Written Examination, May 24th, 2017 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: 10%, Problem 2: 15%, Problem 3: 25%, Problem 4: 20%, 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 they appear in the book.

Problem 1 (10%)

1. Declare a function `repeatList: 'a list -> int -> 'a list`, so that

   $$\texttt{repeatList}\, xs\, n = xs\,@\,xs\,@\cdots@\,xs, \qquad \text{with } n \text{ occurrences of } xs$$

   For example, `repeatList` $[1;2]\ 3 = [1;2;1;2;1;2]$ and `repeatList` $[1;2]\ 0 = [\,]$.

2. Declare a function `merge: 'a list * 'a list -> 'a list`, so that

   $$\texttt{merge}([x_0;x_1;\ldots;x_m],[y_0;y_1;\ldots;y_n]) = \begin{cases} [x_0;y_0;x_1;y_1\ldots;x_m;y_m;y_{m+1};y_{m+2};\ldots;y_n] & \text{when } m<n \\ [x_0;y_0;x_1;y_1\ldots;x_m;y_m] & \text{when } m=n \\ [x_0;y_0;x_1;y_1\ldots;x_n;y_n;x_{n+1};x_{n+2};\ldots;x_m] & \text{when } m>n \end{cases}$$

   That is, the function `merge` can merge the elements of two lists, where the lists need not have the same size. For example, `merge`$([1;2],[3;4]) = [1;3;2;4]$, `merge`$([1;2],[3;4;5]) = [1;3;2;4;5]$ and `merge`$([1;2;3;4],[5;6]) = [1;5;2;6;3;4]$.

Problem 2 (15%)

Consider the following F# declarations:

```
let rec f  = function
           | 0          -> [0]
           | i when i>0 -> i::g(i-1)
           | _          -> failwith "Negative argument"
and g = function
        | 0 -> []
        | n -> f(n-1);;

let h s k  = seq { for a in s do
                     yield k a };;

let rec sum xs = match xs with
                 | []      -> 0
                 | x::rest -> x + sum rest;;
```

1. Give the values of `f 5` and `h (seq [1;2;3;4]) (fun i -> i+10)`. Furthermore, give the (most general) types for `f` and `h`, 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.

2. The function `sum` is *not* tail recursive.

   - Provide a tail-recursive variant that is based on an accumulating parameter, and
   - provide a continuation-based tail-recursive variant of `sum`.

Problem 3 (25%)

We consider here *articles* in a *stock* as modelled by the following type declarations:

```
type Article = string
type Amount  = int
type Price   = int
type Desc    = Amount*Price
type Stock   = (Article*Desc) list

let st = [("a1",(100,10)); ("a2",(50,20)); ("a3",(25,40))];;
```

A description (type `Desc`) of an article is a pair (n, p) where p is the *price* and n is the *amount* of articles that is available in the stock. A stock is a list of pairs (a, d), where a is an article and d is a description of a. The example stock `st` above describes three articles, where, for example, the price of `"a2"` is 20 and 50 pieces of `"a2"` are available in `st`.

1. The value of an article with a description (n, p) is its price times the available amount of the article, that is $n * p$, and the value of a stock is the sum of the values of all its articles. The value of `st` is 3000. Declare a function that computes the value of a stock.

2. The prices and amounts occurring in descriptions of articles must be positive integers. Furthermore, the articles occurring in a stock $[(a_0, d_0); (a_1, d_1); \ldots; (a_n, d_n)]$ must all be distinct, that is, $a_i = a_j$ implies $i = j$, for $0 \leq i \leq n$ and $0 \leq j \leq n$. Declare a function: `inv: Stock -> bool`, that checks whether a stock satisfies these constraints.

Consider now the following declarations, where an *order* of k pieces of article a is given by a pair (a, k):

```
type Order       = Article*Amount
type Status<'a>  = Result of 'a | Error of string
```

3. Declare a function `get` : `Order` → `Stock` → `Status<Price*Stock>` to get a order from a stock. The value of `get` (a, k) st is `Result`(p, st') when there is a sufficient amount of article a available in st, p is the price of k pieces of a, and st' is the stock obtained from st by removal of k pieces of a. In case an insufficient amount of article a is available in st, the value of `get` (a, k) st has the form `Error` *str*, where *str* is a suitable error message. For example: `get ("a2",10) st` = `Result(200,`st'`)`, where the new stock st' is `[("a1",(100,10));("a2",(40,20));("a3",(25,40))]`, and `get ("a2",60) st` = `Error "Insufficient supply for a2"`.

4. Declare a function `getAll` : `Order list` → `Stock` → `Status<Price * Stock>`, where `getAll` *os* *st* gets all orders in *os* from *st*. If there is a sufficient amount of articles available for the orders, then the value is `Result`(p, st'), where p is the total price for all orders and st' is the resulting stock. Otherwise, the value has the form `Error` *str*, where *str* is an error message.

