CSE 428: Solutions to exercises on ML

The superscript "(d)" stands for "difficult". Exercises similar to those marked with "(d)" might appear in candidacy exams, but not in the standard exams of CSE 428.

---

1. Define a function which computes the product of all integers between m and n (with n >= m) inclusive. Use this function to define the function $C_{n,k}$ (the number of combinations of n elements taken k by k), which is defined by

   $C_{n,k} = n!/(k!*(n-k)!)$

   ## Solution

   ```
   fun prod(m,n) = if n <= m then m else n * prod(m,n-1);

   fun C(n,k) = prod(n-k+1,n) div prod(1,k);
   ```

   ---

2. Define a function

   power : int * int -> int

   so that, for m >=0

   power(n,m) = n^m

   holds. Note: we assume that 0^0 is defined as 1.

   ## Solution

   ```
   fun power(n,m) = if m=0 then 1 else n * power(n,m-1);
   ```

   ---

3. The positive integer square root of a non-negative integer is a function `introot` such that: if `introot` m = n, then n is the largest integer such that n^2 is less than or equal to m. Define the function `introot` in ML.

   ## Solution

   ```
   fun introot m = let fun aux(k,m) = if k*k > m then k-1 else aux(k+1,m)
                    in aux(0,m)
                   end;
   ```

   ---

4. In ML, like in any other language, the if-then-else construct is non-strict, i.e. only one of the branches is evaluated, depending on the the result of the test. What would be the consequences for recursive definitions if the if-then-else were strict (meaning that first both the branches are evaluated, and then one of the results is picked, depending on the result of the test), and pattern-matching were not available?

   ## Solution

   It would not be possible to define recursive functions anymore. In fact, the recursive definition of a function f has typically the following scheme

```
fun f x = if x=0 then < expression > else < expression_with_recursive_call_of f >
```

if the if-then-else were strict, then the evaluation of the recursive call would always be required, even when x=0, and therefore we would always loop. Note that if definition by pattern-matching were available, we could give the alternative definition (which does not require if-then-else):

```
fun f 0 = < expression >
  | f x = < expression_with_recursive_call_of f >
```

---

5. Define a function `copy: int * 'a -> 'a list` such that `copy(k,x)` gives the list containing `k` occurrences of `x`. Examples:

```
copy(0,5) = []
copy(1,5) = [5]
copy(3,"a") = ["a","a","a"]
copy(3,copy(1,8)) = [[8],[8],[8]]
```

Solution

```
fun copy(0,x) = []
  | copy(n,x) = x::copy(n-1,x);
```

---

6. Define a function `sumlists: int list * int list -> int list` which takes in input two lists of integers and gives as result the list of the sums of the elements in corresponding position in the input lists. The shortest list has to be seen as extended with 0's. Examples:

```
sumlists([],[]) = []
sumlists([1,2],[3,4]) = [4,6]
sumlists([1],[3,4,2]) = [4,4,2]
sumlists([1,6],[3]) = [4,6]
```

Solution

```
fun sumlists(l,[]) = l
  | sumlists([],k) = k
  | sumlists(x::l,y::k) = (x+y)::sumlists(l,k);
```

---

7. Define a function `remove_dup: ''a list -> ''a list` which takes in input a list and removes all the duplicates, Examples:

```
remove_dup [] = []
remove_dup [1,2,1] = [1,2]
remove_dup ["a","a","a"] = ["a"]
remove_dup [[1],[1,2],[1,2,3],[1,2],[4,5]] = [[1],[1,2],[1,2,3],[4,5]]
```

Is it possible to define `remove_dup` with a more generl type, i.e. `remove_dup: 'a list -> 'a list`?

Solution

```
fun delete(x,[]) = []
  | delete(x,y::l) = if x=y then delete(x,l) else y::delete(x,l);

fun remove_dup [] = []
  | remove_dup (x::l) = x::remove_dup(delete(x,l));
```

It is not possible to define `remove_dup` with type `'a list -> 'a list`, because we need to check the equality of elements in order to delete duplications.

---

8. Define a function `first_list: ('a * 'b) list -> 'a list` which takes in input a list of pairs and gives back the list consisting of the first elements only, Examples:

```
first_list [] = []
first_list [(1,2),(1,3)] = [1,1]
first_list [(1,"a"),(2,"b"),(3,"c")] = [1,2,3]
first_list [([],"a"),([1],"b"),([1,2],"c")] = [[],[1],[1,2]]
```

Solution

```
fun first_list [] = []
  | first_list((x,y)::l) = x::first_list l;
```

---

9. Define a function `flatten: 'a list list -> 'a list` which takes in input a list of lists and gives back the list consisting of all the elements, in the same order in which they appear in the argument. Examples:

```
flatten [] = []
flatten [[]] = []
flatten [[1,2],[2,3,4],[5],[],[6,7]] = [1,2,2,3,4,5,6,7]
flatten [["a"],["b","a"]] = ["a","b","a"]
```

Solution

```
fun flatten [] = []
  | flatten (x::l) = x @ flatten l;
```

---

10. Consider the data structure "binary tree", defined in Assignment 7:

```
datatype 'a btree = emptybt | consbt of 'a * 'a btree * 'a btree;
```

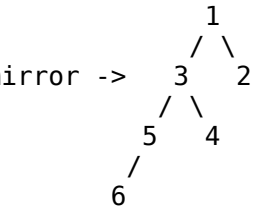
Define a function `sum_tree : int btree -> int` which returns the sum of all the elements of the tree (0 if the tree is empty)

Solution

```
fun sum_tree emptybt = 0
  | sum_tree (consbt(x,t1,t2)) = x + sum_tree t1 + sum_tree t2;
```

