Written Examination, December 20th, 2011 Course no. 02157

The duration of the examination is 2 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: 30%, Problem 2: 30%, Problem 3: 20%, Problem 4: 20%

Marking: 7 step scale.

Problem 1 (Approx. 30%)

In this problem we will consider a simple *register* for members of a sports club. It could be a tennis club, a fishing club, or .... To keep the problem simple we identify members by their names and each member is described by a *phone number* and a *level*, where the level is an indication of how well the member is performing in this sport. Phone numbers and levels are modelled by integers and we arrive at the following declarations:

```
type name  = string;;
type phone = int;;
type level = int;;

type description = phone * level;;
type register = (name * description) list;;
```

1. Declare a value of type `register`, that contains four members: Joe (having phone number: 10101010 and level: 4) , Sal (having phone number: 11111111 and level: 2), Sam (having phone number: 12121212 and level: 7), Jane (having phone number: 13131313 and level: 1).

2. Declare a function `getPhone: name -> register -> phone` to extract the phone number of a member in a register. The function should raise an exception `Register` if the member is not occurring in the register.

3. Declare a function `delete: name * register -> register` to delete the entry for a member in a register.

4. We say that two levels l and l' *match* if one is at most two larger than the other, i.e. if $|l - l'| < 3$.

   Declare a function `getCandidates: level -> register -> (name*phone) list`, that for a given level l and register *reg* gives the name and phone number of all members of *reg* with a level matching l. In the example from question 1, Joe and Sam have levels matching the level 5.

 ... Continued on next page

Problem 2 (Approx. 30%)

In this problem we consider simple *expressions*, like $3 + 5 * 2$, which can be constructed from integer constants using binary operators. Such expressions are modelled using the following declaration of the type `exp`:

```
type exp = | C of int
           | BinOp of exp * string * exp;;
```

where the constructor `C` generates an integer constant and the operator is given as a string (e.g. `"+"` and `"*"`) when generating an expression using the constructor `BinOp`.

1. Give three different values of type `exp`.

2. Declare a function `toString: exp -> string`, that gives a string representation for an expression. Put brackets around every subexpression with operators, e.g. `(3+(5*2))` is a string representation of the above example.

3. Declare a function to extract the set of operators from an expression.

4. The type for expressions is now extended to include *identifiers* (constructor `Id`) and *local definitions* (constructor `Def`):

   ```
   type exp = | C of int
              | BinOp of exp * string * exp
              | Id of string
              | Def of string * exp * exp;;
   ```

   where `Def("x", C 5 , BinOp(Id "x", "+", Id "x"))`, for example, denotes the expression where x is defined by the constant 5 in the expression $x+x$. This expression would evaluate to 10.

   We say that an expression is *defined* if it evaluates to an integer value, i.e. if there is a definition for every identifier occurring in the expression. We have, for example, that `Def("x", C 5, BinOp(Id "x", "+", Id "x"))` is defined, whereas the expression `Def("x", C 5, BinOp(Id "y", "+", Id "x"))` is not defined since there is no definition for `"y"`.

   Declare a function `isDef: exp -> bool` that can test whether an expression is defined.

   Hint: make use of an auxiliary function having an extra argument that takes care of defined identifiers.

 ... Continued on next page

Problem 3 (20%)

Consider the following F# declarations:

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

let rec f(n,t) =
    match t with
    | Lf              -> Lf
    | Br(a, t1, t2) -> if n>0 then Br(a, f(n-1, t1), f(n-1, t2))
                       else Lf;;

let rec g p = function
  | Br(a, t1, t2) when p a -> Br(a, g p t1, g p t2)
  | _                      -> Lf;;

let rec h k = function
  | Lf            -> Lf
  | Br(a, t1, t2) -> Br(k a, h k t1, h k t2);;
```

1. Give the types of `f`, `g` and `h`, and describe what each of these three functions compute. Your description for each function should focus on *what* it computes, rather than on individual computation steps.

Problem 4 (Approx. 20%)

Consider the following F# declarations:

```
let rec map f = function
    | []    -> []                       (* m1 *)
    | x::xs -> f x :: map f xs;;        (* m2 *)

let rec rev = function
  | []    -> []                         (* r1 *)
  | x::xs -> rev xs @ [x];;             (* r2 *)
```

Prove that

$$\texttt{rev}\ (\texttt{map}\ f\ xs) \ = \ \texttt{map}\ f\ (\texttt{rev}\ xs)$$

holds for all functions f and lists xs of appropriate types.

In your proof you can assume that

$$\texttt{map}\ f\ (xs\ \texttt{@}\ ys) \ = \ (\texttt{map}\ f\ xs)\ \texttt{@}\ (\texttt{map}\ f\ ys)$$

holds for all functions f and lists xs and ys of appropriate types.