CSCI 301, Lab # 8

Fall 2018

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.

Due: Your program, `lab08.rkt`, must be submitted to Canvas before midnight, Monday, December 3. **Submit both `lab07.rkt` and `lab08.rkt` with this assignment!**

Unit tests: At a minimum, your program must pass the unit tests found in the file `lab08-test.rkt`. Place this file in the same folder as `lab08.rkt`, and also `lab07.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 `car`s and `cdr`s. **Racket** has a builtin expression reader which does this automatically.

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

```
(require rackunit "lab08.rkt" "lab07.rkt")
(define e2 (list (list '+ +)))
(check equal?
       (parse "(+ 1 2)")
       '(+ 1 2))
(check equal?
       (evaluate (parse "(+ 1 2)") e2)
       3)
```

Note that your interpreter will be the one defined in `lab07.rkt`, and will provide `evaluate`, and the parser will be defined in `lab08.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 `rpncalculator.rkt`. Thus, a string will be converted to a list of characters, which will be stored in the global variable `*the-input*`, and will be accessed *only* by the procedure that takes a string as a parameter, `set-input!`, and the three procedures (which take no parameters) `end-of-input?`, `current-char`, and `skip-char`. This will ensure we're doing LL(1) parsing, since we never look at more than the one character at the front of the input, we never put characters back on the input to back up and start over, *etc.*

Note: in my RPNcalculator I also used `*the-input*` for meaningful error messages. We can allow this, since it is only used in errors which stop all processing.

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

$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid ...$	Digit
$N \rightarrow DN \mid D$	Number
$A \rightarrow a \mid b \mid c \mid d \mid ...$	Symbolic
$S \rightarrow AS \mid A$	Symbol
$E \rightarrow N \mid S \mid (\ L \ )$	Expression
$L \rightarrow E \ L \mid \epsilon$	List of expressions

Note that, as in our RPNcalculator, we can use some builtin Scheme procedures to avoid parts. 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 #\())
         (not (eq? char #\)))))))
```

Recursive descent procedures We will define four recursive descent procedures to parse Scheme:

1. `parse-expression` is the top-level procedure, and gets the ball rolling. From the grammar we know there will be only three kinds of expressions. The LL(1) character of the grammar tells us how to do the processing:
   - If we find a digit, call `parse-number`.
   - If we find a symbolic character, call `parse-symbol`.
   - If we find a left parenthesis, remove that character from the input and call `parse-list`.
   - Anything else should be an error.
2. `parse-number` is handled just as in the RPNcalculator: keep scanning digits until you find something not a digit, multiplying by 10 and adding along the way. When you get to the end of a number, though, don't push it on a stack, return it as the value of the expression. We are not building an interpreter here (like the RPNcalculator), we're building a parser. A properly parsed number is ... the number itself!
3. `parse-symbol` is handled very similarly to numbers: keep scanning as long as the input is symbolic, and concatenate all the characters into a symbol when you're done. You may find the builtin **Racket** routine `string->symbol` helpful. Return the symbol as the value of the procedure.
4. `parse-list` is a bit tricky, but the grammar tells us we only need to handle two cases:
   - If the input is a right parenthesis, the return `'()`. We found the end of the list.
   - Otherwise, call `parse-expression`, then call `parse-list`, `cons` the two results together and return that.

That wasn't so hard, was it? Note: I left out discussing the handling of whitespace, but that's not hard. Look at what we did in the RPNcalculator.

Parse The `parse` procedure is a simple interface to the grammar:

```
(define parse
  (lambda (str)
    (set-input! str)
    (parse-expression)))
```

It is usually the case that when you have a complex recursive descent parser, you need one more "top-level" routine to get everything rolling.

Program files 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`.