Major Assignment 3

Date of Submission - **3rd April, Monday**

This assignment will be explained during the lab hours on Saturday. In the assignment, you have to write a parser for a small language whose grammar is given below.

$$
\begin{array}{lcl}
number & \rightarrow & \{digit\}^{+} \\
identifier & \rightarrow & letter\{letter \mid digit\}^{*} \\
variable & \rightarrow & identifier \mid identifier[expression] \\
term & \rightarrow & number \mid variable \mid (expression) \\
expression & \rightarrow & term\{+\ expression\}^{*} \\
assignment & \rightarrow & variable = expression
\end{array}
$$

Examine the grammar carefully and see how it generates the following statements.

```
a = 3
a = 3 + i
a[a[1]] = a[3+i]
a[a[1]+j] = a[3+i]
a[a[1]] = a[3+i] + b[2] + xyz
```

The parser should take a string and, if the parse succeeds, return a pair consisting of a "parsed-stuff" and the rest of the string. It should return the symbol `'fail` if the parse does not succeed. The nature of the parsed stuff will be clear from the rest examples. Include the following `define-structs` to construct trees.

```
#lang racket

(define-struct gnode (sym list) #:transparent)
(define-struct ident (str) #:transparent)
(define-struct num (val) #:transparent)
```

First define a function called a predicate parser. This returns the next character in the input string, if it happens to satisfy a given predicate `p`.

```
(define (pred-p p)  (lambda (str) ...))
```

```
> ((pred-p (lambda (c) (char=? c #\=))) "=abc")
(#\= . "abc")
> ((pred-p (lambda (c) (char=? c #\=))) "+abc")
fail
```

Now, define a `single-digit` parser and a single-alphabet using `pred-p`.

```
(define single-digit-p (lambda (str) ...))
(define single-alphabet-p (lambda (str) ...))
```

```
> (single-alphabet-p "a1234")
(#\a . "1234")
> (single-digit-p "a1234")
fail
> (single-digit-p "1234")
(#\1 . "234")
```

Next write a sequential parser combiner seq. This takes two parsers `p1` and `p2` and returns a parser which "does" a `p1` followed by `p2`. The results of the parsers are combined using a function `f`.

```
(define (seq p1 p2 f) (lambda (str)...))
```

Define a alternate parser combiner alt. This takes two parsers p1 and p2 and returns a parser that first tries p1, and if that does not succeed, tries p2.

```
(define (alt p1 p2) (lambda (str)...))
```

```
> ((seq single-digit-p single-alphabet-p combine-cc) "1a234")
("1a" . "234")
> ((seq single-digit-p single-alphabet-p combine-cc) "a1234")
fail
> ((alt single-digit-p single-alphabet-p) "a1234")
(#\a . "1234")
> ((alt single-digit-p single-alphabet-p) "1a234")
(#\1 . "a234")
```

The definitions of the combining functions `combine-cc` and others are given at the end.

Define a epsilon parser.

```
(define epsilon-p (lambda (str) ...)

> (epsilon-p "1234")
("" . "1234")
```

Using these write a `zero-or-more` parser. This takes a parser `p` and applies it on the input string zero or more times. The reuslts of these parses are combines using a given function `f`.

```
(define (zero-or-more p f) (lambda (str)...))

> ((zero-or-more single-digit-p combine-cs) "1234a")
("1234" . "a")
```

Notice that the zero-or-more applies `p` as many times as possible. Similarly write a `one-or-more` parser.

Define a whitespace parser that eats up whitespace at the beginning.

```
> (whitespace-p "    a12345")
("" . "a12345")
```

Now write a number parser.

```
(define number-p  (lambda (str) ...))
```

and a identifier parser.

```
(define identifier-p (lambda (str) ...))

(#(struct:num 1234) . "")
> (identifier-p "a12345")
(#(struct:ident "a12345") . "")
> (identifier-p "    a12345")
(#(struct:ident "a12345") . "")
```

Notice that the number and identifier parsers have started returning trees as the parsed stuff. Also they remove whitespaces in the beginning.

