Written Examination, May 23th, 2022 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 4 problems which are weighted approximately as follows:
Problem 1: 35%, Problem 2: 20%, Problem 3: 15%, Problem 4: 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 (35%)

A teacher named Robin has a bookshelf with book that are lent to colleagues and students. The following models of the shelf and the loans are introduced to keep track of books:

```
type Book  = string
type Shelf = Book list   // ordered alphabetically

type Date  = int
type Name  = string
type Loan  = Book * Name * Date
```

Books are just modelled by strings and we assume below that the books appear in alphabetic order in a shelf. (Built-in orderings `<`, `<=`, etc. can be used to compare books.) A shelf may contain multiple copies of the same book.

A loan is modelled by a triple (b, n, d), where b is a book, n the name of the borrower and d the date when the book was borrowed. Names are strings and dates are integers. Consider, for example, the following declarations of a shelf `sh0` with three books and a list `ls0` containing four loans.

```
let sh0 = ["Introduction to meta-mathematics";
           "To mock a mockingbird";
           "What is the name of this book"];;

let ls0 = [("Communication and concurrency", "Bob", 4);
           ("Programming in Haskell", "Paul", 2);;
           ("Communicating Sequential processes", "Mary", 7);
           ("Elements of the theory of computation", "Dick", 1)];;
```

The questions 1. to 6. in this problem should be solved without using functions from the libraries `List`, `Seq`, `Set` and `Map`. That is, the requested functions should be declared using explicit recursion.

In the declarations you can assume that books are ordered alphabetically in shelf arguments to functions. It is required that books are ordered alphabetically in shelves returned by functions.

1. Declare a function `onShelf: Book -> Shelf -> bool` that can check whether a book is on a shelf.
2. Declare a function `toShelf: Book -> Shelf -> Shelf` so that `toShelf` b bs is the shelf obtained from bs by insertion of b in the right position.
3. Declare a function `fromShelf: Book -> Shelf -> Shelf option`. The value of the expression `fromShelf` b bs is `None` if bs does not contain b. Otherwise, the value is `Some` bs', where bs' is obtained from bs by deletion of one occurrence of b.

 ... Continued on next page

4. Declare a function `addLoan` b n d ls, that adds the loan (b, n, d) to the list of loans ls.

   Furthermore, declare a function `removeLoan` b n ls. The value of the function is the list obtained from the list of loans ls by deletion of the first element of the form (b, n, d), where d is some date, if such an element exists. Otherwise ls is returned. For example, `removeLoan "Programming in Haskell" "Paul" ls0` gives the list

```
[("Communication and concurrency", "Bob", 4);
 ("Communicating Sequential processes", "Mary", 7);
 ("Elements of the theory of computation", "Dick", 1)]|
```

5. Declare a function `reminders: Date -> Loan list -> (Name * Book) list`. The value of `reminders` d_0 ls is a list of pairs (n, b) from loans (b, n, d) in ls where $d < d_0$. We interpret $d < d_0$ as "date d is before date d_0".

   For example, `reminders 3 ls0` has two elements: `("Paul","Programming in Haskell")` and `("Dick", "Elements of the theory of computation")`.

6. In this problem, we consider a textual form of the reminders from Question 5, where, for example, a letter reminding Paul to return "Programming in Haskell" has the form:

```
"Dear Paul!
 Please return "Programming in Haskell".
 Regards Robin"
```

   Declare a function `toLetters: (Name * Book) list -> string list`, that transforms a list pairs (n, b) to a list of corresponding strings (letters). Notice, the escape characters `\n` and `\"` denote newline and citation quotation, respectively.

7. This question should be solved using functions from the `List` library. You should *not* use explicit recursion in the declarations.

   1. Give an alternative declaration for `toLetters` using `List.map`.
   2. Give an alternative declaration for `reminders` using `List.foldBack`.

 ... Continued on next page

Problem 2 (20%)

The functions `skipWhile` and `takeWhile` from the `List` library could have the following declarations:

```
let rec skipWhile p = function
                    | x::xs when p x -> skipWhile p xs
                    | xs             -> xs;;
val skipWhile: ('a -> bool) -> 'a list -> 'a list

let rec takeWhile p = function
                    | x::xs when p x -> x::takeWhile p xs
                    | _              -> [];;
val takeWhile: ('a -> bool) -> 'a list -> 'a list
```

Notice that the F# system automatically infers the types of these functions.

1. Give an argument showing that `('a -> bool) -> 'a list -> 'a list` is the most general type of `takeWhile`. That is, any other type for `takeWhile` is an instance of `('a -> bool) -> 'a list -> 'a list`.

Let `diff5` be declared by:

```
let diff5 n = n<>5;;
```

2. Give an evaluation of the expression `skipWhile diff5 [2;6;5;1;5;6]`. Use the notation $e_1 \rightsquigarrow e_2$ from the textbook and include at least as many steps as there are recursive calls.
3. Describe what `takeWhile` and `skipWhile` compute. Your descriptions should focus on *what* they compute, rather than on individual computation steps.
4. Consider each of the above declarations and explain briefly whether the considered function is tail recursive or not. If you encounter a function that is not tail recursive, then provide a declaration of a tail-recursive variant with an accumulating parameter for that function.

