Written Examination, May 24th, 2023 Course no. 02157

The duration of the examination is 4 hours.

Course Name: Functional programming

Allowed aids: All written material

The problem set consists of 5 problems which are weighted approximately as follows: Problem 1: 15%, Problem 2: 15%, Problem 3: 15%, Problem 4: 25%, Problem 5: 30%

Marking: 7 step scale.

In your programs you are allowed to introduce helper functions; but you must also provide a declaration for each of the required functions, so that it has exactly the type and effect asked for.

You are, in general, allowed to use the .NET library including the modules described in the textbook, e.g., List, Set, Map, Seq, etc. But be aware of the special condition stated in Problem 1.

You are not allowed to use imperative features, like assignments, arrays and so on, in your solutions.

... Continued on next page

Problem 1 (15%)

All questions in this problem should be solved without using functions from the libraries `List`, `Seq`, `Set` and `Map`.

1. Declare a function `numberOf` x ys that returns the number of times x occurs in the list ys. For example, `numberOf 2 [0;2;3;3;0;2;4;2;1]` $= 3$.

2. Declare a function `positionsOf` x ys that returns the list containing the positions of occurrences of x in the list ys. For example, `positionsOf 2 [0;2;3;3;0;2;4;2;1]` $= [1; 5; 7]$. Notice that the position of the first element of a non-empty list is 0.

   Hint: You may consider introducing a helper function.

3. Declare a function `filterMap: ('a->bool) -> ('a->'b) -> 'a list -> 'b list`. The value of `filterMap` p f xs is the list obtained from xs by applying f to the elements that satisfy the predicate p.

   For example, `filterMap (fun x -> x>=2) string [0;2;3;3;0;2;4;2;1]` returns the list [”2”;”3”;”3”;”2”;”4”;”2”].

   Notice that `filterMap` p f xs could be defined by `List.map` f (`List.filter` p xs). Your task is here to declare `filterMap` using an explicit recursion.

Problem 2 (15%)

Consider the declaration:

```
let rec splitAt i xs =
   if i<=0 then ([],xs)
   else match xs with
        | []      -> ([],[])
        | x::tail -> let (xs1,xs2) = splitAt (i-1) tail
                     (x::xs1,xs2);;
```

1. What are the values of
   - `splitAt -1 [1;2;3]`,
   - `splitAt 3 [1;2;3;4;5]` and
   - `splitAt 4 [1;2;3]`.

2. What is the type of `splitAt`? Justify your answer briefly.

3. Describe what `splitAt` is computing by stating the value of

$$\texttt{splitAt}\ k\ [x_1; x_2; \ldots; x_{i-1}; x_i; \ldots; x_n], \text{ where } n \geq 0.$$

4. Is `splitAt` a tail-recursive function? Justify your answer briefly.

 ... Continued on next page

Problem 3 (15%)

In this problem you should declare three functions on sequences. You have a free choice of using sequence expressions or functions from the `Seq`-library. Some functions from the `Seq`-library are listed at the end of the problem for inspiration.

In descriptions of functions in this problem we let

- s denote the sequence seq $[x_0; x_1; \ldots; x_j; \ldots]$ and
- *pos* the sequence seq $[1; 2; 3; \ldots; \ldots]$ of positive natural numbers.

1. Declare a function `characteristicSeq: ('a->bool) -> seq<'a> -> seq<int>`. The value of `characteristicSeq` p s is the sequence obtained from s by transforming x_j to 1 if $p\ x_j =$ true and transforming x_j to 0 if $p\ x_j =$ false, for $j \geq 0$.

   For example, `characteristicSeq` (`fun n` $\rightarrow$ `x` > 2) *pos* = seq $[0; 0; 1; 1; 1; 1; 1; 1; 1; \ldots]$.

2. Declare a function `fracSeq: seq<int> -> seq<float>`. The value of `fracSeq` s is the sequence obtained from s by transforming x_j to $x_j/(j+1)$, for $j \geq 0$.

   For example, `fracSeq` *pos* = seq $[1.0; 1.0; 1.0; 1.0; \ldots]$.

   Hint: You may use the function `float` to convert an integer to a float value.

3. Declare a function `accSum: seq<int> -> seq<int>`. The value of `accSum` s, is the sequence seq $[y_0; y_1; \ldots; y_j; \ldots]$ where $y_j = \Sigma_{i=0}^{j-1} x_i$, for $j \geq 0$. (Notice that by definition $\Sigma_{i=0}^{-1} x_i = 0$.) Thus, `accSum` s returns the sequence containing the accumulated sums of the elements of argument s, starting with 0.

   For example, `accSeq` *pos* = seq $[0; 1; 3; 6; \ldots]$.

