Written Examination, December 18th, 2019 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: 30%, Problem 2: 15%, Problem 3: 30%, Problem 4: 25%

Marking: 7 step scale.

Do not use imperative features, like assignments, arrays and so on, in your solutions.

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 restriction in Problem 1.

You are allowed to use functions from the textbook. If you use such a function, then provide a reference to the place where it appears in the textbook.

... Continued on next page

Problem 1 (30%)

The five questions of this problem should be solved without the use of functions from the `List` and `Seq` libraries. If you use such functions in a solution to a question, then you will get no credit for that solution.

A *menu* in a restaurant comprises a list of *identified courses*, where each course is described by a *name* and a *price*. This is modelled by:

```
type Price  = int
type Name   = string
type Course = Name * Price

type Identifier = int
type Menu       = (Identifier * Course) list
```

It can be assumed that each course has its own identifier. The following is an example of a simple menu:

```
[(1, ("Salad", 35));  (2, ("Soup", 30)); (3, ("Salmon", 120));
 (4, ("Chicken", 60)); (5, ("Spicy Beans", 70)); (6, ("Lamb", 115))];
```

1. Declare a function `findCourse: Identifier -> Menu -> Course`, so that the value of `findCourse` i m is the course identified by i in the menu m. If no course is identified by i in m, then the function should terminate with an exception mentioning i.

An *order* is given by a list of identifiers:

```
type Order = Identifier list
```

For the above menu, the list `[1;4;2;1;6]` is an order o of two salads, one chicken, one soup and one lamb.

2. Declare a function `priceOf: Order*Menu -> int`, which gives the total price of an order for a given menu.

By a *counting* we understand a list of the form $[(i_1, c_1); (i_2, c_2); \ldots; (i_k, c_k)]$, where i_j is an identifier for a course and c_j is a count for the number of occurrences (i.e. orders) of course i_j, for $1 \leq j \leq k$. The identifiers occurring in a counting are all different and the order of the pairs in a counting is of no importance. The counting $[(1, 2); (4, 1); (2, 1); (6, 1)]$ corresponds to the above-mentioned order o.

```
type Count    = int
type Counting = (Identifier * Count) list
```

... Continued on next page

3. Declare a function `increment`(i, cnt), where i is an identifier and cnt is a counting. The value of `increment`(i, cnt) is the counting obtained from cnt by incrementing the count for i by one.

   For example, `increment(6,[(1,2);(4,1);(2,1);(6,1)])` is a counting containing four pairs: (1,2), (4,1), (2,1) and (6,2) and `increment(3,[(1,2);(4,1);(2,1);(6,1)])` is a counting containing five pairs: (1,2), (4,1), (2,1), (6,1) and (3,1).

4. Declare a function `toCounting: Order -> Counting`, that makes a counting from an order.

An *overview* is a list of tuples: (i, n, c, p), where i is a course identifier, n is a course name, c is the count of course i, and p is the total price for the ordered number of course i. Overviews are modelled by:

```
type Overview = (Identifier * Name * Count * Price) list
```

5. Declare a function `makeOverview: Counting*Menu -> Overview`, which gives an overview for given counting and menu.

Problem 2 (15%)

Consider the following F# declaration:

```
let rec f a xs g = match xs with
                   | []    -> a
                   | x::xr -> f (g a x) xr g;;
```

1. Give the type of `f`.
2. Give a step-by-step evaluation (using $\rightsquigarrow$) for the expression `f` a $[x_0; x_1; x_2]$ g, where
   - there should at least be one step for every recursive call of `f` and
   - the last expression of the evaluation does not contain `f`.
3. Give three values v_1, v_2 and v_3 so that the expression `f` v_1 v_2 v_3 has type `int*string`.
4. Declare a function `h` x y that gives the infinite sequence with elements $x \cdot y - i \cdot (x - y)$, where $i = 0, 1, 2, 3, \ldots$. The identifier i is the index of the elements of the sequence. Give the type of `h`.

