Thursday June 3, 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` 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: Dungeon crawler (25%)

Consider the following types that help model a position on a grid and a facing direction.

```
type direction = North | East | South | West
type coord     = C of int * int
```

The center of the coordinate system is, for instance, modelled as `C (0, 0)`.

Question 1.1

Create the following functions:

- `move : int -> direction -> coord -> coord` that given a distance `dist`, a direction `dir` and a coordinate `C(x, y)` returns the same coordinate but moved `dist` steps in direction `dir` where
  - `North` decreases the `y` value
  - `South` increases the `y` value
  - `West` decreases the `x` value
  - `East` increases the `x` value
- `turnRight : direction -> direction` that given a direction `dir` returns `dir` turned a quarter turn to the right.
- `turnLeft : direction -> direction` that given a direction `dir` returns `dir` turned a quarter turn to the left.

Examples:

```
> move 10 North (C (0, 0));;
val it : coord = C (0, -10)

> turnRight North;;
val it : direction = East

> turnLeft North;;
val it : direction = West
```

Question 1.2

A position is composed of a coordinate and a facing direction. A move can either turn left, turn right, or move forwards in the facing direction a certain number of steps.

```
type position = P of (coord * direction)
type move     = TurnLeft | TurnRight | Forward of int
```

Create a function `step : position -> move -> position` than given a position `p` and a move `m` returns the position `p` but with

- the direction turned right if `m` is equal to `TurnRight`
- the direction turned left if `m` is equal to `TurnLeft`
- the coordinate moved forward `x` steps in the facing direction if `m` is equal to `Forward x`.

Examples:

```
> step (P (C (0, 0), North)) TurnRight;;
val it : position = P (C (0, 0), East)

> step (P (C (0, 0), North)) TurnLeft;;
val it : position = P (C (0, 0), West)

> step (P (C (0, 0), North)) (Forward 10);;
val it : position = P (C (0, -10), North)
```

Question 1.3

Create a recursive function `walk : position -> move list -> position` that given a starting position `p` and a list of moves `ms` executes all the moves in the list in order and returns the final position (the first move in `ms` is applied to the starting position, the second move in `ms` to the resulting position from the first move, and so on.)

Create a non-recursive function `walk2`, using higher-order functions from the list library, that behaves in exactly the same way as `walk`.

Examples:

```
> walk (P (C (0, 0), North)) [TurnRight; Forward 10; TurnLeft];;
val it : position = P (C (10, 0), North)

> walk2 (P (C (0, 0), North)) [TurnRight; Forward 10; TurnLeft];;
val it : position = P (C (10, 0), North)
```

Question 1.4

Create a standard recursive function (not a tail-recursive one) `path : position -> move list -> coord list` that given a starting position `p` and a list of moves `ms` returns the list of positions that the moves from `ms` pass through, plus the starting position. You may assume that you teleport between positions so `Forward 10` should not add a list of 10 coordinates, just one. You should not add coordinates to the list when you turn.

The complexity of this function must at most be linear.

Examples:

```
> path (P (C (0, 0), North)) [TurnRight; Forward 10; TurnLeft];;
val it : coord list = [C (0, 0); C (10, 0)]

> path (P (C (0, 0), North))
       [Forward 5; TurnRight; Forward 5; TurnRight;
        Forward 5; TurnRight; Forward 5];;
val it : coord list = [C (0, 0); C (0, -5); C (5, -5); C (5, 0); C (0, 0)]
```

Question 1.5

Create a tail-recursive version of `path`, called `path2`, that uses an accumulator that does exactly the same thing as `path`.

Question 1.6

Your solution for `path` is not tail recursive. 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).

Create a tail-recursive version of `path` called `path3`, using continuations (not an accumulator), that does exactly the same thing as `path`.

