Concepts of Programming Languages, CSCI 305, Fall 2014
LL Parser for MiniGo14, Due Friday, Dec. 5 by midnight

Updated Dec. 2, 2014

Write an LL parser in a language of your choice. Your parser should call one of the versions of your lexical analyzer to tokenize the input. The lexical analyzer should return a $$ when there are no more tokens left.

The parser does not need to enforce there being a main function.

MiniGo allows arbitrary spaces, tabs and newlines, except within a variable and an opening { cannot appear by itself on a line. Your parser does not need to enforce that { cannot appear by itself on a line. For all test programs, correct and incorrect, no { will appear by itself on a line.

Unary operators will not be used as the lexical analyzer assignment used the single token op for +, - * and /.

Following is an LL version of a grammar for MiniGo14.
Grammar:

program → func_def_list $$

func_def_list → func_def func_def_list | ε
func_def → func id lparen parm_list rparen
(data_type | ε) lbrace stmt_block rbrace

parm_list → parm_pair parm_list_tail | ε
parm_list_tail → comma parm_pair parm_list_tail | ε
parm_pair → id data_type

stmt_block → stmt stmt_block | ε
stmt → var_decl | const_decl | assign_stmt | for_stmt | if_stmt | return_stmt

var_decl → var id_list data_type
const_decl → const id_list data_type
id_list → id id_list_tail
id_list_tail → comma id_list | ε

assign_stmt → id assign expr

for_stmt → for condition (stmt | lbrace statement_block rbrace)
if_stmt → if condition (stmt | lbrace statement_block rbrace) if_tail
if_tail → (else (stmt | lbrace statement_block rbrace)) | ε

condition → expr comp_op expr

return_stmt → return (expr | ε)

expr → term term_tail
term_tail → op term term_tail | ε
term → factor factor_tail
factor_tail → op factor factor_tail | ε
factor → lparen expr rpare | int_lit | float_lit | id factor_rest
factor_rest → lparen arg_list rparen | ε
arg_list → expr arg_list_tail | ε
arg_list_tail → comma expr arg_list_tail | ε
data_type → int | float

The interface to your program should:

- Describe the purpose of the parser and how to use it.
- Allow the user to repeatedly enter a path to a file containing a MiniGo14 program. (If the parser is in the middle of parsing a program, warn the user that the parser is not done.)
- When a legal path is given, and the entire program is parsed correctly, display a message to that effect.
- Display helpful errors when the lexical analyzer finds an error and attempt to continue parsing. Note that lexical analyzers handle errors differently, so parses may end up in different places after a lexical error.
- Display helpful errors when the parser finds an error and attempt to continue parsing (this may cause more error messages as it is hard for the parser to anticipate what the program is meant to do).
- Allow the user to exit the program.

For maximum credit:

- Your program should be well commented, which includes descriptive variable, class and method names, and a block of code before every function definition describing the inputs, the outputs and what the function does.
- Your program should be well designed.
- Your program should not include any extra, unused code.

For extra credit, output a parser tree when a program is parsed correctly. This parser tree can be in any format you choose, but be sure to explain the format along with the purpose of the parser.

You cannot use the Unix tool lex or yacc for this assignment.

To submit your program zip it and email it to me.