Tuesday August 16, 2022

- 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 may **NOT** use any form of code completion tools (like Rider's auto pilot). All code you hand in must either be in the template we provide, or written by yourself. The use of any such tool is grounds for disciplinary action.
- 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 `FPExam2022` 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.

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, or tools that write code for you (such as Rider's auto pilot)

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

1: Grayscale images (25%)

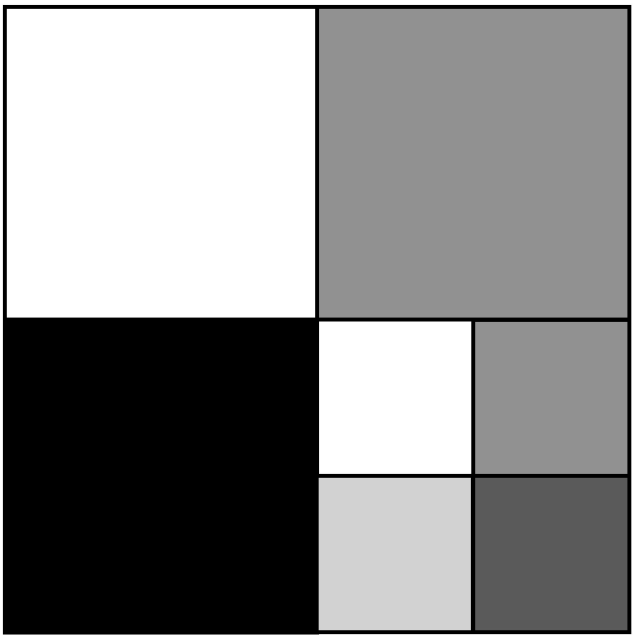
A grayscale image is either completely coloured with one grayscale colour (a number between 0 and 255) or recursively divided into four equally sized grayscale quadrants starting at the top left corner and counting clockwise.

```
type grayscale =
| Square of uint8
| Quad of grayscale * grayscale * grayscale * grayscale
```

As an example, the term `img` (which we will use as an example for the rest of this question)

```
  let img =
    Quad (Square 255uy,
          Square 128uy,
          Quad(Square 255uy,
               Square 128uy,
               Square 192uy,
               Square 64uy),
          Square 0uy)
```

has the following visual representation



Note that

- unsigned 8-bit integers literals are written with the `uy` postfix (`128uy` for example)
- If we need to talk about a specific quadrants we number them in the same order as they are represented in the datatype. For the image above that means that
  - Quadrant one is white
  - Quadrant two is gray
  - Quadrant three is checkered (and we could number its sub-quadrants)

- Quadrant four is black

Question 1.1

Create the recursive, but not tail recursive function `maxDepth` of type `grayscale -> int` that given a grayscale image `img` returns the maximum number of nested quadrants in a grayscale image, where `Square` has depth `0`.

Examples:

```
> maxDepth (Square 123uy)
val it : int = 0

> maxDepth img
val it : int = 2
```

Question 1.2

Create a function `mirror` of type `grayscale -> grayscale` that given a grayscale image `img` mirrors `img` around the Y-axis (quadrant one becomes quadrant two, quadrant two becomes quadrant one, quadrant three becomes quadrant four, and quadrant four becomes quadrant three).

Hint For this to work you must apply the mirroring recursively

Examples:

```
> mirror (Square 123uy)
val it : grayscale = Square 123uy

> mirror (Quad(Square 0uy, Square 85uy, Square 170uy, Square 255uy))
val it : grayscale = Quad (Square 85uy, Square 0uy, Square 255uy, Square 170uy)

> mirror img
val it: grayscale =
  Quad
    (Square 128uy, Square 255uy, Square 0uy,
     Quad (Square 128uy, Square 255uy, Square 64uy, Square 192uy))
```

Question 1.3

Create the function `operate` of type `(grayscale -> grayscale -> grayscale -> grayscale -> grayscale) -> grayscale -> grayscale` that given a function `f` and an image `img` returns

- `Square v` if `img` is equal to `Square v`
- If `img` is equal to `Quad(a, b, c, d)` then the function recurses down the four individual quadrants and combines the results using the function `f`

Examples:

```
let average a b c d =
    match a, b, c, d with
    | Square v1, Square v2, Square v3, Square v4 ->
      Square (((float v1 + float v2 + float v3 + float v4) / 4.0) |> uint8)
    | _, _, _, _ -> Quad (a, b, c, d)

operate average img;
val it : grayscale = Square 135uy
```

Using your `operate` function, create a non-recursive function `mirror2` of type `grayscale -> grayscale` that behaves exactly like the `mirror` function from Q1.2 for all possible inputs.

