Written Examination, May 25th, 2018 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 4 problems which are weighted approximately as follows:
Problem 1: 20%, Problem 2: 30%, Problem 3: 20%, Problem 4: 30%

Marking: 7 step scale.

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

Unless you are explicitly instructed not to do so, you are allowed to use the .NET library including the modules described in the textbook, 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.

You may also use functions introduced in a problem from this exam set in your solution to another problem (even without having solved the original problem).

Problem 1 (20%)

Consider the following F# declaration:

```
let rec f xs ys = match (xs,ys) with
                  | (x::xs1, y::ys1) -> x::y::f xs1 ys1
                  | _                -> [];;
```

1. Give an evaluation (using $\rightsquigarrow$) for `f [1;6;0;8] [0; 7; 3; 3]` thereby determining the value of this expression.
2. Give the (most general) type for `f`, and describe what `f` computes. Your description should focus on *what* it computes, rather than on individual computation steps.
3. The declaration of `f` is *not* tail recursive. Give a brief explanation of why this is the case and provide a declaration of a tail-recursive variant of `f` that is based on an accumulating parameter. Your tail-recursive declaration must be based on an explicit recursion.
4. Provide a declaration of a continuation-based, tail-recursive variant of `f`.

... Continued on next page

Problem 2 (30%)

A list of elements `[`$e_0; e_1; e_2; \ldots; e_{n-1}$`]`, where $n \geq 0$, is *ordered* if the elements satisfy:

$$e_0 < e_1 < e_2 < \cdots < e_{n-2} < e_{n-1}$$

or if it is the empty list. Three ordered integer lists are $[\,], [5]$, and $[3; 12; 14]$.

We use the following type for ordered lists:

```
type Ordered<'a when 'a:comparison > = 'a list
```

For the below mentioned functions: `insert`, `delete` and `union`, you can assume that their list arguments are ordered lists and you must declare each function so that its result is an ordered list as well.

1. Declare a value `empty` for the empty ordered list and a function `singleton` x for the ordered list having x as its only element.
2. You should solve the following questions $A.$, $B.$ and $C.$ without using library functions like those from the `List` library.

   $A.$ Declare a function `insert` x xs, where xs is an ordered list. The value of the function is the ordered list obtained from xs by insertion of the element x. For example, `insert` $13\ [3; 12; 14] = [3; 12; 13; 14]$ and `insert` $12\ [3; 12; 14] = [3; 12; 14]$.

   $B.$ Declare a function `delete` x xs, where xs is an ordered list. The value of the function is the ordered list obtained from xs by deletion of the element x. For example, `delete` $13\ [3; 12; 13; 14] = [3; 12; 14]$ and `delete` $13\ [3; 12; 14] = [3; 12; 14]$.

   $C.$ Declare a function `union` xs ys, where xs and ys are ordered lists. The value of the function is the ordered list that contains the elements of xs and ys. For example, `union` $[3; 12; 14; 15]\ [1; 5; 12; 13; 14] = [1; 3; 5; 12; 13; 14; 15]$.

   $D.$ Give the most general types of `empty`, `singleton`, `insert`, `delete` and `union`.
3. Give another declaration for `union` using either
   - `List.fold: (('a -> 'b -> 'a) -> 'a -> 'b list -> 'a)` or
   - `List.foldBack: (('a -> 'b -> 'b) -> 'a list -> 'b -> 'b)`
4. Give an example of a list that cannot be an ordered list due to the type constraint `'a when 'a:comparison` occurring in the declation of the type `Ordered`.

... Continued on next page

Problem 3 (20%)

Consider the following F# declarations:

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

let rec g(C(_,x,ts)) = x + h ts
and h = function
        | []    -> 0
        | t::ts -> g t + h ts;;

let rec i p (C(a,x,ts)) = if p a then C(a,x,[])
                          else C(a,x,List.map (i p) ts);;

let rec j m (C(_,x,ts)) = C(m, x, k (m+1) ts)
and k n = function
          | [] -> []
          | t::ts -> j n t :: k n ts;;
let q t = j 0 t;;
```

1. Give four values of type `T<bool>`.
2. Give the (most general) types for `g`, `i` and `q` and describe what `g`, `i` and `q` computes. Your description for each function should focus on *what* it computes, rather than on individual computation steps.

Problem 4 (30%)

We now consider *expression trees* for simple arithmetic expressions constructed from *identifiers* and *integer constants* using *addition* (or sum) and *let expressions*. A let-expression `Let`(x, e_1, e_2) reads: "let x be defined by e_1 in e_2" as we have seen in the course. We capture such expressions in F# by the following type declaration:

```
type ExprTree = Const of int
              | Ident of string
              | Sum of ExprTree * ExprTree
              | Let of string * ExprTree * ExprTree
```

where `Const` is the constructor for integer constants, `Ident` is the constructor for identifiers, `Sum` is the constructor for addition and `Let` is the constructor for let-expressions.

1. Declare four values of type `ExprTree` using all four constructors.
2. Declare a function that gives an integer list with the constants occurring in an expression tree.

... Continued on next page

The *free identifiers* of an expression tree are (intuitively) the identifiers for which you need values in order to (be able to) compute an integer value for the expression tree. For example:

- `Sum(Const 2, Const 3)` has no free identifier – you can compute the value 5 for this expression tree.
- `"x"` and `"z"` are the free identifiers of `Sum(Ident "x", Ident "z")` as you need values for `"x"` and `"z"` in order to compute an integer value for this expression tree.
- `Let("x", Const 2, Sum(Ident "x", Ident "x")` has no free identifier – its value 4 can be computed using the given definition for `"x"`.
- The expression tree `Let("x", Sum(Const 2,Ident "x"), Sum(Ident "x",Const 3))` has `"x"` as free identifier. A value for the occurrence of `"x"` in `Sum(Const 2,Ident "x")` is needed in order to compute an integer value for this let-expression.

The *free identifiers* of an expression tree can be defined recursively as follows:

- `Const` n has no free identifier.
- `Ident` x has the string x as it only free identifier.
- The free identifiers of `Sum`(e_1, e_2) consists of the free identifiers of e_1 and the free identifiers of e_2.
- The free identifiers of `Let`(x, e_1, e_2) consists of the free identifiers of e_1 and the identifiers obtained by deleting x from the free identifiers of e_2.

3. Declare a function extracting a list of the free identifiers occurring in an expression tree.
4. Declare a function `subst` x n t, where x is a string, n is an integer, and t is an expression tree. The value of the function is the expression tree obtained from t by replacing every free occurrence of the identifier given by x with the constant given by n.
5. Give the type for the function `subst` from Question 4.
6. Show how to extend the type `ExprTree` with a new constructor `App` so that `App("f", [])`, `App("g", [Const 2])` and `App("h", [Const 2; Ident "x"; Const 3])` will be values having type `ExprTree`. Extend your solution to Question 2 in this problem to cope with the new constructor.