EECS 311: Data Structures and Data Management	Program 1
Assigned: 10/21/10	Checkpoint: 11/2/10; Due: 11/9/10

1 Project: Scheme Parser.

In many respects, "the ultimate program" is an interpreter. Why? Because given the right input (i.e., program) it can do anything. Any application that allows the user to define "user defined macros" to manipulate the application has a built in interpreter. In fact, building an interpreter into an application is a common way to make the application extremely extensible. Popular examples of these applications include text editors such as *emacs*, spreadsheet programs such as *excel*, and web browsers (e.g., using AppleScript). While you will not write a scheme interpreter in this project, you will get pretty close. Your first project is to write a scheme parser.

1.1 Scheme Review.

Scheme is a popular programming language that is especially easy to interpret. One of the reasons for this is that scheme data types look just like scheme programs. Scheme supports standard data objects such as strings and numbers as well as lists of data objects (note the recursive definition). Consider: `(+ 1 (* 2 3))`. This is a list with three elements. The first element is the symbol `+`. The second element is the number `1`. The third element is the three element list `(* 2 3)`. Note that this scheme data looks exactly like a scheme program, the one that adds 1 to the product of 2 and 3.

Let us take a more detailed look at scheme lists. The building blocks of scheme lists are

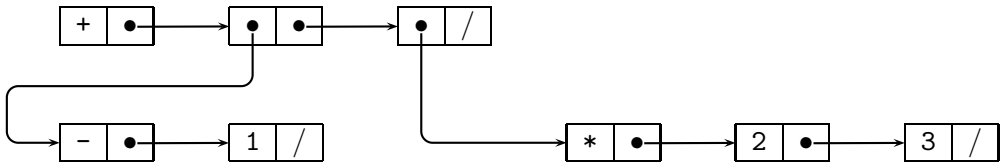
- the *emptylist*, represented as `()`, and
- the *pair a b*, represented as `(a . b)`.

A *list* is either an emptylist or it is a pair where the second field is a list (note the recursive definition). Thus, `(+ . (1 . (2 . ())))` is a list; though, it is a bit hard to read. Since lists are so commonly used in scheme a special list syntax has been adopted, e.g., `(+ 1 2)` instead of `(+ . (1 . (2 . ())))`. Obviously the special list syntax is easier to write and read for a human, but the pair syntax is closer to what is going on internally in a scheme data structure. The list and pair syntaxes can be combined in arbitrary ways, e.g, `(a b . (c d))` is equivalent to `(a b c d)`.

1.2 Scheme Objects in Java.

To represent scheme data you will create a class hierarchy of scheme objects. The parent class is `SchemeObject`. Subclasses you will implement are `SchemePair(`a`,`b`)`, where member fields a and b are `SchemeObject`s themselves; `SchemeNull`, which both represents an emptylist as well as the end of a non-empty list; and `SchemeSymbol(`x`)`, where member field x is a string.

The very first step of writing an interpreter is parsing. That is, reading from input the string "`(+ (- 1) (* 2 3))`" and turning it into a `SchemeObject`; in particular, the following list of lists:



For this programming assignment you are to write such a parser. We will divide this task into two steps, *tokenizing* and *parsing*.

1.3 Tokenizing.

The first step (commonly called "tokenizing") is turning a sequence of characters into a sequence of *tokens*. For instance, "punctuation" such as the `"("`, `")"`, and `"."` characters are special tokens. Symbols such as `"+"`, `"car"`, and `"25"` are tokens.[1] As an example the sequence of characters `")hello  25 . ( ("` should be converted into a sequence of tokens: `CloseParen`, `TokenSymbol("hello")`, `TokenSymbol("25")`, `Period`, `OpenParen`, `OpenParen`. Notice that whitespace helps delineate symbols, but is not itself a token. Also notice, that this particular sequence of tokens really does not correspond to a well-formed scheme object.

Tokenizing can be quite complicated; fortunately, the subset of scheme we are parsing is quite simple. To aid in this task, `java.io.BufferedReader` provides a `readLine()` method that returns a `String` object. The `String` class provides a `split()` method. For instance if `String s = ")hello 25 . ( ("` then `s.split("\\s+")` returns the five element array `[")hello", "25", ".", "(", "("]`. While this does not quite do the trick because it does not separate the string `")hello"`, it is a good start.

1.4 Parsing.

The second step (commonly called "parsing") is to convert the sequence of tokens into a sequence of `SchemeObject`s. Here the most important task is to match up parentheses. This is actually very easy using a stack. The approach we outline here is for *shift-reduce* parsing. Shift-reduce parsing simply repeats the following steps:

1. While the top of the stack matches any of a set of patterns, *reduce* (i.e., simplify).
2. *Shift* (i.e., push) a new token (from the input) onto the top of the stack.
3. Repeat.

Our stack needs to hold both `Token`s and `SchemeObject`s. While shift only pushes new tokens onto the stack, reduce will replace combinations of `Token`s and `SchemeObject`s with `SchemeObject`s that combine them.

