Assignment 6

For this assignment we will finalise the domain-specific language that we use to model the scrabble boards we will be playing on.

1: Handling errors and state

The answers to these assignments go in `StateMonad.fs`. Its interface, which you do not have to touch, is in `StateMonad.fsi`.

Green Exercises

Exercise 6.1

Create a function `pop : SM<unit>` that pops the top variable environment off the stack of the state. This function will be called when we exit a branch from an if-statement or a while loop. You do **not** have to consider the case when the stack is empty and can either have the program fail, throw an exception, or just not handle the case at all. Ending up in an empty stack is not the fault of the programmer, but an error in the evaluation functions that we write later on.

Exercise 6.2

Create a function `wordLength : SM<int>` that returns the length of the word of the state and leaves the state unchanged.

Exercise 6.3

Create a function `characterValue : int -> SM<char>` that given a position `pos` returns the character at position `pos` of the word in the state. If `pos` is not a valid word index then fail with `IndexOutOfBounds pos`. The state itself is left unchanged.

Exercise 6.4

Create a function `pointValue : int -> SM<int>` that given a position `pos` returns the point value of the character at position `pos` of the word in the state. If `pos` is not a valid word index then return `IndexOutOfBounds pos`. The state itself is left unchanged.

Yellow Exercise

Exercise 6.5

Using the function `lookup` for inspiration, create a function `update : string -> int -> SM<unit>` that given a variable name `x` and a value `v` updates the first ocurring occurrence of `x` in the stack with the variable `v`. The rest of the state is left untouched. If `x` does not appear in the stack, the error `VarNotFound x` is returned.

Exercise 6.6

Create a function `declare : string -> SM<unit>` that given a variable name `x` adds that variable to the top variable environment of the stack with a starting value of `0`. In addition

1. If the variable has the same name as one of the reserved words, fail with `ReservedName x`.
2. If the variable already exists in the top variable environment, fail with `VarExists x`
3. If the stack is empty then the function is undefined and you can do anything you like, including returning the *wrong* answer.

2: Evaluation functions using railway-oriented programming

The answers to these assignments go in `Eval.fs`. You may **not** place these functions in `StateMonad.fs` but in `Eval.fs` which imports `StateMonad.fsi`.

Green Exercises

Assignment 6.7

Create a funciton `add : SM<int> -> SM<int> -> SM<int>` that given two numbers `a` and `b` returns `a + b`. The state does not change. You only need to use `>>=`, `ret`, and standard arithmetic to construct your function.

Assignment 6.8

Create a function `div : SM<int> -> SM<int> -> SM<int>` that given two numbers `a` and `b` returns the integer division `a / b` if `b` is not equal to `0` and fails with `DivisionByZero` otherwise. The state does not change. You only need to use `>>=`, `ret`, `fail`, possibly an if-statement, and standard arithmetic to construct your function.

Assignment 6.9

Create functions `arithEval : aExp -> SM<int>`, `charEval : cExp -> SM<char>` and `boolEval : bExp -> SM<bool>` that given an expression evaluates that expression in the input state. You may declare helper functions (as in Assignments 6.7 and 6.8) if you want, or inline them as every individual case is small. You may never expose the state but all state manipulation must be done via the functions described in Section 1. You will, however, find functions that you have created for previous versions of these evaluation functions (like `isLetter` or `isDigit`) from Assignment 3 useful.

The following errors should be raised:

1. `DivisionByZero` whenever we divide or use modulo by `0`.
2. `VarNotFound x` whenever we reference a variable `x` that does not exist in the state.
3. `IndexOutOfBounds x` whenever we try to get a point value or a character from the word at an invalid index `x`.

Note that errors 2 and 3 are all handled by the functions that we created in Section 1 and will be propagated automatically as long as you use `>>=` and `>>>=` properly.

Yellow Exercise

Assignment 6.10

Create a function `evalStmnt : stm -> SM<unit>` that given a statement `stm` evaluates that statement. This function has the same specification as the one you wrote in 3.6 but with three additions:

1. `Declare x` declares a new variable x with a default value of `0` in the top variable environment on the stack. Moreover,
   - If `x` is equal to one of the reserved names then fail with `ReservedName x`
   - If `x` is already declared in the top environment then fail with `VarExists x`
2. Whenever you enter the branch of an if-statement or a while loop you must push an empty variable environment to the top of the stack (use `push` and `>>>=`).
3. Whenever you exit the branch of an if-statement or a while loop you must pop the top variable environment from the stack (use `pop` and `>>>=`).

Note that the errors for `Declare` are already handled by the `declare` function that you wrote in Assignment 6.3 and you do not have to do anything extra here.

