Functions in Haskell

- Read also Chapter 4 of the text book "Programming in Haskell.

Overview

In this lesson, we study various ways how functions can be defined in Haskell. We will study the following ways:

1. Composition of existing functions
2. Conditionals (`if _ then _ else _`)
3. Guarded equations
4. Pattern matching
5. Lambda expressions

At the end, we will also look at **operators** (infix function symbols such as `++`), and how to turn them into functions.

Composing functions

A video for this section, including explanation of the exercise, is here.

We can compose existing functions to get new ones. For instance:

```
removeLast :: [a] -> [a]
removeLast xs = reverse (tail (reverse xs))

removeElem :: Int -> [a] -> [a]
removeElem n xs = removeLast (take n xs) ++ drop n xs
```

Exercise: Using the functions above, write a function that removes both the first and the last element of a list.

Conditionals

Haskell provides `if _ then _ else _`. It is typed `Bool -> a -> a -> a`, polymorphically.

```
abs' :: Integer -> Integer
abs' n = if n >= 0 then n else -n
```

Note: The `else` branch is mandatory.

We can nest `if _ then _ else _`:

```
howMuchDoYouLikeHaskell :: Int -> String
howMuchDoYouLikeHaskell x = if x < 3 then "I dislike it!" else
                                if x < 7 then "It's ok!" else
                                  "It's fun!"
```

This is difficult to read, however; guarded equations (see below) can be more pleasant to read. We will avoid conditionals.

Exercise: Read the discussion about `if _ then _ else _` on the Haskell wiki.

Guarded equations

A video for this section, including explanation for the exercise, is here.

Guarded equations are an alternative to `if _ then _ else _` expressions. They are often more readable:

```
abs :: Int -> Int
abs n | n >= 0    = n
      | otherwise = -n
```

Here, `n >= 0` and `otherwise` are called **guards**; they are Booleans. The function returns the value after the **first** guard that evaluates to `True`.

Guarded equations are more convenient to use than `if _ then _ else _`:

```
howMuchDoYouLikeHaskell2 :: Int -> String
howMuchDoYouLikeHaskell2 x | x < 3       = "I dislike it!"
                           | x < 7       = "It's ok!"
                           | otherwise   = "It's fun!"
```

Exercise: Using guarded equations, write a function of type `Int -> Int -> Bool` that returns `True` if the first argument is greater than the second and less than twice the second.

Pattern matching

Pattern matching analyzes the input according to how it is built. The input is matched against a sequence of patterns; the first pattern that matches determines the output of the function.

Overview

There are only two possibilities for what a Boolean value can be: `True` or `False`. It is hence sufficient to have patterns for these two cases:

```
notB :: Bool -> Bool
notB False = True
notB True = False
```

There is only one way to make a pair:

```
swap :: (a, b) -> (b, a)
swap (x,y) = (y,x)
```

There are two ways to make a list:

```
isEmpty :: [a] -> Bool
isEmpty []     = True
isEmpty (x:xs) = False
```

We will look at all of these in detail now.

On Booleans

One of the simplest patterns is to match for Booleans.

If the input is just one Boolean, there are only two patterns:

```
notB' :: Bool -> Bool
notB' False = True
notB' True = False
```

If a function takes two Booleans as input, there are 2^2 = 4 patterns:

```
andB :: Bool -> Bool -> Bool
andB True True = True
andB True False = False
andB False True = False
andB False False = False
```

The last three patterns can be combined. Here, the wildcard pattern `_` matches anything, and discards it:

```
andB' :: Bool -> Bool -> Bool
andB' True True = True
andB' _ _       = False
```

There is a difference between these two versions: in the latter, if the first argument is `False`, then the second argument does not need to be evaluated: `False` is returned immediately.

In the next example, the pattern `b` matches anything. However, in contrast to `_`, **we can use `b`** on the right-hand side of `=`:

```
andB'' :: Bool -> Bool -> Bool
```

```haskell
andB'' True b  = b
andB'' False _ = False
```

Exercise: Write a function `orB :: Bool -> Bool -> Bool` that returns `True` if at least one argument is `True`.

Non-exhaustive patterns

Consider the following example:

```haskell
isTrue :: Bool -> Bool
isTrue True = True
```

Question: What will `isTrue False` evaluate to?

Answer: This is a non-exhaustive pattern, and `isTrue False` will raise an exception:

```
*Main> isTrue False
*** Exception: defining-functions.hs:36:1-18: Non-exhaustive patterns in function isTrue
```

We can choose to throw a custom-made exception instead:

```haskell
isTrue' :: Bool -> Bool
isTrue' True = True
isTrue' False = error "not True"
```

On tuples

If the function we are defining expects as input a **tuple**, we can match against the individual components:

```haskell
fst :: (a,b) -> a
fst (x,y) = x
```

We actually don't use `y` in the output of the function, so we can use `fst (x,_) = x` instead. Similarly,

```haskell
snd :: (a,b) -> b
snd (_,y) = y
```

This generalizes to tuples of three or more components:

```haskell
third :: (a, b, c) -> c
third (_, _, z) = z
```

We can match several tuples at the same time:

```haskell
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

Exercise: Write a function `swap :: (a, b) -> (b, a)` that swaps the elements of a pair.

On lists

See also this video.

All lists are built by prepending `(:)`, successively, elements to an existing list, starting with the empty list `[]`. That means, the list `[1, 2, 3]` has been obtained as `1:[2,3]`, etc. - `[1, 2, 3]` is short for `1:(2:(3:[]))`. In other words, every list in `[a]` is either

1. the empty list; or
2. of the form `x:xs` for `x :: a` and `xs :: [a]`.

