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

Program/Problem Set 5: Parser for Little C

Due: 11:59 PM, Tuesday, October 23, 2012

For this assignment, you will be implementing a parser for a further variant of the *lC* (little C) language you used in the previous assignment. You will use the tokenizer you have developed as the first stage in this larger program that will perform a full syntax analysis of (*i.e.*, parse) a *lC* program.

A parser is a very complex program. As such, you are strongly encouraged to form groups of 2 or 3 again for this assignment. You need not maintain the same groups you had for the tokenizer unless you wish to do so.

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

Simplified *lC* Language

The following is a simplified version of the BNF rules for *lC*, modified to facilitate its parsing with a recursive descent procedure. It assumes that numeric constants have been tokenized to a terminal called `INT_LIT` for integers, `FLOAT_LIT` for floating point. Also, identifiers have been tokenized to a terminal called `IDENT`.

Left recursion has been removed, and a handful of problematic constructs have been removed.

None of the changes made should change anything in your tokenizer code.

As before,

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

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

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

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

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

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

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

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

<floating-type-specifier> ::= float

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

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

<initialized-declarator-list> ::= <identifier> [, <initialized-declarator-li

<integer-type-specifier> ::= int

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

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

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

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

<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> <relatio

<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>
```

Parser Requirements

Your tasks are

1. Determine from the BNF grammar above which rules need to "make choices" and how they will make those choices. That is, those rules that have two or more options on their right hand side, how will your parser know which rule to apply. For this, you will need to determine what the "first" token set is for each choice.

   For example, the `<iterative-statement>` rule can be either a `<while-statement>` or a `<for-statement>` We can readily determine which of these to apply. If the next token is the `while` keyword, we have encountered a `while` statement, a `for` keyword indicates a `for` statement, and any other token means there is an error.

2. Write a C program `parser.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 improved "recdescent" example in how it scans the initializes the lexer, and calls the start nonterminal (in this case, `program`).

To get started, combine your *lC* tokenizer code with the basic framework found in the "recdescent" example. You will need to replace the functions `expr()`, `term()`, and `factor()` with functions for all of the nonterminals in the *lC* grammar. You should name the functions the same as the nonterminals, but replace dashes with underscores.

You will have a function for each nonterminal in the grammar. Some are quite short, others have more work to do. In most cases, it will be clear what you need to do from looking at the BNF rule and the "first" tokens that will cause a particular rule to be applied. The trickiest rule might be the `<statement-list>`, which consists of a statement, possibly followed by another `<statement-list>`. In this case, you need to look at the BNF rule that produces a `<statement-list>` to determine when we need to call it again, and when the `<statement-list>` should end.

The output of your program should be primarily through the provided `match`, `entryMsg` and `exitMsg` functions. Any time a function in your parser has determined that part of a rule "matches" a token on the input, call `match` with the current function name (*i.e.*, the name of the BNF rule currently being applied), and an appropriately indented message about the token matched will be printed. This, combined with calls to `entryMsg` and `exitMsg` will result in the "parse tree"-like format of the output.

When you encounter a parse error, call the `error` function with an appropriate message. The messages in my version are short and probably not that helpful in many circumstances. If you get the parser working and still have time, see if you can improve on these messages.

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 `parser`. My `Makefile` is on `mogul.strose.edu` in `/home/cs433/probsets/parser/` . Please feel free to use or modify as you see fit.

Submission

To submit this assignment, send all necessary files (your C source file and your `Makefile`) as attachments to *terescoj@strose.edu* by 11:59 PM, Tuesday, October 23, 2012.

Please include a meaningful subject line (something like "CS433 Program/Problem Set 5 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 75 points.

Grading Breakdown	
Basic recursive descent parser organization	5 points
`add_op` function	1 point
`additive_expression` function	1 point
`assignment_expression` function	2 points
`compound_statement` function	3 points
`conditional_expression` function	$\frac{1}{2}$ point
`conditional_statement` function	3 points
`constant` function	2 points
`declaration_list` function	2 points
`declaration` function	2 points
`equality_expression` function	1 point
`equality_op` function	1 point
`expression_statement` function	2 points
`floating_type_specifier` function	$\frac{1}{2}$ point
`for_expressions` function	3 points
`for_statement` function	2 points
`initialized_declarator_list` function	3 points
`integer_type_specifier` function	$\frac{1}{2}$ point
`iterative_statement` function	2 points
`logical_and_expression` function	1 point
`logical_or_expression` function	1 point
`mult_op` function	1 point
`multiplicative_expression` function	1 point
`null_statement` function	1 point
`parenthesized_expression` function	2 points
`primary_expression` function	3 points
`program` function	3 points
`relational_expression` function	2 points
`relational_op` function	1 point
`statement_list` function	$2\frac{1}{2}$ points
`statement` function	3 points
`type_specifier` function	2 points
`while_statement` function	3 points
Command-line parameter for file name	1 point
Appropriate output format	3 points
Program documentation	4 points
Program efficiency, style, and elegance	3 points
Working `Makefile`	1 point