 32
```

В этом примере, функция (<$>) делает значение Just 2 функцией вида Just (2^).

Далее полученный результат взаимодействует со значением Just 5 при помощи функции (<*>).

Итог – Just (2^5) = Just 32.

Функции же стрелочек влево (<*) и вправо (*>) выбирают результат, который у нас будет:

```
---------------------------------------------------------------------------------
main :: IO()
main = do
        print $ Just 5 <* Just 10
        print $ Just 5 *> Just 10
---------------------------------------------------------------------------------

> Just 5
> Just 10
```

Со списками эти функции работают примерно так:

```
---------------------------------------------------------------------------------
main :: IO()
main = do
        print $ [5,6] <* [1,2,3]
        print $ [1,2,3] *> [5,6]
---------------------------------------------------------------------------------

> [5,5,5,6,6,6]
> [5,6,5,6,5,6]
```

Примеры программ

Данный раздел создан для демонстрации лаконичности языка программирования Haskell на примере готовых программ.

I. Нахождение делителей числа, проверка на простое число и вывод определённого количества простых чисел:

```
divisors :: Int -> [Int]
divisors n = [x | x <- [1..n], mod n x == 0]

isPrime :: Int -> Bool
isPrime n | n <= 1 = False
isPrime n = divisors n == [1,n]

primeNums :: Int -> [Int]
primeNums n = 2 : take (n-1) (filter isPrime [3,5..])

main :: IO()
main = do
        print $ divisors 50
        print $ isPrime 23
        print $ primeNums 10
```

```
> [1,2,5,10,25,50]
> True
> [2,3,5,7,11,13,17,19,23,29]
```

II. Нахождение количества символов с частотой встречаемости:

```
-------------------------------------------------------------------------------
import Data.List (group, sort)

quantity :: Ord a => [a] -> [(Int, a)]
quantity = map (\xs -> (length xs, head xs)) . group . sort

frequency :: Int -> [(Int, a)] -> [(Int, a, Float)]
frequency n = map (\(a,b) -> (a, b, fromIntegral a / fromIntegral n * 100))

main :: IO()
main = print $ frequency (length list) (quantity list)
       where list = "AABCAACBCBAACBBAACCABC"
-------------------------------------------------------------------------------

> [(9,'A',40.909092),(6,'B',27.272728),(7,'C',31.818182)]
```

25. *Реализация функции group*:

```
-------------------------------------------------------------------------------
-- group' = group

group' :: Eq a => [a] -> [[a]]
group' [] = []
group' (x:xs) = ([x] ++ takeWhile (x ==) xs) : (group' $ dropWhile (x ==) xs)
-------------------------------------------------------------------------------
```

26. *Реализация функции sort*:

```
-------------------------------------------------------------------------------
-- qsort = sort

qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y >= x]
-------------------------------------------------------------------------------
```

III. Калькулятор с обратной польской записью:

```
calculate :: String -> Double
calculate = head . foldl func [] . words
        where
                func :: [Double] -> String -> [Double]
                func (x:y:zs) "+" = (y + x):zs
                func (x:y:zs) "-" = (y - x):zs
                func (x:y:zs) "*" = (y * x):zs
                func (x:y:zs) "/" = (y / x):zs
                func (x:y:zs) "^" = (y ** x):zs
                func xs y = read y:xs

main :: IO()
main = print $ calculate "1 2 - 4 5 + *"
```

```
> -9.0
```

27. *Реализация функции drop*:

```
-- drop' = drop

drop' :: Int -> [a] -> [a]
drop' _ [] = []
drop' n qq@(_:xs)
        | n /= 0 = drop' (n-1) xs
        | otherwise = qq
```

IV. Шифр Цезаря:

```
-------------------------------------------------------------------------------
import Data.Char (chr, ord, toUpper)

encrypt :: (Int -> Int -> Int) -> Int -> Char -> Char
encrypt f k c
        | elem c ['A'..'Z'] = chr $ (f (ord c) k - 65) `mod` 26 + 65
        | otherwise = c

caesar :: Char -> Int -> String -> String
caesar c k xs = case toUpper c of
        'E'     -> [encrypt (+) k (toUpper x)  | x <- xs]
        'D'     -> [encrypt (-)  k (toUpper x)  | x <- xs]

main :: IO()
main = print $ caesar 'e' 3 "hello, world"
-------------------------------------------------------------------------------

> "KHOOR, ZRUOG"
```

28. *Реализация функции elem*:

```
-------------------------------------------------------------------------------
-- elem' = elem

elem' :: Eq a => a -> [a] -> Bool
elem' _ [] = False
elem' y (x:xs)
        | y == x = True
        | otherwise = elem' y xs
-------------------------------------------------------------------------------
```

29. *Реализация функции toUpper*:

```
-- toUpper' = toUpper

toUpper' :: Char -> Char
toUpper' c =
        let index = getValue $ elemIndex c ['a'..'z']
        in if index /= -1 then ['A'..'Z'] !! index else c
        where
                getValue :: Maybe Int -> Int
                getValue (Just x) = x
                getValue Nothing = -1
```

---

30. *Реализация функции elemIndex*:

