EECS 214/395 Data Structures Fall 2014 Program 1

Posted: Thursday October 9th
Due: 11:59pm Friday October 24th
% of Final Grade: 5%

Programming projects in this course will emphasize both fundamentals of data structures as discussed in lecture; and add a design component whereby students should be able to implement multiple data structures (including understanding inheritance where applicable) as building blocks towards a larger, integrated purpose.

In this project, students will build a simplified Scheme parser.

Scheme Introduction

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.

So Scheme uses "pre-order" of operations within parenthesis. Rather than writing "(1 + 2)" it writes "(+ 1 2)" to mean the exact same thing; the list structure also allows "(+ 1 2 3)" which actually saves a list element compared to "(1 + 2 + 3)". The "+" could be replaced by any number of other operators. For instance it could be "inc" to mean "increment," such that a list as "(inc 4)" should evaluate to 5.

Parsing

Parsing requires reading from input the example string "(+ (- 1) (* 2 3))" and turning it into a SchemeObject; in particular, the following list of lists:

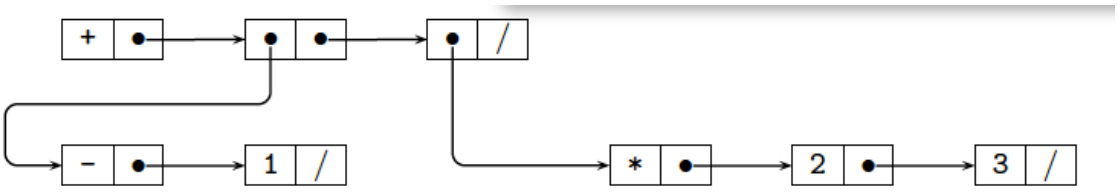

Figure 1

Notice that a SchemeObject is inherently a linked list. And then within it, there are possibly embedded lists. Look at the original example string. It is "inherently" one long parenthetical expression; and then within it, there are possibly embedded parenthetical expressions. For parsing, the most important task is to match up the parentheses. This is actually very easy using a stack.

Think of the characters of the example string as "tokens." So '(' is a token, and '-' is a token, and '1' is a token, etc. We can read a list of tokens into a stack. Any time we encounter a ')' token, we become aware that a sub-list is ending. At this point, the sub-list can be built by popping successive elements from our stack until we find the beginning '('. *Then a pointer to the sub-list should be returned to our stack*, such that sub-list is an element embedded into a higher-order list (the discovered expression is embedded in parentheses into a higher-order expression).

Your Tasks

- Download the python file template "Pboth_Project1_acj123.py", which is attached to this assignment.

- Test code appears within a "__main__" block at the bottom; there are also some print statements which appear within the function "Get_Scheme_Object_acj123", which you should not remove; these are required to facilitate grading.

- Where my netID appears, change it to be yours.

- When you submit your file, it should be named as "P##_Project1_[netid]" where ## replaces "both" in the template, and is either 27 or 34 *to indicate which version of Python you are using*.

- Write the five total methods within Stack and Queue classes (10 points per method)

- Write the function Get_Scheme_Object_[netid] to convert the input string to a SchemeObject similar to the picture above. (50 points) You should take the following steps:

  - You may assume that the input string has been pre-processed such that there is exactly one space between tokens. With an iterative process, convert the input string into a queue of list nodes, one node per token. You should leave the numbers as strings. (A queue of list nodes is not a strictly necessary step, as we could process the string directly into the stack; however, this assignment asks you to design and use a queue as an intermediate step for the sake of designing and using a queue.)
  - Move elements from the queue into a Working stack. When you encounter a node with value ')', a parenthetical expression is ending. Pop elements back off the stack and into a new stack (as a Scheme_Object in the code) to

represent this expression. When the new stack is finished, Push a pointer to it back onto the Working stack so that it is included in higher-order expressions.

