Written Examination, December 19th, 2018 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: 20%, Problem 3: 15%, Problem 4: 35%

Marking: 7 step scale.

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

You are allowed to use the .NET library including the modules described in the textbook, e.g., List, Set, Map, Seq etc.

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.

Problem 1 (30%)

Flight travellers may check-in the pieces of luggage, that should follow them on their journey, also when it contains multiple stops. A piece of luggage is marked with an *identification* (type `Lid`) by the start of the journey and that identification is associated with the *route* (type `Route`) of the journey. A route is a list of pairs identifying the *flights* (type `Flight`) and *airports* (type `Airport`) the luggage is passing on the journey.

Furthermore, a *luggage catalogue* (type `LuggageCatalogue`) is maintained, that uniquely identifies the routes of all pieces of luggage leaving some airport.

This is captured by the type declarations:

```
type Lid     = string
type Flight  = string
type Airport = string

type Route            = (Flight * Airport) list
type LuggageCatalogue = (Lid * Route) list
```

An example of a luggage catalogue is

```
[("DL 016-914", [("DL 189","ATL"); ("DL 124","BRU"); ("SN 733","CPH")]);
 ("SK 222-142", [("SK 208","ATL"); ("DL 124","BRU"); ("SK 122","JFK")])]
```

where first element in the list describes that the piece of luggage with identification "DL 016-914" is following a route, where it is first flown to Atlanta ("ATL") with flight "DL 189", then flown to Bruxelles "BRU" with flight "DL 124", and so on.

1. Declare a function `findRoute: Lid*LuggageCatalogue -> Route`, that finds the route for a given luggage identification in a luggage catalogue. A suitable exception should be raise if a route is not found.

2. Declare a function `inRoute: Flight -> Route -> bool`, that decides whether a given flight occurs in a route.

3. Declare a function `withFlight` f lc, where f is a flight and lc is a luggage catalogue. The value of the expression `withFlight` f lc is a list of luggage identifiers for the pieces of luggage that should travel with f according to lc. The sequence in which the identifiers occur in the list is of no concern.

   For the above example, both "DL 016-914" and "SK 222-142" should travel with the flight "DL 124".

 ... Continued on next page

An *arrival catalogue* associates with every airport, identifications of all pieces of luggage that should arrive at the airport. This is captured by the type declaration:

```
type ArrivalCatalogue = (Airport * Lid list) list
```

The following arrival catalogue is derived from the luggage catalogue appearing on the previous page:

```
[("ATL", ["DL 016-914"; "SK 222-142"]);
 ("BRU", ["DL 016-914"; "SK 222-142"]);
 ("JFK", ["SK 222-142"]);
 ("CPH", ["DL 016-914"])]
```

4. Declare a function `extend: Lid*Route*ArrivalCatalogue -> ArrivalCalalogue` so that `extend`(lid, r, ac) is the arrival catalogue obtained by extending ac with the information that lid will arrive at each airport contained in route r.

5. Declare a function `toArrivalCatalogue: LuggageCatalogue -> ArrivalCatalogue`, that creates an arrival catalogue from the information of a given luggage catalogue.

   You should solve this exercise using `extend` from the previous question in combination with either `List.fold` or `List.foldBack`. The types of these functions are:

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

   Notice that this exercise can be solved even in the case that you did not provide a declaration for `extend`.

Problem 2 (20%)

Consider the following F# declarations:

```
let rec f(xs,rs) = match xs with
                   | []   -> rs
                   | [x]  -> x::rs
                   | x1::x2::xs -> x1::f(xs,x2::rs)
let g xs = f(xs,[]);;
```

1. Give a step-by-step evaluation (using $\rightsquigarrow$) for `g [1;2;3;4;5]` determining the value of the expression. There should at least be one step for every recursive call of `f`.
2. Give the types for `f` and `g`, and describe what `g` computes. Your description should focus on *what* it computes, rather than on individual computation steps.
3. The declaration of `f` is *not* tail recursive. Explan briefly why this is the case.

