Written Examination, May 28th, 2014 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.

In Problem 2 you are asked to provide F# declarations for a number of functions. You may use a function specified in Problem 2 in a solution to a question in Problem 3 even when you do not provide a declaration for the used function.

Problem 1 (30%)

Consider the following F# declarations:

```
let rec f n = function | 0          -> 1
                       | k when k>0 -> n * (f n (k-1))
                       | _          -> failwith "illegal argument";;

let rec g p f = function
                | []             -> []
                | x::xs when p x -> f x :: g p f xs
                | _::xs          -> g p f xs;;

type T = | A of int
         | B of string
         | C of T*T;;

let rec h = function
            | A n      -> string n
            | B s      -> s
            | C(t1,t2) -> h t1 + h t2;;

let sq = Seq.initInfinite (fun i -> 3*i);;

let k j = seq {for i in sq do
                  yield (i,i-j) };;

let xs = Seq.toList (Seq.take 4 sq);;
let ys = Seq.toList (Seq.take 4 (k 2));;
```

1. Give an example of an application of each of the functions `f`, `g` and `h`.
2. Give the (most general) types of `f`, `g` and `h` 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.
3. 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`.
4. Give types for `sq` and `k`. Characterize the value of `sq` and describe what the function `k` computes.
5. Give the values of `xs` and `ys`.

 ... Continued on next page

Problem 2 (Approx. 30%)

In the following two questions we are interested in deciding two properties (called `ordered` and `smallerThanAll`) in connection with integer lists:

- `ordered` xs is true iff xs is a *weakly ascending* list, that is $x_0 \leq x_1 \leq x_2 \leq \cdots \leq x_{n-1}$ when $xs = [x_0; x_1; x_2; \ldots\ x_{n-1}]$.
- `smallerThanAll` $x\ xs$ is true iff x is smaller than every element of the list xs.

1. Make an F# declaration for the function `ordered`. What is the (most general) type of the declared function?
2. Make an F# declaration for the function `smallerThanAll`. What is the (most general) type of the declared function?

Consider now the following insertion function on lists:

```
insertBefore: ('a -> bool) -> 'a -> 'a list -> 'a list
```

where `insertBefore` $p\,x\,xs$ inserts x in xs just before the first element x_k of xs satisfying p (i.e. $p(x_k) = \text{true}$). Let $xs = [x_0; \ldots; x_{n-1}]$ in the following more formal definition:

`insertBefore` $p\,x\,xs$

$$= \begin{cases} [x_0; \ldots; x_{n-1}; x] & \text{if } p(x_i) = \text{false for all } i\text{: } 0 \leq i < n \\ [x_0; \ldots; x_{k-1}; x; x_k; \ldots; x_{n-1}] & \text{if for some } k : 0 \leq k < n,\ p(x_k) = \text{true and} \\ & \qquad p(x_i) = \text{false for all } i\text{: } 0 \leq i < k. \end{cases}$$

3. Give an F# declaration for `insertBefore`.

4. Consider the following type for the sex of a person:

```
type Sex = | M              // male
           | F              // female
```

   Declare a function `sexToString: Sex -> string`, where the string representation of `M` is `"Male"` and the string representation of `F` is `"Female"`.

5. Declare a function `replicate` with the type `int -> string -> string`. The value of `replicate` n *str* is the string obtained by concatenating n copies of *str*. The function should raise an exception when the integer argument n is negative. For example, `replicate 3 "abc"` = `"abcabcabc"` and `replicate 0 "abc"` = `""`.

Problem 3 (40%)

We shall now consider *family trees*, also called *trees of descendants*, which give an overview of the descendants of a person, called the *ancestor*, in the form of a tree. The list of *children* of a person form the basis for the trees of descendants of that person. Information about persons, such as *name*, *sex* and *year of birth*, occurs in the nodes of the trees.

