Friday June 3, 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%)

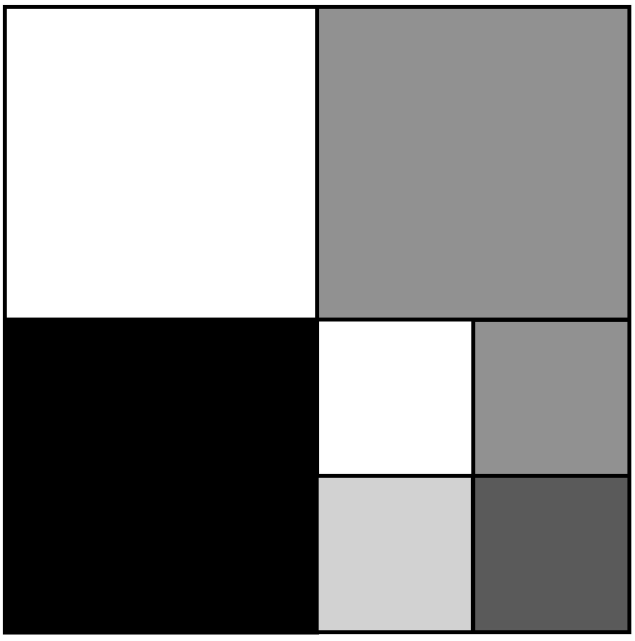
A grayscale image is either completely coloured with one grayscale colour (a number between 0 and 255) or recursively divided into four equally sized grayscale quadrants starting at the top left corner and counting clockwise.

```
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 `countWhite` of type `grayscale -> int` that given a grayscale image `img` returns the number of white cells in `img` (cells with value `255uy`)

Examples:

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

> countWhite img;;
val it : int = 2
```

Question 1.2

Create a function `rotateRight` of type `grayscale -> grayscale` that given a grayscale image `img` rotates `img` 90 degrees clockwise (quadrant one becomes quadrant two, quadrant two becomes quadrant three, quadrant three becomes quadrant four, and quadrant four becomes quadrant one).

Hint For this to work you must apply the rotation recursively

Examples:

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

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

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

Question 1.3

Create the function `map` of type `(uint8 -> grayscale) -> grayscale -> grayscale` that works, similarly to maps on lists or sets, and given a function `mapper` and a grayscale image `img` recursively descends down `img` and applies `mapper` to all solid squares.

Examples:

```
map (fun x -> Square (x + 10uy)) (Square 0uy)
val it: grayscale = Square 10uy

> map (fun x -> Square (x + 10uy))
```

```
      (Quad (Square 0uy, Square 85uy, Square 170uy, Square 255uy));;
val it : grayscale =
  Quad (Square 10uy, Square 95uy, Square 180uy, Square 9uy)
  // note that 8-bit unsigned integer overflow (255uy + 10uy = 9uy)

> map (fun x -> Quad (Square (x + 10uy),
                      Square (x + 20uy),
                      Square (x + 30uy),
                      Square (x + 40uy)))
      (Square (123uy));;
val it: grayscale =
  Quad (Square 133uy, Square 143uy, Square 153uy, Square 163uy)
```

Using your map function, create a non-recursive function `bitmap` of type `grayscale -> grayscale` that given an image `img` returns a black-and-white version of that image, i.e. changes the colour of all squares with a value less than or equal to `127uy` to black (`0uy`) and all other squares to white (`255uy`).

Examples:

```
> bitmap (Square 120uy);;
val it : grayscale = Square 0uy

> bitmap (Square 150uy);;
val it: grayscale = Square 255uy

bitmap img;;
val it : grayscale =
  Quad
    (Square 255uy,
     Square 255uy,
     Quad (Square 255uy, Square 255uy, Square 255uy, Square 0uy),
     Square 0uy)
```

Question 1.4

Create a function `fold` of type `('a -> uint8 -> 'a) -> 'a -> grayscale -> 'a` that, similarly to fold functions on lists or sets, given a function `folder`, a starting accumulator `acc`, and a grayscale image `img` recursively descends down `img` and applies `folder` in quadrant order, calculating new values for `acc` as it does so.

For instance,

`fold f acc (Quad (Square v1, Square v2, Square v3, Square v4))`

should evaluate to

`f (f (f (f acc v1) v2) v3) v4`

or, for nested quadrants,

`fold f acc (Quad (Square v1, Square v2, Quad (Square v3, Square v4, Square v5, Square v6), Square v7))`

should evaluate to

`f (f (f (f (f (f (f acc v1) v2) v3) v4) v5) v6) v7`.

