```
// Michael R. Hansen   29-11-2021


// Problem 3 from December 2013

type Title = string;;

type Section = Title * Elem list
and  Elem    = Par of string | Sub of Section;;

type Chapter = Title * Section list;;
type Book    = Chapter list;;

let sec11 = ("Background", [Par "bla"; Sub(("Why programming", [Par "Bla."]))]);;
let sec12 = ("An example", [Par "bla"; Sub(("Special features", [Par "Bla."]))]);;
let sec21 = ("Fundamental concepts",
              [Par "bla"; Sub(("Mathematical background", [Par "Bla."]))]);;
let sec22 = ("Operational semantics",
              [Sub(("Basics", [Par "Bla."])); Sub(("Applications", [Par "Bla."]))]);;
let sec23 = ("Further reading",  [Par "bla"]);;
let sec31 = ("Overview", [Par "bla"]);;
let sec32 = ("A simple example", [Par "bla"]);;
let sec33 = ("An advanced example", [Par "bla"]);;
let sec41 = ("Status", [Par "bla"]);;
let sec42 = ("What's next?", [Par "bla"]);;
let ch1 = ("Introduction", [sec11;sec12]);;
let ch2 = ("Basic Issues", [sec21;sec22;sec23]);;
let ch3 = ("Advanced Issues", [sec31;sec32;sec33]);;
let ch4 = ("Conclusion", [sec41;sec42]);;
let book1 = [ch1; ch2; ch3; ch4];;

// Q1
let rec maxL = function
              | []       -> 0
              | [x]      -> x
              | x::y::xs ->  maxL((max x y)::xs);;

// Q2
let rec overview = function
                  | []        -> []
                  | (t,_)::cs -> t :: overview cs;;

// Q3
let rec depthSection(_,es) = 1 + maxL(List.map depthElem es)

and depthElem = function | Par _ -> 0
```

```
                             | Sub s -> depthSection s;;

let depthChapter(_,ss) =  1 + maxL(List.map depthSection ss)

let depthBook(cs) = maxL(List.map depthChapter cs);;

type Numbering = int list;;
type Entry = Numbering * Title;;
type Toc = Entry list;;

// Q4
let rec tocB(cs) = tocChapters cs 1

and tocChapters cl n   = match cl with
                         | [] -> []
                         | (t,ss)::cs ->  ([n],t)::tocSections ss [n] 1 @ tocChapters cs (n+1)

and tocSections secs ns i = match secs with
                            | []    -> []
                            | s::ss -> tocSection s ns i @ tocSections ss ns (i+1)

and tocSection(t,es) ns i = let ns'=ns@[i]
                            (ns',t) :: tocElems es ns' 1

and tocElems es ns i = match es with
                       | []         -> []
                       | Par _::es  -> tocElems es ns i
                       | Sub s:: es -> tocSection s ns i @ tocElems es ns (i+1);;


let toc1 = tocB book1;;


// Problem 1 from May 2018

let rec f xs ys = match (xs,ys) with
                  | (x::xs1, y::ys1) -> x::y::f xs1 ys1
                  | _                -> [];;
// Q 1.1
(*
f [1;6;0;8] [0;7;3;3] evaluates to
1::0::f [6;0;8] [7;3;3] evaluates to
1::0::6::7::f [0;8] [3; 3] evaluates to
1::0::6::7::0::3::f [8] [3] evaluates to
1::0::6::7::0::3::8::3::f [] [] evaluates to
1::0::6::7::0::3::8::3::[] = [1;0;6;7;0;3;8;3]
```

```
*)

// Q 1.2
(*
The most general type of f is f: 'a list -> 'a list -> 'a list

f [x1; ...;xm] [y1; ...;yn] = [x1;y1;...;xk;yk] where k = min {m,n}
*)

// Q 1.3
(*
f is not tail recursive because the recursive call in the first match-clause
|  .... -> x::y::f xs1 ys1 is not in a tail call. When f xs1 ys1 returns
a value res, the expression x::y::res must still be computed.

A tail-recursive variant of f based on an accumulating parameter is below, where
f xs ys = fA xs ys []
*)
let rec fA xs ys acc =  match (xs,ys) with
                        | (x::xs1, y::ys1) -> fA xs1 ys1 (y::x::acc)
                        | _                -> List.rev acc;;

// Q 1.4
(*
A tail-recursive variant of f based on a continuation is given below, where
f xs ys = fA xs ys id
*)
let rec fC xs ys k = match (xs,ys) with
                     | (x::xs1, y::ys1) -> fC xs1 ys1 (fun res -> k(x::y::res))
                     | _                -> k [];;



// Problem 2.1 from May 2017

let rec f  = function
           | 0          -> [0]
           | i when i>0 -> i::g(i-1)
           | _          -> failwith "Negative argument"
and g = function
      | 0 -> []
      | n -> f(n-1);;

let h s k  = seq { for a in s do
                     yield k a };;


// Q 2.1
```

```
(*
f 5 = [5; 3; 1] as can by an evaluation
f 5 evaluates to ("curly arrow" should be used as in the textbook)
5::g 4 evaluates to
5::f 3 evaluates to
5::3::g 2 evaluates to
5::3::f 1 evaluates to
5::3::1::g 0 evaluates to
5::3::1::[]

the type of f is int -> int list

If i is negative the f i raises an exception
if i is positive and odd, then f i = [i; i-2; ....;1]
otherwise f i = [i; i-2; ....;0]

h (seq [1;2;3;4]) (fun i -> i+10) = seq [11; 12; 13; 14]

h has type seq<'a> -> ('a -> 'b) -> seq<'b> and

h sq k is the sequence obtained from sq by application of k to every element, that is, the value of
h sq k is the same as the value of Seq.map k sq.
