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. More precisely we will add

1. arithmetic division
2. arithmetic modulo
3. explicit variable declarations
4. variable shadowing (if-statements and while-loops will allow local variable declarations with their sub-statements). This requirement is necessary in order to have sencible variable declaration as variables declared within a loop would otherwise persist outside of the loop.
5. Restricted names - variable names that store arguments to the function and that may not be re-declared. They may, however, be accessed and modified.

The complete language definition looks as follows

```
type aExp =
| N of int                 (* Integer literal *)
| V of string              (* Variable reference *)

| WL                       (* Word length *)
| PV of aExp               (* Point value lookup at word index *)

| Add of aExp * aExp       (* Addition *)
| Sub of aExp * aExp       (* Subtraction *)
| Mul of aExp * aExp       (* Multiplication *)
| Div of aExp * aExp       (* NEW: Division *)
| Mod of aExp * aExp       (* NEW: Modulo *)

| CharToInt of cExp        (* Cast to integer *)

and cExp =
| C  of char                (* Character literal *)
| CV of aExp                (* Character lookup at word index *)

| ToUpper of cExp           (* Convert character to upper case *)
| ToLower of cExp           (* Convert character to lower case *)

| IntToChar of aExp         (* Cast to character *)

type bExp =
| TT                      (* True *)
| FF                      (* False *)
```

```
| AEq of aExp * aExp    (* Numeric equality *)
| ALt of aExp * aExp    (* Numeric less than *)

| Not of bExp           (* Boolean not *)
| Conj of bExp * bExp   (* Boolean conjunction *)

| IsVowel of cExp       (* Check for vowel *)
| IsConsonant of cExp   (* Check for constant *)

type stm =
| Declare of string        (* NEW: Variable declaration *)
| Ass of string * aExp     (* variable assignment *)
| Skip                     (* Nop *)
| Seq of stm * stm         (* Sequential composition *)
| ITE of bExp * stm * stm (* If-Then-Else statement *)
| While of bExp * stm      (* While statement *)
```

These new statements and operators have the posibility of going wrong - a program can, for instance, divide by zero, declare a variable that has already been declared, or try to reference a variable that has not been declared. Exactly what we consider to be errors will be made clear in Section 1.

To write the evaluation functions for this program we will use railway-oriented programming. More precisely we will use a combination of the state and the error monad as demonstrated in the lecture. The assignment is split into two mandatory parts and two optional part. Focus on the mandatory part, the optional stuff is optional.

1. Create the auxiliary functions needed to work effectively with changing state and errors.
2. Write evaluation functions for our DSL using these functions using railway-oriented programming
3. (optional) Write the same functions as in 2 but using computational expressions.
4. (optional) Write functions to help model the board using either the results from 2 or 3.

Note that 4 does not require 3 to do so if you want to get a head start on the Scrabble project you have the option.

For this assignment you will be given an F# project that you can load. It contains the code for the state monad in a standalone module to ensure that your evaluation functions for parts 2-4 do not break abstraction. You can load these files in VSCode as well, using Visual Studio you should be able to load this project directly. We have also included test cases that we present here for your functions. To run these tests just execute the project. They are also on CodeJudge.

If you have problems loading the project you can create your own and just imports the .fs- and the .fsi-files. Remember that order is important in F# projects. The order used is `StateMonad.fsi`, `StateMonad.fs`, `Eval.fs`, and `Program.fs`. This is also the order to use if you compile from the command line.

Before you start: Insert your definition of `hello` from Assignment 2.13 in Program.v at the marked place. Do note that that assignment erroniously stated that O was worth two points when it is in fact worth only one point. You can fix this if you wish, but none of the tests check for the value of O so it is optional.

1: Handling errors and state

The answers to these assignments go in `StateMonad.fs` and its interface (which you do not have to touch) is in `StateMonad.fsi`.

