# STEPIK | Haskell | Module 1

yd

June 14, 2018

## Contents

# 1 Part 1

## 1.1 Evaluation Strategy

In imperative languages a program is a sequence of instructions. These instructions are read and executed in the order in which they are written and the results of execution are stored in memory. Any instruction can acess the results of preceding steps either reading from memory "cell" or writing to it. These *named memory location* correspond to program variables.

Functional languages, such as Haskell, use a different approach. In this approach a program is an **expression**, perhaps a complicated one, and program execution (or *evaluation*) is a process of reduction of this expression. The reduction proceeds until we reach some final, most basic expression which represents the results of the program.

Here is an example of evaluation of an arithmetic expression using a series of reductions:

```
(5 + 4 * 3) ^ 2
(5 + 12) ^ 2
17 ^ 2
289
```

## 1.2 Function Application and Parentheses

Let's start with an example:

```
foo bar
```

This is an application of a function **foo** to its argument `bar`. In other languages you may find the following syntax for function application:

```
foo(bar)
```

In Haskell we do not have to enclose function arguments into parentheses. Parentheses can be used to avoid ambiguous situations when applying functions:

```
acos (cos pi)
3.141592653589793
```

Here we first apply **cos** to the argument `pi` and after that apply **acos** to the result of the first application. Without the parentheses the expression

```
acos cos pi
```

would mean "apply acos to cos and the result of this application apply to pi".

Functions of several variables are treated in the following way:

```
f x y
```

is an example of a function **f** applied to its arguments $x$ and $y$. Consider a function **max** from the standard Haskell library:

```
max 5 42
42
```

which is a function of two variables. Note that trying to call this function like this

```
max (5, 42)
```

will results in error, since we apply a function **max** to a single argument (a pair of numbers), not two arguments.

## 1.3 Partial Function Application

Note the following interesting property of parentheses in Haskell:

```
(max 5) 42
42
```

It means that two expressions

```
max 5 42
((max 5) 42)
```

are equivalent which is related to the fact that in Haskell function application is ***left associative***.

The expression

```
(max 5)
```

is an example of **partially applied** function. In this particular case a function of two arguments is applied to only one. The result is a function of a single argument which can be used just like any other function of a single argument. For example:

```
3 + sin 42
2.0834784520843663

3 + (max 5) 42
45
```

Here **sin** and **(max 5)** are both functions of a single argument.

Partial function application is a very powerful tool. It allows us to view any function of N arguments as a function if just one argument. When a single argument is provided to this function it returns another function which accepts (N-1) arguments. The same view can be applied to the resulting function of (N-1) arguments. Thus we may view all functions in Haskell as accepting only one argument.

## 1.4 Quiz 1

In Haskell standard library there is a function **logBase** which calculates logarithm in an arbitrary base. This is a function of two arguments – the base and the number. Choose the expressions which correspond to logarithm of 8 in the base 2?

```
[1] (logBase, 2, 8)
[2] logBase (2, 8)
[3] logBase (2 8)
[4] logBase 2 8
[5] (logBase 2) 8
```

**Answer**: [4, 5].

## 1.5 Defining Functions

```
fun param = body
```

In Haskell users can define their own functions by providing the following:

- Name of the function
- Function parameters/arguments
- Body of the function.

The body of the function is separated from name and parameters by the equality sign =.

Consider the following example:

```
sumSquares x y = x ^ 2 + y ^ 2
```

Here we define a function of two arguments, that returns the sum of squares of all arguments.

The body of the function can use the parameters mentioned before the = sign (such as $x$ and $y$ in the example of **sumSquares**), any built-in function (such as + and ^ used in the exampel above) or any other function, defined by a user earlier.

Note that Haskell is case-sensitive and requires that the name of a function and the names of the variables start with a lower-case letter. The names of the types of data (such as **Integer** or **Bool**, for example) start with a capital letter. The name of a variable may contain any letter, number, underscore or a quotation mark (').

```
value2_res' = 4
rock'n'roll = 42
```

Here we defined two functions of zero arguments, which always return the same value.

