BSWU BFNP **IT University, F2015**

Written exam, Functional Programming
Tuesday 2 June 2015

Version 1.10 of 2015-05-27

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., `bfnp2015.fsx`. Do not use time on formatting your solution in Word or PDF.

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 *multimap* as a generalisation of a map in which more than one value may be associated with and returned for a given key. A multimap m from a set A to a set B is a finite subset A' of A together with a function m defined on A' whose type is $m : A' \to P(B)$, where $P(B)$ is the set of all subsets of B, also called the *power set* of B.
A multimap m can be described in tabular form as shown below.

a_0	$B_0 = \{b_{a0_1}, \ldots\}$
a_1	$B_1 = \{b_{a1_1}, \ldots\}$
$\vdots$	$\vdots$
a_n	$B_n = \{b_{an_1}, \ldots\}$

The left column contains the keys $a_0, \ldots, a_n$ of set A' while the right column contains the corresponding values $m(a_0) = B_0, m(a_1) = B_1, \ldots, m(a_n) = B_n$ where B_i are subsets of elements from the set B.
The *map*-concept in F# is explained in HR on page 113.
A multimap can be implemented using an F# map, `Map`, mapping values from the set A' to an F# list representing a subset of B.
We will use the type `multimap<'a,'b>` declared as follows

```
type multimap<'a,'b when 'a: comparison> =
  MMap of Map<'a,list<'b>>
```

The `Map` type expects the elements in A to be comparable. We do not assume any ordering on the elements in B, i.e., the elements from B are just inserted into a list.
For instance, the value

```
let ex = MMap (Map.ofList [("record",[50]);("ordering",[36;46;70])])
```

has type `multimap<string,int>` and represents two entries from the Index in the book HR:

`ordering`	`[36;46;70]`
`record`	`[50]`

You must either declare your own exception or use `failwith` to signal errors in your functions below.

Question 1.1

The students Sine, Hans, Grete and Peter are signed up for a number of courses as shown below:

`Grete`	
`Hans`	TOPS, HOPS
`Peter`	IFFY
`Sine`	HOPS, IFFY, BFNP

- Declare a value `studReg`, of type `multimap<string,string>`, with the course registrations for the four students above.
- Can you declare another value `studReg2`, of type `multimap<string,string>` with the same course registrations, but where the comparison `studReg = studReg2` returns `false`? You must motivate your answer with an example.

Question 1.2

We define the *canonical* representation of a multimap to be the representation where the elements in the value-lists are ordered.

- Declare a function

```
canonical: multimap<'a,'b> -> multimap<'a,'b>
  when 'a : comparison and 'b : comparison
```

where `canonical m` returns the canonical representation of `m`. For instance, the application `canonical studReg` will return a value corresponding to below mapping:

Grete	[]
Hans	["HOPS"';"TOPS"]
Peter	["IFFY"]
Sine	["BFNP"; "HOPS"; "IFFY"]

Hint: The extra type constraint on `'b` is because we now also require an ordering on the value-elements. The F# `Map` implementation already preserves the ordering of the key-elements.

- Declare a function

```
toOrderedList: multimap<'a,'b> -> ('a * 'b list) list
  when 'a : comparison and 'b : comparison
```

For instance, `let studRegOrdered = toOrderedList studReg` defines

```
val studRegOrdered : (string * string list) list =
  [("Grete", []); ("Hans", ["HOPS"; "TOPS"]); ("Peter", ["IFFY"]);
   ("Sine", ["BFNP"; "HOPS"; "IFFY"])]
```

Question 1.3

- Declare a function

```
newMultimap : unit -> multimap<'a,'b> when 'a : comparison
```

that returns a new empty multimap.

- Declare a function

```
sizeMultimap : multimap<'a,'b> -> int * int when 'a : comparison
```

that returns a pair with the number of keys as first component and the total number of value-elements as second component. For instance `let sizeStudReg = sizeMultimap studReg` gives the binding

```
val sizeStudReg : int * int = (4, 6)
```

Question 1.4

- Declare the function `addMultimap k v m` with type

```
addMultimap : 'a -> 'b -> multimap<'a,'b> -> multimap<'a,'b>
  when 'a : comparison and 'b : equality
```

that inserts the value `v` for key `k` in the map `m`. You must make sure that the value `v` does not exists more than once for key `k`. The result does not have to be in canonical order. Verify the following test cases returns true:

```
sizeMultimap (addMultimap "Sine" "BFNP" studReg) = (4,6)
sizeMultimap (addMultimap "Grete" "TIPS" studReg) = (4,7)
sizeMultimap (addMultimap "Pia" "" studReg) = (5,7)
```