Examples:

```
> fold (fun acc x -> acc + int x) 0 (Square 123uy);;
val it: int = 123

> fold (fun acc x -> acc + int x) 0
       (Quad (Square 0uy, Square 85uy, Square 170uy, Square 255uy));;
val it : int = 510

> fold (fun acc x -> acc + int x) 0 img;;
val it: int = 1022
```

Create a non-recursive function `countWhite2`, using your `fold` function, of type `grayscale -> int` that behaves exactly like your `countWhite` function from Assignment 1.1 for all possible inputs.

Examples:

```
> countWhite2 (Square 123uy);;
val it : int = 0

> countWhite2 img;;
val it : int = 2
```

2: Code Comprehension (25%)

Consider the following two functions

```
let rec foo =
    function
    | 0 -> ""
    | x when x % 2 = 0 -> foo (x / 2) + "0"
    | x when x % 2 = 1 -> foo (x / 2) + "1"

let rec bar =
    function
    | []      -> []
    | x :: xs -> (foo x) :: (bar 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` does not return reasonable results for all possible inputs. What requirements must we have on the input to `foo` in order to get reasonable results?

Question 2.2

The function `foo` compiles with a warning.

- What warning and why?
- Create a function `foo2` that behaves in exactly the same way as `foo` for all possible inputs but that does not produce this warning.

Question 2.3

Create a non-recursive function `bar2` that behaves the same as `bar` for all possible inputs but which is constructed using a higher-order function from the `List` library.

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).

Even though neither `foo` nor `bar` is tail recursive only one of them runs the risk of overflowing the stack. Which one and why does the other one not risk overflowing the stack?

Question 2.5

Create a function `fooTail` that behaves the same way 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: Matrix operations (25%)

A matrix is a two-dimensional collection of numbers and is heavily used in linear algebra.

A matrix A with r rows and c columns (denoted with dimensions $r \times c$) can be written as

$$\begin{bmatrix} A_{0,0} & A_{0,1} & \cdots & A_{1,c-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,c-1} \\ \vdots & & \ddots & \\ A_{r-1,0} & A_{r-1,1} & \cdots & A_{r-1,c-1} \end{bmatrix}$$

Where $A_{i,j}$ represents the value of a matrix A at row i and column j. Note that we, as with lists and arrays, start counting at 0.

We encode our matrices using two-dimensional arrays, but we provide a small API to handle them. You do not need any other array operations than the ones we provide here.

```
type matrix = int[,]

let init f rows cols = Array2D.init rows cols f

let numRows (m : matrix) = Array2D.length1 m
let numCols (m : matrix) = Array2D.length2 m

let get (m : matrix) row col   = m.[row, col]
let set (m : matrix) row col v = m.[row, col] <- v

let print (m : matrix) =
  for row in 0..numRows m - 1 do
    for col in 0..numCols m - 1 do
      printf "%d\t" (get m row col)

    printfn ""
```

where

- `init f rows cols`, where `f` has the type `int -> int -> int`, creates a new matrix with `rows` rows and `cols` columns and and every cell att row `i` and column `j` has the value `f i j`.
- `numRows m` returns the number of rows in the matrix `m`
- `numCols m` returns the number of columns in the matrix `m`
- `get m row col` gets the value of the cell at row `row` and column `col` in the matrix `m`.
- `set m row col v` sets the value of the cell at row `row` and column `col` in matrix `m` to `v`. You will only need this for Question 3.4 and you should not use it otherwise (use `init` instead)
- `print m` prints the matrix `m`. This is for debugging purposes only.

Question 3.1

Most array operations require that arrays be of a certain size. Create a function `failDimensions` of type `matrix -> matrix -> 'a` that given two matrices `m1` and `m2` fail with the error message

`Invalid matrix dimensions: m1 rows = <number of rows in m1>, m1 columns = <number of columns in m1>, m2 roms = <number of rows in m2>, m2 columns = <number of columns in m2>`

using `failwith`.

Note: if the return type of `'a` surprises you then recall that `failwith` must be able to be called anywhere no matter the expected return type; as long as `failDimensions` calls `failwith` at the end, this type will be inferred automatically.

Examples:

```
> let _ : unit = failDimensions (init (fun _ _ -> 0) 3 4) (init (fun _ _ -> 1) 8 9)
System.Exception: Invalid matrix dimensions: m1 rows = 3, m1 columns = 4, m2 roms = 8,
m2 columns = 9
// possible stack trace
```

If your output in the terminal differs slightly then that's fine, but the error message must match exactly.

Question 3.2

Matrix addition of two matrices A and B requires that they have the same dimensions and returns a new matrix C, with the same dimension, where $C_{i,j} = A_{i,j} + B_{i,j}$ for all cells at row i and column j.

Create a function `add` of type `matrix -> matrix -> matrix` that given two matrices `m1` and `m2` of equal dimensions returns the matrix `m1` added to `m2` and fails using your `failDimensions` function from Q3.1 if the dimensions of `m1` and `m2` are not the same.

Hint: Use the `init` function (do not use `set` that is just complicating matters at this point)

Examples:

