Written Examination, December 17th, 2024 Course no. 02157

The duration of the examination is 4 hours.

Course Name: Functional programming

Allowed aids: No Aid

The problem set consists of 3 problems which are weighted approximately as follows: Problem 1: 35%, Problem 2: 30%, Problem 3: 35%

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 Questions 1.1, 1.2 and 1.3 of 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%)

In this problem we consider a simple shop for bikes and bike accessories, called *articles*. The shop maintains a *register* (type `Register`) containing descriptions of articles: $[(a_1, d_1); \ldots; (a_n, d_n)]$, where a_i is an *identifier* (type `Id`) for an article and d_i is the associated *description* (type `Desc`). You can assume that the identifiers $a_1, \ldots, a_n$ are distinct. An article a is described by a triple (n, p, es), where n is the name of the article, p the price and es (type `Extra`) describes the possible *extra articles* that fit with article a. A *purchase* (type `Purchase`) is described by a list of identifiers of articles.

This is modelled by the following type declarations:

```
type Id       = string
type Name     = string
type Price    = int
type Extra    = Id list
type Desc     = Name * Price * Extra

type Register = (Id * Desc) list
type Purchase = Id list

let reg = [("a1", ("bell",1,[]));   ("a2", ("light",7,[]));
           ("a3", ("bag v1",4,[])); ("a4", ("bag v2",6,[]));
           ("a5", ("rear rack v1",4,["a3";"a4"]));
           ("a6", ("rear rack v2",5,["a3";"a4"]));
           ("a7", ("bike 1",100,["a1";"a2";"a5"]));
           ("a8", ("bike 2",150,["a1";"a2";"a6"])) ]
```

A register `reg` containing descriptions of eight articles is also declared above. The element `("a8", ("bike 2",150,["a1";"a2";"a6"]))` in `reg` describes a bike with name `"bike 2"` and price 150. Three articles in the register fit on that bike: the bell `"a1"`, the light `"a2"` and the rear rack identified by `"a6"` in the register. Both bags in the register, identified by `"a3"` and `"a4"`, respectively, fit on that rear rack (`"rear rack v2"`).

The Questions 1., 2. and 3. below must be solved without using functions from the `List` library.

1. Declare a function `find: Register -> Id -> Desc`, so that `find` *reg a* is the description associated with a in *reg*. A suitable exception should be raised if there is no description associated with a.

2. Declare a function `price: Register -> Purchase -> int` that computes the total price of a purchase for a given register. For example, the total price of `["a1";"a2";"a8"]` is 158 for the register `reg`.

3. Declare a function `extraOf` *reg a* that extracts the list of extra articles from the description of a in *reg*. For example, `extraOf reg "a8" = ["a1";"a2";"a6"]`.

 ... Continued on next page

A *bill* for a purchase consists of a list of pairs (n, p) comprising name n and price p of each bought article together with a total price. The order in which pairs (n, p) occur in a bill is not important. We use the following type for bills:

```
type Bill = (Name * Price) list * Price
```

For example, a bill for the purchase `["a1";"a2";"a8"]` could be

```
([("bell", 1); ("light", 7); ("bike 2", 150)], 158)
```

using our example register.

4. Declare a function `makeBill: Register -> Purchase -> Bill` that computes a bill for given purchase and register.

We now focus on *accessories* for articles (represented by identifiers) in a register. Let a be an identifier for an article having $[a_1; \ldots; a_n]$ as extra articles. Then, for $i = 1 \ldots, n$:

- a_i is an accessory of a and
- every accessory a_i' of a_i is also an accessory of a.

Consider our example register `reg`. Some examples:

- Articles `"a1"` (bell), `"a2"` (light), `"a3"` (bag v1), `"a4"` (bag v2) have no accessory, as their lists of extra articles are empty.
- Article `"a6"` (rear rack v2) has `"a3"` and `"a4"` as accessories because they are in the list of extra articles for `"a6"`. Article `"a6"` has no further accessories as `"a3"` and `"a4"` have no accessory.
- The accessories of `"a8"` (bike 2) are `"a1"`, `"a2"`, `"a6"`, `"a3"` and `"a4"`.