The following functions from the `Seq`-library could be candidates:

- `Seq.map: ('a -> 'b) -> seq<'a> -> seq<'b>`.
  The value of `Seq.map` f s is the sequence seq $[fx_0; fx_1; \ldots; fx_j; \ldots]$
- `Seq.mapi: (int -> 'a -> 'b) -> seq<'a> -> seq<'b>`.
  The value of `Seq.mapi` f s is the sequence seq $[f\ 0\ x_0; f\ 1\ x_1; \ldots; f\ j\ x_j; \ldots]$
- `Seq.scan: ('a->'b->'a) -> 'a -> seq<'b> -> seq<'a>`.
  The value of `Seq.scan` f e s is the sequence seq $[y_0; y_1; \ldots; y_j; \ldots]$,
  where

$$\begin{aligned} y_0 &= e \\ y_1 &= f\ y_0\ x_0 \\ y_2 &= f\ y_1\ x_1 \\ &\ldots \\ y_j &= f\ y_{j-1}\ x_{j-1} \\ &\ldots \end{aligned}$$

Problem 4 (25%)

A customer pays deposit when buying bottles and cans at a supermarket. When empty bottles and cans are returned for recycling or reuse, the associated deposit will be repaid. We call bottles and cans that are subjects to deposit *pieces*. See type `Piece` below.

```
type Volume   = float
type Piece    = A | B | C | Plastic of Volume | Glass of Volume
```

Some glass bottles and cans are used once and then *recycled* when returned. Such pieces are melted to raw material (glass or aluminium). Bottles and cans that are recycled are marked with either A, B and C. See corresponding constructors in the declaration of type `Piece`. Other glass and plastic bottles are *reused* when returned, that is, they are cleaned, refilled and distributed to supermarkets. The deposits of such pieces depend of their *volumes*. See constructors `Plastic` and `Glass` in the declaration of type `Piece`.

The deposits are described in the following tables. For example, the deposit for a recycled piece marked `B` is 1.5 kr. and the deposit for a 1 litre plastic bottle is 3 kr.

Deposit: Recyclable pieces

A	1.0 kr.
B	1.5 kr.
C	3.0 kr.

Deposit: Reusable pieces

`Plastic` v	1.0 kr.	if volume $v \leq 0.5$
`Plastic` v	3.0 kr.	if volume $v > 0.5$
`Glass` v	1.5 kr.	if volume $v < 1.0$
`Glass` v	3.0 kr.	if volume $v \geq 1.0$

A reusable plastic or glass piece is well-formed only if its volume is positive. Recyclable pieces are (trivially) well-formed.

1. Declare a function `isWellformed: Piece -> bool` that checks whether a given piece is well-formed.

2. Declare a function `isWellformedList: Piece list -> bool` that checks whether every piece in a given list is well-formed.

You can from now on assume that pieces are well-formed.

3. Declare a function `deposit: Piece -> float` that returns the deposit for a given piece.

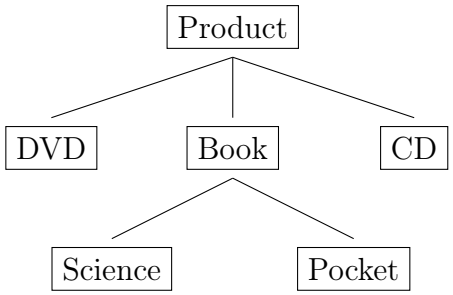
4. Declare a function `totalDeposit: Piece list -> float` that returns the total deposit for a given list of pieces.

5. Declare a function `toSummary` ps, where ps is a list of pieces, that returns a pair (nrc, nru) where nrc is the number of recyclable pieces in ps and nru is the number of reusable pieces in ps. State the type of `toSummary`.

 ... Continued on next page

Problem 5 (30%)

In this problem we shall consider a simple kind of *ontology* where *concepts* are *classified*. For example, the tree in the following figure shows an ontology with six concepts: 'Product', 'DVD', 'Book', 'CD', 'Science' and 'Pocket'.



The concept 'Product' in the root of the tree is classified as (can be) a 'DVD', a 'Book' or a 'CD'. A 'Book' is classified as a 'Science' book or a 'Pocket' book. There is no classification of the concepts 'DVD', 'CD', 'Science' and 'Pocket'. Such concepts are called *elementary*. We model ontologies and classifications using the following F# type declarations:

```
type Concept = string;;
type Ontology = O of Concept * Classification
and Classification = Ontology list;;
```

Thus, a concept is a string. When we speak about the ontology for a concept we mean a tree of type `Ontology` having this concept in the root. For example, the product ontology above contains a book ontology comprising the concepts `"Book"`, `"Science"` and `"Pocket"`. We assume that a given string occurs at most once in an ontology.

1. Declare a value `o1` of type `Ontology` corresponding to the above product ontology.
2. Declare a function `occursIn: Concept -> Ontology -> bool` that tests whether a given concept occurs in an ontology. For example, the concept `"Book"` occurs in the ontology `o1`, whereas the concept `"Autobiography"` does not occur in `o1`.
3. Declare a function `elementaryConcepts: Ontology -> Concept list` that gives a list containing the elementary concepts of a given ontology. For example, the value of `elementaryConcepts o1` should be a list containing `"DVD"`, `"CD"`, `"Science"` and `"Pocket"`. The sequence in which concepts occur in the result is of no importance.
4. Declare a function `find`, where the value of `find` c o should be `Some` o' if o' is the ontology for concept c in ontology o. The value is `None` if c does not occur in o.
5. What is the type of `find`? (This question can be answered even if 4. is not solved.)
6. A *path* in ontology o is a list of concepts met traversing o from the root to an elementary concept. For example, `["Product"; "Book"; "Pocket"]` is one out of 4 paths of `o1`.

   Declare a function `pathsOf` o that returns a list containing all paths of o.