- Declare the function `removeMultimap k vOpt m` with type

```
removeMultimap : 'a -> 'b option -> multimap<'a,'b> -> multimap<'a,'b>
  when 'a : comparison and 'b : equality
```

If `vOpt` is `None`, then remove the key `k` and all values v that `k` maps to, from the map. If `vOpt` is `Some v` then only remove the value `v` that the key `k` maps to. In case `vOpt` is `Some v`, then the set of keys in the map `m` is unchanged, that is, the key `k` may end up mapping to the empty list, like `"Grete"` above. Verify the following test cases returns true:

```
sizeMultimap (removeMultimap "Sine" None studReg) = (3,3)
sizeMultimap (removeMultimap "Sine" (Some "PLUR") studReg) = (4,6)
sizeMultimap (removeMultimap "Kenneth" (Some "BLOB") studReg) = (4,6)
sizeMultimap (removeMultimap "Peter" (Some "IFFY") studReg) = (4,5)
```

Question 1.5

Declare a function `mapMultimap f m` with type

```
mapMultimap : ('a -> 'b -> 'c) -> multimap<'a,'b> -> multimap<'a,'c>
  when 'a : comparison
```

that applies a function `f` on all elements in the multimap `m`. The function `f` takes both the key and value element as argument. The function `f` can be applied to the value elements in any order. For instance,

```
mapMultimap (fun k v -> v+"-F2015") studReg
```

returns a new multimap

```
MMap
  (map
     [("Grete", []); ("Hans", ["TOPS-F2015"; "HOPS-F2015"]);
      ("Peter", ["IFFY-F2015"]);
      ("Sine", ["HOPS-F2015"; "IFFY-F2015"; "BFNP-F2015"])])
```

Question 1.6

Declare a function `foldMultimap f s m` of type

```
foldMultimap : ('s -> 'k -> 't -> 's) -> 's -> multimap<'k,'t> -> 's
  when 'k : comparison
```

with similar semantics as `Map.fold`. For instance,

```
foldMultimap (fun acc k v -> String.length v + acc) 0 studReg
```

returns 24.

Question 2 (20%)

Consider the following F# declaration:

```
let rec f i j xs =
  if xs = [] then
    [i*j]
  else
    let (x::xs') = xs
    x*i :: f (i*j) (-1*j) xs'
```

The type of `f` is `int -> int -> int list -> int list`. The expression `f 10 1 [1 .. 9]` returns the value `[10; 20; -30; -40; 50; 60; -70; -80; 90; -10]`.

Question 2.1

- Describe what `f` computes. Your description should focus on what `f` computes, rather than on individual computation steps.
- Explain why the F# compiler reports the warning below when compiling the function `f`.

  > *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`, called `fMatch`, using pattern matching instead of the if-then-else and inner let expressions. Explain why the warning above disappears.

Question 2.2

The function `f` is not tail recursive. Declare a tail-recursive variant, `fA`, of `f`, or `fMatchA` of `fMatch`, using an accumulating parameter.

Question 3 (20%)

Question 3.1

Consider the F# declaration:

```
let myFinSeq n m = seq { for i in [n .. m] do
                           yield [n .. i] }
```

of type `int -> int -> seq<int list>`

- Describe the sequence returned by `myFinSeq` when called with an arbitrary integer `n` and `m`.
- How many times does the number 12 occur in the value returned by `myFinSeq 10 14`?

Question 3.2

Declare a function `myFinSeq2 n m` of type
`myFinSeq2:int->int->seq<int>`
such that `myFinSeq2 n m` produces the same sequence numbers as `myFinSeq`. The difference is that the result sequence contains integer values only, i.e., no list elements. This is reflected in the type of `myFinSeq2`. For instance `myFinSeq2 3 6` returns the value

```
seq [3; 3; 4; 3; 4; 5; 3; 4; 5; 6]
```

Question 3.3

Consider the function `sum` to sum all elements in an integer list together with a big sequence, `seq4000`, and corresponding array, `array4000`.

```
let sum xs = List.fold (fun r x -> r+x) 0 xs
let seq4000 = myFinSeq 10 4000
let array4000 = Array.ofSeq seq4000
```

- How many lists does the value `array4000` contain?
- An array containing the sums of all lists can be computed sequentially

  ```
  let sums = Array.map sum array4000
  ```

  Give an alternative declaration of `sums` that computes the sums of the lists in parallel.