```
elemIndex' :: Eq a => a -> [a] -> Maybe Int
elemIndex' _ [] = Nothing
elemIndex' y (x:xs)
        | y == x = Just 0
        | otherwise = 1 + elemIndex' y xs
```

---

31. *Реализация функции chr*:

```
-- chr' = chr

chr :: Int -> Char
chr n = if elem n [32..126]
        then [' '..'~'] !! (n - 32) else ' '
```

---

32. *Реализация функции ord*:

```
-- ord' = ord

ord :: Char -> Int
ord c =
        let index = getValue $ elemIndex c [' '..'~']
        in if index /= -1 then index + 32 else 0
        where
                getValue :: Maybe Int -> Int
                getValue (Just x) = x
                getValue Nothing = -1
```

Реализация функций

33. *Реализация функции all*:

```
-- all' = all

all' :: (a -> Bool) -> [a] -> Bool
all' _ [] = True
all' f (x:xs)
        | f x = all' f xs
        | otherwise = False
```

---

34. *Реализация функции any*:

```
-- any' = any

any' :: (a -> Bool) -> [a] -> Bool
any' _ [] = False
any' f (x:xs)
        | not $ f x = any' f xs
        | otherwise = True
```

---

35. *Реализация функции break*:

```
-- break' =  break

break' :: (a -> Bool) -> [a] -> ([a],[a])
break' f xs = (takeWhile (not . f) xs, dropWhile (not . f) xs)
```

---

36. *Реализация функции concat*:

```
-- concat' = concat

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

---

37. *Реализация функции cycle*:

```
-- cycle' = cycle

cycle' :: [a] -> [a]
cycle' x = x ++ cycle' x
```

---

38. *Реализация функции delete*:

```
-- delete' = delete

delete' :: Eq a => a -> [a] -> [a]
delete' _ [] = []
delete' n (x:xs)
        | n /= x = x : delete' n xs
        | otherwise = xs
```

---

39. *Реализация функции deleteBy*:

```
-- deleteBy' = deleteBy

deleteBy' :: (a -> a -> Bool) -> a -> [a] -> [a]
deleteBy' _ _ [] = []
deleteBy' f n (x:xs)
        | f n x = xs
        | otherwise = x : deleteBy' f n xs
```

---

40. *Реализация функции find*:

```
-- find' = find

find' :: (a -> Bool) -> [a] -> Maybe a
find' _ [] = Nothing
find' f (x:xs)
        | f x = Just x
        | otherwise = find' f xs
```

---

41. *Реализация функции flip*:

```
-- flip' = flip

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

---

42. *Реализация функции foldl1*:

```
-- foldl1' = foldl1

foldl1' :: (a -> a -> a) -> [a] -> a
foldl1' _ [] = undefined
foldl1' _ [x] = x
foldl1' f (x:y:ys) = foldl1' f (f x y : ys)
```

---

43. *Реализация функции foldr1*:

```
-- foldr1' = foldr1

foldr1' :: (a -> a -> a) -> [a] -> a
foldr1' _ [] = undefined
foldr1' _ [x] = x
foldr1' f (x:xs) = f x (foldr1' f xs)
```

---

44. *Реализация функции init*:

```
-- init' = init

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

---

45. *Реализация функции intersperse*:

```
-- intersperse' = intersperse

intersperse' :: a -> [a] -> [a]
intersperse' _ [] = []
intersperse' _ [x] = [x]
intersperse' y (x:xs) = x : y : intersperse' y xs
```

---

46. *Реализация функции iterate*:

```
-- iterate' = iterate

iterate' :: (a -> a) -> a -> [a]
iterate' f x = x : iterate' f (f x)
```

---

47. *Реализация функции last*:

```
-- last' = last

last' :: [a] -> a
last' [] = undefined
last' xs = reverse xs !! 0
```

---

48. *Реализация функции lines*:

```
-- lines' = lines

lines' :: String -> [String]
lines' [] = []
lines' (x:xs) | x == '\n' = lines' xs
lines' xs = takeWhile (/= '\n') xs : (lines' $ dropWhile (/= '\n') xs)
```

---

49. *Реализация функции max*:

```
-- max' = max

max' :: Ord a => a -> a -> a
max' x y | x > y = x | otherwise = y
```

---

50. *Реализация функции maximum*:

```
-- maximum' = maximum

maximum' :: Ord a => [a] -> a
maximum' [] = undefined
maximum' [x] = x
maximum' (x:xs) = max x $ maximum' xs
```

---

51. *Реализация функции min*:

```
-- min' = min

min' :: Ord a => a -> a -> a
min' x y | x < y = x | otherwise = y
```

---

52. *Реализация функции minimum*:

```
-- minimum' = minimum

minimum' :: Ord a => [a] -> a
minimum' [] = undefined
minimum' [x] = x
minimum' (x:xs) = min x $ minimum' xs
```

