Thursday August 19, 2021

- 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 `FPExam2021_2` 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: Binary lists (25%)

A binary list is like a regular list except it can contain two types of elements and not just one.

```
type binList<'a, 'b> =
| Nil
| Cons1 of 'a * binList<'a, 'b>
| Cons2 of 'b * binList<'a, 'b>
```

Question 1.1

Create the recursive, but not tail recursive function `length : binList<'a, 'b> -> int` that returns the length of the binary list.

Examples:

```
length Nil;;
val it : int = 0

length (Cons1 (3, Cons2 (true, Cons1 (4, Cons2 (false, Cons2(true, Nil))))));;
val it : int = 5
```

Question 1.2

Create the function `split : binList<'a, 'b> -> list<'a> * list<'b>` that given a binary list `lst` returns a tuple of lists `(lst1, lst2)` where `lst1` contains all elements of type `'a` in `lst`, with the order maintained, and `lst2` contains all elements of type `'b` in `lst`, with the order maintained.

Examples:

```
split (Nil : binList<int, bool>);;
val it : int list * bool list = ([], [])

split (Cons1 (3, Cons2 (true, Cons1 (4, Cons2 (false, Cons2(true, Nil))))));;
val it : int list * bool list = ([3; 4], [true; false; true])
```

Create the function `length2 : binList<'a, 'b> -> int * int` that given a binary list `lst` returns a tuple containing the number of elements of type `'a` in `lst` and the number of elements of type `'b` in `lst`.

Examples:

```
length2 (Nil : binList<int, bool>);;
val it : int * int = (0, 0)

length2 (Cons1 (3, Cons2 (true, Cons1 (4, Cons2 (false, Cons2(true, Nil))))));;
val it : int * int = (2, 3)
```

Question 1.3

Create the function `map : ('a -> 'b) -> ('c -> 'd) -> binList<'a, 'c> -> binList<'b, 'd>`, that works similarly to maps for regular lists, and given two functions `f` and `g`, and a binary list `lst`, returns `lst` but where all elements of type `'a` have had `f` applied to them and where all elements of type `'b` have had `g` applied to them, maintaining the order of both element types.

Examples:

```
map (fun x -> x % 2 = 0)
    (function | true -> 0 | false -> 1)
    (Nil : binList<int, bool>);;
val it : binList<bool,int> = Nil

map (fun x -> x % 2 = 0)
    (function | true -> 0 | false -> 1)
    (Cons1 (3, Cons2 (true, Cons1 (4, Cons2 (false, Cons2(true, Nil))))));;
val it : binList<bool,int> =
  Cons1 (false, Cons2 (0, Cons1 (true, Cons2 (1, Cons2 (0, Nil)))))
```

Question 1.4

Create a function `filter : ('a -> bool) -> ('b -> bool) -> binList<'a, 'b> -> binList<'a, 'b>`, that works similarly to filter of regular lists, and given two functions `f` and `g`, and a binary list `lst`, returns `lst` but only keeping elements `a` of type `'a` where `f a` holds and elements `b` of type `'b` where `g b` holds, maintaing the order of both element types.

Examples:

```
filter (fun x -> x % 2 = 0)
       id
       (Nil : binList<int, bool>);;
val it : binList<int,bool> = Nil

filter (fun x -> x % 2 = 0)
       id
       (Cons1 (3, Cons2 (true, Cons1 (4, Cons2 (false, Cons2(true, Nil))))));;
val it : binList<int,bool> = Cons2 (true, Cons1 (4, Cons2 (true, Nil)))
```

Question 1.5

Create the function `fold :  ('b -> 'a -> 'b) -> ('b -> 'c -> 'b) -> 'b -> binList<'a, 'c> -> 'b`, that works similarly to fold of regular lists, and given two functions `f` and `g`, an accumulator `acc` , and a binary list `lst` traverses the list from left-to-right and updates the accumulator by using `f` on elements of type `'a` and `g` on elements of type `'c`.

