For this assignment you have to use the project supplied on CodeJudge. It contains the following files:

- `Types.fs` A small file containing the types that you will need for the assignment
- `StateMonad.fsi`, `StateMonad.fs` and `Eval.fs`. These are the same files as in Assignment 6. `StateMonad.fsi` is provided but the other two have to be replaced by the corresponding files in your solution to Assignment 6.
- `JParsec.fsi` and `JParsec.fs`. A bare-bones parser-combinator library for `F#` based on Scott Wlaschin's web-site F# For Fun and Profit.
- `FParsecLight.fsi` and `FParsecLight.fs`. A wrapper for the industry-grade parser combinator library FParsec. The interfaces in `JParsec.fsi` and `FParsecLight.fsi` are identical and interchangeable. Any string successfully parsed by one should be successfully parsed by the other but speeds will vary (FParsec is significantly faster) and the error messages are not the same. Use JParsec to understand how Parser Combinators can be implemented and for your CodeJudge submissions. For The Scrabble project you will need to use `FParsecLight`.
- `Parser.fs` This is the file that you will exclusively be developing in this assignment. It contains placeholders for all functions as well as a bare-bones parser for arithmetic expressions.
- `Program.fs` The main executable file. It contains test cases for all exercises in the assignment.

To submit to CodeJudge you need to hand in `Types.fs`, `StateMonad.fsi`, `StateMonad.fs`, `Eval.fs`, `JParsec.fsi`, `JParsec.fs`, and `Parser.fs`.

If you have trouble getting this program to run then that may be due to you having another SDK than the one we've been working with. In the `.fsproj` file you can find `netcoreapp3.1` as the target framework. This can be changed to whatever you have installed. If you are still having problems, either contact a TA or create a fresh project with the files in the order described in the `.fsproj` file. You will also need to import FParsec version 1.1.1 from NuGet.

Before you start with this assignment. Look over both parser combinator `.fsi` files to see what combinators are required for the assignment. They will give you a good overview of what functions are available to you.

You will be writing a parser for the Imp-language. In practice that means converting a string to the following type that we have seen before.

```
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 *)
| IsDigit of cExp      (* Check for digit *)
| IsLetter of cExp     (* Check for leter *)

let (.+.) a b = Add (a, b)
let (.-.) a b = Sub (a, b)
let (.*.) a b = Mul (a, b)
let (./.) a b = Div (a, b)
let (.%.) a b = Mod (a, b)

let (~~) b = Not b
let (.&&.) b1 b2 = Conj (b1, b2)
```

```
let (.||.) b1 b2 = ~~(~~b1 .&&. ~~b2)       (* boolean disjunction *)

let (.=.) a b = AEq (a, b)
let (.<.) a b = ALt (a, b)
let (.<>.) a b = ~~(a .=. b)                (* inequality *)
let (.<=.) a b = a .<. b .||. ~~(a .<>. b)  (* smaller than or equal to *)
let (.>=.) a b = ~~(a .<. b)                (* greater than or equal to *)
let (.>.) a b = ~~(a .=. b) .&&. (a .>=. b) (* greater than *)

type stm =                (* statements *)
| Declare of string       (* 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 *)
```

The grammar for the Imp-language is written as follows

```
n : int
c : char
v : string

A ::= A + A
    | A - A
    | A * A
    | A / A
    | A % A
    | pointValue ( A )
    | charToInt ( C )
    | -A
    | n
    | v
    | ( A )

B ::= true
    | false
    | B /\ B
    | B \/ B
    | ~B
    | A = A
    | A <> A
```

```
    | A < A
    | A <= A
    | A > A
    | A >= A
    | isDigit ( C )
    | isLetter ( C )
    | isVowel ( C )
    | ( B )


C ::= 'c'
    | charValue ( A )
    | intToChar ( A )
    | toUpper ( C )
    | toLower ( C )

S ::= v := A
    | declare v // Mandatory space between keyword and variable
    | S; S
    | if ( B ) then { S } else { S }
    | if ( B ) then { S }
    | while ( B ) do { S }
```

There are a few things to note about this grammar

- The semicolons `;` do ***not*** terminate commands, rather they separate them. This means that, for example, the last command in an if- or a while-statements does not end with a semicolon.
- We have a lot more combinators here (`\/`, `>=`, ...) than you have in the ASTs you have been given so far but all of these can be encoded with the operators you already have. For instance, `P \/ Q` is the same as `~(~P /\ ~Q)` and `5 >= 4` is the same as `~(5 < 4)`. These additional connective encodings are provided for you.
- The number of white spaces are not important and the parser must manage to parse programs no matter how many, if any, whitespaces there are between terms. The only exception to this rule is `declare` that requires at least one white space between the keyword and the identifier.
- This grammar does not enforce an evaluation order, and it is left-recursive which is a problem. We will address both points later.

