Assignment 7

For this assignment 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 *)
| IsConsonant of cExp      (* Check for constant *)

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

A grammar for the Imp-language can be 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
     | ( B )

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

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 whitespaces there are between terms, and some times no whitespaces, separate the 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.

Parsing literals

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

Exercise 7.1

The function `pstring : string -> Parser<TextInputState, string>` takes a string `s` and will parse any input that exactly matches s.

Create parser functions of type `Parser<TextInputState, 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"`
- `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:

```
> runTextParser pif "if" |> printResult
Success "if"

> runTextParsen pif "if-some random text" |> printResult
Success "if"

> runTextParsen pif "ithen-some random text" |> printResult
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. This information is useful for debugging.

Assignment 7.2

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`. 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,'b> -> Parser<'a,'c> -> Parser<'a,'b * 'c>` that given two parsers `p1` and `p2` returns a parser that returns a parser that parses into the pair of the results of first parsing using `p1` and then using `p2`.
- `(.>>) : Parser<'a,'b> -> Parser<'a,'c> -> Parser<'a,'b>>` which is similar to `.>>.` but where the result from the trailing parser is discarded.
- `(>>.) : Parser<'a,'b> -> Parser<'a,'c> -> Parser<'a,'c>` which is similar to `.>>.`

but where the result from the leading parser is discarded.

The parser `spaces : Parser<TextInputState,char list>` parses an arbitrary amount of whitespaces in a list - this list is nearly always discarded in practice as the number of whitespaces does not interest us.

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:

```
> runTextParser (pif .>*>. pthen) "if    then" |> printResult
Success: ("if", "then")

> runTextParser (pif .>*>. pthen) "ifthen" |> printResult
Success: ("if", "then")

> runTextParser (pif .>*>. pthen) "if   then    " |> printResult
Success: ("if", "then")  // The trailing spaces remain to be parsed.

> runTextParser (pif .>*>. pthen) "    if   then" |> printResult
Error parsing (<your definition will appear here>)
Line: 0 Column: 0
    if   then
^unexpected ' '

> runTextParser (pif .>*>. pthen .>*>. pelse) "if  then        else" |>
printResult
Success: (("if", "then"), "else")
```

Assignment 7.3

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

Create a parser `parenthesise : Parser<TextInputState, 'a> -> Parser<TextInputState, '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:

```
> runTextParser (parenthesise pint)  "(    5    )" |> printResult
Success: 5

> runTextParser (parenthesise pthen) "(    then )" |> printResult
Success: "then"

> runTextParser (parenthesise pint)  "(    x    )" |> printResult
Error parsing <parser definition>
```

```
Line: 0 Column: 5
(    x    )
     ^unexpected 'x'

> runTextParser (parenthesise pint)  "(  5    x )" |> printResult
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.

Assignment 7.4

The infix mapping function for parsers `(|>>)  : Parser<'a, 'b> -> ('b ->' c) -> Parser<'a, 'c>` 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 7.2 (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

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

and the following one sums three integers together.

```
pint .>*>. pint .>*>. pint |>> 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 followed by a (possibly empty) sequence of alphanumeric characters (letters and numbers).

The parser `letterChar : Parser<TextInputState,char>` parses a single letter; the parser `alphaNumeric :  : Parser<TextInputState,char>` parses a single letter or number; the parser `many : Parser<'a,'b> -> Parser<'a,'b list>` takes a single parser `p` and tries to parse using `p` for as long as it can and puts the results in a (possibly empty) list in the order they were parsed.

Using these three parsers along with the combinators `(|>>)` and `.>>.` (not `.>*>.` as we do not want spaces in our identifiers), create a parser `pid : Parser<TextInputState,string>` that parses identifiers that start with a single letter followed by an arbitrary number of letters or numbers.

Examples:

```
> runTextParser pid "x" |> printResult
Success: "x"

> runTextParser pid "x1" |> printResult
Success: "x1"

> runTextParser pid "1x" |> printResult
Error parsing identifier
Line: 0 Column: 0
1x
^unexpected '1'

> runTextParser pid "longVariableName" |> printResult
Success: "longVariableName"
```

Assignment 7.5

Create a function `unop : Parser<'a, 'b> -> Parser<'a, 'c> -> Parser<'a, 'c>` 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:

```
> runTextParser (unop (pchar '-') pint) "-5"                |> printResult
Success: 5

> runTextParser (unop (pchar '-') pint) "-     5"           |> printResult
Success: 5

> runTextParser (unop (pchar '-') pint |>> ( * ) (-1)) "-5" |> printResult
Success: -5

