BSWU BFNP **IT University, F2016**

Written exam, Functional Programming
Friday 3 June 2016

Version 1.01 of 2016-06-06

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., `bfnp2016.fsx`. Do not use time on formatting your solution in Word or PDF.

You are welcome to download the accompanying file `jun2016Snippets.txt` from the course homepage in LearnIT: `https://learnit.itu.dk/course/view.php?id=3005303`. 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 *multiset* as an unordered collection of elements that may contain duplicates. A multiset is a generalization of a set (HR page 104). For instance, $\{a, b, b, c\}$ and $\{a, b, c\}$ represents the same set $\{a, b, c\}$ but are different multisets. The *multiplicity* of an element is the number of instances of the element in the multiset. The multiplicity of a is 1, b is 2 and c is 1 for the multiset $\{a, b, b, c\}$. The two multisets $\{a, b, b, c\}$ and $\{a, b, c, b\}$ are equal as the order of the elements is irrelevant.
Formally a multiset may be defined as a tuple (A, m) where A is the underlying set of elements and m a (possibly partial) function $A \to \mathbb{N}_{\geq 1}$ (positive natural numbers), i.e., a partially defined multiplicity function ranging over the elements in A. If $m(a)$ is not defined for some $a \in A$, then the element a is not in the multiset represented by (A, m).
A multiset can be implemented using an F# map, `Map` (HR page 113). The map represents the partially defined multiplicity function over some type $'a$:

```
type Multiset<'a when 'a: comparison> = MSet of Map<'a, int>
```

For instance, the value

```
let ex = MSet (Map.ofList [("a",1);("b",2);("c",1)])
```

represents the multiset $\{a, b, b, c\}$ (also written $\{a, b, c, b\}$ etc.). The map must fulfil the invariant that for all keys in the map, the associated value is ≥ 1. The below value, for the multiset $\{b, b, c\}$, does **not** fulfil this invariant

```
let wrong = MSet (Map.ofList [("a",0);("b",2);("c",1)])
```

Question 1.1

The table below shows the result of rolling a dice 12 times, i.e., we got 1 eye 2 times, 3 eyes 5 times etc.

Num eyes	**1**	**2**	**3**	**4**	**5**	**6**
Result	2	1	5	0	2	2

- Declare a multiset value `diceSet`, representing the result of rolling the dice above, i.e., the multiset
  $\{1, 1, 2, 3, 3, 3, 3, 3, 5, 5, 6, 6\}$.
- What is the type of the value `diceSet`.
- Say you have an F# list of standard mathematical functions, e.g., `[System.Math.Sin; System.Math.Cos; System.Math.Sin]`. Are you able to represent this as a multiset with the representation chosen above, e.g., a value of type `Multiset<float->float>`? Explain your answer.

Question 1.2

- Declare a function

  ```
  newMultiset : unit -> Multiset<'a> when 'a : comparison
  ```

  that returns a new empty multiset.

- Declare a function

```
isEmpty : Multiset<'a> -> bool when 'a : comparison
```

that returns true if a multiset is the empty set. For instance `isEmpty(newMultiset())` returns true.

Question 1.3

- Declare a function

```
add : 'a -> Multiset<'a> -> Multiset<'a> when 'a : comparison
```

where `add k ms` returns the multiset `ms` with the element `k` added. For instance `add "a" ex` returns the value

```
MSet (map [("a", 2); ("b", 2); ("c", 1)])
```

- Declare a function

```
del : 'a -> Multiset<'a> -> Multiset<'a> when 'a : comparison
```

where `del k ms` returns the multiset `ms` with the element `k` deleted, if exists. If the element `k` does not exists, then the multiset `ms` is returned. For instance, `del "c" ex` returns the value

```
MSet (map [("a", 1); ("b", 2)])
```

Remember to fulfil the invariant of the values in the map explained above.

Question 1.4

- Declare a function

```
toList : Multiset<'a> -> 'a list when 'a : comparison
```

where `toList ms` returns the multiset as a list value. For instance `toList ex` may return the list

```
["a"; "b"; "b"; "c"]
```

- Declare a function

```
fromList : 'a list -> Multiset<'a> when 'a : comparison
```

where `fromList xs` returns the multiset corresponding to the elements in the list `xs`. For instance `fromList ["a";"a";"b"]` returns the multiset

```
MSet (map [("a",2);("b",1)])
```

Question 1.5

- Declare a function

```
map : ('a -> 'b) -> Multiset<'a> -> Multiset<'b>
      when 'a : comparison and 'b : comparison
```

  where `map f ms` returns the multiset where the function `f` has been applied on all elements in the multiset `ms`. For instance `map (fun (c:string) -> c.ToUpper()) ex` returns the multiset:

```
MSet (map [("A", 1); ("B", 2); ("C", 1)])
```

- Declare a function

```
fold : ('a -> 'b -> 'a) -> 'a -> Multiset<'b> -> 'a
       when 'b : comparison
```

  where `fold f a ms` returns the accumulated value obtained by applying the accumulator function `f` on all elements in the multiset `ms`, similar to `List.fold`. For instance `fold (fun acc e -> acc+e) "" ex` returns the string `"abbc"`.

