Written exam, Functional Programming Tuesday June 6, 2017

Version 1.03 of 2017-06-25

These exam questions comprise 9 pages. Check immediately that you have all the pages.

The exam duration is 4 hours.

There are 4 questions. To obtain full marks you must answer all the subquestions satisfactorily.

You are allowed to use books, lecture notes, lecture slides, hand-ins, solutions to assignments, calculators, computers, software, on-line resources etc. during the examination. This includes any form of device that can execute programs written in F#.

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

If a subquestion requires you to define a particular function, then you may **use that function in subsequent subquestions**, even if you have not managed to define it yourself.

If a subquestion requires you to define a particular function, then you may **define as many helper functions as you want**, but in any case you must define the required function so that it has exactly the type and effect that the subquestion asks for.

The grading will favour functional solutions, i.e., solutions without side effects. Recursion is also favoured over loops. An imperative solution is of course preferred over no solution.

You should hand-in one ASCII file only, e.g., `bfnp2017.fsx`. Do not use time on formatting your solution in Word or PDF.

You are welcome to download the accompanying file `jun2017Snippets.txt` from the course homepage in LearnIT: `https://learnit.itu.dk/course/view.php?id=3016512`. The file contains some of the code snippets included in the exam set for your convenience to copy into your solution.

You MUST include explanations and comments to support your solutions. You simply write them as comments around your code.

Your exam hand-in must be made by yourself and yourself only, and this holds for program code, examples, the explanation you provide for the code, and all other parts of the answers. It is illegal to make the exam answers as group work or to enlist the help of others in any way.

Your hand-in must contain the following declaration:

I hereby declare that I myself have created this exam hand-in in its entirety without help from anybody else.

Question 1 (30%)

We define a *priority set* as a prioritised collection of elements that may not contain duplicates. A priority set is a combination of a set (HR page 104) and a priority queue. An element can only be in the set once (set property) but elements are prioritised according to insertion order within the set (priority queue).
We can add and remove elements and ask for membership as with ordinary sets. What is unusual is that we keep track of the order in which elements are inserted, so one can ask for eg. the oldest element in the priority set, the one inserted before all the others.
For instance, inserting the elements a, b and c in an empty priority set in that order results in the priority set $\{a^1, b^2, c^3\}$. We define the *priority number* as an unique number representing the priority given to an element in the set. The priority number is annotated in superscript on each element. The element a is inserted first and hence gets priority number 1, b is inserted second and gets priority number 2 etc.
The smallest possible priority number is defined to be 1. Inserting the element b a second time will not change the priority set and the element b still has the priority number 2. We define the *first element* in the priority set as the element with lowest priority number, i.e., 1. Element a is the first element in the example above.
The unique priority number assigned an element may change as the set changes. For instance, removing element a from the priority set above results in a new priority set where the priority number for b and c has changed: $\{b^1, c^2\}$; b is then the first element in the set .
A priority set can be implemented using a simple F# list, `List` (HR page 93). We do not explicitly store the priority number of an element as it can be calculated from the position of the element in the list. The element with priority number 1 is the first element in the list.

```
type PrioritySet<'a when 'a: equality> = PrioritySet of List<'a>
```

The example above is declared as

```
let psEx = PrioritySet ["a";"b";"c"]
```

Question 1.1

Consider the following elements and assume they are inserted in an empty priority set in this order: `"a"`, `"q"`, `"a"`, `"b"`, `"b"`, `"q"`, `"d"`, `"a"`.

- Declare a value `priSetEx`, being the result of inserting the elements above according to the definition of a priority set.
- What is the type of the value `priSetEx`.
- Declare a value `empty` representing an empty priority set, i.e., priority set with no elements.

Question 1.2

- Declare a function

  ```
  isEmpty : PrioritySet<'a> -> bool when 'a : equality
  ```

  that returns true if a priority set is the empty set. For instance `isEmpty(empty)` returns true. The value `empty` is defined above.