Question 1.4

Create a funtion `compress` of type `grayscale -> grayscale` that in a bottom-up fashion converts any quadrant `Quad (a, b, c, d)` into any of its components as long as they are all equal, and does nothing otherwise.

Examples:

```
> compress (Square 123uy)
val it : grayscale = Square 123uy

> compress (Quad (Square 123uy, Square 123uy, Square 123uy, Square 123uy))
val it : grayscale = Square 123uy

compress (Quad (Square 123uy, Square 123uy, Square 123uy, Square 0uy))
val it : grayscale = Quad (Square 123uy, Square 123uy, Square 123uy, Square 0uy)

compress (Quad (Square 123uy,
                Square 123uy,
                Quad (Square 123uy, Square 123uy, Square 123uy, Square 123uy),
                Square 123uy))
val it : grayscale = Square 123uy
```

2: Code Comprehension (25%)

Consider the following two functions

```
let rec foo f =
    function
    | []                -> []
    | x :: xs when f x -> x :: (foo f xs)
    | _ :: xs           -> foo f xs

let rec bar fs xs =
    match fs with
    | []       -> xs
    | f :: fs' -> bar fs' (foo f xs)
```

Question 2.1

- What are the types of functions `foo` and `bar`?
- What do the functions `foo` and `bar` do. Focus on what they do rather than how they do it.
- What would be appropriate names for functions `foo`, and `bar`?
- The function `foo` uses an underscore `_` in its third case. Is this good coding practice, if so why, and if not why not?

Question 2.2

Create a non-recursive function `bar2` that behaves the same as `bar` for all possible inputs where

- any recursive function that you use must be higher-order functions from the `List` library
- any auxiliary functions that you write yourself or take from elsewhere must not be recursive.

In particular you may not use `foo`.

Question 2.3

Create a function `baz` of type `('a -> bool) list -> 'a -> bool` such that `bar fs xs` and `foo (baz fs) xs` behave exactly the same for all possible inputs `fs` and `xs`.

Question 2.4

Only one of the functions `foo` and `bar` is tail recursive. Which one? Demonstrate why the other one is not tail recursive. 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 function 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).

Question 2.5

Create a function `fooTail` or `barTail`, whichever is **not** already tail recursive, that behaves the same way as `foo` or `bar` respectively but which is tail recursive and coded using continuations.

3: Guess a number (25%)

For this assignment we will be guessing numbers between `1` and some maximum number `max`. To aid with this we have an oracle which contains the maximum number and a function that given a number `x` between `1` and `max`

- returns `Higher` if `x` is strictly smaller than the target number
- returns `Lower` if `x` is strictly greater than the target number
- returns `Equal` if `x` is equal to the target number

```
type guessResult = Lower | Higher | Equal
type oracle =
    { max : int
      f : int -> guessResult }
```

Question 3.1

Create a function `validOracle` of type `oracle -> bool` that given an oracle `o` returns true if `o` is a valid oracle and false otherwise. A valid oracle

1. only returns `Equal` for a single number `x` between `1` and `max` inclusive
2. returns `Higher` for all numbers greater than or equal to `1` and strictly smaller than `x` (`x` is strictly greater than the number guessed)
3. returns `Lower` for all numbers strictly greater than `x` and smaller than or equal to `max` (`x` is strictly lower than the number guessed)

Examples:

```
> validOracle {max = 10; f = fun _ -> Lower}
val it : bool = false

> validOracle
    { max = 10;
      f = fun x -> if x = 5 then Equal else if x < 5 then Higher else Lower }
val it : bool = true
```

