FP 2009-2010, Eindtoets
2010, Feb 3 , 14.00-17.00

Hand in the separate sheet with the 6 solutions. Don't forget to fill out your name! If you are sure your solution does not even come near to what is expected leave the corresponding entry blank; this will considerably speed up the marking process.

1. **Flattening search trees (1)**
   Consider the type of binary search trees:
   ```
   data Tree a = Leaf
               | Node (Tree a) a (Tree a)
   ```
   and assume that for each node it holds that the values in the left subtree are strictly smaller than the value of type *a* in that node, and that the values in the right subtree are strictly larger than that value.

   Write a function *rft*:: *Tree a* → [*a*] (*rft* stands for *reverse flatten tree*), which returns a descending list containing the *a*-values from the nodes of the tree. Do not use the function *reverse*!

2. **Induction Proof (1)**
   Prove by induction that *map f* (*x* ++*y*) = *map f x* ++*map f y*.

3. **Permutations (2)**
   Write a functions *perms* :: [*a*] → [[*a*]] which returns a list containing all permutations of the input list.

4. **Substitution and Evaluation (2)**
   The following data type was used to represent terms in the Prolog interpreter:
   ```
   type Ident = String
   data Term = Con Int
             | Var Ident
             | Fun String [Term]
             deriving Eq
   ```
   The function *subst* :: [(*Ident*, *Term*)] → *Term* → *Maybe Term* tries to replace all variables in a *Term* by the *Term* associated with that variable in the first parameter. In case the *Term* contains variables which cannot be found in the [(*Ident*, *Term*)] the whole subsititution returns *Nothing*, and otherwise *Just t* in which *t* stands for the result of the substitution.

   (a) Define the function *subst*.

   (b) Write a function *evalTerm* :: *Term* → *Int* which evaluates a *Term*, assuming that the *Term* does not contain free variables anymore, and that the function symbols which occur are either "+" or "-". You may assume these operators have indeed the right number of arguments in the list.

5. **Classes (2)**
   The operators for parser combinators we have seen are introduced in a Haskell module *Control.Applicative* by the classes:
   ```
   class Applicative f where
     (<*>) :: f (b -> a) -> f b -> f a
     pure  :: a               -> f a   -- like pSucceed
   class Alternative f where
     (<|>) :: f a -> f a -> f a
     fail  ::               f a
   ```
   Give corresponding instances of these classes for the type constructor *Maybe*. Keep in mind that a *Maybe* result resembles the list of successes method, with the difference that we return at most one succesful result.

(see other side for exercise 6)

6. **Maximal Segment Sum (2)**

(a) Complete the following definition, for the function which computes all consecutive sequences of elements from a list:

$segs\ xs \quad = fst\ .\ segsinits\ \$\ xs$
$segsinits ::\ [a] \rightarrow ([[a]], [[a]])$
$segsinits\ [\,] \quad = ([[\,]], [[\,]])$
$segsinits\ (x : xs) = \ldots$

Example of the use of segs:

$\gg segs\ [1, 2, -3, 4, 1]$
$[[\,], [1], [4], [4, 1], [-3], [-3, 4], [-3, 4, 1], [2], [2, -3], [2, -3, 4], [2, -3, 4, 1],$
$[1], [1, 2], [1, 2, -3], [1, 2, -3, 4], [1, 2, -3, 4, 1]]$

(b) The specification of the *maximal segment sum problem* is:

$mss\ xs = foldr\ max\ 0\ .\ map\ sum\ .\ segs\ \$\ xs$

Note that in the evaluation of this specification a lot of common computation is going on. Give a solution for *mss* which does not construct the intermediate list structures, and which takes time linear in the length of the list.

The following hints may be useful:

- First modify your *segsinits* definition of the previous item into a definition of:

  type $Sum\ a = a$
  $segsinitis' :: Num\ a \Rightarrow [a] \rightarrow ([(Sum\ a, [a])],$
  $\quad , [(Sum\ a, [a])]$
  $\quad)$

  which tuples each list occurring in the resulting lists with its sum.
- Apply the same method once more for the lists of segments and inits, and thus tuple those lists with the maximum value of the individual sums:

  type $Max\ a = a$
  $segsinitis'' :: Num\ a \Rightarrow [a] \rightarrow ((Max\ a, [(Sum\ a, [a])])$
  $\quad , (Max\ a, [(Sum\ a, [a])])$
  $\quad)$
- Now remove the superfluous computations from the program.
- Keep in mind that $sum\ (x : xs) \equiv x + sum\ xs$
- Realise that $(a + b)\ `max`\ (a + c) = a + (b\ `max`\ c)$

You do not have to give the intermediate results. The hints are just there to help you to derive the solution.

Name: **Student nr:** **Program: Inf/CKI/...**

1	2	3	4	5	6

Please leave the above boxes blank.

QUESTION 1: The function *rft*:

QUESTION 2: The proof that $map\ f\ (xs +\!\!+ ys) \equiv map\ f\ xs +\!\!+ map\ f\ ys$ (place the empty and the non-empty case next to each other):

QUESTION 3: The function *perms* :

QUESTION 4a: The function *subst*:

QUESTION 4b: The function *evalTerm*:

QUESTION 5:

instance *Applicative Maybe* **where**

instance *Alternative Maybe* **where**

QUESTION 6a: The function *segsinitis*:

QUESTION 6b: The function *mss*: