EXPERIMENT - II
LEXICAL ANALYSER USING LEX TOOL

November 26, 2020

ADITHYA D RAJAGOPAL
ROLL NO : 9
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING TRIVANDRUM

AIM

To implement a lexical analyzer using Lex Tool.

THEORY

LexTool

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed. The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.
The general format of Lex source is:

```
Definitions
Rules
User Subroutines
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus %% (no definitions, no rules) which translates into a program which copies the input to the output unchanged. In the outline of Lex programs shown above, the rules represent the user's control decisions; they are a table, in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer printf("found keyword INT");
```

to look for the string integers in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function printf is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change the number of words from British to American spelling. Lex rules such as

```
     colour  printf("color");
mechanise  printf("mechanize");
     petrol  printf("gas");
```

would be a start. These rules are not quite enough, since the word petroleum would become gaseum;

Metacharacter	**Matches**
.	Any character except newline
\n	Newline
*	Zero or more copies of the preceding expression
+	One or more copies of the preceding expression
?	Zero or one copies of the preceding expression
^	Beginning of line
$	End of line
a|b	a or b
(ab)+	One or more copies of ab
"a+b"	Literal "a+b"
[]	Character class

Table 1: Pattern Matching Primitives

Expressions	**Matches**
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	One of a,b,c
[a-z]	Any letter from a to z
[a-z]	One of a,z
[A-Za-z0-9]	One or more alphanumeric characters
[\b\n]	White space
[^ab]	Anything except a,b
[a\b]	One of a,\,b
[a|b]	One of a,b,|
a|b	One of a,b

Table 2: Pattern Matching Examples

Name	**Function**
int yylex(void)	Call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yyval	value associated with token
int yywrap(void)	Wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN	condition switch start condition
ECHO	write matched string

Table 3: Lex predefined variables

ALGORITHM

Algorithm 1 Algorithm for Lexical Analyser using Lex Tool

```
1: Lex program contains three sections: definitions, rules, and user subroutines. Each
   section must be separated from the others by a line containing only the delimiter, %%.
   The format is as follows:
2:        Definitions
3:              %%
4:        Rules
5:              %%
6:        User_Subroutines
7: In definition section, the variables make up the left column, and their definitions make up
   the right column. Any C statements should be enclosed in %..%. Identifier is defined such
   that the first letter of an identifier is alphabet and remaining letters are alphanumeric.
8: In rules section, the left column contains the pattern to be recognized in an input file to
   yylex(). The right column contains the C program fragment executed when that pattern is
   recognized. The various patterns are keywords, operators, new line character, number,
   string, identifier, beginning and end of block, comment statements, preprocessor directive
   statements etc.
9: Each pattern may have a corresponding action, that is, a fragment of C source code to
   execute when the pattern is matched.
10: When yylex() matches a string in the input stream, it copies the matched text to an
    external character array, yytext, before it executes any actions in the rules section.
11: In user subroutine section, main routine calls yylex(). yywrap() is used to get more input.
12: The lex command uses the rules and actions contained in file to generate a program,
    lex.yy.c, which can be compiled with the cc command. That program can then receive
    input, break the input into the logical pieces defined by the rules in file, and run program
    fragments contained in the actions in file.
```

SOURCE CODE

p2.h

```
#define IDENTIFIER      1
#define ASSIGN_OP       2
#define KEYWORD         3
#define LT              4
#define GT              5
#define GTE             6
#define LTE             7
#define DIGIT           8
#define SEMICOLON       9
#define ADD_OP          10
#define SUB_OP          11
#define MUL_OP          12
#define DIV_OP          13
#define EQU_OP          14
#define LITERAL         15
#define DOT             16
#define ERROR           99
```

p2.l

```
%{
  #include <stdio.h>
  #include <stdlib.h>
  #include "p2.h"
%}

letter [a-zA-Z]
digit [0-9]
id {letter}*|({letter}{digit})+
notid ({digit}{letter})+

%%
```

```
prog|integer|begin|read|if|then|endif|while|do|endwhile|write|end return KEYWORD;
{id} return IDENTIFIER;
":=" return ASSIGN_OP;
"<" return LT;
">" return GT;
">=" return GTE;
"<=" return LTE;
[1-9][0-9]* return DIGIT;
";" return SEMICOLON;
"+" return ADD_OP;
"-" return SUB_OP;
"*" return MUL_OP;
"/" return DIV_OP;
"=" return EQU_OP;
[{,},(,)] return LITERAL;
"." return DOT;
{notid} {printf("Invalid identifier %s in line %d\n",yytext,yylineno);exit(0);}
[\n] {yylineno++;}
[\t] ;

%%

int yywrap(void)
{
 return 1;
}
```

