BSWU BFNP **IT University, F2014**

Written exam, Functional Programming
Wednesday 11 June 2014

Version 1.00 of 2014-06-02

These exam questions comprise 8 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 other 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.

If you hand-in electronically, then hand-in one ASCII file only, e.g., `bfnp2014.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 an *ordered list* as a generalisation of a list where you can add and remove elements at both ends. An ordered list, *l*, with *n* elements is written $e_1, \ldots, e_n$. We use a comma to separate the elements in the ordered list.

An ordered list is implemented using two F# lists; a *front* list: $e_1::e_2::\ldots$ and a *rear* list: $e_n::e_{n-1}::\ldots$. The idea is to have constant time support for adding elements to the ordered list in both ends. Removing elements from the ordered list may require linear time.

The last element added to the end of the ordered list is the first element in the *rear* list. We require the invariant, that *front* `@ List.rev` *rear* represents the ordered list of elements. You can assume, that

all ordered lists occurring in arguments below satisfy this invariant, and all functions must preserve the invariant.
We will use the type `OrderedList<'a>` declared as follows

```
type OrderedList<'a when 'a : equality> =
  {front: 'a list;
   rear: 'a list}
```

For instance, the value

```
let ex = {front = ['x']; rear = ['z';'y']}
```

has type `OrderedList<char>` and represents the ordered list `'x'`, `'y'`, `'z'`.
You must either declare your own exception or use `failwith` to signal errors in your functions below.

Question 1.1

The declaration below represents an ordered list with the following elements: "Hans", "Brian", "Gudrun".

```
let ol1 = {front = ["Hans"; "Brian"; "Gudrun"]; rear = []}
```

- Declare two values, `ol2` and `ol3`, representing the same ordered list but with the rear list being non-empty.
- How many representations exists with the three elements in the given order that fulfils the invariant?

Question 1.2

We define the *canonical* representation of an ordered list to be the representation where the *rear* list is empty.

- Declare a function `canonical:OrderedList<'a>->OrderedList<'a>`, where `canonical ol` returns the canonical representation of `ol`.
- Declare a function `toList:OrderedList<'a'>->List<'a>`, that returns the list of elements. For instance `toList ex` returns the list `['x';'y';'z']`.

Question 1.3

- Declare a function `newOL:unit->OrderedList<'a>`, that returns a new empty ordered list.
- Declare a function `isEmpty:OrderedList<'a>->bool`, that returns true if the ordered list is empty and otherwise false. For instance, `isEmpty (newOL())` returns true and `isEmpty ex` returns false.

Question 1.4

A stack, LIFO, is a special version of an ordered list where you can only add and remove elements from the front of the list.

- Declare the function `addFront:'a->OrderedList<'a>->OrderedList<'a>`, that adds an element to the front of the list. For instance `addFront 'w' ex` returns the ordered list `'w','x','y','z'`
- Declare the function `removeFront:OrderedList<'a>->'a*OrderedList<'a>`, that removes the first element in the ordered list and returns both the first element and the new list. For instance `removeFront ex` returns a pair with the first element being `'x'` and the second element being the ordered list `'y','z'`.
- Declare the function `peekFront:OrderedList<'a>->'a`, that returns the first element in the ordered list, without removing it from the list. For instance `peekFront ex` returns the element `'x'`.

Question 1.5

Declare the function
`append:OrderedList<'a>->OrderedList<'a>->OrderedList<'a>`,
that concatenates two ordered lists. For instance, `append ex ex` returns the ordered list `'x'`, `'y'`, `'z','x','y','z'`.

Question 1.6

Declare a function
`map:('a->'b)->OrderedList<'a>->OrderedList<'b>`,
that applies a function on all elements in the ordered list. The function can be applied on the elements in any order. For instance, `map (fun e -> e.ToString()) ex` returns a new ordered list `"x"`, `"y"`, `"z"` of type `OrderedList<string>`.

Question 1.7

Declare a function
`fold:('State->'T->'State)->'State->OrderedList<'T>->'State`,
with similar semantics as `List.fold`. For instance, `fold (fun acc e -> acc + e.ToString()) "" ex` returns the string `"xyz"`.
Hint: You can use `toList` from above.