- In this project, two kinds of objects are used as "nodes" to build a SchemeObject as in Figure 1. Scheme_Objects are the Heads of lists in the picture, and are themselves lists (inherit from stack) with Head pointers; they have values which are operators (stored as strings). A SchemeObject can be an element in a higher-order list. All other nodes are List_Nodes, whose values are either pointers to SchemeObjects or numbers (stored as strings). So in Figure 1, the head node with value '+' is a SchemeObject with a pointer Head to a list. The other two nodes in the top row are List_Nodes.

- **CODING RESTRICTIONS.** Part of the objective of this assignment is to build fundamental data structures, not to use language built-ins. You may not use the built-in functions list, tuple, dict or similar; you may not use the built-in string functions split, append, or similar. You may not use the built in list or dict constructors as [ ] or { }. In fact, you should not use characters [ ] or { } anywhere in the program with one exception: within the initial iterative processing of the input string, you may use Python's [ ] operator to "slice substrings." Submissions might be uniformly searched by program or manual check for these prohibited uses of Python. If you're not sure if you can use something or not, please ask. See the course syllabus for further explanation on restrictions.

- You can check your answer by running the test case given in __main__ block against the output below.

- Submit your project solutions via Blackboard.

Method-Writing "Check List"

Always, when you write methods, be mindful of:

- Invariants; if you are changing the data structure, be sure to fix up its pointers and fields so that future methods can be called correctly
- Boundary conditions; are you removing something from an empty data structure, or do you need more memory?
- Pointer order of operations; if you have to give new assignments to two pointers, does the order in which you do it matter? One might be your only pointer to a part of your data structure.

Python Tips

Formatting. Remember, Python does not requires brackets as { }, or end-of-line signifiers as ;. Instead, it uses strict indenting to delineate blocks within classes, methods, conditionals and loops. By default, lines end where the text ends; this can be subverted by enclosing a multiple-line expression in parentheses ( ) such that it is "open" when the first line ends.

Object Types. You should not expect any problems with Types in this project, but you should be aware of them. Python is an "un-typed" language. This means that you do not have to declare the types of variables before assigning them, and you will not see Type declarations in the template. This has both benefits and drawbacks. The benefit is that a variable's type can be dynamically changed by simply assigning it to anything else as needed. (We use this when we let the value of a List_Node be either a string or a pointer to an embedded list!) The drawback is that if you do this by accident, Python will not throw you an error as a red flag. Casting is sometimes necessary (usually between string and "number" types).

Classes. You do not need to design any classes from scratch for this assignment but do need to know how to navigate them, and how to implement methods. When the class is declared, inheritance is designated in parentheses immediately following the class' name. Constructors are located within "__init__" methods. All class methods, when they are declared, are required to explicitly identify a pointer back to the current object in the first place of its input list. By convention this variable is named "self". When class methods are called, they do not pass a value into "self". So if a method is declared as MethodA(self, v1), it is called with just MethodA(v2_into_v1). Within methods, class field FieldB should then be accessed using self.FieldB. For examples of these, see the template.

Testing

The template comes with a string provided within the __main__ function. This string is given to help you test your code, but it will not be the input string used for final grading. Please remember, do not change any of the print statements or __str__ methods (which tell objects how to "print" themselves") within classes. Then it should result in the following output to the screen, which shows how the data structures build up.

```
begin queue
(
( inc
( inc (
( inc ( +
( inc ( + 1
( inc ( + 1 2
( inc ( + 1 2 (
( inc ( + 1 2 ( -
( inc ( + 1 2 ( - 12
( inc ( + 1 2 ( - 12 4
( inc ( + 1 2 ( - 12 4 )
```

(inc (+ 1 2 (- 12 4) (
(inc (+ 1 2 (- 12 4) (*
(inc (+ 1 2 (- 12 4) (* (
(inc (+ 1 2 (- 12 4) (* (-
(inc (+ 1 2 (- 12 4) (* (- 1
(inc (+ 1 2 (- 12 4) (* (- 1)
(inc (+ 1 2 (- 12 4) (* (- 1) 7
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (+
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (+ 3
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (+ 3 (
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (+ 3 (-
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (+ 3 (- (
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (+ 3 (- (half
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (+ 3 (- (half 12
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (+ 3 (- (half 12)
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (+ 3 (- (half 12) 1
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (+ 3 (- (half 12) 1)
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (+ 3 (- (half 12) 1))
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (+ 3 (- (half 12) 1)))
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (+ 3 (- (half 12) 1))))
(inc (+ 1 2 (- 12 4) (* (- 1) 7 (+ 3 (- (half 12) 1)))))