---

11. Consider the data structure "binary tree" of previous exercise. Define a function `mirror : 'a btree -> 'a btree` which returns the "mirror" of the input tree. Example:



Solution

```
fun mirror emptybt = emptybt
  | mirror (consbt(x,t1,t2)) = consbt(x, mirror t2, mirror t1);
```

---

12. Define a function `breadth_first: 'a btree -> 'a list` which takes a binary tree and returns a list representing the breadth first visit of the tree (visit by level). Example:

```
        1
       / \
      2   3
     / \   \
    4   5   6  - breadth_first -> [1,2,3,4,5,6,7,8,9]
   /    /
  7    8
        \
         9
```

Solution

The idea is to use an auxiliary function, which visit by level a forest of tree (i.e. several trees) organized as a list of trees. Each time we consider a tree in this forest, we put its root in the list and we append its subtrees at the end of the forest. This FIFO-like discipline ensures that the visit will be performed by level: the next level is appended at the end, hence it won't be considered until we visit all notes of the current level.

```
fun breadth_first t = let fun aux [] = []
                          | aux (emptybt::l) = aux l
                          | aux (consbt(x,t1,t2)::l) = x:: aux(l@[t1,t2])
                      in aux [t]
                     end;
```

---

13. Define a data structure 'a tree, which has an arbitrary number of subtrees for every node. On this structure, define the function `pre_visit`. Example:

```
        1
       / \
      2   3
     /|\  |
    4 5 6 7_     - pre_visit -> [1,2,4,5,6,3,7,8,9,0,9]
        /|\ \
       8 9 0 9
```

Solution

The data structure can be defined by associating to every node a list of subtrees:

```
datatype 'a tree = emptyt | const of 'a * ('a tree) list;
```

In order to define the `pre_visit` function, it is convenient to define an auxiliary function which visits a forest of trees, organized in a list.

```
fun pre_visit t = let fun aux [] = []
                        | aux (emptyt::l) = aux l
                        | aux (const(x,l)::k) = x::(aux l)@(aux k)
                  in aux [t]
                 end;
```

---

14. We want to organize a data base containing information about a number of people in such a way that it is easy to perform a search. To such purpose, we want to use a *search three*. For each person we will have a record containing the following information:
    - the name of the person (represented as a string)
    - the date of birth (represented in the format (DD, MM , AAAA) where DD , MM and AAAA are numbers)
    - the place of birth (represented as a string)

    1. Define the data type suitable to represent the search three
    2. Define a function *build_search_tree* which, given a (unordered) list of records, transforms it into a search tree, as balanced as possible. The key field is the name, i.e. the information has to be organized in the tree with respect to the alphabetical order on names.

3. Define a function *search* that, given a search three and a name, returns the date and place of birth of the person with that name, if it is in the tree, and some message of unsuccessful search otherwise.

Solution

```
(* Data_base of people organized in a search-tree    *)

type date_of_birth = int * int * int;

datatype person = record of string * date_of_birth * string;

(* selection of the fields "name", "date of birth" and "place of birth"
in a person *)

fun name (record(na,da,pl)) = na;
fun date (record(na,da,pl)) = da;
fun place(record(na,da,pl)) = pl;

(* definition of comparison between (the names of) two persons    *)

fun smaller p1 p2 = (name p1) < (name p2);

(* Definition of quick_sort (to be used for ordering the initial list).
   It makes use of an auxiliary function "split" which, given an element
   x and a list l, gives as result a pair (l1,l2) where l1 contains
   the element of l smaller than x, and l2 the elements bigger than x *)

fun split x nil = (nil, nil)
  | split x (y::l) = let val (l1,l2) = split x l
                      in if (smaller y x) then (y::l1 , l2)
                                          else (l1 , y::l2)
                     end;

fun quick_sort nil = nil
  | quick_sort (x::l) = let val (l1,l2) = split x l
                         in (quick_sort l1) @ (x :: (quick_sort l2))
                        end;

(* Definition of the Datatype a' btree *)

datatype 'a btree = emptybt | consbt of 'a * 'a btree * 'a btree;

(* Definition of a function which builds a balanced tree from a list.
 It uses an auxiliary function "partition" which takes a list l and
 an integer k and gives as result the triple (l1,x,l2), where
 l1 is the initial part of l of length k, x is the element which comes
 immediately afterwards, and l2 is the rest.
 Note that the call partition l ((length l) div 2) will produce a
 triple (l1,x,l2) where l1 and l2 have the same length (or almost
 the same length, i.e. they will differe at most by 1), and x is the
 middle element of l *)

fun partition l k = if k = 0 then (nil,(hd l),(tl l))
                             else let val (l1,x,l2) = partition (tl l) (k-1)
                                      in ((hd l)::l1,x,l2)
                                  end;
fun length nil = 0
  | length (x::l) = 1 + (length l);

fun build_balanced nil = emptybt
  | build_balanced l = let val (l1,x,l2) = partition l ((length l) div 2)
                        in consbt(x , (build_balanced l1) ,
                                      (build_balanced l2) )
                       end;

(* Definition of build_search_tree: given a list of persons l,
it first orders it and then applies build_balanced to the
```

```
      resulting (ordered) list *)

      fun build_search_tree l = build_balanced (quick_sort l)

      (* Definition of search: given a name n and a search tree,
      it returns the date and place of birth of the person with the
      the same name in the tree, if any, and a pair ((0,0,0),"") otherwise *)

      fun search n emptybt = ((0,0,0),"")
        | search n (consbt(p,t1,t2)) = if (name p) = n
                                       then ((date p),(place p))
                                       else if (name p) > n
                                              then (search n t1)
                                              else (search n t2)

      (* ------------------------------------------------------------- *)
```