*)


// Problem 3 from May 2016

type Container =  | Tank of float * float * float // (length, width, height)
                  | Ball of float               // radius
                  | Cylinder of float * float   // (radius, height)         // Q 3.4

// Q 3.1

let tank = Tank(3.0,4.0,5.0)
let ball = Ball 5.0

// Q 3.2

let wf = function
                | Tank(l,w,h) -> l>=0.0 && w>0.0 && h>0.0
                | Ball r      -> r>0.0
                | Cylinder(r,h) -> r>0.0 && h>0.0;;                     // Q 3.4


// Q 3.3
```

```
let volume = function
                       | Tank(l,w,h)   -> l*w*h
                       | Ball r        -> 4.0/3.0 *System.Math.PI * r*r*r
                       | Cylinder(r,h) -> System.Math.PI * r*r*h;;           // Q 3.4



type Name = string
type Contents = string
type Storage = Map<Name, Contents*Container>

// Q 3.5
let stg = Map.ofList [("tank1",("oil",tank)); ("ball1", ("water", ball))]


let find n st = match Map.tryFind n st with
                    | Some(cnt, c) -> (cnt, volume c)
                    | None         -> failwith (n + " is not a name of a container")



// Problem 4 from May 2016

type T<'a> = L | N of T<'a> * 'a * T<'a>

let rec f g t1 t2 = match (t1,t2) with
                       | (L,L) -> L
                       | (N(ta1,va,ta2), N(tb1,vb,tb2)) -> N(f g ta1 tb1, g (va,vb), f g ta2 tb2);;

let rec h t = match t with
                 | L             -> L
                 | N(t1, v, t2) -> N(h t2, v, h t1);;

let rec g =
   function
   | (_,L)                      -> None
   | (p, N(t1,a,t2)) when p a -> Some(t1,t2)
   | (p, N(t1,a,t2))          -> match g(p,t1) with
                                   | None -> g(p,t2)
                                   | res  -> res;;

let t = N(N(L, 1, N(N(L, 2, L), 1, L)), 3, L);;

// Q 4.1
// The type of t is T<int>, i.e. t: T<int>

```

```
// three values of type T<bool list>

let ta = L
let tb = N(ta, [false],ta);;
let tc = N(tb, [true;false],tb);;


// Q 4.2
(*
The most general type of f is ('a * 'b -> 'c) -> T<'a> -> T<'b> -> T<'c>

For a justification of this consider the expression f g t1 t2.
The type of f has the form: tg -> type1 -> type2 -> type3,
where g: tg, t1: type1, t2: type2 and (f g t1 t2): type3

1. From the match construction on (t1,t2) we observe that t1 and t2 are two trees with types, say type1=T<'a> and type2=T<'b>.
2. from g(va,vb) we see that va: 'a, vb: 'b  and hence the type of g has the form:
   tg = 'a * 'b -> 'c, where 'c is a new type variable
3. From expression in the second clause we see that the value of the expression must have the type
   type3 = T<'c>.
Since there are no further type constraints, we have f: ('a * 'b  -> 'c) -> T<'a> -> T<'b> -> T<'c>

The value of (f g t1 t2) is defined when t1 and t2 are two trees of the same shape
and the value of the expression is a tree t with the same shape as that of t1 and t2.
The value in a node n of t is g(v1,v2), there vi is the value in node of ti appearing
in the same position as n, for i=1,2. For example

if t1 has the form:
             N
      _______|______
      |      x     |
      N            N
  ____|___      ___|___
  .   y  .      .  z   .
  .      .      .      .

and t2 has the form:
             N
      _______|______
      |      o     |
      N            N
  ____|___      ___|___
```

```
 .    p .       . q    .
 .      .       .      .

then t has the form:
             N
      _______|______
      |     v1     |
      N            N
  ____|___      ___|___
 .   v2 .       . v3   .
 .      .       .      .
 where v1 = g(x,o), v2=g(y,p) and v3=g(z,q)
*)

(*
h has the type T<'a> -> T<'a> and the value of h(t) is the mirror image of t, in other words h t makes a reflection of t
-- it is natural to supply a suitable drawing as done for f.

g has type ('a -> bool) * T<'a> -> (T<'a>*T<'a>) option

g (p,t) makes a depth-first (left to right) traversal of t searching for a node N(left,a,right)
where the value a in the node satiefies predicate p, that is, p a = true.

If such node exists, then the value is Some(left,right); otherwise the value is None.
-- it is natural to supply a suitable drawing as done for f.
*)

// Q 4.3
let rec count a = function
                  | L  -> 0
                  | N(t1,v,t2) when v=a -> 1 + count a t1 + count a t2
                  | N(t1,_,t2)          -> count a t1 + count a t2;;

// Q 4.4
let rec replace a b = function
                      | L  -> L
                      | N(t1,v,t2) when a=v -> N(replace a b t1, b, replace a b t2)
                      | N(t1,v,t2)          -> N(replace a b t1, v, replace a b t2);;
```