CSCI 301, Lab # 8

Fall 2022

Goal: This is the last in a series of labs that will build an interpreter for Scheme. In this lab we will build our own parser for Scheme expressions using a simple LL(1) grammar to guide us.

Unit tests: At a minimum, your program must pass the unit tests found in the file `lab8-test.rkt`. Place this file in the same folder as `parse.rkt`, and also `eval.rkt`, and run it; there should be no output. Include your unit tests and both labs in your submission.

From strings to expressions. Up until now we let **Racket** handle the conversion from characters to lists, that is, changing the characters you typed into a file, for example, `'(+ 1 2)`, into an actual Scheme list structure with `cars` and `cdrs`. **Racket** has a builtin expression reader which does this automatically.

Now we are going to do that part ourselves. Our unit test file, for example, can look like this:

```
(require rackunit "parse.rkt" "eval.rkt")
(define e2 (list (list '+ +)))
(check equal? (parse "(+ 1 2)")
              '((+ 1 2)))
(check equal? (evaluate (first (parse "(+ 1 2)")) e2)
              3)
```

Note that your interpreter will be the one defined in `eval.rkt`, and will provide `evaluate`, and the parser will be defined in `parse.rkt` and provide `parse`. The parser and the evaluator should be completely independent.

Input We will use the same strategy for input that we used in the RPN calculator. Thus, a string will be converted to a list of characters. That list will be passed between functions representing the variables of the grammar. The functions will consume the input list and produce a list that can be passed to `evaluate`.

LL(1) parsing for Scheme The grammar for our Scheme is incredibly simple:

$L \to \text{␣}L \mid EL \mid \epsilon$	Expression List
$E \to DN \mid AS \mid (\ L\ )$	Expression
$S \to AS \mid \epsilon$	Symbol
$N \to DN \mid \epsilon$	Number
$D \to 0 \mid 1 \mid 2 \mid 3 \mid ...$	Digit
$A \to a \mid b \mid c \mid d \mid ...$	Symbolic

There are a couple of things to note with the grammar shown above. First, the visible-whitespace character is "␣". Second it is not, strictly speaking LL(1); we will fix this in the first and follow sets. Finally, as in our RPN calculator, we used some builtin Scheme procedures to classify characters. For example, `char-numeric?` will identify digits, `char-whitespace?` will identify whitespace.

For identifying characters we can use in a symbol (called "Symbolic" in the grammar above), we want to include just about everything except whitespace and the two parentheses. This is easy to define:

```
(define char-symbolic?
  (lambda (char) (and (not (char-whitespace? char))
                      (not (eq? char #\())
                      (not (eq? char #\))))))
```

We are interested in five kinds of characters: `#\(`, `#\)`, `char-numeric?`, `char-symbolic?`, and `char-whitespace?`

	Null	First	Follow
L	yes	{(,␣,0..9,a..}	{)}
E	no	{(,0..9,a..}	{(,),␣,0..9,a..,$}
S	yes	{a..}	{(,),␣,0..9,**a..**,$}
N	yes	{0..9}	{(,),␣,**0..9**,a..,$}
D	no	{0..9}	{(,),␣,0..9,a..,$}
A	no	{a..}	{(,),␣,0..9,a..,$}

Notice how S and N are nullable and that the intersection of their First and Follow sets are not empty? This means that the grammar is not LL(1).

We are faced with two options: make the grammar more complicated or directly modify the First and Follow sets. Let's modify the sets.

By removing the common symbols from the follow set, the resulting parser will generate the longest symbol or number. That is, a digit cannot directly follow an N. Similarly, a symbol character cannot follow an S. The modified sets look like this:

	Null	First	Follow
L	yes	{(,␣,0..9,a..}	{)$}
E	no	{(,0..9,a..}	{(,),␣,0..9,a..,$}
S	yes	{a..}	{(,),␣,0..9,$}
N	yes	{0..9}	{(,),␣,a..,$}
D	no	{0..9}	{(,),␣,0..9,a..,$}
A	no	{a..}	{(,),␣,0..9,a..,$}

Write a parser following the same pattern as we did for the RPN calculator. There are only a couple of significant differences between the calculator and this parser.

The $E \rightarrow (L)$ rule creates a nested list. This is similar to the way that `rpn-parser.rkt` handled numbers. This was wrong for numbers, but is right for lists.

The symbols in the RPN parser were a handful of single characters. Symbols in scheme must be at least a single character, but can be longer. They cannot have embedded numbers nor can they have embedded parentheses. You might have to do something similar to how we handled integers in the calculator parser.

Read-Eval-Print Loop: optional The read-eval-print loop is the main body of an interactive interpreter. Note that we are missing special forms that modify the environment (e.g., `define`).

```
(define repl (lambda ()
    (map (lambda (exp) (display (evaluate exp env))) (parse (read-line)))
    (newline)
    (repl)
    )
  )
```

Program files: optional Note that our parser only handles strings typed in, but that we usually want an interpreter to interpret program files, not strings. The solution is pretty trivial: check out the **Racket** procedure `file->string`.

With a little work you could get your interpreter to interpret itself...