```
> add (init (fun x y -> x + y) 2 3) (init (fun x y -> x * y) 2 3) |> print
0 1 2
1 3 5
val it: unit = ()

add (init (fun x y -> x + y) 2 3) (init (fun x y -> x * y) 3 2)
System.Exception: Invalid matrix dimensions: m1 rows = 2, m1 columns = 3, m2 roms = 3,
m2 columns = 2
// Possible stack trace
```

Question 3.3

For this question we will be working with matrices A and B where A has the same number of columns as B has rows.

Dot product

Given two matrices A and B of dimensions $a \times b$ and $b \times c$ respectively, the dot product of a row i in A with a column k in B, written $A_i^r \cdot B_k^c$, is computed as

$$A_i^r \cdot B_k^c = \sum_{j=0..b-1} A_{i,j} * B_{j,k}$$

For instance, given the two matrices

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

then

- $A_0^r \cdot B_1^c = 1 * 2 + 2 * 4 + 3 * 6 = 28$
- $A_1^r \cdot B_0^c = 4 * 1 + 5 * 3 + 6 * 5 = 49$

Create a function `dotProduct` of type `matrix -> matrix -> int -> int -> int` that given matrices `m1` and `m2`, a row `row` and a column `col` returns the dot product of the row `row` in `m1` with the column `col` in `m2`. You may assume that the dimensions of `m1` and `m2` are correct (that `m1` has the same number of columns as `m2` has rows).

Example (using the matrices above):

```
let m1 = (init (fun i j -> i * 3 + j + 1) 2 3)
let m2 = (init (fun j k -> j * 2 + k + 1) 3 2)

> dotProduct m1 m2 0 1;;
val it: int = 28

> dotProduct m1 m2 1 0;;
val it: int = 49
```

Matrix multiplication

Given two matrices A and B of dimensions $a \times b$ and $b \times c$ respectively, multiplying A and B, written $A * B$, results in a matrix C of dimensions $a \times c$ where, for every row i and every column k in C, $C_{i,k} = A_i^r \cdot B_k^c$

For our example matrices A and B, that we used when computing the dot product above, we have that

$$A * B = \begin{bmatrix} A_0^r \cdot B_0^c & A_0^r \cdot B_1^c \\ A_1^r \cdot B_0^c & A_1^r \cdot B_1^c \end{bmatrix} = \begin{bmatrix} 1*1+2*3+3*5 & 1*2+2*4+3*6 \\ 4*1+5*3+6*5 & 4*2+5*4+6*6 \end{bmatrix} = \begin{bmatrix} 22 & 28 \\ 49 & 64 \end{bmatrix}$$

Create a function `mult` of type `matrix -> matrix -> matrix` that given two matrices m1 and m2 returns a new matrix equal to `m1` multiplied by `m2` assuming that `m1` has the same number columns as `m2` does rows, and fails with the `failDimensions` function you created in Q3.1 otherwise.

Hint: Aside from the error handling this is a one-liner if you use `init` and `dotProduct`.

Example (using the matrices above):

```
let m1 = (init (fun i j -> i * 3 + j + 1) 2 3)
let m2 = (init (fun j k -> j * 2 + k + 1) 3 2)

> mult m1 m2 |> print;;
22  28
49  64
val it: unit = ()

> mult m1 (init (fun _ _ -> 0) 1 9);;
System.Exception: Invalid matrix dimensions: m1 rows = 2, m1 columns = 3, m2 roms = 1,
m2 columns = 9
// possible stack trace
```

Question 3.4

Create a function `parInit` of type `(int -> int -> int) -> int -> int -> matrix` (the exact same type as `init`) that behaves exactly as `init` does for all possible inputs. However, when creating a matrix with `rows` rows and `cols` columns, `parInit` must spawn `rows` * `cols` threads where every thread

- is responsible for initialising a unique cell at a row `i` and a column `j` in the resulting matrix
- sets the value of its cell to `f i j` (where `f` is the function passed to `parInit`).

Hint: By far the easiest way of solving this question is to initialise an empty matrix using `init` with the correct dimensions (fill it with anything you like) and then have every individual thread update their dedicated cell using the `set` function from the introduction. This is a use of side effects, but we highly encourage that you solve this question this way.

Examples:

```
parInit (fun i j -> i * 3 + j + 1) 2 3 |> print
1 2 3
4 5 6
val it: unit = ()
```

4: Stack machines (25%)

A stack machine can run programs that use a stack to store and retrieve intermediate results of computations. It's syntax is typically straightforward and rather than having commands of the type `e1 + e2` to evaluate expressions `e1` and `e2` individually and then add the results together, a stack machine will have an `ADD` command that pops the top two elements from the stack, adds them together, and pushes the result to the top of the remaining stack.

The code of our stack machine is a list of commands