Problem 4 (20%)

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

```
type T<'a> = | L
             | A of 'a * T<'a>
             | B of 'a * T<'a> * T<'a>
             | C of 'a * T<'a> * T<'a> * T<'a>;;
```

1. Give a value of type `T<int>` using all four constructors `L`, `A`, `B` and `C`. Furthermore, give a brief informal description of the values of type `T<'a>`.

Consider now the following declarations:

```
let rec f1 t = match t with
               | B(_, t1,t2) -> f1 t1 && f1 t2
               | L           -> true
               | _           -> false;;

let rec f2 t = match t with
               | L             -> L
               | A(i,t)        -> A(i, f2 t)
               | B(i,t1,t2)    -> B(i, f2 t2, f2 t1)
               | C(i,t1,t2,t3) -> C(i, f2 t3, f2 t2, f2 t1);;

let rec f3 h = function
               | L             -> L
               | A(i,t)        -> A(h i, f3 h t)
               | B(i,t1,t2)    -> B(h i, f3 h t1, f3 h t2)
               | C(i,t1,t2,t3) -> C(h i, f3 h t1, f3 h t2, f3 h t3);;
```

2. Give the (most general) types for `f1`, `f2` and `f3`, and describe what each of these three functions computes. Your description for each function should focus on *what* it computes, rather than on individual computation steps.

 ... Continued on next page

Problem 5 (30%)

In this example we shall consider structured text documents, having paragraphs and (sub)sections, as modelled by the type declarations:

```
type Title    = string
type Document = Title * Element list
and  Element  = Par of string | Sec of Document;;
```

A *document* is a pair (t, es) consisting of a title t and a list of elements es, where an element can be a paragraph (constructor `Par`) characterized by a string or a (sub)section (constructor `Sec`) characterized by a document. An example is:

```
let s1   = ("Background", [Par "Bla"])
let s21  = ("Expressions", [Sec("Arithmetical Expressions", [Par "Bla"]);
                            Sec("Boolean Expressions", [Par "Bla"])])
let s222 = ("Switch statements", [Par "Bla"])
let s223 = ("Repeat statements", [Par "Bla"])
let s22  = ("Statements",[Sec("Basics", [Par "Bla"]) ; Sec s222; Sec s223])
let s23  = ("Programs", [Par "Bla"])
let s2   = ("The Programming Language", [Sec s21; Sec s22; Sec s23])
let s3   = ("Tasks", [Sec("Frontend", [Par "Bla"]);
                      Sec("Backend", [Par "Bla"])])
let doc  = ("Compiler project", [Par "Bla"; Sec s1; Sec s2; Sec s3]);;
```

where `doc` describes a document with title `"Compiler project"`. This document has three sections, where the section with title `"Statements"` has a subsection with the title `"Repeat statements"`, and so on.

Hint: Notice the mutual recursion in the declarations of the types `Document` and `Element`. You may consider using mutually recursive auxiliary functions in your solutions to the below questions.

1. Declare a function `noOfSecs` d that counts the number of sections (including subsections) in the document d. For example, `noOfSecs doc` is 13 (the number occurrences of constructor `Sec` in the value of `doc`).

2. Declare a function `sizeOfDoc` d that gives the number of characters in document d, that is, the sum of the lengths of all strings occurring in titles and paragraphs in d.

3. Declare a function `titlesInDoc` d that gives a list containing all the titles of sections and subsections occurring in document d. For example, `titlesInDoc doc` should contain 13 strings including `"Backend"` and `"Background"`; but it should *not* contain `"Compiler Project"` as this is the title of the entire document and not a section title.

 ... Continued on next page

We shall use integer lists, called *prefixes*, to identify sections in documents in the way you are familiar with from text documents. The empty prefix, that is [], identifies the title of the entire document. The *table of contents* of a document is a list of pairs of prefixes and titles (of matching (sub)sections or of the entire document):

```
type Prefix = int list;;
type ToC    = (Prefix * Title) list
```

For example, the subsection with title `"Boolean Expressions"` is identified by the prefix [2; 1; 2] as it occurs in the second subsection, of the first subsection in the second section of `doc`. The title for this subsection could appear as **2.1.2 Boolean Expressions** in a text document. The table of contents for `doc` is

```
[([], "Compiler project");
 ([1], "Background");
 ([2], "The Programming Language");
 ([2;1], "Expressions");
 ([2;1;1], "Arithmetical Expressions");
 ([2;1;2], "Boolean Expressions");
 ([2;2], "Statements");
 ([2;2;1], "Basics");
 ([2;2;2], "Switch statements");
 ([2;2;3], "Repeat statements");
 ([2;3], "Programs");
 ([3], "Tasks");
 ([3;1], "Frontend");
 ([3;2], "Backend")]
```

4. Declare a function `toc: Document` → `ToC` that generates the table of contents for a document.

   Hint: You may consider using mutually recursive auxiliary functions that may take prefixes and (in some cases) section counters as extra arguments.