Fr. Conceicao Rodrigues College of Engineering
Department of Computer Engineering

Academic Term : Jan-May 2024 - 25

Class   : T.E. (Computer - A)
Subject Name : System Programming and Compiler Construction
Subject Code : (CPC601)

Practical No:	5
Title:	Lex and Yaac
Date of Performance:	18/03/2025
Date of Submission:	24/03/2025
Roll No:	9913
Name of the Student:	Mark Lopes

Evaluation:

Sr. No	Rubric	Grade
1	**Time Line (2)**	
2	**Output(3)**	
3	**Code optimization (2)**	
4	**Postlab (3)**	

Signature of the Teacher   :

FR. CONCEICAO RODRIGUES COLLEGE OF ENGINEERING

System Programming and Compiler Construction

Experiment No 5

Aim : Study of Lexical analyzer tool -Flex/Lex

Learning Objective: Recognise lexical pattern from given input file.

Theory:

Lex is a program generator designed for lexical processing of character input streams. Itaccepts a high-level, problem oriented specification for character string matching, andproduces 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 theinput stream into strings matching the expressions. At the boundaries between stringsprogram sections provided by the user are executed. The Lex source file associates theregular expressions and the program fragments. As each expression appears in the input tothe program written by Lex, the corresponding fragment is executed.The user supplies the additional code beyond expression matching needed to complete histasks, possibly including code written by other generators. The program that recognizesthe expressions is generated in the general purpose programming language employed for theuser's program fragments. Thus, a high level expression language is provided to write thestring expressions to be matched while the user's freedom to write actions isunimpaired. This avoids forcing the user who wishes to use a string manipulationlanguage for input analysis to write processing programs in the same and ofteninappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language featurewhich can be added to different programming languages, called ``host languages." Just asgeneral purpose languages can produce code to run on different computer hardware, Lexcan write code in different host languages. The host language is used for the output codegenerated by Lex and also for the program fragments added by the user. Compatible runtimelibraries for the different host languages are also provided. This makes Lex adaptableto different environments and different users. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere where appropriate compilers exist.

Lex turns the user's expressions and actions (called source in this pic) into the hostgeneral-purpose language; the generated program is named yylex. The yylex program willrecognize expressions in a stream (called input in this pic) and perform the specifiedactions for each expression as it is detected.

```
             +-------+
 Source -> |  Lex  |  -> yylex
             +-------+


             +-------+
 Input ->   | yylex | -> Output
             +-------+

      An overview of Lex
```

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

%%

[\t]+$;

is all that is required. The program contains a %% delimiter to mark the beginning of therules, and one rule. This rule contains a regular expression which matches one or moreinstances of the characters blank or tab (written \t for visibility, in accordance with the Clanguage convention) just prior to the end of a line. The brackets indicate the characterclass made of blank and tab; the + indicates ``one or more ...''; and the $ indicates ``end ofline,'' as in QED. No action is specified, so the program generated by Lex (yylex) willignore these characters. Everything else will be copied. To change any remaining string ofblanks or tabs to a single blank, add another rule:

%%

[\t]+$;

[\t]+ printf("");

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase.

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 integer 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

libraryfunction printf is used to print the string. The end of the expression is indicated by the firstblank or tab character. If the action is merely a single C expression, it can just be given on theright side of the line; if it is compound, or takes more than a line, it should be enclosed inbraces.

Implementation Details:

1. Open file in text editor

2. Enter keywords, rules for identifier and constant, operators and relational operators. In

the following format

   a) % {

   Definition of constant /header files

   % }

   b) Regular Expressions

   % %

   Transition rules

   % %

   c) Auxiliary Procedure (main( ) function)

3. Save file with **.l** extension e.g. **Mylex.l**

4. Call lex tool on the terminal e.g. [root@localhost]# lex Mylex.l This lex tool will convert

".l" file into ".c" language code file i.e. **lex.yy.c**

5. Compile the file lex.yy.c e.g. **gcc lex.yy.c** .After compiling the file lex.yy.c, this will

create the output file **a.out**

6. Run the file a.out e.g. **./a.out**

7. Give input on the terminal to the **a.out** file upon processing output will be displayed

Sample Code

```
%{
#include<stdio.h>
int key_word=0;
%}
%%
"include"|"for"|"define"  {key_word++;}
%%
int main()
{
printf("enter the sentence");
yylex();
printf("keyword are: %d\n ",key_word);
}
int yywrap() { return 1; }
```

Example: Program for counting number of vowels and consonant

```
% {

#include <stdio.h>

int vowels = 0;

int consonants = 0;

% }

% %
```

[aeiouAEIOU] vowels++;

[a-zA-Z] consonants++;

[\n] ;

. ;

%%

int main()

{

printf ("This Lex program counts the number of vowels and ");

printf ("consonants in given text.");

printf ("\nEnter the text and terminate it with CTRL-d.\n");

yylex();

printf ("Vowels = %d, consonants = %d.\n", vowels, consonants);

return 0;

}

Output:

#lex alphalex.l

#gcc lex.yy.c

#./a.out

This Lex program counts the number of vowels and consonants in given text.