Come back to this grammar regularly. It is the basis of the entire assignment.

Writing a parser

We will start with easy exercises that gradually build up to a complete parser and also teach some best-practice approaches along the way.

Green Exercises

Exercise 7.1

The function `pstring : string -> Parser<string>` takes a string `s` and will parse any string that starts with `s` and leave the remaining string to be parsed by other parsers.

Create parser functions of type `Parser<string>` for each of the keywords of the language. To be more concrete, let

- `pIntToChar` be a parser for `"intToChar"`
- `pPointValue` be a parser for `"pointValue"`
- `pCharToInt` be a parser for `"charToInt"`
- `pToUpper` be a parser for `"toUpper"`
- `pToLower` be a parser for `"toLower"`
- `pCharValue` be a parser for `"charValue"`
- `pTrue` be a parser for `"true"`
- `pFalse` be a parser for `"false"`
- `pIsDigit` be a parser for `"isDigit"`
- `pIsLetter` be a parser for `"isLetter"`
- `pIsVowel` be a parser for `"isVowel"`
- `pdeclare` be a parser for `"declare"`
- `pif` be a parser for `"if"`
- `pthen` be a parser for `"then"`
- `pelse` be a parser for `"else"
- `pwhile` be a parser for `"while"`
- `pdo` be a parser for `"do"`

Examples:

```
> run pif "if" |> printfn "%A"
Success: "if"

> run pif "if-some random text" |> printfn "%A"
Success: "if"
```

```
> run pif "ithen-some random text" |> printfn "%A"
Failure:
Error parsing if
Line: 0 Column: 1
ithen-some random text
 ^unexpected 't'
```

Note how the column number increases on successful parsers and stays still at failed parse attempts (it succeeds at parsing `i` but expects an `f` in stead of a `t` so it fails at column 1). This information is useful for debugging.

Assignment 7.2

Recall from the lecture that

- The parser `satisfy : (char -> bool) -> Parser<char>` takes a function `f` and returns a parser `p` that succesfully parses any character `c` such that `f c = true`.
- The parser `many : Parser<'a> -> Parser<'a list>` takes a parser `p` and returns a parser that parses 0 or more occurrences of whatever `p` parses and puts the result in a list.
- The parser `many1 : Parser<'a> -> Parser<'a list>` that works exactly like `many` except it requires `p` to be able to be parsed at least once.

Create a parser `pletter : Parser<char>` that parses any letter character (**Hint:** use `satisfy` and `System.Char.IsLetter`)

Create a parser `palphanumeric : Parser<char>` that parses any letter or number character (**Hint:** use `satisfy` and `System.Char.IsLetteOrDigit`)

Create a parser `whitespaceChar : Parser<char>` that parses any whitespace character (**Hint:** use `satisfy` and `System.Char.IsWhitespace`)

Create a parser `spaces : Parser<char list>` that parses zero or more white spaces and puts the result in a list

Create a parser `spaces1 : Parser<char list>` that works exactly like `spaces` except it requires at least one white space to be parsed.

Examples:

```
> run pletter "abcd" |> printfn "%A"
Success: 'a'

> run pletter "1234" |> printfn "%A"
```

```
Failure:
Error parsing satisfy
Line: 0 Column: 0
1234
^unexpected '1'

> run palphanumeric "abcd" |> printfn "%A"
Success: 'a'

> run palphanumeric "1234" |> printfn "%A"
Success: '1'

> run palphanumeric "!@#$" |> printfn "%A"
Failure:
Error parsing satisfy
Line: 0 Column: 0
!@#$
^unexpected '!'

> run whitespaceChar " hello" |> printfn "%A"
Success: ' '

> run whitespaceChar "hello " |> printfn "%A"
Failure:
Error parsing <name depending on your labels>
Line: 0 Column: 0
hello
^unexpected 'h'

> run spaces "            hello" |> printfn "%A"
Success: [' '; ' '; ' '; ' '; ' '; ' '; ' '; ' '; ' '; ' '; ' '; ' ']

> run spaces "hello " |> printfn "%A"
Success: []

> run spaces1 "            hello" |> printfn "%A"
Success: [' '; ' '; ' '; ' '; ' '; ' '; ' '; ' '; ' '; ' '; ' '; ' ']

> run spaces1 "hello " |> printfn "%A"
Failure:
Error parsing <name depending on your labels>
Line: 0 Column: 0
hello
^unexpected 'h'
```

Assignment 7.3

A common thing for a parser to do is to discard all spaces that are encountered. Usually we do not want white spaces to have any semantic significance. In F# and Haskell this is not true as indentation does matter, but for our purposes we will treat white spaces in a similar way as languages like Java or C#. To help with this we will create parser combinators that combine two parsers but discard any white space between them. Judicious use of these connectives will allow us to forget about white spaces entirely as we write the rest of our parser.

