Functional Programming Exam 2023-06-02

- 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 `FPExam2023` 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: Logic (25%)

Consider the following type declaration for logic propositions containing only four constructors -- true, false, conjunction and disjunction.

```
type prop =
| TT
| FF
| And of prop * prop
| Or of prop * prop
```

In our descriptions we write:

- $\top$ for `TT`
- $\bot$ for `FF`
- $P \wedge Q$ for `And(P, Q)`
- $P \vee Q$ for `Or(P, Q)`

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

Notation	F# Code
$p_1 = \top \wedge \bot$	`let p1 = And(TT, FF)`
$p_2 = \top \vee \bot$	`let p2 = Or(TT, FF)`
$p_3 = (\top \vee (\top \wedge \bot)) \wedge \top$	`let p3 = And(Or(TT, And(TT, FF)), TT)`
$p_4 = (\top \vee (\top \wedge \bot)) \wedge (\bot \vee (\top \wedge \bot))$	`let p4 = And(Or(TT,And(TT,FF)), Or(FF,And(TT,FF)))`

When writing code we stick to the F# type definition `prop`.

1.1: Evaluation

Create a function `eval : prop -> bool` that evaluates a proposition `p` to a Boolean value using the following rules, where we write $[[P]]$ for `eval P`.

$$\begin{aligned} [[\top]] &= \text{true} \\ [[\bot]] &= \text{false} \\ [[P \wedge Q]] &= [[P]] \;\&\&\; [[Q]] \\ [[P \vee Q]] &= [[P]] \;||\; [[Q]] \end{aligned}$$

Here `true`, `false`, `&&`, and `||` are the corresponding operators in `F#` code
Hint: Do not try tail recursion. It's significantly more complicated than using plain recursion.

Examples:

```
> eval TT
- val it: bool = true

> eval FF
- val it: bool = false
```

```
> eval p1
- val it: bool = false

> eval p2
- val it: bool = true
```

```
> eval p3
- val it: bool = true

> eval p4
- val it: bool = false
```

1.2: Negation and Implication

Create a function `negate : prop -> prop` that given a proposition negates that proposition according to the following rules, where we write $\neg P$ for `negate P`.

$$\begin{aligned} \neg\top &= \bot \\ \neg\bot &= \top \\ \neg(P \wedge Q) &= (\neg P) \vee (\neg Q) \\ \neg(P \vee Q) &= (\neg P) \wedge (\neg Q) \end{aligned}$$

Create a function `implies : prop -> prop -> prop` that given propositions `P` and `Q` returns a proposition that holds when `P` implies `Q`. Implication is typically encoded in the following way, where we write $P \longrightarrow Q$ for `implies P Q`.

$$P \longrightarrow Q \quad = \quad (\neg P) \vee Q$$

Examples:

```
> negate TT
- val it: prop = FF

> negate FF
- val it: prop = TT

> negate p1
- val it: prop = Or (FF, TT)

> negate p2
- val it: prop = And (FF, TT)

> negate p3
- val it: prop =
      Or (And (FF, Or (FF, TT)), FF)

> negate p4
- val it: prop =
     Or (And (FF, Or (FF, TT)),
           And (TT, Or (FF, TT)))
```

```
> implies TT FF
- val it: prop = Or (FF, FF)

> implies FF TT
- val it: prop = Or (TT, TT)

> implies p3 p4
- val it: prop =
  Or
    (Or (And (FF, Or (FF, TT)), FF),
     And (Or (TT, And (TT, FF)),
              Or (FF, And (TT, FF))))
```

1.3: Bounded Universal Quantifiers

Create a tail-recursive function `forall : ('a -> prop) -> 'a list -> prop`, using an accumulator, that given a function `f` and a list `lst` creates the conjunction of applying `f` to all elements of `lst`. More precisely, where we write $\forall l.\ f$ for `forall f l`:

$$\begin{array}{rcll} \forall [].\ f & = & \top & \text{Empty list} \\ \forall [x_1;\ \cdots\ ; x_n].\ f & = & f\ x_1 \wedge \cdots \wedge f\ x_n & \text{List contains at least one element} \end{array}$$

We only require that you return a proposition that evaluates to the same Boolean value as the propositions listed here. In particular, the order of your conjuncts are allowed to vary and, depending on your implementation, you may have an extra $\top$ somewhere.

