```
// Solution to the exam set in 02157 Functional Programming, 2016 Dec
//                                       Michael R. Hansen, 24-11-2020
//

//Problem 1

type Name = string
type Event = string
type Point = int
type Score = Name * Event * Point

type Scoreboard = Score list

let sb = [("Joe", "June Fishing", 35); ("Peter", "May Fishing", 30);
          ("Joe", "May Fishing", 28); ("Paul", "June Fishing", 28)];;

//1  inv: Scoreboard -> bool
let rec inv = function
              | []                        -> true
              | [(n,e,p)]                 -> p>=0
              | (n,e,p)::(n1,e1,p1):: sb -> p>=p1 && inv ((n1,e1,p1)::sb)


//2  insert: Score -> Scoreboard -> Scoreboard
let rec insert (n,e,p) = function
                         | []                        -> [(n,e,p)]
                         | (n1,e1,p1)::sb when p>p1 -> (n,e,p)::(n1,e1,p1)::sb
                         | (n1,e1,p1)::sb           -> (n1,e1,p1)::insert (n,e,p) sb;;
//3  get: Name*Scoreboard -> (event*Point) list
let rec get(n,sb) =
                match sb with
                | []                        -> []
                | (n1,e1,p1)::sb1 when n=n1 -> (e1,p1)::get(n,sb1)
                | _::sb1                     -> get(n,sb1);;

//4  top: int -> Scoreboard -> Scoreboard option
let rec top n = function
                | _ when n=0 -> Some []
                | []         -> None
                | s::sb       -> match top (n-1) sb with
                                | None  -> None
                                | Some res  -> Some(s::res)

top 2 sb

// Problem 2
//1
let rec replace a b = function
```

```
                      | x::xs when a=x -> b::replace a b xs     (* 1 *)
                      | x::xs          -> x::replace a b xs     (* 2 *)
                      | []             -> [];;

// The most general type must have the form: t1 -> t2 -> t3 -> t4,
// for some types t1, t2, t3, t4, due to the form of the declaration,
// where a:t1, b:t2, t3 = t1 list due to x=a in (* 1 *)
// Furthermore, t2 = t1 and t4 = t1 list due to b::... in (* 1 *) and x::... in (* 2 *)
// The only constraint on t1 is that it must support equality.
// Hence, most general type is: 'a -> 'a -> 'a list -> 'a list when 'a: equality

// Replace is not tail-recursive, because when the recursive call in (* 1 *) terminates,
// then the cons operation b:: ... remains to be executed,
// that is, this recursive call is not a tail call. Similarly for (* 2 *)


//3 A version with an accumulating parameter is:
let rec replaceA res a b = function
                           | x::xs when a=x -> replaceA (b::res) a b xs
                           | x::xs          -> replaceA (x::res) a b xs
                           | []             -> List.rev res;;


// Problem 3

let pos = Seq.initInfinite (fun i -> i+1) ;;
let seq1 = seq { yield (0,0)
                 for i in pos do
                    yield (i,i)
                    yield (-i,-i) }

let val1 = Seq.take 5 seq1;;

let nat = Seq.initInfinite id;;
let seq2 = seq { for i in nat do
                   yield (i,0)
                   for j in [1 .. i] do
                      yield (i,j) }

let val2 = Seq.toList(Seq.take 10 seq2);;

//1
// pos has type seq<int> and denotes the infinite sequence of
// positive natural numbers: 1, 2, ... i, ...
// seq1 has type seq<int*int> and denotes the infinite sequence of
// pairs: (0,0), (1,1), (-1,-1), ... (i,i), (-i,-i), ...
```

```
// val1 has type seq<int*int> and denotes the following sequence
// of pairs: (0, 0), (1, 1), (-1, -1), (2, 2), (-2, -2)

// seq2 has the type seq<int*int>. It denotes the infinite sequence:
// (0,0), (1,0), (1,1), (2,0), (2,1), (2,2), ....., (i,0), (i,1), ..., (i, i-1), (i,i), ...
