CS4318/CS5331
Assignment 2: Parser for `mC`
100 Points
Due: Mar 5, Friday 11:59 PM

Objective

Write a parser for `mC` (minimal C) using `yacc`

Description

Your task for this project is to write a parser for `mC` that will parse the token stream generated by your lexical analyzer. The parser will detect syntax errors for programs that do not meet the `mC` grammar specification. For all syntactically correct programs the parser will construct an abstract syntaxt tree (AST) representation. The AST will be used later by the semantic analyzer and code generator. In addition, your parser will need to build a symbol table for keeping track of *names* within the program. You may choose to augment the string table that you created for your scanner or you can write one from scratch.

Syntax Specification

The grammar for `mC` is attached to this handout. Your first task is to express this grammar using `yacc` specification rules. You will want to run your specification through `yacc` to make sure there are no conflicts in the grammar. If there are conflicts, you will need to eliminate them by rewriting the specifications without changing the meaning of the grammar.

Abstract Syntaxt Tree

The AST is a tree representation of the syntax of the program. The leaf nodes in this tree can be an identifier, integer constant or a character constant. For each leaf node you will need to store the corresponding semantic value (i.e., name for identifier, numeric value for integer constant, character value for character constant). The internal nodes will generally correspond to some non-terminal in the grammar.

Symbol Table

At this phase the symbol table should contain three types of information for each identifier : `name`, `type` and `scope`. The `type` information will not be used in this phase. The `name` and `scope` information will be used to detect undeclared variable errors. Like `C`, `mC` supports only two scopes for variables : local and global.

Error Handling

Your parser needs to generate meaningful error messages for all types of syntax errors in `mC`. In generating the error message the parser should attempt to provide the line number where the error occured.

Implementation Instructions

- You can use any flavor of `yacc` to implement your parser
- You need to have a routine that dumps the AST to standard output in a useful way. This will be helpful for debugging and testing.

Submission

Create a README.txt that contains a listing of files required to build your parser. The README should also contain build instructions, special comments and known bug information. For this assignment, you also need to submit a makefile that builds your parser. Your executable should be called `mcc`. Note, you will also need to resubmit your scanner. If you have made corrections to your scanner since the last submission you should submit the newer version.

Create a tar archive called assg2.yourlastname with the README.txt and all files required to build your parser (.l, .y, .h, makefile etc). Submit the tar archive using the drop box on the course web page by the due date.

mC Grammar

program	: *declList*
declList	: *decl*
	| *decl declList*
decl	: *varDecl*
	| *funDecl*
varDecl	: *typeSpecifier* **ID** [**NUM**] ;
	| *typeSpecifier* **ID** ;
typeSpecifier	: **int**
	| **char**
	| **void**
funDecl	: *typeSpecifier* **ID** (*formalDeclList*) *funBody*
formalDeclList	:
	| *formalDecl*
	| *formalDecl* , *formalDeclList*
formalDecl	| *typeSpecifier* **ID**
	| *typeSpecifier* **ID**[ ]
funBody	: { *localDeclList statementList* }
localDeclList	:
	| *varDecl localDeclList*
statementList	:
	| *statement statementList*
statement	: *compoundStmt*
	| *assignStmt*
	| *condStmt*
	| *loopStmt*
	| *returnStmt*
compoundStmt	: { *statementList* }
assignStmt	: *var* = *expression* ;
	| *expression* ;
condStmt	: **if** (*expression*) *statement*
	| **if** (*expression*) *statement* **else** *statement*
loopStmt	: **while** (*expression*) *statement*
returnStmt	: **return** ;
	| **return** *expression* ;

var : **ID**
| **ID** [*addExpr*]
expression : *addExpr*
| *expression relop addExpr*
relop : <=
| <
| >
| >=
| ==
| ! =
addExpr : *term*
| *addExpr addop term*
addop : +
| -
term : *factor*
| *term mulop factor*
mulop : *
| /
factor : (*expression*)
| *var*
| *funcCallExpr*
| **NUM**
| **CHAR**
| **STRING**
funcCallExpr : **ID** (*argList*)
argList :
| *expression*
| *expression* , *argList*