Informatik and Mathematical Modelling — 27/11 2012
DTU — Michael R. Hansen

Below are solutions to the exam in 02157, December 2011.

## Problem 1

1. ```
   let reg1 = [("Joe", (10101010,4));
               ("Sal", (11111111,2));
               ("Sam", (12121212,7));
               ("Jane", (13131313,1))];;
   ```

2. ```
   exception Register;;
   let rec getPhone n = function
      | []   -> raise Register
      | (n1,(p,_))::reg -> if n=n1 then p else getPhone n reg;;
   ```

   The type of `getPhone` is `'a -> ('a * ('b * 'c)) list -> 'b when 'a : equality`. The specified type `name -> register -> phone` is an instance of this type.

3. ```
   let rec delete(n,reg) =
      match reg with
      | []  -> reg
      | (n1,_)::reg' when n=n1 -> reg'
      | entry::reg'            -> entry:: delete(n,reg');;
   ```

   The type of `delete` is `'a * ('a * 'b) list -> ('a * 'b) list when 'a : equality`. The specified type `name * register -> register` is an instance of this type.

4. ```
   let rec getCandidates l = function
      | []               -> []
      | (n,(p,l'))::reg -> if abs(l-l')<3 then (n,p) :: getCandidates l reg
                           else getCandidates l reg;;
   ```

   The type of `getCandidates` is `int -> ('a * ('b * int)) list -> ('a * 'b) list`. The specified type `level -> register -> (name*phone) list` is an instance of this type.

## Problem 2

1. Three values: `C 3`, `BinOp(C 5,"*",C 2)` and `BinOp(C 3, "+", BinOp(C 5,"*",C 2))`.

2. 
```
let rec toString = function
     | C n -> string n
     | BinOp(e1,o,e2) -> "(" + toString e1 + o + toString e2 + ")";;
```

3. 
```
let rec ops = function
   | BinOp(e1,o,e2) -> Set.add o (Set.union (ops e1) (ops e2))
   | _              -> Set.empty;;
```

   The type of `ops` is `exp -> Set<string>`

4. 
```
let rec isDefAux ids = function
   | Id x           -> Set.contains x ids
   | Def(x,e,e1)    -> isDefAux ids e && isDefAux (Set.add x ids) e1
   | BinOp(e1,_,e2) -> isDefAux ids e1 && isDefAux ids e2
   | _              -> true;;

let isDef e = isDefAux Set.empty e;;
```

   The type of `isDefAux` is `Set<string> -> exp -> bool` and the type of `isDef` is `exp -> bool`

## Problem 3

1. We call values of type `'a tree` trees as usual.

   - The type of `f` is `int * 'a tree -> 'a tree`. `f` "cuts" trees at a given depth in the sense that $\mathtt{f}(n, t)$ is the three obtained from $t$ by replacing every subtree occurring in $t$ at depth $n$ with `Lf`.
   - The type of `g` is `('a -> bool) -> 'a tree -> 'a tree`. `g` is a filter function on trees in the sense that `g` $p$ $t$ is the tree obtained from $t$ by replacing every subtree $\mathtt{Br}(a, t_1, t_2)$ in $t$, where $p(a)$ is false, with `Lf`. Note that it is of no significance if $p$ does not hold on values occurring in the subtrees $t_1, t_2$, because the entire tree is replaced by `Lf` when $p(a)$ is false.
   - The type of `h` is `('a -> 'b) -> 'a tree -> 'b tree`. `h` is a map function on trees in the sense that `h` $k$ $t$ is the tree obtained from $t$ by replacing every value $a$ occurring in some node $\mathtt{Br}(a, \_, \_)$ in $t$, with $k\ a$.

## Problem 4

We prove

$$\forall xs.\texttt{rev}\ (\texttt{map}\ f\ xs) = \texttt{map}\ f\ (\texttt{rev}\ xs) \tag{1}$$

by structural induction on lists.

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

The base case $P([\,])$ is established by:

$$\begin{array}{lll} & \texttt{rev}\ (\texttt{map}\ f\ [\,]) & \\ = & \texttt{rev}\ [\,] & \text{using m1} \\ = & [\,] & \text{using r1} \\ = & \texttt{map}\ f\ [\,] & \text{using m1} \\ = & \texttt{map}\ f\ (\texttt{rev}\ [\,]) & \text{using r1} \end{array}$$

In the inductive step we must establish: $\forall xs, x.(P(xs) \implies P(x :: xs)$.

Let $xs$ be an arbitrary list, $x$ an arbitrary element (of appropriate types). Assume $P(xs)$, that is:

$$\texttt{rev}\ (\texttt{map}\ f\ xs) = \texttt{map}\ f\ (\texttt{rev}\ xs) \qquad \text{(induction hypothesis)}$$

We must prove $P(x :: xs)$, that is: $\texttt{rev}\ (\texttt{map}\ f\ (x :: xs)) = \texttt{map}\ f\ (\texttt{rev}\ (x :: xs))$.

This is done as follows:

$$\begin{array}{lll} & \texttt{rev}\ (\texttt{map}\ f\ (x :: xs)) & \\ = & \texttt{rev}\ (f\ x :: \texttt{map}\ f\ xs) & \text{using m2} \\ = & (\texttt{rev}\ (\texttt{map}\ f\ xs))\ \texttt{@}\ [f\ x]) & \text{using r2} \\ = & (\texttt{map}\ f\ (\texttt{rev}\ xs))\ \texttt{@}\ [f\ x]) & \text{using ind. hyp} \\ = & (\texttt{map}\ f\ (\texttt{rev}\ xs))\ \texttt{@}\ (\texttt{map}\ f\ [x]) & \text{using m1,m2 - see below} \\ = & (\texttt{map}\ f\ ((\texttt{rev}\ xs)\ \texttt{@}\ [x]) & \text{using given assumption} \\ = & \texttt{map}\ f\ (\texttt{rev}\ (x :: xs)) & \text{using r2} \end{array}$$

Since $\texttt{map}\ f\ [x] = f\ x :: \texttt{map}\ f\ [\,] = f\ x :: [\,] = [f\,x]$ by use of m1 and m2, we have established the induction step.

Therefore, by the structural induction rule for lists we have proved that (1) holds.