Written Examination, May 23th, 2025 Course no. 02157

The duration of the examination is 4 hours.

Course Name: Functional programming

Allowed aids: No Aid

The problem set consists of 5 problems which are weighted approximately as follows: Problem 1: 25%, Problem 2: 25%, Problem 3: 13%, Problem 4: 12%, Problem 5: 25%

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, 1.3, 1.4 and 1.5 of Problem 1.

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

Problem 1 (25%)

Questions 1 to 5 in this problem should be solved without using functions from the libraries `List`, `Seq`, `Set` and `Map`.

In this problem we consider meals and some of their properties, in particular, the protein content and the carbon (CO_2) footprint. We start with a simple model of a *meal* (type `Meal`), that is a list of pairs $[(n_1, w_1); \ldots; (n_n, w_n)]$, where n_i is a food (type `Food`) and w_i is the associated weight (type `Weight`) given in grams. You can assume that the n_i's are all different:

```
type Food   = string
type Weight = float                 // weigth in grams

type Meal   = (Food * Weight) list

let m0 = [("Lettuce", 100.0); ("Soy beans", 100.0); ("Eggs", 150.0)];;
let m1 = [("Rice", 80.0); ("Soy beans", 100.0); ("Chicken", 125.0)];;
```

The declarations above also contain declarations of two meals.

1. Declare a function `totalWeight: Meal -> Weight` that computes the total weight of the ingredients of a meal. For example, `totalWeight m0` $= 350.0$.

Food in meals is characterized by a pair of two ratios (pwr, cwr), where pwr (type `PWR`) is the protein weight in 100 grams of a food and cwr (type `CWR`) is the carbon footprint of 100 grams of a food. The carbon footprint is given in grams. A *food table* (type `FoodTable`) associates a pair of ratios with food. A food has at most one occurrence in a food table:

```
type PWR = float        // protein weight in grams per 100 grams food
type CWR = float        // CO2 footprint in grams per 100 grams food
type FoodTable = (Food * (PWR * CWR)) list

let ft0 = [("Rice", (9.0,4.9)); ("Soy beans", (10.7,6.1));
           ("Lettuce", (1.3,1.4)); ("Eggs", (12.3,2.0));
           ("Chicken", (19.3,3.0))];;
```

For example, in food table `ft0` above we see that 100 grams of egg contain 12.3 grams protein and "contribute" with 2 grams CO_2.

2. Declare a function `find` n ft that can find the pair of ratios associated with food n in food table ft. A suitable exception should be raised if no ratio pair is associated with n. State the type of your function.

The *profile* (type `Profile`) of w grams of a food n is a triple of three weights (tw, pw, cw), all given in grams, where $tw = w$, pw is the protein weight and cw is the carbon weight. A meal profile (type `MealProfile`) is a list associating a profile with every food in a meal:

```
type Profile     = Weight * Weight * Weight
type MealProfile = (Food * Profile) list
```

For example, the profile of 150 grams of egg, given `ft0`, is $(150.0, 18.45, 3.0)$ and the meal profile of `m0` given `ft0` is $mp0 =$

```
[("Lettuce",   (100.0,  1.3,  1.4));
 ("Soy beans", (100.0, 10.7,  6.1));
 ("Eggs",      (150.0, 18.45, 3.0))].
```

A *summary profile* of a meal profile mp is a profile (ws, pws, cws), where ws, pwa and cws are obtained by adding up all weights, protein weights and carbon weights, respectively, in mp. For example, the summary profile of $mp0$ is (350.0, 30.45, 10.5).

3. Declare a function `toProfile` n w ft that returns the profile of w grams of food n given food table ft.

4. Declare a function `toMealProfile` ft m that returns the meal profile for meal m given food table ft.

5. Declare a function `toSummary: MealProfile -> Profile` that computes the summary profile for a given meal profile.

6. Make non-recursive declarations of

   1. the function `toMealProfile` from Question 4 and
   2. the function `toSummary` from Question 5.

   You may select among the following functions from the `List` library in your declarations.

```
List.fold     : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.foldBack : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
List.map      : ('a -> 'b) -> 'a list -> 'b list
List.filter   : ('a -> bool) -> 'a list -> 'a list
```

 ... Continued on next page

Problem 2 (25%)

Consider the following declaration of a function `ch`:

```
let rec ch f xs = match xs with
                  | []      -> []
                  | x::tail -> match f x with
                               | None   -> ch f tail
                               | Some v -> v::ch f tail;;
```

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

2. Describe what `ch` 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 `ch` that reflects what it is computing.

3. Provide arguments f_0 and xs_0 to `ch` so that `ch` f_0 xs_0 $=$ ["1";"2";"3"], where xs_0 must be a list with five elements.

