Tutorial 1: Recursion

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 consult the **Getting Started** document on Moodle:

Getting Started with Haskell

Load the file `tutorial_1.hs` into `WinGHCi`, and open it in the text editor.

Error & undefined

In your file `tutorial_1.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) Familiarize yourself with the interactive environment. Ask it to compute some simple arithmetic like `3+(4*5)` or `2^2^2^2^2`. Declare a variable `x = 7` and try `6*x`.

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

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

d) 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.

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.

Adventure Game

In the Haskell tutorials you will construct a text adventure game. In this tutorial we will start building the functions that manage your party of adventurers. We will represent each adventurer as a `String`, and your party as a list of strings `[String]`. We will assume that a party contains no duplicate items (adventurers with the same name).

Lists

Lists are defined inductively, and consist of either the **empty list** `[]` or a **cons** `x:xs` of a head and a tail. Your `tutorial_1.hs` provides two lists on which to test your functions.

```
party1 = ["Robert","Cersei","Ned","Jamie"]
party2 = ["Daenerys","Jorah","Tyrion","Grey Worm","Daario","Missandei"]
```

Exercise 3:

a) Complete the function `member xs y` which returns `True` if the string `y` is in the list `xs`, and `False` otherwise. In the given patterns, the underscore (`_`) is a variable that isn't used. Like any variable, it matches any pattern.

b) Write a second version named `member'`, this time using boolean "or" (`||`) to replace the guards (hint: `y` is a member of `x:xs` if it is equal to `x` or a member of `xs`).

c) Complete `removeOne xs y` which removes the string `y` from the list `xs`. If `y` is not in the list, return all of `xs`. (Recall: we assume there are no duplicates in `xs`.)

Test your functions:

```
*Main> member party1 "Tyrion"
False
*Main> member party2 "Tyrion"
True
*Main> removeOne party1 "Robert"
["Cersei","Ned","Jamie"]
*Main> removeOne party1 "Missandei"
["Robert","Cersei","Ned","Jamie"]
```

Exercise 4:

a) Complete the function `members xs ys` which returns `True` if the strings in the list `ys` also belong to the list `xs`. Use your function `member`.

b) Write a second version named `members'` using "and" (`&&`) to replace the guards.

c) Complete the function `removeAll xs ys` that removes every string in `ys` from the list `xs`. Use your function `removeOne`. **Hint:** you should not be using guards.

Test your functions:

```
*Main> members party2 ["Jorah","Daario"]
True
*Main> members party2 ["Tyrion","Cersei"]
False
*Main> removeAll party1 ["Cersei","Jamie","Grey Worm"]
["Robert","Ned"]
```