Question 3.2

Create a function `randomOracle` of type `int -> int option -> oracle` that given a maximum number `m` and an optional seed for a random number generator `oseed` initialises a random number generator with `oseed` (if available), picks a random number `r` between `1` and `m` inclusive, and returns an oracle with maximum number `m` that

1. returns `Equal` if the user guesses the random number `r` correctly
2. returns `Higher` if the user guesses a number strictly lower than `r`
3. returns `Lower` if the user guesses a number strictly higher than `r`

Use the following code to generate a random number between `1` and `m` inclusive:

- `(System.Random()).Next(1, m + 1)` if the user did not provide a seed (`oseed = None`)
- `(System.Random(seed)).Next(1, m + 1)` if the user provided a seed (`oseed = Some seed`)

Hint: Make sure to generate the random number before creating the oracle that you return. If you provide the input to the oracle before generating your random number you will generate a new number for every guess.

Examples:

```
> let o = randomOracle 10 (Some 42)
val o : oracle = { max = 10; f = <random function name> }
// At this point the random number has already been chosen, it will not change
// during the following function calls.

> o.f 5
val it : guessResult = Higher

> o.f 8
val it : guessResult = Lower

> o.f 7
val it : guessResult = Equal

> validOracle (randomOracle 10 (Some 42))
val it : bool = true
```

Do not worry in case your seed generates another random number sequence than in the examples, but do make sure to only initialise the random number generator once.

Question 3.3

The fastest way to guess a correct number, assuming that the oracle is valid as described in Q3.1, is through a binary search. That means that given a range between `a` and `b`, where `b` is greater than or equal to `a`, make your first guess `g` to be `(a + b) / 2`, or equivalently `a + ((b - a) / 2)` (integer division).

- If you guess correctly, terminate
- If you guess too low then repeat but with the new range `g + 1` to `b`
- If you guess too high then repeat but with the new range `a` to `g - 1`

This tactic will always terminate in $\log_2 ((b - a) + 1)$ steps rounded up. For a number between `1` and `10` you would only need `4` guesses ($\log_2 10 \approx 3.3219 \approx 4$) and for a number between `1` and `1 000 000` you would need at most `20` guesses.

Create a function `findNumber` of type `oracle -> int list` that given a valid oracle `o` returns a list of numbers containing the guesses made to `o` to arrive at the correct answer. The length of this list must not be greater than $\log_2 m$ where m is the maximum number of `o`.

Examples:

```
> findNumber (randomOracle 10 (Some 42))
val it : int list = [5; 8; 6; 7]

> findNumber (randomOracle 20 (Some 42))
val it : int list = [10; 15; 12; 13; 14]

> findNumber (randomOracle 1000000 (Some 42))
val it : int list =
  [500000; 750000; 625000; 687500; 656250; 671875; 664062; 667968; 669921;
   668944; 668456; 668212; 668090; 668151; 668120; 668105; 668112; 668108;
   668106; 668107]
```

Your lists must not match the examples exactly, but their lengths must be within the bounds described above and the guesses must converge on the correct result.

Question 3.4

Create a function `evilOracle` of type `int -> int option -> oracle` that given a maximum number `m` and an optional seed for a random number generator `oseed` returns an oracle with maximum number `m` that does not decide its target number ahead of time but will, as long as it is able to without being caught in a lie, return a `guessResult` that maximises the search space for the user. This means that given a range between `a` and `b` and if the user guesses the number `x` the oracle will

- return `Higher` if there are more numbers between `x` and `b` then there are between `a` and `x`
- return `Lower` if there are more numbers between `a` and `x` then there are between `x` and `b`
- randomly return `Lower` or `Higher` using a random number generator initialised with the optional seed if there are exactly the same number of higher or lower numbers.
- only return `Equal` when it has no other option as returning `Lower` or `Higher` would contradict a previous result.