2: Code Comprehension (25%)

Consider and run the following three functions

```
let foo f =
  let mutable m = Map.empty
  let aux x =
    match Map.tryFind x m with
    | Some y when Map.containsKey x m -> y
    | None    ->
      m <- Map.add x (f x) m; f x

  aux

let rec bar x =
  match x with
  | 0 -> 0
  | 1 -> 1
  | y -> baz (y - 1) + baz (y - 2)

and baz = foo bar
```

Question 2.1

- What are the types of functions `foo`, `bar`, and `baz`?
- What do functions `foo` and `baz` do (skip `bar`)? Focus on what they do rather than how they do it.
- The function `foo` uses a mutable variable.
  - What function does it serve (why is it there)?
  - What would happen if you removed the `mutable` keyword from the line `let mutable m = Map.empty`? Would the function `foo` still work? If yes, why; if no, why not?
- What would be appropriate names for functions `foo`, `bar`, and `baz`?

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 baz = foo bar` to `let baz = foo bar`)?

Question 2.3

The function `foo` generates a warning during compilation: `Warning: Incomplete pattern matches on this expression.`.

- Why does this happen, and where?
- For these particular three functions will this incomplete pattern match ever cause problems for any possible execution of `baz`? If yes, why; if no, why not.
- The function `foo` has two redundant computations and is hence not as efficient as it could be. What are these two computations and why are they redundant?
  Write a function `foo2` that does exactly the same thing as `foo` except that it does not generate any warnings and is where these two redundant computations have been eliminated.

Question 2.4

Consider the following combined version of `bar` and `baz`

```
let rec barbaz x =
  let baz = foo barbaz
  match x with
  | 0 -> 0
  | 1 -> 1
  | y -> baz (y - 1) + baz (y - 2)
```

Without explicitly timing the execution times, compare the execution times of `baz` and `barbaz`. One is slower than the other. Why? You do not have to give exact times, just spot which one is slower and explain why.

Hint: You should start noticing a difference in execution times at around `baz 30` and `barbaz 30` respectively but feel free to go higher if you wish (not much though or you will be waiting for a while on one of them).

Question 2.5

Write an infinite sequence `bazSeq : int seq` such that the first element of the sequence is equal to `baz 0`, the second to `baz 1`, the third to `baz 2` and so on.

For full credit it must be close to instantanous to access large indexes of the sequence. Do not worry when the resulting integers overflow.

Examples:

```
- Seq.item 10 bazSeq
> val it : int = 55

> Seq.item 100100 bazSeq;;
val it : int = -472869427
```

Note: The integers have overflowed for the last example. This is perfectly fine.

3: Guess the next sequence element (25%)

Consider the following sequence of numbers

```
1
11
21
1211
111221
312211
```

If you are a group of friends a fun challenge can be to have them find the next number in the sequence, but here we are giving the solution away.

The easiest way to think of this sequence is to read out loud how many numbers you see up until the next one on a single row. For example

- The initial number is `1`
- On the previous line you see one one, so the next number is `11`.
- On the previous line you see two ones, so the next number is `21`
- On the previous line you see one two, and finally one one, so the next number is `1211`
- On the previous line you see one one, then one two, and finally two ones, so the next number is `111221`
- On the previous line you see three ones, then two twos, and finally one one, so the next number is `312211`
- and so on.

The special case you need to consider is if you start from a number like `1111111111` (ten ones). The sequence would in that case look as follows:

```
1111111111
101
111011
311021
1321101211
```

Hint: For these questions you can gain a lot by using functions from the standard library.

Question 3.1

Create a type `element` that would be a good type for a single element of this sequence. There must not be a limit on the size of these elements (other than memory). Why is this a good representation?

Warning: An integer or unsigned integer is not a good fit as these numbers will get very large and should not really be thought of as numbers in the first place (we will not be doing any arithmetic on them, just computing elements of the sequence).

Question 3.2

Create a function `elToString : element -> string` that given an element `el` returns its string representation.

Create a function `elFromString : string -> element` that given a string `s` returns the corresponding element. You may assume that the string will always be well formed (it will only contain numbers.)

