Functional Programming Exam 2023-08-17

- 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 etc. during the examination. This includes any form of device that can execute programs written in F#.
- You may **NOT** use the internet other than LearnIT for the duration of the exam.
- You may **NOT** use any form of code generators like Visual Studio CoPilot. 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. Standard auto-completion like Intellisense is ok.
- 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. Solutions with side effects will not be considered. The one exception to this rule is `Async.Parallel` which returns the results of the individual threads in an array and these results may be used.
- You must use the provided code project `FPReExam2023` as a basis for your submission and you should **only hand in** the `Exam.fs` file (no other file). The project can be run independently, 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 Visual Studio CoPilot)

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

1: Arithmetic (25%)

Consider the following type declaration representing addition of numbers

```
type arith =
| Num of int
| Add of arirth * arith
```

In our descriptions we write:

- $\ulcorner x \urcorner$ for `Num x`
- $a \oplus b$ for `Add(a, b)`

For example, we have the following correspondences between notation and F# code.

Notation	F# Code
$p_1 = \ulcorner 42 \urcorner$	`let p1 = Num 42`
$p_2 = \ulcorner 5 \urcorner \oplus \ulcorner 3 \urcorner$	`let p2 = Add(Num 5, Num 3)`
$p_3 = (\ulcorner 5 \urcorner \oplus \ulcorner 3 \urcorner) \oplus (\ulcorner 7 \urcorner \oplus \ulcorner -9 \urcorner)$	`let p3 = Add(Add(Num 5, Num 3), Add(Num 7, Num (-9)))`

1.1: Evaluation

Create a function `eval : arith -> int` that evaluates a arithmetic expression `a` to an integer value using the following rules, where we write $[[a]]$ for `eval a`.

$$\begin{array}{rcl} [[\ulcorner x \urcorner]] & = & x \\ [[a \oplus b]] & = & [[a]] + [[b]] \end{array}$$

Here + is standard integer addition in `F#` code

Hint: Do not try tail recursion. It's significantly more complicated than using plain recursion.

Examples:

`> eval p1`
`- val it: int = 42`
	`> eval p2`
	`- val it: int = 8`
		`> eval p3`
		`- val it: int = 6`

1.2: Negation and subtraction

Without using the `eval` function, or any similar function, create a function `negate : arith -> arith` that given an arithmetic expression negates that expression according to the following rules, where we write $\ominus a$ for `negate a`.

$$\begin{aligned} \ominus \ulcorner x \urcorner &= \ulcorner -x \urcorner \\ \ominus(a \oplus b) &= (\ominus a) \oplus (\ominus a) \end{aligned}$$

Here `-` is standard negation in `F#`.

Without using the `eval` function, or any similar function, create a function `subtraction : arith -> arith -> arith` that given arithmetic expressions `a` and `b` returns the expression `b` subtracted from `a`. Subtraction is typically encoded the following way, where we overload $\ominus$ and write $a \ominus b$ for `subtraction a b`.

$$a \ominus b = a \oplus (\ominus b)$$

Examples:

```
> negate p1
- val it: arith = Num -42

> negate p2
- val it: arith = Add (Num -5, Num -3)

> negate p3
- val it: arith = Add (Add (Num -5, Num -3), Add (Num -7, Num 9))

> subtract p1 p2
- val it: arith = Add (Num 42, Add (Num -5, Num -3))

> subtract p2 p3
- val it: arith =
    Add (Add (Num 5, Num 3), Add (Add (Num -5, Num -3),
                                  Add (Num -7, Num 9)))

> subtract p3 p1
- val it: arith = Add (Add (Add (Num 5, Num 3), Add (Num 7, Num -9)),
                       Num -42)

> eval (subtract p3 p1)
- val it: int = -36
```

1.3: Multiplication

Without using the `eval` function, or any similar function, create a function `multiply : arith -> arith -> arith` that given arithmetic expressions `a` and `b` returns the expression `a` multiplied by `b` according to the following rules, where we write $a \otimes b$ for `multiply a b`.

$$\begin{aligned} \ulcorner x \urcorner \otimes \ulcorner y \urcorner &= \ulcorner x * y \urcorner \\ \ulcorner x \urcorner \otimes (a \oplus b) &= (\ulcorner x \urcorner \otimes a) \oplus (\ulcorner x \urcorner \otimes b) \\ (a \oplus b) \otimes c &= (a \otimes c) \oplus (b \otimes c) \end{aligned}$$

Here * is standard integer multiplication in `F#` code

Examples:

```
multiply p1 p1
val it: arith = Num 1764

> multiply p2 p2
- val it: arith = Add (Add (Num 25, Num 15), Add (Num 15, Num 9))

> multiply p3 p3
- val it: arith =
  Add
    (Add
       (Add (Add (Num 25, Num 15), Add (Num 35, Num -45)),
        Add (Add (Num 15, Num 9), Add (Num 21, Num -27))),
     Add
       (Add (Add (Num 35, Num 21), Add (Num 49, Num -63)),
        Add (Add (Num -45, Num -27), Add (Num -63, Num 81))))

> eval (multiply p3 p3)
- val it: int = 36
```

1.4: Exponents

Create a tail-recursive function `pow : arith -> arith -> arith`, using an accumulator, that given arithmetic expressions `a` and `b` returns the expression `a` to the power of `b` according to the following rules, where we write a^b for `pow a b`.

$$a^b = \begin{cases} a & \text{if } [[b]] = 1 \\ a \otimes (a^{b \ominus \ulcorner 1 \urcorner}) & \text{otherwise} \end{cases}$$

You may

- only use the `eval` function to check that `b` evaluates to `1`.
- assume that `b` is strictly greater than 0, i.e. greater than or equal to 1.

Examples:

```
pow (Num 2) (Num 8)
val it: arith = Num 256

> pow p2 (Num 3)
- val it: arith =
  Add
    (Add (Add (Num 125, Num 75), Add (Num 75, Num 45)),
     Add (Add (Num 75, Num 45), Add (Num 45, Num 27)))
```

1.5 Iteration

Create a function `iterate : ('a -> 'a) -> 'a -> arith -> 'a` that given a function `f`, an accumulator (not a continuation) `acc`, and an arithmetic expression `a` iterates the function `f` on `acc` a total of `a` times, i.e. `f (f (... a times ... (f acc) ...))`. For example

- `iterate f acc a = acc` if `eval a = 0`
- `iterate f acc p2 = f (f (f (f (f (f (f acc))))))`

Moreover

- You may only use the `eval` function to check that `a` evaluates to `0` in the base case.
- You may assume that `a` is greater than or equal to zero.

Create a non-recursive function `pow2` that behaves exactly the same as `pow` but which is implemented using `iterate`. It is ok if your multiplications end up in a different order, but the results must evaluate to the same value.

Examples:

```
> iterate negate p1 p2
- val it: arith = Num 42

> iterate negate p1 (subtract p2 (Num 1))
- val it: arith = Num -42

> pow2 (Num 2) (Num 8)
val it: arith = Num 256

> eval (pow2 p2 (Num 3))
- val it: int = 512
```

2: Code Comprehension (25%)

Consider and run the following three functions

```
let rec foo =
    function
    | 0            -> true
    | x when x > 0 -> bar (x - 1)
    | x            -> bar (x + 1)

and bar =
    function
    | 0            -> false
    | x when x > 0 -> foo (x - 1)
    | x            -> foo (x + 1)

let rec baz =
    function
    | []                  -> [], []
    | x :: xs when foo x ->
        let ys, zs = baz xs
        (x::ys, zs)
    | x :: xs = ->
        let ys, zs = baz xs
        (ys, x::zs)
```

2.1: Types, names, and behaviour

- 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` ?

2.2: Code snippets

The function baz contains the following three code snippets.

- A: `baz xs`
- B: `bar x`
- C: `(ys, x::zs)`

In the context of the baz function, i.e. assuming that `x`, `xs`, `ys`, and `zs` all have the correct types, what are the types of snippets A, B, and C, expressed using the F# syntax for types, and what are they -- focus on what they do rather than how they do it.

