Written exam, Functional Programming

Monday MAY 25, 2020

- The exam duration is five hours
- There are four 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 may **NOT** copy code found online that you yourself have not written and hand that in as your solution. To be safe stick to resources such as *F# for fun and profit* or Microsoft's documentation of .NET and the F# language.
- You are (unless otherwise instructed) 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 (unless otherwise instructed) use that function in subsequent subquestions, even if you have not managed to define it. Providing the signature of the missing function will help in such cases.
- If a subquestion requires you to define a particular function, then you may (unless otherwise instructed) 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 asked for.
- Unless explicitly stated you are required to provide functional solutions, and solutions with side effects will not be considered. The one exception to this rule concerns parallelism as `Async.Parallel` returns the results of the individual processes in an array and these results may be used.
- You are required to use the provided code project `FPExam2020` as a basis for your submission and you should only hand in the `Exam.fs` file (no other file). The project includes everything you need to run as an independent project, but you may also use the F# top loop. See the `README` for details. Any helper functions that we provide in `Exam.fs` file may also be part of your submission.
- Most functions that you need to write are present in the code skeleton. If an assignment asks that you write a function `isEven : int -> bool`, for instance, then there is nearly always a corresponding `let isEven _ = failwith "Not implemented"` in the source file. You may change these functions (changing a `let` to a `let rec` for instance) as long as their signatures correspond to those given in the assignment. In this case that could be `let isEven x = x % 2 = 0`. Be wary of polymorphic variables as notation sometimes differs and some IDEs, for instance, will write `MyType<'a when 'a : equality>` while others may write `MyType<'a> when 'a : equality`. These are identical.
- After the exam is done we will be doing a random check of around 20% of the students. You will get to know promptly at the end of the exam if you have been chosen for this. If so, you must be present in the provided Zoom room 30 minutes after the exam, and up until 2 hours after the exam, or until told by a teacher that you are allowed to leave.

You MUST include explanations and comments to support your solutions for the questions that require them. 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. This includes using solutions or code found online.

Your solution MUST compile. We reserve the right to fail any submission that does not meet this requirement.

1: Insertion Sort (25%)

Insertion sort is a simple sorting algorithm that works by inserting elements into an already sorted list. On a high level the algorithm looks like this

1. The empty list is sorted
2. For non-empty lists, recursively sort the tail of the list and then insert the head of the list in the result (a sorted list)

There are already sorting algorithms in the standard library. You may of course **not** use these. Other generic functions like `rev`, `@`, `::`, `fold`, or `map` may, however, come in handy.

Remember your complexity theory. Insertion sort is a quadratic algorithm and any solution that has a higher complexity will not net full points. Do not, for instance, append single elements to the end of lists.

Question 1.1

Create a recursive (not tail-recursive) function `insert : 'a -> 'a list -> 'a list` that given an element `x` and a sorted list `lst` inserts `x` into the correct spot in `lst`. Your function **does not** have to do anything sensibile if `lst` is not sorted so just assume that `lst` is sorted.

Create a recursive (not tail-recursive) function `insertionSort : 'a list -> 'a list` that given a list `lst` sorts the list using insertion sort.

Examples:

```
- insert true []
> val it : bool list = [true]

- insert 5 [1; 3; 8; 9]
> val it : int list = [1; 3; 5; 8; 9]

- insert 'c' ['a'; 'd'; 'e']
> val it : char list = ['a'; 'c'; d'; 'e']

- insertionSort [5; 3; 1; 8; 9]
> val it : int list = [1; 3; 5; 8; 9]
```

```
- insertionSort ['o'; 'l'; 'H'; 'e'; 'l']
> val it : char list = ['H'; 'e'; 'l'; 'l'; 'o']
```

Question 1.2

Create a tail recursive function `insertTail : 'a -> 'a list -> 'a list`, using an accumulator, that behaves the same as `insert` but that does not overflow the stack.

Create a tail recursive function `insertionSortTail : 'a list -> 'a list`, using an accumulator, that behaves the same as `insertionSort` but that does not overflow the stack.

Question 1.3

Higher-order functions in the `List` library like `map`, `fold`, or `filter` can often be used instead of recursive functions.

Why are the higher-order functions from the `List` library not a good fit to implement `insert`? If you are unsure, try writing `insert` using a higher-order function and see where things start to get troublesome.