... Continued on next page

Problem 3 (30%)

We now consider the types `Exp` and `Pat`:

```
type Exp = | Const of int
           | Var of string
           | Add of Exp * Exp

type Pat = | PConst of int
           | PVar of string
           | PAdd of Pat * Pat
```

Values of type `Exp` are called *expressions* and values of type `Pat` are called *patterns*. Expressions are constructed from constants (constructor `Const`) and variables (constructor `Var`) using a constructor `Add` to form additions of two expressions. Patterns are constructed in a similar fashion.

1. Declare a function `vars: Pat -> string list` that gives the list of those strings s that appear as `PVar` s in a pattern.
2. A pattern p is *illegal* if the same string s occurs multiple times as `PVar` s in p. Declare a function `legal: Pat -> bool` that gives value `false` for illegal patterns and otherwise the value `true`.

The types `Binding` and `BindingList` are declared as follows:

```
type Binding     = string*Exp
type bindingList = Binding list
```

A pattern p can *match* an expression e, and, if this is the case, a list of bindings is formed according to the following rules:

a. Pattern `PConst` n_1 matches expression `Const` n_2 when $n_1 = n_2$ and no binding is formed.

b. Pattern `PVar` s matches any expression e forming the list $[(s, e)]$ containing one binding.

c. Pattern `PAdd`(p_1, p_2) matches expression `Add`(e_1, e_2) when p_1 matches e_1 and p_2 matches e_2, and the list of bindings formed consists of all bindings formed by matching p_1 with e_1 and p_2 with e_2.

d. Any matching of a pattern with an expression can be obtained by repeated use of the above rules $a., b.$ and $c.$

... Continued on next page

3. Declare a function

   ```
   patMatch: Pat*Exp -> BindingList option
   ```

   where `patMatch`(p, e) = `None` if p does not match e and `patMatch`(p, e) = `Some` bs if p matches e and bs is a list of bindings formed when matching p with e.

We shall now consider a revision of the types `Pat` and `Exp` in order to support infix operators in general and not just addition. In particular, we would like to replace `Add` by a new constructor `InfixOp` so that `InfixOp(InfixOp(Const 2,"+",Var "x"),"-",Var "y")`, for example, becomes a value of the revised expression type.

Similarly, `PAdd` should be replaced by a new constructor `PInfixOp` so that, for example, `PInfixOp(PInfixOp(PConst 2,"+",PVar "x"),"-",PVar "y")` becomes a value of the revised pattern type.

Furthermore, the matching rule relating to the new constructors is:

c'. Pattern `PInfixOp`(p_1, o, p_2) matches expression `InfixOp`(e_1, o', e_2) when $o = o'$, p_1 matches e_1 and p_2 matches e_2, and the list of bindings formed consists of all bindings formed by matching p_1 with e_1 and p_2 with e_2.

The following two questions concern some of the revisions needed to accommodate infix operators:

4. Show the revised type declarations for `Pat` and `Exp`.

5. Show your revised declaration for `patMatch`.

... Continued on next page

Problem 4 (25%)

Consider the following F# declaration:

```
let rec f g (h1,h2) = function
                    | []             -> []
                    | x::xs when g x -> h1 x :: f g (h1,h2) xs
                    | x::xs          -> h2 x :: f g (h1,h2) xs;;
```

1. Give the type for `f` and explain the value of the expression: `f` $g\ (h_1, h_2)\ [x_0; ...; x_{n-1}]$.
2. Give another declaration of `f` that is based on a single higher-order function from the `List` library. The new declaration of `f` should not be recursive.

Consider now the following declarations:

```
type A<'a> = | D of 'a * bool
             | E of A<'a> * A<'a>

let rec g acc x = match x with
                  | E(y,z)    -> g (g acc z) y
                  | D(a,true) -> a::acc
                  | _         -> acc;;

let h x = g [] x;;
```

3. Give 3 different values of type `A<string list>` using all constructors.
4. Determine the types of `g` and `h` and describe what h computes. Your description should focus on what it computes, rather than on individual computation steps.
5. Is `g` a tail-recursive function? Your answer must be accompanied with an explanation.
6. Provide declarations of continuation-based, tail-recursive variants of both `g` and `h`.