Before we start, we need to make clear what errors we support and what our states are. We will start with the errors.

```
type Error =
     | VarExists of string      (* Declaring an existing variable in the same
scope *)
     | VarNotFound of string    (* Referincg a variable that does not exist *)
     | IndexOutOfBounds of int (* Trying to get the get the point or character
value
                                  that is outside the index of the word *)
     | DivisionByZero           (* Division or modulo by zero *)
     | ReservedName of string   (* Trying to declare a reserved name *)
```

The state is a record type that stores our stack of maps of varables, the word, and a set of reserved variable names. Note that this is different from our previous implementations where we stord the variable environment and the word separately. Our approach here keeps the state in one place.

```
type State = { vars     : Map<string, int> list
               word     : (char * int) list
               reserved : Set<string> }
```

The intuition of `vars` is that it is a stack of variable bindings where an empty map is pushed on the stack when entering a block (in a while-loop for instance) and popped when you exit the block, effectively forgetting any variable that was declared in the block. This will be made precise soon.

The function `mkState : (string * int) list -> word -> string list -> State` takes an initial set of variable mappings `vs`, a word `w`, a list of reserved symbols `res` and returns a state with `vars` set to the singleton stack containing the names and values from `vs`, word set to `w`, and `reserved` set to `res` converted to a set.

```
let mkState vs w res =
       { vars = [Map.ofList vs];
         word = w;
         reserved = Set.ofList res }
```

For nearly all of the examples in this section we will be using the state

```
let state = mkState [("x", 5); ("y", 42)] hello (Set.ofList ["_pos_";
"_result_"])
```

where we use `hello` from Assignment 2.13. This is thus a state where the variable `x` is mapped to the value `5`, the variable `y` is mapped to the value `42`, the word being used is `hello`, and the reserved names are `_pos_` and `_result_` (similar to the states used in Assignment 3.7).

If you have not done so already now would be a good time to brush up on how records are created, accessed, and modified.

Finally, before we start with the actual assignments for this week, we need to set up the features we need for our railroad. This is very similar to the seciton on Parametric Results from the lecture.

```
type Result<'a, 'b>  =
    | Success of 'a
    | Failure of 'b

type SM<'a> = S (State -> Result<'a * State, Error>)

let evalSM : (s : State) (a : SM<'a>) : Result<'a, Error> =
    match a s with
    | Success (result, _) -> Success result
    | Failure error -> Failure error

let bind (f : 'a -> SM<'b>) (a : SM<'a>) : SM<'b> =
    S (fun s ->
          match a s with
          | Success (b, s') -> f b s'
          | Failure err     -> Failure err)


let ret (v : 'a) : SM<'a> = S (fun s -> Success (v, s))
let fail err     : SM<'a> = S (fun s -> Failure err)

let (>>=)  (a : SM<'a>)   (f : 'a -> SM<'b>) : SM<'b> = bind f x
let (>>>=) (u : SM<unit>) (a : SM<'a>)       : SM<'a> = u >>= (fun () -> a)
```

Note that the type for the state monad is encapsulated in an algebraic datatype with the constructor `S` rather than having a type alias. We do this to be able to put it into a `.fsi` file similar to what you have already done with your dictionaries and multisets for Assignment 4.

Go back to the lecture slides for examples on how these binders are used. Do remember, however, that `>>=` takes an `SM<'a>` and then a function that works with the return value of type `'a`. The operator `>>>=` works in the same way but takes an `SM<unit>` as the first argument and in effect only applies the side effects to the state before returning it's second argument of type

`SM<'a>`. You can think of `>>>=` as a more advanced version of `;` in imperative languages like Java or C# - a programmable semicolon if you will. This will be made clear in the following examples.

We will now start creating infrastructure for updating and accessing the state of our programs. First, consider the following function `push : SM<unit>` that updates the variable state by pushing an empty map to the top of the stack. This function will be called whenever we enter a branch of an if-statement or a while loop.

This description may confuse you. After all, there is no state directly in our type. However, if you unfold the definition of `SM` you will find that `push` also has the type `S (state -> Result<unit * state, Error>)` where the state is present as an argument to the function, and in the return type for the `Success` case, and hence we can reason about it. The reason we don't spell the types out this way is that one of the main selling points of railroad-oriented programming is that it allows us to hide the state within the types and to work at a higher level of abstraction.

```
let push : SM<unit> =
    S (fun s -> Success ((), {s with vars = Map.empty :: s.vars}))
```

Note that the return type is `SM<unit>` which means that there are no visible effects to this computation, but the stack is updated under the surface by modifying the state `s` that is the argument to the returned value of type `SM<unit>`. This is exactly what we want.