Enter the text and terminate it with CTRL-d.

Iceream

Vowels =4, consonants =3.

Test Cases:

1. Input integer constant
2. Input special symbols

Conclusion:

Aim :Study of Parser generator tool – <mark>Yacc</mark>

Theory:

Parser for a grammar is a program which takes in the language string as its input and produces either a corresponding parse tree or a error.Syntax of a LanguageThe rules which tells whether a string is a valid program or not are called the syntaxSemantic s of Language The rules which give meaning to programs are called the semantic of a languageTokensWhen a string representing a program is broken into sequence of substrings, such that each substring represents a constant, identifier, operator, keyword etc of the language, these substrings are called the tokens of the language.

Lexical Analysis

The function of lexical Analyzer is to read the input stream representing the source program , one character at a time and translate into valid tokens.

Implementation Details
1: Create a lex file

The general format for lex file consists of three sections:

1. Definitions

2. Rules

3. User subroutine Section

Definitions consists of any external 'C' definitions used in the lex actions or subroutines. The other types of definitions are definitions are lex definitions which are essentially the lex substitution strings, lex start states and lex table size declarations. The rules is the basic part which specifies the regular expressions and their corresponding actions. The user Subroutines are the functions that are used in the Lex actions.
2 : Yacc is the Utility which generates the function 'yyparse' which is indeed the Parser. Yacc describes a context free, LALR(1) grammer and supports both bottom up and top-down parsing. The general format for the yacc file is very similar to that of the lex file.

1. Declarations

2. Grammar Rules

3. Subroutines

In declarations apart from the legal 'C' declarations here are few Yacc specific declarations which begins with a % sign.

1. %union It defines the Stack type for the Parser.

It is union of various datas/structures/objects.

2. % token These are the terminals returned by the yylex function to the yacc. A token cal also have type associated with it for good type checking and syntax directed translation. A type of a token can be specified as % token <stack member> tokenName.

3. %type The type of non-terminal symbol in the grammar rule can be specified with this. The format is %type <stack member> non termainal.

4. %noassoc Specifies that there is no associativity of a terminal symbol.

5. %left Specifies the left associativity of aterminal symbol.

6. %rightSpecifies the right associativity of a terminal symbol.

7. %start specifies the L.H.S. non-terminal symbol of a production rule which specifies starting point of grammar rules.

8. %prac changes the precedence level associated with a particular rule to that of the following token name or literal.

The Grammar rules are specified as follows:

Context free grammar production-

p-> AbC

Yacc Rule-

P: A b C { /* 'C' actions*/}

The general style of coding the rules is to have all Terminals in lower –case and all non-terminals in upper –case.

To facilitate a proper syntax directed translation the Yacc has something calls pseudo-variables which forms a bridge between the values of terminals/non-terminals and the actions. These pseudo variables are $$, $1, $2, $3,… ....The $$ is the L.H.S value of the rule whereas $1 is the first R. H. S value of the rule, so is the $2 etc. The default type for pseudo variables is integer unless they are specified by % type.

%token <type> etc.

Perform the following steps, in order, to create the desk calculator example program:

1. Process the **yacc** grammar file using the **-d** optional flag (which tells the **yacc** command to create a file that defines the tokens used in addition to the C language source code):

```
yacc -d calc.yacc
```

2. Use the **li** command to verify that the following files were created:
   y.tab.c The C language source file that the **yacc** command created for the parser.
   y.tab.h A header file containing define statements for the tokens used by the parser.
3. Process the **lex** specification file:
```
lex calc.lex
```

4. Use the **li** command to verify that the following file was created:

   lex.yy.c The C language source file that the **lex** command created for the lexical
   analyzer.

