CS4318/CS5331
Assignment 2: Parser for `mC`
100 Points
Due: Friday, Mar 7 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 specifications of the `mC` grammar. For all syntactically correct programs the parser will construct an abstract syntax tree (AST) representation. The AST will be used later by the semantic analyzer and code generator. Your parser will also need to build a symbol table for keeping track of *names* within the program.

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 Syntax Tree (AST)

The AST is a tree representation of the syntax of the input program. You have some flexibility as to how you structure this tree. A reasonable strategy might be to use an *n-ary* tree where the internal nodes correspond to non-terminals in the grammar and the leaf nodes represent terminals (e.g., identifier, integer constants etc). Note, this rule does not need to be enforced strictly. In some cases, it may be convenient to have a node in the tree that is neither a terminal nor a non-terminal. In other situations, it may be helpful to add a child node to a terminal. Whatever your implementation strategy, your AST should contain (or have access to) sufficient information so that by traversing the tree it is possible to reproduce source code that is equivalent to the original source. This implies that for some terminals, you will need to store the corresponding semantic value (i.e., name for identifiers, numeric value for integer constants, ASCII value for character constants).

Symbol Table

At this phase, the symbol table should contain three types of information for each identifier : `name`, `type` and `scope`. Like `C`, `mC` has two kinds of scopes for variables : local and global. Variables declared outside any function are considered globals, whereas variables (and parameters) declared inside a function *foo* are local to *foo*. Note, although you are populating the symbol table at this phase, the information stored within will not be used until the next phase.

Error Handling

In generating error messages your parser should attempt to provide the line number where the error occurred.

Implementation Instructions

- You can use any flavor of `yacc/bison` 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.
- Like Assignment 1, you need to have a separate driver file where you call `yyparse()`.

Extra Credit

`mC` is a subset of `C` and therefore does not contain many of the features available in `C`. Consider extending the `mC` grammar to support one addtional feature. The additional feature does not necessarily have to be something that comes from `C`, it can be a feature available in any high-level language or something *you* think would be useful to have in a language.

In this phase all you need to do is specify the syntax for the new feature using `yacc` rules. You would however want to see it all the way through to code generation. Adding of the new feature should not affects other parts of the `mC` language and you should attempt the extra credit only after you have the rest of the parser fully functional. Be sure to clearly document in your README what feature(s) you added.

You can earn upto 10 extra credit points on this assignment.

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 code as part of your parser. If

you have made corrections to your scanner since the last submission you should submit the newer version.

Create a tar archive called assg2.lastname.tar.gz 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*
	| *declList decl*
decl	: *varDecl*
	| *funDecl*
varDecl	: *typeSpecifier* **ID** [**NUM**] ;
	| *typeSpecifier* **ID** ;
typeSpecifier	: **int**
	| **char**
	| **void**
funDecl	: *typeSpecifier* **ID** (*formalDeclList*) *funBody*
	| *typeSpecifier* **ID** ( ) *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*)
| **ID** ( )
argList | *expression*
| *argList* , *expression*