To solve this assignment you are highly encouraged to use mutable variables that keep track of the current range that the target number can be in. Make sure to

- Initialise the mutable variables inside the `evilOracle` function but before constructing the oracle that you return, then have the oracle function mutate these variables.
- Initialize the random number generator where you initialise your mutable varibles. If you do it inside the oracle function itself then the random number sequence will be reset with every guess. The oracle function can still use the initialised random number generator to generate fresh random numbers.

Examples:

```
> findNumber (evilOracle 10 (Some 42))
val it : int list = [5; 8; 9; 10]

> findNumber (evilOracle 20 (Some 42))
val it : int list = [10; 15; 18; 19; 20]

> findNumber (evilOracle 1000000 (Some 42))
val it : int list =
```

```
  [500000; 750000; 875000; 937500; 968750; 984375; 992188; 996094; 998047;
   999024; 998535; 998779; 998901; 998962; 998993; 998977; 998985; 998989;
   998987; 998986]

> validOracle (evilOracle 10 (Some 42))
val it : bool = true
```

For these tests we generated a random number between `0` and `1` inclusive and chose `Lower` on `0` and `Higher` on `1`.

Question 3.5

Create a function `parFindNumbers` of type `oracle list -> int list list` that given a list of oracles `os` runs `findNumber` on every element `o` in `os` in parallel and returns the results in a list.

Examples:

```
> parFindNumber [randomOracle 10 (Some 42);
                 randomOracle 20 (Some 42);
                 evilOracle 1000000 (Some 42)]

val it : int list list =
  [[5; 8; 6; 7]; [10; 15; 12; 13; 14];
   [500000; 750000; 875000; 937500; 968750; 984375; 992188; 996094; 998047;
    999024; 998535; 998779; 998901; 998962; 998993; 998977; 998985; 998989;
    998987; 998986]]
```

4: Assembly (25%)

For this assignment we will be working with a small assembly language for a machine that contains three registers and a program counter (the unsigned integer address of the current command being executed).

```
type register = R1 | R2 | R3
type address = uint

type assembly =
| MOVI of register * int
| MULT of register * register * register
| SUB of register * register * register
| JGTZ of register * address
```

The commands of the language are the following:

- `MOVI(r, v)` stores the value `v` in the register `r`
- `MULT(r1, r2, r3)` multiplies the values stored in registers `r2` and `r3` and stores the result in register `r1`

- `SUB(r1, r2, r3)` subtracts the values stored in registers `r3` from `r2` and stores the result in register `r1`
- `JGTZ(r, a)` (jump if greater than zero) changes the program pointer to the adress `a` if the value stored in `r` is greater than zero.

An example program that we will be using for this assignment is a program that calculates the factorial of a number greater than `0` and stores the result in register `R1`.

```
let factorial x =            // Address
    [MOVI (R1, 1)            // 0
     MOVI (R2, x)            // 1
     MOVI (R3, 1)            // 2
     MULT (R1, R1, R2)       // 3 (Loop starts here)
     SUB  (R2, R2, R3)       // 4
     JGTZ (R2, 3u)]          // 5 (Loop ends here)
```

The register `R2` is decreased by one for every iteration of a loop and the program terminates once it reaches `0`. The addresses of the individual commands are given by their position in the list. The final command is a jump that jumps to address `3u` as long as the value in register `R2` is greater than `0`.

Question 4.1

The factorial program above is represented as a list which given an address (an index in the list) requires a linear lookup time for the command stored att that address.

- Create a type `program` that can store a program like `factorial` but which has at worst a logarithmic lookup time from an address to the corresponding command.
- Create a function `assemblyToProgram` of type `assembly list -> program` that given a list of assembly commands produces the corresponding program.

Examples:

```
> assemblyToProgram (factorial 10)
val it : program = <your representation of the program goes here>
```

Question 4.2

During program execution the state of the program contains the following information

- The current value of the program counter (the index of the current command being executed)
- The current values of the three registers
- The source code of the program being run

Create a type `state` that contains these three things

Create a function `emptyState` of type `assembly list -> state` that given a list of assembly commands `cmds` creates a new state with the program counter set to `0u`, the three registers set to `0` and the program source code set to `assemblyToProgram cmds`.