For instance, we have the following derivation:

```
fold f g acc (Cons1 (x1, Cons2 (y1, Cons1(x2, Cons2(y2, Nil))))) =
g (f (g (f acc x1) y1) x2) y2
```

Examples:

```
fold (+)
     (fun acc -> function | true -> acc | false -> -acc)
     0
     (Nil : binList<int, bool>);;
val it : int = 0

fold (+)
     (fun acc -> function | true -> acc | false -> -acc)
     0
     (Cons1 (3, Cons2 (true, Cons1 (4, Cons2 (false, Cons2(true, Nil))))));;
val it : int = -7
```

2: Code Comprehension (25%)

Consider the following two functions

```
let rec foo xs ys =
  match xs, ys with
  | [], ys -> ys
  | xs, [] -> xs
  | x :: xs, y :: ys when x < y ->
    x :: (foo xs (y :: ys))
  | x :: xs, y :: ys ->
    y :: (foo (x :: xs) ys)

and bar =
  function
  | [] -> []
  | [x] -> [x]
  | xs ->
    let (a, b) = List.splitAt (List.length xs / 2) xs
    foo (bar a) (bar b)
```

Question 2.1

- What are the types of functions `foo` and `bar`?
- What does the function `bar` do. Focus on what it does rather than how it does it.
- What would be appropriate names for functions `foo`, and `bar`?
- What would be appropriate names of the values `a` and `b` in `bar`.

Question 2.2

The code includes the keyword `and`.

- What function does this keyword serve in general (why would you use `and` when writing any program)?
- What would happen if you removed it from this particular program and replaced it with a standard `let` (change the line `and bar =` to `let rec bar =`)? Explain why the program either does or does not work.

Question 2.3

Create a function `foo2` that behaves the same as `foo` but which is constructed using the higher-order function `List.unfold` from the list library.

Compare `foo2` to `foo`. Are the functions equally efficient for all possible inputs?

Hint: Your code should look like this:

`let foo2 xs ys = List.unfold <a function goes here> (xs, ys)`.

Question 2.4

Neither `foo` nor `bar` is tail recursive. Pick one (not both) of them and explain 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 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` that behaves the same way, and has the same complexity, as `foo` but which is tail recursive and coded using an accumulator.

Question 2.6

Create a function `barTail` that behaves the same way as `bar` but which is tail recursive and coded using continuations. Your function may use either `foo` or `fooTail` internally even though `foo` is not tail recursive.

3: Approximating square roots (25%)

A perfect square is an integer whose square root is also an integer (not a floating point number), such as $0, 1, 4, 9, 16, 25, 36, \ldots$.

The square root of a number x, i.e. $\sqrt{x}$, can be approximated in num steps in the following way

- Let y be the closest perfect square to x. If two perfect squares are equally close to x choose the lowest one.
- let r be $\sqrt{y}$.
- Repeat the following step num times: Update r with the value $\dfrac{x/r + r}{2}$.

- Return r

For instance, approximating the square root of five gives you the following result:

The closest perfect square to 5 is 4 (the next one is 9) The steps then look as follows:

1. $\frac{(5/\sqrt{4})+\sqrt{4}}{2} = \frac{(5/2)+2}{2} = \frac{2.5+2}{2} = \frac{4.5}{2} = 2.25$
2. $\frac{(5/2.25)+2.25}{2} = \frac{2.22+2.25}{2} = \frac{4.47}{2} = 2.2361$
3. $\frac{(5/2.2361)+2.2361}{2} = \frac{2.23602+2.2361}{2} = \frac{4.472136}{2} = 2.236068$
4. ...

As you can see this algorithm converges very quickly ($\sqrt{5} = 2.236067977$).

Question 3.1

Create a function `approxSquare : int -> int -> float` that given a non-negative integer `x` and an integer `num` returns the square root of `x` approximated `num` steps as per the definition above. `approxSquare x 0` should be equal to the square root of the nearest perfect square of `x`.

Examples:

```
approxSquare 5 0;;
val it : float = 2.0

approxSquare 5 1;;
val it : float = 2.25

approxSquare 5 2;;
val it : float = 2.236111111

approxSquare 5 3;;
val it : float = 2.236067978

approxSquare 5 4;;
val it : float = 2.236067977
```

Observation: You will need this function for the rest of this exercise. If you are unable to code it, you may use this one in stead without losing any points on those exercises.