Finally

- explain the use of the `and`-operator that connect the `foo` and the `bar` functions.
- argue if the program would work if you replaced `and` with `let rec`.

2.3: No recursion

Create non recursive functions `foo2` and `bar2` that behave the same for all possible inputs as `foo` and `bar` respectively.

2.4: Tail recursion

The function `baz` is not tail recursive. Demonstrate 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).

You do not have to step through the foo- or the bar functions. You are allowed to evaluate these function immediately to their final results.

Important

Be careful with the `let`-operators. If the function `f` has the form

`fun x -> <function body including x>`

then the term

```
let a = f 5
<more code containing a>
```

will reduce to

```
let a = <function body including 5>
<more code containing a>
```

and the final line of code (containing a) will not reduce until the function body has been completely evaluated and its result stored in `a`. Keeping parentheses around these newly introduced function bodies can help.

2.5: Continuations

Create a tail-recursive version, `bazTail` using continuations (not accumulators) that behaves exactly the same way as `bar` for all possible inputs.

3: Balanced brackets

A string is balanced if all opening brackets have a corresponding closing one. For example, the strings `()`, `({})`, and `()({[]})` are balanced, but the strings `(`, `(()`, and `({)}` are not.

The algorithm for checking that a string is balanced is relatively straightforward using a stack and iterating over the string.

- If the next character is an opening bracket, recurse and push the corresponding closing bracket to the stack
- If the next character is a closing bracket, pop the top element from stack and recurse if they are equal. Return `false` if they are not equal, or if the stack is empty and there is no top element to pop.
- If there are no next characters return `true` if the stack is empty, and `false` otherwise.

An example to see that the string `()({[]})` is balanced can be seen here

String	Stack
`()({[]})`	empty
`)({[]})`	`)`
`({[]})`	empty
`{[]})`	`)`
`[]})`	`})`
`]})`	`]})`
`})`	`})`
`)`	`)`
	empty

Since the stack is empty when we finish with the string, we know that the string is balanced.

For this assignment you may use these two functions that turns strings to lists of characters and vice versa.

```
let explode (str : string) = [for c in str -> c]
let implode (lst : char list) = lst |> List.toArray |> System.String
```

3.1: Balanced brackets

Create a function `balanced : string -> bool` that given a string `str` that only contains the characters `{`, `}`, `(`, `)`, `[`, and `]` returns `true` if `str` is balanced and false otherwise.

Hint: Your job will be easier if you convert `str` to a list and check that the list is balanced.

Examples:

```
> balanced "()"
- val it: bool = true

> balanced "(){([])}"
- val it: bool = true

> balanced "(){([])}("
- val it: bool = false
```

3.2 Arbitrary delimiters

Things get a bit more interesting if we allow users to define which opening character should be matched with which closing one. Consider the following mapping

```
(1) a -> b
(2) b -> c
```

which states that the character `a` is closed by `b` and the character `b` is closed by `c`. The string `abcb` can be successfully balanced, but the algorithm needs to be tweaked. When you encounter the first `b` you can either close the previous `a` using the first rule, which will ultimately fail, or continue using the second rule and close the `b` with a `c` and finally the first `a` with the last `b`, which will succeed.

The algorithm above can be used but whenever you have a choice of either opening a character or closing one you must try both and return true if one approach succeeds.

String	Stack	Action
`abcb`	empty	open `a` using rule (1)
`bcb`	`b`	here we have a choice

Case 1: (close b using rule (1))

String	Stack	Action
`cb`	empty	Stuck

We cannot continue, there is no rule that opens `c`

Case 2: (open b using rule (2))

String	Stack	Action
`cb`	`cb`	close `c` using rule (2)
`b`	`b`	close `b` using rule (1)
	empty	Done

This case succeeds.

Create a function `balanced2 : Map<char, char> -> string -> bool` that given a mapping `m` from opening to closing characters, and a string `str`, returns `true` if `str` is balanced with respect to `m` and false otherwise.