Create a non-recursive function `insertionSort2 : 'a list -> 'a list`, that behaves the same way as `insertionSort` using higher-order functions (which themselves are recursive). You should use `insert` or preferably `insertTail` for the insertion.

Question 1.4

For this assignment we will have lists that are sorted with respect to some function. A list `[x1; ...; xn]` is sorted with respect to a function `f` if `[f x1; ...; f xn]` is sorted. This is useful for sorting lists that do not have a clear ordering or when you want to sort with respect to somthing else than the default ordering - sorting a list of strings depending on their length rather than in alphabetical order, for instance.

Create a function `insertBy : ('a -> 'b) -> 'a -> 'a list -> 'a list` that given a function `f`, an element `x`, and a list `lst` that is sorted with respect to `f` , inserts `x` into `lst` such that the resulting list is sorted with respect to `f`.

Create a function `insertionSortBy : ('a -> 'b) -> 'a list -> 'a list` that given a function `f` and a list `lst` sorts `lst` with respect to `f`.

Examples:

```
- insertBy String.length "abc" ["q"; "bb"; "lbcd"]
> val it : string list = ["q"; "bb"; "abc"; "lbcd"]

- insertionSortBy String.length ["bb"; "lbcd"; "q"; "abc"])
> val it : string list = ["q"; "bb"; "abc"; "lbcd"]
```

2: Code Comprehension (25%)

Consider and run the following three functions

```
let rec foo x =
    function
    | y :: ys when x = y -> ys
    | y :: ys            -> y :: (foo x ys)

let rec bar x =
    function
    | []        -> []
    | xs :: xss -> (x :: xs) :: bar x xss

let rec baz =
    function
    | [] -> []
    | [x] -> [[x]]
    | xs  ->
        let rec aux =
            function
            | []      -> []
            | y :: ys -> ((foo y >> baz >> bar y) xs) @ (aux ys)
        aux xs
```

Question 2.1

- What are the types of functions `foo`, `bar`, and `baz`?
- What do functions `foo`, `bar`, and `baz` do? Focus on what they do rather than how they do it.
- What would be appropriate names for functions `foo`, `bar`, and `baz`?

Question 2.2

The function `foo` generates a warning during compilation: `Warning: Incomplete pattern matches on this expression.`.

- Why does this happen, and where?
- For these particular three functions will this incomplete pattern match ever cause problems for any possible execution of `baz`? If yes, why; if no, why not.
- Write a function `foo2` that does exactly the same thing as `foo` except that it does not have this problem.

Question 2.3

In the function `baz` there is a sub expression `foo y >> baz >> bar y`

- What is the type of this expression

- What does it do? Focus on what it does rather than how it does it.

Question 2.4

Write a non-recursive version of `bar` (using higher-order functions) called `bar2` that does exactly the same thing as `bar`.

Question 2.5

The internal function `aux` in `baz` can be replaced with a single call to a higher-order function. Write a version of `baz` called `baz2` that does exactly the same thing as `baz` and that does not contain this internal function. Creating other external or internal functions is permissible as long as these functions are used as arguments to higher-order functions like `map` or `fold`.

Hint: The only thing that needs to change is the final case in the pattern match of `baz`, all other cases remain the same.

Question 2.6

The function `foo` is not tail-recursive. Why? To make a compelling argument you should evaluate a function call of the function, similarly to what is done in Chapter 1.4 of HR, and reason about that evaluation. You need to make clear what aspects of the evaluation tell you that the funciton is not tail recursive. Keep in mind that all steps in an evaluation chain must evaluate to the same value (`(5 + 4) * 3 --> 9 * 3 --> 27`, for instance).

Create a tail-recursive version of `foo` called `fooTail`, using continuations, that does exactly the same thing as `foo` except that it does not generate any warnings.

3: Rock Paper Scissors (25%)

Rock papers scissors is a classic game in which two players at the same time use one hand to represent a rock, a paper, or a pair of scissors where:

- Rock beats scissors
- Scissors beats paper
- Paper beats rock

Question 3.1

Create an enumeration type `shape` to represent these three shapes and another enumeration type `result` for the result of a game which can either be that player one wins, that player two wins, or that you have a draw. Using types like strings or integers works but will not give you full credit.

Create a function `rps : shape -> shape -> result` that given two shapes `s1` (for player one) and `s2` (for player 2) returns who won the game or if the game is a draw.