Recall from the lecture that we have the connectives

- `(.>>.) : Parser<'a> -> Parser<'b> -> Parser<'a * 'b>` that given two parsers `p1` and `p2` returns a parser that first parses `p1` and subsequently parses `p2` and puts their respective results in a tuple.
- `(.>>) : Parser<'a> -> Parser<'b> -> Parser<'a>` which is similar to `.>>.` but where the result from the trailing parser is discarded.
- `(>>.) : Parser<'a> -> Parser<'b> -> Parser<'b>` which is similar to `.>>.` but where the result from the leading parser is discarded.

Construct the infix parser combinators `(>*>)`, `(.>*>)`, and `(>*>.)` that behave like `(>>)`, `(.>>)`, and `(>>.)` respectively but where any white spaces between (**not** in front or after) the data parsed by `p1` and `p2` are discarded.

Examples:

```
> run (pif .>*>. pthen) "if    then"    |> printfn "%A"
Success: ("if", "then")

> run (pif .>*> pthen) "if    then"     |> printfn "%A"
Success: "if"

> run (pif >*>. pthen) "if    then"     |> printfn "%A"
Success: "then"

> run (pif .>*>. pthen) "ifthen"        |> printfn "%A"
Success: ("if", "then")

> run (pif .>*>. pthen) "if   then    " |> printfn "%A"
Success: ("if", "then")  // The trailing spaces remain to be parsed.

> run (pif .>*>. pthen) "    if   then" |> printfn "%A"
Failure:
Error parsing (<your definition will appear here>)
Line: 0 Column: 0
    if   then
```

```
^unexpected ' '

> run (pif .>*>. pthen .>*>. pelse) "if  then        else" |> printfn "%A"
Success: (("if", "then"), "else")
```

Assignment 7.4

The function `pchar : char -> Parser<char>` takes a character `c` and returns a parser that parses that specific character.

Create a parser `parenthesise : Parser<'a> -> Parser<'a>` that given a parser `p` returns a parser that parses a string of the form `(<arbitrary many spaces><whatever p parses><arbitrary many spaces>)`

Examples:

```
> run (parenthesise pint32)  "(    5    )" |> printfn "%A"
Success: 5

> run (parenthesise pthen) "(    then )"   |> printfn "%A"
Success: "then"

> run (parenthesise pint32)  "(    x    )" |> printfn "%A"
Failure:
Error parsing <parser definition>
Line: 0 Column: 5
(    x    )
     ^unexpected 'x'

> run (parenthesise pint32)  "(  5    x )" |> printfn "%A"
Failure:
Error parsing <parser definition>
Line: 0 Column: 8
(  5    x )
        ^unexpected 'x'
```

Hint: use `>*>.` and `.>*>` to discard the parentheses from the result.
Hint: It may be beneficial to create a similar function that uses `{` and `}` in stead of parentheses. Such a function will come in handy when we parse statements like while-loops or if-statements.

Assignment 7.5

The infix mapping function for parsers `(|>>) : Parser<'a> -> ('a ->' b) -> Parser<'b>` takes a parser `p` and a function `f` and reurns a parser that parses the same input as `p` but which maps the result using `f`. It is an exceptionally useful tool in combination with the parser combinators described in 7.3 (both the original ones and the ones you created). For instance, the following parser parses two integers, with any number of spaces between them, and returns their sum

```
pint32 .>*>. pint32 |>> fun (x, y) -> x + y
```

and the following one sums three integers together.

```
pint32 .>*>. pint32 .>*>. pint32 |>> fun ((x, y), z) -> x + y + z
```

Note how the results from the parsers are collected in tuples. Using the mapping function you can operate on tuples of arbitrary size to handle arbitrary many uses of the `.>*>.` or the `.>>.` combinators.

