# Functional programming, Seminar No. 2

Danya Rogozin Lomonosov Moscow State University, Serokell ÖÜ

Higher School of Economics Faculty of Computer Science



# Bindings

The equality sign in Haskall denotes binding:

```
fortyTwo = 42
coolString = "coolString"
```

Local binding with the let-keyword:

```
fortyTwo = \mathbf{let} number = 43 \mathbf{in} number -1
```

### Function definitions

#### Functions are also defined as bindings:

add  $\times$  y = x + y userName name = "Username: " ++ name id  $\times$  = x

#### The same functions defined via lambda:

add =  $\xy - \xy = \xy$ 

## Function application

As in lambda calculus, function application is right associative by default

```
\begin{cases}
- \\
5 & \text{foo } x \text{ y } z = f \text{ x y } z = ((f \text{ x}) \text{ y}) z
\end{cases}
```

One may use the dollar infix operator in order to avoid brackets overuse. For example, the following functions have exactly the same behaviour:

- function  $f \times y z = f((x y) z)$ function1  $f \times y z = f \$ \times y \$ z$



# Currying and partial application

Let us recall the function add once more:

add 
$$x y = x + y$$

Here is an example of a partial application in the following GHCi session

```
add x y = x + y
addFive = add 5
twentyEight = addFive 23
-28
```

Partial application is well-defined since all many-argument functions in Haskell are curried by default.



## Immutability and laziness

In Haskell, values are immutable. For example, the following code doesn't terminate since it yields an infinite loop in contrast to Python or JS:

$$\begin{array}{ll}
1 & x = 5 \\
2 & x = x + x
\end{array}$$

The following program doesn't fail since our semantics is lazy:

seventyTwo = const 72 undefined



### Recursion

The straightforward recursion:

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

The factorial function implemented via so-called tail recursion:

```
tailFactorial n = helper 1 n
where
helper acc x = if x > 1
then then helper (acc * x) (x - 1)
else acc
```

### Guards

Let us take a look at the factorial implementation via guards:

```
tailFactorial n = helper 1 n

where

helper acc \times \mid \times > 1 = helper (acc \times \times) (x - 1)

otherwise acc
```

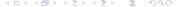
## Basic datatypes

#### The basic datatypes are:

- Bool: Boolean values
- Int: Bounded integer datatype
- Integer: Unbounded integer datatype
- Char: Unicode characters
- (): Unit value datatype
- If a and b are types, then a -> b is a type
- If a and b are types, then (a,b) is a type
- If a is a type, then [a] is a type

A type declaration has the following form:

term :: type



## Function declaration with datatypes

Let us recall the examples of function declarations:

```
add \times y = x + y
userName name = "Username: " ++ name
id \times = x
```

One may annotate these functions with types as follows:

```
add :: Int -> Int -> Int
add x y = x + y

userName :: String -> String
userName name = "Username: " ++ name

id :: Char -> Char
id x = x
```

Note that such calls as userName 5 or id ''hello stewart'' cause type errors.