`let approxSquare x (_ : int) = System.Math.Sqrt(float x)`

You may not use `System.Math.Sqrt` for any of the following questions but you must use this wrapper function or the correct function that you have written yourself.

Question 3.2

Quadratic equations of the form $ax^2 + bx + c = 0$ can be solved using the following formula:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

For the purposes of this exercise we will assume that these formulas always have two solutions, x_1 and x_2, and you do not have to consider cases where they do not (when a is zero for instance).

As an example, the formula $5x^2 - 4x - 1$ has the solutions

- $x_1 = \frac{- -4 + \sqrt{(-4)^2 - 4 \times 5 \times -1}}{2 \times 5} = \frac{4 + \sqrt{16 - -20}}{10} = \frac{4 + \sqrt{36}}{10} = \frac{4 + 6}{10} = \frac{10}{10} = 1$
- $x_2 = \frac{- -4 - \sqrt{(-4)^2 - 4 \times 5 \times -1}}{2 \times 5} = \frac{4 - \sqrt{16 - -20}}{10} = \frac{4 - \sqrt{36}}{10} = \frac{4 - 6}{10} = \frac{-2}{10} = -0.2$

Create a function `quadtratic : int -> int -> int -> int -> (float * float)` that given four integers `a`, `b`, `c`, and `num` returns the pair `(x1, x2)` where `x1` and `x2` are solutions of the quadratic equation $ax^2 + bx + c = 0$, as described above, and where the square root has been approximated `num` steps using your function from Q3.1.

Examples:

```
quadratic 5 (-4) (-1) 1;;
val it : float * float = (1.0, -0.2)

quadratic 5 (-3) (-1) 1;;
val it : float * float = (0.84, -0.24)

quadratic 5 (-3) (-1) 2;;
val it : float * float = (0.8385185185, -0.2385185185)

quadratic 5 (-3) (-1) 3;;
val it : float * float = (0.8385164807, -0.2385164807)
```

Question 3.3

Create a function `parQuadratic : (int * int * int) list -> int -> int -> (float * float) list` that given a list `eqs` of three-tuples of the form (a, b, c) where each tuple represents the corresponding quadratic equation $ax^2 + bx + c$ = 0, an integer `numProcesses`, representing the number of threads running in parallel, and an integer `num`, returns a list with solutions for each quadratic equation in `eqs` where the square root has been approximated `num` times using your function from Q3.1.

The algorithm must split the list into `numProcesses` chunks of roughly equal size, solve the equations in these lists in parallel, and then combine the results into one list. To make your life easy, we recommend you use the `splitInto` function from the `List` standard library to split your list into chunks of *roughly equal size*.

Examples:

```
[1..10] |> List.map (fun x -> (x, -(x + 1), -(x + 2))) |>
           fun eqs -> parQuadratic eqs 3 5;;
val it : (float * float) list =
  [(3.0, -1.0); (2.350781059, -0.8507810594); (2.119632981, -0.7862996478);
   (2.0, -0.75); (1.926649916, -0.7266499161); (1.877014558, -0.7103478914);
   (1.841170631, -0.6983134882); (1.814061525, -0.6890615247);
   (1.792836525, -0.681725414); (1.775765067, -0.6757650672)]
```

Question 3.4

Create a function `solveQuadratic : string -> int -> (float * float)` that given a string `str` of the form `"<a>x^2 <op> <b>x <op> <c> = 0"` where

- `<a>`, `<b>`, and `<c>` are 32 bit integers,
- `<op>` is either `+` or `-` ,
- any space can be replaced by arbitrarily many spaces (no tabs, newlines, etc), including zero,

and a number `num`, parses `str` using parser combinators from the `JParsec` library and returns the pair `(x1, x2)` where `x1` and `x2` are solutions of the quadratic equation represented by `str`, and where the square root has been approximated `num` steps using your function from Q3.1.

If the string is not well formed the function should fail.