A natural approach would be to see if the top of the stack looks like "..., `OpenParen`, $\texttt{SchemeObject}_1$, ..., $\texttt{SchemeObject}_k$, `CloseParen`." and then reduce it to the appropriate scheme list object; however, this requires looking at $k+2$ stack elements at once. In general, it would be costly (in runtime) to pattern match large numbers of stack elements at once; instead you should use a reduce procedure that only ever operates with the few elements at the very top of the stack. The idea behind this

[1] Real scheme interpreters treat numbers and other symbols differently. For simplicity, we will treat numbers as symbols.

reduce technique is a syntactical equivalence, e.g., between `(+ 1 2)` and `(+ . (1 . (2 . ())))`. As discussed in Section 1.1, the former is easier to read and write for a human, but the latter is closer to what is going on internally in a data structure.

This motivates the transformation from what a human would like to write to what a computer can understand. To illustrate this, consider taking the expression `(+ 1 2)` and transforming it, starting from the tail of the list, from human readable notation to the internal data representation. Below, the internal data representation is underlined.

$$\begin{aligned}\texttt{(+ 1 2)} &\Rightarrow \texttt{(+ 1 2 . ())} \\ &\Rightarrow \texttt{(+ 1 . (2 . ()))} \\ &\Rightarrow \texttt{(+ . (1 . (2 . ())))} \\ &\Rightarrow \texttt{(+ . (1 . (2 . ())))}.\end{aligned}$$

This suggests a reduce technique that builds up the list backwards and each reduce uses at most the top four elements in the stack. Consider, below, the patterns that reduce should process and what the stack contents should look like after the reduce.

1. Convert a symbol token to a symbol object:

   Before: ..., `TokenSymbol(`x`)`.

   After: ..., `SchemeSymbol(`x`)`.

2. Make a long list smaller by contracting the last two elements into a pair:[2]

   Before: ..., `SchemeObject`$_a$, `Period`, `SchemeObject`$_b$, `CloseParen`.

   After: ..., `Period`, `SchemePair(SchemeOjbect`$_a$`,SchemeObject`$_b$`)`, `CloseParen`.

3. To finish a list simply remove the opening parenthesis, period, and closing parenthesis:[3]

   Before: ..., `OpenParen`, `Period`, `SchemeObject`, `CloseParen`.

   After: ..., `SchemeObject`.

4. If the top item on the stack is a closing parenthesis and none of the above patterns matched, then this is the end of a long list. Convert closing parenthesis to a period preceding an empty list:[4]

   Before: ..., `CloseParen`.

   After: ..., `Period`, `SchemeNull`, `CloseParen`.

Notice that all of these patterns involve matching only the very top several elements on the stack. This is an important aspect of the shift-reduce parser! A successful parse should result in a stack containing only `SchemeObject`s. See Section 4 for an example.

[2]This is appropriate because `(a ... x y . z)` is equivalent to `(a ... x . (y . z))`.
[3]This is appropriate because `(. x)` is equivalent to `x`.
[4]This is appropriate because `(a ... z)` is equivalent to `(a ... z . ())`.

1.5 Interpreting.

If we were implementing an interpreter, the next step would be to take the output of the parser (a list of `SchemeObject`s) and interpret them one at a time, that is, treat them as scheme code and evaluate them. We will not complete the interpreter for this homework. Instead we will just write a routine to print out each each object (recursively).

2 Tasks.

1. Implement the `SchemeObject` class hierarchy: `SchemePair(`a`,`b`)`, `SchemeNull`, and `SchemeSymbol(`x`)`.
2. Implement a method on each class in the `SchemeObject` hierarchy that (recursively) prints out the object in proper scheme syntax, i.e., the same syntax your parser reads. Your lists should be printed out in the special list syntax, e.g., `(a b c)`.
3. Implement the Stack ADT. You may wish to add, to the standard stack operations, the operation *peek* where `peek(`i`)` returns the ith element from the top of the stack. This will be useful in your shift-reduce parser. You can assume that i is a constant, i.e., it is ok for `peek(`i`)` to take $O(i)$ time.
4. Implement the `Token` class hierarchy: `CloseParen`, `OpenParen`, `Period`, and `TokenSymbol(`x`)`.
5. Implement a class that allows tokens to be read from an input stream. For instance the constructor for such a class might take an input stream as input (such as `System.in`). The class might provide a method `nextToken()` that returns the next token that is on the input stream. Repeated calls to `nextToken()` would return the sequence of tokens on the input stream.
6. Implement the shift-reduce parser as discussed above. For instance you might implement a class with a constructor that takes your tokenizing class from Task 5. The class might provide a method `nextObject()` operates the shift-reduce parcer, calling `nextToken()` needed, until there is a single `SchemeObject` on the stack. The routine should then pop this object off the stack and return it.
7. Implement an interactive program that reads in scheme expressions and prints them out. Each expression read in should be printed on a newline. In the example below the input is underlined:

```
parse> 1 (2) hello ()
1
(2)
hello
()
parse> (*      (+ 2    3 )     (- 4 5))
(* (+ 2 3) (- 4 5))
parse> ( hello
world )
(hello world)
parse>
```

This example demonstrates how whitespace in input can be arbitrary, while the whitespace in the output is always in a standard form.

