```
(* Draft solutions to ITU exam in
   Functional programming Spring 2013     Michael R. Hansen 5/12/2013 *)


(* 2.1 *)
let rec f n m = if m=0 then n
                else n * f (n+1) (m-1);;

(*   f : int -> int -> int   *)

(*    f n m = n * (n+1) * ...  * (n + m)    *)

(* 2.2 *)
let rec fA n m a = if m=0 then a
                   else fA (n+1) (m-1) (a*(n+1))

let f1 n m = fA n m n;;

let rec fC n m c = if m=0 then c n
                   else fC (n+1) (m-1) (fun r -> c(r*n));;

let f2 n m = fC n m id;;

(* 2.3 *)
let rec z xs ys = match (xs, ys) with
                  | ([],[])       -> []
                  | (x::xs,[])    -> (x,x) :: (z xs ys)
                  | ([],y::ys)    -> (y,y) :: (z xs ys)
                  | (x::xs,y::ys) -> (x,y)::(z xs ys);;

(*    z : 'a list -> 'a list -> ('a * 'a) list   *)

(* z [x1; ...; xm] [y1; ...; yn] = [(x1,y1); (x2,y2); ...; (xm,ym); (y(m+1),y(m+1)); ... ; (yn,yn)]
                                   if n >= m
   z [x1; ...; xm] [y1; ...; yn] = [(x1,y1); (x2,y2); ...; (xn,yn); (x(n+1),x(n+1)); ... ; (xm,xm)]
                                   if n < m                                        *)

(*  z [1..3] [5.. 10] = [(1, 5); (2, 6); (3, 7); (8, 8); (9, 9); (10, 10)]
    z [5..10] [1.. 3] = [(5, 1); (6, 2); (7, 3); (8, 8); (9, 9); (10, 10)]   *)


(* 2.4 *)
let rec s xs ys = match (xs,ys) with
                  | ([],[]) -> []
                  | (xs,[]) -> xs
                  | ([],ys) -> ys
                  | (x::xs,y::ys) -> x::y::s xs ys;;

(*   s : 'a list -> 'a list -> 'a list   *)
(* s [x1; ...; xm] [y1; ...; yn] = [x1; y1; x2; y2; ...; xm; ym; y(m+1); ... ;
 yn]
                                   if n >= m
   s [x1; ...; xm] [y1; ...; yn] = [x1; y1; x2; y2; ...; xn; x(n+1); ... ; xm]
                                   if n < m
                                      *)

(*  s [1..3] [5.. 10] = [1; 5; 2; 6; 3; 7; 8; 9; 10]

    s [5..10] [1.. 3] = [5; 1; 6; 2; 7; 3; 8; 9; 10]
                        *)



(*  2.5  *)

let rec sC xs ys c = match (xs,ys) with
                     | ([],[]) -> c []
                     | (xs,[]) -> c xs
                     | ([],ys) -> c ys
                     | (x::xs,y::ys) -> sC xs ys (fun res -> c(x::y::res));;

let s1 xs ys = sC xs ys id;;

(* 3.1 *)

type Latex<'a> = | Section of string * 'a * Latex<'a>
                 | Subsection of string * 'a * Latex<'a>
                 | Label of string * Latex<'a> // from question 3.4
                 | Text of string * Latex<'a>
                 | Ref of string * Latex<'a>   // from question 3.4
                 | End;;


let text1 = Section ("Introduction", None,
              Text ("This is an introduction to ...",
                Subsection ("A subsection", None,
                  Text ("As laid out in the introduction we ...",
                    End))));;

(*  text1 : Latex<'a option>  *)

(* 3.2 *)

let rec addSecNumbersAux sec subsec =
    function
    | Section(s,_,doc)     -> Section(s, string (sec+1), addSecNumbersAux (sec+1)
       1 doc)
    | Subsection(s,_,doc)  -> Subsection(s, (string sec)+"."+(string subsec),
      addSecNumbersAux sec (subsec+1) doc)
    | Text(s,doc)          -> Text(s, addSecNumbersAux sec subsec doc)
    | doc                     -> doc;;

let addSecNumbers doc = addSecNumbersAux 0 1 doc;;

let text2 = Section ("Introduction", None,
              Text ("This is an introduction to ...",
                Subsection ("A subsection", None,
                  Text ("As laid out in the introduction we ...",
                    Subsection ("Yet a subsection", None,
                      Section ("And yet a section", None,
                        Subsection ("A subsection more...", None,
                          End)))))));;


(*  2.3  *)

(* addSecNumbers Latex<'a> -> Latex<string> *)

(*  2.4  *)
// Auxiliary function where argument currO keeps track of the current (sub)-section. It is an option type, where None signals that no section has bee defined yet.