Hint:

- The `pint32` parser also parses negative numbers
- To ensure that your parser only parses valid strings it may help to manually put a terminating character (like a newline that will never occur in the strings we provide) in the end that the parser looks for to terminate the equation.

Example:

```
solveQuadratic "-4x^2 - 5x + 6 = 0" 5;;
val it : (float * float) = (-2.0, 0.75)

solveQuadratic "-4x^2     -  5x+ 6=     0" 5;;
val it : (float * float) = (-2.0, 0.75)

solveQuadratic "-4x^2-5x+6=0" 5;;
val it : (float * float) = (-2.0, 0.75)

solveQuadratic "-4x^3 - 5x + 6 = 0" 5;;
<fails in any way you like>

solveQuadratic "-4x^2 - 5x + 6 = 0 Hello World" 5;;
<fails in any way you like>
```

4: Rational numbers (25%)

For this assignment we will be working with rational numbers $(\mathbb{Q})$. Recall that a rational number is written as $\frac{n}{d}$ where n is called the numerator and d the denominator. Examples of rational numbers are:

$\frac{1}{2}$, $\frac{2}{1}$, or $\frac{5}{4}$

A few things to notice are:

- Whole numbers are written as divided by 1. For instance, the number 2 is written as $\frac{2}{1}$.
- All rational numbers are simplified as far as they will go. For instance,
  rather than writing $\frac{15}{10}$
  we write $\frac{3}{2}$, since 5 is the greatest number that evenely divides both 15 and 10,
  and $\frac{15}{10} = \frac{15/5}{10/5} = \frac{3}{2} = 1.5$.
- Rational numbers that divide by zero, like $\frac{5}{0}$, are not well formed and must be handled separately.

Question 4.1

Provide a good datatype `rat` that represents rational numbers. Note that a type

- should **not** enforce that the demonimator is non-zero
- should **not** require that the rational numbers are simplified as far as they will go

Both these cases are important, but they will be handled by the code you write in subsequent questions and not by the type itself.

Question 4.2

Create a function `mkRat : int -> int -> rat option` that given two integers `n` and `d` returns

- `None` if $d = 0$
- `Some` $\frac{n/g}{d/g}$ if $d \neq 0$ where g is the largest number that evenly divides both n and g. Moreover
  - If n and d are both negative then the negation is removed from both since $\frac{-a}{-b} = \frac{a}{b}$.
  - if either of n or d is negative, but not both, then the negation is moved to the numerator, since $\frac{a}{-b} = \frac{-a}{b}$.

Your algorithm must at most have linear complexity with respect to n or d, but faster algorithms do exist.

Hint: Create a helper function that calculates g.

Create a function `ratToString : rat -> string` that given a rational number $\frac{n}{d}$ returns the string `"n / d"` where the numerator `n` and denominator `d` have been converted to strings.

Examples:

```
> mkRat 5 6 |> Option.get |> ratToString;;
val it : string = "5 / 6"
```

```
> mkRat 15 10 |> Option.get |> ratToString;;
val it : string = "3 / 2"

> mkRat -15 10 |> Option.get |> ratToString;;
val it : string = "-3 / 2"

> mkRat 15 -10 |> Option.get |> ratToString;;
val it : string = "-3 / 2"

> mkRat -15 -10 |> Option.get |> ratToString;;
val it : string = "3 / 2"

> mkRat 0 5 |> Option.get |> ratToString;;
val it : string = "0 / 1"

> mkRat 5 0;;
val it : rat option = None
```

Question 4.3

The arithmetic operations addition, subtraction, multiplication, and division for rational numbers are defined as follows:

- $\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$
- $\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$
- $\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$
- $\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$

For all of the functions below the result must be simplified as far as possible.