A more complicated example is the function that looks up variables from the stack. The function `lookup : string -> SM<int>` takes a variable name `x` and looks up the first occurrence of the variable on the stack. If there is no such variable then the error `VarNotFound x` is returned.

```
let lookup (x : string) : SM<int> =
    let rec aux =
        function
        | []      -> None
        | m :: ms ->
            match Map.tryFind x m with
            | Some v -> Some v
            | None   -> aux ms

    S (fun s ->
          match aux (s.vars) with
          | Some v -> Success (v, s)
          | None   -> Failure (VarNotFound x))
```

The important thing to note here is again that the state does not appear in the type of our function but is abstracted away from completely. We are just getting started, but even now we can create some simple tests and also demonstrate how to connect the functions we have just created.

```
> lookup "x" |>
  evalSM state;;
- val it : Result<int,Error> = Success 5

> push >>>= push >>>= lookup "y" |>
  evalSM state;;
- val it : Result<int,Error> = Success 42

> push >>>= push >>>= push >>>= lookup "z" |>
  evalSM state;;
- val it : Result<int,Error> = Failure (VarNotFound "z")
```

Note that our programs composed of `push` and `lookup` never mention the state. An initial state is only given when evaluating our program. Also note that since `push` has the type `SM<unit>` we use `>>>=` to comopse it with the rest of the program and not `>>=`.

Exercise 6.1

Create a function `pop : SM<unit>` that pops the top variable map 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.

Examples:

```
> push >>>= pop >>>= lookup "x" |>
  evalSM state;;
- val it : Result<int,Error> = Success 5

> pop >>>= push >>>= lookup "x" |>
  evalSM state;;
- val it : Result<int,Error> = Failure (VarNotFound "x")

> pop >>>= pop >>>= lookup "x" |>
  evalSM state;;
- (* Fails in any way you like. If you simply ignore this case the
     program will fail automatically. This is perfectly fine. *)
```

Exercise 6.2

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

Examples:

```
> wordLength >>=
  evalSM state;;
- val it : Result<int,Error> = Success 5
```

Exercise 6.3

Create a function `characterValue : int -> SM<char>` that given a position `pos` returns the character at position `pos` of the word. If `pos` is not a valid word index then return `IndexOutOfBounds pos`.

Examples:

```
> characterValue 0 |> evalSM state;;
- val it : Result<char,Error> = Success 'H'

> characterValue 4 |> evalSM state;;
- val it : Result<char,Error> = Success 'O'

> characterValue 8 |> evalSM state;;
- val it : Result<char,Error> = Failure (IndexOutOfBounds 8)
```

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. If `pos` is not a valid word index then return `IndexOutOfBounds pos`.

Examples:

```
> pointValue 0 |> evalSM state;;
- val it : Result<char,Error> = Success 4

> pointValue 3 |> evalSM state;;
- val it : Result<char,Error> = Success 1

> pointValue 8 |> evalSM state;;
- val it : Result<char,Error> = Failure (IndexOutOfBounds 8)
```

Exercise 6.5

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.

```
> update "x" 7 >>>= lookup "x" |>
```

```
  evalSM state;;
- val it : Result<int,Error> = Success 7

> push >>>= update "x" 7 >>>= lookup "x" |>
  evalSM state;;
- val it : Result<int,Error> = Success 7

> push >>>= update "x" 7 >>>= pop >>>= lookup "x" |>
  evalSM state;;
- val it : Result<int,Error> = Success 7

> pop >>>= update "x" 7 >>>= push >>>= lookup "x" |>
  evalSM state;;
- val it : Result<int,Error> = Failure (VarNotFound "x")

> lookup "x" >>=
  (fun v1 -> lookup "y" >>=
             (fun v2 -> update "x" (v1 + v2))) >>>=
  lookup "x" |>
  evalSM state;;
- val it : Result<int,Error> = Success 47

> lookup "x" >>=
  (fun v1 -> lookup "y" >>=
             (fun v2 -> update "x" (v1 + v2))) >>>=
  lookup "y" |>
  evalSM state;;
- val it : Result<int,Error> = Success 42
```

In particular pay attention to the two last examples and see how we use `>>=`, rather than `>>>=`, to use the result for `lookup` in later parts of our program.