3 Logistics.

This assignment will be graded out of a total of 40 points.

This program will be completed in two parts. The first part is due on Thursday, 10/28/10 (at midnight) and will be graded out of 20 points. The second part is due on 11/04/10 (at midnight) and will be graded out of 20 points. If you fix any problems with the first part when you turn in the second part you can regain up to half the points lost. For example, you made two mistakes on the first part and lost two points each. If you fix one of the mistakes for the final submission you will get one point back. If you fix both mistakes for the final submission you will get two points back.

Part 1: Complete Tasks 1-3. You should also write and turn in a test program that thoroughly verifies that all tasks for this part are properly completed.

Part 2: Complete Tasks 4-7.

Submitting your code. The T-Lab (Tech F252) is available for your programming needs and the TA will hold extra office hours (TBA) in the T-Lab to assist you. Your programs should compile using `javac` on the command line. To submit your program send the zipped source code by email to the TA. Use the subject line "SUBMIT PROG1 PART1" and "SUBMIT PROG1 PART2" for each respective part. Do not submit executables. If you use a *makefile*, include the makefile in your submission. If you compile your program directly on the command line, specify the command you use for compilation in your email. Anything more complicated should be explained in a README file.

Grading guidelines. You will be graded on your ability to write efficient code. You will be graded on your ability to write reasonable Java code. You will be graded on your ability to implement the required tasks. You will be graded on your ability to implement and use data structures from class.

Resources. You may consult your text book or other books on Java and data structures. You must build your own advanced data structures; you may not use data structure implementations in the Java class library. You must not copy code from anywhere. You may talk with your classmates about the project at a high level, but your implementations must be 100% original. You may consult with the instructor and TA on any aspect of the project.

4 Example: Shift-Reduce in Action

Suppose we read in input `"(+ 1 2)"` into token sequence `OpenParen`, `TokenSymbol("+")`, `TokenSymbol("1")`, `TokenSymbol("2")`, and `CloseParen`. The shift-reduce parser takes the following steps. Initially the stack is empty. Changes from one step to the next are underlined.

0. Initially empty stack.
   Stack: .

1. Nothing to reduce, shift token `OpenParen` onto stack.
   Stack: `OpenParen`.

2. Nothing to reduce, shift token `TokenSymbol("+")` onto stack.
   Stack: `OpenParen`, `TokenSymbol("+")`.

3. Reduce using Pattern 1.
   Stack: `OpenParen`, `SchemeSymbol("+")`.

4. Nothing to reduce, shift token `TokenSymbol("1")` onto stack.
   Stack: `OpenParen`, `SchemeSymbol("+")`, `TokenSymbol("1")`.

5. Reduce using Pattern 1.
   Stack: `OpenParen`, `SchemeSymbol("+")`, `SchemeSymbol("1")`.

6. Nothing to reduce, shift token `TokenSymbol("2")` onto stack.
   Stack: `OpenParen`, `SchemeSymbol("+")`, `SchemeSymbol("1")`, `TokenSymbol("2")`.

7. Reduce using Pattern 1.
   Stack: `OpenParen`, `SchemeSymbol("+")`, `SchemeSymbol("1")`, `SchemeSymbol("2")`.

8. Nothing to reduce, shift token `CloseParen` onto stack.
   Stack: `OpenParen`, `SchemeSymbol("+")`, `SchemeSymbol("1")`, `SchemeSymbol("2")`, `CloseParen`.

9. Reduce using Pattern 4.
   Stack: `OpenParen`, `SchemeSymbol("+")`, `SchemeSymbol("1")`, `SchemeSymbol("2")`, `Period`, `SchemeNull`, `CloseParen`.

10. Reduce using Pattern 2.
    Stack: `OpenParen`, (`SchemeSymbol "+"`), (`SchemeSymbol "1"`), `Period`, `SchemePair(SchemeSymbol("2"), SchemeNull)`, `CloseParen`.

11. Reduce using Pattern 2.
    Stack: `OpenParen`, `SchemeSymbol("+")`, `Period`, `SchemePair(SchemeSymbol("1"), SchemePair(SchemeSymbol("2"), SchemeNull))`, `CloseParen`.

12. Reduce using Pattern 2.
    Stack: `OpenParen`, `Period`, `SchemePair(SchemeSymbol("+"), SchemePair(SchemeSymbol("1"), SchemePair(SchemeSymbol("2"), SchemeNull)))`, `CloseParen`.

13. Reduce using Pattern 3.
    Stack: `SchemePair(SchemeSymbol("+"), SchemePair(SchemeSymbol("1"), SchemePair(SchemeSymbol("2"), SchemeNull)))`.

14. Nothing to reduce, single scheme object on stack, pop it off stack and return it.
    Stack: .