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

5. Make an alternative non-recursive declaration of `ch` using `List.foldBack` by completing the following schema:

   ```
   let ch f xs = List.foldBack  ...  ...  ...
   ```

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

Problem 3 (13%)

Consider the following declaration of a function `f1`:

```
let rec f1 j xs = match xs with
                  | []      -> []
                  | x::tail -> (j,x)::f1 (j+1) tail;;
```

1. Make an evaluation for the expression

   ```
   f1 4 [("x",true); ("y",false); ("z",false); ("v",true)]
   ```

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

2. The declaration of `f1` is *not* tail recursive. Make a continuation-based tail-recursive variant of `f1`.

3. Make an alternative non-recursive declaration of `f1` using the function `mapi` from the `List` library by completing the following schema:

   ```
   let f1 j xs = List.mapi  ...  ...
   ```

   The function `List.mapi` has the type `(int -> 'a -> 'b) -> 'a list -> 'b list` and description:

   `List.mapi` $g\ [x_0; x_1; \ldots; x_{n-1}] = [g\ 0\ x_0; g\ 1\ x_1; \ldots; g\ (n-1)\ x_{n-1}]$ where $n \geq 0$.

Problem 4 (12%)

Consider the following declarations of three functions `h1`, `h2` and `h3`:

```
let h1 f sq = seq { for x in sq do
                        if f x then
                            yield x };;

let h2 f sq = seq { for x in sq do
                        yield f x };;

let h3 f sq = seq { for x in sq do
                        yield! f x };;
```

1. For each function `h`i, $i = 1, 2, 3$, state the type of `h`i and exemplify what the function computes by describing the value of `h`i f_i sq_i for concrete values of f_i and sq_i.

Problem 5 (25%)

Consider now the polymorphic type `T<'a>` for binary trees with two kinds of nodes: branch nodes (constructor `B`) and leaf nodes (constructor `L`). A branch node carries a value of type `'a` while a leaf node carries no further information.

```
type T<'a> = L | B of T<'a> * 'a * T<'a>;;
let t1 = B(L, 0, B(B(L,2,L), 5, L));;
let t2 = B(B(L, 1, B(L,3,L)), 0, B(L,2,L));;
```

Two F# values `t1` and `t2` are declared above and shown as trees in the following figure:

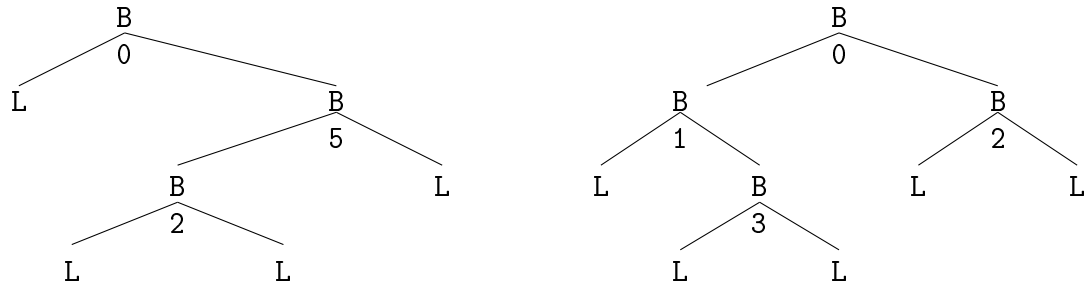

Figure 1: `t1` is shown to the left and `t2` to the right

The *height of a tree* is the maximal number of edges from the root to a leaf node (constructor `L`). For example, `t1` and `t2` have both height 3.

The *depth of a node n in a tree* is the number of edges from the root to n. For example, the node in `t2` that carries value 3 has depth 2.

A binary tree t is *balanced* if for every branch node $\texttt{B}(t_l, v, t_r)$ in t: the heights of the left and right subtrees t_l and t_r are at most 1 apart. For example, `t1` is not balanced; while `t2` is balanced.

1. State the type of `t1` and give three values of type `T<string list>`.
2. Declare a function `height` t that computes the height of tree t.
3. Declare a function `isBalanced` t that checks whether t is balanced.
4. Declare a function `atDepth` $i\,t$ that returns a list with all values occurring at depth i in tree t. For example, `atDepth 1 t2` contains 1 and 2, and `atDepth` i `t2` $= []$, when $i < 0$ or $i > 3$. State the type of `atDepth`.
5. Declare a function `makeBalanced: 'a list -> T<'a>`, so that `makeBalanced` xs creates a balanced tree having the elements of xs in branch nodes. Hint: The function `List.splitAt` $i\,[x_0; \ldots; x_{i-1}; x_i; \ldots; x_{n-1}] = ([x_0; \ldots; x_{i-1}], [x_i; \ldots; x_{n-1}])$ for $0 \leq i \leq n$ and $n \geq 0$ may be useful.