Exercise 6.6

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

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.

Examlpes:

```
> declare "z" >>>= lookup "z" |>
  evalSM state;;
- val it : Result<int,Error> = Success 0

> declare "z" >>>= update "z" 123 >>>= lookup "z" |>
```

```
  evalSM state;;
- val it : Result<int,Error> = Success 123

> declare "x" >>>= lookup "x" |>
  evalSM state;;
- val it : Result<int,Error> = Failure (VarExists "x")

> declare "z" >>>= declare "z" |>
  evalSM state;;
- val it : Result<unit,Error> = Failure (VarExists "z")

> declare "z" >>>= update "z" 123 >>>= push >>>=
  declare "z" >>>= update "z" 456 >>>= lookup "z" |>
  evalSM state;;
- val it : Result<int,Error> = Success 456

> declare "z" >>>= update "z" 123 >>>= push >>>=
  declare "z" >>>= update "z" 456 >>>= pop >>>=
  lookup "z" |>
  evalSM state;;
 val it : Result<int,Error> = Success 123

> declare "_pos_" >>>= lookup "_pos_" |>
  evalSM state;;
- val it : Result<int,Error> = Failure (ReservedName "_pos_")
```

2: Evaluation functions using railway-oriented programming

The answers to these assignments go in `Eval.fs`.

We will now create an evaluation function for our domain specific language using the constructs you have just created above. You may **not** place these functions in `StateMonad.fs` but in their own file that imports `StateMonad.fsi`. Such a file is present in the project provided with this assignment. We have now contained our plumbing in one place and we no longer have to concern ourselves with how the state is updated internally.

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.

Hint: Look at the last two examples of Exercise 6.5 and/or the lecture slides for inspiration.

Examples:

```
> add (ret 5) (ret 7) |>
  evalSM state;;
- val it : Result<int,Error> = Success 12

> add (lookup "x") (lookup "y") |>
  evalSM state;;
- val it : Result<int,Error> = Success 47

> add wordLength (lookup "z") |>
  evalSM state;;
- val it : Result<int,Error> = Failure (VarNotFound "z")
```

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`, and standard arithmetic to construct your function.

Examples:

```
> div (ret 7) (ret 5) |>
  evalSM state;;
- val it : Result<int,Error> = Success 1

> div (lookup "y") (lookup "x") |>
  evalSM state;;
- val it : Result<int,Error> = Success 8

> div wordLength (lookup "z") |>
  evalSM state;;
- val it : Result<int,Error> = Failure (VarNotFound "z")

> declare "z" >>>= div (lookup "x") (lookup "z") |>
  evalSM state;;
- val it : Result<int,Error> = Failure DivisionByZero
```

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 `isVowel`) 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.

Examples:

```
> arithEval (V "x" .+. N 10) |>
  evalSM state;;
- val it : Result<int,Error> = Success 15

> arithEval (WL .*. N 10) |>
  evalSM state;;
- val it : Result<int,Error> = Success 50

> arithEval (CharToInt (CV (N 0))) |>
  evalSM state;;
- val it : Result<int,Error> = Success 72

> arithEval (PV (N -5)) |>
  evalSM state;;
- val it : Result<int,Error> = Failure (IndexOutOfBounds -5)

> arithEval (V "x" .%. N 0) |>
  evalSM state
- val it : Result<int,Error> = Failure DivisionByZero
```

```
> charEval (C 'H') |>
  evalSM state;;
- val it : Result<char,Error> = Success 'H'

> charEval (ToLower (CV (N 0))) |>
  evalSM state;;
- val it : Result<char,Error> = Success 'h'

> charEval (ToUpper (C 'h')) |>
  evalSM state;;
- val it : Result<char,Error> = Success 'H'

> charEval (CV (V "x" .-. N 1)) |>
  evalSM state;;
- val it : Result<char,Error> = Success 'O'
```

```
> boolEval TT |>
  evalSM state;;
```

```
- val it : Result<bool,Error> = Success true

> boolEval FF |>
  evalSM state;;
- val it : Result<bool,Error> = Success false

> boolEval ((V "x" .+. V "y") .=. (V "y" .+. V "x")) |>
  evalSM state;;
- val it : Result<bool,Error> = Success true

> boolEval ((V "x" .+. V "y") .=. (V "y" .-. V "x")) |>
  evalSM state;;
- val it : Result<bool,Error> = Success false

> boolEval (IsVowel (CV (V "x"))) |>
  evalSM state;;
- val it : Result<bool,Error> = Failure (IndexOutOfBounds 5)

> boolEval (IsConsonant (CV (V "x" .-. N 1))) |>
  evalSM state;;
- val it : Result<bool,Error> = Success true

> boolEval (IsVowel (CV (V "x" .-. N 1))) |>
  evalSM state;;
- val it : Result<bool,Error> = Success true
```