```
type cmd = Push of int | Add | Mult
type stackProgram = cmd list
```

where

- `Push x` pushes the value `x` to the top of the stack
- `Add` pops the first two elements off the stack, adds them and pushes the result to the stack
- `Mult` pops the first two elements off the stack, multiplies them and pushes the result to the stack

For instance, the program `[Push 5; Push 4; Add; Push 8; Mult]` is equivalent to the expression `(5 + 4) * 8` and evaluates to `72`.

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. The program above is then evaluated as follows

```
step     stack           stack machine code
-------------------------------------------------------------
0        {}              [Push 5; Push 4; Add; Push 8; Mult]
1        {5}             [Push 4; Add; Push 8; Mult]
2        {5, 4}          [Add; Push 8; Mult]
3        {9}             [Push 8; Mult]
4        {8, 9}          [Mult]
5        {72}            []
```

At the end, the result is stored at the top of the stack.

We say that a program is ill-formed if it ever reaches a state, starting from the empty state, during execution where it requires elements from the stack that are not there. Examples of ill-formed programs are `[Add]`, `[Push 3; Mult]`, or `[]` as there are not enough elements at the top of the stack to run the `Add` or `Mult` commands, and nothing at the top of the stack to use as a result for the empty program `[]`.

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 function `emptyStack` of type `unit -> stack` which returns an empty stack.

Question 4.2

Create a function `runStackProg` of type `stackProgram -> int` that given a stack program `prog` evaluates `prog` as explained above with an empty inital stack and returns the top element of the stack once the program has finished executing. If `prog` is ill-formed it should fail with the error message `empty stack`

Examples:

```
> runStackProg [Push 5];;
val it: int = 5

> runStackProg [Push 5; Push 4; Add; Push 8; Mult];;
val it: int = 72

> runStackProg [Push 5; Push 4; Add; Push 8; Mult; Push 42; Add];;
```

```
val it: int = 114

> runStackProg [Push 5; Push 4; Add; Push 8; Mult; Mult];;
System.Exception: empty stack // Possibly a stack trace after this

> runStackProg [];;
System.Exception: empty stack // Possibly a stack trace after this
```

Question 4.3

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 StateMonad<'a> = SM of (stack -> ('a * stack) option)

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

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

let evalSM (SM 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, like `bind` or `ret`, 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 4.4

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 (you may not pattern match on anything of type `StateMonad`).

Create a function `runStackProgram2` of type `stackProgram -> StateMonad<int>` that given a program `prog` evaluates `prog` and monadically returning the value at the top of the stack after the program has finished executing. The `evalSM` function can then be used to evaluate the program from an initial empty stack as demonstrated in the examples bellow. The function should fail (monadic fail, not `failwith`) if `prog` is ill formed.

Examples:

```
> [Push 5] |> runStackProg2 |> evalSM |> Option.map fst;;
val it: int option = Some 5

> [Push 5; Push 4; Add; Push 8; Mult] |> runStackProg2 |> evalSM |> Option.map fst;;
val it: int option = Some 72

> [Push 5; Push 4; Add; Push 8; Mult; Push 42; Add] |>
  runStackProg2 |> evalSM |> Option.map fst;;
val it: int option = Some 114

> [Push 5; Push 4; Add; Push 8; Mult; Mult] |> runStackProg2 |> evalSM |> Option.map
fst;;
val it: int option = None

> [] |> runStackProg2 |> evalSM |> Option.map fst;;
val it: int option = None
```

Question 4.5

The textual representation for our stack programs are

- `"PUSH <x>"`, where `<x>` is a 32-bit integer, for `Push <x>`
- `"ADD"` for `Add`
- `"MULT"` for `Mult`

where commands are separated by newlines and can include any number of white spaces between, before, and after numbers and keywords. For example, the program `[Push 5; Push 4; Add; Push 8; Mult]` can be represented by the following string

```
PUSH      5
PUSH 4
ADD
   PUSH   8
MULT
```

Note that there can also be spaces before the newlines.

Create a function `parseStackProg` , using parser combinators from the `JParsec` library , that has the type `string -> ParserResult<stackProgram>` that parses a string into a stack program.

Examples:

```
"PUSH 5\nPUSH 4   \nADD  \n   PUSH   8\nMULT   \n" |>
run parseStackProg |>
getSuccess |>
runStackProg2 [Push 5] |>
evalSM |>
Option.map fst;;
> val it: int option = Some 72
```

Note: Even if you have not gotten `runStackProg` or `runStackProg2` working you can still get full points for this assignment as long as your parser is correct.