3: Evaluation functions using computational expressions

Assignment 6.11

Create functions `arithEval2 : aExp -> SM<int>`, `charEval2 : cExp -> SM<char>`, `boolEval2 : bExp -> SM<bool>`, and `stmntEval2 : stmnt -> SM<unit>` that work in the same way as their corresponding functions from 2 but that use computational expressions in stead.

4: Modelling the Scrabble board

This assignment will be used for the Scrabble project. These assignments are all very small. Both 6.12 and 6.13 boil down to evaluating a statement using `stmntEval` in a specific initial state, that you can create using `mkState`, and then doing a lookup for the `_result_` variable and finally getting the result using `evalSM`. This is the same pattern that we have used in nearly every single example so far.

Red Exercises

Assignment 6.12

For this assignment we will change the definition of `squareFun` from Assigenmnt 3.7 to allow for failures and set the type to

```
type squareFun = word -> int -> int -> Result<int, Error>
```

Create a function `stmntToSquarerFun : stmnt -> squareFun` that given a statement `stm` returns a function that given a word `w`, a position `pos`, and an accumultor `acc` evaluates `stm` in an inital state (created using `mkState`) where:

- The variable environment contains the variables `_pos_`, `_acc_`, and `_result_` where
  - the variable `_pos_` is initialised to `pos`
  - the variable `_acc_` is initialised to acc
  - the variable `_result_` is initialised to `0`
- the word is set to `w`
- the restricted names are `_pos_`, `_acc_`, and `_result_`.

The function should return the value of the `_result_` variable of the final state when the statement has been executed and fail if the evaluation fails. You should use the evaluation function from either Section 2 or Section 3.

Assignment 6.13

So far we have only modelled squares on the board. We will now model the entire board. At their core, boards are modelled as functions from coordinates to square options.

```
type coord = int * int

type boardFun = coord -> Result<squareFun option, Error>
```

The intuition behind this function is that we have a coordinate `coord` that represents the `x` and the `y` coordinate of a board and that the board function returns the square at a certain coordinate and `None` if the square is blank. For a standard board any coordinate outside the board would be empty, but we do support infinite boards as well, or boards with big holes in them.

To create a board functions we will again use our DSL above in a similar way to how we create square functions. This time around, however, the function takes the arguments `_x_` and `_y_` to represent the coordinate and a return variable `_result_` that is an integer representing which square is placed at that coordinate. We will use a map of type `Map<int, squareFun>` to obtain a concrete square from the integer stored in `_result_`.

Create a function `stmntToBoardFun : stmnt -> Map<int, squareFun> -> boardFun` that given a statement `stm` and a a lookup table of identifiers and square functions `squares` runs `stm` in the state where:

- The variable environment contains the variables `_x_`, `_y_`, and `_result_` where:
  - the variables `_x_` and `_y_` are initialised to be the x- and the y-values of the coordinate to the board function
  - the variable `_result_` is initialised to `0`
- The word is set to the empty word (we guarantee it will never be used when evaluating boards)
- `_x_`, `_y_`, and `_result_` are reserved words.

after which it looks up the integer value `id` stored in `_result_` and returns

- `Success (Some sf)`, if looking up the key `id` in `squares` results in `sf`
- `Success None` if the key `id` does not exist in `squares`
- Fails if the evaluation of `stm` fails.

It is important to note that a result of `Success None` is not considered a failure, it just means that the coordinate is empty (outside the board, for instance).

Assignment 6.14

Boards are modelled using the following record type:

```
type board {
  center        : coord
  defaultSquare : squareFun
  squares       : boardFun
}
```

Every board has a center coordinate `center` over which the first word must be placed. Moreover it has a default square, which is the square that is used whenever a letter has already been placed on the board. In standard scrabble this would be the Single Letter Square, as any tile placed on that square can be counted as a single letter for words built later no matter what the original square type was. Finally, the board itself is modelled as a `boardFun` described above.

Create a `function mkBoard : coord -> stmnt -> stmnt -> (int * stmnt) list -> board` that given

- a center coordinate `c`
- a statement representing the default square `defaultSq`
- a statement `boardStmnt` representing the board
- a list of identifiers and statements `ids` representing the different squares on the board

returns the board that has

- the center coordinate `c`
- the default square `defaultSq` compiled with `stmntToSquareFun`
- the squares `boardStmnt` compiled with `stmntToBoardFun` where the square lookup map is derived from `ids` by
  - mapping all pairs `(k, sq)` in `ids` to `(k, sq')` where `sq'` is `sq` compiled with `stmntToSquareFun` (use `List.map`)
  - converting the resulting list to a map (use `Map.ofList`)