- The *size* of a priority set is the number of elements in the set. Declare a function

```
size : PrioritySet<'a> -> int when 'a : equality
```

that returns the size of a priority set. For instance, `size psEx` returns 3.

- Declare a function `contains` *e ps* of type

  ```
  contains : 'a -> PrioritySet<'a> -> bool when 'a : equality
  ```

  that returns true if the priority set ps contains an element e. For instance `contains "b" psEx` returns true.

- Declare a function `getPN` *e ps* of type

  ```
  getPN : 'a -> PrioritySet<'a> -> int when 'a : equality
  ```

  that returns the priority number of element e if exists in priority set ps. Otherwise raises an error exception (`failwith`). For instance `getPN "a" psEx` returns 1.

Question 1.3

- Declare a function `remove` *e ps* of type

  ```
  remove : 'a -> PrioritySet<'a> -> PrioritySet<'a> when 'a : equality
  ```

  that removes element e from the priority set ps and returns a new priority set. Nothing changes if e does not exists in ps. For instance, `remove "b" psEx` returns the priority set `PrioritySet ["a";"c"]`.

- Declare a function

  ```
  add : 'a -> PrioritySet<'a> -> PrioritySet<'a> when 'a : equality
  ```

  where `add e ps` returns the priority set `ps` with the element `e` added with lowest priority (highest priority number) unless already in the set `ps`. Adding element *h* to priority set $\{a^1, b^2, c^3\}$ gives the priority set $\{a^1, b^2, c^3, h^4\}$. Adding element *b* to $\{a^1, b^2, c^3\}$ gives the unchanged priority set $\{a^1, b^2, c^3\}$.

Question 1.4

- Declare a function `map` *f ps* of type

  ```
  map : ('a -> 'b) -> PrioritySet<'a> -> PrioritySet<'b>
    when 'a : equality and 'b : equality
  ```

  where `map` *f ps* returns the priority set where the function *f* has been applied on all elements in the priority set *ps* in order of priority number. For instance `map (fun (c:string) -> c.ToUpper()) psEx` returns the priority set value `PrioritySet ["A";"B";"C"]`.

- Declare a function `cp` of type

```
cp : PrioritySet<'a> -> PrioritySet<'b> -> PrioritySet<'a * 'b>
       when 'a : equality and 'b : equality
```

where `cp` *ps1 ps2* returns the cartesian product of *ps1* and *ps2*. The result set is generated from *ps1* and *ps2* in order of priority number of *ps1* first and then *ps2*. For instance the cartesian product of $\{A^1, B^2, C^3\}$ and $\{h^1, i^2\}$ is $\{(A,h)^1, (A,i)^2, (B,h)^3, (B,i)^4, (C,h)^5, (C,i)^6\}$. A cartesian product involving an empty set is the empty set, eg. `cp psEx empty` is the empty set.

Question 2 (20%)

Consider the F# declaration:

```
let f curRow =
  let rec f' = function
      []              -> []
    | [_]             -> [1]
    | xs              -> let (x1::x2::xs) = xs
                         x1 + x2 :: f' (x2::xs)
  (1 :: f' curRow)
```

with type `int list -> int list`.

Question 2.1

Describe what `f` computes given the examples below:

- `f [1]` gives `[1; 1]`
- `f [1; 1]` gives `[1; 2; 1]`
- `f [1; 2; 1]` gives `[1; 3; 3; 1]`
- `f [1; 3; 3; 1]` gives `[1; 4; 6; 4; 1]`

Question 2.2

Compiling the function `f` and `f'` above gives the warning:

> *warning FS0025: Incomplete pattern matches on this expression. For example, the value '[_]' may indicate a case not covered by the pattern(s).*

Write a version of `f` and `f'`, called `fMatch` and `fMatch'`, without this warning. Explain why the warning disappears.

Question 2.3

The function `f'` is not tail recursive. Write a tail-recursive version, `fA'` of `f'` (or `fMatch'`) using an accumulating parameter.

