Functional Programming

Lecture 3 — Defining functions

Pedro Vasconcelos
DCC/FCUP

2025

Defining functions

We can define new simple functions using equations and pre-defined functions.

```
lowercase :: Char → Bool
lowercase c = c>='a' && c<='z'

factorial :: Integer → Integer
factorial n = product [1..n]
```

Conditional expressions

A condition between two alternative can be written using ‘if. . . then. . . else’.

```
absoluteValue :: Float → Float
absoluteValue x = if x>=0 then x else -x
```

Conditional expressions

A condition between two alternative can be written using 'if. . . then. . . else'.

```
absoluteValue :: Float → Float
absoluteValue x = if x>=0 then x else -x
```

Conditional expressions can be nested.

```
classify :: Int → String
classify x = if x>0 then "positive" else
               (if x==0 then "zero" else "negative")
```

Unlike imperative languages like C, Java or Python:

- 'if. . . then . . . else' is an expression and not a statement
- the 'else' branch is mandatory

Guards

We can use equation guards instead of conditional expressions:

```
absoluteValue :: Float → Float
absoluteValue x | x>=0      = x
                | otherwise = -x
```

```
classify :: Int → String
classify x | x>0       = "positive"
           | x==0      = "zero"
           | otherwise = "negative"
```

Guards (cont.)

- Guards are tested in the top-to-bottom order
- The result is the first true alternative
- A function will be undefined if none of its guards is true (runtime exception)
- The name ‘otherwise’ is a synonym for `True`

Guards (cont.)

Local bindings scope over the conditions if the keyword where is aligned with the the guards.

```
-- roots of a 2nd degree polynomial
roots :: Float → Float → Float → [Float]
roots a b c
     | delta>0    = [(-b+sqrt delta)/(2*a),
                     (-b-sqrt delta)/(2*a)]
     | delta==0   = [-b/(2*a)]
     | otherwise = []
     where delta = b^2 - 4*a*c
```

Guards (cont.)

We can also define local bindings using 'let...in...'. This way the scope of the definition does not extend to other alternatives.

```
roots :: Float → Float → Float → [Float]
roots a b c
     | delta>0    = let r = sqrt delta
                    in [(-b+r)/(2*a),(-b-r)/(2*a)]
          -- r scopes over the above expression
     | delta==0   = [-b/(2*a)]
     | otherwise = []
     where delta = b^2 - 4*a*c
```

Pattern matching

We can also define functions by several equations using patterns to distinguish alternatives.

```
not :: Bool → Bool
not True = False
not False = True

(&&) :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

(Note: these functions are part of the Prelude)

Pattern matching (cont.)

A shorter definition for (&&):

```
(&&) :: Bool → Bool → Bool
False && _ = False
True  && x = x
```

- The pattern "_" matches any value
- The pattern variable *x* can be used in the right hand side
- The definition above ignores the second argument if the first is `False`

Pattern matching (cont.)

It is an error to repeat pattern variables:

```
x && x = x      -- ERROR
_ && _ = False
```

We can instead use a guard to impose an equality constraint:

```
x && y | x==y = x
_ && _        = False
```

Patterns on tuples

Example: the `fst` and `snd` functions project the first and second element of a pair.

```
fst :: (a,b) → a
fst (x,_) = x

snd :: (a,b) → b
snd (_,y) = y
```

These functions are also defined in the Prelude.

List constructors

All lists are constructed by adding elements one-by-one to the empty list using "`:`" (read *cons* for "construtor").

```
[1, 2, 3, 4]   =   1 : (2 : (3 : (4 : [])))
```

Pattern over lists

We can use a pattern `x:xs` to decompose a list.

```
head :: [a] → a
head (x:_) = x     -- first element

tail :: [a] → [a]
tail (_:xs) = xs   -- remaining elements
```

Pattern over lists (cont.)

The parenthesis around the pattern are necessary because application has higher precedence than operators:

```
head x:_ = x      -- ERROR

head (x:_) = x    -- OK
```

The pattern `x:xs` matches only non-empty lists.

```
ghci> head []
```

ERROR

Patterns over integers

Examples: testing if a an integer is “small” (0, 1 or 2).

```
small :: Int → Bool
small 0     = True
small 1     = True
small 2     = True
small _     = False
```

The final equation matches any remaining cases (“catch all”).

Case expressions

Instead of pattern matching with equations we can use 'case…of…':

Example:

```
null :: [a] → Bool
null xs = case xs of
            [] → True
            (_:_) → False
```

Case expressions (cont.)

Patterns in case expressions are tried in top-to-bottom order. Hence, the following definition is equivalent to the one above:

```
null :: [a] → Bool
null xs = case xs of
              [] → True
              _  → False
```

Lambda expressions

We can define an *anonymous function* using a *lambda expression*.

Example:

```
\x → 2*x+1
```

is a function that maps a value x to $2x + 1$.

This notation is based on the λ-calculus, the mathematical formalism that is the theoretical basis for functional programming.

The backslash (\) was chosen because it is similar to a lowercase lambda (λ).

Lambda expressions (cont.)

We can a use lambda-expression just an named function.

```
ghci> (\x -> 2*x+1) 1
3

ghci> (\x -> 2*x+1) 3
7
```

Why lambda expressions?

Lambda expressions allow us to define functions that return other functions.

In particular, they allow explaining the use of *currying* for handling multiple arguments.

Example:

```
add x y = x+y
```

is equivalent to

```
add = \x → (\y → x+y)
```

Why lambda expressions? (cont.)

Lambda expressions are also useful for avoid the need to give a name to short functions that we are going to use only once.

Example: usando the `map` function (that applies a function to every value in a list); instead of writing

```
squares_1_to_10 = map f [1..10]
      where f x = x^2
```

we can write

```
squares_1_to_10 = map (\x → x^2) [1..10]
```

Operator sections

Recall that any binary operator can be used as binary function by wrapping it in parenthesis.

```
ghci> (+) 1 2
3
ghci> (++) "Abra" "cadabra!"
"Abracadabra!"
```

Operator sections (cont.)

We can obtain a function from an operator by providing the left or right argument inside the parenthesis:

```
ghci> (+3) 2
5
ghci> (/2) 1
0.5
ghci> (++"!!!") "Bang"
"Bang!!!"
```

- Expression of the form `(x` $\otimes$`)` and `(`$\otimes$ `x)` are called sections
- They are a shorter notation for the corresponding lambda expression

$$\begin{aligned} (x \otimes) &\equiv \backslash y \to x \otimes y \\ (\otimes x) &\equiv \backslash y \to y \otimes x \end{aligned}$$

Examples

```
(1+)        \x → 1+x
(2*)        \x → 2*x
(^2)        \x → x^2
(1/)        \x → 1/x
(++"!!")    \x → x++"!!"
```

Hence we can re-write the previous example

```
squares_1_to_10 = map (\x → x^2) [1..10]
```

in an even more succint way:

```
squares_1_to_10 = map (^2) [1..10]
```