Purpose

This project is intended to give you experience in using a scanner generator (Lex), a parser generator (YACC), writing a syntax specification (grammar) for a language, performing parsing and semantic analysis (attribute grammar), and practicing error handling in a compiler.

Summary

Your task is to write an interpreter/translator for a simple calculator whose programming language, CALC, contains basic language constructs such as variables and assignment statements. The interpreter/translator will be written using a compiler generator (YACC). Your program should perform the functions of both the interpreter and translator. The program should call the lexical analyzer (Lex) for the next token, parse the token stream, report grammatical errors, perform static and dynamic semantic checks, interpret the program statement-by-statement (including print statements which produce outputs), and translate the CALC program into an equivalent working C++ program called mya.cpp.

Lexical Specification

The table below defines the tokens that must be recognized, with their associated symbolic names. Comments are enclosed in (* ... *), and cannot be nested. An identifier is a sequence of (at least one) upper or lower case letters followed by zero, one or more digits. CALC is case insensitive. There is no limit on the length of identifiers. However, you may impose limits on the total number of distinct identifiers and string lexemes and on the total number of characters in all distinct identifiers and strings taken together. There should be no other limitation on the number of lexemes that the lexical analyzer will process. An integer constant is an unsigned sequence of digits representing a 10-based number.
<table>
<thead>
<tr>
<th>Token</th>
<th>Symbolic Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>SEMInumber</td>
</tr>
<tr>
<td>(</td>
<td>LPARENnumber</td>
</tr>
<tr>
<td>integer constant</td>
<td>ICONSTnumber</td>
</tr>
<tr>
<td>begin</td>
<td>BEGINnumber</td>
</tr>
<tr>
<td>program</td>
<td>PROGRAMnumber</td>
</tr>
<tr>
<td>-</td>
<td>MINUSnumber</td>
</tr>
<tr>
<td>*</td>
<td>TIMESnumber</td>
</tr>
<tr>
<td>var</td>
<td>VARnumber</td>
</tr>
<tr>
<td>integer</td>
<td>INTnumber</td>
</tr>
<tr>
<td>end of file</td>
<td>EOFnumber</td>
</tr>
<tr>
<td>,</td>
<td>COMMAnumber</td>
</tr>
<tr>
<td>)</td>
<td>RPARENnumber</td>
</tr>
<tr>
<td>identifier</td>
<td>IDnumber</td>
</tr>
<tr>
<td>end</td>
<td>ENDnumber</td>
</tr>
<tr>
<td>is</td>
<td>ISnumber</td>
</tr>
<tr>
<td>+</td>
<td>PLUSnumber</td>
</tr>
<tr>
<td>div</td>
<td>DIVnumber</td>
</tr>
<tr>
<td>print</td>
<td>PRINTnumber</td>
</tr>
<tr>
<td>=</td>
<td>EQnumber</td>
</tr>
</tbody>
</table>

Token attributes

A unique identification of each token must be returned by the lexical analyzer. In addition, the lexical analyzer must pass extra information for identifier and integer constant tokens to the parser. For identifier and integer constant tokens, the extra information is passed to the parser as a single value of the integer type, through a global variable as described below (yylval.semantic value). For integer constants, the numeric value of the constant is passed. In order to allow other phases of the compiler to access the original identifier lexeme, the lexical analyzer passes an integer uniquely identifying the identifier. The unique identifying number for identifiers should be an index into a string table created by the lexical analyzer to record the lexemes.

Implementation

The central routine of the scanner is yylex, an integer function that returns a token number, indicating the type (identifier, integer constant, semicolon, etc...), of the next token in the input stream. In addition to the token type, yylex must set the global variables yyline and yycolumn to the line and column number at which that token appears (hint: modify yyline when you see a newline character and modify yycolumn when you see anything else), and in the case of identifiers and constants, put the extra information needed, as described above, into a global integer variable yylval.semantic value. Lex will write yylex for you, using the patterns and rules defined in your lex input file (which should be called lexer.l). Your rules must include the code to maintain yyline, yycolumn and yylval. You are to write a routine Lex error that takes a message and line and column numbers and reports an error, printing the message and indicating the position of the error. You need only print the line and column...
number to indicate the position. The #define mechanism should be used to allow the lexical analyzer to return token numbers symbolically. In order to avoid using token names that are reserved or significant in C/C++ or in the parser, the token names have been specified for you in the subsequent table. The parser and the lexical analyzer must agree on the token numbers to ensure correct communication between them. The token number can be chosen by you, as the compiler writer, or, by default, by Yacc. Regardless of how token numbers are chosen, the end-marker must have token number 0 or negative, and thus, your lexical analyzer must return a 0 (or a negative) as token number upon reaching the end of input. Your lexical analyzer should recover from all malformed lexemes, as well as such things as string constants that extended across a line boundary or comments that are never terminated.