Hint: You may use `List.map` and `List.fold`.

Question 1.6

- Declare a function

```
union : Multiset<'a> -> Multiset<'a> -> Multiset<'a>
        when 'a : comparison
```

  where `union ms1 ms2` returns the multiset where all elements in `ms2` have been added to the multiset `ms1`. For instance `union ex ex` returns the multiset:

```
MSet (map [("a", 2); ("b", 4); ("c", 2)])
```

- Declare a function

```
minus : Multiset<'a> -> Multiset<'a> -> Multiset<'a>
        when 'a : comparison
```

  where `minus ms1 ms2` returns the multiset where all elements in `ms2` have been deleted from `ms1`. For instance `minus ex ex` returns the empty multiset `MSet (map[])`.

Hint: You are allowed to use previously declared multiset functions.

Question 2 (20%)

Consider the following F# declarations of `f` and `g`:

```
let rec f n =
  if n < 10 then "f" + g (n+1) else "f"
and g n =
  if n < 10 then "g" + f (n+1) else "g"
```

The types of `f` and `g` are `int -> string`. The expression `f 0` returns the value `"fgfgfgfgfgf"`.

Question 2.1

- The result of evaluating `f 0` is a string that starts and ends with the letter `f`. Describe what arguments `n` to `f`, maybe infinitely many, that will generate a result string that starts with `f` and ends with `f`. The singleton string `"f"` by definition both starts and ends with `f`. Example strings are `"f"`, `"fgf"`, `"fgfgf"`, etc.
- Can you generate the result string `"gfgfgfg"` with the two functions `f` and `g` above?
- Can you for any argument to `f` start an infinite computation?

Question 2.2

The functions `f` and `g` are not tail recursive. Declare tail-recursive variants, `fA` of `f`, and `gA` of `g`, using an accumulating parameter.

Question 3 (20%)

Question 3.1

Consider the following F# declaration:

```
let myFinSeq (n,m) = seq { for i in [0 .. n] do
                             yield! seq { for j in [0 .. m] do yield j }}
```

The type of `myFinSeq` is `int * int -> seq<int>`. The expression `myFinSeq(1,2)` returns the value `seq [0; 1; 2; 0; 1; 2]`.

- Describe the sequence returned by `myFinSeq` when called with arbitrary non-negative integers `n` and `m`.
- Can you for any arguments to `myFinSeq` generate the following value `seq [0; 1; 2; 0; 1; 2; 0; 1]`. Explain your answer.

Question 3.2

Consider the output below of calling a F# function `myFinSeq2 (n,m)` of type `int * int -> seq<int*seq<int>>` with different values of `n` and `m`:

- `myFinSeq2 (0,0)` returns the value `seq [(0, seq [0])]`
- `myFinSeq2 (1,1)` returns the value `seq [(0, seq [0; 1]); (1, seq [0; 1])]`
- `myFinSeq2 (1,2)` returns the value `seq [(0, seq [0; 1; 2]); (1, seq [0; 1; 2])]`
- `myFinSeq2 (2,1)` returns the value `seq [(0, seq [0; 1]); (1, seq [0; 1]); (2, seq [0; 1])]`

Declare the F# function `myFinSeq2` producing the output with the given arguments above. You can assume `n` and `m` are non-negative.

Question 4 (30%)

We shall now consider *CALClite*, a simple spreadsheet system. The spreadsheet is defined by the following type declarations:

```
type Row = int
type Col = char
type CellAddr = Row * Col
type ArithOp = Add | Sub | Mul | Div
type RangeOp = Sum | Count
type CellDef =
    FCst of float
  | SCst of string
  | Ref of CellAddr
  | RangeOp of CellAddr * CellAddr * RangeOp
  | ArithOp of CellDef * ArithOp * CellDef
type CellValue =
    S of string
  | F of float
type Sheet = Map<CellAddr,CellDef>
```

A *cell address* (`CellAddr`) is defined as a *row*- and *column*-index, written `A1` for column `A` and row `1`. We assume columns are in the interval `'A' ... 'Z'` and rows greater or equal to 1. A *cell definition* (`CellDef`) can be a float constant (`FCst`), a string constant (`SCst`), a reference to another cell (`Ref`), an operation on a range (`RangeOp`) or an binary arithmetic operation (`ArithOp`). A range is defined as all cells within the rectangle defined by the top left cell address and the bottom right cell address. The only operations allowed on a range are summing all cells (`Sum`) and counting the number of cells (`Count`). Evaluating a cell definition (`CellDef`) will either result in a string or a float value represented with the type `CellValue`. A spreadsheet, type `Sheet`, is defined as a mapping from cell adresses to cell definitions.
The result of rolling 12 dice from question 1.1 is repeated below:

```
   |        A |        B |        C |        D |        E |        F |        G |        H
---+----------+----------+----------+----------+----------+----------+----------+---------
 1 |    #EYES |        1 |        2 |        3 |        4 |        5 |        6 |    Total
 2 |   RESULT |     2.00 |     1.00 |     5.00 |     0.00 |     2.00 |     2.00 |    12.00
 3 |      PCT |    16.67 |     8.33 |    41.67 |     0.00 |    16.67 |    16.67 |   100.00
```