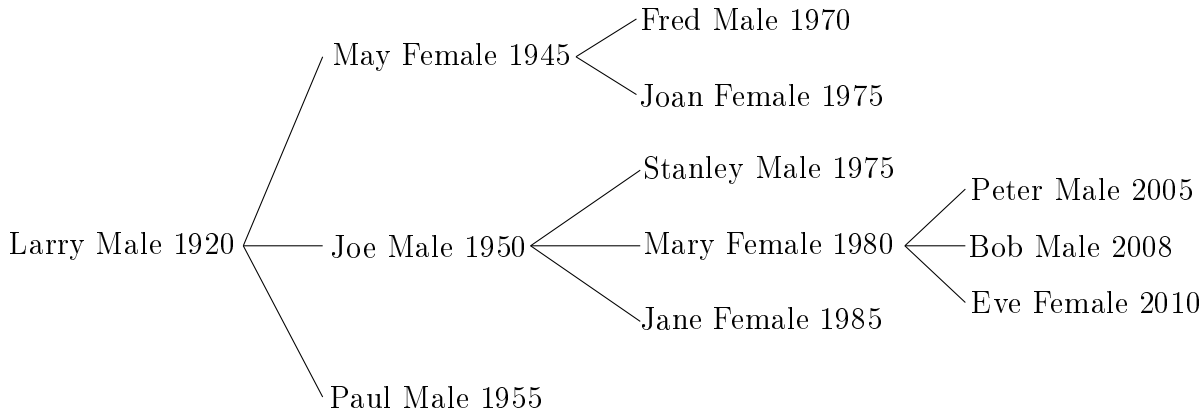

Figure 1: A Family Tree

A family tree is shown in Figure 1, where the name of the ancestor is Larry, a male person born in 1920. The children of Larry are May, Joe and Paul. Larry has five grandchildren — one is Mary and she has three children.

Family trees are represented in F# as follows:

```
type Name = string;;
type Sex = | M              // male
           | F              // female
type YearOfBirth = int;;

type FamilyTree = P of Name * Sex * YearOfBirth * Children
and Children = FamilyTree list;;
```

The type for the sex is as in Problem 2. From now on we assume that the name of a person is *unique* in a family tree and that two persons in a family tree do not share children.

For F# representations of family trees we shall consider two properties:

1. Every person must be older than his/her children.
2. For every list of children, the siblings occur with decreasing ages so that the eldest occurs first and the youngest last.

A family tree satisfying these two properties is called *well-formed*.

1. Declare a function `isWF: FamilyTree -> bool` that can check whether a family tree is well-formed.

In the following questions you can assume that an argument (of type `FamTree` or type `Children`) to a function is well-formed. Furthermore, it is expected that the functions produce well-formed values (when they are of type `FamTree` or type `Children`).

2. Declare a function `makePerson: Name*Sex*YearOfBith -> FamilyTree` that can create a family tree for a child-less person on the basis of the name, sex and year of birth.

3. Declare two mutually recursive functions:

```
insertChildOf: Name -> FamilyTree -> FamilyTree -> FamilyTree option
insertChildOfInList: Name -> FamilyTree -> Children   -> Children option
```

   The value of `insertChildOf` $n\,c\,t$ = `Some` t' when t' is the family tree obtained from t by insertion of c as a child of the person with name n. The value is `None` if such an insertion is not possible (i.e. would create a tree that is not well-formed). Similarly, the value of `insertChildOfInList` $n\,c\,cs$ = `Some` cs' when cs' is the list of children obtained from cs by inserting c as a child of a person named n in one of the children in cs. The value is `None` if such an insertion is not possible. Note that the person named n may occur anywhere in the family tree.

4. Declare a function `find` so that `find` $n\,t$ extracts information about the person named n in the family tree t. This information comprises the sex, year of birth and the names of all children of that person.

Family trees are sometimes presented using an indentation scheme where each person in the tree occurs on a separate line and where the next generation is indented a specific number of positions. For example, the presentation of the family tree in Figure 1 is:

```
Larry Male 1920
      May Female 1945
            Fred Male 1970
            Joan Female 1975
      Joe Male 1950
            Stanley Male 1975
            Mary Female 1980
                  Peter Male 2005
                  Bob Male 2008
                  Eve Female 2010
            Jane Female 1985
      Paul Male 1955
```

using six blank characters as indentation between generation. The ancestor, that is Larry, appears at the first line. His eldest child May appears six positions indented at the second line. Her eldest child appears twelve positions indented at the third line and her second (and youngest) child appears twelve positions indented at the fourth line, and so on.

5. Declare a function `toString` n t that gives a string representation for the family tree t using n blank characters as indentation between generation. The above example string is, for example, generated using `toString 6 larry`, where `larry` is the F# value for the family tree in Figure 1.

6. A restricted kind of family trees is occasionally considered where daughters are included in the tree; but their children are not. Declare a function `truncate` that produces such a restricted family tree from a given family tree. Truncating the family tree in Figure 1 should give a family tree with the following string representation:

```
Larry Male 1920
      May Female 1945
      Joe Male 1950
            Stanley Male 1975
            Mary Female 1980
            Jane Female 1985
      Paul Male 1955
```

when using `toString 6 (truncate larry)`.