Written Examination, December 19th, 2016 Course no. 02157

The duration of the examination is 4 hours.

Course Name: Functional programming

Allowed aids: All written material

The problem set consists of 5 problems which are weighted approximately as follows:
Problem 1: 20%, Problem 2: 15%, Problem 3: 10%, Problem 4: 25%, Problem 5: 30%

Marking: 7 step scale.

Do not use imperative features, like assignments, arrays and so on, in your solutions.

You are allowed to use the .NET library including the modules described in the book, e.g., List, Set, Map, Seq etc.

You are allowed to use functions from the textbook. If you do so, then provide a reference to the place where it appears in the book.

In this set there is a minor revision (by Michael R. Hansen, 30-11-2017) involving the type of the function `tryFindPathTo` in Problem 5.

# Problem 1 (20%)

We consider here *scoreboards* containing information about the *scores*, where a score describes the *points* a *named person* has obtained in an *event*. This is modelled by the following type declarations:

```
type Name = string
type Event = string
type Point = int
type Score = Name * Event * Point

type Scoreboard = Score list

let sb = [("Joe", "June Fishing", 35); ("Peter", "May Fishing", 30);
          ("Joe", "May Fishing", 28); ("Paul", "June Fishing", 28)];;
```

The example scoreboard `sb` describes four scores, where Joe, for example, has obtained 35 points in the fishing event in June and 28 points in the May fishing event. The other two scores have similar explanations.

1. The points occurring in scores must be non-negative integers, and scores must occur in scoreboards in a sequence respecting weakly decreasing points, that is, if $(n, e, p)$ occurs before $(n1, e1, p1)$ in a scoreboard, then $p \geq p1$. Hence, a score with the highest number of points occurs first and a score with the lowest number of points occurs last in a scoreboard.
   Declare a function: `inv: Scoreboard -> bool`, that checks whether a scoreboard satisfies this constraint.

The functions below must respect the invariant `inv`, that is, it can be assumed that argument scoreboards satisfy `inv`, and it is required that scoreboard results of functions must satisfy `inv`.

2. Declare a function `insert: Score -> Scoreboard -> Scoreboard`, so that `insert` $s$ $sb$ gives the scoreboard obtained from $sb$ by insertion of $s$. The result must satisfy `inv`.
3. Declare a function `get: Name*Scoreboard -> (Event*Point) list`, where the value of `get`$(n, sb)$ is a list of pairs of events and points obtained from $n$'s scores in $sb$. For example `get("Joe",sb)` must be a list with the two elements: `("June Fishing", 35)` and `("May Fishing", 28)`.
4. Declare a function `top: int -> Scoreboard -> Scoreboard option`. The value of `top` $k$ $sb$ is `None` if $k < 0$ or $sb$ does not contain $k$ scores; otherwise the value is `Some` $sb'$, where $sb'$ contains the first $k$ scores of $sb$.

 ... Continued on next page

# Problem 2 (15%)

1. Declare a function `replace` $a\,b\,xs$ that gives the list obtained from $xs$ by replacing every occurrence of $a$ by $b$. For example, `replace 2 7 [1; 2; 3; 2; 4]` $=$ `[1; 7; 3; 7; 4]`.
2. Give the (most general) type of `replace`.
3. Is your `replace` function tail recursive? Give the brief informal explanation of your answer. If it is not tail recursive, then provide a tail-recursive variant that is based on an accumulating parameter.

# Problem 3 (10%)

Consider the following F# declarations:

```
let pos = Seq.initInfinite (fun i -> i+1) ;;
let seq1 = seq { yield (0,0)
                 for i in pos do
                    yield (i,i)
                    yield (-i,-i) }

let val1 = Seq.take 5 seq1;;

let nat = Seq.initInfinite id;;
let seq2 = seq { for i in nat do
                    yield (i,0)
                    for j in [1 .. i] do
                       yield (i,j) }

let val2 = Seq.toList(Seq.take 10 seq2);;
```

1. Give the types of the sequences `pos`, `seq1` and `val1` and describe their values.
2. Give the type of `seq2` and describe the sequence. Furthermore, give the value of `val2`.