For these examples the notation `<element>` will mean an element of the sequence using your `element` type, `<1113221>` for instance.

```
> elToString <1113221>;;
val it : string = "1113221"

> elFromString "1113221";;
val it : element = <1113221>

> elToString (elFromString "1113221")
- val it : string = "1113221"
```

Question 3.3

Create a function `nextElement : element -> element` that given an element returns the next element.

```
> "1" |> elFromString |> nextElement |> elToString;;
val it : string = "11"

> "1" |> elFromString |> nextElement |> nextElement |> elToString;;
val it : string = "21"

> "1" |> elFromString |> nextElement |> nextElement |>
         nextElement |> nextElement |> nextElement |> elToString;;
val it : string = "312211"

> "11111111112222222" |> elFromString |> nextElement |> elToString;;
val it : string = "10172"
```

Question 3.4

Create a sequence `elSeq : element -> element seq` that given a starting element `el` uses `Seq.unfold` to create the infinite sequence with starting element `el`.

Create a sequence `elSeq2 : element -> element seq` that behaves exactly the same as `elSeq` but which uses sequence comprehension instead.

Documentation for `Seq.unfold` can be found here

Why would `Seq.initInfinite` not be an appropriate choice to write a function like `elSeq`?

Question 3.5

Using the `JParsec` library provided in the project write a parser combinator `elParse : string -> Parser<element>` that given a string `str` parses `str` into an `element`. Unlike `elFromString` you **may not** assume that the input is well formed. The parser should return an error if run on ill-formed input.

Create a function `elFromString2 : string -> element` that given a string `str` runs the parser `elParse` on `str` and returns the corresponding element. The function should fail if the string is ill-formed. This function should behave the same as `elFromString` except that

- it uses parser combinators through the `elParse` parser.
- it fails if the input string `str` is ill formed and `elParse` fails to parse `str`.

Examples:

```
> elFromString2 "1113221";;
val it : element = <1113221>

> elFromString2 "Hello world!";;
<fails in any way you wish >
```

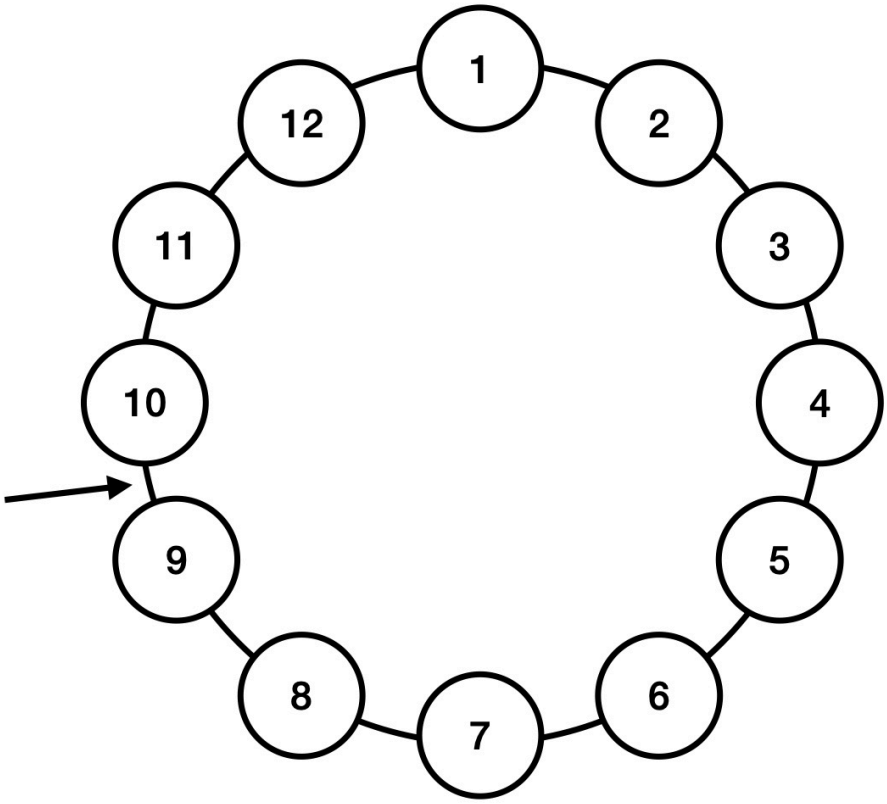
4: Rings (25%)

For this assignment we will use the following notation for lists.