5. Declare a function `accessories` *reg a* that computes the accessories of article a in register *reg*. Hint: You may declare `accessories` in mutual recursion with a helper function having the type: `Register -> Id list -> Id list`.
6. Does your function `accessories` *reg a* terminate for every *reg* and a?
   - A "yes" answer must be accompanied with a justification.
   - A "no" answer must be accompanied with example values reg_0 and a_0 for which evaluation of `accessories` reg_0 a_0 will not terminate.

 ... Continued on next page

Problem 2 (30%)

Consider the following declaration of a function `f`:

```
let rec f x v ys =
   match ys with
   | []                    -> [(x,v)]
   | (y,_)::tail when x=y -> (y,v)::tail
   | pair::tail            -> pair::f x v tail;;
```

1. State the most general type of `f`. (Notice that any other type of `f` is an instance of the most general type.) Furthermore, give a justification of your answer to `f`'s type.

2. Make an evaluation for the expression

   ```
   f "z" 5 [("x",3); ("y",1); ("z",2); ("v",1)]
   ```

   Make use of the $\rightsquigarrow$ notation. Furthermore, the evaluation must have at least one step for each recursive call.

3. Describe what `f` is computing. Your description should focus on what the function is computing rather than on how the computations are performed.

   Furthermore, suggest an appropriate name for `f` that reflects what it is computing.

4. The declaration of `f` is not tail recursive. Explain briefly why and make a tail-recursive variant of `f` that is based on an accumulating parameter.

5. Make a continuation-based tail-recursive variant of `f`.

6. Consider now the following declaration of a function `g`:

   ```
   let rec g p xvs =
      match xvs with
      | []                    -> 0
      | (x,v)::tail when p x -> v + g p tail
      | _::tail               -> g p tail;;
   ```

   6(a). State the (most general) type of `g`.

   6(b). Make an alternative non-recursive declaration of `g` using `List.foldBack` by completing the following schema:

   ```
   let g p xvs = List.foldBack  ...  ...  ...
   ```

   Notice that `List.foldBack: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`.

 ... Continued on next page

Problem 3 (35%)

Let an *occurrence count* be a possibly empty list of pairs: $occ = [(s_1, n_1); \ldots; (s_k, n_k)]$, where the *occurrence-count conditions* hold: (1) n_i is a positive natural number, for $1 \leq i \leq k$, and (2) $s_1, \ldots, s_k$ are all different.

A pair (s_i, n_i) in *occ* reads: "s_i occurs n_i times in *occ*". If s is not in $\{s_1, \ldots, s_k\}$, then we say that s occurs zero times in *occ*.

We use the following type for occurrence count:

```
type OccCount<'a when 'a: equality> = ('a * int) list;;
```

1. Declare a function: `countOf: OccCount<'a> -> 'a -> int` so that `countOf` *occ* s is the number of occurrences of s in *occ*.

We consider now a polymorphic tree type `T<'a>` with arbitrary branching where nodes carry values of type `'a`:

```
type T<'a> = N of 'a * T<'a> list;;
```

2. Declare a function `map: ('a -> 'b) -> T<'a> -> T<'b>` so that `map` f t is the tree obtained from t by replacing every node value v of type `'a` by $f(v)$.

3. Declare a function `tryFind: ('a -> bool) -> T<'a> -> T<'a> option`. The call `tryFind` p t will cause a search for a sub-tree `N`(v', ts') of t where the value v' in the node satisfies the predicate p, that is $p(v')$ holds. If this search succeeds, then `Some` t_s is returned, where t_s is some sub-tree where the value in the (root) node of t_s satisfies p. Otherwise, `None` is returned.

   Note that several node values in t may satisfy p. In this case an arbitrary sub-tree where the node value satisfies p can be returned.

By an *occurrence tree* we understand a tree of type `T<'a*int>`:

```
type OccTree<'a> = T<'a * int>;;
```

where node values now are pairs (s, n), where n is an integer, with the intended meaning: "s occurs n times".

4. Declare a function `toOccTree: OccCount<'a> -> T<'a> -> OccTree<'a>`, so that the value of `toOccTree` *occ* t is the occurrence tree obtained from t by replacing every node value s by (s, n), where n is the number of occurrences of s in *occ*.

