Dated:

EXPERIMENT-1

Aim: Practice of LEX/YACC of compiler writing.

Theory:

INTRODUCTION TO LEX AND YACC

We code patterns and input them to lex. It will read the patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on patterns written in the input file, and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing. The translation using Lex is shown below.

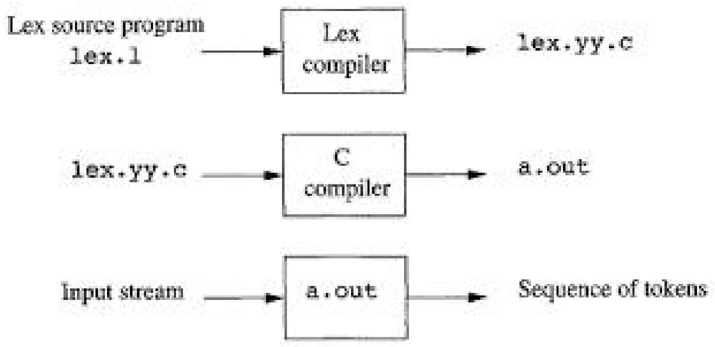


Figure 3. The translation using Lex

When the lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and location of the variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index. We code a grammar and input it to Yacc. Yacc will read the grammar and generate C code for a syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical structure the tokens. The translation using Yacc as shown below. The below figure illustrate combined use and the file naming convention used by lex and yacc, such as "bas.y" for yacc compiler and "bas.l" for lex compiler.

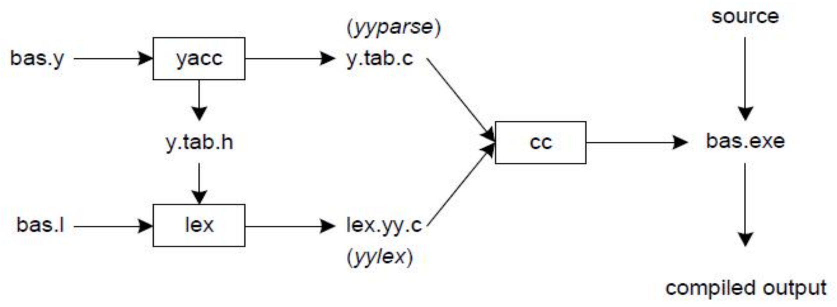


Figure 4. Lex and YACC compilers

First, we need to specify all pattern matching rules for lex (**bas.l**) and grammar rules for yacc (**bas.y**). Commands to create our compiler, **bas.exe**, are listed below:

```
    yacc  –d   bas.y                          # create y.tab.h, y.tab.c
    lex  bas.l                               # create lex.yy.c
    gcc -o bas  y.tab.c  lex.yy.c  -ll            # compile/link (output file
name ‘bas.exe’)
```

Yacc reads the grammar descriptions in **bas.y** and generates a syntax analyzer (parser), that includes function **yyparse**, in file **y.tab.c**. Included in file **bas.y** are token declarations. The **–d** option causes yacc to generate definitions for tokens and place them in file **y.tab.h**. Lex reads the pattern descriptions in **bas.l**, includes file **y.tab.h**, and generates a lexical analyzer, that includes function **yylex**, in file **lex.yy.c**.

Finally, the lexer and parser are compiled and linked together to form the executable, **bas.exe**. From **main**, we call **yyparse** to run the compiler. Function **yyparse** automatically calls **yylex** to obtain each token.

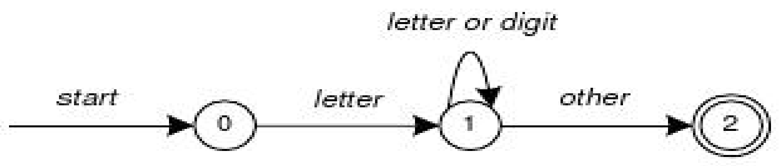
More about Lex:

The first phase in a compiler reads the input source and converts strings in the source to tokens. Using regular expressions, we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser.