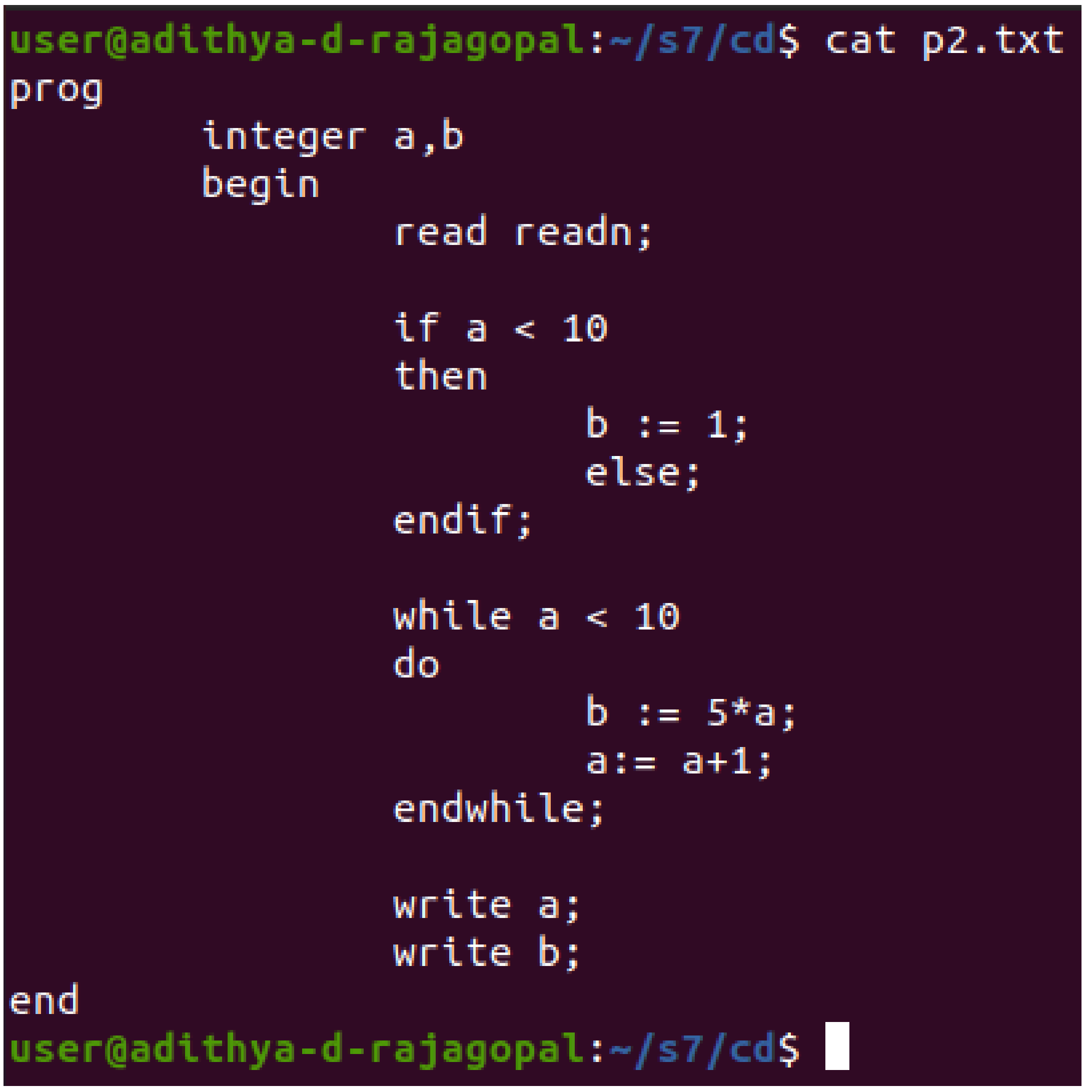
p2.c

```
#include <stdio.h>
#include "p2.h"

extern int yylex();
extern int yylineno;
extern char* yytext;
```

```
void main()
{
 int token;
 token=yylex();
 while(token)
 {
  printf("Token : %d\tValue : %s\n",ntoken,yytext);
  token=yylex();
 }
}
```