- ϵ represents the empty list.
- $|\vec{a}|$ represents the the length of the list $\vec{\alpha}$.
- $x \cdot \vec{a}$ represents concatenating the element x to the front of the list $\vec{a}$
- $\vec{a} \cdot \vec{b}$ represents appending the two lists $\vec{a}$ and $\vec{b}$ (the $(\cdot)$ operator is overloaded for both concatenation and append) .
- $\vec{a}^{-}$ represents the reversed list $\vec{a}$.

A ring (not to be confused with the algebraic ring) is a data structure containing values that allows you to go both clockwise and counter clockwise in amortized constant time.

A visual representation can be found here



where the current position of the ring is between the numbers 9 and 10 and the structure continues clockwise with the number 10 and counter clowise with the number 9 (and so on).

A ring $(\vec{a}, \vec{b})$ consists of two standard lists $\vec{a}$ and $\vec{b}$ where:

- the list $\vec{a}$ represents the elements clockwise of the current position
- the list $\vec{b}$ represents the elements counter clockwise of the current position
- The next clockwise element of the ring $(\vec{a}, \vec{b})$ is
  - the head of $\vec{a}$ if $\vec{a}$ is non-empty
  - the head of $\vec{b}^{-}$ if $\vec{a}$ is empty.
- The next counter-clockwise element of the ring $(\vec{a}, \vec{b})$ is
  - the head of $\vec{b}$ if $\vec{b}$ is non-empty
  - the head of $\vec{a}^{-}$ if $\vec{b}$ is empty.

There are several valid instantiations of the ring in the image above, but three of them are:

1. $(\epsilon, \quad 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 12 \cdot 11 \cdot 10 \cdot \epsilon)$
2. $(10 \cdot 11 \cdot 12 \cdot 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot 8 \cdot 9 \cdot \epsilon, \quad \epsilon)$
3. $(10 \cdot 11 \cdot 12 \cdot 1 \cdot 2 \cdot 3 \cdot 4 \cdot \epsilon, \quad 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot \epsilon)$

A part of getting this to work is to move elements between the two lists in an efficient way. We will get back to the exact details for Question 4.3, but, to offer some brief intuition before then, moving counter clockwise in the ring $(x \cdot \vec{a}, y \cdot \vec{b})$ results in the ring $(y \cdot x \cdot \vec{a}, \vec{b})$ and moving clockwise results in the ring $(\vec{a}, x \cdot y \cdot \vec{b})$. Taking the third example from above would change the ring from

$$(10 \cdot 11 \cdot 12 \cdot 1 \cdot 2 \cdot 3 \cdot 4 \cdot \epsilon, \quad 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot \epsilon)$$

to

$$(9 \cdot 10 \cdot 11 \cdot 12 \cdot 1 \cdot 2 \cdot 3 \cdot 4 \cdot \epsilon, \quad 8 \cdot 7 \cdot 6 \cdot 5 \cdot \epsilon)$$

if moving counter clockwise.

Question 4.1

Provide a good datatype `'a ring` that represent rings as described above.

Question 4.2

Create a function `length` of type `'a ring -> int` that given a ring $(\vec{a}, \vec{b})$ returns $|\vec{a}| + |\vec{b}|$

Create a function `ringFromList` of type `'a list -> 'a ring` that given a list $\vec{a}$ returns the ring $(\epsilon, \vec{a})$.

Create a function `ringToList` of type `'a ring -> 'a list` that given a ring $(\vec{a}, \vec{b})$ returns the elements in counter-clockwise order (the list $\vec{b} \cdot \vec{a}^{-}$).

