SIENA*college*
Computer Science

Computer Science 340
Programming Languages
Siena College
Fall 2023

Problem Set 3: Tokenizer for Little C

Due: 11:59 PM, Thursday, October 12, 2023

For this problem set, 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. Even though the quantity of code you will write is only on the order of a couple hundred lines, it will take some thought and planning. As such, you are 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 `noreaster.teresco.org` in `/home/cs340/probsets/tokenizer/`.

Commit and push often, and use meaningful commit messages. It's only a small part of the grade, but it's The Right Thing To Do.

Getting Set Up

In Canvas, you will find a link to follow to set up your GitHub repository, which will be named `tokenizer-probset-yourgitname`, for this problem set. Only one member of the group should follow the link to set up the repository on GitHub, then others should request a link to be granted write access.

All GitHub repositories must be created with all group members having write access and all group member names specified in the `README.md` file by 11:59 PM, Friday, October 6, 2023. This applies to those who choose to work alone as well!

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, or starting) production of the language is `<program>`

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

<additive-expression> ::= <multiplicative-expression> [ <add-op> <additive-expression> ]

<assignment-expression> ::= IDENT = <conditional-expression>

<compound-statement> ::= { [<declaration-list>] [<statement-list>] }

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

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

<constant> ::= INT_LIT | FLOAT_LIT

<declaration> ::= <type-specifier> <initialized-declarator-list> ;

<declaration-list> ::= <declaration> [ <declaration-list> ]

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

<equality-expression> ::= <relational-expression> [ <equality-op> <equality-expression> ]

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

<floating-type-specifier> ::= float

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

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

<initialized-declarator-list> ::= IDENT [, <initialized-declarator-list> ]

<integer-type-specifier> ::= int

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

<logical-and-expression> ::= <equality-expression> [ && <logical-and-expression> ]

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

<multiplicative-expression> ::= <primary-expression> [ <mult-op> <multiplicative-expression> ]

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

<null-statement> ::= ;

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

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

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

<relational-expression> ::= <additive-expression> [ <relational-op> <relational-expression> ]

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

<statement> ::= <expression-statement> | <compound-statement>
           | <conditional-statement> | <iterative-statement>
           | <null-statement>

<statement-list> ::= <statement> [ <statement-list> ]

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

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

In the above, you will identify several types of operators and other punctuation, as well as several keywords. Your tokenizer should match each of these with a unique token.

There are three token types which can match a variety of lexemes: `IDENT`, `INT_LIT`, and `FLOAT_LIT`. An `IDENT` is a lexeme that begins with a letter or underscore, and is followed by 0 or more letters, numbers, and underscores. An `INT_LIT` consists exclusively of a sequence of numbers. A `FLOAT_LIT` consists of a sequence of 0 or more digits, followed by a decimal point, followed by a sequence of 0 or more digits, with the restriction that there must be at least one digit before or after the decimal point. Note that we specifically disallow negative `INT_LIT` and `FLOAT_LIT` values.

Tokenizer Requirements

Your tasks are

1. 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 `sebesta-lex` example from the text and in class in how it scans the input, builds lexemes, and prints out the tokens and lexemes it finds.
2. 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*, and all major forms of the tokens whose lexemes vary (`IDENT`, `INT_LIT`, `FLOAT_LIT`).

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 consists of the sentence

```
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 noreaster 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 `noreaster.teresco.org` in `/home/cs340/probsets/tokenizer/` . Please feel free to use or modify as you see fit.

Bonus Opportunities

You can earn up to 12 points of bonus credit for handling the following (2 points each):

- negative `INT_LIT` values
- negative `FLOAT_LIT` values
- octal `INT_LIT` values
- hexadecimal `INT_LIT` values
- e notation `INT_LIT` values
- e notation `FLOAT_LIT` values

Submission

Commit and push!

Grading

This assignment will be graded out of 120 points.

Feature	Value	Score
New character classes and token codes	6	
Match new one-character operators and punctuation	8	
Match multi-character operators	10	
Match integer literals	5	
Match floating point literals	8	
Match keywords as appropriate tokens	10	
Report correct lexemes	7	
Command-line parameter for file name with error check	2	
Appropriate output format	5	
Git commit frequency and message quality	10	
Program documentation	15	
Program efficiency, style, and elegance	7	
Working `Makefile`	1	
3 valid example `lC` programs	6	
Programs cover all token types	10	
Programs cover all major cases for `IDENT`, `INT_LIT`, `FLOAT_LIT`	10	
Bonus opportunity	0	
Total	120	