The type `T<string>` will now be used to represent a classification of *birds of prey*. Birds of prey may be partitioned into different families. The families hawk and falcon are two examples. A family may be further partitioned into sub-families. The hawk family can be partitioned into the families: kite, eagle and buzzard. Individual species (like common kestrel and red kite) are at the end of the classification.

```
BirdOfPrey                    | let Buzzard =
  Falcon                      |    N("Buzzard",
    common kestrel            |      [N("common buzzard",[]);
    peregrine falcon          |       N("rough-legged buzzard",[])]);;
  Hawk                        | let Eagle = N("Eagle",[N("golden eagle",[]);
    Kite                      |                        N("sea eagle",[])]);;
      black kite              | let Kite = N("Kite", [N("black kite",[]);
      red kite                |                       N("red kite",[])]);;
    Eagle                     | let Hawk = N("Hawk", [Kite;Eagle;Buzzard]);;
      golden eagle            | let Falcon = N("Falcon",
      sea eagle               |                [N("common kestrel",[]);
    Buzzard                   |                 N("peregrine falcon",[])]);;
      common buzzard          |
      rough-legged buzzard    | let BirdOfPrey = N("BirdOfPrey",[Falcon;Hawk]);;
```

Figure 1: Birds of prey: Textual representation and F# value

In Figure 1, the right-hand side provides an F# example for a classification of birds of prey. The left-hand side is a corresponding textual representation, where indentation is used to exhibit the classification hierarchy. Notice that family names start will capital letters while names of individual species start with small letters in the examples.

We shall use occurrence counts to record observations of birds - individual species as well as families - in a region. An example *observation* is:

```
let obs = [("BirdOfPrey",1); ("golden eagle",1); ("red kite",2);
           ("Buzzard",2); ("common kestrel",4)];;
```

5. Consider `let obsTree =` e_1. Define the expression e_1 so that `obsTree` becomes the occurrence tree for `BirdOfPrey` that is based on `obs`. The textual representation of `obsTree` is shown in Figure 2a on the next page. Three node values occurring in `obsTree` are `("BirdOfPrey",1)`, `("Kite",0)` and `("red kite",2)`.

The *accumulated occurrence tree* (abbreviated *aot*) of an occurrence tree $\mathtt{N}((s,n),[t_1;\ldots;t_k])$, where $t_i = \mathtt{N}((s_i,n_i),ts_i)$, is defined as follows:

- If $t'_i = \mathtt{N}((s_i,n'_i),ts'_i)$ is the aot of t_i, for $1 \leq i \leq k$,
  then $\mathtt{N}((s,n+\Sigma_{i=1}^{k} n'_i),[t'_1;\ldots;t'_k])$ is the aot of $\mathtt{N}((s,n),[t_1;\ldots;t_k])$.

Notice that the base case is contained in above definition when $k = 0$: The aot of a leaf $\mathtt{N}((s,n),[\,])$ is $\mathtt{N}((s,n),[\,])$. See example on the next page.

 ... Continued on next page

```
(BirdOfPrey, 1)
   (Falcon, 0)
      (common kestrel, 4)
      (peregrine falcon, 0)
   (Hawk, 0)
      (Kite, 0)
         (black kite, 0)
         (red kite, 2)
      (Eagle, 0)
         (golden eagle, 1)
         (sea eagle, 0)
      (Buzzard, 2)
         (common buzzard, 0)
         (rough-legged buzzard, 0)
```

(a) The occurrence tree: `obsTree`

```
(BirdOfPrey, 10)
   (Falcon, 4)
      (common kestrel, 4)
      (peregrine falcon, 0)
   (Hawk, 5)
      (Kite, 2)
         (black kite, 0)
         (red kite, 2)
      (Eagle, 1)
         (golden eagle, 1)
         (sea eagle, 0)
      (Buzzard, 2)
         (common buzzard, 0)
         (rough-legged buzzard, 0)
```

(b) The aot based on `obsTree`

Figure 2: An occurrence tree and the corresponding accumulated occurrence tree

The aot based on `obsTree` is shown in Figure 2b. Three node values occurring in that tree are `("BirdOfPrey",10)`, `("Kite",2)` and `("red kite",2)`. The count for a family, like "Hawk", in an aot is the total number of birds belonging to the family according to the used observation.

6. Declare a function `toAccOccTree` *ot* computing the accumulated occurrence tree of occurrence tree *ot*.

7. Give an expression computing from `obsTree` the number of birds from the hawk family using functions from this problem.
   Hint: You may use `tryFind` and `toAccOccTree` even in that case where you did not provide declarations for these functions.