Create the functions:

- `plus : rat -> rat -> rat option` that given two rational numbers r_1 and r_2 returns `Some` $r_1 + r_2$ if the resulting rational number is well formed and `None` otherwise.
- `minus : rat -> rat -> rat option` that given two rational numbers r_1 and r_2 returns `Some` $r_1 - r_2$ if the resulting rational number is well formed and `None` otherwise.
- `mult : rat -> rat -> rat option` that given two rational numbers r_1 and r_2 returns `Some` $r_1 \times r_2$ if the resulting rational number is well formed and `None` otherwise.
- `div : rat -> rat -> rat option` that given two rational numbers r_1 and r_2 returns `Some` $r_1 \div r_2$ if the resulting rational number is well formed and `None` otherwise.

Examples:

```
let r1 = mkRat 2 3 |> Option.get
let r2 = mkRat 3 4 |> Option.get

> plus r1 r2 |> Option.get |> ratToString;;
```

```
val it : string = "17 / 12"

> minus r1 r2 |> Option.get |> ratToString;;
val it : string = "-1 / 12"

> minus r2 r2 |> Option.get |> ratToString;;
val it : string = "0 / 1"

> mult r1 r2 |> Option.get |> ratToString;;
val it : string = "1 / 2"

> div r1 r2 |> Option.get |> ratToString;;
val it : string = "8 / 9"

> div r1 (minus r2 r2 |> Option.get)
val it : rat option = None
```

Question 4.4

For this assignment you will need the solutions from Q4.3 and the following state monad:

```
type SM<'a> = SM of (rat -> ('a * rat) option)
let ret x = SM (fun st -> Some (x, st))
let bind (SM m) f =
  SM (fun st ->
        match m st with
        | None -> None
        | Some (x, st') ->
          let (SM g) = f x
          g st')

let (>>=) m f = bind m f
let (>>>=) m n = m >>= (fun () -> n)
let evalSM (SM f) s = f s
```

Whenever any of these functions fail, that means that option part of the state monad should be `None`.

Create the functions (where "the state" refers to the internal rational number of the state monad):

- `smPlus : rat -> SM<unit>` that given a rational number `rat` adds `rat` to the state if the result is well-formed and fails otherwise.
- `smMinus : rat -> SM<unit>` that given a rational number `rat` subtracts `rat` from the state if the result is well-formed and fails otherwise.
- `smMult : rat -> SM<unit>` that given a rational number `rat` multiplies `rat` with the state if the result is well-formed and fails otherwise.
- `smDiv : rat -> SM<unit>` that given a rational number `rat` divides the state with `rat` if the result is well-formed and fails otherwise.

Warning: Pay close attention to subtraction and division here as the order does matter.

Examples:

```
let r1 = mkRat 2 3 |> Option.get
let r2 = mkRat 3 4 |> Option.get

> r1 |> evalSM (smPlus r2) |> Option.get |> snd |> ratToString;;
val it : string = "17 / 12"

> r1 |> evalSM (smMinus r2) |> Option.get |> snd |> ratToString;;
val it : string = "-1 / 12"

> r1 |> evalSM (smMult r2) |> Option.get |> snd |> ratToString;;
val it : string = "1 / 2"

> r1 |> evalSM (smDiv r2) |> Option.get |> snd |> ratToString;;
val it : string = "8 / 9"
```

Question 4.5

Create a function that incrementally applies arithmetic operations from a list. For this assignment you **may not** unfold the definition of `SM` but you **must use** the functions from Q4.4.

For this question you may, if you wish, use computation expressions

```
type StateBuilder() =

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

let state = new StateBuilder()
```

Create the function `calculate : (rat * (rat -> SM<unit>)) list -> SM<unit>` that given a list `[(r1, op1); (r2, op2); ...; (rn-1, opn-1); (rn, opn)]` returns `op1 r1 >>>= (op2 r2 >>>= ... >>>= (opn-1 rn-1 >>>= opn rn)...)`.

Examples:

```
let r1 = mkRat 2 3 |> Option.get
let r2 = mkRat 3 4 |> Option.get
let r3 = mkRat 4 5 |> Option.get
let r4 = mkRat 5 6 |> Option.get
let r5 = mkRat 6 7 |> Option.get

> evalSM (calculate [(r2, smPlus); (r3, smMinus); (r4, smMult); (r5, smDiv)]) r1 |>
  Option.get |> snd |> ratToString;;
val it : string = "259 / 432"
```