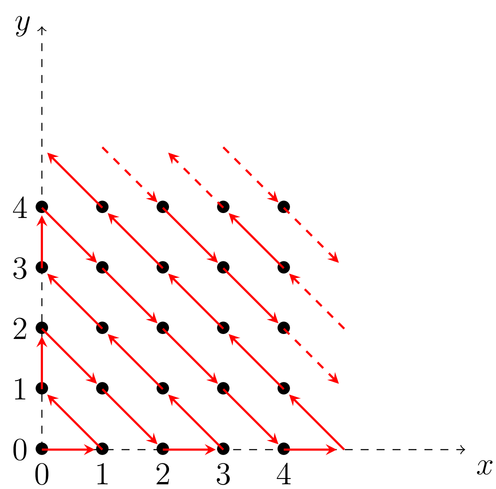
Problem 3 (15%)

1. Declare a function `flip: seq<'a*'b> -> seq<'b*'a>`. The function `flip` transforms a sequence $(a_0, b_0), (a_1, b_1), \ldots, (a_i, b_i), \ldots$ to the sequence $(b_0, a_0), (b_1, a_1), \ldots, (b_i, a_i), \ldots$.

2. Declare a function `dia` n, where n is a non-negative integer, that generates the sequence of pairs $(0, n), (1, n-1), \ldots, (n-1, 1), (n, 0)$. For example, `dia 0` is a sequence containing just $(0, 0)$, `dia 2` is the sequence $(0, 2), (1, 1), (2, 0)$ and `dia 3` is the sequence $(0, 3), (1, 2), (2, 1), (3, 0)$.

The following figure illustrates a traversal of all integer coordinates in the first quadrant. Following the red arrows, we see that the sequence of coordinates starts with $(0, 0), (1, 0), (0, 1), (0, 2), (1, 1), (2, 0), (3, 0), (2, 1), (1, 2), (0, 3), (0, 4), \ldots$.

This infinite sequence is named `allCoordinates`.



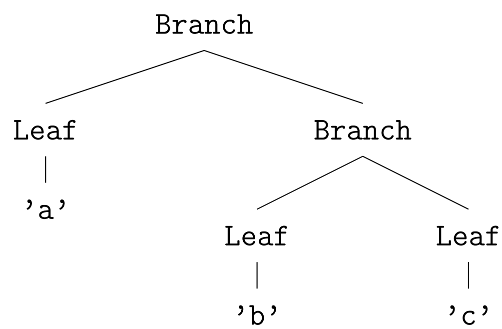
3. Give a declaration of `allCoordinates`.
Hint: You may use `dia` and `flip` as helper functions, even if you did not provide declarations for these functions.

Problem 4 (30%)

Consider now binary trees where leaf nodes (constructor `Leaf`) carry characters:

```
type T = Leaf of char | Branch of T*T
```

The figure below shows a tree `t0` of type `T` containing three characters: `'a'`, `'b'` and `'c'`.



A tree t is called *legal* if any character occurs at most once in t and t contains at least 2 characters. Thus, `t0` is a legal tree.

1. Make an F# value for the tree `t0` shown above and declare a function

   ```
   toList: T -> char list
   ```

   that gives the list of characters occurring in a tree. The sequence in which the characters occur in the list is of no significance.

2. Declare a function `legal` t that can check whether a tree t is legal.

We assume from now on that trees are legal and consider the so-called Huffman coding for characters in a given tree t, where a code $ds = [d_1; d_2; \ldots; d_n]$ (type `Code`) is a list of directions denoting a path from the root to a leaf in t.

```
type Dir  = | L        // go left
            | R        // go right
type Code = Dir list
type CodingTable = Map<char, Code>
```

For example, the codes for `'a'`, `'b'` and `'c'` in `t0` are `[L]` `[R;L]` `[R;R]`, respectively.

Furthermore, a *coding table* (for a given tree) is a map from characters to their codes. The coding table for `t0`, for example, has the entries (`'a'`, `[L]`), (`'b'`, `[R;L]`) and (`'c'`, `[R;R]`).

The code for a list of characters $cs = [c_1; \ldots; c_m]$, given a coding table, is obtained by appending the codes for the individual characters of cs. For example, the code for `['c';'a';'a';'b']` is `[R;R;L;L;R;L]`.

 ... Continued on next page

3. Declare a function `encode: CodingTable -> char list -> Code` that gives the code for a list of characters for a given coding table. The function should raise an exception if the coding table does not contain a code for some character in the list.

4. Declare a function `ofT: T -> CodingTable` that gives the coding table for a tree.

We now consider a function to reproduce the character list *cs* from a code *ds* on the basis of the underlying tree *t*. This function is called *decode*:

```
decode: T -> Code -> char list
```

For example, `decode t0 [R;R;L;L;R;L]` = `['c';'a';'a';'b']`.

It is convenient to use a helper function

```
firstCharOf: T -> Code -> char * Code
```

in the declaration of `decode`.

This helper function decodes the first character of the code and returns that character and the remaining code. For example,

```
firstCharOf t0 [R;R;L;L;R;L] = ('c', [L;L;R;L])
firstCharOf t0     [L;L;R;L] = ('a', [L;R;L])
firstCharOf t0       [L;R;L] = ('a', [R;L])
firstCharOf t0         [R;L] = ('b', [])
```

5. Give declarations for the functions `firstCharOf` and `decode`.