We will now construct a parser for identifiers. An identifier starts with a character in the alphabet, or an underscore '_', followed by a (possibly empty) sequence of alphanumeric characters (letters and numbers) or underscores.

Using the parsers `pletter`, `palphanumeric`, and `many` from Assignment 7.2, along with the combinators `(|>>)` and `.>>.` (not `.>*>.` as we do not want spaces in our identifiers), and `pchar`, create a parser `pid : Parser<string>` that parses identifiers that start with a single letter or underscore followed by an arbitrary number of letters, numbers, or underscores.

Hint: A useful auxiliary function that you can create converts a list of characters to a string.

Hint: Recall the infix parser combinator `<|> : Parser<'a> -> Parer<'a> -> Parser<'a>` that given two parser combinators `p1` and `p2` parses either `p1` or `p2` (`p2` only if `p1` fails). This is useful to write a parser that parses either a letter or an underscore, for instance.

Examples:

```
> run pid "x" |> printfn "%A"
Success: "x"

> run pid "x1" |> printfn "%A"
Success: "x1"
```

```
> run pid "1x" |> printfn "%A"
Error parsing identifier
Line: 0 Column: 0
1x
^unexpected '1'

> run pid "longVariableName" |> printfn "%A"
Success: "longVariableName"

> run pid "_pos_" |> printfn "%A"
Success: "_pos_"
```

Assignment 7.6

Create a function `unop : Parser<'a> -> Parser<'b> -> Parser<'b>` that given a parser for an operator `op`, and an argument parser `a` returns a parser that parses a string of the form `<whatever op parses><arbitrary many spaces><whatever a parses>` but only keeps the result from `a`.

Even though we discard which operator we use this is still a very useful combinator in combination with `(|>>)` as the third example demonstrates.

Examples:

```
> run (unop (pchar '-') pint32) "-5"               |> printfn "%A"
Success: 5

> run (unop (pchar '-') pint32) "-     5"          |> printfn "%A"
Success: 5

> run (unop (pchar '-') pint32 |>> ( * ) (-1)) "-5" |> printfn "%A"
Success: -5

> run (unop (pchar '-') pint32) "-     x"          |> printfn "%A"
Error parsing <parser definition here>
Line: 0 Column: 6
-     x
      ^unexpected 'x'
```

Assignment 7.7

Create a function `binop : Parser<'a> -> Parser<'b> -> Parser<'c> -> Parser<'b * 'c>` that given a parser for an operator `op`, and argument parsers `a` and `b` returns a

parser that parses a string of the form `<parse a><spaces><parse op><spaces><parse b>` but only keeps the result from `a` and `b` in a pair.

Again, the mapping function `(|>>)` is useful for post processing the results of the parser.

Examples:

```
> run (binop (pchar '+') pint32 pint32) "5 +  7"  |> printfn "%A"
Success: (5, 7)

> run (binop (pchar '+') pint32 pid) "5+var"      |> printfn "%A"
Success: (5, "var")

> run (binop (pchar '+') pint32 pint32 |>>
                 (fun (a, b) -> a + b)) "5 +  7"  |> printfn "%A"
Success: 12
```

Assignment 7.8

We will now create parsers for our arithmetic and character expressions. You can find the complete grammar at the top of the exercise as the non-terminal `A`. As we discussed in the lecture it is important that our grammars are not left-recursive - at any point in time we must be able to tell where we are going by reading the next character. A simple grammar that only includes integer literas, addition, multiplication, and parenthesised expressions can be written as follows

```
A ::= n | A + A | A * A | ( A )
```

and can be rewritten in right-recursive form and guaranteing that multiplication binds harder than addition as

```
Term ::= Prod + Term
       | Prod

Prod ::= Atom * Prod
       | Atom

Atom ::= n
      |  ( Term )
```

Here the leftmost operators descend the grammar hierarchy and even though `Term` does appear in the rules for `Atom` it is guarded by a left parenthesis which is enough to

progress. Have a look at this grammar and make sure you understand why `5 + 3 * 4` results in the same AST as `5 + (3 * 4)` and **not** `(5 + 3) * 4`.

A parser using the provided parser combinator for arithmetic expressions looks as follows:

```
(* Set up forward references *)
let TermParse, tref = createParserForwardedToRef<aExp>()
let ProdParse, pref = createParserForwardedToRef<aExp>()
let AtomParse, aref = createParserForwardedToRef<aExp>()

let AddParse = binop (pchar '+') ProdParse TermParse |>> Add <?> "Add"
do tref := choice [AddParse; ProdParse]

let MulParse = binop (pchar '*') AtomParse ProdParse |>> Mul <?> "Mul"
do pref := choice [MulParse; AtomParse]

let NParse   = pint32 |>> N <?> "Int"
let ParParse = parenthesise TermParse
do aref := choice [NParse; ParParse]

let AExpParse = TermParse
```

Ultimately, this produces a parser `AExpParse : Parser<aExp>` that parses strings into arithmetic expressions.