Examples:

Your output will vary depending on your types, but here we write `<rock>`, for instance, to represent your rock.

```
- rps <rock> <paper>
> val it : result = <playerTwoWin>

- rps <paper> <rock>
> val it : result = <playerOneWin>

- rps <scissors> <scissors>
> val it : result = <draw>
```

Question 3.2

Believe it or not people actully compete in this game, and there is even a robot that will consistently beat any human. To play a game you need a strategy and while humans use psychological tricks to increase their chances to win, we will use functions of type `(shape * shape) list -> shape` which given a list of previously played moves `moves`, where

- the latest move is at the head of `moves`
- for any element `(s1, s2)` in `moves` the shape `s1` was played by you and shape `s2` was played by your opponent (regardless of which one is the first or the second player)

```
type strategy = (shape * shape) list -> shape
```

Create a strategy `parrot : shape -> strategy` that given a starting shape `s`, and a list of previous moves `moves` will always copy what your opponent last played, and play `s` if this is the first round.

Examples:

```
- parrot <rock> []
> val it : shape = <rock>

- parrot <paper> [(<rock>, <scissors>); ...]
> val it : shape = <scissors>
```

Create a strategy `beatingStrat : strategy` that given list of previous moves `moves` plays the shape that beats whatever your oponent has played the most. In the case of a tie, the priority order of what to play is `<rock> -> <paper> -> <scissors>` so play the first shape of that sequence that beats whatever the opponent has played the most.

Hint: There are several functions in the `List` library that will make this exercise a lot simpler. If you find yourself writing lots of auxiliary functions you are most likely re-inventing the wheel.

Examples:

```
- beatingStrat []
> val it : shape = <rock>

- beatingStrat [(<scissors>, <paper>)]
> val it : shape = <scissors>

- beatingStrat [(<scissors>, <paper>); (<rock>, <rock>)]
> val it : shape = <paper>
```

Create a function `roundRobin : shape list -> strategy` that given a non-empty list of shapes `shapes` and a list of moves `moves` plays the next element from `shapes` every time the function is called regardless of what has been played prior. When the list is empty it starts from the beginning again. Just using partial application to call this function with a shape list will return a strategy function of the correct type (examples will make this clear).

Your function does not have to handle the case where `shapes` is empty, but you may wish to include a `failwith` to surpress warnings if you get them.

For **this function only** you are allowed to use mutable variables but these should be kept internal to the function and not be exposed to the outside.

Examples:

```
- let strat = roundRobin [<paper>; <rock>; <rock>]
> val strat : (shape * shape) list -> shape

- strat <any move>
> val it : shape = <paper>

- strat <any move>
> val it : shape = <rock>

- strat <any move>
> val it : shape = <rock>

- strat <any move>
> val it : shape = <paper>
.
.
.
```

Question 3.3

Read to the end of this question before you start implementing.

We will be writing a function `bestOutOf : strategy -> strategy -> (int * int) seq` that given two strategies `strat1` and `strat2` for player one and player two respectively returns a sequence of pairs of integers where the nth pair `(p1, p2)` of the sequence means that after `n` rounds played player one has won `p1` times and player two has won `p2` times. For each round the moves played are updated and sent to `strat1` and `strat2` to generate the next move. Remember that `strategy = (shape * shape) list -> shape` and that each strategy assumes that its own moves are the first element of the tuples in the list - you will have to keep separate records for the separate players.

It may be tempting to generate a function that calculates your point tuple after `n` rounds and then use `Seq.initInfinite` to generate the sequence. This is not a good solution. Why?

Create the `bestOutOf` function using the `Seq.unfold` function from the standard library. This is by far the simplest solution so take a minute to look up what `Seq.unfold` does. Using `Seq.initInfinite` will not give you any points.

Examples:

```
- bestOutOf <any strategy> <any strategy> |> Seq.item 0
> val it : int * int = (0, 0)

- bestOutOf (fun _ -> <rock>) (fun _ -> <rock>)) |> Seq.item 5
> val it : int * int = (0, 0)

- bestOutOf (fun _ -> <rock>) (fun _ -> <paper>) |> Seq.item 5
> val it : int * int = (0, 5)

> bestOutOf (fun _ -> <rock>) (fun _ -> <scissors>) |> Seq.item 5
> val it : int * int = (5, 0)

- bestOutOf (roundRobin [<rock>; <paper>; <scissors>]) (parrot <rock>) |>
Seq.item 50
> val it : int * int = (49, 0)

- bestOutOf (roundRobin [<rock>; <rock>; <scissors>]) beatingStrat |> Seq.item
50
> val it : int * int = (16, 33)
```