// That is, it denotes the infinite sequence of natural number pairs (i,j) where i >= j.
// The order of the pairs is determined by the lexicographical ordering:
// (i,j) occurs before (i',j') if i < i' or (i=i' and j<j').

// Problem 4

type Tree<'a,'b> = A of 'a | B of 'b | Node of Tree<'a,'b>*Tree<'a,'b>;;

// Three values of type Tree<bool, int list>

let v1 = A false;;
let v2 = B [1];;
let v3 = Node(v1,v2);;

//2
let rec countA_Leaves = function | A _ -> 1
                                 | B _ -> 0
                                 | Node(t1,t2) -> countA_Leaves t1 + countA_Leaves t2;;

//3
let rec subst a va b vb = function
                        | Node(t1,t2)    -> Node(subst a va b vb t1, subst a va b vb t2)
                        | A a' when a=a' -> A va
                        | B b' when b=b' -> B vb
                        | leaf           -> leaf;;

let rec g = function
          | Node(t1,t2) -> Node(g t2, g t1)
          | leaf        -> leaf;;

let rec f = function
          | A a         -> ([a],[])
          | B b         -> ([], [b])
          | Node(t1,t2) -> let (xs1,ys1) = f t1
                           let (xs2,ys2) = f t2
                           (xs1@xs2, ys1@ys2);;
// 4
// The most general type for g is Tree<'a,'b> -> Tree<'a,'b>
// The value of g t is a tree t' that is the mirror image of t.
// It would be natural to support this description with a figure.
```

```
// It is formed by exchange of left and right subtrees in t all the way down.

// The most general type of f is Tree<'a,'b> -> 'a list * 'b list
// The value of f t:
// let A x_1, ..., A x_m be the sequence of A-leaves of t as they appear from left to right and
// let B y_1, ..., B y_n be the sequence of B-leaves of t as they appear from left to right.
// Then f t = ([x_1; ...;x_m] ,[y_1; ...; y_n]).

// 5
let rec fK t k = match t with
                 | A a         -> k([a],[])
                 | B b         -> k([], [b])
                 | Node(t1,t2) -> fK t1 (fun (xs1,ys1) -> fK t2 (fun (xs2,ys2) -> k(xs1@xs2,ys1@ys2)));;


// Problem 5

type T<'a> = N of 'a * T<'a> list;;
type Path = int list;;

let td = N("g", []);;
let tc = N("c", [N("d",[]); N("e",[td])]);;
let tb = N("b", [N("c",[])]);;
let ta = N("a", [tb; tc; N("f",[])])

//1
let rec toList (N(v,ts)) = v::List.collect toList ts;;

// A solution based on mutual recursive functions:
let rec toList1(N(v,ts)) = v :: toListAux ts
and toListAux = function
                | []    -> []
                | t::ts -> toList1 t @ toListAux ts

//2

// map has type: ('a -> 'b) -> T<'a> -> T<'b>

let rec map f (N(v,ts)) = N(f v, List.map (map f) ts);;

// A solution based on mutual recursion
let rec map1 f (N(v,ts)) = N(f v, mapAux f ts)
and mapAux f = function
               | [] -> []
               | t::ts -> map1 f t :: mapAux f ts;;

```

```
//3
let rec isPath path t =
   match (path, t) with
   | ([], _)                                                 -> true
   | (i::path', N(v,ts)) when 0 <= i && i < List.length ts -> isPath path' (List.item i ts)
   | _                                                       -> false;;

//4
let rec get1 path t =
   match (path,t) with
   | ([],_)           -> t
   | (i::is, N(_,ts)) -> get1 is (List.item i ts);;


//5
let rec tryFindPathTo v (N(v',ts)) = if v=v' then Some []
                                     else tryFindInList 0 v ts
and tryFindInList i v = function
                        | []                   -> None
                        | N(v',_)::ts when v=v' -> Some [i]
                        | N(_,ts')::ts          -> match tryFindInList 0 v ts' with
                                                   | None -> tryFindInList (i+1) v ts
                                                   | Some is -> Some(i::is);;
```