Expand the parsers above to include all arithmetic expressions, except for casting characters to integers. You do not have to create any more levels in the hierarchy. They bind in the following order (loosest to hardest).

1. Addition and subtraction
2. Multiplication, division, and modulo
3. negation, point value, variables, and integer literals

Finally, to make your life simple, parse negation as multiplication by minus one (`-x = x * -1` or `-x = -1 * x`) similarly to the `unop` examples in Assignment 7.6. There is a test here that tests for this, but it is not in CodeJudge as there are several correct solutions.

Examples:

```
> run AexpParse "4" |> printfn "%A"
Success: N 4

> run AexpParse "x2" |> printfn "%A"
```

```
Success: V "x2"

> run AexpParse "5 + y * 3" |> printfn "%A"
Success: Add (N 5,Mul (V "y",N 3))

> run AexpParse "(5 - y) * -3" |> printfn "%A"
Success: Mul (Sub (N 5,V "y"),Mul (N -1,N 3))

> run AexpParse "pointValue (x % 4) / 0" |> printfn "%A"
Success: Div (PV (Mod (V "x",N 4)),N 0)
```

Assignment 7.9

Create a parser `CExpParse : Parser<cExp>` That parses character expressions. You can find the complete grammar at the top of the exercise as the non-terminal `C`. Here you will only need one hierarchy level but you do have to consider the mutual recursive nature of `aExp` and `cExp`. You will also need to add a case to your parser for arithmetic expressions (Assignment 7.8) that handles character casting.

Examples:

```
> run CexpParse "'x'" |> printfn "%A"
Success: C 'x'

> run CexpParse "toLower (toUpper( 'x'))" |> printfn "%A"
Success: ToLower (ToUpper (C 'x'))

> run CexpParse "toLower (toUpper 'x')" |> printfn "%A"
<fails as there are no parentheses around 'x'>

> run CexpParse "intToChar (charToInt (' '))" |> printfn "%A"
Success: IntToChar (CharToInt (C ' '))

> run AexpParse "charToInt (charValue (pointValue (5)))" |> printfn "%A"
Success: CharToInt (CV (PV (N 5)))
```

Yellow Exercises

Assignment 7.10

Create a parser for boolean expressions called `bExpParse` of type `Parser<bExp>`. You can find the complete grammar at the top of the exercise as the non-terminal `B`. You will need to rewrite the grammar in order for this to function and the operators bind in the following order (loosest to hardest):

1. `/\` and `\/`
2. `=`, `<>`, `<`, `<=`, `>`, and `>=`
3. `~`, `isLetter`, `isVowel`, and `isDigit`

Remember that all of these operators can be encoded by the ones you already have and that they are provided for you.

```
> run BexpParse "true" |> printfn "%A"
Success: TT

> run BexpParse "false" |> printfn "%A"
Success: FF

> run BexpParse "5 > 4 \/ 3 >= 7" |> printfn "%A"
Success: Not
  (Conj
     (Not (Conj (Not (AEq (N 5,N 4)),Not (ALt (N 5,N 4)))),
      Not (Not (ALt (N 3,N 7)))))

> run BexpParse "~false" |> printfn "%A"
Success: Not FF

> run BexpParse "(5 < 4 /\ 6 <= 3) \/ ~false" |> printfn "%A"
Success: Not
  (Conj
     (Not
        (Conj
           (ALt (N 5,N 4),
            Not (Conj (Not (ALt (N 6,N 3)),Not (Not (Not (AEq (N 6,N
3))))))))),
      Not (Not FF)))

> run BexpParse "(5 < 4 \/ 6 <= 3) \/ ~true" |> printfn "%A"
Success: Not
  (Conj
     (Not
        (Not
           (Conj
              (Not (ALt (N 5,N 4)),
               Not
                 (Not
                    (Conj (Not (ALt (N 6,N 3)),Not (Not (Not (AEq (N 6,N
3)))))))))),
      Not (Not TT)))
```

Assignment 7.11

Write a parser `stmntParse : Parser<stm>` that parses valid programs of the Imp language. You can find the complete grammar at the top of the exercise as the non-terminal `S`.

