Rewriting `read` in Scheme

CSCI 312 Homework 3

December 25, 2022

File names: Names of files and variables, when specified, must be EXACTLY as specified. This includes simple mistakes such as capitalization.

Individual work: All work must be your own. Do not share code with anyone other than the instructor and teaching assistants. This includes looking over shoulders at screens with the code open. You may discuss ideas, algorithms, approaches, *etc.* with other students but NEVER actual code

Project: A simple implementation of Scheme's `read` procedure in Python is given in the file `read.py` available on the homework website.

Translate this procedure into Scheme. You will have to research Scheme's character and string processing, which has a slightly different flavor from Python's. It is recommended that, just as in Python, you convert the string to a list to do all the processing, since list handling is way easier than string handling. (Just like in Python.)

Pure functions: The biggest difference is that you will make these procedures pure functions. The Python procedures advance through the string by converting it to a list of characters and then repeatedly calling `s.pop(0)` to remove the first character and advance down the string. While the program uses no global variables, *per se*, it nevertheless treats the list of characters as a global since each procedure affects the same object.

While we could do something similar in Scheme, we want to get some practice in fully functional programming. No data structures are destructively modified in this approach. Any new structures we want are nondestructively created from the old data structures.

Likewise there will be no assignment statemeents!

Use recursion: Do not use Scheme's looping constructs, but use recursion for all repeated actions. You can use a named `let` if you like, as this is simply a sugared version of a recursive function, but it is not necessary.

Two return values: Therefore, all procedures except the top-level `read` procedure will return two values, the parsed object, and the remainder of the string.

To return two values within a function, for example 9 and 10, simply wrap them in a `values` form:

```
(define return-two-values
  (lambda ()
    (values 9 10)))
```

To catch both values at the other end, use the `let-values` form:

```
(let-values (((a b) (return-two-values)))
  (list a b (+ a b)))
  =>
'(9 10 19)
```

Tree recursion: Finding both the parsed object and the remaining characters is easy for `parse-symbol` and `parse-number`. Both objects are available by the time the recursion ends.

The only place these values are used are in the `parse-list` procedure, which parses the head of the list, and then the rest of the list. After parsing the head, you need to parse the rest from the list of characters that results *after* you're done parsing the head. Likewise, after parsing the head and the rest of the list, you need to return the list of characters remaining.

There will thus be several intermediate objects to handle:

1. the object at the head of the list
2. the list of characters remaining after the head has been found
3. the object list after the head
4. the list of characters remaining after the end of the list has been found

Make sure you understand clearly where each of these comes from, and what to do with them after they have been found.