```haskell
isEmpty' :: [a] -> Bool
isEmpty' [] = True
isEmpty' (x:xs) = False
```

We are not actually using `x` or `xs` in the output of the second pattern, so we can write the function more simply as

```haskell
isEmpty'' :: [a] -> Bool
isEmpty'' [] = True
isEmpty'' (_:_) = False
```

Note that the parentheses around `_:_` in the second pattern are necessary!

We can write more complex list patterns. To return the second element of a list:

```
sndElem :: [a] -> a
sndElem (_:x:_) = x
```

Case expressions

The aforementioned patterns are special forms, for Booleans, and lists. The general form for such pattern matching is via `case` expressions:

```
isEmpty2 :: [a] -> Bool
isEmpty2 x = case x of [] -> True
                       (_:_) -> False
```

Here, it is important that all the patterns are exactly aligned, i.e., the `[]` and `(_:_)` must start in exactly the same column.

Lambda expressions

Lambda expressions are **nameless** functions. They are particularly useful in **higher-order functions**, which will be discussed in a later lesson. A video accompanying this section is here.

Lambda expressions are of the form `\<input variables> -> <output>`. For instance, we can define a function that returns its double as `\x -> 2 * x`. Here, the input variable is indicated by the backslash `\`. After the arrow `->`, the output of the function is specified. (`\` stands for the greek letter λ (lambda), see Wikipedia). Thus, the following definitions are equivalent:

```
double :: Int -> Int
double x = 2 * x

double' :: Int -> Int
double' = \x -> 2 * x
```

Lambda expressions can have **several** input variables:

```
mult :: Int -> Int -> Int
mult x y = x * y

mult' :: Int -> Int -> Int
mult' = \x y -> x * y
```

Here, the second variant is a short form of

```
mult'' :: Int -> (Int -> Int)
mult'' = \x -> (\y -> x * y)
```

Just like a pattern can ignore (part of) the input, a lambda expression can ignore its input

```
alwaysZero :: Bool -> Int
alwaysZero = \_ -> 0
```

One important application of lambda expressions are **higher-order functions**, where functions are arguments to other functions. Consider

```
apply :: (a -> b) -> a -> b
apply f x = f x
```

```
*Main> apply (\_ -> 5) 'r'
5
*Main> apply (\ x -> if x < 0 then "Less than zero!" else "Greater or equal than zero!") (-3)
"Less than zero!"
```

Operators and sections

There is also a video on operators and sections.

When a function has two arguments, such as `(:)`, we can write it infix, between its two arguments. A function that is used infix (hence necessary binary) is called an **operator**.

1. Any binary function can be turned into an operator by enclosing it in backticks. E.g. `div 7 2` can be written ``7 `div` 2``.
2. Conversely, any operator can be used prefix by enclosing it in parentheses, e.g., `(:) 1 [2,3]`.

Every operator `⊗` with inputs of type `a` and `b` and output of type `c` gives rise to three **sections**:

1. `(⊗) :: a -> b -> c`. Here, `(⊗) = \x y -> x ⊗ y`.
2. `(x ⊗) :: b -> c`, where `x :: a`. Here, `(x ⊗) = \y -> x ⊗ y`.
3. `(⊗ y) :: a -> c`, where `y :: b`. Here, `(⊗ y) = \x -> x ⊗ y`.

Sections can be used to concisely define functions:

```
square :: Int -> Int
square = (^2)

reci :: Fractional a => a -> a
reci = (1 /)
```

Remarks:

1. An operator `⊗` by itself is not a valid Haskell expression: it needs to be used as a section, e.g., `(⊗)`.
2. Sections are useful when programming with higher-order functions (cf. later lesson.).

Exercises

(Adapted and expanded from the book "Programming in Haskell)

1. Define three variants of a function `third :: [a] -> a` that returns the third element in any list that contains at least this many elements, using
   1. `head` and `tail`
   2. list indexing `!!`
   3. pattern matching
2. Define a function `safetail :: [a] -> [a]` that behaves like tail except that it maps `[]` to `[]` (instead of throwing an error). Using `tail` and `isEmpty :: [a] -> Bool`, define `safetail` using
   1. a conditional expression
   2. guarded equations
   3. pattern matching

Quiz time

Test your understanding by taking this quiz. Don't worry, it is not marked, and you can take it as many times as you want.

See also

1. Chapter 3, "Synax in Functions" of "Learn You a Haskell"
2. Haskell Wiki on Sections

Summary

1. We have seen several ways to define functions: composition, conditionals, guard equations, pattern matching, lambda expressions.
2. When patterns are not exhaustive, functions raise an exception whenever no pattern matches. To avoid this, one may use a catch-all `otherwise` pattern at the end.
3. Any pattern matching can be expressed using a `case` expression.
4. Anonymous functions can concisely be written using lambda expressions.