Question 3 (20%)

Question 3.1

Consider the F# declaration:

```
let mySeq s1 s2 =
  seq { for e1 in s1 do
          for e2 in s2 do
            yield! [e1;e2] }
```

of type `seq<'a> -> seq<'a> -> seq<'a>`.

- Describe the sequence returned by `mySeq` when called with arbitrary sequences `s1` and `s2`.
- Can you for any arguments to `mySeq` generate the following value `seq ['A';'D'; 'A';'E'; 'A';'F'; 'B';'D'; 'B';'E'; 'B';'F']`

Question 3.2

Declare a function `mySeq2 s1 s2` of type `seq<'a> -> seq<'b> -> seq<'a * 'b>` such that the cartesian product of `s1` and `s2` is returned. For instance `mySeq2 [1;2] ['A';'B';'C']` gives the result `seq [(1,'A'); (1,'B'); (1,'C'); (2,'A'); (2,'B'); (2,'C')]`.

Question 3.3

Declare a function `mySeq3` of type `int -> seq<int>`, such that `mySeq3 n` produces the infinite sequence $\mathtt{n}^2 - \mathtt{n} * i$ for $i >= 0$. The identifier i is the index of the element in the sequence.
Hint: Consider using `Seq.initInfinite`.

Question 4 (30%)

We shall now consider an internal DSL called *DataSpec* to be used for generating test data. With *DataSpec* one can create a specification of a process to generate an arbitrary number of data records. The DSL for *DataSpec* uses the following F# type declaration:

```
type DataSpec =
    RangeInt of int * int
  | ChoiceString of string list
  | StringSeq of string
  | Pair of DataSpec * DataSpec
  | Repeat of int * DataSpec
```

- `RangeInt` (b, e) specifies the generation of a random number in the integer interval $[b \dots e]$.
- `ChoiceString`$[s_1, \dots, s_n]$ specifies the choice of one string among n different strings.
- `StringSeq` s specifies the generation of an unique string by appending an unique number to each string generated. E.g., `StringSeq "A"` may generate the strings `A1`, `A2`, `A3`, ...
- `Pair` (ds_1, ds_2) specifies the generation of a pair of values.
- `Repeat` (n, ds) specifies the generation of a collection $v_1, \dots, v_n$ of n values where v_i is the i'th value generated by ds, for $1 <= i <= n$. For instance, `Repeat (3,RangeInt(1,5))` may generate the following 3 values `5`, `1` and `4`, i.e., 3 arbitrary values between 1 and 5.

Consider the cash register example (HR page 83), below expressed as an F# value:

```
  let reg = [("a1",("cheese",25));
             ("a2",("herring",4));
             ("a3",("soft drink",5))]
```

A specification, of type `DataSpec`, for a similar register using the DSL above is

```
  let reg =
    Repeat(3,Pair(StringSeq "a",
                  Pair(ChoiceString["cheese";"herring";"soft drink"],
                       RangeInt(1,100))))
```

The *article codes* are generated as a string sequence, e.g., `a1`, `a2` etc. The *price* is an arbitrary number between 1 and 100. We have only three possible *article names*: cheese, herring and soft drink.

Question 4.1

Declare an F# value `pur`, of type `DataSpec` that is a specification for a purchase like below:

```
  let pur = [(3,"a2"); (1,"a1")]
```

The first element in each pair is the *number of pieces* which we choose to be an arbitrary integer between 1 and 10 (`RangeInt`). The second element of each pair is an *article code* specified as a sequence of strings (`StringSeq`). Use the constructors `Pair` and `Repeat` to generate two pairs.

