Project PL3

Due Dates: Part 1: October 19
Part 2: November 7

Purpose

In this project, you will get first-hand experience using a BNF grammar and writing part of a compiler. You will also gain insight writing software in a group. To this end, you may form a group of N students such that $1 \leq N \leq 3$.

Problem

The GNU Project has hired your group to help implement a compiler for a new, smaller version of C, called *lC* (little C). You are responsible for the syntax checking (parsing) portion of the compiler. Since the C language is stable and used for system programs, the parser will be written in C. The program should implement top-down (predictive) parsing, similar to what we did in class. To write the program, you will use the recursive descent technique applied to the given BNF rules. You can find another example of this technique on the Wikipedia page and `littleComp.c`; links for both are on the course web page. There are additional tutorials/notes available on the web.

This is an involved project. To make sure you are on the right track, the first task is to write a scanner/tokenizer. That is the part of the compiler that *scans* the input file and finds *tokens*, which is essential for a working parser. Thus, the smaller Part 1 is to complete the tokenizer code.

The larger Part 2 is to write a complete *lC* parser in C.

Input

The input to your program (Parts 1 and 2) is a program written in *lC*. To make things easier, there will be a blank between tokens. The input filename should be given on the command line, as shown below. For Part 1, the name of your tokenizer should be **tokenize**:

tokenize aSampleProgramInC.c

The name of your syntax analyzer should be **parse**:

parse lCprogram.c

Note that these names only have meaning in the Makefile, where you create the executable files.

Output

Part 1: The tokenizer should read the input program and simply display a list of the tokens it finds. The output should show five tokens per line, except the last line, which may show fewer. The tokens should be separated by commas and a space, as in:

```
x, =, y, ;
```

Part 2: The program should check the input for any syntax errors as defined by the BNF grammar. If there are no errors, the program should display a message to that effect. If there are errors, the program should display the line number, the offending line, and give an appropriate error message(s). The more lines/errors that your parser can find, the better. You will now see how difficult it can be to write an "appropriate" error message and may get a new appreciation for the compilers you use regularly. To get the maximum points, you should also indicate where in the line the error occurred by using a ˆ or other symbol:

```
7: int a .
         ^
Error - semicolon expected
```

Specifics

- The parser should be written in ANSI standard C.
- The code should be adequately commented; see your earlier projects.
- Except for the current and possibly the look-ahead token, you should not use global variables.
- The grading for this project will be based mostly on how well your program finds and reports syntax errors. I will barely look at your code except for to read comments and to check that you have used functions adequately (that is, functions are not too long, including `main()`, and there is an overall program design).

Notes

- It may be easier to write the scanner/tokenizer using a finite state machine (FSM).
- Hint: it may be best to read the file one character at a time (rather than using strings).
- It may be easier to write the parser from syntax diagrams than from the BNF rules. Thus, you may want to convert all of the given BNF rules to syntax diagrams, and then generate your code from the diagrams.
- Since you are still new to C, this may take a while, even with a group. Start by writing code that will read and display the input properly. Then be sure that you can get tokens properly. Only at that point should you implement some of the short rules, one at a time. Continue adding in new rules until your parser is complete.
- A nice thing about this assignment is that you can use the readily-available C compilers in Linux, which are `gcc` and `cc`. You can use these to check your work as you go along; that is, your test program written in *lC* should compile properly (or not!) using one of the available C compilers.
- It is not easy to write a parser. Although Part 1 is due two weeks before the entire project is due, your group should already be working on the parser *before* the tokenizer is turned in!
- For Part 1, send **one** tarball of your group's tokenizer via email the usual way by 11:59:59 PM on October 19^{th}. The submission should include a Makefile. In class the next day, hand in a sheet of paper listing the names of your group members and the name that will be on the electronic submission. A good choice for the submission would be a creative name for your group or "company." Then append a "PL3.1" to your file name to indicate Part 1; for example: `GooseCoPL3.1.tar`. This portion will be worth 25% of the entire project. The grading of this portion will be based solely on the output of your program.
- For Part 2, send **one** tarball of your group's completed project via email, using the same name as for Part 1, except append a 2 to the name: `GooseCoPL3.2.tar`. Once again, the submission should include a Makefile. In the next class, submit **only** one or two sheets of your program that include your introductory comments which, in turn, should include the names of everyone in the group. Do **not** print out the entire program! This portion will be worth 75% of the project. (Note you still have more time to work on the project that evening.)

