Written Examination, December 17th, 2020 Course no. 02157

The duration of the examination is 4 hours.

Course Name: Functional programming

Allowed aids: All aid

The problem set consists of 4 problems which are weighted approximately as follows:
Problem 1: 25%, Problem 2: 20%, Problem 3: 35%, Problem 4: 20%

Marking: 7 step scale.

... Continued on next page

Formalities

This is a written 4 hours' exam to be taken at the student's home. It is an online exam with all aids allowed and open access to the internet. In particular, it is expected that you use an F# system. The exam set is announced on the Inside group of 02157 Functional Programming E20 under Assignments (as you may know from mandatory assignments in other courses). The exam set is released at 9:00 on Thursday, December 17, and your solution must be uploaded 4 hours later, that is no later than 13:00.

The exam set consists of a pdf-file `exam02157.pdf`. Furthermore, there is an accompanying file `ProgramSkeleton02157.fsx` containing program snippets from the pdf-file. You should hand in your solution in the form of a single F# file (with extension `.fsx` or `.fs`) , where the file name consists of your name and study number, for example, `JohnSmithS012345.fsx`. The file contents should start with your name and study number.

Textual answers to questions, explanations etc. should be included as comments in the solution file. For example, the answer to Question 2 in Problem 3 could appear as follows:

```
(*
Question 3.2

----- your answer to this question -----

*)
```

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. If a program you want to hand in does not pass the compiler, then include it in a comment as shown above.

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

Exam Fraud

This is a strictly individual exam. You are not allowed to discuss any part of the exam with anyone in or outside the course. Submitting answers (code or text) you have not written entirely by yourself, or sharing your answers with others, is considered exam fraud.

This is an open-book exam, and so you are welcome to make use of any reading material from the course, or elsewhere. However, make sure to use proper and specific citations for any material from which you draw inspiration - including what you may find on the Internet, such as snippets of code.

Also note that it is not allowed to copy any part of the exam text (or supplementary skeleton file) and publish or share it with others during or after the exam.

Breaches of the above policy will be handled in accordance with DTU's disciplinary procedures.

... Continued on next page

Problem 1 (25%)

Consider a *register* of the following form: $[(p_1, ps_1); \ldots ; (p_n, ps_n)]$, where p_i is a *person* and ps_i is a list of *persons*, for $1 \leq i \leq n$ and $n \geq 0$. The interpretation of this register is that p_i has had *close contact* with all persons in the list ps_i. This is captured by the type declarations:

```
type Person = string
type Contacts = Person list
type Register = (Person * Contacts) list

let reg1 = [("p1", ["p2"; "p3"]); ("p2", ["p1"; "p4"]);
            ("p3", ["p1"; "p4"; "p7"]); ("p4", ["p2"; "p3"; "p5"]);
            ("p5", ["p2"; "p4"; "p6"; "p7"]);
            ("p6", ["p5"; "p7"]); ("p7", ["p3"; "p5"; "p6"])];;
```

For a register $[(p_1, ps_1); \ldots ; (p_n, ps_n)]$ we require that the persons in ps_i are all different. We call this condition *invariant 1.*

Furthermore, we require that the p_i's are all different and that each list ps_i is a non-empty list. We call these two conditions *invariant 2.*

For simplicity reasons we allow p_i to be an element of ps_i. Furthermore, the sequence in which persons occur in lists is of no concern in this problem.

1. Declare a function that checks whether invariant 1 holds for a given register.
2. Declare a function that checks whether invariant 2 holds for a given register.

You may find the following two functions useful when you solve the remaining questions of this problem.

```
let rec insert p ps = if List.contains p ps then ps else p::ps

let rec combine ps1 ps2 = List.foldBack insert ps1 ps2;;
```

The list of *immediate contacts* of a person p for a register reg is ps if (p, ps) is an element of reg. Otherwise, the list of immediate contacts is the empty list [].

3. Declare a function that extracts the list of immediate contacts of a person for a given register.
4. Declare a function `addContact` p_1 p_2 reg that gives the register obtained from reg by adding p_2 as a close contact of p_1. You can assume that reg satisfies the two invariants. The value of the function must satisfy these invariants as well.

... Continued on next page

The list of *contacts* of a person p for a register reg consists of

- the immediate contacts of p in reg together with
- the immediate contacts of all immediate contacts of p_i.

For example, the list of contacts of `"p1"` in register `reg1` consists af the persons: `"p1"`, `"p2"`, `"p3"`, `"p4"` and `"p7"`.

A list of contacts must contain no duplicates.

5. Declare a function that extracts the list of contacts of a person for a given register.

... Continued on next page

Problem 2 (20%)

The function mapi from the List library could have the following declaration:

```
let rec h f xs j = match xs with
                   | []      -> []
                   | x::rest -> f j x :: h f rest (j+1);;
val h : (int -> 'a -> 'b) -> 'a list -> int -> 'b list

let mapi f xs = h f xs 0;;
val mapi : (int -> 'a -> 'b) -> 'a list -> 'b list
```

where h is a helper function. Notice that the F# system automatically infers the types of h and mapi.