Examples:

```
> let mod2 x = if x % 2 = 0 then TT else FF
- val mod2: x: int -> prop

> [0 .. 10] |> forall mod2 |> eval
- val it: bool = false

> [0 .. 2 .. 10] |> forall mod2 |> eval
- val it: bool = true
```

1.4 Bounded Existential Quantifiers

Create a non-recursive function `exists : ('a -> prop) -> 'a list -> prop`, using higher-order function(s) from the List-library, that given a function `f` and a list `lst` creates the disjunction of applying f to all elements of `lst`. More precisely, where we write $\exists l.\ f$ for `exists f l`:

$$\begin{array}{rcll} \exists [].\ f & = & \bot & \text{Empty list} \\ \exists [x_1;\ \cdots\ ; x_n].\ f & = & f\ x_1 \vee \cdots \vee f\ x_n & \text{List contains at least one element} \end{array}$$

We only require that you return a proposition that evaluates to the same Boolean value as the propositions listed here. In particular, the order of your disjuncts are allowed to vary and, depending on your implementation, you may have an extra $\bot$ somewhere.

Examples:

```
> let mod2 x = if x % 2 = 0 then TT else FF
- val mod2: x: int -> prop

> [0 .. 10] |> exists mod2 |> eval
- val it: bool = true

> [1 .. 2 .. 10] |> forall mod2 |> eval
- val it: bool = false
```

1.5 Bounded Unique Existential Quantifiers

Create a function `existsOne : ('a -> prop) -> 'a list -> prop`, in any way you like, that given a function `f` and a list `lst` creates a proposition that is true if there exists exactly one element `x` in `lst` such that `f x` evaluates to `true` and for all other elements `x'`, `f x'` evalutes to `false`. More precisely, where we write $\exists_1 l.\ f$ for `existsOne f l`:

$$
\begin{array}{rcl}
\exists_1[].\ f & = & \bot \\
\exists_1[x_1;\ x_2;\ \cdots\ ;\ x_{n-1};\ x_n].\ f & = & (f\ x_1 \wedge (\neg(f\ x_2) \wedge \cdots \wedge \neg(f\ x_n))) \vee \\
 & & (f\ x_2 \wedge (\neg(f\ x_1) \wedge \neg(f\ x_3) \wedge \cdots \wedge \neg(f\ x_n))) \vee \\
 & & \vdots \\
 & & (f\ x_n \wedge (\neg(f\ x_1) \wedge \cdots \wedge \neg(f\ x_{n-1})))
\end{array}
$$

As with previous quantifiers, the second case requires at least one element in the list to be used -- we display more in the rule for readability purposes. For each element in the list ($x_1 \cdots x_n$ in the rules above), we have a conjunction stating that that element is true when applied to `f` and all others are false. These conjunctions are then joined using disjunctions.

We only require that you return a proposition that evaluates to the same Boolean value as the proposition listed here. The internal order of your conjuncts and disjuncts do not matter -- i.e. for each row the conjuncts can appear in any order, and the rows themselves can appear in any order.

Examples:

```
> [1 .. 10] |> existsOne (fun x -> if x % 2 = 0 then TT else FF) |> eval
- val it: bool = false

> [1 .. 10] |> existsOne (fun x -> if x % 5 = 0 then TT else FF) |> eval
- val it: bool = false

> [1 .. 10] |> existsOne (fun x -> if x % 6 = 0 then TT else FF) |> eval
- val it: bool = true
```

2: Code Comprehension (25%)

Consider and run the following three functions

```
let rec foo xs ys =
    match xs, ys with
    | _       , []                  -> Some xs
    | x :: xs', y :: ys' when x = y -> foo xs' ys'
    | _       , _                   -> None

let rec bar xs ys =
    match foo xs ys with
    | Some zs -> bar zs ys
    | None -> match xs with
              | [] -> []
              | x :: xs' -> x :: (bar xs' ys)

let baz (a : string) (b : string) =
    bar [for c in a -> c] [for c in b -> c] |>
    List.fold (fun acc c -> acc + string c) ""
```

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: `[for c in a -> c]`
- B: `[for c in b -> c]`
- C: `List.fold (fun acc c -> acc + string c) ""`

In the context of the baz function, i.e. assuming that `a` and `b` are strings, what are the types of snippets A, B, and C and what are they -- focus on what they do rather than how they do it.

