Functional Programming course
Exercise set 1

due to October 12th

All exercise solutions should be a part of a single module that can be loaded into the Haskell interpreter.

Exercise 1. Similarly as it was done for the function `exOr` (see the Lecture 2 slides), define in Haskell two different versions of a function

`nAnd :: Bool -> Bool -> Bool`,

which returns the result `True` in all cases except the one when both arguments are `True`.

Moreover, define the third version of the function that encodes the truth table for this mathematical function. In other words, the literal values `True` and `False` should be used instead of both function arguments, resulting in four different definition cases (equations).

Exercise 2. Import the module `Test.QuickCheck` by adding the command "`import Test.QuickCheck`" at the beginning of your module (after the keyword **where**). After that, test that all the three versions of `nAnd` from the previous exercise are functionally identical by defining the corresponding properties of the type `::Bool-> Bool -> Bool` and running

`quickCheck prop_nAnd???`

in the interpreter.

Think of one more property that any implementation of `nAnd` must satisfy (for example, that it should return `True` if any of the arguments is `False`) and test it.

Exercise 3. The pre-defined function `length` returns the size of a string (or any sequence). Relying on application of this function, define your own function `nDigits ::Integer->Int` that takes any integer number and returns the number of its digits.

To distinguish the cases of negative and natural numbers, use guards in your function definition.

Exercise 4. Write a function `nRoots::Float−>Float−>Float−>Int` which returns the number of solutions for a quadratic equation

$$a * x^2 + b * x + c = 0.0,$$

for the given real coefficients a, b, and c.

Reminder: the quadratic equation has

- two real roots, if $b^2 > 4.0 * a * c$,
- one real root, if $b^2 = 4.0 * a * c$,
- no real roots, if $b^2 < 4.0 * a * c$,

provided that $a \neq 0.0$

Please distinguish different definition cases using guards. In the case when $a = 0.0$, your function must return an error (using the pre-defined `error` function), e.g., `error "the first argument should be non-zero!"`.

Exercise 5. Using your solution the last exercise, define the functions

`smallerRoot::Float−>Float−>Float−>Float`

and

`largerRoot::Float−>Float−>Float−>Float`,

which respectively return the smaller and larger root of a quadratic equation, for the given real coefficients a, b, and c.

Reminder: the formula for the roots of a quadratic equation is

$$\frac{(-b) \pm \sqrt{(b^2 - 4 * a * c)}}{2 * a}$$

In the case of one root, both functions should return the same result. In the case of no roots, the corresponding error message has to be returned.

Exercise 6. Using the primitive recursion mechanism, write a function `power2::Integer−>Integer` that calculates the power of 2 for the given natural number n, i.e., 2^n.

When a negative number is supplied as the parameter value, the function must return 0.

Exercise 7. Write a function `mult::Integer->Integer->Integer` that recursively redefines the multiplication operation by using only addition, i.e.,

$$m * n = m + m + ... + m$$

n times, or

$$m * n = n + n + ... + n$$

m times.

Use either the first or second function argument for implementing primitive recursion, i.e. defining the base and recursive cases. Please make sure that the function cases are exhaustive, i.e., it is defined also for the cases when any argument is negative.

Exercise 8. Define a recursive function

`prod::Integer->Integer->Integer`

that, for the given numbers m and n, multiplies all the numbers from the range $m..n$. In other words, the produced result must be equal to

$$m * (m + 1) * ... * (n - 1) * n$$

Note that, to follow the primitive recursion pattern, it is not necessary to choose a single parameter in order to define the base and recursive cases. Any expression on the parameter values can be used instead, provided that, according to the pattern, this expression is checked to be equal to 0 for the base case, and be greater than 0 for the recursive case. Moreover, the recursive call should make this expression smaller by 1.

An error message must be returned for an invalid range, i.e., when $m > n$.

Finally, redefine the factorial function (from the Lecture 2 slides) as a special case of `prod`.