Parser

For this part of the assignment, you'll implement a parser for Scheme, in C. (You must implement the parser from scratch - don't use Yacc, Bison, ANTLR, or any similar tool, even if you know what it is and how to use it.)

This is a "pair" assignment, which means that if you are working on a team with someone else, you and your partner should do your best to engage in the pair programming model. At any point in time, one of you is "driving," i.e. actually using the keyboard and mouse. The other is actively engaged following along, preventing bugs, and providing ideas.

You should make sure that over the course of an assignment that you spend roughly the same amount of time each "driving." I will also ask you to turn in a form rating the work that your partner does. My recommendation is to take turns approximately every 15 minutes or so. Set a timer to help you remember.

1. Get started

Download this 04-parser.zip file, and extract the zip file to your ProgrammingLanguages folder that you've been working in.

2. Assignment details

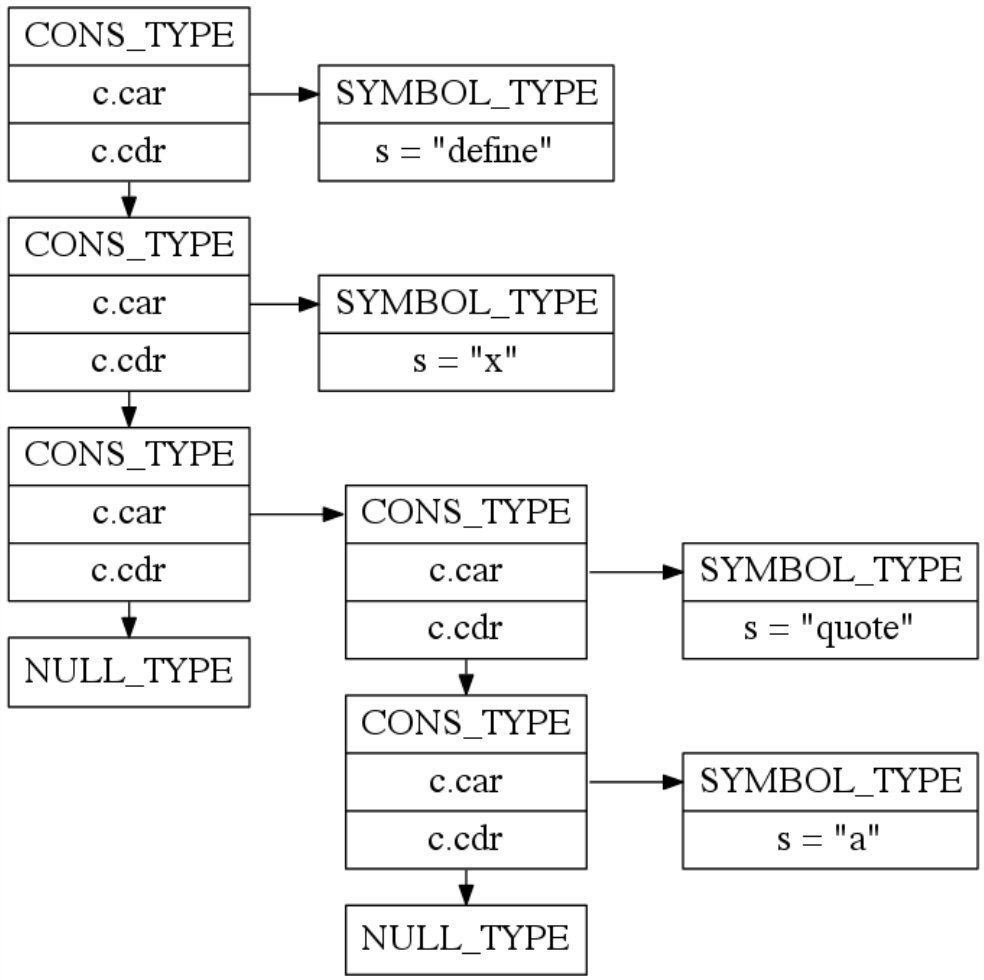
You should create a `parser.c` file that implements the functions specified in `parser.h`. Particularly, you will need to write a function `parse` that uses the token list from the last assignment to construct a list of parse trees.

For example, here is a syntactically correct Scheme program:

```
(define x (quote a))
(define y x)
(car y)
```

Any Scheme program consists of a list of S-expressions. An S-expression always has parentheses around it, unless it is an atom. Note that a Scheme program itself, then, is not a single S-expression; it is a list of S-expressions.

Your `parse` function should therefore return a list, using the linked list structure that we have been using, of parse trees. A parse tree, in turn, uses the linked list structure again to represent a tree. For example, the expression `(define x (quote a))` would become the following parse tree, where each box is a `SchemeVal`:



The above parse tree structure would be the first item in a 3-item linked list that represents the above Scheme program. (***NOTE: Where it says `NULL_TYPE`, it should say `EMPTY_TYPE`. That's an update we made for this term.)

Do not include parentheses in your parse tree. Their purpose is to tell you the structure of the tree, so once you have an actual tree you don't need them anymore - they'll only get in the way. (Technically, this means that you're creating an *abstract syntax tree*, not a parse tree, since not all of the input is preserved.)

Parsing languages can be pretty complicated. The good news is that parsing Scheme is really easy, at least in a relative sense. Since your Scheme code IS a tree, you just need to be able to read it as such and turn it into a tree in memory. You can do it with a stack of tokens:

- Initialize an empty stack (which we implement as a linked list of SchemeVals).
- While there are more tokens:
  - Get the next token.
  - If the token is anything other than a close paren, push it onto the stack.
  - If the token is a close paren, start popping items from your stack until you pop off an open paren (and form a list of them as you go). Push that list back on the stack.