BNF Rules for *lC*, in alphabetical order (taken from *C: A Reference Manual*, by Harbison and Steele, Jr., 4th Edition, Tartan Inc., 1995, with revisions):

- the top level (i.e., root) is `<program>`
- this BNF has not been reduced; that is, you can make it simpler by applying some basic reductions
- let me know if you find any typos or missing rules!

```
<add-op> ::= + | -

<additive-expression> ::= <multiplicative-expression> |
                          <additive-expression> <add-op> <multiplicative-expression>

<assignment-expression> ::= <conditional-expression> |
                            <identifier> <assignment-op> <unary-expression>

<assignment-op> ::= =

<compound-statement> ::= { } | { <declaration-list> <statement-list> } |
                         { <declaration-list> } | { <statement-list> }

<conditional-expression> ::= <logical-or-expression>

<conditional-statement> ::=  <if-statement> | <if-else-statement>

<constant> ::= <integer-constant> | <floating-constant>

<declaration> ::= <declaration-specifiers> <initialized-declarator-list> ;

<declaration-list> ::= <declaration> | <declaration-list> <declaration>

<declaration-specifiers> ::= <type-specifier>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<digit-sequence> ::= <digit> | <digit> <digit-sequence>

<equality-op> ::=  == | !=

<equality-expression> ::= <relational-expression> |
                          <equality-expression><equality-op><relational-expression>

<expression> ::= <assignment-expression>

<expression-statement> ::= <expression> ;
```

```
<floating-constant> ::= <digit-sequence> . <digit-sequence>

<floating-type-specifier> ::= float

<following-character> ::= <letter> | <digit>

<identifier> ::= <letter> | <identifier> <following-character>

<if-statement> ::= if ( <conditional-expression> ) <statement>

<if-else-statement> ::= if ( <conditional-expression> ) <statement> else <statement>

<initialized-declarator-list> ::= <identifier> |
                                  <initialized-declarator-list> , <identifier>

<integer-constant> ::= <digit> | <integer-constant> <digit>

<integer-type-specifier> ::= int

<letter> ::= a | b | ... | z | A | B | ... | Z

<logical-and-expression> ::= <equality-expression> |
                             <logical-and-expression> && <equality-expression>

<logical-or-expression> ::= <logical-and-expression> |
                            <logical-or-expression> || <logical-and-expression>

<multiplicative-expression> ::= <unary-expression> |
                                <multiplicative-expression><mult-op><unary-expression>

<mult-op> ::= * | / | %

<null-statement> ::= ;

<parenthesized-expression> ::= ( <expression> )

<primary-expression> ::= <identifier> | <constant> | <parenthesized-expression>

<program> ::= void main () <compound-statement>

<relational-expression> ::= <additive-expression> |
                            <relational-expression> <relational-op> <additive-expression>

<relational-op> ::= < | <= | > | >=
<statement> ::= <expression-statement> | <compound-statement> |
                <conditional-statement> | <while-statement> | <null-statement>

<statement-list> ::= <statement> | <statement-list> <statement>

<type-specifier> ::= <floating-type-specifier> | <integer-type-specifier>

<unary-expression> ::= - <primary-expression> | <primary-expression>

<while-statement> ::= while ( <conditional-expression> ) <statement>
```

Visual Studio includes a number of doodads, froufrous, and whatnots that are unimportant for your core task of learning C++.

– Ray Lischner, in *C++: The Programmer's Introduction to C++* (Apress, 2009)