Tutorial 1B: Basics

In this tutorial we will look at some basic concepts in Haskell: functions, types, recursion, and lists. You can use the lecture slides, available on Moodle, as a function reference. To set up your Haskell interpreter, please do the first tutorial first.

Load the file `1B-Basics.hs` into `GHCi`, and open it in the text editor.

Error & undefined

In your file `1B-Basics.hs` you should see the code:

```
square :: Int -> Int
square x = undefined
```

The expression `undefined` is a placeholder, and not working code. An attempt to evaluate it will result in an error. Internally it is defined as follows:

```
undefined = error "Prelude.undefined"
```

We will use `undefined` to present you with partial code (in particular because Haskell does not accept a type signature without a matching function declaration). With the function `error` you can define your own exceptions.

Exercise 1:

a) Try using the function `square`. Look up the types of `undefined` and `error`.

b) Complete `square`, replacing `undefined` with an appropriate expression to compute the square x^2 of an input number x.

c) Use `square` to write a function `pythagoras` that, for positive integers a, b, c, determines if they form a Pythagorean triple, $a^2+b^2 = c^2$. First, give a type signature.

```
*Main> square 4
16
*Main> pythagoras 6 8 10
True
*Main> pythagoras 1 2 3
False
```

Guards

You should see the code:

```
factorial :: Int -> Int
factorial n
    | n <= 1     = undefined
    | otherwise = undefined
```

The vertical bars, called **guards**, create a conditional. Operationally, each guard is evaluated in turn, and the first to evaluate to `True` gives the return value for the function. The suggestively named expression `otherwise` is defined as `True`.

Exercise 2:

a) Complete the function `factorial`.

b) The Euclidean algorithm for the greatest common divisor (GCD) of two natural numbers is this: for input x and y, if x and y are equal, that is also their GCD; otherwise, take the GCD of the smaller one of x and y and the difference between x and y. Implement this as the function `euclid`.

c) Try to run the algorithm with one argument negative or zero. Stop the interpreter by pressing `ctrl-c`. Add an extra guard to the function `euclid` so that it gives an error in the case where any of the two inputs is zero or negative.

d) Write a function `power` that computes a^b given a and b. It should throw an exception when b is negative. Do not use the built-in exponentiation function `a^b`. You may either use a straightforward recursion, or the **exponentiation-by-squaring** method (see Wikipedia). In the latter case you will need the predefined functions `even` and `div`, and the function `square` from the previous exercise.

```
*Main> factorial 20
2432902008176640000
*Main> euclid it 298572039485
5
*Main> power 6 7
279936
```

Lists

Lists are the standard data structure in Haskell. Recall that a list of `Int` s is defined inductively as either of:

`[]` ⟵ the empty list

`<head> : <tail>` ⟵ a "cons" of `<head> :: Int`
and `<tail> :: [Int]`

Functions can **build** and **decompose** lists using these constructors. For example, a function that does absolutely nothing is:

```
nothing :: [Int] -> [Int]
nothing [] = []
nothing (x:xs) = x : xs
```

A useful pre-defined list is `[n..m]` which hold all elements counting up from an integer `n` to an integer `m` .

a) Complete the function `range` so that `range n m` behaves as `[n..m]` . That is, it should give the list of `Int` s from `n` to `m` inclusive.

b) Complete the function `times` to compute the product of the elements in a list.

c) Complete the function `fact` to be the factorial function, but this time by combining `range` and `times` .

```
*Main> range 4 9
[4,5,6,7,8,9]
*Main> times it
60480
*Main> fact 10
3628800
```

Because Haskell is **lazy**, which means it only computes what it needs, lists can be infinite. For example, the special syntax `[n..]` gives the list of all integers counting up from `n` . Try it! The benefit of these "lazy" lists is that you don't need to decide in advance how many elements you will need.