5. Compile and link the two C language source files:

```
cc y.tab.c lex.yy.c
```

6. Use the **li** command to verify that the following files were created:
   y.tab.o The object file for the **y.tab.c** source file
   lex.yy.o The object file for the **lex.yy.c** source file
   a.out   The executable program file

7. To then run the program directly from the **a.out** file, enter:
8. $ a.out

CODE:

1. Program **to count the number of vowels and consonants in a given string**

```
lex_yac > C vowel_counter.l
 1   %{
 2       #include <stdio.h>
 3
 4       int vowels = 0;
 5       int consonants = 0;
 6   %}
 7
 8   %option noyywrap
 9
10   %%
11   [aeiouAEIOU]        { vowels++; }        // Count vowels
12   [b-df-hj-np-tv-zB-DF-HJ-NP-TV-Z] { consonants++; } // Count consonants
13   \n                  ;                    // Ignore newline characters
14   .                   ;                    // Ignore other characters
15   %%
16
17   int main() {
18       printf("This Lex program counts the number of vowels and consonants in given text.\n");
19       printf("Enter the text and terminate it with CTRL-d.\n");
20
21       yylex(); // Start Lex scanning
22
23       printf("Vowels = %d, Consonants = %d.\n", vowels, consonants);
24       return 0;
25   }
26
```

Output:

```
hapcv@Lenovo:~/spcc$ cd lex_yac/
hapcv@Lenovo:~/spcc/lex_yac$ flex vowel_counter.l
hapcv@Lenovo:~/spcc/lex_yac$ gcc lex.yy.c -o vowel_counter -lfl
hapcv@Lenovo:~/spcc/lex_yac$ ./vowel_counter
This Lex program counts the number of vowels and consonants in given text.
Enter the text and terminate it with CTRL-d.
Mark Lopes
Vowels = 3, Consonants = 6.
hapcv@Lenovo:~/spcc/lex_yac$ []
```

2. Program to count the number of characters,words,spaces,end if lines in a given input file.

```
lex_yac > C count.l
  1   %{
  2       #include <stdio.h>
  3       int lines = 0, spaces = 0, tabs = 0, words = 0, other_chars = 0;
  4   %}
  5
  6   %%
  7
  8   \n      { lines++; }
  9   [ ]     { spaces++; }
 10   \t      { tabs++; }
 11   [A-Za-z0-9]+ { words++; }
 12   .       { other_chars++; }
 13
 14   %%
 15
 16   int main() {
 17       printf("Enter text (Ctrl+D to end input):\n");
 18       yylex();
 19       printf("Lines: %d\nSpaces: %d\nTabs: %d\nWords: %d\nOther characters: %d\n",
 20             lines, spaces, tabs, words, other_chars);
 21       return 0;
 22   }
```

Output:

```
hapcv@Lenovo:~/spcc/lex_yac$ flex count.l
hapcv@Lenovo:~/spcc/lex_yac$ gcc lex.yy.c -o count -lfl
hapcv@Lenovo:~/spcc/lex_yac$ ./count
Enter text (Ctrl+D to end input):
My Name is Mark Lopes
Lines: 1
Spaces: 4
Tabs: 0
Words: 5
Other characters: 0
hapcv@Lenovo:~/spcc/lex_yac$
```

3. Program to count no of:

    a) +ve and -ve integers

    b) +ve and -ve fractions

```
lex_yac > C pos_neg.l
  1  %{
  2      #include <stdio.h>
  3      int postiveno = 0;
  4      int negtiveno = 0;
  5      int positivefractions = 0;
  6      int negativefractions = 0;
  7  %}
  8
  9  DIGIT [0-9]
 10  %%
 11
 12  \+?{DIGIT}+          { postiveno++; }
 13  -{DIGIT}+            { negtiveno++; }
 14
 15  \+?{DIGIT}*\.{DIGIT}+ { positivefractions++; }
 16  -{DIGIT}*\.{DIGIT}+   { negativefractions++; }
 17  . ;
 18
 19  <<EOF>>              {
 20      printf("\nNo. of positive numbers: %d", postiveno);
 21      printf("\nNo. of Negative numbers: %d", negtiveno);
 22      printf("\nNo. of Positive numbers in fractions: %d", positivefractions);
 23      printf("\nNo. of Negative numbers in fractions: %d\n", negativefractions);
 24      return 0;
 25  }
 26  %%
 27
 28  int main()
 29  {
 30      printf("Enter numbers (Ctrl+D to end input):\n");
 31      yylex();
 32      return 0;
 33  }
```

Output:

```
hapcv@Lenovo:~/spcc/lex_yac$ gcc lex.yy.c -o pos_neg -lfl
hapcv@Lenovo:~/spcc/lex_yac$ ./pos_neg
Enter numbers (Ctrl+D to end input):
9913, 11, -11, 3.3 , -12.8


No. of positive numbers: 2
No. of Negative numbers: 1
No. of Positive numbers in fractions: 1
No. of Negative numbers in fractions: 1
hapcv@Lenovo:~/spcc/lex_yac$
```