1. Make sure to use the connectives you made in 7.3 to ensure that spaces are properly discarded.
2. Remember that semicolons separate commands, they do not end them (this makes your life much easier).
3. Remember that the `declare` statement requires at least one space between the keyword and the following identifier. Use `spaces1` or `whitespaceChar` from Assignment 7.2 for this.

Hint: The If-Then construct (without the else) can be encoded using standard if-then-else by using `Skip` in the else branch.

Hint If you get parsing errors, trim down the program to find the culprit using either the interactive top loop or by printing output to the console. Under no circumstances should you use CodeJudge to debug at this point.

```
> run stmntParse "x := 5" |> printfn "%A"
Success: Ass ("x", N 5)

> run stmntParse "declare myVar" |> printfn "%A"
Success: Declare "myVar"

> run stmntParse "declaremyVar" |> printfn "%A"
<This should return a parser error. The error message my differ depending on
your setup>

> run stmntParse "declare x; x := 5" |> printfn "%A"
Success: Seq (Declare "x",Ass ("x",N 5))

> run stmntParse "declare x; x := 5  ;y:=7" |> printfn "%A"
Success: Seq (Declare "x",Seq (Ass ("x",N 5),Ass ("y",N 7)))

> run stmntParse "if (x < y) then { x := 5 } else { y := 7 }" |> printfn
"%A"
Success: ITE (ALt (V "x",V "y"),Ass ("x",N 5),Ass ("y",N 7))

> run stmntParse "if (x < y) then { x := 5 }" |> printfn "%A"
Success: ITE (ALt (V "x",V "y"),Ass ("x",N 5),Skip)
```

```
> run stmntParse "while (true) do {x5 := 0} " |> printfn "%A"
Success: While (TT,Ass ("x5",N 0))

> runParserFromFile stmntParse "../../Factorial.txt" |> printfn "%A"
Success: Seq
  (Declare "arg",
   Seq
     (Ass ("arg",N 10),
      Seq
        (Declare "result",
         ITE
           (Not (ALt (V "arg",N 0)),
            Seq
              (Declare "acc",
               Seq
                 (Ass ("acc",N 1),
                  Seq
                    (Ass ("x",N 0),
                     Seq
                       (While
                          (Not (AEq (V "arg",V "x")),
                           Seq
                             (Ass ("x",Add (V "x",N 1)),
                              Ass ("acc",Mul (V "acc",V "x")))),
                        Ass ("result",V "acc")))))),Skip))))
```

The Scrabble Project

Red Exercises

Due to the complexity of the Scrabble program you will need to use `FParsecLight` rather than `JParsec`. This also means that these tests are not available in CodeJudge, but must be run from the command line in the provided template.

In this part of the assignment we finalise the parsing of the square- and board programs used for the Scrabble Project.

Internally, the scrabble board is represented as a collection of strings that you have to parse with the parsers you've made in this assignment and evaluate with the evaluators you made in Assignment 6.

Before we start with the assignments we cover the types you will be working with. We will get more precise in the assignments themselves.

```
type coord      = int * int
type squareProg = Map<int, string>
type boardProg  = {
        prog       : string;
        squares    : Map<int, squareProg>
        usedSquare : int
        center     : coord

        isInfinite : bool   // For pretty-printing purposes only
        ppSquare   : string // For pretty-printing purposes only
    }
```

The `isInfinite` flag and the `ppSquare` function are for pretty-printing purposes by the server only and you do not have to deal with them at all. Nevertheless, they are in the type so you need to be aware that they exist, but that is all.

- The `prog` string is the source code for the Scrabble board.
- The `squareProg` map is a map from priorities (covered in Assignment 2.14) to source code for the square functions.
- The `squares` map is a map from identifiers to square programs (similar to Assignment 6.13)
- The `usedSquare` is an integer that represents the default square that is used when a tile has already been placed on a square (single-letter square on a standard Scrabble board)
- The `center` coordinate represents the coordinate that the first word must be placed over

The square functions for the standard board are the following:

```
let singleLetterScore =
        Map.add 0 "_result_ := pointValue(_pos_) + _acc_" Map.empty
let doubleLetterScore =
        Map.add 0 "_result_ := pointValue(_pos_) * 2 + _acc_" Map.empty
let tripleLetterScore =
        Map.add 0 "_result_ := pointValue(_pos_) * 3 + _acc_" Map.empty

let doubleWordScore = Map.add 1 "_result_ := _acc_ * 2" singleLetterScore
let tripleWordScore = Map.add 1 "_result_ := _acc_ * 3" singleLetterScore
```

Recall from Assignment 6.12 that there are three reserved variable names `_result_`, `_pos_` ,and `_acc_` for the evaluation. You see them in the square functions.

