Written Examination, December 18th, 2013 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 3 problems which are weighted approximately as follows: Problem 1: 30%, Problem 2: 30%, Problem 3: 40%

Marking: 7 step scale.

Problem 1 (Approx. 30%)

A *multiset* (or *bag*) is a generalization of a set, where an element e is associated with a *multiplicity*, i.e. the number of times e occurs in the multiset. We shall represent a finite multiset ms by a list of pairs $[(e_1, n_1); \ldots; (e_k, n_k)]$, where a member (e_i, n_i) represents that e_i is a member of ms with multiplicity n_i, i.e. e_i occurs n_i times in ms.

For a representation $[(e_1, n_1); \ldots; (e_k, n_k)]$ of a multiset we require that every multiplicity n_i is positive, and that the elements are distinct, i.e. $e_i \neq e_j$, for $i \neq j$. This property is called the *multiset invariant*. A consequence of this is that the empty multiset is represented by the empty list.

We shall use the type `Multiset<'a>` declared as follows:

```
type Multiset<'a when 'a : equality> = ('a * int) list;;
```

For example `[("b",3); ("a",5); ("d",1)]` has type `Multiset<string>` and represents the multiset with 3 occurrences of "b", 5 of "a" and 1 of "d".

1. Declare a function `inv: Multiset<'a> -> bool` such that `inv`(ms) is true when ms satisfies the multiset invariant.

In your solutions to the below questions, you can assume that multisets occurring in arguments satisfy the multiset invariant, and the declared functions must preserve this property, i.e. results must satisfy this invariant as well.

2. Declare a function `insert: 'a -> int -> Multiset<'a> -> Multiset<'a>`, where `insert` e n ms is the multiset obtained by insertion of n occurrences of the element e in ms. For example: `insert "a" 2 [("b",3); ("a",5); ("d",1)]` will result in a multiset having 7 occurrences of "a".

3. Declare a function `numberOf`, where `numberOf` e ms is the multiplicity (i.e. the number of occurrences) of e in the multiset ms. State the type of the declared function.

4. Declare a function `delete`, where `delete` e ms is the multiset obtained from ms by deletion of one occurrence of the element e.

5. Declare a function `union: Multiset<'a> * Multiset<'a> -> Multiset<'a>`, for making the union of two multisets. This function generalizes the union function on sets in a natural way taking multiplicities into account, e.g. the result of

   ```
   union ([("b",3); ("a",5); ("d",1)], [("a",3); ("b",4); ("c",2)])
   ```

   is the multiset containing 8 occurrences of "a", 7 of "b", 2 of "c", and 1 of "d".

 ... Continued on next page

We shall now represent multisets by maps from elements to multiplicities:

```
type MultisetMap<'a when 'a : comparison> = Map<'a,int>;;
```

This representation of a multiset ms has a simpler invariant: the multiplicity n of each entry (e, n) of ms satisfies $n > 0$.

6. Give new declarations for `inv`, `insert` and `union` on the basis of the map representation.

Problem 2 (30%)

Consider the following F# declarations:

```
let rec f i = function
              | []    -> []
              | x::xs -> (i,x)::f (i*i) xs;;


type 'a Tree = | Lf
               | Br of 'a Tree * 'a * 'a Tree;;

let rec g p = function
              | Lf                  -> None
              | Br(_,a,t) when p a -> Some t
              | Br(t1,a,t2)        -> match g p t1 with
                                      | None -> g p t2
                                      | res  -> res;;
```

Please remember that the declaration of `'a option` is

```
type 'a option = None | Some of 'a;
```

1. Give the types of `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.
2. The function `f` is *not* tail recursive.
    1. Make a tail-recursive variant of `f` using an accumulating parameter.
    2. Make a continuation-based tail-recursive variant of `f`.
    3. Give a brief discussion of which tail-recursive version of `f` you prefer?

Consider now the F# declarations:

```
let rec h f (n,e) = match n with
                    | 0 -> e
                    | _ -> h f (n-1, f n e);;

let A = Seq.initInfinite id;;

let B = seq { for i in A do
                  for j in seq {0 .. i} do
                        yield (i,j)        };;

let C = seq { for i in A do
                  for j in seq {0 .. i} do
                      yield (i-j,j)        };;

let X = Seq.toList (Seq.take 4 A);;
let Y = Seq.toList (Seq.take 6 B);;
let Z = Seq.toList (Seq.take 10 C);;
```

3. Consider the function `h` in this question:

    1. What is the value of `h ( * ) (4,1)`?
    2. What is the type of `h`?
    3. Describe briefly what `h` computes.

3. Consider the declarations for `A`, `B`, `C`, `X`, `Y` and `Z`:

    1. Give types for `A`, `B` and `C`.
    2. Give the values of `X`, `Y` and `Z`?
    3. Characterize the values of `A`, `B` and `C`.