Finally, explain the use of the `|>`-operator in the baz function.

2.3: No recursion

Create a non-recursive function `foo2` that behaves exactly the same as `foo` for all possible inputs

Hint: Use the `List.splitAt` function from the standard library. You may use other functions as well.

2.4: Tail recursion

The function `bar` 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-function. You are allowed to evaluate that function immediately to its final result.

2.5: Continuations

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

3: Collatz Conjecture

Collatz Conjecture is a fascinating, but as yet unproven conjecture (hence the name conjecture) that given any positive integer (not including zero) the following function will eventually terminate.

$$\mathsf{f}\ x \quad = \quad \begin{cases} 1 & \text{if } x = 1 \\ \mathsf{f}\ (x/2) & \text{if } x \text{ is even} \\ \mathsf{f}\ (3x+1) & \text{if } x \text{ is odd} \end{cases}$$

This function terminates for all positive numbers up to $2^{68} \approx 2.95 \times 10^{20}$ (source Wikipedia) which is an astronomically large number, but f may diverge for larger numbers. We do not know.

3.1 Collatz Sequences

For this assignment you will be calculating Collatz sequences, i.e. the sequence of numbers from an initial number `x` down to `1` as calculated by function `f` above. For example:

- The Collatz sequence of `1` is `[1]`
- The Collatz sequence of `2` is `[2; 1]`
- The Collatz sequence of `3` is `[3; 10; 5; 16; 8; 4; 2; 1]`
- The Collatz sequence of `42` is `[42; 21; 64; 32; 16; 8; 4; 2; 1]`
- ...

This is all computed using the f function listed above, and we will never come near numbers of the size where we do not know if Collatz sequences exists.

Write a function `collatz : int -> int list` that given an integer `x`

- returns the Collatz sequence of `x` if `x` is positive
- fails, using `failwith`, with the message `"Non positive number: <x>"`, where `<x>` is the number that the function was called with if `x` is negative. This case is useful if you start from a high number that eventually overflows. Your algorithm could otherwise loop forever.

For full credit, make sure that your algorithm runs in linear time with respect to the length of the sequence. In particular, **do not** append single elements to the end of lists, i.e. `xs @ [x]`.

Examples:

```
> collatz 1
- val it: int list = [1]

> collatz 2
- val it: int list = [2; 1]

> collatz 3
- val it: int list = [3; 10; 5; 16; 8; 4; 2; 1]

> collatz 42
- val it: int list = [42; 21; 64; 32; 16; 8; 4; 2; 1]

> collatz 1048574
fails with "Non positive number: -1970444364" and most likely a stack trace
```

3.2: Even and odd Collatz sequence elements

Create a function `evenOddCollatz` of type `int -> int * int` that given an integer `x` returns a pair with two integers where

- The first element contains the number of even elements in the Collatz sequence of `x`
- The second element contains the number of odd elements in the Collatz sequence of `x`

Examples:

```
> evenOddCollatz 1
- val it: int * int = (0, 1)

> evenOddCollatz 2
- val it: int * int = (1, 1)

> evenOddCollatz 3
- val it: int * int = (5, 3)

> evenOddCollatz 77031
- val it: int * int = (221, 130)
```

3.3: Maximum length Collatz Sequence

Create a function `maxCollatz : int -> int -> int * int` that given two integers `x` and `y`, where `x <= y` (you can ignore the case where `x > y`) returns a tuple where

- the first element is a number `a` between `x` and `y` inclusive that has the longest Collatz sequence of all numbers between `x` and `y` inclusive.
- The second element is the length of the Collatz sequence of `a`

Examples:

```
> maxCollatz 1 10
- val it: int * int = (9, 20)

> max Collatz 100 1000
- val it: int * int = (871, 179)

> maxCollatz 1000 1000
- val it: int * int = (1000, 112)
```

3.4: Collecting by length

Create a function `collect : int -> int -> Map<int, Set<int>>` that given two integers `x` and `y`, where `x <= y` (you can ignore the case where `x > y`), returns a map where the keys are the length of the Collatz sequences of the numbers between `x` and `y` inclusive and the values are the sets of numbers whose Collatz sequence has the length of their corresponding key.