The cell `H2` is the sum of the range `B2` to `G2`. The cell `B3` is the percentage number of 1's, i.e., `B2/H2*100.0`. The cells `C3`, ..., `H3` are defined similar to `B3`. The definition of the sheet, called `dice`, is as follows:

```
let header = [((1,'A'),SCst "#EYES");((1,'B'),SCst "1");((1,'C'),SCst "2");
              ((1,'D'),SCst "3");((1,'E'),SCst "4");((1,'F'),SCst "5");
              ((1,'G'),SCst "6");((1,'H'),SCst "Total")]
let result = [((2,'A'),SCst "RESULT");((2,'B'),FCst 2.0);((2,'C'),FCst 1.0);
              ((2,'D'),FCst 5.0);((2,'E'),FCst 0.0);((2,'F'),FCst 2.0);
              ((2,'G'),FCst 2.0);((2,'H'),RangeOp((2,'B'),(2,'G'),Sum))]
let calcPct col = ArithOp(FCst 100.0, Mul, ArithOp(Ref(2,col),Div,Ref(2,'H')))
let pct = [((3,'A'),SCst "PCT");((3,'B'),calcPct 'B');((3,'C'),calcPct 'C');
           ((3,'D'),calcPct 'D');((3,'E'),calcPct 'E');((3,'F'),calcPct 'F');
           ((3,'G'),calcPct 'G');((3,'H'),calcPct 'H')]
let dice = Map.ofList (header @ result @ pct)
```

Question 4.1

Declare an F# value, `heights`, of type `Sheet` corresponding to the sheet below:

```
    |        B |        C
----+----------+---------
  4 |     NAME |   HEIGHT
  5 |     Hans |   167.40
  6 |    Trine |   162.30
  7 |    Peter |   179.70
  8 |          |
  9 |     3.00 |   169.80
```

The cells `B4`, `B5`, `B6`, `B7` and `C4` are constant strings. The cells `C5`, `C6` and `C7` are constant floats. The cell `B9` is the count of cells in the range defined by the cell addresses `B5` and `B7`. The cell `C9` is defined as the sum of the range `C5` and `C7` divided by cell `B9`, i.e., the average height of the three persons.

Question 4.2

In order to evaluate a cell definition we need to evaluate range operations `Sum` and `Count` and to evaluate the arithmetic operations `Sum`, `Div`, `Sub` and `Mul`. The operation `Count` works on both float and string values because we are only counting the number of values. The other operations only work on float values. We use the function `getF` of type `CellValue -> float` to ensure a cell value is a float:

```
let getF = function
    F f -> f
  | S s -> failwith "getF: expecting a float but got a string"
```

Declare the following F# functions using `getF`:

1. `evalRangeOp xs op` of type `CellValue list -> RangeOp -> float`. For instance
   - `evalRangeOp [F 33.0; F 32.0] Sum` returns 65.0
   - `evalRangeOp [] Sum` returns 0.0
   - `evalRangeOp [F 23.0; S "Hans"] Sum` throws `System.Exception`
   - `evalRangeOp [F 23.0; S "Hans"] Count` returns 2.0
2. `evalArithOp v1 v2 op` of type `CellValue -> CellValue -> ArithOp -> float`. For instance
   - `evalArithOp (F 33.0) (F 32.0) Sub` returns 1.0
   - `evalArithOp (S "Hans") (F 1.0) Add` throws `System.Exception`

Question 4.3

In order to print a spreadsheet we need to evaluate all cells. This is done by two mutually recursive functions

- `evalValue v sheet` of type `CellDef -> Sheet -> CellValue`
- `evalCell ca sheet` of type `CellAddr -> Sheet -> CellValue`

Declare the two F# functions above. You can assume the sheet contains no cyclic cell definitions. You may use the following template

```
let rec evalValue v sheet =
  match v with
    FCst f -> F f
  | SCst s -> ..
  | Ref ca -> ..
  | RangeOp ((r1,c1),(r2,c2),op) -> ..
  | ArithOp (v1,op,v2) -> ..
and evalCell ca sheet =
  match Map.tryFind ca sheet with
    None -> S "" // We define an empty cell to be the empty string value.
  | Some v -> evalValue v sheet
```

For instance `evalCell (3,'G') dice` returns the cell value `F 16.67`.

Question 4.4

Declare a F# function `ppBoard sheet` of type `Sheet -> string` that returns a string similar to the layout used for `dice` and `heights` above. For instance, `ppBoard dice` returns the string

```
    |        A |        B |        C |        D |        E |        F |        G |        H
----+----------+----------+----------+----------+----------+----------+----------+---------
  1 |    #EYES |        1 |        2 |        3 |        4 |        5 |        6 |    Total
  2 |   RESULT |     2.00 |     1.00 |     5.00 |     0.00 |     2.00 |     2.00 |    12.00
  3 |      PCT |    16.67 |     8.33 |    41.67 |     0.00 |    16.67 |    16.67 |   100.00
```

You must include the actual output from the `ppBoard` function on the `dice` example in order to obtain full points.