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.
3. Whenever you exit the branch of an if-statement or a while loop you must pop the top variable environment from the stack.

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.

For some of the examples we will use the following empty state.

```
let emptyState = mkState [] [] Set.empty
```

Examples:

```
> stmntEval (Ass ("x", N 5)) >>>= lookup "x" |>
  evalSM emptyState;;
- val it : Result<int,Error> = Failure (VarNotFound "x")

> stmntEval (Seq (Declare "x", Ass ("x", N 5))) >>>= lookup "x" |>
  evalSM emptyState;;
- val it : Result<int,Error> = Success 5

> stmntEval
  (Seq (Declare "x",
        Seq (Declare "y",
             Seq (Ass ("x", WL),
                  Ass ("y", N 7))))) >>>=
  lookup "x" >>= (fun vx -> lookup "y" >>= (fun vy -> ret (vx, vy))) |>
  evalSM emptyState;;
- val it : Result<(int * int),Error> = Success (0, 7)

> stmntEval
  (Seq (Declare "x",
        Seq (Declare "y",
             Seq (Ass ("x", WL),
                  Ass ("y", N 7))))) >>>=
  lookup "x" >>= (fun vx -> lookup "y" >>= (fun vy -> ret (vx, vy))) |>
  evalSM state;; (* Running in state, where x is already declared, rather than
emptyState *)
- val it : Result<(int * int),Error> = Failure (VarExists "x")
```

3: (optional) 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.

To set things up you may use the following builder:

```
type StateBuilder() =

member this.Bind(f, x)      = f >>= x
member this.Return(x)       = ret x
member this.ReturnFrom(x) = x
member this.Delay(f)        = f ()
member this.Combine(a, b) = a >>= (fun _ -> b)

let prog = new StateBuilder()
```

Recall from the lecture that:

1. `let! x = a` corresponds to `>>=`. It evaluates an expression of type `SM<'a>` and stores the result of type `'a` in `x`.
2. `do! a` corresponds to `>>>=`. It evaluates an expression of type `SM<unit>` and continues with any updates to the state being passed along.
3. `return x` corresponds to `ret`. Given a return value of type `'a` it returns the corresponding value of type `SM<'a>`.
4. `return! a` simply returns `a` that has type `SM<'a>` directly.

For instance, the expression

```
declare "x" >>>= update "x" 5 >>>=
declare "y" >>>= lookup "x" >>= (fun vx -> lookup "y" (fun vy -> (vx, vy)))
```

can be written as

```
prog {
   do! declare "x"
   do! update "x" 5
   do! declare "y"
   let! vx = lookup "x"
   let! vy = lookup "y"

   return (vx, vy)
}
```

and both expressions will evaluate to

```
Success (5, 0)
```

4: (optional) modelling the Scrabble board

This assignment is optional, but it will be used for the Scrabble project and you have done most of it before. With that said, it is perfectly fine to read over it now and return to this later, but you can get started if you want.

Assignment 6.12

We will now remake assignment 3.7 in our new system.

Recall that we had the type `squareFun` where

```
type squareFun = word -> int -> int -> int
```

and the arguments are a word, a position, and an accumulator respectively.

For this assignment we will change it 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 statemnt `stm` returns a function that given a word `w`, a position `pos`, and an accumultor `acc` evaluates `stm` in the inital state where the variable `_pos_` is mapped to `pos`, the variable `_acc_` is mapped to acc, the variable `_result_` is mapped to `0`, the word is set to `w`, and 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 may use the evaluation function from either Section 3 or Section 4 to model this.