SAMPLE INPUT



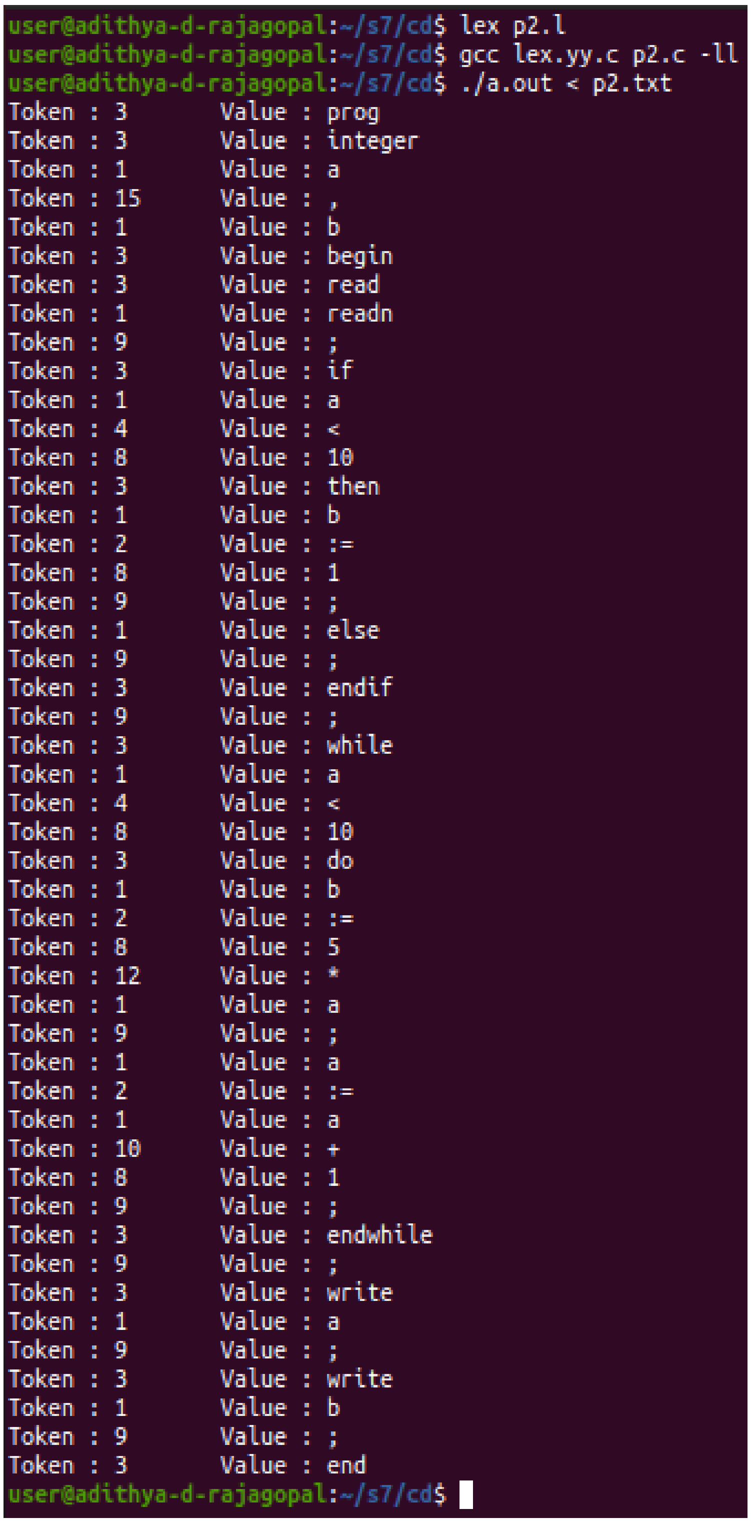
```
user@adithya-d-rajagopal:~/s7/cd$ cat p2.txt
prog
        integer a,b
        begin
                read readn;

                if a < 10
                then
                        b := 1;
                        else;
                endif;

                while a < 10
                do
                        b := 5*a;
                        a:= a+1;
                endwhile;

                write a;
                write b;
end
user@adithya-d-rajagopal:~/s7/cd$ 
```

Sample Output



```
user@adithya-d-rajagopal:~/s7/cd$ lex p2.l
user@adithya-d-rajagopal:~/s7/cd$ gcc lex.yy.c p2.c -ll
user@adithya-d-rajagopal:~/s7/cd$ ./a.out < p2.txt
Token : 3       Value : prog
Token : 3       Value : integer
Token : 1       Value : a
Token : 15      Value : ,
Token : 1       Value : b
Token : 3       Value : begin
Token : 3       Value : read
Token : 1       Value : readn
Token : 9       Value : ;
Token : 3       Value : if
Token : 1       Value : a
Token : 4       Value : <
Token : 8       Value : 10
Token : 3       Value : then
Token : 1       Value : b
Token : 2       Value : :=
Token : 8       Value : 1
Token : 9       Value : ;
Token : 1       Value : else
Token : 9       Value : ;
Token : 3       Value : endif
Token : 9       Value : ;
Token : 3       Value : while
Token : 1       Value : a
Token : 4       Value : <
Token : 8       Value : 10
Token : 3       Value : do
Token : 1       Value : b
Token : 2       Value : :=
Token : 8       Value : 5
Token : 12      Value : *
Token : 1       Value : a
Token : 9       Value : ;
Token : 1       Value : a
Token : 2       Value : :=
Token : 1       Value : a
Token : 10      Value : +
Token : 8       Value : 1
Token : 9       Value : ;
Token : 3       Value : endwhile
Token : 9       Value : ;
Token : 3       Value : write
Token : 1       Value : a
Token : 9       Value : ;
Token : 3       Value : write
Token : 1       Value : b
Token : 9       Value : ;
Token : 3       Value : end
user@adithya-d-rajagopal:~/s7/cd$
```

RESULT

A lexical analyzer using Lex Tool has been designed and implemented using Python and the outputs were verified.