Computer Science 433
Programming Languages
The College of Saint Rose
Fall 2012

Program/Problem Set 4: Tokenizer for Little C

Due: 11:59 PM, Wednesday, October 10, 2012

For this assignment, you will be implementing a tokenizer (*i.e.*, lexical analyzer) in C for a language called *lC* (little C). You will later be using this tokenizer as the first stage in a larger program that will perform a full syntax analysis (*i.e.*, parse) a *lC* program.

The tokenizer, and especially the parser to come, are quite complex programs. As such, you are strongly encouraged to form groups of 2 or 3 for this assignment and the next.

You can find and run the executable for my solution code for this program on `mogul.strose.edu` in `/home/cs433/probsets/tokenizer/` .

The *lC* Language

lC is a smaller, simpler version of the C programming language. Several familiar C features, such as arrays and functions (other than `main`) are not present in *lC*, but since *lC* is a proper subset of C, any *lC* program should compile correctly with a C compiler.

We describe the language with BNF rules (taken from *C: A Reference Manual*, by Harbison and Steele, Jr., 4^{th} Edition, Tartan Inc., 1995, with revisions).

- `[ ]` denotes an optional part (there are no `[ ]` brackets in this language)
- the top level (*i.e.*, root) production of is `<program>`

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

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

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

<assignment-op> ::= =

<compound-statement> ::= { [<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> . <digit-sequence>
           | . <digit-sequence>

<floating-type-specifier> ::= float

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

<for-statement> ::= for <for-expressions> <statement>

<for-expressions> ::= ( <expression> ; <expression> ; <expression> )

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

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

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

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

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

<integer-type-specifier> ::= int

<iterative-statement> ::= <while-statement> | <for-statement>

<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> ::= <primary-expression>
          | <multiplicative-expression><mult-op><primary-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> | <iterative-statement>
          | <null-statement>

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

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

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

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

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

Tokenizer Requirements

Your tasks are

1. Determine from the BNF grammar above which rules should be handled entirely by the tokenizer. All terminal symbols, including keywords, should be handled by the tokenizer. For example, the `<while-statement>` includes three terminals: the `while` keyword, the `(`, and the `)`, which should be tokens. You will also find the parser to be simpler if you also completely handle some other BNF rules (such as `<identifier>` and numeric constants) in the tokenizer.

2. Write a C program `tokenizer.c` that takes as its input a single command-line parameter, the name of a file that contains a *lC* program. It should follow the model of the "front" example from the text and in class in how it scans the input, builds lexemes, and prints out the tokens and lexemes it finds.

3. Develop at least 3 nontrivial *lC* example programs. These programs should compile with your favorite C compiler and should, as a group, test all of the token types needed by the grammar for *lC*.

If you use the "front" example as a guide (or better yet as a starting point), you will find that you need to introduce several new token types and extend the `lex` function significantly. You will also need to add a capability to differentiate between identifiers and keywords and the `lookup` function will need to be expanded to handle multi-character operators.

It does not matter which specific token codes you assign to token types. Just don't reuse any. However, you may find it useful to group them as is done in the "front" example, where token codes that start with 1 are for one category, start with 2 are for operators and punctuation. Perhaps a separate code grouping for keywords would be appropriate.

Remember that your tokenizer need not be concerned with whether a sequence of tokens is valid *lC* code, just whether the tokens themselves are valid and what they are. For example, if your input consist of

```
if } ( + * 23.4 while float ;;;;
```

this would be perfectly fine with the tokenizer. The parser will certainly not be happy, though (when we get to that part).

A slow and steady approach will be essential here. You will definitely need to ask questions. You will definitely need to discuss your approach with your partner(s). No one piece is huge, though, so tackle it one step at a time and keep making progress.

General Requirements

Your code should be commented appropriately throughout. Please also include a longer comment at the top of your program describing your implementation. And, of course, it should include your name(s).

Your program should compile without warnings using `gcc` on mogul when the `-Wall` flag is included. This flag turns on extra warnings that will help you avoid some of the pitfalls of C programming. If you encounter any warnings that you don't know how to fix, ask!

Include a `Makefile` that compiles the program with the `-Wall` flag. This `Makefile` should produce an executable program called `tokenizer`. My `Makefile` is on `mogul.strose.edu` in `/home/cs433/probsets/tokenizer/` . Please feel free to use or modify as you see fit.

Submission

To submit this assignment, send all necessary files (your C source file, your example *lC* programs, and your `Makefile`) as attachments to *terescoj@strose.edu* by 11:59 PM, Wednesday, October 10, 2012.

Please include a meaningful subject line (something like "CS433 Program/Problem Set 4 Submission"). Please do not include any additional files, such as emacs backup files, object files, or executable programs.

Grading

This lab will be graded out of 50 points.

Grading Breakdown	
One-character operators and punctuation	5 points
Multi-character operators	7 points
Integer literals	5 points
Floating point literals	8 points
Keywords	5 points
Correct lexeme for each	3 points
Command-line parameter for file name	1 point
Appropriate output format	2 points
Program documentation	4 points
Program efficiency, style, and elegance	3 points
Working `Makefile`	1 point
Example *lC* programs	6 points