So you'll need a stack… your linked list is a fine candidate for it.

You'll also need to write a `printTree` function. The output format for a parse tree of valid Scheme code is very simple: it looks exactly like Scheme code! To output an internal node in the parse tree, output `(` followed by the outputs of the children of that internal node from left to right followed by `)`. To output a leaf node (a non-parenthetical token), just output its value as you did in the tokenizer, sans type.

3. Syntax errors

As with the tokenizer, your parser should never crash with a segmentation fault, bus error, or the like. Here are the different types of syntax errors you should handle:

1. If the input code ever has too many close parentheses (in other words, if you ever encounter a close paren that doesn't match a previous open paren), print `Syntax error: too many close parentheses`.
2. If the `parse` function returns an incomplete parse tree and the end of the input is reached and there are too few close parentheses, print `Syntax error: not enough close parentheses`.

If you ever encounter a syntax error, your program should exit after printing the error - don't do any more parsing. This is why we wrote the function `texit`. Before exiting, you should output the text "Syntax error". Also feel free to print Scheme code around the error to be helpful if you can, though this optional.

Most production parsers continue to try to parse after an error to provide additional error messages. Your efforts here may help you to gain some sympathy as to why all error messages after the first one are questionable!

4. Sample executions

```
$ cat test-in-01.scm
(if 2 3)
(+ ))

$ cat test-in-02.scm
(define map
  (lambda (f L)
    (if (null? L)
        L
        (cons (f (car L))
              (map f (cdr L))))))

$ cat test-in-03.scm
1 2 (3)

$ ./interpreter < test-in-01.scm
Syntax error: too many close parentheses

$ ./interpreter < test-in-02.scm
(define map (lambda (f L) (if (null? L) L (cons (f (car L)) (map f (cdr L))))))

$ ./interpreter < test-in-03.scm
1 2 (3)
```

5. Formatting your output

You may find that it is easier to produce output similar to the above but with extraneous white space. For me, I had to hack some extra code to make sure that the last item of a list didn't have a space between it and the closing paren that followed it. You don't have to worry about this if you don't want to; you may have extraneous whitespace in your output if you wish.

6. Sample code

Here is a rough approximation of part of my parse function. My `addToParseTree` function takes in a pre-existing tree, a token to add to it, and a pointer to an integer `depth`. `depth` is updated to represent the number of unclosed open parentheses in the parse tree. The parse tree is complete if and only if `depth` is 0 when the parse function returns.

```
SchemeVal *parse(SchemeVal *tokens) {

    SchemeVal *tree = makeNull();
    int depth = 0;

    SchemeVal *current = tokens;
    assert(current != NULL && "Error (parse): null pointer");
    while (current->type != NULL_TYPE) {
        SchemeVal *token = car(current);
        tree = addToParseTree(tree, &depth, token);
        current = cdr(current);
    }
    if (depth != 0) {
        syntaxError();
    }
}
```

7. How to build your code

Look back at the tokenizer assignment, in the section titled "The files that you start with," on how to build your code as well as how to use my binaries for the previous assignments if you wish.

8. Testing your code

Run the tests with `python3 test-m` and `python3 test-e`, as usual, which will automatically compile all of your code and run the tests.

Your code should have no memory errors when running on any input (correct or incorrect) using `valgrind`. The testing scripts will automatically run `valgrind` on your code, and show you if there are memory errors.

9. The single quote

Dealing with the special case of the single quote (i.e. `'a`, or `'(a b c)`) means that we need to redefine "push onto the stack," which appears a few times in the parsing algorithm. Here is the new definition of "push onto the stack", which you only need to pass all of the E tests:

- Proceed as usual so long if the stack doesn't have a single quote `'` on top.
- Also proceed as usual if pushing on a left paren, no matter what the stack looks like.
- In all other cases…
    - Pop the single quote off the stack. Then proceed as if the token sequence had `(quote ....)` wrapped around the value you're pushing. In other words, create a new subtree consisting of `quote` and the new token, and then push that subtree onto the stack instead of the original value you were going to push.

10. What to submit

Follow the instructions for the last assignment: submitting should be the same.

Good luck, and have fun!

11. Preparing for celebrations of knowledge

The celebration of knowledge may ask you to write on paper C code to solve problems similar to the above. It might have you write the code from scratch; it might have you fill in some blanks in partially written code; or it might use some other method for having you demonstrate that you can write code. To prepare, you should practice writing C code on paper that can solve any of the above functions, or others like it.

There may also be questions regarding the parsing algorithm that you used, in the abstract. You might be asked to demonstrate with pseudocode, or by demonstrating partial output, or other similar mechanisms, that you are aware of how this parsing algorithms works.

If you're looking for more examples to practice:

- Format your output so that it resembles Guile output as closely as possible. Specifically, don't have an extraneous space after the last item in a list before the closing paren.
- Correctly handle square brackets. Some dialects of Scheme allow you to use square brackets as an alternative to parentheses anywhere you like, so long as they match accordingly. Here is an example:

  ```
  (define [lambda (x) [+ x 1]])
  ```

  Once your parse tree is constructed, you should retain no memory of whether parentheses or square brackets were used, and your output should only show parentheses (just as Guile does). However, you'll need to add some extra details to the parse to make sure the input is correct. For example, even though you'll eventually transform `(a [b c] d)` to `(a (b c) d)`, the input `(a [b c) d]` should result in a syntax error.

- Correctly parse dotted pairs, e.g. `(a . b)`.

This assignment was originally created by David Liben-Nowell and has since been updated by Dave Musicant, Jed Yang, and Laura Effinger-Dean.