DEPARTMENT EEMCS
Date: June 2, 2016

Test: Programming Paradigms — Functional Programming

June 10, 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!

Opgave 1.

a. A number n is *perfect* if the *total* of all dividers of n (including 1, but excluding n) equals n itself. For example, 6 and 28 are perfect numbers, since $1+2+3 = 6$ and $1+2+4+7+14 = 28$.

Write a function *perfect* which yields the list of all perfect numbers smaller than a given number m.

b. A list xs of length n is called a *jolly jumper* if the absolute values of the differences between all pairs of consecutive numbers are precisely all numbers in the range $1, \ldots, n-1$. For example, $[1, 4, 2, 3]$ is a jolly jumper, since the list of absolute diffences of consecutive elements in the given list is $[3, 2, 1]$.

Write two variants of a function which tests whether a given list is a jolly jumper: one with recursion, one with higher order functions.

c. A matrix is a list of lists of numbers, where the "inner lists" are the rows of the matrix. All rows are equally long.

Define two functions *addRows* and *addColumns* that yield the totals of all rows and columns (respectively) of a matrix. Both functions have to be defined in three ways: with recursion, with higher order functions, and with list comprehension.

d. Define the function *map* by using *foldl*.

Opgave 2.

a. Define a type for trees in which a node may have an arbitrary number of subtrees, and a node contains a value of a type that is the same for every node. A *Leaf* then is a node with zero subtrees.

Define special cases of this type for trees with a number (*Int*) at its nodes, and for trees with a tuple of a character and a boolean at each node. You have to use the above defined type for these specialisations.

b. Write a function which yields the maximum of all numbers in a tree (for the special case where the values at the nodes are numbers).

c. Write a function *mapTree* which applies a function f to all values in a tree.

d. A *path* in a tree is a list of numbers that indicate which subtree to choose at each node. For example, the path $[2, 1, 4]$ starts at the root of a tree, and ends a the node that is found as follows: take subtree with index 2 at the root of the tree, then the subtree with index 1, and then the subtree with index 4.

Write a function that yields the path from the root to a specific value in the tree. You may assume that all values in the tree are different.

Opgave 3.

Pocket calculator

In this exercise you'll have to define an evaluation function *eval* for a pocket calculator, which has the operations $+, \times, -, /$ as operations, and of course the equal sign $(=)$ for requesting the end result. The inputs for the calculator (i.e., the keys that are touched on a real calculator) have to be given in the form of a list of the separate inputs. For example,

$$input \;=\; [2, +, 3, \times, 4, =]$$

is a possible list of inputs (clearly, this example is not well typed in Haskell). We will assume that the list of inputs starts with a number, and an operation symbol is always immediately preceded by a number (i.e., there is no repetition of an operation using previous inputs as with most real pocket calculators). You may also assume that "=" only occurs as the last element of the input list.

a. Define an algebraic datatype ("embedded language") for inputs. Give also the type of the above example input list *input*.

b. *Preliminary remark*: here you may assume that the calculator has a key for every number, so each number only takes one position in the list of inputs.

A "naive" pocket calculator will calculate the result of everything typed in so far whenever one of the keys $(+, \times, -, /, =)$ is hit. Thus, in the above example, "$2 + 3$" is calculated at the moment that "$\times$" is hit, and "5×4" is calculated when "$=$" comes in. The *output* of the pocket calculator is the list of intermediate results produced at the moments when these operation symbols come in. So, in the example above, the output is given by the list:

$$[2, 5, 20].$$

Note that a naive calculator calculates the example above as: $(2 + 3) \times 4$.

Define a function *eval* which calculates the output list given an input list.

Hints:

- Your function needs to remember *three* things (*two* numbers and *one* operation) in order to be able to calculate the next result: the previous result, the number that was typed in as the last one, and the operation that has to be exeuted. A practical way to do so is to give your function a 3-tuple as extra argument, that is updated at each input.

- The initial situation has the (nasty) problem that the previous situations are undefined. One possible way to solve this is to write special variant(s) of the function *eval* that take(s) care of the first two inputs, and then continue with the function *eval* itself.

c. Extend the type and function defined above by the possibility for functions of *one* argument (for example, e^x, $\sqrt{x}$, $\sin x$) which have to be executed immediately at the moment when they come in (hence, you don't have to remember these functions). A one place function may be preceded by a number, or by a function or operation. Thus, there are two cases for the value to which a one-place function f should be applied:

- if f is typed in immediately after a number x, it has to be applied to that number x,

- otherwise it has to be applied to the intermediate result that was calculated last.