The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

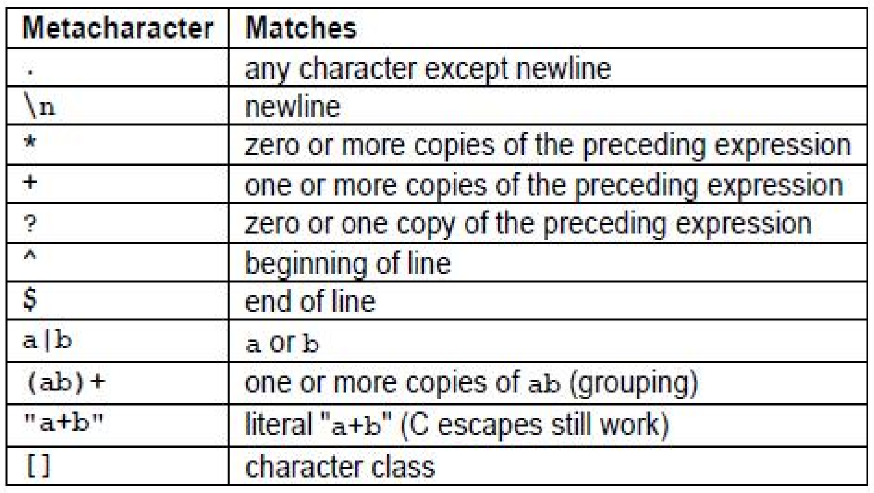
Letter (letter | digit)*

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state, and one or more final or accepting states.


$$\begin{tikzcd}
{} \arrow[r, "start"] & 0 \arrow[r, "letter"] & 1 \arrow[loop above, "letter\ or\ digit"] \arrow[r, "other"] & \boxed{2}
\end{tikzcd}$$

igure 5. Finite State Automaton

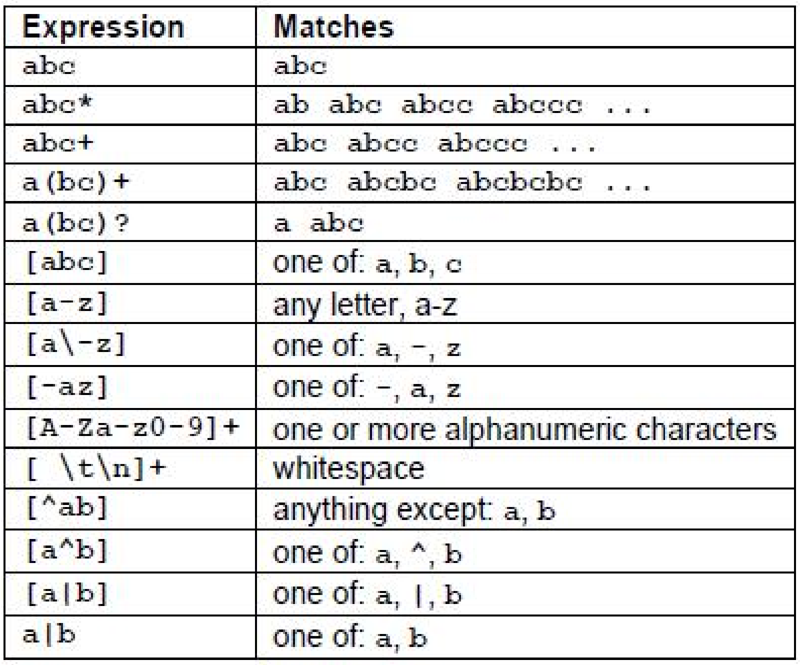
This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimic an FSA. Regular expressions in lex are composed of meta characters (Table 1). Pattern-matching examples are shown in Table 2. Within a character class, normal operators lose their meaning.



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 copy of the preceding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

Table 1: Pattern Matching Primitives

Dated:



Expression	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, a-z
[a\-z]	one of: a, -, z
[-az]	one of: -, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^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

Two operators allowed in a character class are the hyphen ("-") and circumflex ("^"). When used between two characters, the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string, the longest match wins. In case both matches are the same length, then the first pattern listed is used.

Lex Format:

```
... definitions ...
        %%
... rules ...
        %%
... subroutines ...
```

Input to Lex is divided into three sections, with %% dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file.

```
        %%
```

Input is copied to output, one character at a time. The first %% is always required, as there must always be a rules section. However, if we don't specify any rules, then the default action is to match everything and copy it to output.

Defaults for input and output are stdin and stdout, respectively.