Note: the new list created by `toList` starts with the element directly counter clockwise of the current position (the head of $\vec{b}$) and ends with the element directly clockwise of the current position (the head of $\vec{a}$).

```
> ringToList (ringFromList [1;2;3;4;5]);;
val it : int list = [1; 2; 3; 4; 5]

> length (ringFromList [1;2;3;4;5]);;
val it : int = 5

> ringToList (ringFromList [1;2;3;4;5]);;
val it : int list = [1; 2; 3; 4; 5]
```

Question 4.3

Create the following functions (precise algorithms are given below):

- `empty` of type `'a ring` that creates the empty ring (ϵ, ϵ).
- `push : 'a -> 'a ring -> 'a ring` that given an element `x` and a ring `r` pushes `x` to the first counter clockwise position of `r`.
- `peek : 'a ring -> 'a option` that given a ring `r` returns `Some` (the next counter-clockwise element

of `r`) if such an element exists and `None` otherwise.
- `pop : 'a ring -> ('a ring) option` that given a ring `r` returns `Some` (`r` with the next counter-clockwise element removed) if such an element exists, and `None` otherwise.
- `cw : 'a ring -> 'a ring` that given a ring `r` moves the current position one step clockwise.
- `ccw : 'a ring -> 'a ring` that given a ring `r` moves the current position one step counter clockwise.

When writing these funtions you have to be a bit carfeul with the two internal lists of the ring. If you want to take `cw` $(\epsilon, \vec{b})$ for a non-empty $\vec{b}$ (which intuitively means that you have the entire list $\vec{b}$ counter clockwise of the current position) then you cannot immediately do that but you first have reverse $\vec{b}$ and take the head of the result giving you the ring $(\vec{c}, x)$ where $\vec{b}^{-} = x \cdot \vec{c}$.

A complete list of the algorithms you will need are given here:

$$\text{empty} = (\epsilon, \epsilon)$$

$$\text{push } x\ (\vec{a}, \vec{b}) = (\vec{a}, x \cdot \vec{b})$$

$$\text{cw } (\vec{a}, \vec{b}) = \begin{cases} (\epsilon, \epsilon) & \text{if } \vec{a} = \epsilon \text{ and } \vec{b} = \epsilon \\ (\vec{c}, x) & \text{if } \vec{a} = \epsilon \text{ and } \vec{b}^{-} = x \cdot \vec{c} \\ (\vec{c}, x \cdot \vec{b}) & \text{if } \vec{a} = x \cdot \vec{c} \end{cases}$$

$$\text{peek } (\vec{a}, \vec{b}) = \begin{cases} \text{Some } x & \text{if } \vec{a}^{-} = x \cdot \_ \text{ and } \vec{b} = \epsilon \\ \text{Some } x & \text{if } \vec{b} = x \cdot \_ \\ \text{None} & \text{otherwise} \end{cases}$$

$$\text{pop } (\vec{a}, \vec{b}) = \begin{cases} \text{Some } (\epsilon, \vec{c}) & \text{if } \vec{a}^{-} = \_ \cdot \vec{c} \text{ and } \vec{b} = \epsilon \\ \text{Some } (\vec{a}, \vec{c}) & \text{if } \vec{b} = \_ \cdot \vec{c} \\ \text{None} & \text{otherwise} \end{cases}$$

$$\text{ccw } (\vec{a}, \vec{b}) = \begin{cases} (\epsilon, \epsilon) & \text{if } \vec{a} = \epsilon \text{ and } \vec{b} = \epsilon \\ (x, \vec{c}) & \text{if } \vec{a}^{-} = x \cdot \vec{c} \text{ and } \vec{b} = \epsilon \\ (x \cdot \vec{a}, \vec{c}) & \text{if } \vec{b} = x \cdot \vec{c} \end{cases}$$