Examples: (here 20 and 21 both have Collatz sequences of length 8, for instance)

```
collect 20 30
val it: Map<int,Set<int>> =
  map
    [(8, set [20; 21]); (11, set [24; 26]); (16, set [22; 23]);
     (19, set [28; 29; 30]); (24, set [25]); (112, set [27])]

collect 100 110
val it: Map<int,Set<int>> =
  map
    [(13, set [104; 106]); (26, set [100; 101; 102]); (39, set [105]);
     (88, set [103]); (101, set [107]); (114, set [108; 109; 110])]
```

3.5: Parallel maximum Collatz Sequence

Create a function `parallelMaxCollatz : int -> int -> int -> int` that given integers `x`, `y`, and `n`, where `x <= y` and `n` divides `y - x` evenly (you can ignore the cases where these constraints do not hold), returns the number `z` between `x` and `y` inclusive that has the longest Collatz sequence. You must do this by spawning `n` threads running in parallel that each handle the `(y - x) / n` numbers.

Hint: Use `maxCollatz` from Q3.3 to find the number with the maximum length Collatz sequence for each thread and then find the actual maximum by comparing the results from the different threads.

Examples:

```
> parallelMaxCollatz 1 1000 1
- val it: int = 871

> parallelMaxCollatz 1 1000 2
- val it: int = 871

> parallelMaxCollatz 1 1000 100
- val it: int = 871

> parallelMaxCollatz 1 1000 500
- val it: int = 871
```

4: Memory Machine (25%)

For this assignment we will use a simple language that stores values in finite memory

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

type stmnt =
| Assign of expr * expr      // Assign value to memory
                             // location
| While  of expr * prog      // While loop

and prog = stmnt list        // Programs are sequences of
                             // statements

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

Important: Note that `stmnt` and `prog` are defined by mutual recursion and functions created on them should also be mutually recursive.

Programs operate on finite memory where memory adresses are represented as positive integers (including zero) and we store integers in memory. We will use the syntax `{v0, ..., v(n - 1)}` to represent a memory block of size `n` with `vi` representing the value stored at memory adress `i`. We use this syntax in examples as well and your output will differ depending on your representation of memory. Moreover,

- `Lookup e`, represents the value stored at memory address `e`. For instance, if the value `42` is stored in memory address `2` (in memory `{0, 0, 42, 0, 0}`, for instance) then `Lookup (Num 2)` evaluates to `42`.
- `Assign(e1, e2)` represents updating the value stored in memory address `e1` by `e2`
- `While(e, p)` terminates when `e` evaluates to `0` and loops otherwise.

For readability we use syntactic sugar

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

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

```
// Starting from memory {0, 0, 2, 0}
let fibProg x =
    [Num 0 .<-. Num x       // {x, 0, 2, 0}
     Num 1 .<-. Num 1       // {x, 1, 2, 0}
     Num 2 .<-. Num 0       // {x, 1, 0, 0}
     While (Lookup (Num 0),
            [Num 0 .<-. Lookup (Num 0) .-. Num 1
             Num 3 .<-. Lookup (Num 1)
             Num 1 .<-. Lookup (Num 1) .+. Lookup (Num 2)
             Num 2 .<-. Lookup (Num 3)
            ]) // after loop {0, fib (x + 1), fib x, fib x}
     ]
```

To work, this program must be run from a memory block with at least size `4`. It stores the number to iterate over at memory address `0`, memory addresses `1` and `2` are used for the actual computation, and memory address `3` is used as a temporary variable. For instance, running `fibProg 10` from the memory block `{0, 0, 0, 0}` changes the memory to `{0, 89, 55, 55}` where the result of the computation ends up at memory address `2` (the 10th Fibonacci number is 55).

Q4.1: Memory blocks

For this assignment, and the remainder of the exam, you may use imperative types using mutable data to update and read from memory blocks, but nothing else.

Create a type `mem` that represents a memory block. As mentioned before, addresses are non-negative integers and values are integers. For full marks memory lookup and assignment must be done in at most logarithmic time, but constant time is possible if you use a mutable data structure.