  Hint: You may use the `Array.Parallel` library as explained in Section 13.6 in the book HR.

Question 4 (30%)

We shall now consider *JSONlite*, a small subset of JSON (JavaScript Object Notation, `www.json.org`). The syntax of *JSONlite* is illustrated with the example below. The example contains the JSON *records*, `"Person1"`, `"Person2"` and `"Address"`. Each person record has a name, `"Name"` and an address, `"Address"`. The address record is the same for both persons. To avoid duplicating the address record we define a *label*, `Addr1`, pointing, `->`, at the address record. For the person, `Person2`, we can *reference*, `Ref`, the address record, `Addr1`. The below string representation is used in Question 4.2.

```
{
  "Person1" : {
    "Name" : "Hans Pedersen",
    "Address" : Addr1 -> {
      "Street" : "Hansedalen",
      "HouseNo" : "27"
    }
  },
  "Person2" : {
    "Name" : "Pia Pedersen",
    "Address" : ref Addr1
  }
}
```

Consider the F# declarations below. The declaration, `persons`, corresponds to the string representation above.

```
type JSONlite =
    Object of list<string * Value>
and Value =
    String of string
  | Record of JSONlite
  | Label of string * Value
  | Ref of string

let address = Object [("Street", String "Hansedalen");
                      ("HouseNo", String "27")]
let person1 = Object [("Name", String "Hans Pedersen");
                      ("Address", Label ("Addr1", Record address))]
let person2 = Object [("Name", String "Pia Pedersen");
                      ("Address", Ref "Addr1")]
let persons = Object [("Person1", Record person1);
                      ("Person2", Record person2)]
```

Question 4.1

Declare an F# value, `student`, of type `JSONlite` corresponding to the example below:

```
{
  "Name" : "Per Simonsen",
  "Field" : "BSWU",
  "Course" : {
    "BFNP" : "10",
    "BPRD" : "7"
  }
}
```

Question 4.2

Declare a function

`ppJSONlite:JSONlite->string`

that returns a string representation of the JSONlite value. For instance, `ppJSONlite persons`, returns a string corresponding to the first example above.

Hint: You can make use of the following template:

```
let nl = System.Environment.NewLine   // New line
let space n = String.replicate n " "  // Make n spaces
let ppQuote s = "\"" + s + "\""       // Put quotes around string s

let ppJSONlite json =
  let rec ppValue' indent = function
      String s -> ...
    | ...
  and ppJSONlite' indent = function
    Object xs -> ...
  ppJSONlite' 0 json
```

The variable `indent` is used to control indentation. You may find the function `String.concat sep xs` useful to concatenate a list of strings, `xs`, with a separator, `sep`.

Question 4.3

Declare a function

`buildEnv: JSONlite -> Map<string,Value>`

such that `buildEnv json` is an environment mapping labels declared in `json` to the corresponding JSON values. The function can assume that no label is defined multiple times. The application `buildEnv person1` returns the following environment:

```
map [("Addr1", Record (Object [("Street", String "Hansedalen");
                               ("HouseNo", String "27")]))]
```

Hint: You can make use of the following template:

```
let buildEnv json =
  let rec buildEnvValue env = function
      String s -> ...
    | ...
  and buildEnvJSONlite env = function
    Object xs -> ...
  buildEnvJSONlite Map.empty json
```

Question 4.4

Labels and references are not part of the JSON specification. Declare a function

`expandRef JSONlite->JSONlite`

that replaces all references to JSON values with the values themselves. You do this using the environment returned by `buildEnv`. The function can assume all references exists. For instance, if `persons` is bound to the value

```
Object
  [("Person1",...);
```

```
   ("Person2",
     Record (Object [("Name", String "Pia Pedersen");
                     ("Address", Ref "Addr1")]))]
```

then `expandRef persons` returns the following value:

```
Object
  [("Person1", ...);
   ("Person2",
    Record (Object [("Name", String "Pia Pedersen");
                    ("Address", Record (Object [("Street", String "Hansedalen");
                                                ("HouseNo", String "27")]))]))]
```

where `Ref "Addr"` has been expanded to `Record (Object [("Street",....`