Question 3.4

Create a function `playTournament : int -> strategy list -> int option` that given a number of rounds `numRounds` list of player with strategies `[strat_1; strat_2; ...; strat_n]` plays a tournament by pairing up the players from the list so that `strat_1` plays `numRounds` rounds against `strat_2`, `strat_3` against `strat_4` and so on until `strat_(n-1)` against `strat_n`. If `n` is an odd number then player `n` sits out the round. Collect the winners in a list and repeat the process, if a game is a draw then both players are eliminated. The function returns an option with the index of the winning player of the final in the original list, and `None` if the final match is a draw.

Note: It is possible, if you have an odd number of players, for the final player to win without ever playing a match if all others play draws.

For full credit this function must be parallelised so that all pairs play against each other in parallel in every level of the competition. So, for instance, if you have 16 players then in the first level 8 pairs play in parallel, in the second level four pairs play in parallel, and so on (assuming that there are no draws).

```
let rock     : strategy = fun _ -> <rock>
let paper    : strategy = fun _ -> <paper>
let scissors : strategy = fun _ -> <scissors>

- playTournament 5 [rock]
> val it : int option = Some 0

- playTournament 5 [rock; paper; scissors]
> val it : int option = Some 2

- playTournament 5 [for i in 1..1000 do yield rock; yield paper; yield
scissors]
> val it : int option = Some 2013
```

4: Reverse Polish Notation (25%)

Reverse Polish notation (RPN) is a means to write arithmetic. In RPN operators follow their operands so in stead of writing `5 + 3`, we write `5 3 +`. While this is harder to read the main advantage is that parentheses are never needed (as long as we know how many arguments each operator takes) which is also why it was used for early calculators.

For instance, the expression `(5 - 3) * 2` is written `5 3 - 2 *` while the expression `5 - (3 * 2)` is written `5 3 2 * -`. The algorithm works using a stack where numbers are pushed on the stack, operators pop the required number of operands from the stack, calculate a result, and push the result to the stack. When the calculation is done, the result will be on the top of the stack.

We will use the notation `{a, b, c}` to represent a stack containing `a`, `b`, and `c`, where `a` is the top of the stack.

```
step     stack           expression
-----------------------------------
0        {}              5 3 2 * -
1        {5}             3 2 * -
2        {3, 5}          2 * -
3        {2, 3, 5}       * -
4        {6, 5}          -
5        {-1}
```

Note that `3 - 2` is written `3 2 -` and not `2 3 -`.

Question 4.1

- Create a type `stack` that is used to hold a stack of integers. It is ok (but not at all required) if your type is more general than that but it must be able to model a stack of integers.
- Create a value `emptyStack` which denotes the empty stack

Question 4.2

For this assignment we will be using a state monad to hide the stack. The state monad you will be working on is very similar to the one that you used for Assignment 6, but the state is much simpler (the stack from Q4.1).

```
type SM<'a> = S of (stack -> ('a * stack) option)

let ret x = S (fun s -> Some (x, s))
let fail  = S (fun _ -> None)
let bind f (S a) : SM<'b> =
    S (fun s ->
        match a s with
        | Some (x, s') ->
            let (S g) = f x
            g s'
        | None -> None)

let (>>=) x f = bind f x
let (>>>=) x y = x >>= (fun _ -> y)

let evalSM (S f) = f emptyStack
```

Create functions `push : int -> SM<unit>` and `pop : SM<int>` where `push` takes an integer `x` and pushes `x` on the stack, and `pop` pops the top element off the stack and returns it. Popping an element off an empty stack should result in failure (monadic `fail`, **not** `failwith`).

Important: You cannot use monadic operators for `push` or `pop` as you must break the abstraction of the state monad to implement these functions, but we include them here for debugging purposes, and you will need them for Q4.4.

Examples:

Remember the stack notation that we use. Your output here will vary depending on how you have implemented your stack.

```
- push 5 >>>= push 6 >>>= pop |> evalSM
> val it : (int * stack) option = Some (6, {5})

- pop |> evalSM
> val it : (int * stack) option = None
```

Question 4.3

From this point onwards we will be reading and writing from and to the terminal. This, unfortunately, does not play nicely with the interactive mode so run this in the provided project.