Create a function `emptyMem : int -> mem` that given an integer `x` returns a memory block of size `x` filled with zeroes. You may assume that `x` is non-negative and do not have to handle cases where it is not.

Create a function `lookup : mem -> int -> int` that given a block of memory `m` and a memory index `i` returns the value stored in `m` at position `i`. You my assume that `i` is within the range of valid memory addresses of `m` and do not have to handle cases where it is not. For full credit lookup must take at most logarithmic time.

Create a function `assign : mem -> int -> int -> mem` that given a memory block `m`, a memory index `i`, and a value `v`, returns a memory block `m'` that is identical to `m` except for memory address `i` that is set to `v`. For full credit this function must run in at most logarithmic time. You may use imperative features that update `m`, and return the same, but updated, `m` to make it run in constant time.

Examples:

```
> let m = emptyMem 5
- val m: mem = {0, 0, 0, 0, 0}

> lookup m 4
- val it: int = 0

> lookup m -23
fails in whichever way you want (easiest is just not to handle this case)

> assign m 2 42
- val it: mem = {0, 0, 42, 0, 0}
```

Warning: If you use mutable state then remember that any changes you do to memory blocks in testing persists which can lead to unexpected results when running consecutive tests if the memory is not being reset between runs.

Q4.2: Evaluation

Create a function `evalExpr : mem -> expr -> int` that given a memory block `m` and an expression `e` where `e` is

- `Num x`, returns `x`
- `Lookup e'`, returns the value stored at memory index obtained from evaluating `e'` in `m`. You may assume that evaluating `e'` results in a valid memory address in `m`.
- `Plus (e1, e2)` returns the evaluation of `e1` in `m` plus the evaluation of `e2` in `m`
- `Minus (e1, e2)` returns the evaluation of `e1` in `m` minus the evaluation of `e2` in `m`

Create mutually recursive functions

- `evalStmnt : mem -> stmnt -> mem` that given a memory block `m` and a statement `s` where `s` is
  - `Assign (e1, e2)` updates `m` at position for the evaluation of `e1` to the evaluation of `e2` and returns m.
  - `While(e, p)`
    - if `e` evaluates to `0` return `m`
    - evaluate the program `p @ [While(e, p)]` in memory block `m` otherwise (you are allowed to append a singleton element to the end of a list here)
- `evalProg : mem -> prog -> mem` that given a memory block `m` and a program `p` where `p` is
  `[]` *returns* `m`
  `[s1; ...; sn]` returns `evalStmnt (... (evalStmnt s1 m) ...) sn`, i.e. evaluating `s1` in `m`, evaluating `s2` in the result of this evaluation, and so on and ultimately evaluating `sn` in the state obtained by evaluating all previous statements. This case works for lists containing at least one element.

Examples:

```
// to create {0, 0, 0, 0} use `emptyMem 4`
//
// To create {0, 0, 42, 0, 0} use
// `assign (emptyMem 5) 2 42`

> evalExpr {0, 0, 42, 0, 0} (Num 5)
- val it: int = 5

> evalExpr {0, 0, 42, 0, 0} (Lookup (Num 2))
- val it: int = 42

> evalExpr {0, 0, 42, 0, 0} (Plus (Lookup (Num 2), Num 5))
- val it: int = 47

> evalStmnt {0, 0, 42, 0, 0} (Assign (Num 4, Num 5))
- val it: mem = {0, 0, 42, 0, 5}
> evalProg {0, 0, 0, 0} (fibProg 10)
- val it: mem = {0, 89, 55, 55}

> lookup (evalProg {0, 0, 0, 0} (fibProg 10)) 2
- val it: int = 55

> [0..3] |>
  List.map (fun x -> Assign(Num x, Num x)) |>
  evalProg {0, 0, 0, 0}
- val it: mem = {0, 1, 2, 3}
```

Q4.3: State Monad

For this assignment we will be using a state monad to hide the memory block. 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 memory block from Q4.1).

```
type StateMonad<'a> = SM of (mem -> ('a * mem) 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 m (SM f) = f m
```

Create a function `lookup2 : int -> SM<int>` that given a memory index `i` returns the value stored at that index of the memory block in the state monad if `i` is a valid memory address (`0 <= i < size of memory block`) and fails otherwise (using monadic `fail` or `None`, depending on where you fail, not `failwith`).