Examples:

```
balanced2 (Map.ofList [('a', 'b'); ('b', 'c')]) "abcb"
val it: bool = true

balanced2 (Map.ofList [('a', 'b'); ('b', 'c')]) "abacb"
val it: bool = false
```

3.3: Matching brackets and symmetric phrases

Both of the functions in this assignment must be implemented using `balanced2`

Matching Brackets

Create a non-recursive function `balanced3 : string -> bool` that behaves exactly the same as `balanced`.

Symmetric phrases

A phrase is symmetric if

- it is empty
- if `s` is symmetric then so is `csc` where `c` can be any letter from `a-z`.
- if `s1` and `s2` are symmetric then so is the concatenation `s1s2`

Examples of symmetric phrases are any even-length palindrome (a string that reads the same forwards as backwards) like `anna` or `bobbob`. Concatenations of even-length palindromes are also symmetric, like `annabobbob`. Odd-length palindroms like `ana` or `bob` are not symmetric.

Create a non-recursive function `symmetric : string -> bool` that given a string `s` returns `true` if `s` is a symmetric phrase, and false otherwise. The string `s`

- may contain non-letters like spaces and punctuation, which should be ignored, but all letters will be from the English alphabet (`a..z`).
- may contain upper and lower case characters which should be treated as being equal.
  See the examples for details

Hint: When implementing `symmetric` it will help significantly if you remove all non-letters from `s` and either make all remaining letters upper case or lower case before you check for symmetry. There are higher-order functions in the `List` library that can help (assuming you convert `s` to a list) and you will find other useful functions in the `System.Char` library.

Examples:

```
> balanced3 "(){([])}"
- val it: bool = true

> balanced3 "(){([])}("
- val it: bool = false

> symmetric "aabbaa"
- val it: bool = true

> symmetric "Dromedaren Alpotto planerade mord!!!"
- val it: bool = true

> symmetric "Dromedaren Alpotto skadar ingen"
- val it: bool = false
```

3.4: Parsing balanced brackets

Create a parser combinator `parseBalanced : Parser<unit>` that succeeds if it parses a balanced string containing only the characters `{`, `}`, `(`, `)`, `[`, and `]`. The string should end with `**END**`, which is not part of the balancing but just for termination.

You may use the following codeSkeleton:

```
let ParseBalanced, bref = createParserForwardedToRef<unit>()

let parseBalancedAux = <Your code goes here>

do bref := parseBalancedAux

let parseBalanced = parseBalancedAux .>> pstring "**END**"
```

Examples:

```
> run parseBalanced "{([]())}{}**END**"
- val it : ParserResult<unit> = Success ()
```

3.5: Parallel counting

Create a function `countBalanced : string list -> int -> int` that given a list of strings `lst` and an integer `x` returns the number of strings in `lst` that are balanced, containing only the characters `{`, `}`, `(`, `)`, `[`, and `]`, and where `lst` is split up into `x` pieces, whose length differ by at most one element, where every piece is handled by a separate thread.

Hint: You can use `List.byChunkSize` to split your list into `x` roughly equal sized pieces. but read the documentation first since the argument to the function is not the number of the chunks but the size of the chunks.

Examples:

```
let lst = [for i in 1..10000 do
              yield! ["()"; "({})"; "()({[]})";
                      "("; "{{}"; "{()"]]

> countBalanced lst 1
- val it: int = 30000

> countBalanced lst 10
- val it: int = 30000

> countBalanced lst 1000
- val it: int = 30000

> countBalanced lst 10000
- val it: int = 30000
```

4: BASIC (25%)

For this assignment we will use a very small variant of one of the first programming languages BASIC. BASIC is (in)famous for having a line number for every command and directing control flow by jumping to explicit line numbers.

```
type var = string

type expr =
| Num    of int              // Integer literal
| Lookup of var              // Variable lookup
| Plus   of expr * expr      // Addition
| Minus  of expr * expr      // Subtraction

type stmnt =
| If of expr * uint32      // Conditional jump to line number
| Let of var * expr        // Variable update/declaration
| Goto of uint32           // Jump to line number
| End                      // Terminate program

type prog = (uint32 * stmnt) list
   // Programs are sequences of line numbers
   // and commands


let (.+.) e1 e2 = Plus(e1, e2)
let (.-.) e1 e2 = Minus(e1, e2)
```