... Continued on next page

Problem 3 (40%)

We shall now consider *books* that are described by a list of *chapters*. Each chapter is described by a *title* and a list of *sections*. A section is described by a title and a list of *elements*, which can either be *paragraphs* (characterized by strings) or *sub-sections*. This is captured by the following type declarations:

```
type Title = string;;

type Section = Title * Elem list
and  Elem    = Par of string | Sub of Section;;

type Chapter = Title * Section list;;
type Book    = Chapter list;;
```

The mutual recursion between sections and elements allows for arbitrary nesting of sub-sections. This is illustrated by the following examples:

```
let sec11 = ("Background", [Par "bla"; Sub(("Why programming", [Par "Bla."]))]);;
let sec12 = ("An example", [Par "bla"; Sub(("Special features", [Par "Bla."]))]);;
let sec21 = ("Fundamental concepts",
             [Par "bla"; Sub(("Mathematical background", [Par "Bla."]))]);;
let sec22 = ("Operational semantics",
             [Sub(("Basics", [Par "Bla."])); Sub(("Applications", [Par "Bla."]))]);;
let sec23 = ("Further reading",  [Par "bla"]);;
let sec31 = ("Overview", [Par "bla"]);;
let sec32 = ("A simple example", [Par "bla"]);;
let sec33 = ("An advanced example", [Par "bla"]);;
let sec41 = ("Status", [Par "bla"]);;
let sec42 = ("What's next?", [Par "bla"]);;
let ch1 = ("Introduction", [sec11;sec12]);;
let ch2 = ("Basic Issues", [sec21;sec22;sec23]);;
let ch3 = ("Advanced Issues", [sec31;sec32;sec33;sec34]);;
let ch4 = ("Conclusion", [sec41;sec42]);;
let book1 = [ch1; ch2; ch3; ch4];;
```

1. Declare a function `maxL` to find the largest integer occurring in a list with non-negative integers. The function must satisfy `maxL [] = 0`.

2. Declare a function `overview` to extract the list of titles of chapters from a book. For example, the overview for `book1` is:

```
overview book1 =
   ["Introduction"; "Basic Issues"; "Advanced Issues"; "Conclusion"]
```

 ... Continued on next page

Each chapter occurs at *depth* 1. A top-level section, i.e. a section which is not a sub-section, occurs a depth 2. A sub-section has a depth which is one larger than the depth of the section of which it is an immediate part. For example, the depth of the sub-section with title "Applications" in `book1` is 3 and the section with title "Overview" has depth 2.

3. Declare functions:

```
depthSection: Section -> int
depthElem:    Elem -> int
depthChapter: Chapter -> int
depthBook:    Book -> int
```

to extract the maximal depth of sections, elements, chapters and books. For example the maximal depth of `book1` is 3, as `book1` has sub-sections, but no sub-sub-section.

We shall now make a *table of contents* (type `Toc` below) for a book. In a table of contents we use lists to number entries (see the types `Entry` and `Numbering` below). A *numbering* such as $[i; j; k; l]$ is the number of the l'th sub-sub-section, of the k'th sub-section of the j'th section in the i'th chapter. Notice that such lists have varying lengths. For example, [2] is the number of Chapter 2, i.e the chapter with title "Basic Issues" in the previous example, and [2;2;1] is the number of the sub-section with title "Basics" in Chapter 2.

```
type Numbering = int list;;
type Entry = Numbering * Title;;
type Toc = Entry list;;
```

The table of contents for the previous example is:

```
[([1], "Introduction");
 ([1; 1], "Background");
 ([1; 1; 1], "Why programming");
 ([1; 2], "An example");
 ([1; 2; 1], "Special features");
 ([2], "Basic Issues");
 ([2; 1], "Fundamental concepts");
 ([2; 1; 1], "Mathematical background");
 ([2; 2], "Operational semantics");
 ([2; 2; 1], "Basics");
 ([2; 2; 2], "Applications");
 ([2; 3], "Further reading");
 ([3], "Advanced Issues");
 ([3; 1], "Overview");
 ([3; 2], "A simple example");
 ([3; 3], "An advanced example");
 ([3; 4], "Summary");
 ([4], "Conclusion");
 ([4; 1], "Status");
 ([4; 2], "What's next?")]
```

4. Declare a function, `tocB: Book` $\rightarrow$ `Toc`, to compute the table of contents for a book.