The following tail-recursive function `fA` is suggested as a variant of f that is based on an additional parameter.

```
let rec fA (xs,rs) acc = match xs with
                         | []         -> (List.rev acc) @ rs
                         | [x]        -> (List.rev acc) @ (x::rs)
                         | x1::x2::xs -> fA (xs, x1::rs) (x2::acc)
let g' xs = fA(xs,[]) [];;
```

That is, the intention is that `g'` xs = `g` xs, for any list xs. But the declaration of `fA` contains programming mistakes, and, for example, `g' [1;2;3;4;5]` gives `[2;4;5;3;1]`, which is different from the value of `g [1;2;3;4;5]`.

4. Find, explain and correct the mistakes in `fA`.
5. Provide a declaration of a continuation-based, tail-recursive variant of `f`. Your tail-recursive declaration must be based on an explicit recursion.

Problem 3 (15%)

The below three `.fsx`-files (`Program`i`.fsx`, for $i = 1, 2, 3$) contain errors and they are all rejected by the FSharp compiler with error messages.

For each of the three programs: Find the cause of the error in the program and explain it in your own words.

You need *not* correct the errors.

Program1.fsx:

```
let rec f(x, ys) = match ys with
                   | []    -> [x*x]
                   | y::ys -> f (x*y) ys;;
```

Program2.fsx:

```
type T = C1 of int | C2 of int * T

let rec g t = match t with
              | T i -> i
              | T (i,t1) -> i + g t1;;
```

Program3.fsx:

```
let rec h n f = if n = 0 then f 1 else h f(n-1) f;;
```

Problem 4 (35%)

Consider the following declaration of a type for figures:

```
type Fig<'a> = | C of 'a*int | R of 'a*int*int
               | V of Fig<'a> * Fig<'a> | H of Fig<'a> * Fig<'a>;;
```

where figures of type `Fig<'a>` are constructed from circles (constructor `C`) and rectangles (constructor `R`) by vertical composition (constructor `V`) of two figures and horizontal composition (constructor `H`) of two figures. A circle `C`(v, r) is characterized by a value v of type `'a` and its radius r and a rectangle `R`(v, a, b) is characterized by a value v of type `'a` and the lengths a and b of its horizontal and vertical sides, respectively. Circles and rectangles are called *elementary figures*. Furthermore, `V`(f_1, f_2) denotes that f_2 is placed on top of f_1, and `H`(f_1, f_2) that f_2 is placed to the right of f_1. Figs. 1 and 2 provide graphical illustrations.

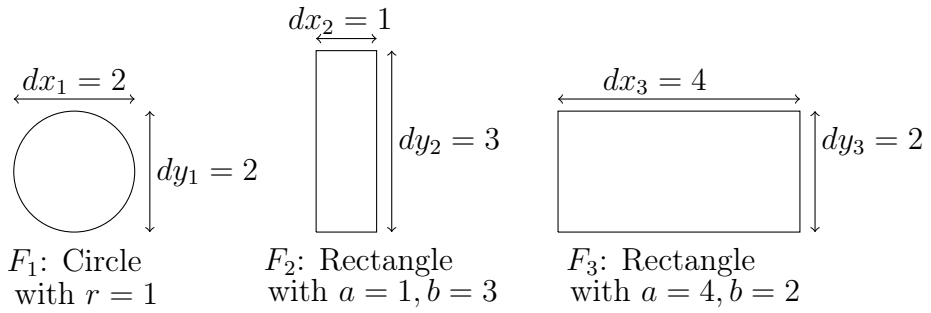

Figure 1: Three elementary figures with their dimensions (dx_i, dy_i)