The source code for a standard Scrabble board looks as follows:

```
let standardBoardSource =
  "declare xabs;
   declare yabs;
   if (_x_ < 0) then { xabs := _x_ * -1 } else { xabs := _x_ };
   if (_y_ < 0) then { yabs := _y_ * -1 } else { yabs := _y_ };
   if ((xabs = 0 /\ (yabs = 7)) \/
       (xabs = 7 /\ (yabs = 0 \/ yabs = 7))) then { _result_ := 4 }
   else { if (xabs = yabs /\ xabs < 7 /\ xabs > 2) then { _result_ := 3 }
   else { if ((xabs = 2 /\ (yabs = 2 \/ yabs = 6)) \/
                  (xabs = 6 /\ (yabs = 2))) then { _result_ := 2  }
   else { if ((xabs = 0 /\ (yabs = 4)) \/
              (xabs = 1 /\ (yabs = 1 \/ yabs = 5)) \/
              (xabs = 4 /\ (yabs = 0 \/ yabs = 7)) \/
              (xabs = 5 /\ (yabs = 1)) \/
              (xabs = 7 /\ (yabs = 4))) then { _result_ := 1 }
   else { if (xabs <= 7 /\ yabs <= 7) then { _result_ := 0 }
   else { _result_ := -1 } } } } }"
```

Recall from Assignment 6.13 that there are three reserved variable names `_result_`, for the result, and `_x_` and `_y_` for the coordinate. Depending on the coordinate the function stores `0` in `_result_` for single letter score, `1` for double letter score, `2` for tripple letter score, `3` for double word score, `4` for triple word score, and `-1` if the coordinate is outside the board.

The squares and the boards themselves (what you are going to parse) is then presented as follows.

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

let standardBoardProg : boardProg = {
  prog = standardBoardSource;
  squares = squares;
  usedSquare = 0;
  center = (0, 0);
  isInfinite = false;
  ppSquare = "" // There will be a pretty-printing function here later
                // but you never have to reason about it
}
```

Exactly how these types, squares, and boards are used will be made clear in the inividual assignments.

In `JParsec.fsi` you will find the function `getSuccess : ParserResult<'a> -> 'a` that given a parser result `Success s` returns `s` and fails otherwise. You will find this useful.

The remaining assignments should be short. If you find yourself complicating them contact a TA.

Assignment 7.12 (Parsing square programs)

In Assignment 3.10 the type of a square was

```
type word   = (char * int) list
type square = (int * word -> int -> int -> int) list
```

Where the integer component is a priority explaining when a function should be run (explained in Assignment 2.14). The priority is only important if you want your scrabble bot to calculate points, but from now on it needs to be here for completeness even if you are not intending to use it.

We will compile programs of type `squareProg` to values of type `square`, which use the `squareFun` type from 6.12.

```
type squareProg = Map<int, string>
type squareFun = word -> int -> int -> Result<int, Error>
type square = Map<int, squareFun>
```

The type `square` is nearly identical to what we had in Assignment 3.10 (where we had not yet covered maps) but using a map in stead of a list removes the possibility of there being duplicate priority keys for the different functions in the square.

Create a function `parseSquareProg : squareProg -> square` that given a square program `sqp` runs the `stmntParse` parser on all source code inside the `sqp` map (the string values) and evaluates them using your new `stmntToSquarerFun` function from Assignment 6.12. The priorities should remain unchanged.

Hint: Use `Map.map`, `getSuccess` from `JParsec`, and use `run` to run `stmntParse` to run the parser on the source code.

Examples:

```
> (parseSquareProg tripleWordScore |> Map.find 0) hello 0 0
- val it = Success 4 : int
```

```
> (parseSquareProg tripleWordScore |> Map.find 0) hello 4 0
- val it = Success 1 : int

> (parseSquareProg tripleWordScore |> Map.find 0) hello 0 42
- val it = Success 46 : int

> (parseSquareProg tripleWordScore |> Map.find 1) hello 0 0
- val it = Success 0 : int

> (parseSquareProg tripleWordScore |> Map.find 1) hello 4 0
- val it = Success 0 : int