Using your function we can create similar square functions as in 3.7.

```
let arithSingleLetterScore = PV (V "_pos_") .+. (V "_acc_")
let arithDoubleLetterScore = ((N 2) .*. PV (V "_pos_")) .+. (V "_acc_")
let arithTripleLetterScore = ((N 3) .*. PV (V "_pos_")) .+. (V "_acc_")

let arithDoubleWordScore = N 2 .*. V "_acc_"
let arithTripleWordScore = N 3 .*. V "_acc_"

let stmntSingleLetterScore = Ass ("_result_", arithSingleLetterScore)
let stmntDoubleLetterScore = Ass ("_result_", arithDoubleLetterScore)
let stmntTripleLetterScore = Ass ("_result_", arithTripleLetterScore)
let stmntDoubleWordScore = Ass ("_result_", arithDoubleWordScore)
let stmntTripleWordScore = Ass ("_result_", arithTripleWordScore)

let singleLetterScore = stmntToSquareFun stmntSingleLetterScore
let doubleLetterScore = stmntToSquareFun stmntDoubleLetterScore
let tripleLetterScore = stmntToSquareFun stmntTripleLetterScore
let doubleWordScore = stmntToSquareFun stmntDoubleWordScore
let tripleWordScore = stmntToSquareFun stmntTripleWordScore

let oddConsonants =
```

```
stmntToSquareFun
   (Seq (Declare "i",
    (Seq (Ass ("_result_", V "_acc_"),
          While (V "i" .<. WL,
                Seq(
                    ITE (IsConsonant (CV (V "i")),
                         Ass ("_result_", V "_result_" .*. N -1),
                         Skip),
                    Ass ("i", V "i" .+. N 1)))))))
```

Examples:

```
> singleLetterScore hello 0 0;;
- val it : int = Success 4

> doubleLetterScore hello 0 0;;
- val it : int = Success 8

> tripleLetterScore hello 0 0;;
- val it : int = Success 12

> singleLetterScore hello 0 42;;
- val it : int = Success 46

> doubleLetterScore hello 0 42;;
- val it : int = Success 50

> tripleLetterScore hello 0 42;;
- val it : int = Success 54

> oddConsonants hello 5 50;;
- val it : int = Success 50

> oddConsonants [('H', 4); ('E', 1); ('L', 1)] 5 50;;
- val it : int = Success -50
```

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 tile to take. In addition, we provide a list of pairs of type `int * squareFun` where the first argument is an identifier and the second is a square on the board. If the result returned by the function is in the list then the corresponding square is returned.

We can, for instance, create the standard Scrabble board in the following way:

```
let abs v result = ITE (v .<. N 0, Ass (result, v .*. N -1), Ass (result, v))

let twsCheck x y = ((V x .=. N 0) .&&. (V y .=. N 7)) .||.
                      ((V x .=. N 7) .&&. ((V y .=. N 7) .||. (V y .=. N 0)))

let dwsCheck x y = (V x .=. V y) .&&. (V x .<. N 7) .&&. (V x .>. N 2)

let tlsCheck x y = ((V x .=. N 6) .&&. (V y .=. N 2)) .||.
                   ((V x .=. N 2) .&&. ((V y .=. N 2) .||. (V y .=. N 6)))

let dlsCheck x y = ((V x .=. N 0) .&&. (V y .=. N 4)) .||.
                   ((V x .=. N 1) .&&. ((V y .=. N 1) .||. (V y .=. N 5)))
.||.
                   ((V x .=. N 4) .&&. ((V y .=. N 0) .||. (V y .=. N 7)))
.||.
                   ((V x .=. N 5) .&&. (V y .=. N 1)) .||.
                   ((V x .=. N 7) .&&. (V y .=. N 4))

let insideCheck x y = ((V x .<. N 8) .&&. (V y .<. N 8))

let checkSquare f v els = ITE (f "xabs" "yabs", Ass ("_result_", N v), els)

let standardBoard =
    Seq (Declare "xabs",
         Seq (Declare "yabs",
              Seq (abs (V "_x_") "xabs",
                   Seq (abs (V "_y_") "yabs",
                        checkSquare twsCheck 4
                           (checkSquare dwsCheck 3
                               (checkSquare tlsCheck 2
                                   (checkSquare dlsCheck 1
                                       (checkSquare insideCheck 0
                                           (Ass ("_result_", N -1))))))))))
```

```
let boardMap = [(0, singleLetterScore); (1, doubleLetterScore); (2, 
tripleLetterScore);
                (3, doubleWordScore); (4, tripleWordScore)] |> Map.ofList
```

Here, the statement `standardBoard` stores a number between `0` and `4` in `_result_` and provides the mapping to the corresponding qsquare types in `boardMap`.

