syntax.lhs

```
> {-# OPTIONS -Wall #-}
```

Так ^ я включаю вывод предупреждений.

```
> -- Это комментарий, кстати.

> {-
> А это многострочный комментарий
> -}

> module Main where
```

Это заголовок модуля. Обычно имя модуля совпадает с именем файла. Модули могут быть вложенными в namespaces, тогда пространствам имён будут соответствовать директории. Т.о. заголовок

```
module Data.List where
```

будет соответствовать такой структуре

```
.
└── Data/
    └── List.hs
```

Импорты

Блок импортов пишется в начале файла. Далее рассмотрим несколько примеров импортов разного вида.

```
> import Prelude hiding (($))
```

Читаем: “импортировать всё содержимое модуля кроме оператора `($)`”

Импортировать конкретно модуль `Prelude` обычно не нужно: этот модуль импортируется неявно. Но если вдруг нужно что-то скрыть при импортировании, можно сделать явный импорт как этот.

```
import Data.List (nub)
```

“импортировать из модуля только функцию `nub`”

```
import qualified Data.List
```

“импортировать сам модуль с полным именем” (это позволит иметь доступ к содержимому по полному имени вроде `Data.List.nub`)

```
import qualified Data.Maybe as M
```

“импортировать модуль под коротким именем M”

Функция main

Этот модуль называется Main, что обязывает его иметь функцию main:

```
> main :: IO () -- это сигнатура функции, т.е. описание её типа
> main = do
>   -- do позволяет в функциях типа IO выполнять
>   -- IO-действия друг за другом
>   print $ f 10
>   -- print имеет тип "Show a => a -> IO ()"
>   -- и позволяет печатать всё, что умеет "отображаться"
>   -- (имеет инстанс класса типов Show)
>   print $ g 3 5
>   -- Оператор $ всего лишь применяет функцию
>   -- слева от него к аргументу справа, но имеет
>   -- самый низкий приоритет, поэтому может служить
>   -- заменой скобок. Данная строчка аналогична строчке
>   -- print (g 3 5)
>   print (f 45 +++ f 100)
>   -- вызов префиксных функций (тех, имя которых пишется перед аргументами)
>   -- всегда имеет больший приоритет, чем применение операторов. Поэтому
>   -- в этой строчке вызовы функции "f" не обёрнуты в скобки. Но в скобки
>   -- обёрнуто всё выражение-аргумент функции print, потому что вызов
>   -- префиксной функции всегда лево-ассоциативен и без скобок выражение
>   -- имело бы такой смысл:
>   -- (((print f) 45) +++ (f 100))
```

Функция `main` использует функцию `f`, объявленную ниже. Это работает потому, что код модулей Haskell не является списком команд. Это всегда набор определений, которые существуют одновременно и могут ссылаться друг на друга.

Функции

Анонимная функция, связанная с именем выглядит так:

```
> lambda = \argument -> argument * 10
```

`bla` — объявление полиморфной функции без аргументов. Функция, в зависимости от контекста, может вернуть и `Int`, и `Float`. И даже пользовательский тип, если этот тип нужным образом подготовлен.

```
> bla = 42
```

Это функция одного аргумента:

```
> f x = x + 1
```

А вот так эта функция “рассахаривается” при компиляции:

```
f = \x -> x + 1
```

Функция двух аргументов внутри состоит из лябмды, которая возвращает лябмду:

```
> g x y = x * y
> -- g = \x y -> (x * y)
> -- g = \x -> \y -> (x * y)
```

Так объявляются инфиксные функции-операторы:

```
> (+++) :: Int -> Int -> Int
> x +++ y = x + y + x + y
```

Ещё их можно объявлять так:

```
(+++) x y = x + y + x + y
```

Так выглядит определение `($)` в модуле `Prelude`:

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

Третья строчка — это указание приоритета оператора и его ассоциативности. По умолчанию все операторы лево-ассоциативны (`infixl`), что в смешанном выражении “`1 +++ 2 +++ 3 +++ 4`” означает “`((1 +++ 2) +++ 3) +++ 4`”,

А у “`$`” ассоциативность правая. Поэтому “`f $ g $ h 1`” соответствует “`f (g (h 1))`”.

Приоритет 0 — самый низкий. Самый высокий приоритет равен девяти.

Условия

```
> sign x =
>   if x < 0
>   then -1
>   else
>     if x > 0
>     then 1
>     else 0
```

Условная конструкция всегда имеет `else` ветку, всегда возвращает значение, тип возвращаемого значения в ветках `then` и `else` должен быть одинаков. Вложенность в случае функции, тело которой является одним выражением, несёт только косметический характер. Я мог бы всё тело функции записать в одну строчку или не указывать отступ у второго условия и это всё равно бы скомпилировалось. Важен только отступ относительно имени функции в определении: его нельзя опускать и писать

```
f x =
x + 1
```

Объявление типов

Это тип-сумма `Foo`, имеющий два значения: `A` и `B`:

```
> data Foo = A | B
```

А вот функция, с ним работающая

```
> fromFoo :: Foo -> String
> fromFoo A = "A"
> fromFoo B = "B"
```

А вот так код функции “рассахаривается”:

```
fromFoo :: Foo -> String
fromFoo x =
  case x of
    A  -> "A"
    B  -> "B"
```

TODO

```
> goodNumber 1                     = True
> goodNumber 7                     = True
> goodNumber x | x < 1000          = False
> goodNumber x | x < 100 && x > 10 = True
> goodNumber _                     = False

> sign' x
>   | x < 0     = -1
>   | x > 0     = 1
>   | otherwise = 0

> matchComplexValue
>   (Just (Just (2, x)), _) = x
> matchComplexValue
>   _                       =
>   error "Oops!"

> l = [1,2,3]
> l' = 1 : (2 : (3 : []))

> len (_:xs) = 1 + len xs
> len []     = 0

> lastElem :: [Int] -> Int
> lastElem []     = error "Oops!"
> lastElem (_:[]) = 42
> lastElem (_:xs) = lastElem xs
```