Corbin Charpentier

Lab 4 report

Purpose:

For this lab, a source code scanner for a C-like language was implemented and tested on a sample C document, test.c. The program, lexer.l, scanned the test document and, depending on the scanned expression, outputted a particular token to an arbitrary file. These tokes will then be analyzed by a parser to ensure their structure/context is correct. The scanner will quit if bad syntax (not accounted for by a regular expression) is encountered with an error.

Design and Implementation:

Architecturally, the programming consists of a .l and a .h file. The header file contains an associated integer constant for each token that can be generated by lexer.l. These integers begin at 256 and ascend so they are not confused with ASCII characters. In lexer.l, the expressions are identified via a series of regular expression rules. In each of these rules, the token is printed and the integer constant is returned. Because of the return, yylex() must be called each time an expression is captured, hence yytext() is wrapped in a while loop that terminates only when the end of the file is reached.

For each possible C expression, there is an associated token and regular expression in lexer.l. The scanner's job is to convert these expressions into tokens. The tokens can be thought of as an all-caps truncated description of the token. For example, the expression 'a' equates to token CHARVAL, while token “some string” is tokenized by STRVAL.

The top to bottom rule of precedence in *.l files was exploited (the lower the line number, the higher thee precedence). For example, of the program input is “int”, it would be desirable to generate token INT. However, “int” matches the description of an ID. The solution is to place the rule that generates an INT token higher in the file than the rule generating ID. This simple idea was applied to all C keywords (also <, <<, <=, etc...).

Problems:

It took me awhile to figure out the programming was terminating after the first expression it parsed from the test file. Because reach was an explicit return in each regular expression rule, yylex() returned after only the first match. To fix this, the call to yylex() was wrapped in a loop that doesn't terminate until the end of the file is reached. I has also originally missed a few keywords, so the program fell through to the default case and crashed.