Question 4.2

Declare a function `genValue` *ds* of type

```
genValue : DataSpec -> string
```

such that `genValue` returns a string representation of the values generated given the specification *ds*. Given the randomness built into the data generator, the result of `genValue reg` could be:

```
"[(a1,(cheese,69));(a2,(herring,94));(a3,(cheese,50))]"
```

The randomness does not prohibit the same *article name* to be used several times, e.g., cheese.
Hint: You need a way to generate random numbers to handle `RangeInt` and `ChoiceString`. The function `next` (i_1, i_2) below returns a random integer in the interval $[i_1, \ldots, i_2[$ using the random generator `rand`. You also need a way to generate unique numbers for `StringSeq`. The function `numGen` () below returns a new unique number each time it is called. You may also use the template for `genValue` below:

```
let rand = System.Random()
let next(i1,i2) = rand.Next(i1,i2)
let numGen =
  let n = ref 0
  fun () -> n := !n+1; !n

let rec genValue = function
    RangeInt(i1,i2) -> next(i1,i2).ToString()
  | ChoiceString xs -> ...
  | ...
```

Question 4.3

The declaration of the register `reg` and purchase `pur` above is independent. This means that nothing prevents a purchase from containing *article codes* that do not exists in the register. We fix this by extending the DSL with a way to *label* generated data and the ability to *pick* from this data later. The new type for `DataSpec` is

```
type DataSpec =
    RangeInt of int * int
  | ...
  | Pick of string
  | Label of string * DataSpec
```

We can then define modified versions of the register and purchase specifications:

```
let reg2 = Repeat(3,Pair(Label("articleCode",StringSeq "a"),
                          Pair(ChoiceString["cheese";"herring";"soft drink"],
                               RangeInt(1,100))))
let pur2 = Repeat(2,Pair(RangeInt(1,10), Pick "articleCode"))
```

Every time we generate a new article code using `StringSeq` under the `Label` we add the result value to an environment under the name `"articleCode"`. With the specification for `reg2` above, we will end with an environment mapping the string `"articleCode"` to three possible values because we repeat 3 times. The environment can thus be defined as a map from strings to a list of strings:

```
type Env = Map<string,string list>
```

Declare a function `addToEnv` *s v dEnv* of type

```
addToEnv : string -> string -> Env -> Env
```

that adds the value *v* to the environment *dEnv* under the name *s*. If *s* already exists in *dEnv*, then add *v* to the list of values under *s* already in *dEnv*. For instance, `addToEnv "x" "42" env`, where `env` = `map [("x", ["43"])]` returns the new environment `map [("x", ["42"; "43"])]`.
Declare a function `pickFromEnv` *s dEnv* of type

```
pickFromEnv : string -> Env -> string
```

that picks an arbitrary value *v* from the environment *dEnv* under the name *s*. Use helper function `next` to pick an arbitrary value. In case *s* does not exists in the environment then raise an exception (`failwith`). For instance, `pickFromEnv "x" env` could return the string `"43"`.

Question 4.4

Declare a function `genValue` *dEnv ds* of type

```
genValue : Env -> DataSpec -> string * Env
```

that returns a string representation of the values generated and a new environment given the input environment *dEnv* and specification *ds*. For instance,

```
let (v,dEnv) = genValue Map.empty reg2
```

may return

```
val v : string = "[(a18,(herring,44));(a19,(herring,7));(a20,(cheese,13))]"
val dEnv : Env = map [("articleCode", ["a20"; "a19"; "a18"])]
```

Applying `genValue dEnv pur2` may then return

```
("[(8,a20);(5,a19)]", map [("articleCode", ["a20"; "a19"; "a18"])])
```

Two article codes `a20` and `a19` have been *picked* from the environment `dEnv` by the `Pick` constructor.
Hint: You can use the template below for `genValue`:

```
let rec genValue dEnv = function
    RangeInt(i1,i2) -> (next(i1,i2).ToString(),dEnv)
  | ChoiceString xs -> let idx = next(0,List.length xs - 1)
                       (...,dEnv)
  | StringSeq s -> ...
  | Pair(ds1,ds2) ->
    let (v1',dEnv1) = genValue dEnv ds1
    let (v2',dEnv2) = genValue dEnv1 ds2
    ...
  | Repeat (n,ds) -> ...
  | Pick s -> (pickFromEnv ..., ...)
  | Label (s,ds) ->
    let (v',dEnv') = genValue dEnv ds
    (v', ...)
```