Written Examination, December 19th, 2012 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: 25%, Problem 2: 35%, Problem 3: 20%, Problem 4: 20%

Marking: 7 step scale.

Problem 1 (Approx. 25%)

In this problem we will consider simple competitions, where persons, identified by their names, achieve scores. A result is a pair (n, sc) consisting of a name n (given by a string) and a score sc (given by an integer). This leads to the following declarations:

```
type Name   = string;;
type Score  = int;;
type Result = Name * Score;;
```

A score is called legal if it is greater than or equal to 0 and smaller than or equal to 100.

1. Declare a function `legalResults: Result list -> bool` that checks whether all scores in a list of results are legal.
2. Declare a function `maxScore` that extracts the best score (the largest one) in a non-empty list of results. If the list is empty, then we do not care about the result of the function.
3. Declare a function `best: Result list -> Result` that extracts the best result from a non-empty list of results. An arbitrary result with the best score can be chosen if there are more than one. If the list is empty, then we do not care about the result of the function.
4. Declare a function `average: Result list -> float` that finds the average score for a non-empty list of results. If the list is empty, then we do not care about the result of the function.
5. Declare a function `delete: Result -> Result list -> Result list`. The value of `delete` r rs is the result list obtained from rs by deletion of the first occurrence of r, if such an occurrence exists. If r does not occur in rs, then `delete` r $rs = rs$.
6. Declare a function `bestN: Result list -> int -> Result list`, where the value of `bestN` rs n, for $n \geq 0$, is a list consisting of the n best results from rs. The function should raise an exception if rs has fewer than n elements.

 ... Continued on next page

Problem 2 (Approx. 35%)

In this problem we consider simple *type checking* in connection with a simple imperative language. We consider types given by the following declaration of a type `Typ`.

```
type Typ = | Integer
           | Boolean
           | Ft of Typ list * Typ;;
```

Hence, we have an integer type (constructor `Integer`), a Boolean type (construct `Boolean`) and function types constructed using the constructor `Ft`, where `Ft`$([t_1; t_2; \ldots; t_n], t)$, is the type for a function having n arguments with types $t_1, \ldots, t_n$ and the value of the function has type t. The addition function has the type `Ft([Integer;Integer],Integer)` and the greater than function has the type `Ft([Integer;Integer],Boolean)`, for example.

A *declaration* is a pair (x, t) of type `Decl`, which associates the type t with a *variable* x:

```
type Decl = string * Typ;;
```

For a list of declarations $[(x_0, t_0); \ldots; (x_n, t_n)]$ we shall require that the variables are all different, that is, $x_i \neq x_j$, when $i \neq j$.

1. Declare a function `distinctVars: Decl list -> bool`, where `distinctVars` *decls* returns true if all variables in *decls* are different.

You can from now on assume that the variables in a declaration list are different.

A *symbol table* associates types with the variables and functions in programs. We model symbol tables by values of the following `Map` type, where an entry associate a type with a string:

```
type SymbolTable = Map<string,Typ>;;
```

2. Declare a function `toSymbolTable: Decl list -> SymbolTable` that transforms a list of declarations into a symbol table.
3. Declare a function `extendST: SymbolTable -> Decl list -> SymbolTable`, where the value of `extendST` *sym decls* is the symbol table obtained from *sym* by adding entries (x, t), for every declaration (x, t) in *decls*. An existing entry in *sym* having x as key will be overridden by this operation.

 ... Continued on next page

We consider expressions generated from variables (constructor `V`) using function application (constructor `A`), where, e.g., `A(">",[V "x";V "y"])` represents the comparison $x > y$:

```
type Exp = | V of string
           | A of string * Exp list;;
```

Suppose that a symbol table *sym* associates the type `Integer` with `"x"` and `"y"`, and the type `Ft([Integer;Integer],Boolean)` with `">"`. All symbols (variables and functions) in the expression `A(">",[V "x";V "y"])` are therefore defined in *sym*. Furthermore, the expression is well-typed since the types of the arguments to `>` match the argument types in `Ft([Integer;Integer],Boolean)`, and the type of `A(">",[V "x";V "y"])` is `Boolean`.

4. Declare a function `symbolsDefined: SymbolTable -> Exp -> bool`, where the value of the expression `symbolsDefined` *sym* *e* is true if there is an entry in *sym* for every symbol (variable or function) occurring in *e*.