> (parseSquareProg tripleWordScore |> Map.find 1) hello 0 42
- val it = Sucess 126 : int
```

Assignment 7.13 (Parsing board functions)

From 6.13 we had the following types and functions:

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

Create a function `parseBoardFun : string -> Map<int, square> -> boardFun` that given the source code for a board `s`, and a lookup table for squares `sqs` (not the source code for the squares, this function assumes they have already been compiled), and returns a `boardFun` constructed using the result of parsing `s` and the squares `sqs`.

Hint: Use `getSuccess` from `JParsec`, and use `run` to run `stmntParse` to run the parser on the source code. Use `stmntToBoardFun` to create your result.

Examples:

```
let bf = parseBoardProg standardBoardSource (Map.map (fun _ ->
parseSquareFun) squares)
let lookup c i acc =
    bf c |>
    fun (StateMonad.Success sq) -> sq |>
    Option.map (fun x -> Map.find i x Eval.hello 0 acc)

> lookup (0, 0) 0 10
- val it =  Some (Sucess 14) : StateMonad.Result<int option, Error>

> lookup (0, 4) 0 10
- val it = Some (Success 18) : StateMonad.Result<int option, Error>

> lookup bf (2, 2) 0 10
```

```
- val it = Some (Success 22) : StateMonad.Result<int option, Error>

> lookup bf (3, 3) 1 10
- val it = Some (Success 20) : StateMonad.Result<int option, Error>

> lookup bf (0, 7) 1 10
- val it = Some (Success 30) : StateMonad.Result<int option, Error>

> lookup (0, 42) 0 10
- val it = None : StateMonad.Result<int option, Error>
```

Assignment 7.14 (Parsing boards)

Use your `mkBoard` function from 6.14 as a reference.

We have the following board type

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

Create a `function mkBoard : boardProg -> board` that

- given a a board program `bp` with
  - a center coordinate `bp.center`
  - a lookup table `m` for source code for squares `bp.squares`
  - the index `x` in `m` of the default square `bp.usedSquare`
  - The source code for the board `bp.prog`
- runs `parseSquareFun` on all elements of `m` (of type `Map<int, squareProg>`) resulting in a map `m'` of type `square` (use `Map.map`) and returns a board with
  - `center` set to `bp.center`
  - `defaultSquare` set to the the square function with key `x` in `m'`.
  - `squares` set to `parseBoardFun bp.prog m'`

To see if everything works run the following code (which is available in `Program.fs`)

```
    let evalSquare w pos acc (m : square) =
        let acc' = Map.find 0 m w pos acc |> fun (StateMonad.Success x) -> x
        m |>
        Map.tryFind 1 |>
        Option.map (fun f -> f w pos acc' |> fun (StateMonad.Success x) ->
```

```
x) |>
        Option.defaultValue acc'

    let board = mkBoard standardBoardProg

    for y in -10..10 do
        for x in -10..10 do
            printf "%s " (board.squares (x, y) |>
                          (fun (StateMonad.Success sq) -> sq |>
                           Option.map (fun x -> evalSquare Eval.hello 1 2 x
|> string)) |>
                          Option.defaultValue "#")
        printfn ""
```

If everything works correctly, you should see the following output

```
# # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # # # # #
# # # 9 3 3 4 3 3 3 9 3 3 3 4 3 3 9 # # #
# # # 3 6 3 3 3 5 3 3 3 5 3 3 3 6 3 # # #
# # # 3 3 6 3 3 3 4 3 4 3 3 3 6 3 3 # # #
# # # 4 3 3 6 3 3 3 4 3 3 3 6 3 3 4 # # #
# # # 3 3 3 3 6 3 3 3 3 3 6 3 3 3 3 # # #
# # # 3 5 3 3 3 5 3 3 3 5 3 3 3 5 3 # # #
# # # 3 3 4 3 3 3 4 3 4 3 3 3 4 3 3 # # #
# # # 9 3 3 4 3 3 3 3 3 3 3 4 3 3 9 # # #
# # # 3 3 4 3 3 3 4 3 4 3 3 3 4 3 3 # # #
# # # 3 5 3 3 3 5 3 3 3 5 3 3 3 5 3 # # #
# # # 3 3 3 3 6 3 3 3 3 3 6 3 3 3 3 # # #
# # # 4 3 3 6 3 3 3 4 3 3 3 6 3 3 4 # # #
# # # 3 3 6 3 3 3 4 3 4 3 3 3 6 3 3 # # #
# # # 3 6 3 3 3 5 3 3 3 5 3 3 3 6 3 # # #
# # # 9 3 3 4 3 3 3 9 3 3 3 4 3 3 9 # # #
# # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # # # # #
```