Examples:

```
> ringToList (empty : int ring);;
val it : int list = []

> [1;2;3;4;5] |> ringFromList |> push 6 |> ringToList;;
val it : int list = [6; 1; 2; 3; 4; 5]

> [1;2;3;4;5] |> ringFromList |> peek;;
val it : int = 1

> ([] : int list) |> ringFromList |> peek;;
val it : int option = None

> [1;2;3;4;5] |> ringFromList |> pop |> ringToList;;
val it : int list = [2; 3; 4; 5]

> [1;2;3;4;5] |> ringFromList |> ccw |> ccw |> ringToList;;
val it : int list = [3; 4; 5; 1; 2]

> [1;2;3;4;5] |> ringFromList |> cw |> cw |> ringToList;;
val it : int list = [4; 5; 1; 2; 3]
```

Question 4.4

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

```
type StateMonad<'a, 'b> = SM of ('b ring -> ('a * 'b ring) 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 ring" refers to the internal state of the state monad):

- `smLength : SM<int>` that returns the length of the ring, but leaves it unchanged in the state.
- `smPush : 'a -> SM<unit, 'a>` that given an element `x` updates the internal state by pushing the element `x` to the the first counter clockwise position of the ring.
- `smPop : SM<'a', 'a>` that pops the first counter-clockwise element from the ring and returns it. The function should fail if the ring is empty.
- `smCW : SM<(), 'a>` that moves the current position of the ring one step clockwise.
- `smCCW : SM<(), 'a>'` that moves the current position of the ring one step counter clockwise.

Examples:

```
> [1;2;3;4;5] |> ringFromList |> evalSM smLength |> Option.get |> fst;;
val it : int = 5

> [1;2;3;4;5] |> ringFromList |> evalSM (smPush 6) |>
  Option.get |> snd |> ringToList;;
val it : int list = [6; 1; 2; 3; 4; 5]

> [1;2;3;4;5] |> ringFromList |> evalSM smPop |>
  Option.get |> snd |> ringToList;;
val it : int list = [2; 3; 4; 5]

> ([] : int list) |> ringFromList |> evalSM smPop;;
val it : (int * int clist) option = None

> [1;2;3;4;5] |> ringFromList |> evalSM (smCW >>>= smCW) |>
  Option.get |> snd |> ringToList;;
val it : int list = [4; 5; 1; 2; 3]
```

```
> [1;2;3;4;5] |> ringFromList |> evalSM (smCCW >>>= smCCW) |>
  Option.get |> snd |> ringToList;;
val it : int list = [3; 4; 5; 1; 2]
```

Question 4.5

Finally let's create an algorithm that incrementally removes adjacent elements, in counter clockwise order, whose sum is an even number from a ring . 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 a function `ringStep : SM<unit, int>` that returns a state monad which does nothing if the length of the ring is strictly smaller than two elements (one element or smaller). Otherwise it checks the first two counter clockwise elements x and y of the ring and

- removes them if their sum is even.
- leave them unchanged otherwise but move the starting position one step counter clockwise.

Create a function `iterRemoveSumEven : uint32 -> SM<unit, int>` that given an unsigned integer `x` runs the `ringStep` function `x` times. Given a large enough number for `x` this will remove all adjacent pairs of a ring that sum to an even number.

Examples:

```
> [1;2;2;4;5] |> ringFromList |> evalSM ringStep |>
  Option.get |> snd |> ringToList;;
val it : int list = [2; 2; 4; 5; 1]

> [1;2;2;4;5] |> ringFromList |> evalSM (ringStep >>>= ringStep) |>
  Option.get |> snd |> ringToList;;
val it : int list = [4; 5; 1]

> [1;2;3;4;5] |> ringFromList |> evalSM (iterRemoveSumEven 0u) |>
  Option.get |> snd |> ringToList;;
val it : int list = [1; 2; 3; 4; 5]

> [1;2;3;4;5] |> ringFromList |> evalSM (iterRemoveSumEven 10u) |>
```

```
  Option.get |> snd |> ringToList
val it : int list = [3]

> [1;2;3;4;5;6] |> ringFromList |> evalSM (iterRemoveSumEven 10u) |>
  Option.get |> snd |> ringToList;;
val it : int list = [5; 6; 1; 2; 3; 4]
```