5. Declare a function `typOf: SymbolTable -> Exp -> Typ`, so that `typOf` *sym* *e* gives the type of *e* for the symbol table *sym*. The function should raise an exception if *e* is not well-typed. You may assume that all symbols in *e* are defined in *sym*.

We consider statements generated from assignments using sequential composition, if-then-else statements, while statements and block statements:

```
type Stm = | Ass of string * Exp          // assignment
           | Seq of Stm * Stm             // sequential composition
           | Ite of Exp * Stm * Stm       // if-then-else
           | While of Exp * Stm           // while
           | Block of Decl list * Stm;;   // block
```

The *well-typedness* of a statement for a given symbol table *sym* is given by:

- An assignment `Ass`(x, e) is well-typed if x and the symbols of e are defined in *sym* and x and e have the same type.
- A sequential composition `Seq`(stm_1, stm_2) is well-typed if stm_1 and stm_2 are.
- An if-then-else statement `Ite`(e, stm_1, stm_2) is well-typed if the symbols in e are defined in *sym*, e has type `Boolean`, and stm_1 and stm_2 are well-typed.
- A while statement `While`(e, stm) is well-typed if the symbols in e are defined in *sym*, e has type `Boolean`, and *stm* is well-typed.
- A block statement `Block`$(decls, stm)$ is well-typed if the variables in *decls* are all different, and *stm* is well-typed in the symbol table obtained by extending *sym* with the declarations of *decls*.

6. Declare a function `wellTyped: SymbolTable -> Stm -> Bool` that checks that a statement is well-typed for a given symbol table, and if so returns true.

 ... Continued on next page

Problem 3 (20%)

Consider the following F# declarations:

```
let rec h a b =
    match a with
    | []   -> b
    | c::d -> c::(h d b);;

type T<'a,'b> = | A of 'a | B of 'b | C of T<'a,'b> * T<'a,'b>;;

let rec f1 = function
    | C(t1,t2) -> 1 + max (f1 t1) (f1 t2)
    | _        -> 1;;

let rec f2 = function
    | A e | B e  -> [e]
    | C(t1,t2)   -> f2 t1 @ f2 t2;;

let rec f3 e b t =
    match t with
    | C(t1,t2) when b -> C(f3 e b t1, t2)
    | C(t1,t2)        -> C(t1, f3 e b t2)
    | _        when b -> C(A e, t)
    | _               -> C(t, B e);;
```

1. Give the type of `h` and describe what `h` computes. Your description should focus on *what* it computes, rather than on individual computation steps.
2. Write a value of type `T<int,bool>` using all three constructors `A`, `B` and `C`.
3. Write a value of type `T<'a list,'b option>` using all three constructors `A`, `B` and `C`.
4. Give the types of `f1`, `f2` and `f3`, and describe what each of these three functions compute.

 ... Continued on next page

Problem 4 (Approx. 20%)

Consider the following F# declarations:

```
type 'a tree = | Lf
               | Br of 'a * 'a tree * 'a tree;;

let rec sumTree = function
    | Lf            -> 0                               (* sT1 *)
    | Br(x, t1, t2) -> x + sumTree t1 + sumTree t2;;   (* sT2 *)

let rec toList = function
    | Lf            -> []                              (* tL1 *)
    | Br(x, t1, t2) -> x::(toList t1 @ toList t2);;    (* tL2 *)

let rec sumList = function
    | []    -> 0                                       (* sL1 *)
    | x::xs -> x + sumList xs;;                        (* sL2 *)

let rec sumListA n = function
    | []    -> n                                       (* sLA1 *)
    | x::xs -> sumListA (n+x) xs;;                     (* sLA2 *)
```

1. Prove that
$$\texttt{sumTree}\ t = \texttt{sumList}(\texttt{toList}\ t)$$
holds for all trees t of type `int tree`.
In the proof you can assume that
$$\begin{aligned}&\texttt{sumList}((\texttt{toList}\,t_1)\ \texttt{@}\ (\texttt{toList}\,t_2))\\&= \texttt{sumList}(\texttt{toList}\,t_1) + \texttt{sumList}(\texttt{toList}\,t_2)\end{aligned}$$
holds for all trees t_1 and t_2 of type `int tree`.
2. Prove that
$$\texttt{sumListA}\ n\ xs = n + \texttt{sumList}(xs)$$
holds for all integers n and all lists xs of type `int list`.