Functional programming, Seminar No. 2

Danya Rogozin
Lomonosov Moscow State University,
Serokell OÜ

Higher School of Economics
Faculty of Computer Science

Bindings

The equality sign in Haskall denotes binding:

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

Local binding with the `let`-keyword:

```haskell
fortyTwo = let number = 43 in number - 1
```

Function definitions

Functions are also defined as bindings:

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

The same functions defined via lambda:

```haskell
add = \x y -> x + y
userName = \name -> "Username: " ++ name
id = \x -> x
```

Function application

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

```
{-
foo x y z = f x y z = ((f x) y) z
-}
```

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 x y z = f ((x y) z)
function1 f x y z = f $ x y $ z
```

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:

```haskell
x = 5
x = x + x
```

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

```haskell
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
```

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

```
tailFactorial n = helper 1 n
  where
  helper acc x | x > 1 = helper (acc * x) (x - 1)
               | otherwise acc
```

Currying and partial application

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:

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

One may annotate these functions with types as follows:

```haskell
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.