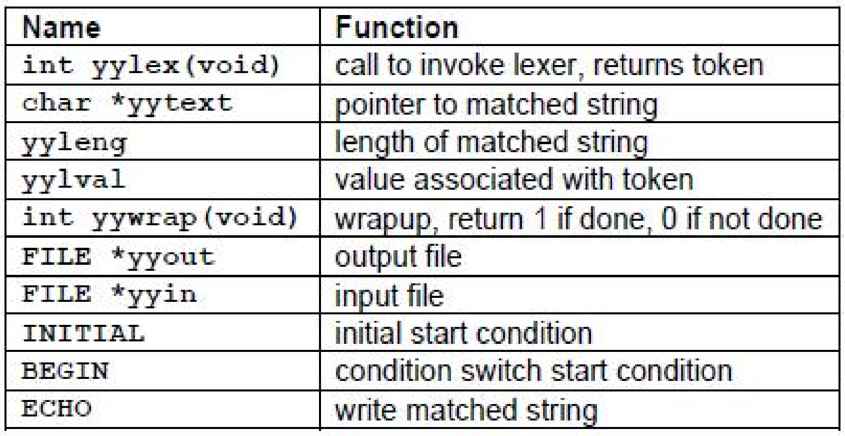
Here is the same example, with defaults explicitly coded.

```
        %%
              /* match everything except newline */
.                   ECHO;
              /* match newline */
              \n    ECHO;
              %%
        int  yywrap(void)
              {
                    return 1;
                    }
       int  main(void)
              {
                    yylex();
                    return 0;
                    }
```

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by whitespace (space, tab or newline), and an optional action associated with the pattern. The action may be a single C statement, or multiple C statements enclosed in braces. Anything not starting in column one is copied exactly to the generated C file (Comments can copied from lex file to C file). In this example there are two patterns, "." and "\n", with an ECHO action associated for each pattern. Several macros and variables are predefined by lex. ECHO is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, ECHO is defined as

#define ECHO fwrite(yytext, yyleng, 1, yyout)

Variable yytext is a pointer to the matched string (NULL-terminated), and yyleng is the length of the matched string. Variable yyout is the output file, and defaults to stdout. Function yywrap is called by lex when input is exhausted. Return 1 if you are done, or 0 if more processing is required. Every C program requires a main function. In this case, we simply call yylex, the main entry-point for lex. Some implementations of lex include copies of main and yywrap in a library, eliminating the need to code them explicitly.



Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	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

yywrap():

This function is called when end of file(or input) is encountered. If it return 1, the parsing stops. So this can be used to parse multiple files. Code can be written in third section, which will allow multiple file to be parsed.

yylineno: Provides current line number information.

Sample Program: Lexical Analyzer for C programming language

```
letter  [a-zA-Z]
digit  [0-9]
%%
{digit}+("E"("+"|"-")?{digit}+)?        printf("\n%s\tis real number",yytext);

{digit}+"."{digit}+("E"("+"|"-")?{digit}+)?  printf("\n%s\tis a float
number",yytext);

"if"|"else"|"int"|"float"|"switch"|"case"|"struct"|"char"|"return"|"for"|"do"|"whil
e"|"void"|"printf"|"scanf"        printf("\n%s\tis a keyword",yytext);

"\t"|"\b"|"\\n"|"\\t"|"\\a"|"\\b"|"\a"   printf("\n%s\tis an escape
sequences",yytext);

{letter}({letter}|{digit})*       printf("\n%s\tis an identifier",yytext);

"["|"]"|"{"|"}"|"("|")"|"#"|"'"|"""|"\"|"\\"|";"|"," printf("\n%s\t is a special
character",yytext);

"&&"|"<"|">"|"<="|">="|"+"|"="|"-"|"?"|"/"|"|"|"*"|"&"|"%"  printf("\n%s\t is
an operator",yytext);
```

```
"%d"|"%s"|"%c"|"%f"|"%e"   printf("\n%s\tis a format specifier",yytext);

%%
int yywrap()
{
return 1;}
int main(int argc, char *argv[])
{
yyin=fopen(argv[1],"r");
yylex();
fclose(yyin);
return 0;
}
```

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it find more than one match, it take the one matching the most text. If it find two or more matches of the same length, the rule listed first in the flex input file is chosen.

Once the match is determined which satisfying one of the regular expression or rule, the text corresponding to the match (token) is made available in the global character pointer yytext and its length in the global integer yyleng. The action corresponding to the matched pattern is then executed and then the remaining input is scanned for another match. If no match is found, then the default rule is executed.