Syntax Specification

The syntax of the CALC language is described by a set of syntax diagrams in syntax.pdf. A syntax diagram is a directed graph with one entry and one exit. Each path through the graph defines an allowable sequence of symbols. For example, the structure of a valid CALC program is defined by the first syntax diagram. The occurrence of the name of another diagram such as declaration and compound statement indicates that any sequence of symbols defined by the other diagram may occur at that point. The following is an example valid CALC program.

```plaintext
program xyz is
begin
var a, b;
a = 2;
b = 3;
var c;
c = a + b;
print c
end
```

Your first task in this assignment is to develop a context free grammar for the CALC language from the syntax diagrams.

YACC Specification

Your second task in this assignment is to express your grammar as a YACC specification. You will want to run your specification through YACC to ensure that the grammar produces no parsing conflicts (compiled with yacc -v). If conflicts are indicated by YACC, you should alter your grammar to eliminate them without changing the language accepted by your grammar, or ensure that YACC’s handling of the conflict agrees with the CALC language specification. It is suggested that you first develop a parser that merely prints out ACCEPT
for syntactically correct CALC programs and REJECT with error messages for incorrectly structured CALC programs, calling your lexical analyzer for the tokens. After you have guaranteed that your YACC specification is syntactically sound, you will extend your YACC grammar by adding attributes and semantic rules with actions to develop an interpreter/translator for CALC that does various static and dynamic semantics checks, performs the calculation specified in the CALC program, and translates the CALC program into a C++ program. Besides reporting grammatical errors, the interpreter/translator also performs the following semantics checks:

- Duplicated declaration when a variable is declared multiple times.
- Undeclared variable when a variable is used in the program, but not declared or before it is declared.
- Uninitialized variable when a variable is referenced before initialized.
- Divided by 0 error when the denominator in a division is 0.

The program can exit after reporting one semantic/syntax error. If there is no error, the program should (1) execute each statement and output the result of the expression in each print statement (interpreter function), and (2) produce a semantically equivalent C++ program called mya.cpp with equivalent expressions/assignments (translator function).

To facilitate semantic checks and program interpretation and translation, a symbol table must be created to store variables and the related information. The symbol table will need to have at least three fields: the name of the variable, the value of the variable (integer type only in CALC), and a flag indicating whether the variable has been initialized. When the parser encounters an identifier in the declaration, the identifier must be inserted into the symbol table. When the parser encounters an identifier in an expression, it must look up the symbol table to check if the variable has been initialized and obtain its value (if initialized). After the parser processes an assignment statement, the value of the variable in the left hand side of the assignment statement must be updated in the symbol table.

The generated mya.cpp file should have the same number of statements as the source program. Each statement in mya.cpp should have the same expression (may differ only in notation) as the original program. For the example CALC program described, the corresponding C++ program would be similar to the following:

```cpp
#include <iostream>

using namespace std;

main()
{
    int a, b;
    a = 2;
    b = 3;
    int c;
    c = a + b;
    cout << c << "\n";
}
```
Interpreting and translating CALC with YACC is relatively simple. Basically, for every assignment statement, the program first evaluates the value of the expression in the right hand side of the statement. This is similar to the cal trans example we gave in class. After that, the interpreter updates the value of the variable in the left hand side of the statement. When processing a print statement, the interpreter evaluates the value of the expression and prints the result to the standard output. To facilitate translation, each expression can have an string attribute that stores the string representation of the C++ expression.

**Error Recovery**
Your parser should print appropriate error messages. You do not have to implement any error recovery. Your program should have similar behavior as the sample executable provided.

**Assignment Submission**
Submissions are due on March 4, 11:59pm, when you should submit all related files to the Blackboard (including a makefile to produce the executable for the project).

- Recognize correct programs and detect incorrect programs (50).
- Semantic checks and error reporting (20).
- Correct calculation (10).
- Correct translation (15).
- Correct translation with minimal parentheses in expressions (5).
- Extra 10 points for generating correct dynamic divide-by-zero checking code in the target C++ program.
- Extra 3 points for the person who first reports a new error in the sample executable.

Your program will be tested with a series of programs. Some of the testing programs are provided in the project package. If the code cannot handle any input file, 0 points will be given (programs that cannot produce the executable with a make command due to whatever error such as wrong makefile, compiling errors, and missing files automatically get 0 points). If your code can process some programs in the test suite, the grade will be assigned based on the number of programs that your code processes correctly.