Question 1.8

Declare a function
`multiplicity:OrderedList<'a>->Map<'a,int>`,
that returns a mapping from the elements in the ordered list to the number of times the elements are represented in the ordered list. For instance, `multiplicity (addFront 'x' ex)` returns the map where `'x'` is mapped to 2, `'y'` is mapped to 1 and `'z'` is mapped to 1.
Hint: You can use `fold` from above.

Question 2 (20 %)

Consider the following F# declaration:

```
let rec f i = function
    [] -> [i]
  | x::xs -> i+x :: f (i+1) xs
```

The type of `f` is `int -> int list -> int list`. The expression `f 10 [0;1;2;3]` returns the value `[10;12;14;16;14]`.

Question 2.1

- Describe what `f` computes. Your description should focus on what `f` computes, rather than on individual computation steps.
- Explain whether the result of calling `f`, with any input, can ever be the empty list?
- Explain whether the function `f`, with any input, can ever go into an infinite loop?

Question 2.2

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

Question 2.3

Declare a continuation-based tail-recursive variant, `fC`, of `f`.

Question 3 (20 %)

Question 3.1

Consider the F# declaration:

```
let myFinSeq n M = Seq.map (fun m -> n+n*m) [0..M]
```

Describe the sequence returned by `myFinSeq` when called with an arbitrary integer `n` and `M`.

Question 3.2

Declare the infinite sequence

```
mySeq:int->seq<int>
```

such that `mySeq n` produces the infinite sequence `n+n*i` for `i` $>= 0$. The identifier `i` represents the index of the element in the sequence.
Hint: You can use `Seq.initInfinite`.

Question 3.3

Declare the finite sequence

```
multTable N M
```

to return the element triples `(n,m,n*m)` where `n` $\in [0,\ldots,N]$ and `m` $\in [0,\ldots,M]$.

Question 3.4

Declare the finite sequence

```
ppMultTable:int->int->seq<string>
```

to return the sequence "<n> * <m> is <n*m>" for `n` $\in$ [0,. . .,N] and `m` $\in$ [0,. . .,M], using the sequence `multTable`. The notation <*n*> denotes the textual representation of the integer *n*.
For instance, `Seq.take 4 (ppMultTable 10 10)` returns the value

```
seq ["0 * 0 is 0"; "0 * 1 is 0"; "0 * 2 is 0"; "0 * 3 is 0"]
```

Question 4 (30 %)

We shall now consider *MousePlot*, a small library to describe drawings. In *MousePlot* you have the following *operations*:

```
type opr = MovePenUp
         | MovePenDown
         | TurnEast
         | TurnWest
         | TurnNorth
         | TurnSouth
         | Step
```

Imagine a *mouse* that can turn and step in four directions. The *mouse* has a pen that can be up or down. If the pen is down, a dot is placed on a media, e.g., a piece of paper.
A plot consists of either one *operation* or a sequence of *operations* described by the type *plot*:

```
type plot = Opr of opr
          | Seq of plot * plot
```

A simple rectangle can be declared as follows:

```
let side = Seq(Opr MovePenDown, Seq(Opr Step, Seq(Opr Step, Opr Step)))
let rect = Seq(Seq(Opr TurnEast, side),
             Seq(Opr TurnNorth, Seq(side,
               Seq(Opr TurnWest, Seq(side,
                 Seq(Opr TurnSouth, side))))))
```

The rectangle consists of four sides. Each side is defined as three steps in one of the four directions.

Question 4.1

Consider the example `rect` above.

- Write a function `ppOpr: opr -> string` that returns a pretty printed version of the argument operation. For instance, `ppOpr MovePenUp` returns the string `"MovePenUp"`.
- Write a function `ppOprPlot: plot -> string` that returns a pretty printed version of the argument plot. The returned string must use the notation opr_1 => opr_2 to show that opr_1 comes before opr_2. For instance `ppOprPlot rect` returns the string `"TurnEast => MovePenDown => Step => Step => Step => TurnNorth => MovePenDown => Step => Step => Step => TurnWest => MovePenDown => Step => Step => Step => TurnSouth => MovePenDown => Step => Step => Step"`

Question 4.2

In order to draw a plot, we need to define the *mouse* movements in more detail.