## 1.6 Exercise 1

Write a function **lenVec3** which will calculatet the lengths of a three-dimentional vector. Assume that the function accepts three arguments, specifying the location of the head of the vector and the tail of vector is at origin. To calculate the square root from a number use the function **sqrt** from the standard library.

**Solution**

```
lenVec3 x y z =  sqrt (x ^ 2 + y ^ 2 + z ^ 2)
```

## 1.7 Function Purity

An important property of functions in functional programming languages is their **purity**. A function is called pure if its return value is completely determined by its arguments and not affected by any other information. All data that may change the return value of a function must be provided as an explicit argument.

As a consequence of purity, a function with no arguments is a constant:

```
fortyTwo = 39 + 3

fortyTwo
42
```

How is it then possible to have a function which returns random numbers? For this purpose Haskell has a special mechanism that returns values in a kind of **container** called **IO**, and it will be discussed later.

## 1.8 Conditional Expression

Many programming languages provide a way to branch program execution based on some condition:

```
if (condition) {
    do_steps_1
} else {
    do_steps_2
}
```

Haskell also has similar construction called **conditional expression**. It has the form

```
if condition then expression1 else expression2
```

For example, we can defined a function using conditional expressions

```
f x = if x > 0 then 1 else (-1)

f 5
1
f (-5)
-1
```

Notice that we wrote (-5) to denote negative number. The expession

```
f -5
```

will be interpreted as an attempt to sutract 5 from **f** and will result an error.

It is important to note that in Haskell **both** parts of the **if** expression must be provided and both parts must have the same type. In the definition of the function **f** in both branches of the **if** expression we return a number.

Since **if-then-else** is an *expression*, it can be used like any other expression to build more complex expressions:

```
g x = (if x > 0 then 1 else (-1)) + 3

g 5
4
g (-7)
2
```

## 1.9 Exercise 2

Implement a function **sign** which returns 1 if its argument is a positive number, (-1) if it is a negative number and 0 if the argument is zero. **Solution(s)**

```
sign x = if (x > 0) then 1 else (if (x == 0) then 0 else (-1))

sign x = if x == 0 then 0 else x / abs x
```

## 1.10 Defining Function Using Partial Application

Partial application can be used to define new functions. Consider first the following function definition:

```
max5 x = max 5 x

max5 4
5

max5 42
42
```

The function **max5** is a function of one argument. This function returns 5 if the argument is less than 5, otherwise it returns the argument. Using partial we can define a function similar to **max5**:

```
max5' = max 5

max5' 4
5

max5' 42
42
```

Here we defined **max5'** as partially applied **max**. This approach of defining a function without explicitely writing argument(s) – as we did for **max5** – is called **point-free** style (an argument is called a **point** in this context). This style is quite often used in Haskell.

Let us consider a more involved example:

```
discount limit perc sum = if sum >= limit them sum * (100 - perc) / 100 else sum
```

We defined a function with three arguments **discount** that calculates a discount on a purchase if a *sum* of money spent on a purchase is greater of equal to specified *limit*. The discount is determined from the second argument – *perc* (percentage).

The order of the arguments in this definition has been chosen on purpose. The first two arguments correspond to certain "technical details" of the discount scheme. Their values are not expected to vary as often as the value of the the last argument. This makes it easier to partially apply the function **discount** to define *special* discount functions:

```
standardDiscount = discount 100 5

standardDiscount 200
190
```

```
standardDiscount 90
90
```

This defined a function **standardDiscount** which calculates a 5% discount for any purchase over 100.

## 1.11 Quiz 2

Imagine that we need to develop an interface for translation between different natural languages. We want to have a function **translate** that accepts three parameters: *text*, *languageFrom*, and *languageTo*. What is the proper order of these parameters if we want to define special functions **translateFromSpanishToRussian**, **translateFromEnglishToRussian**, and **translateToRussian**.

```
[1] translate languageTo languageFrom text
[2] translate languageTo text languageFrom
[3] translate languageFrom languageTo text
[4] translate text languageTo languageFrom
[5] translate languageFrom text languageTo
[6] translate text languageFrom languageTo
```

**Answer: [1]**