We provide an extensions to our state monad with functions for reading and writing from the terminal (you do **not** have to write these). The function `read : SM<string option>` reads a word from the terminal, until separated by a white space and returns `Some str`, where `str` was the word input from the console, and `None` if no symbols are input, but only an arbitrary amount of white spaces followed by a newline. The function `write : string -> SM<unit>` takes a string `str`, returns `()` but prints `str` on the screen.

Examples:

Here `m>` represents a line in your main function and `<-` indicates input to and `->` output from your program (they wont be shown). We use `ignore` to surpress the warning that the return value is discarded.

```
m> write "Hello!!!" |> evalSM |> ignore
-> Hello!!!        <this line is output on the screen>

m> push 42 >>>= pop >>= (fun x -> write (string x)) |> evalSM |> ignore
-> 42              <this line is output on the screen>

The following example has a non-exhaustive pattern matching and the program
will
crash if non-integers are input.
m> read >>= (fun (Some a1) -> read >>= (fun (Some a2) -> write (string (int a1
+ int a2))))
<- 40 2            <this line is input by the user ending with a newline
                    or trailing whitespace>
-> 42              <this line is output on the screen>
```

The code for the functions is the following

```
let write str : SM<unit> = S (fun s -> printf "%s" str; Some ((), s))

let read =
    let rec aux acc =
        match System.Console.Read() |> char with
        | '\n' when acc = [] -> None
        | c    when System.Char.IsWhiteSpace c ->
            acc |> List.fold (fun strAcc ch -> (string ch) + strAcc) "" |>
Some
        | c -> aux (c :: acc)

    S (fun s -> Some (aux [], s))
```

Consider the definition of `write` There is a reason that the definition is `S (fun s -> printf "%s" str; Some ((), s))` and not just `ret (printf "%s" str)`. For a similar reason, in `read`, we write `S (fun s -> Some (aux [], s))` and not `ret (aux [])`. What is the problem with using `ret` in both of these cases?

Question 4.4

For this final assignment you may, if you want to, use computational expressions in which case you will need the following definitions.

```
type StateBuilder() =

    member this.Bind(f, x)       = bind x f
    member this.Return(x)        = ret x
    member this.ReturnFrom(x)  = x
    member this.Combine(a, b)  = a >>= (fun _ -> b)

let state = new StateBuilder()
```

You may also solve the assignment using monadic operators, but you may not break the abstraction of the state monad, and you may not use any other forms of IO than the ones provided in Q4.3 (look at the examples for guidance)

Create a function `calculateRPN : unit -> unit` that reads a formula containing integers, addition (+), multiplication (*) and subtraction (-), in reverse Polish notation from the terminal, calculates its result and outputs it on the screen.

Your program should fail (again monadic fail, and not `failWith`) if

- The input is ilformed (`+ + *` for instance). One exception to this is that `5 4 3 +` is allowed to return `7` even though there is a `5` left on the stack.
- The input contains strings which are neither numbers, whitespaces, or the allowed operators

A handy function to tell if a string is an integer or not is the following

```
let isInt (str : string) : bool = System.Int32.TryParse str |> fst
```

that given a string `str` returns `true` if `str` is an integer and `false` otherwise.

Remember:

- All inputs must be separated by a whitespace, including a trailing whitespace after the last input.
- `read` will return `None` if a newline is encountered. You can match against this to know that you are done. If you do not have a trailing white space the newline will count as that white space and you will need two newlines to end the program

Important:

- In some terminals cut-and-paste may be unreliable, but these formulas are short enough to enter by hand.
- Wher printing `None` you may see `<null>` in stead. This is ok, so do not worry about it.

Hint:

- It can be a good idea to have a recursive auxiliary function of type `SM<int>` that does the actual work for you. This will also simplify debugging as you can use `evalSM` to see the state of the stack.

You may get the following warning `This and other recursive references to the object(s) being defined will be checked for initialization-soundness at runtime through the use of a delayed reference. This is because you are defining one or more recursive objects, rather than recursive functions. This warning may be suppressed by using '#nowarn "40"' or '--nowarn:40'.` This warning can be ignored in this particular case, but if you want to surpress it, have your auxiliary function be of type `unit -> SM<int>` rather than `SM<int>`.

Examples:

```
m> calculateRP ()
<- 5 3 - 2 *
-> 4

m> calculateRP ()
<- 5 3 2 * -
-> -1

m> calculateRP ()
<- 4 2 5 * + 1 3 2 * + *
-> 98
```