- The *mouse* can turn in four directions: East, West, North and South.
- The *mouse* steps around in a coordinate system defined as:

Step in direction	Δx	Δy
East	+1	0
West	-1	0
North	0	+1
South.	0	-1

For instance, a Step in direction East will increase the x coordinate with one and leave y unchanged.

- The *mouse* stays at the same coordinate when it turns.
- The *mouse* has a pen that can be up (PenUp) or down (PenDown). When down, a dot is placed on a media, e.g., a piece of paper.
- The operation `MovePenDown` makes the *mouse* place a dot at the coordinate where the mouse is located.
- If the pen is down (PenDown), then the operation `Step` will make the *mouse* place a dot at the coordinate it is targeting as defined above.
- Initially the pen is up (PenUp), placed at coordinate (0,0) and heading in the direction East.

Given a plot, the task is to simulate mouse movements and thereby calculate the coordinates where the pen is down, i.e., where the mouse place dots. The simulation uses a *state* containing three components: the *direction* the mouse is heading, the *current coordinate* and the state of the *pen*, i.e., PenUp or PenDown. The *state* is defined as follows:

```
type dir = North
         | South
         | West
         | East
type pen = PenUp
         | PenDown
type coord = int * int
type state = coord * dir * pen
```

For instance, the initial state is defined as

```
let initialState = ((0,0),East,PenUp)
```

The state `((2,-1),South,PenDown)` represents the mouse at coordinate (2,-1) heading South with the pen in down position.

- Declare a function `goStep:state->state` that given a state returns a new state where the mouse has moved one step in the direction given by the state. For instance `goStep initialState` returns the value `((1,0),East,PenUp)`
- Declare a function `addDot:state->coord list->opr->coord list*state`. Given a state, a list of coordinates and an operation, the function performs the operation and if the mouse places a dot, i.e., the pen is down, then adds the coordinate to the list of coordinates. For instance,

```
let (coords1,s1) = addDot initialState [] MovePenDown
let (coords2,s2) = addDot s1 coords1 Step
```

returns

```
val s1 : (int * int) * dir * pen = ((0, 0), East, PenDown)
val coords1 : (int * int) list = [(0, 0)]
val s2 : (int * int) * dir * pen = ((1, 0), East, PenDown)
val coords2 : (int * int) list = [(1, 0); (0, 0)]
```

- Declare a function `dotCoords:plot->coord list`. Given a plot, the function returns the coordinates where the mouse has placed a dot on the media. For instance

```
let coords = dotCoords rect
```

returns

```
val coords : (int * int) list =
  [(0, 0); (0, 1); (0, 2); (0, 3); (0, 3); (1, 3); (2, 3); (3, 3); (3, 3);
   (3, 2); (3, 1); (3, 0); (3, 0); (2, 0); (1, 0); (0, 0)]
```

- Declare a function `uniqueDotCoords:plot->Set<coord>` similar to `dotCoords` except that the coordinates are now returned as a set, i.e., no duplicate coordinates. For instance

```
let coordSet = uniqueDotCoords rect
```

returns

```
val coordSet : Set<int * int> =
  set
    [(0, 0); (0, 1); (0, 2); (0, 3); (1, 0); (1, 3); (2, 0); (2, 3); (3, 0);
     ...]
```

Question 4.3

To shorten the notation used when specifying plots, we shall now extend the type `plot` with a binary overloaded operator, `(+):plot*plot->plot`, to put two plots in sequence to each other. For instance the example rectangle can be written

```
let side2 = Opr MovePenDown + Opr Step + Opr Step + Opr Step
let rect2 = Opr TurnEast + side2 + Opr TurnNorth + side2 +
            Opr TurnWest + side2 + Opr TurnSouth + side2
```

Extend the type `plot`, using *type augmentation*, with the overloaded operator + such that the above can be declared.