Michael R. Hansen
DTU Compute, 29-11-2021

Problem 3 from December 2013 (Approx. 96 minutes)

We shall now consider *books* that are described by a list of *chapters*. Each chapter is described by a *title* and a list of *sections*. A section is described by a title and a list of *elements*, which can either be *paragraphs* (characterized by strings) or *sub-sections*. This is captured by the following type declarations:

```
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;;
```

The mutual recursion between sections and elements allows for arbitrary nesting of sub-sections. This is illustrated by the following examples:

```
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;sec34]);;
let ch4   = ("Conclusion", [sec41;sec42]);;
let book1 = [ch1; ch2; ch3; ch4];;
```

1. Declare a function `maxL` to find the largest integer occurring in a list with non-negative integers. The function must satisfy `maxL [] = 0`.

2. Declare a function `overview` to extract the list of titles of chapters from a book. For example, the overview for `book1` is:

```
overview book1 =
   ["Introduction"; "Basic Issues"; "Advanced Issues"; "Conclusion"]
```

Each chapter occurs at *depth* 1. A top-level section, i.e. a section which is not a sub-section, occurs at depth 2. A sub-section has a depth which is one larger than the depth of the section of which it is an immediate part. For example, the depth of the sub-section with title "Applications" in `book1` is 3 and the section with title "Overview" has depth 2.

3. Declare functions:

```
depthSection: Section -> int
depthElem:    Elem -> int
depthChapter: Chapter -> int
depthBook:    Book -> int
```

   to extract the maximal depth of sections, elements, chapters and books. For example the maximal depth of `book1` is 3, as `book1` has sub-sections, but no sub-sub-section.

We shall now make a *table of contents* (type `Toc` below) for a book. In a table of contents we use lists to number entries (see the types `Entry` and `Numbering` below). A *numbering* such as $[i; j; k; l]$ is the number of the l'th sub-sub-section, of the k'th sub-section of the j'th section in the i'th chapter. Notice that such lists have varying lengths. For example, [2] is the number of Chapter 2, i.e the chapter with title "Basic Issues" in the previous example, and [2;2;1] is the number of the sub-section with title "Basics" in Chapter 2.

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

The table of contents for the previous example is:

```
[([1], "Introduction");
 ([1; 1], "Background");
 ([1; 1; 1], "Why programming");
 ([1; 2], "An example");
 ([1; 2; 1], "Special features");
 ([2], "Basic Issues");
 ([2; 1], "Fundamental concepts");
 ([2; 1; 1], "Mathematical background");
 ([2; 2], "Operational semantics");
 ([2; 2; 1], "Basics");
 ([2; 2; 2], "Applications");
 ([2; 3], "Further reading");
 ([3], "Advanced Issues");
 ([3; 1], "Overview");
 ([3; 2], "A simple example");
 ([3; 3], "An advanced example");
 ([4], "Conclusion");
 ([4; 1], "Status");
 ([4; 2], "What's next?")]
```

4. Declare a function, `tocB: Book` $\rightarrow$ `Toc`, to compute the table of contents for a book.

Problem 1 from May 2018 (approx 48 minutes)

Consider the following F# declaration:

```
let rec f xs ys = match (xs,ys) with
                  | (x::xs1, y::ys1) -> x::y::f xs1 ys1
                  | _                -> [];;
```

1. Give an evaluation (using $\rightsquigarrow$) for `f [1;6;0;8] [0; 7; 3; 3]` thereby determining the value of this expression.
2. Give the (most general) type for `f`, and describe what `f` computes. Your description should focus on *what* it computes, rather than on individual computation steps.
3. The declaration of `f` is *not* tail recursive. Give a brief explanation of why this is the case and provide a declaration of a tail-recursive variant of `f` that is based on an accumulating parameter. Your tail-recursive declaration must be based on an explicit recursion.
4. Provide a declaration of a continuation-based, tail-recursive variant of `f`.

Problem 2.1 from May 2017 (approx 20 minutes)

Consider the following F# declarations:

```
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 };;
```

1. Give the values of `f 5` and `h (seq [1;2;3;4]) (fun i -> i+10)`. Furthermore, give the (most general) types for `f` and `h`, and describe what each of these two functions computes. Your description for each function should focus on *what* it computes, rather than on individual computation steps.

Problem 3 from May 16 (approx 48 minutes)

We shall now consider *containers* that can either have the form of a *tank*, that is characterized by it length, width and height, or the form of a *ball*, that is characterized by its radius. This is captured by the following declaration:

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

1. Declare two F# values of type `Container` for a tank and a ball, respectively.
2. A tank is called *well-formed* when its length, width and height are all positive and a ball is well-formed when its radius is positive. Declare a function `isWF` : `Container` → `bool` that can test whether a container is well-formed.
3. Declare a function `volume` c computing the volume of a container c. (Note that the volume of ball with radius r is $\frac{4}{3} \cdot \pi \cdot r^3$.)

A *cylinder* is characterized by its radius and height, where both must be positive float numbers.

4. Extend the declaration of the type `Container` so that it also captures cylinders, and extend the functions `isWF` and `volume` accordingly. (Note that the volume of cylinder with radius r and height h is $\pi \cdot r^2 \cdot h$.)

A *storage* consist of a collection of uniquely named containers, each having a certain *contents*, as modelled by the type declarations:

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

where the name and contents of containers are given as strings.

Note: You may choose to solve the below questions using a list-based model of a storage (`type Storage = (Name * (Contents*Container)) list`), but your solutions will, in that case, at most count 75%.

5. Declare a value of type `Storage`, containing a tank with name `"tank1"` and contents `"oil"` and a ball with name `"ball1"` and contents `"water"`.
6. Declare a function `find` : `Name` → `Storage` → `Contents` ∗ `float`, where `find` n stg should return the pair (cnt, vol) when cnt is the contents of a container with name n in storage stg, and vol is the volume of that container. A suitable exception must be raised when no container has name n in storage stg.

Problem 4 from May 16 (approx. 48 minutes)

Consider the following F# declarations of a type `T<'a>` for binary trees having values of type `'a` in nodes, three functions `f`, `h` and `g`, and a binary tree `t`:

```
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);;
```

1. Give the type of `t`. Furthermore, provide three values of type `T<bool list>`.
2. Give the (most general) types of `f`, `h` and `g` and describe what each of these three functions computes. Your description for each function should focus on *what* it computes, rather than on individual computation steps.
3. Declare a function `count` $a\,t$ that can count the number of occurrences of a in the binary tree t. For example, the number of occurrences of 1 in the tree `t` is 2.
4. Declare a function `replace`, so that `replace` $a\,b\,t$ is the tree obtained from t by replacement of every occurrence of a by b. For example, `replace 1 0 t` gives the tree `N(N(L, 0, N(N(L, 2, L), 0, L)), 3, L)`.