Programs are sequences of line numbers and commands and control flow is handled by jumping to specific line numbers. Variables are strings and are updated during execution using a variable environment similar to what you have already done in assignment 6. Moreover,

- `Lookup v`, represents the value of the variable `v` stored in the variable environment.
- `If(e, l)` jumps to the line `l` if `e` evaluates to a non-zero value
- `Let(v, e)` evaluates the expression `e` and stores the result in the variable `v`, overwriting the previous value if the variable was already declared.
- `Goto l` jumps to the line `l`
- `End` terminates the program. All programs we give you will eventually reach this command.

For readability we use syntactic sugar

- `e1 .+. e2` for `Plus(e1, e2)`
- `e1 .-. e2` for `Minus(e1, e2)`

A running example throughout this assignment will be the following program that calculates the Fibonacci sequence of a given number `x`.

```
let fibProg xarg =
    [(10u, Let("x",    Num xarg)) // x = xarg
     (20u, Let("acc1", Num 1))    // acc1 = 1
     (30u, Let("acc2", Num 0))    // acc2 = 0

     (40u, If(Lookup "x", 60u))   // if x > 0 then goto 60 (start loop)

     (50u, Goto 110u)             // Goto 110 (x = 0, terminate program)

     (60u, Let ("x", Lookup "x" .-. Num 1)) // x = x - 1
     (70u, Let ("result", Lookup "acc1"))   // result = acc1
     (80u, Let ("acc1", Lookup "acc1" .+.
                        Lookup "acc2"))     // acc1 = acc1 + acc2
     (90u, Let ("acc2", Lookup "result"))   // acc2 = result
     (100u, Goto 40u)                       // Goto 40u (go to top of loop)

     (110u, End) // Terminate program
                 // the variable result contains the
                 // fibonacci number of xarg
    ]
```

- Lines 10-30 initialise the program
- Line 40 contains the guard for the loop which will continue as long as `x` is non-zero
- Line 50 jumps to the end of the program if `x` is equal to 0 (the loop has terminated)
- Lines 60-90 contain the loop body
- Line 100 jumps back to the beginning of the loop to check the guard again
- Line 110 terminates the program

4.1: Program representation

Programs are represented as a sequence of line numbers and commands. We will use a map from line numbers to commands to store our programs.

`type basicProgram = Map<uint32, stmnt>`

Important: The following questions are all one liners if you use the correct functions from the `Map` library.

Create a function `mkBasicProgram : prog -> basicProgram` that given a list of line numbers and statements `p` returns the corresponding BASIC program where all line numbers are mapped to their corresponding statements in `p`. You may assume that there are no duplicate line numbers in p.

Create a function `getStmnt : uint32 -> basicProgram -> stmnt` that given a line number `l` and a program `p` returns the statement in `p` at line `l`. You do not have to handle the case when `l` is not a valid line number in `p`.

Create a function `nextLine : uint32 -> basicProgram -> uint32` that given a line number `l` and a program `p` returns the next line number in `p` after `l`, i.e. the smallest line number `l'` such that `l' > l`. Do not take the statement at line `l` into consideration, even if it is a `Goto`. You do not have to handle the case where there is no greater line `l'`.

Crete a function `firstLine : basicProgram -> uint32` that given a program `p` returns the line number of the first statement in `p`, i.e. the smallest line number in `p`. You do not have to handle the case when `p` is empty.

Examples:

```
> mkBasicProgram [(10, End)]
- val it: Map<int,stmnt> = map [(10, End)]

let fp = 10 |> fibProg |> mkBasicProgram
val fp: Map<uint32,stmnt> = map [(10u, Let ("x", Num 10)); ...]

> getStmnt 50u fp
- val it: stmnt = Goto 110u

> nextLine 50u fp
- val it: uint32 = 60u // Note NOT 110u even though the statement is a Goto

> firstLine fp
- val it: uint32 = 10u
```

4.2: State

To evaluate a BASIC program you require state that contains both the current line number of the execution as well as a variable environment that maps variables (strings) to values (integers).