Examples:

```
> emptyState (factorial 10)
val it : state = <your representation of the state goes here>
```

Question 4.3

Create the following functions to access and modify the state

- `setRegister` of type `register -> int -> state -> state` that given a register `r`, a value `v` and a state `st` updates the register `r` in `st` to have the value `v`. The rest of `st` is left unchanged.
- `getRegister` of type `register -> state -> int` that given a register `r` and a state `st` returns the value of the register `r` in the state `st`.
- `setProgramCounter` of type `uint -> state -> state` that given an address `addr` and a state `st` sets the program counter in `st` to `addr`. The rest of `st` is left unchanged.
- `getProgramCounter` of type `state -> uint` that given a state `st` returns the program counter of `st`
- `getProgram` of type `state -> program` that given a state `st` returns the source code of the program in `st`.

Examples:

```
> factorial 10 |> emptyState |> getRegister R1
val it : int = 0

> factorial 10 |> emptyState |> setRegister R1 10 |> getRegister R1
val it : int = 10

> factorial 10 |> emptyState |> setProgramCounter 100u |> getProgramCounter
val it : uint32 = 100u
```

Note that `uint32` is an alias for `uint`.

Question 4.4

For this assignment we will be using a state monad to hide the state. 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 state from Q4.2) and it does not contain an option type.

```
type StateMonad<'a> = SM of (state -> 'a * state)

let ret x = SM (fun s -> x, s)
let bind f (SM a) : StateMonad<'b> =
  SM (fun s ->
  let x, s' = a s
  let (SM g) = f x
  g s')

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

let evalSM prog (SM f) = f (emptyState prog)
```

Create the following functions (using the functions you created in Q4.3 will help a lot)

- `setReg` of type `register -> int -> StateMonad<unit>` that given a register `r` and a value `v`, updates the register `r` in the state to have the value `v`. The rest of state is left unchanged.
- `getReg` of type `register -> StateMonad<int>` that given a register `r` returns the value of the register `r` in the state.
- `setPC` of type `uint -> StateMonad<unit>` that given an address `addr` sets the program counter in the state to `addr`. The rest of state is left unchanged.
- `incPC` of type `StateMonad<unit>` that increases the program counter in the state by one. The rest of the state is left unchanged.
- `lookupCmd` of type `StateMonad<assembly option>` that looks up the command in the source code at the address pointed to by the program counter and returns `None` if the program counter is at an invalid index.

Important: You cannot use monadic operators, like `bind` or `ret`, here 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 Q4.5.

Examples:

```
> getReg R1 |> evalSM (factorial 10) |> fst
val it : int = 0

> setReg R1 10 >>>= getReg R1 |> evalSM (factorial 10) |> fst
val it : int = 10

> lookupCmd |> evalSM (factorial 10) |> fst
val it : assembly option = Some (MOVI (R1, 1))

> incPC >>>= lookupCmd |> evalSM (factorial 10) |> fst
val it : assembly option = Some (MOVI (R2, 10))

> incPC >>>= incPC >>>= incPC >>>= lookupCmd |> evalSM (factorial 10) |> fst
val it : assembly option = Some (MULT (R1, R1, R2))
```

```
> setPC 100u >>>= lookupCmd |> evalSM (factorial 10) |> fst
val it : assembly option = None
```

Question 4.5

For this assignment you may, if you want to, use computation 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 in any way (you may not pattern match on anything of type `StateMonad` nor may you use the `SM` constructor). Use the functions from Q4.4.

Refer to the start of this section for the behaviour of the individual commands.

Create a function `runProgram` of type `unit -> StateMonad<unit>` that looks up the next command from the state to run using `lookupCmd` and

- Terminates if `lookupCmd` returns None
- Executes the command `cmd` if `lookupCmd` returns `Some cmd` and
    - incresases the program counter using `incPC` unless `cmd` successfully performed a jump using `JGTZ`
    - recurses to execute the next command

Examples:

```
> runProgram () >>>= getReg R1 |> evalSM (factorial 5) |> fst
val it : int = 120

> runProgram () >>>= getReg R1 |> evalSM (factorial 10) |> fst
val it : int = 3628800
```