// In case of multiple definitions of a label, the last one is chosen. The function can be refined to raise an exception in this case.
let rec bLE currO env =
    function
    | Section(s,sec,doc)            -> bLE (Some sec) env doc
    | Subsection(s,subsec ,doc)     -> bLE (Some subsec) env doc
    | Text(s,doc)                   -> bLE currO env doc
    | Label(lb,doc)                 -> match currO with
                                      | None    -> bLE currO (Map.add lb "?" env) doc
                                      | Some ss -> bLE currO (Map.add lb ss env) doc
    | Ref(lb, doc)                  -> bLE currO env doc
    | End                           -> env;;


let buildLabelEnv doc = let doc' = addSecNumbers doc
                        bLE None Map.empty doc;;


let text3 = Section ("Introduction", "1",
              Label("intro.sec",
                Text ("In section",
                  Ref ("subsec.sec",
                    Text (" we describe ...",
                      Subsection ("A subsection", "1.1",
                        Label("subsec.sec",
                          Text ("As laid out in the introduction, Section ",
                            Ref ("intro.sec",
                              Text (" we ...",
                                End))))))))));;


(*  3.5  *)

let nl : string = System.Environment.NewLine

// an auxiliary function having a label environment as argument.
// Each section and subsection start on a new line. Otherwise no formatting is performed.

let rec toS env =
    function
    | Section(s,sec,doc)       -> nl + sec+ " " + s + nl + toS env doc
    | Subsection(s,subsec,doc) -> nl + subsec+ " " + s + nl + toS env doc
    | Text(s,doc)              -> s + toS env doc
    | Label(lb,doc)            -> toS env doc
    | Ref(lb, doc)             -> " " + Map.find lb env + " " + toS env doc
    | End                      -> "";;

let toString doc = let doc' = addSecNumbers doc
                   let env  = bLE None Map.empty doc'
                   toS env doc';;


(*  4.1  *)

let mySeq = Seq.initInfinite (fun i -> if i % 2 = 0 then -i else i);;

(* the type of mySeq is seq<int> *)
(* the expression Seq.take 10 mySeq denote the finite sequence consisting of
   the 10 numbers: 0, 1, -2, 3, -4, ..., -8, 9
   Note that no elements in the sequence is acrtally computed until they as demanded.
   The interactive env. prints out the first few elements though.      *)

(*  4.2  *)

(* the sequence consisting of n, n + 2, n + 4, ... n + 2M can be generated, for example, by using sequence expressions or the library function Seq.init *)

let finSeq n M = seq { for i in seq [0..M] do
                          yield n+2*i     };;

let finSeq1 n M = Seq.init (M+1) (fun i -> n+2*i);;


(* 4.3  *)

type X = A of int | B of int | C of int * int;;
```

```

let rec zX xs ys = match (xs,ys) with
                   | (A a::aS,B b::bS) -> C(a,b) :: zX aS bS
                   | ([],[]) -> []
                   | _ -> failwith "Error";;

let rec uzX xs = match xs with
                 | C(a,b)::cS -> let (aS,bS) = uzX cS
                                 (A a::aS,B b::bS)
                 | [] -> ([],[])
                 | _ -> failwith "Error";;

(* the type of zX  is

    zX : X list -> X list -> X list

For two equal-length lists xs = [A a1; A a2; ... A an] and ys = [B b1; B b2; ... B bn]
it compute: zX xs ys = [C(a1,b1); ...; C(an,bn)]

The function raises and exception if
1. the two lists xs ys have different lengths,
2. an element in the list xs does not have the form A a, or
3. an element in the list ys does not have the form B b.

zX can be consider a kind of zip-function

*)

(* the type of uzX is

    uzX : X list -> X list * X list

and it is a kind of unzip function. When every element in the argument has the form C(a,b) the function is defined and computes:

   uzX  [C(a1,b1); ...; C(an,bn)] = ([A a1; A a2; ... A an], [B b1; B b2; ... B bn])

otherwise it is undefined.
*)



```