DEPARTMENT EEMCS
Date: June 9, 2017

Test: Programming Paradigms — Functional Programming

June 16, 2017
13:45 – 16:45

Remarks:
- During this test you may use the syllabus: *A Short Introduction to Functional Programming* as published on Blackboard, 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. 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*.

Exercise 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.

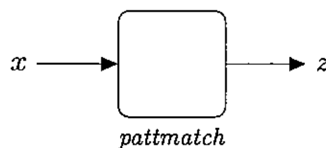
d. A *path* in a tree is a list of numbers that indicate which subtree to choose at each node. For example, the path $[2, 1, 4]$ starts at the root of a tree, and ends a the node that is found as follows: take subtree with index 2 at the root of the tree, then the subtree with index 1, and then the subtree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

Exercise 3.

The Pattern Match Machine

The *Pattern Match Machine* contains a sequence of elements ps (the *pattern*) which have to be matched against an incoming stream of values x.



For example, if the pattern is as follows:

    [1, 3, 1]

then the stream of input values

    [2, 5, 1, 3, 1, 3, 1, 0, 2, 1, 3, 2]

contains this pattern *two* times.

The Pattern Match Machine is realized by a recursive function *pattmatch*, that is, this function consumes *one* input value at a time, and checks whether the pattern occurs in the input stream. The function should have as its *first*

argument a given pattern, and it should yield *True* if the current input completes the pattern, and *False* otherwise. Thus, in the above example the output should be (T, F are shorthand for *True*, *False*):

$$[F, F, F, F, T, F, T, F, F, F, F, F]$$

Note that if the length of the pattern is n, then your function has to "remember" $n-1$ previous input values.

a. Give the type for the function *pattmatch*. You may assume that the elemnts of the pattern and the input stream are *Int*s.

b. Write the function *pattmatch*.

c. Write a variant of this function *pattmatch* that counts how many times the pattern passes by. The result of your function should now be the number of matches so far. Thus, in the above example the output should be:

$$[0, 0, 0, 0, 1, 1, 2, 2, 2, 2, 2, 2]$$