Implementation of A Scanner with Lex

Noha Abuaesh

Noha.abuaesh@gmail.com

Introduction

Lex is a popular scanner (lexical analyzer) generator that was developed by M.E. Lesk and E. Schmidt of AT&T Bell Labs. Somer versions of Lex exist, most notably flex (for Fast Lex). The input to Lex is called Lex specification or Lex program and the output is generated as a scanner module in C from a Lex specification file. Scanner module can be compiled and linked with other C/C++ modules.

Problem Specification

In this project, I write a Lex specification and use the Lex generator to construct a lexical analyzer for a language with the following lexical properties:

Lexical Conventions:

The same properties in Appendix A section A.1 of [REF1], with replacing the definition of ID and NUM as follows:

ID = letter (letter | digit)* (“.”|@ | “_” | □□) (letter | digit)+

NUM = digit + | digit+ “.” digit +(((E |e) (+|- |□) digit +)| □□

Also, the characters in the keywords and identifiers can appear as capital or small letters (i.e. no distinction between capital and small. **A** is like **a** and **if** is like **IF** and **If**).

Error Handling

The following errors are handled:

1. Reaching the end of file while a comment is not closed.

2. Finding a character which is not in the alphabet of the language.

3. Exiting from the scanning of a token while not reaching the final state. For example, while recognizing an identifier, you find "*" after "-" (aaa_*) , or while recognizing a number you find a letter after "e" (2.3eX).

4. Do not handle the problem of having a number followed by an identifier without leavening spaces and similar problems. This will be handled by the parser. For example your scanner will produce as an output **num id** for the string "12ab" because this string, "12 ab", is still erroneous even if you leave spaces.

Error messages indicate the line number and character position where the error occurs.

Output

The output is a list of tokens, and lexemes and error messages appearing in the proper place indicating the line at which the error has occurred.

Procedure

Even though the steps to create a scanner using lex are quite easy and straight forward but I will list the steps I followed to do so anyways:

1. Working on an MS Windows platform, the first step was to install a cygwin shell to get lex running. However, I used bison and flex executables with other scripts that made it possible to run flex on Windows as well, refer to [REF2] for detailed steps.
2. Once lex was ready to run on my machine, I wrote a lex file with the language specification given in the previous section. The resulting file was named a1.l
3. I compiled my lex file a1.l in the lex compiler using the cygwin shell/DOS command:
   flex scanner.l
   The result of this command is a C file generated by Lex named lex.yy.c and saved in the root directory(cygwin home)
4. I compiled lex.yy.c using C compiler and saved the generated executable as a file, by writing:
   g++ lex.yy.c –lfl –o output
   By that the file output.exe is created in the root directory.
5. After running the generated executable by typing in the command below and specifying the input file to be tokenized:
   .\output.exe in.txt
   we can get the final results as a series of tokens and a list of syntax errors. <- There is a bug in here: FIX IT IF YOU CAN

Results and Conclusion

The expected results as mentioned before are the tokens and lexemes that appear in the input file in.txt in addition to a list of any violations to the syntax rules specified by the lex file we wrote.

If you get warnings of unrecognized rules when you compile the file using lex, and may be even worse, errors of undefined variables lex.yy.c is compiled by the C compiler, then most probably there is a mixing between C and Lex codes in different sections of the Lex file. That is, writing C code where lex is expected and vice versa. Try removing the regular expression entries that contain complicated C code, to keep track of the detected errors, perhaps then the file will compile and run successfully with valid output.

Appendix A shows the source code of my lex file scanner.l and the generated output.

References

[REF1] Kenneth Louden, Compiler Construction: Principles and Practice, Course Technology.

[REF2] Jais Antony blog article about: Lex and YACC (Bison) in Windows, March 16, 2008: http://jaisantonyk.wordpress.com/2008/03/16/lex-and-yaccbison-in-windows/

Appendix A

A1.l

```
%{

#include <stdio.h>

//global static int line = 1;

//global static int position = 1;

//global int error_count =0;

//int errors_list[100][3] ; //type, line , character position

//static bool unclosed_comment_err = false;

const char* errors_catalog[] ={

        "Unclosed comment encountered. Expected end of comment, found end of file.",

        "Nested comments are not allowed.",

        "Undefined symbol. Use only the characters defined by the language.",

        "Wrong identifier. Identifiers must start with a letter and end with a letter or a digit.",
```

```
                    "Wrong number. Numbers must end with a digit."
                    };
static int line = 1;
static int position = 1;
static int error_count =0;
static int error_list[100][3] ;
static bool unclosed_comment_err = false;
%}
%%
[eE][lL][sS][eE]            printf("ELSE");
[iI][fF]                    printf("IF");
[iI][nN][tT]                printf("INT");
[rR][eE][tT][uU][rR][nN]    printf("RETURN");
[vV][oO][iI][dD]            printf("VOID");
[wW][hH][iI][lL][eE]        printf("WHILE");
\+                          printf("ADDOP: ADDITION");
\-                          printf("ADDOP: SUBTRACTION");
\*                          printf("MULOP: MULTIPLICATION");
\/                          printf("MULOP: DIVISION");
\<                          printf("RELOP: LESS THAN");
\<=                         printf("RELOP: LESS OR EQUAL");
\>                          printf("RELOP: GREATER THAN");
\>=                         printf("RELOP: GREATER OR EQUAL");
==                          printf("RELOP: EQUAL");
!=                          printf("RELOP: NOT EQUAL");
```

```
=                                   printf("ASSIGNMENT OPERATOR");

;                                   printf("SEMICOLON ");

\,                                  printf("COMMA");

\(                                  printf("OPEN PARANTHESIS");

\)                                  printf("CLOSE PARANTHESIS");

\[                                  printf("OPEN BRACKET");

\]                                  printf("CLOSE BRACKET");

\{                                  printf("OPEN BRACE");

\}                                  printf("END BRACE");

"\/\*"([^"\*\/"])*"\*\/"            printf("COMMENT");

"\/\*"[^\*]*"\/\*"([^"\*\/"])*"\*\/"                printf("NESTED COMMENTS");
        error_list[error_count][0]=1;  error_list[error_count][1]=line;
        error_list[error_count][2]=position; error_count++;

\/\*([^"\*\/"])*            printf("UNCLOSED COMMENT"); printf("UNCLOSED COMMENT");
        error_list[error_count][0]=0;  error_list[error_count][1]=line;
        error_list[error_count][2]=position; error_count++;

\n                                  line++; position=1;

[^\n]                               position ++;

[ \t]+                              {//ignore whitespace};

[^symbol]                           error_list[error_count][0] =  2; error_list[error_count][1]=line;
        error_list[error_count][2] = position; error_count++;

%%

 int main(int argc, char **argv) {

     ++argv; --argc;                 /* skip over program name */

     if (argc > 0) { yyin = fopen(argv[0], "r");  } else { yyin = stdin; }

     yylex();

  }
```

Output

Not Available due to errors.