1. Give an argument showing that (int -> 'a -> 'b) -> 'a list -> int -> 'b list is the most general type of h and that (int -> 'a -> 'b) -> 'a list -> 'b list is the most general type of mapi. That is, any other type for h is an instance of (int -> 'a -> 'b) -> 'a list -> int -> 'b list. Similarly for mapi.

An example using mapi is:

```
mapi (fun i x -> (i,x)) ["a";"b";"c"];;
val it : (int * string) list = [(0, "a"); (1, "b"); (2, "c")]
```

2. Give an evaluation showing that [(0, "a"); (1, "b"); (2, "c")] is the value of the expression mapi (fun i x -> (i,x)) ["a";"b";"c"]. Present your evaluation using the notation $e_1 \rightsquigarrow e_2$ from the textbook, where you can use => in your F# file rather than $\rightsquigarrow$. You should include at least as many evaluation steps as there are calls of mapi and h.
3. Give the most general types for the expressions (fun i x -> (i,x)) and ["a";"b";"c"] and explain why these are the most general types.

   Furthermore, explain why the type of mapi (fun i x -> (i,x)) ["a";"b";"c"] is (int*string) list.
4. Provide a declaration of a tail-recursive variant of h that is based on an accumulating parameter. Your tail-recursive declaration must be based on an explicit recursion.

... Continued on next page

Problem 3 (35%)

Consider a type `Part` for trees, where leaves are *simple parts* and nodes are *composite parts*. A simple part is described by a *name*, and a composite part is described by a name and a *non-empty* list of parts – its immediate sub-parts.

```
type Name = string;;

type Part = | S of Name                  // Simple part
            | C of Name * Part list;;   // Composite part
```

1. Declare a Boolean-valued function that, for a given part p, checks whether each composite part occurring in p is associated with a non-empty list of sub-parts.

You may from now on assume that lists of sub-parts are non-empty.

The *depth* of a part is defined as follows:

- The depth of a simple part is 0.
- The depth of a composite parts $\texttt{C}(n, [p_1, \ldots, p_n])$ is 1 plus the maximal depth of the sub-parts p_i, where $1 \leq i \leq n$ and $n > 0$.

2. Declare a function that computes the depth of a part.
3. The type declaration for `Part` comprises: `Part`, `S`, `C`, `Name`, `*` and `list`. Give a brief description of each of these 6 components of the declaration.
4. Make a declaration of a well-formed part p subject to the following conditions:
   - The depth of p is 3.
   - Five different simple parts occur somewhere in p.
   - Four different composite parts occur in p including p.
   - The four composite parts should have different lengths of their list of immediate sub-parts.
5. Declare a function of type `Part -> Set<Name>` that for a given part p extracts the set of names of the simple parts occurring in p.

... Continued on next page

The same part name may occur several times in a Part. The notion of *occurrence count* keeps track of the number of occurrences of names. This is captured by the following declaration

```
type OccurrenceCount =  Map<Name,int>
```

where an entry (n, c) with key n and value c in an occurrence count denotes c occurrences of name n in a part.

6. Declare a function with type `Part -> OccurrenceCount`. For a given part p, the function should return the occurrence count for the names occurring in p.

Problem 4 (20%)

Consider the following declaration of a tail-recursive function `gC`:

```
let rec gC i k =
   if i=0 then k 0
   else if i=1 then k 1
        else gC (i-1) (fun v1 -> gC (i-2) (fun v2 -> k(v1+v2)));;
```

1. Give a declaration of a function `g` so that
   - `g` i = `gC` i `id`, where `id` is the identity function, and
   - `g` is not tail recursive.

Consider now the sequence 1 -3 5 -7 9 -11 $\cdots$ of integers, where the i'th element n_i (for $i \geq 0$) is given by:

$$n_i = \begin{cases} 2 \cdot i + 1 & \text{if } i \text{ is even} \\ -(2 \cdot i + 1) & \text{if } i \text{ is odd} \end{cases}$$

2. Give an F# declaration for the above infinite sequence $n_0\ n_1\ n_2\ \cdots$.

3. Give an F# declaration of the sequence of type `Seq<float>` with elements $x_0\ x_1\ x_2\ \cdots$, where $x_i = 1/n_i$. Thus, the list containing the first three elements of this sequence is: `[1.0; -0.3333333333; 0.2]`.

4. Give an F# declaration of the sequence of type `Seq<float>` with elements $y_0\ y_1\ y_2\ \cdots$, where $y_i = \Sigma_{k=0}^{i} x_i$. Thus, the list containing the first three elements of this sequence is: `[1.0; 0.6666666667; 0.8666666667]`.