CSCI 301, Lab # 8

Fall 2024

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.

Submission: Your parser will be implemented in `parse.rkt`; it will produce expressions that are ready to be evaluated by your existing interpreter in `eval.rkt`. Skeleton code for `parse.rkt` is provided. Make sure you include your name in the block comment at the top of the file.**Include both `eval.rkt` and parse.rkt in your submission.**

Unit tests: 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 `eval.rkt`, and run it; there should be no output.

From strings to expressions. Up until now we have passed quoted Racket expressions into our interpreter. For example, our interpreter can take something like `'(+ 1 2)` and evaluate it to `3`. What we haven't handled yet is taking program text (i.e., expressions) as a string and turning it into this quoted list structure. When you enter code in the definitions or interactions window, **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: we will begin by converting a string into 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 As programming language grammars go, our Scheme grammar is extremely simple:

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

As in the RPN grammar, any whitespace character is represented by an underscore (_). We will use some builtin Scheme procedures to classify characters. We need to be able to identify five kinds of characters:

- `#\(` and `#\)`, which we can check by direct comparison using `equal?`,
- digits, which we can check using the built-in `char-numeric?`,
- whitespace characters, which we can check using the built-in `char-whitespace?`, and
- characters we can use in a symbol (called "Symbolic" in the grammar). For this, we want to include just about everything except whitespace, digits, and the two parentheses. We can define this as follows:

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

Patching the Grammar The grammar above is not actually LL(1). Here are the nullable, first and follow sets:

	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). To fix this, we have 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, and 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..,$}
A	no	{a..}	{(,),_,0..9,a..,$}

Your Task Write a recursive descent parser following the same pattern as we did for the RPN calculator. The parse table and function headers are provided to you with the skeleton code.

There are a couple of significant differences between the RPN calculator and this parser that are worth discussing.

The $E \to (L)$ rule creates a nested list. The parser should ensure that the character *following* the parsed list L is indeed a `#\)`.

Handling digits will work much like it does in the `rpn-ast.rkt` example. This means that the `D` function returns a digit as a number (you'll find the provided `char->number` helpful), but the `N` function takes an extra parameter (called `inherit` in the skeleton) with the numerical value of the digits parsed so far. When handling the $N \to DN$ production, the inherit value passed to the recursive call to `N` is built from the existing inherited value and the value of the digit just parsed by `D`. When eventually $N \to \epsilon$, the inherited value can simply be returned.

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. The recommended approach is similar how numbers are handled, except in this case, the inherited value is a *list* of characters that will be accumualted until $S \rightarrow \epsilon$, at which point the list of characters can be converted into a string (using `{list->string}` and the string to a symbol (using `string->symbol`).

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...