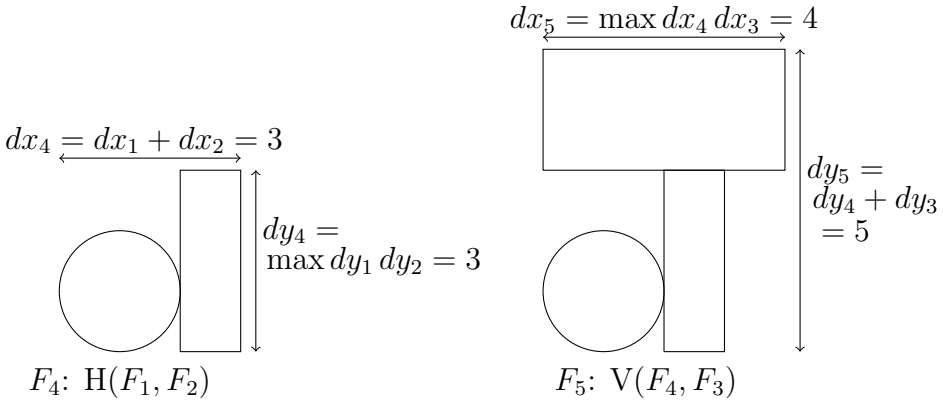

Figure 2: Dimensions for horizontal and vertical combinations

The *bounding box* or *dimension* of a figure is a pair of integers (dx, dy) describing the smallest horizontal and vertical side lengths of a rectangle that can surround the figure:

- A circle with radius r has dimension $(2r, 2r)$.
- A rectangle with horizontal and vertical side lengths a and b has dimension (a, b).
- Dimensions for horizontal and vertical combinations of figures are shown in Fig. 2.

... Continued on next page

1. Give four values of type `Fig<int*int>` using all four constructors.

2. Declare a function that counts the number of elementary figures occurring in a figure.

3. Declare a function `extract: Fig<'a> -> 'a list` that gives a list with the values occurring in a figure. The sequence in which values occur in the list is of no concern.

4. Declare a function `dim: Fig<'a> -> int*int` to compute the dimension of a figure.

A figure should now be placed in a coordinate system, so that the *lower left corner* of its bounding box is at a given position (x, y), where x and y are integers. We shall also call (x, y) the position of the figure.

We follow the convention that the x-axis is horizontal and grows to the right and the y-axis is vertical grows upwards.

```
type PosFig = Fig<int*int>
```

A figure *fig* can be transformed into a *positioned figure* (type `PosFig`) by substituting in positions for the values in its elementary figures. We shall consider a transformation that is guided by the following rules for placing *fig* in position (x, y):

- if *fig* is either a circle or a rectangle, then it is placed at (x, y),
- if *fig* is a vertical combination of fig_1 and fig_2, then the resulting positioned figure is the vertical combination obtained by placing fig_1 at (x, y) and fig_2 at $(x, y + dy_1)$, where dy_1 is the vertical extent of fig_1, and
- if *fig* is a horizontal combination of fig_1 and fig_2, then the resulting positioned figure is the horizontal combination obtained by placing fig_1 at (x, y) and fig_2 at $(x+dx_1, y)$, where dx_1 is the horizontal extent of fig_1.

For example, placing F_5 (see Figure 2) in position $(0, 0)$ according to these rules would result in placing the circle (F_1) at position $(0, 0)$, the small rectangle (F_2) at position $(2, 0)$ and the big rectangle (F_3) at position $(0, 3)$.

5. Declare a function `toPosFig: int*int -> Fig<'a> -> PosFig`. The value of the expression `toPosFig` (x, y) f is the positioned figure obtained by placing f in position (x, y) according to the above rules. Hint: the function `dim` can be used to extract the horizontal and vertical extents of a figure.

A vector v is given by a pair of integers.

```
type Vector = int*int
```

A positioned figure *fig* can be *displaced or moved according to a vector* $v = (v_1, v_2)$ by replacing every position (x, y) occurring in *fig* by $(x + v_1, y + v_2)$.

6. Declare a function `move: Vector -> PosFig -> PosFig` so that `move` v *fig* is the figure obtained by moving *fig* according to vector v.