> runTextParser (unop (pchar '-') pint) "-     x"           |> printResult
Error parsing <parser definition here>
Line: 0 Column: 6
-     x
      ^unexpected 'x'
```

Assignment 7.6

Create a function `binop : Parser<'a, 'b> -> Parser<'a, 'c> -> Parser<'a, 'd> -> Parser<'a, 'c * 'd>` 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:

```
> runTextParser (binop (pchar '+') pint pint) "5 +  7" |> printResult
Success: (5, 7)

> runTextParser (binop (pchar '+') pint pid) "5+var"   |> printResult
Success: (5, "var")

> runTextParser (binop (pchar '+') pint pint |>>
                 (fun (a, b) -> a + b)) "5 +  7"       |> printResult
Success: 12
```

Assignment 7.7

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<TextInputState, aExp>()
let ProdParse, pref = createParserForwardedToRef<TextInputState, aExp>()
let AtomParse, aref = createParserForwardedToRef<TextInputState, 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   = pint |>> N <?> "Int"
let ParParse = parenthesise TermParse
do aref := choice [NParse; ParParse]

let AExpParse = TermParse
```

Ultimately, this produces a parser `AExpParse : Parser<TextInputState, aExp>` taht 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

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

Examples:

```
> runTextParser AexpParse "4"  |> printResult
Success: N 4

> runTextParser AexpParse "x2"  |> printResult
Success: V "x2"

> runTextParser AexpParse "5 + y * 3"  |> printResult
Success: Add (N 5,Mul (V "y",N 3))

> runTextParser AexpParse "(5 - y) * -3"  |> printResult
Success: Mul (Sub (N 5,V "y"),Mul (N -1,N 3))

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

Assignment 7.8

Create a parser `CExpParse : Parser<TextInputState, 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 that handle character casting.

Examples:

```
> runTextParser CexpParse "'x'" |> printResult
Success: C 'x'

> runTextParser CexpParse "toLower (toUpper( 'x'))" |> printResult
Success: ToLower (ToUpper (C 'x'))

> runTextParser CexpParse "intToChar (charToInt (' '))" |> printResult
Success: IntToChar (CharToInt (C ' '))

> runTextParser AexpParse "charToInt (charValue (pointValue (5)))" |>
printResult
Success: CharToInt (CV (PV (N 5)))
```

Exercise 7.9

Create a parser for boolean expressions called `bExpParse` of type `Parser<TextInputState, 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. `~`

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

```
> runTextParser BexpParse "true" |> printResult
Success: TT

> runTextParser BexpParse "false" |> printResult
Success: FF

> runTextParser BexpParse "5 > 4 \/ 3 >= 7" |> printResult
Success: Not
  (Conj
     (Not (Conj (Not (AEq (N 5,N 4)),Not (ALt (N 5,N 4)))),
      Not (Not (ALt (N 3,N 7)))))
```

```
> runTextParser BexpParse "~false" |> printResult
Success: Not FF

> runTextParser BexpParse "(5 < 4 /\ 6 <= 3) \/ ~false" |> printResult
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)))

> runTextParser BexpParse "(5 < 4 \/ 6 <= 3) \/ ~true" |> printResult
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)))
```

Exercise 7.10

Write a parser `stmParse : Parser<TextInputState, 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 7.2 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. There is a good parser `whitespaceChar : Parser<TextInputState, char>` that parses a single white space that is useful 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.

```
> runTextParser stmParse "x := 5" |> printResult
Success: Ass ("x",N 5)

> runTextParser stmParse "declare myVar" |> printResult
Success: Declare "myVar"

> runTextParser stmParse "declaremyVar" |> printResult
<This should return a parser error. The error message my differ depending on
your setup>

> runTextParser stmParse "declare x; x := 5" |> printResult
Success: Seq (Declare "x",Ass ("x",N 5))

> runTextParser stmParse "declare x; x := 5  ;y:=7" |> printResult
Success: Seq (Declare "x",Seq (Ass ("x",N 5),Ass ("y",N 7)))

> runTextParser stmParse "if (x < y) then { x := 5 } else { y := 7 }" |>
printResult
Success: ITE (ALt (V "x",V "y"),Ass ("x",N 5),Ass ("y",N 7))

> runTextParser stmParse "if (x < y) then { x := 5 }" |> printResult
Success: ITE (ALt (V "x",V "y"),Ass ("x",N 5),Skip)

> runTextParser stmParse "while (true) do {x5 := 0} " |> printResult
Success: While (TT,Ass ("x5",N 0))

> runParserFromFile stmParse "../../Factorial.txt" |> printResult
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))))
```