For the rest of the exam we will, for example, write `{lineNumber = 10, x -> 1; y -> 2; z -> 3}` for the state where the current line number is `10` and the variable environment contains the variables `x`, `y`, and `z`, which are mapped to `1`, `2`, and `3`, respectively.

Create a type `state` that contains the current line number of the execution as well as a variable environment.

Important: If you have chosen a good representation for `state` then, with the help of the functions you made in 4.1, the following functions are all one-liners.

Create a function `emptyState : basicProgram -> state` that given a program `p` returns a state with the current line number set to the first line of `p`, and an empty variable environment.

Create a function `goto : uint32 -> state -> state` that given a line number `l` and a state `st`, returns the same state `st` but where the current line number is set to `l`. You do not have to handle the case where `l` is not a valid line in `p`.

Create a function `getCurrentStmnt : basicProgram -> state -> stmnt` that given a program p and a state `st` returns the statement in `p` that is stored at the current line number of `st`. You do not have to handle the case where the current line number is not a valid line number in the program.

Create a function `update : var -> int -> state -> state` that given a variable `v`, a value `a`, and a state `st`, returns the same state `st` but where the variable environment has been updated such that the variable `v` maps to `a`.

Create a function `lookup : var -> state -> int` that given a variable `v` and a state `st` returns the value of the variable `v` in the variable environment of `st`. You do not have to handle the case where `v` is not in the variable environment.

Examples:

```
> emptyState fp
- val it: state = { lineNumber = 10u }

> goto 50u (emptyState fp)
- val it: state = { lineNumber = 50u }

> getCurrentStmnt fp (emptyState fp)
- val it: stmnt = Let ("x", Num 10)

> getCurrentStmnt fp (goto 50u (emptyState fp))
- val it: stmnt = Goto 110u

> update "x" 42 (emptyState fp)
- val it: state = { currentLine = 10u; "x" -> 42 }

> lookup "x" (update "x" 42 (emptyState fp))
- val it: int = 42
```

4.3: Evaluation

Create a function `evalExpr : expr -> state -> int` that given an expression `e` and a state `st` where `e` is

- `Num x`, returns `x`
- `Lookup v`, returns the value of the variable `v` from the variable environment in `st`. You may assume that `v` exists in the variable environment.
- `Plus (e1, e2)` returns the evaluation of `e1` in `st` plus the evaluation of `e2` in `st`
- `Minus (e1, e2)` returns the evaluation of `e1` in `st` minus the evaluation of `e2` in `st`

Create a function `step : program -> state -> state` that given a program `p` and a state `st` returns a state identical to `st` but where the current line has been changed to the next available line in `p`. This function is a one-liner. Do not take the statement stored at the line into consideration, even if it is a `Goto`, just use the next line. You do not have to handle the case where there is no next line.

Create a function `evalProg : basicProgram -> state` that given a program `p` evaluates `p`, starting from the empty state initialised with `p`. It does this by looking up the statement at the current line number in the current state `st` and if the current statement is:

- `If(e, l)` evaluate `e` in `st` and if the result is non zero jump to `l` (by updating the current line of `st`) and step to the next line otherwise (using the `step` function). Continue the evaluation from the new line.
- `Let(v, e)`
  - evaluate `e` and update the variable environment in `st` by mapping `v` to the result
  - Continue the evaluation from the next line
- `Goto l` jump to `l` and continue the evaluation from there
- `End` Terminate and return `st`.