4. Program to count the no of comment line in a given C program. Also eliminate them and copy that program into separate file.

```
5. /* Safer Comment Removal Lex Program */
6. %{
7.     #include <stdio.h>
8.     #include <stdlib.h>
9.     FILE *tempFile;
10.%}
11.
12.%%
13.\/\/.*                              { /* Ignore single-line comments */
   }
14.\/\*([^*]|\*+[^*/])*\*+\/           { /* Ignore multi-line comments */
   }
15..                                   { fputc(yytext[0], tempFile); }  /*
   Copy non-comment content */
16.\n                                  { fputc('\n', tempFile); } /*
   Preserve newlines */
17.%%
18.
19./* Driver function */
20.int main(int argc, char **argv) {
21.     if (argc < 2) {
```

```
22.         printf("Usage: %s <input_file>\n", argv[0]);
23.         return 1;
24.     }
25.
26.     FILE *inputFile = fopen(argv[1], "r");
27.     if (!inputFile) {
28.         perror("Error opening input file");
29.         return 1;
30.     }
31.
32.     char tempFileName[] = "temp_XXXXXX";
33.     int tempFd = mkstemp(tempFileName);
34.     if (tempFd == -1) {
35.         perror("Error creating temporary file");
36.         fclose(inputFile);
37.         return 1;
38.     }
39.
40.     tempFile = fdopen(tempFd, "w");
41.     if (!tempFile) {
42.         perror("Error opening temporary file");
43.         close(tempFd);
44.         fclose(inputFile);
45.         return 1;
46.     }
47.
48.     yyin = inputFile;
49.     yylex();
50.
51.     fclose(inputFile);
52.     fclose(tempFile);
53.
54.     // Safely replace the original file
55.     if (rename(tempFileName, argv[1]) != 0) {
56.         perror("Error replacing original file");
57.         unlink(tempFileName);  // Remove temporary file if rename fails
58.         return 1;
59.     }
60.
61.     printf("Comments removed and %s updated successfully!\n", argv[1]);
62.     return 0;
63.}
```

OUTPUT

Input.c(before)

```c
#include <stdio.h>

// This is a single-line comment at the top of the file
/* This is a multi-line comment
   that spans multiple lines
   with different indentations */

int main() {
    // Single-line comment before a variable declaration
    int x = 10; /* Inline single-line comment */

    /* Another multi-line comment
       demonstrating nested comments // like this */

    // Printing a simple message
    printf("Hello, World!"); // Print greeting

    /* Complex multi-line comment
       with some /* nested */ comment-like /* sections */ */

    return 0; // Return success
}
```

Input.c(after)

```c
#include <stdio.h>




int main() {

    int x = 10;



    printf("Hello, World!");

     comment-like */

    return 0;
}
```

64.   To count identifier, keywords, operators, special symbols, digits.

```c
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>

    /* Counters */
    int identifier_count = 0;
    int keyword_count = 0;
    int operator_count = 0;
    int special_symbol_count = 0;
    int digit_count = 0;

    /* List of C Keywords */
    const char *keywords[] = {
```

```c
        "int", "float", "double", "char", "if", "else", "while", "for", "do",
        "return", "void", "switch", "case", "break", "continue", "default",
        "typedef", "struct", "union", "static", "const", "sizeof", "goto",
"enum"
    };
    #define KEYWORDS_COUNT (sizeof(keywords) / sizeof(keywords[0]))

    /* Function to check if a word is a keyword */
    int is_keyword(const char *word) {
        for (int i = 0; i < KEYWORDS_COUNT; i++) {
            if (strcmp(word, keywords[i]) == 0)
                return 1;
        }
        return 0;
    }
%}

/* Regular Expressions */
DIGIT   [0-9]
LETTER  [a-zA-Z]
IDENTIFIER  {LETTER}({LETTER}|{DIGIT})*

%%

{IDENTIFIER} {
    if (is_keyword(yytext))
        keyword_count++;
    else
        identifier_count++;
}

{DIGIT}+ {
    digit_count++;
}

"=="|"!="|"<="|">="|"<"|">"|"\+"|"-"|"*"|"/"|"%"  {
    operator_count++;
}


"{"|"}"|"("|")"|"["|"]"|";"|","  {
    special_symbol_count++;
}

. ; /* Ignore other characters */
```

```
%%

/* Driver Code */
int main() {
    FILE *inputFile = fopen("out.c", "r");
    if (!inputFile) {
        perror("Error opening out.c");
        return 1;
    }

    yyin = inputFile;
    yylex();
    fclose(inputFile);

    printf("\nNumber of Keywords: %d", keyword_count);
    printf("\nNumber of Identifiers: %d", identifier_count);
    printf("\nNumber of Operators: %d", operator_count);
    printf("\nNumber of Special Symbols: %d", special_symbol_count);
    printf("\nNumber of Digits: %d\n", digit_count);

    return 0;
}
```