Now you have to simultaneously build a variable parser a term parser and an expression parser.

```
(define variable-p (lambda (str) ...))
(define term-p  (lambda (str) ...))
(define expression-p  (lambda (str) ...))

> (variable-p "a12345")
(#(struct:ident "a12345") . "")

> (variable-p "x[a12345+1]")
(#(struct:gnode ARRAY
  (#(struct:ident "x")
   #(struct:gnode PLUS
    (#(struct:ident "a12345")
     #(struct:num 1)))))
 . "")

> (term-p "(a+(b+c))")
(#(struct:gnode PLUS
  (#(struct:ident "a")
   #(struct:gnode PLUS
    (#(struct:ident "b")
     #(struct:ident "c")))))
 . "")

> (expression-p "(a+(b+c))+d[e]")
(#(struct:gnode PLUS
  (#(struct:gnode PLUS
    (#(struct:ident "a")
     #(struct:gnode PLUS
      (#(struct:ident "b")
       #(struct:ident "c")))))
    #(struct:gnode ARRAY
     (#(struct:ident "d")
      #(struct:ident "e")))))
 . "")
```

Finally write the assignment parser:

```
(define assignment-p  (lambda (str) ...))
```

Now here are the results of parsing the examples that we had given at the beginning of the write-up

```
> (assignment-p "a = 3")
```

```
(#(struct:gnode ASSIGN (#(struct:ident "a")
                        #(struct:num 3))) . "")

> (assignment-p "a = 3 + i")
(#(struct:gnode
  ASSIGN
  (#(struct:ident "a")
   #(struct:gnode PLUS (#(struct:num 3)
                        #(struct:ident "i")))))
 ."")

> (assignment-p "a[a[l]] = a[3+i]")
(#(struct:gnode ASSIGN
  (#(struct:gnode ARRAY
     (#(struct:ident "a")
      #(struct:gnode ARRAY (#(struct:ident "a")
                            #(struct:ident "l")))))
   #(struct:gnode ARRAY
      (#(struct:ident "a")
       #(struct:gnode PLUS (#(struct:num 3)
                            #(struct:ident "i")))))))
. "")

> (assignment-p "a[a[l]+j] = a[3+i]")
(#(struct:gnode ASSIGN
  (#(struct:gnode ARRAY
      (#(struct:ident "a")
       #(struct:gnode PLUS
         (#(struct:gnode ARRAY
            (#(struct:ident "a")
             #(struct:ident "l")))
          #(struct:ident "j")))))
   #(struct:gnode ARRAY
      (#(struct:ident "a")
       #(struct:gnode PLUS
         (#(struct:num 3)
          #(struct:ident "i")))))))
. "")


> (assignment-p "a[a[l]] = a[3+i] + b[2] + xyz")
(#(struct:gnode ASSIGN
  (#(struct:gnode ARRAY
```

```
    (#(struct:ident "a")
     #(struct:gnode ARRAY
       (#(struct:ident "a")
        #(struct:ident "l")))))
   #(struct:gnode PLUS
     (#(struct:gnode ARRAY
        (#(struct:ident "a")
         #(struct:gnode PLUS
           (#(struct:num 3)
            #(struct:ident "i")))))
      #(struct:gnode PLUS
         (#(struct:gnode ARRAY
            (#(struct:ident "b")
             #(struct:num 2)))
          #(struct:ident "xyz")))))))
 ."")
```

Finally, here are the definitions of `combine-cc`, `combine-sc`, `combine-cs`, and `combine-cc`.

```
(define (combine-cc char1 char2)
  (list->string (list char1 char2)))

(define (combine-sc str char)
  (list->string (append (string->list str)
                        (list char))))

(define (combine-cs char str)
  (list->string (cons char (string->list str))))

(define (combine-ss str1 str2)
  (list->string (append (string->list str1)
                        (string->list str2))))
```