Examples:

```
let st = emptyState fp
let st' = update "x" 42 st

> evalExpr (Num 5) st
- val it: int = 5

> evalExpr (Lookup "x") st'
- val it: int = 42

> evalExpr (Plus (Lookup "x", Num 5)) st'
- val it: int = 47

let smallProg = [(10u, Let ("x", Num 42)); (20u, End)] |>
                mkBasicProgram

> evalProg smallProg
- val it: state = { lineNumber = 20u; "x" -> 42 }

> evalProg fp
- val it: state = { lineNumber = 110u;
-                   "acc1" -> 89, "acc2" -> 55, "result" -> 55, "x" -> 0 }

> fp |> evalProg |> lookup "result"
- val it: int = 55
```

4.4: State Monad

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 simplified considerably as

- the state is much simpler (your state from Q4.2).
- There is no error handling -- the return type is `'a * state` and not `('a * state) option'`.
- The function inside `SM` takes a `basicProgram` as an argument as well as the state. Since the program does not change during execution this program is not returned (the return type is just `'a * state`) but just propagated by `bind`. This allows your functions to use both state and programs to calculate their result, but they only have to update the state.

Impartant: Make sure you understand the definition of `StateMonad` before you continue.

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

let ret x = SM (fun _ s -> (x, s))

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

// Note that the state is updated from s to s' in bind when
// evaluating a p s and g p s', but the program p remains the same.

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

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

Important: The following functions are all one-liners if you use your functions from Q4.1 and Q4.2.

Create a function `goto2 : uint32 -> SM<unit>` that given a line number `l` updates the state by setting the current line number to `l`. You do not have to handle the case where `l` is not a valid line in the program.

Create a function `getCurrentStmnt2 : SM<stmnt>` that returns the statement in the program that is stored at the current line number of the state. You do not have to handle the case where the current line number is not valid in the program.

Create a function `update2 : var -> int -> SM<unit>` that given a variable `v` and a value `a`, updates the variable environment of the state such that the variable `v` maps to `a`.

Create a function `lookup2 : var -> SM<int>` that given a variable `v` returns the value of `v` in the variable environment of the state. You do not have to handle the case where `v` is not in the variable environment.

Create a function `step2 : SM<unit>` that updates the state such that the current line has been changed to the next available line in the program. You do not have to handle the case where there is no next line.

Examples:

```
> goto2 50u |> evalSM fp
- val it: unit * state = ((), { lineNumber = 50u })

> goto2 50u >>>= getCurrentStmnt2 |> evalSM fp
- val it: stmnt * state = (Goto 110u, { lineNumber = 50u })

update2 "x" 42 |> evalSM fp
- val it: unit * state = ((), { currentLine = 10u; "x" -> 42 })

> update2 "x" 42 >>>= lookup2 "x" |> evalSM fp
- val it: int * state = (42, { currentLine = 10u; "x" -> 42 })

> goto2 50u >>>= step2 |> evalSM fp
val it: unit * state = ((), { lineNumber = 60u })
```

4.5: State Monad Evaluation

For this assignment you may, but you do not have 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 = 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` nor construct state monads using `SM` directly).

Create the function `evalExpr2 : expr -> SM<int>` that works exactly like `evalExpr` but where the state is abstracted by the state monad.

Create the function `evalProg2 : SM<unit>` that work exactly like `evalProg` but where the state is abstracted by the state monad.

Examples:

```
> Num 5 |> evalExpr2 |> evalSM fp
- val it: int * state = (5, { lineNumber = 10u })

> update2 "x" 42 >>>= evalExpr2 (Lookup "x") |> evalSM fp
- val it: int * state = (42, { currentLine = 10u; "x" -> 42 })


> update2 "x" 42 >>>= evalExpr2 (Plus (Lookup "x", Num 5)) |>
  evalSM fp
- val it: int * state = (47, { currentLine = 10u; "x" -> 42 })

let smallProg = [(10u, Let ("x", Num 42)); (20u, End)] |>
                mkBasicProgram

> evalProg2 |> evalSM smallProg
- val it: unit * state = ((), { lineNumber = 20u; "x" -> 42 })

> evalProg2 |> evalSM fp
- val it: unit * state =
      ((), { lineNumber = 110u;
             "acc1" -> 89, "acc2" -> 55, "result" -> 55, "x" -> 0 }

> evalProg2 >>>= lookup2 "result" |> evalSM fp
- val it: int * state =
      ((), { lineNumber = 110u;
             "acc1" -> 89, "acc2" -> 55, "result" -> 55, "x" -> 0 }
```