Create a function `stmntToBoardFun : stmnt -> Map<int, squareFun> -> boardFun` that given a statement `stmnt` and a a lookp table of identifiers and square functions `squares` runs `stmnt` in the state where `_x_` and `_y_` are initialised to be the x- and the y-values of the coordinate to the board function, where `_result_` stores the identifier of the square function to use, and where `_x_`, `_y_`, and `_result_` are reserved words. The board function ultimately does a lookup of the integer value `id` stored in `_result_` and returns `Success (Some sf)`, where `sf` is the square function with identifier `id` from `squares` and `Success None` if there is no such identifier in `squares`. If the evaluation of `stmnt` fails then the error is returned. It is important to note that a result of `None` is not considered a failure, it just means that the coordinate is empty.

You can then test the output of your entire board using the following code (available in `Program.fs`)

```
let evalSquare w pos acc =
    function
    | Success (Some f) ->
      match f w pos acc with
       | Success res -> Success (Some res)
       | Failure e   -> Failure e
    | Success None -> Success None
    | Failure e        -> Failure e

let toString =
    function
    | Success (Some x) -> string x
    | Success None     -> "#"
    | Failure err      -> failwith (sprintf "Error: %A" err)

for y in -10..10 do
    for x in -10..10 do
        printf "%s " (stmntToBoardFun standardBoard boardMap (x, y) |>
                      evalSquare hello 1 3 |>
                      toString)
    printfn ""
```

and get the result

```
# # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # # #
```

```
# # # 9 4 4 5 4 4 4 9 4 4 4 5 4 4 9 # # #
# # # 4 6 4 4 4 6 4 4 4 6 4 4 4 6 4 # # #
# # # 4 4 6 4 4 4 5 4 5 4 4 4 6 4 4 # # #
# # # 5 4 4 6 4 4 4 5 4 4 4 6 4 4 5 # # #
# # # 4 4 4 4 6 4 4 4 4 4 6 4 4 4 4 # # #
# # # 4 6 4 4 4 6 4 4 4 6 4 4 4 6 4 # # #
# # # 4 4 5 4 4 4 5 4 5 4 4 4 5 4 4 # # #
# # # 9 4 4 5 4 4 4 4 4 4 4 5 4 4 9 # # #
# # # 4 4 5 4 4 4 5 4 5 4 4 4 5 4 4 # # #
# # # 4 6 4 4 4 6 4 4 4 6 4 4 4 6 4 # # #
# # # 4 4 4 4 6 4 4 4 4 4 6 4 4 4 4 # # #
# # # 5 4 4 6 4 4 4 5 4 4 4 6 4 4 5 # # #
# # # 4 4 6 4 4 4 5 4 5 4 4 4 6 4 4 # # #
# # # 4 6 4 4 4 6 4 4 4 6 4 4 4 6 4 # # #
# # # 9 4 4 5 4 4 4 9 4 4 4 5 4 4 9 # # #
# # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # # #
```

where empty squares are marked with `#`.

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` that given a center coordinate `c`, a statement representing the default square `defaultSq`, a statement `boardStmnt` representing the board as described above and 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 tile the compiled version of `defaultSq` and the squares by compiling `boardStmnt` using `stmntToBoardFun` with `ids` as the square identifier map.

The following code creates the standard board

```
let ids =
    [(0, stmntSingleLetterScore); (1, stmntDoubleLetterScore); (2,
stmntTripleLetterScore);
     (3, stmntDoubleWordScore); (4, stmntTripleWordScore)]

let standardBoard =
    mkBoard (0, 0) stmntSingleLetterScore standardBoardFun ids
```

and the following test code outputs the same result as above.

```
for y in -10..10 do
    for x in -10..10 do
        printf "%s " (standardBoard.squares (x, y) |> evalSquare hello 1 3 |>
toString)
    printfn ""
```