Written Examination, May 27th, 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 3 problems which are weighted approximately as follows:
Problem 1: 30%, Problem 2: 35%, Problem 3: 35%

Marking: 7 step scale.

... Continued on next page

Formalities

This is a written exam to be taken at the student's home. The duration of the exam is 4 hours. It is an on-line 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 F20 under Assignments as you know from the mandatory assignments in the course. The exam set is released at 9:00 on Wednesday, May 27, 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` that contains program snippets from the pdf-file. You should hand in your solution in the form af a single F# file (named either `02157.fsx` or `02157.fs`). 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.

Do not 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 on, 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 (30%)

We consider now lists of of the form: $[(x_1, ys_1); (x_2, ys_2); \ldots; (x_n, ys_n)]$, where ys_i is a list of elements, for $0 \leq i \leq n$. That is, we consider values of the following type:

```
type Tab<'a,'b> = ('a * 'b list) list

let t1: Tab<int,string> = [(1, ["a"; "b"; "c"]); (4,["b"; "e"])];;
```

We require that the x_i's in $t = [(x_1, ys_1); (x_2, ys_2); \ldots; (x_n, ys_n)]$ are all different. Furthermore, we require that the lists ys_i are not empty; but we do not care about repetitions and the order of the elements in ys_i. We consider t is a special kind of a table, where we call x_i a *key* and ys_i the *associated value.*

1. Declare a function: `isKey: 'a -> Tab<'a,'b> -> bool  when 'a : equality`. The value of `isKey` $x\,t$ is true if and only if x is a key in t. For example: `isKey 1 t1` is true and `isKey 2 t1` is false.
2. Declare a function: `insert`(x, y, t). If x is a key of t with associated value ys, then the value of `insert`(x, y, t) is obtained from t by replacing the pair (x, ys) in t with $(x, y :: ys)$. If x is not a key of t, then the value of the function is obtained by adding the pair $(x, [y])$ to t. For example: `insert(4,"a",t1)` could contain the pairs: `(1, ["a";"b";"c"])` and `(4,["a";"b";"e"])]`.
3. Declare a function `deleteKey` $x\,t$. If x is a key of t with associated value ys, then the value of `deleteKey` $x\,t$ is obtained from t by removing the pair (x, ys) from t. Otherwise the value of the function is t.
4. Declare a function `deleteElement` $y\,t$, where $t = [(x_1, ys_1); (x_2, ys_2); \ldots; (x_n, ys_n)]$ is a value of type `Tab<'a,'b>`. The value of the function is obtained from t by deletion of all occurrences of y from the lists ys_i, for $1 \leq i \leq n$.
5. Declare a function `fromPairs:('a*'b) list -> Tab<'a,'b>` that converts a list of pairs to a value of type `Tab<'a,'b>`, e.g. `fromPairs[(2,"c");(1,"a");(2,"b")]` could give `[(2,["c";"b"]);(1,["a"])]`.

... Continued on next page

Problem 2 (35%)

The function `allPairs` from the `List` library could have the following declaration:

```
let rec f x = function
              | []    -> []
              | y::ys -> (x,y)::f x ys;;
val f : 'a -> 'b list -> ('a * 'b) list

let rec allPairs xs ys =
   match xs with
   | []       -> []
   | x::xrest -> f x ys @ allPairs xrest ys;;
val allPairs : 'a list -> 'b list -> ('a * 'b) list
```

where `f` is a helper function. Notice that the F# system automatically infers the types of `f` and `allPairs`.

1. Give an argument showing that `'a -> 'b list -> ('a * 'b) list` is indeed the most general type of `f` and that `'a list -> 'b list -> ('a * 'b) list` is indeed the most general type of `allPairs`. That is, any other type for `f` is an instance of `'a -> 'b list -> ('a * 'b) list`. Similarly for `allPairs`.

An example using `f` is:

```
f "a" [1;2;3];;
val it : (string * int) list = [("a", 1); ("a", 2); ("a", 3)]
```

2. Give an evaluation showing that `[("a", 1); ("a", 2); ("a", 3)]` is the value of the expression `f "a" [1;2;3]`. Present your evaluations 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 recursive calls.
3. Explain why the type of `f "a" [1;2;3]` is `(string * int) list`.
4. The declaration of `f` is *not* tail recursive. Explan briefly why this is the case.
5. Provide a declaration of a tail-recursive variant of `f` that is based on an accumulating parameter. Your tail-recursive declaration must be based on an explicit recursion.
6. Provide a declaration of a continuation-based, tail-recursive variant of `f`. Your tail-recursive declaration must be based on an explicit recursion.
7. 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.

... Continued on next page

Problem 3 (35%)

We now consider a type `T<'a>` for trees, where leaves (constructor `A`) and nodes (constructors `B` and `C`) carry values of type `'a`. Furthermore, nodes can have one subtree (constructor `B`) or two subtrees (constructor `C`):

```
type T<'a> = | A of 'a
             | B of 'a * T<'a>
             | C of 'a * T<'a> * T<'a>
```

1. Declare four different values of type `T<int list * string>`. All three constructors must be used in each declaration.

The below declaration is an (erroneous) attempt to declare a function `mapT` f t, that takes a function f : `'a -> 'b` and a tree t : `T<'a>` as arguments. It should return a tree of type `T<'b>` that is created from t by application of f to the values in leaves and nodes:

```
let rec mapT f t =
   match t with
   | A v         -> A f v
   | T B(v,t1)   -> B(f v, mapT f t1)
   | C(v,t1,t2)  -> C(f v, mapT(f, t1), mapT(f,t2));;
```

2. Find all errors in the declaration. For each error found: explain in your own words what causes the error.
3. Make a new declaration of `mapT` where all errors are corrected.
4. Test your corrected function on the four values from the first question. For these tests you should invent a suitable function f.
5. Declare a function `toSet: T<'a> * ('a -> bool) -> Set<'a>`. The value of `toSet`(t, p) is the set of all values v occurring in t that satisfy the predicate p, that is where $p(v)$ is true.
6. You should extend the above declaration of `T<'a>` to include an extra constructor `D`. It should be possible to construct nodes with an arbitraty number of subtrees using `D`. That is, `D`$(v, [t_1; t_2; \dots t_n])$ is a value of the extended type, when v is a value and $t_1, t_2, \dots t_n$ are trees. Furthermore, declare a value of type `T<int>` where constructor `D` is used.
7. Extend your declaration of `mapT` from Question 3. so that it covers constructor `D`.