 ... Continued on next page

# Problem 4 (25%)

Consider the following F# declaration of a type for binary trees:

```
type Tree<'a,'b> = | A of 'a | B of 'b
                   | Node of Tree<'a,'b> * Tree<'a,'b>;;
```

where a value `A` $a$ is called an *A leaf* and a value `B` $b$ is called a *B-leaf*.

1. Give three values of type `Tree<bool,int list>` using the constructors `A`, `B` and `Node`.
2. Declare a function that counts the number of occurrences of A-leaves in a tree.
3. Declare a function

   ```
   subst:'a -> 'a -> 'b -> 'b -> Tree<'a,'b> -> Tree<'a,'b>
         when 'a : equality and 'b : equality
   ```

   where `subst` $a\ a'\ b\ b'\ t$ is the binary tree obtained from $t$ by substituting every occurrence of the `A` $a$ by `A` $a'$ and by substituting every occurrence of the `B` $b$ by `B` $b'$.

Consider the following F# declarations of two functions `f` and `g`:

```
let rec g = function
            | Node(t1,t2) -> Node(g t2, g t1)
            | leaf        -> leaf;;

let rec f = function
            | A a         -> ([a],[])
            | B b         -> ([], [b])
            | Node(t1,t2) -> let (xs1,ys1) = f t1
                             let (xs2,ys2) = f t2
                             (xs1@xs2, ys1@ys2);;
```

4. Give the (most general) types for `f` and `g`, and describe what each of these two functions computes. Your description for each function should focus on *what* it computes, rather than on individual computation steps.
5. Make a continuation-based tail-recursive variant of `f`.

# Problem 5 (30%)

We consider now trees where nodes can have an arbitrary number of subtrees:

```
type T<'a> = N of 'a * T<'a> list;;

let td = N("g", []);;
let tc = N("c", [N("d",[]); N("e",[td])]);;
let tb = N("b", [N("c",[])]);;
let ta = N("a", [tb; tc; N("f",[])])
```

The tree $t = \mathtt{N}(v, [t_0; \ldots; t_{n-1}])$ describes a node that contains the *value* $v$ and has $n$ (immediate) subtrees $t_i$, for $0 \leq i < n$. For example, the four trees `ta` - `td` illustrate trees having 3, 1, 2 and 0 immediate subtrees, where the values contained in the nodes are the seven strings `"a"` - `"g"`.

1. Declare a function `toList` $t$ which returns a list of all the values occurring in the nodes of the tree $t$. The order in which values occur in the list is of no significance.
2. Declare a function `map` $f$ $t$, which returns the tree obtained from the $t$ by applying the function $f$ to the values occurring in the nodes of $t$. Give the type of `map`.

We shall use integer lists to denote paths in trees:

```
type Path = int list;;
```

A *path* (type `Path`) in the tree $t = \mathtt{N}(v, [t_0; \ldots; t_i; \ldots; t_{n-1}])$ is a list of integers that *identifies* a subtree of $t$ in the following recursive manner:

- The empty path (list) [ ] identifies the entire tree $t$.
- If $is$ identifies $t'$ in $t_i$, then $i :: is$ identifies $t'$ in $\mathtt{N}(v, [t_0; \ldots; t_i; \ldots; t_{n-1}])$.

For example, the path `[0]` identifies the subtree `tb` of `ta` and the path `[1; 1; 0]` identifies the subtree `td` of `ta`.

3. Declare a function `isPath` $is$ $t$ that checks whether $is$ is a path in $t$.
4. Declare a function `get: Path` $\rightarrow$ `T<'a>` $\rightarrow$ `T<'a>`. The value of `get` $is$ $t$ is the subtree identified by $is$ in $t$.
5. Declare a function `tryFindPathto : 'a` $\rightarrow$ `T<'a>` $\rightarrow$ `Path option`. When $v$ occurs in some node of $t$, then the value of `tryFindPathto` $v$ $t$ is `Some` *path*, where $v$ occurs in the node of $t$ identified by *path*. The value of `tryFindPathto` $v$ $t$ is `None` when $v$ does not occur in a node of $t$. There is no restriction concerning which path the function should return when $v$ occurs more than once in $t$.