Create a function `assign2 : int -> int -> SM<unit>` that given an index `i` and a value `v` updates the value stored at memory address `i` to `v` and returns `()`. The function should fail if `i` is not a valid memory adress (using monadic `fail` or `None`, depending on where you fail, not `failwith`).

```
// to create {0, 0, 0, 0, 0} use `emptyMem 5`

> lookup2 2 |> evalSM {0, 0, 0, 0, 0} |> Option.map fst
- val it: int option = Some 0

> lookup2 -23 |> evalSM {0, 0, 0, 0, 0}  |> Option.map fst
- val it: int option = None

> assign2 2 42 >>>= lookup2 2 |> evalSM {0, 0, 0, 0, 0} |> Option.map fst
- val it: int option = Some 42

> assign2 2 42 >>>= lookup2 4 |> evalSM {0, 0, 0, 0, 0} |> Option.map fst
- val it: int option = Some 0

> assign2 2 42 |> evalSM {0, 0, 0, 0, 0} |> Option.map snd
- val it: int option = Some {0, 0, 42, 0, 0}
```

Q4.4: 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 memory block is abstracted by the state monad and failures are caught using monadic fail.

Create mutually recursive functions `evalStmnt2 : stmnt -> SM<unit>` and `evalProg2 : prog -> SM<unit>` that work exactly like `evalStmnt` and `evalProg` respectively but where the memory block is abstracted by the state monad.

Examples:

```
// to create {0, 0, 0, 0} use `emptyMem 4`

// To create {0, 0, 42, 0, 0} use
// `assign (emptyMem 5) 2 42`

> (Num 5) |> evalExpr2 |> evalSM {0, 0, 0, 0} |>
  Option.map fst
- val it: int option = Some 5

> Lookup (Num 2) |> evalExpr2 |> evalSM {0, 0, 42, 0, 0} |>
  Option.map fst
- val it: int option = Some 42

> Plus (Lookup (Num 2), Num 5) |> evalExpr2 |>
  evalSM {0, 0, 42, 0, 0} |> Option.map fst
- val it: int option = Some 47

> Assign (Num 4, Num 5) |> evalStmnt2 |>
  evalSM {0, 0, 42, 0 0} |> Option.map snd
- val it: mem option = Some {0, 0, 42, 0, 5}

> evalProg2 (fibProg 10) |> evalSM {0, 0, 0, 0} |>
  Option.map snd
- val it: mem option = Some {0, 89, 55, 55}

> evalProg2 (fibProg 10) >>>= lookup2 2 |>
  evalSM {0, 0, 0, 0} |> Option.map fst
val it: int option = Some 55
```

Q4.5: Parsing

Finally create a parser for the expressions of our language only using the following grammar

```
Expressions
e, e1, e2 ::=
        x         parses to (Num x) (if x is an integer)
        [e]       parses to (Lookup e)
        e1+e2     parses to e1 .+. e2
        e1-e2     parses to e1 .-. e2
```

For simplicity, there are no white spaces in our expressions, i.e. `5+3` is valid, but `5 + 3` is not.

Use the following code skeleton

```
let ParseExpr, eref = createParserForwardedToRef<expr>()
let ParseAtom, aref = createParserForwardedToRef<expr>()

let parseExpr = <Parse addition and minus>
let parseAtom = <Parse numbers and lookups>

do aref := parseAtom
do eref := parseExpr
```

Fill in the code for `parseExpr` and `parseAtom`. Note that they form a parsing hierarchy (arithmetic operations first, and numbers and lookups after).

You may, but are not required to, create any auxiliary functions you want. This assignment is solvable by just filling in the code marked in `<...>`. Auxiliary functions may be good to structure your own code but nothing else has been left out that you have to code.

Refer to your own parser from Assignment 7 if you get stuck.

Examples:

```
> "5+4" |> run parseExpr |> getSuccess
- val it: expr = Plus (Num 5, Num 4)

> "[4+[3]]" |> run parseExpr |> getSuccess
- val it: expr =  Lookup (Plus (Num 4, Lookup (Num 3)))

> "[5 + 4]" |> run parseExpr |> getSuccess
- fails since there are spaces in the expression
```