all.c

```c
/* Sample C program to demonstrate lexical analysis */

#include <stdio.h>

#include <stdlib.h>


#define MAX_SIZE 100


// Function to calculate factorial

int factorial(int n) {

    if (n == 0 || n == 1) {

        return 1;
```

```c
    }

    return n * factorial(n - 1);

}


// Main function with multiple lexical elements

int main() {

    int num = 5;

    float result = 0.0;


    // Demonstrate nested loops and conditionals

    for (int i = 1; i <= num; i++) {

        if (i % 2 == 0) {

            result += factorial(i);

        } else {

            result -= factorial(i);

        }

    }


    printf("Final result: %f\n", result);

    return 0;

}
```
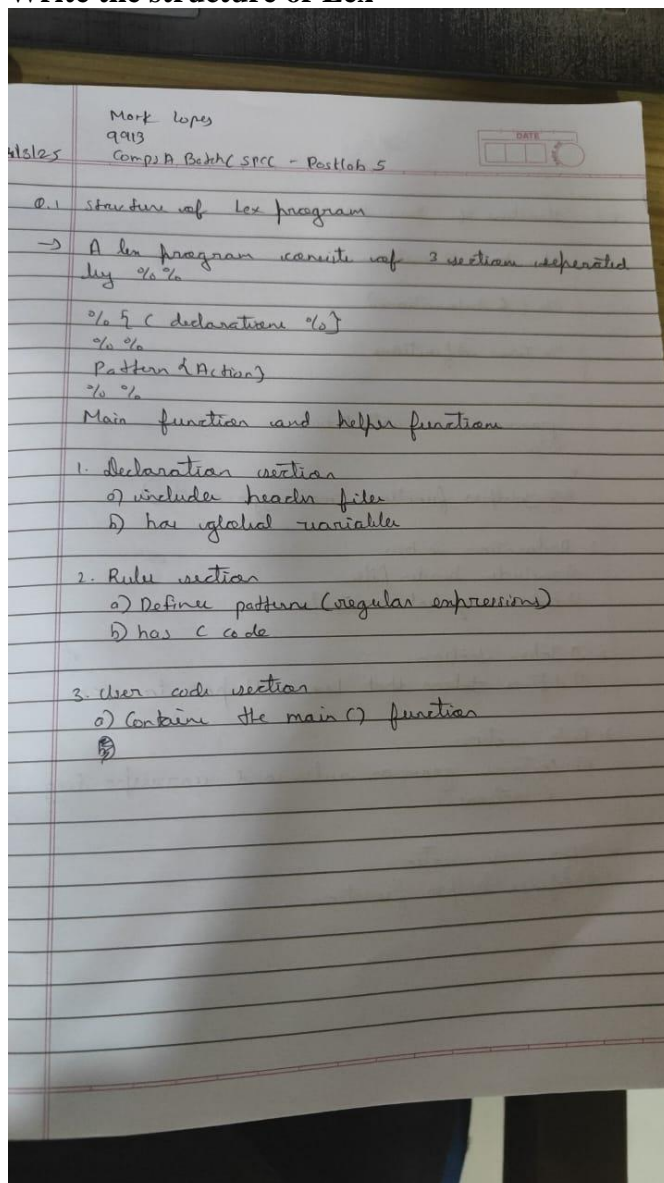
Output:


```
Number of Keywords: 13
Number of Identifiers: 58
Number of Operators: 26
Number of Special Symbols: 41
Number of Digits: 12
○ hapcv@Lenovo:~/spcc/lex_yac$ ▯
```

Postlab:

1. **Write the structure of Lex**

2. Write the structure of Yacc

Q.2 Structure of Yacc program

→ This has 3 sections :-

%{ C declarations %}

% Token definitions

%%
grammar rules with Action
%%

Helper functions (main, yyerror, etc)

1. Declaration Section
   a) Include header files
   b) declare global variables

2. Token section
   a) defines tokens that Lex will pass to yacc.

3. Rule section
   a) contains grammar rules and corresponding
      C actions.

4. User code section
   a) defines helper function