DEPARTMENT EEMCS
Date: June 23, 2016

Test: Programming Paradigms — Functional Programming

June 27, 2016
13:45 – 16:45

Remarks:
- During this test you may use the syllabus: *Functional Programming – an overview*, nothing else.
- You may use predefined Haskell functions and operators from the packages *Prelude*, *Data.List*, *Data.Char*, *Data.Maybe*.
- **Mention the type for *every* function that you define.**
- Judgement: there are three exercises of equal weight.
- Style and elegancy also play a role in judgement, e.g., do not use unnecessary helper functions.
- Good luck!

Exercise 1.

a. The first five lines of *Pascal's Triangle* are as follows:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

Every number in a line s calculated by adding the two adjacent numbers immediately above that number. In addition, every line begins and ends with a 1. Thus, the line 6 is calculated from line 5 by:

```
1  1+4  4+6  6+4  4+1  1
```

The result thus is:

```
1  5  10  10  5  1
```

Write a function which calculates the first n lines of Pascal's Triangle, in the form of a list of lists.

b. A *polynome* (of degree n) is an expression of the form

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Answer the following questions:

1. Define a datatype *Polynome* to indicate a polynome.
2. Write a function f such that

   $$f\ p\ x$$

   calculates the value of polynome p (i.e., p should be an element of type *Polynome* defined above) for argument x.
3. Write a function *derivative* which calculates the derivative of a polynome. Remember that the derivative of a term ax^n is

   $$n\, a\, x^{n-1}$$

c. Answer the following questions:

1. Define a function *prime* that checks whether an integer number p is a prime number or not.
2. Write a function which calculates the list of 2-tuples (p, q) such that p and q are both prime numbers, and $p + q = 100$.

Exercise 2.

a. Define a type *Tree* of trees which contain at every node and at every leaf a number of type *Int*, and which contains either *one* or *two* subtress at every node.

b. Write a function *replaceByMax* which replaces each number in a tree by the maximum number in that tree.

c. Write a function *total* which calculates the total of all numbers in a tree.

d. Write a function *depth* which calculates the depth of a tree of type *Tree* (the *depth* of a tree is the length of the longest path from the root to any leaf).

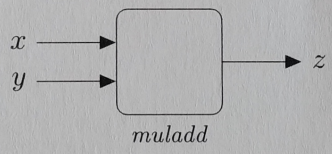
e. Write a function *replaceByPair* which replaces the number at each node in a tree of type *Tree* by a pair of numbers which consists of the *total* of all numbers in the subtree at that node, and the depth of that subtree.

Also, mention the type of the resulting tree.

Exercise 3.

MulAdd Machine

The *MulAdd* machine can only do *two* operations on numbers: *add* $(+)$ and *multiply* $(*)$, but only *one at a time*. It gets two streams of inputs, and every incoming 2-tuple (x, y) has to be multiplied, and added to an internal value z that initially starts at 0. The value of z is also produced as output.



For example, if the input stream of pairs (x, y) is as follows:

$$[(1,2), (3,4), (5,6), (7,8), \ldots]$$

then the produced output stream of values for z is

$$[0, 2, 14, 44, 100, \ldots]$$

a. Define recursively the function *muladd* that "consumes" the input stream, multiplies x and y, and updates z by adding the result of this multiplication to z, Finally, *muladd* should produce the the list of resulting z-values.
Remember: as mentioned above the function *muladd* can *not* execute a multiplication and an ddition at the same time, so it has to remember the result of the multiplication before it can add it to the value of z. That also means that it should execute a multiplication and an addition in alternating order, and only when a multiplication is executed it consumes the next input pair.

b. Extend the definition of *muladd* with the possibility to "reset" the value of z to 0 (*before* the new product of x and y is added) if z becomes larger than a given value m.

For example, if $m = 10$, then in the example above z is set back to 0 several times, and the output (on the same input as above) then becomes:

$[0, 2, 14, 30, 56, \ldots]$