begin stack

(

inc (

(inc (

+ (inc (

1 + (inc (

2 1 + (inc (

(2 1 + (inc (

- (2 1 + (inc (

12 - (2 1 + (inc (

4 12 - (2 1 + (inc (

(SO-start - 12 4 SO-end) 2 1 + (inc (

((SO-start - 12 4 SO-end) 2 1 + (inc (

* ((SO-start - 12 4 SO-end) 2 1 + (inc (

(* ((SO-start - 12 4 SO-end) 2 1 + (inc (

- (* ((SO-start - 12 4 SO-end) 2 1 + (inc (

1 - (* ((SO-start - 12 4 SO-end) 2 1 + (inc (

(SO-start - 1 SO-end) * ((SO-start - 12 4 SO-end) 2 1 + (inc (

7 (SO-start - 1 SO-end) * ((SO-start - 12 4 SO-end) 2 1 + (inc (

(7 (SO-start - 1 SO-end) * ((SO-start - 12 4 SO-end) 2 1 + (inc (

+ (7 (SO-start - 1 SO-end) * ((SO-start - 12 4 SO-end) 2 1 + (inc (

3 + (7 (SO-start - 1 SO-end) * ((SO-start - 12 4 SO-end) 2 1 + (inc (

(3 + (7 (SO-start - 1 SO-end) * ((SO-start - 12 4 SO-end) 2 1 + (inc (

- (3 + (7 (SO-start - 1 SO-end) * ((SO-start - 12 4 SO-end) 2 1 + (inc (

(- (3 + (7 (SO-start - 1 SO-end) * ((SO-start - 12 4 SO-end) 2 1 + (inc (

half (- (3 + (7 (SO-start - 1 SO-end) * ((SO-start - 12 4 SO-end) 2 1 + (inc (

12 half (- (3 + (7 (SO-start - 1 SO-end) * ((SO-start - 12 4 SO-end) 2 1 + (inc (

(SO-start half 12 SO-end) - (3 + (7 (SO-start - 1 SO-end) * ((SO-start - 12 4 SO-end) 2 1 + (inc (

1 (SO-start half 12 SO-end) - (3 + (7 (SO-start - 1 SO-end) * ((SO-start - 12 4 SO-end) 2 1 + (inc (

(SO-start - (SO-start half 12 SO-end) 1 SO-end) 3 + (7 (SO-start - 1 SO-end) * ((SO-start - 12 4 SO-end) 2 1 + (inc (

(SO-start + 3 (SO-start - (SO-start half 12 SO-end) 1 SO-end) SO-end) 7 (SO-start - 1 SO-end) * ((SO-start - 12 4 SO-end) 2 1 + (inc (

(SO-start * (SO-start - 1 SO-end) 7 (SO-start + 3 (SO-start - (SO-start half 12 SO-end) 1 SO-end) SO-end) SO-end) (SO-start - 12 4 SO-end) 2 1 + (inc (

(SO-start + 1 2 (SO-start - 12 4 SO-end) (SO-start * (SO-start - 1 SO-end) 7 (SO-start + 3 (SO-start - (SO-start half 12 SO-end) 1 SO-end) SO-end) SO-end) SO-end) inc (

(SO-start inc (SO-start + 1 2 (SO-start - 12 4 SO-end) (SO-start * (SO-start - 1 SO-end) 7 (SO-start + 3 (SO-start - (SO-start half 12 SO-end) 1 SO-end) SO-end) SO-end) SO-end) SO-end)

(SO-start inc (SO-start + 1 2 (SO-start - 12 4 SO-end) (SO-start * (SO-start - 1 SO-end) 7 (SO-start + 3 (SO-start - (SO-start half 12 SO-end) 1 SO-end) SO-end) SO-end) SO-end) SO-end)