---

53. *Реализация функции negate*:

```
-- negate' = negate

negate' :: Num a => a -> a
negate' x = -x
```

---

54. *Реализация функции notElem*:

```
-- notElem' = notElem

notElem' :: Eq a => a -> [a] -> Bool
notElem' _ [] = True
notElem' y (x:xs)
        | y == x = False
        | otherwise = notElem' y xs
```

---

55. *Реализация функции null*:

```
-- null' = null

null' :: [a] -> Bool
null' [] = True
null' _ = False
```

---

56. *Реализация функции product*:

```
-- product' = product

product' :: Num a => [a] -> a
product' = foldl1 (*)
```

---

57. *Реализация функции recip*:

```
-- recip' = recip

recip' :: Fractional a => a -> a
recip' x = 1 / x
```

---

58. *Реализация функции repeat*:

```
-- repeat' = repeat

repeat' :: a -> [a]
repeat' x = x : repeat' x
```

---

59. *Реализация функции replicate*:

```
-- replicate' = replicate

replicate' :: Int -> a -> [a]
replicate' x _ | x <= 0 = []
replicate' x y = y : replicate' (x-1) y
```

---

60. *Реализация функции scanl*:

```
-- scanl' = scanl

scanl' :: (a -> b -> a) -> a -> [b] -> [a]
scanl' _ n [] = [n]
scanl' f n (x:xs) = n : scanl' f (f n x) xs
```

---

61. *Реализация функции scanl1*:

```
-- scanl1' = scanl1

scanl1' :: (a -> a -> a) -> [a] -> [a]
scanl1' _ [] = []
scanl1' _ [x] = [x]
scanl1' f (x:y:zs) = x : scanl1' f (f x y : zs)
```

---

62. *Реализация функции scanr*:

```
-- scanr' = scanr

scanr' :: (a -> b -> b) -> b -> [a] -> [b]
scanr' f n [] = [n]
scanr' f n (x:xs) = f x (head $ scanr' f n xs) : scanr' f n xs
```

---

63. *Реализация функции scanr1*:

```
-- scanr1' = scanr1

scanr1' :: (a -> a -> a) -> [a] -> [a]
scanr1' f [] = []
scanr1' f [x] = [x]
scanr1' f (x:xs) = f x (head $ scanr1' f xs) : scanr1' f xs
```

---

64. *Реализация функции signum*:

```
-- signum' = signum

signum' :: (Ord a, Num a) => a -> Int
signum' x | x > 0 = 1
signum' x | x < 0 = -1
signum' _ = 0
```

---

65. *Реализация функции subtract*:

```
-- subtract' = subtract

subtract' :: Num a => a -> a -> a
subtract' x y = (-) y x
```

---

66. *Реализация функции tails*:

```
-- tails' = tails

tails' :: [a] -> [[a]]
tails' [] = [[]]
tails' qq@(_:xs) = qq : tails' xs
```

---

67. *Реализация функции until*:

```
-- until' = until

until' :: (a -> Bool) -> (a -> a) -> a -> a
until' f g x
        | f (g x) = x
        | otherwise = until f g $ g x
```

---

68. *Реализация функции unzip*:

```
-- unzip' = unzip

unzip' :: [(a,b)] -> ([a],[b])
unzip' [] = ([],[])
unzip' ((x,y):zs) = (x:fst(unzip' zs), y:snd(unzip' zs))
```

---

69. *Реализация функции zip*:

```
-- zip' = zip

zip' :: [a] -> [b] -> [(a,b)]
zip' [] _ = []
zip' _ [] = []
zip' (x:xs) (y:ys) = (x,y) : zip' xs ys
```

---

70. *Реализация функции zipWith*:

```
-- zipWith' = zipWith

zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ _ [] = []
zipWith' _ [] _ = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

Конец

Выражаю огромную благодарность авторам следующих трудов:

“Learn You a Haskell for Great Good” (Miran Lipovaca)
“О Haskell по-человечески” (Денис Шевченко)
“Учебник по Haskell” (Антон Холомьёв)

Без ваших книг, не вышла бы на свет и моя книга.

Также, отдельное спасибо тебе, читатель.
Надеюсь эта книга была интересна и полезна.

Связаться со мной можно здесь:

```
--------------------------------------------------------------------------------
module Main where

class Link a where
        link :: a -> String

data Account = YouTube | GitHub deriving Show

instance Link Account where
        link YouTube    = "https://www.youtube.com/c/CryptFunIT"
        link GitHub     = "https://github.com/Number571"

printLinks :: (Show a, Link a) => [a] -> IO()
printLinks [] = return ()
printLinks (x:xs) = (putStrLn $ show x ++ ": " ++ link x) >> printLinks xs

main :: IO()
main = printLinks [YouTube, GitHub] >> putStrLn "Thanks for reading"
--------------------------------------------------------------------------------
```