YACC (Yet Another Compiler Compiler)

YACC is an LALR parser generator. The first version of YACC was created by S.C Johnson. Yacc translate a yacc file into a C file y.tab.c using LALR method

A Yacc source program has three parts:

```
Declarations
%%
Translation rules
%%
Supporting C- routines
```

The Declarations parts:

There are two optional sections in the declarations part of the Yacc program. In the first section put ordinary C declarations, delimited by %{ and %}. Here we place any temporaries used by the translation rules or procedures of the second and third sections. Also in the declarations part are declarations of grammar tokens.

Each grammar rule defines a symbol in terms of: Other symbols (Non terminals) and tokens (terminals) which come from the lexical analyzer.

Terminal symbols are of three types:

Named token: Defined via the %token identifier. By convention these are all uppercase

Character token: As same as character constant written in c language, Eg: + is a character

Constant: Literal string constant. Like C string constant Eg: "abc" is string constant.

The Lexer returns named tokens. Non terminals are usually represented by using lower case letters.

A few Yacc specific declarations which begins with a % sign in Yacc specification file

1) %union. It defines the stack type for the parser. It is union of various data/ structures/ objects
2) %token. These are terminals return by the yylex function to the yacc.
   %token NUM   NUM is a named token
3) %type. The type of non-terminal symbol in the grammar can be specified by using this rule.
4) %nonassoc. Specifies that there is no associativity of a terminal symbol
   %nonassoc ′<′   no associativity
5) %left. Specifies that there is left associativity of a terminal symbol
   %left ′+′ ′ - ′ make + and – be the same precedence and left associative
6) %right. Specifies that there is right associativity of a terminal symbol
   %right ′+′ ′- ′ make + and – be the same precedence and right associative
7) %start. Specifies that the L.H.S non terminal symbol of production rule which should be taken as the starting point of the grammar rules.

8) %prec. Change the precedence level associated with a particular rule to that of the following token name or literal. %prec <terminal>

```
%left ′+′ ′ - ′
%right ′*′ ′/′
```

The above two statements declare the associativity and precedence exist among the tokens. The precedence of the token increases from top to bottom, so the bottom listed rule has the highest precedence compared to the rule listed above it. Here * and / have the higher precedence than the addition and subtraction.

Translation rules part:

After the first %% put the translation rules. Each rule consists of a grammar production and the associated semantic action. Consider this production

<Left_side> → <alt 1>| <alt 2>|............|<alt n>

would be written in yacc as

```
<left_side>        :       <alt 1> { semantic action 1}
                   |       <alt 2> { semantic action 2}
        ……..................................
                   |       <alt n> { semantic action n}
                   ;
```

In Yacc production, quoted single character ′ c ′ is taken to be the terminal c and unquoted strings letter and digits not declare to be token to be taken as nonterminals.

The Yacc semantic action is a sequence of C statements, In a semantic action the symbol $$ refers to the attribute value associated with non terminals on the left, while $i refers to the value associated with the ith grammar symbol(terminal or non terminals) on the right. The semantic action is performed whenever we reduce by the associated production, So normally the semantic action computes a value for $$ in terms of the $i′s.

Example: Here the non-terminal term in the first production is the third grammar symbol on the right, while ′+′ is the second terminal symbol. We have omitted the semantic action for the second production, since copying the value is the default action for productions with a single grammar symbol on the right. ($$= $1) is the default action.

Supporting C routines part:

The third part of the yacc specification consists of supporting C- routines. A lexical analyzer by the name yylex() must be provided . Other procedures such as error recover routine may be added as necessary. The lexical analyzer yylex() produce pair consists of a token and its associated attribute value. The attribute value associated with a token is communicated to the parser through a yacc defined variable yylval.

How to compile Yacc and Lex files

1. Write a parser for a given grammar in a .y file
2. Write the lexical analyzer to process input and pass tokens to the parser when it needed, save the file as .l file
3. write error handler routine(like yyerror())
4. Compile .y file using the command yacc -d <filename.y> then it generate two file y.tab.c amd y.tab.h. If you are using bison, it generate <filename>.tab.c and <filename>.tab.h respectively
5. Compile lex file using the command flex <filename.l> it generate lex.yy.c
6. Compile the c file generated by the yacc compiler using gcc y.tab.c -ll it will generate .exe file a.out
7. Run the executable file and give the necessary input

History of lex and Yacc

Bison is descended from yacc, a parser generator written between 1975 and 1978 by Stephen C. Johnson at Bell Labs. As its name, short for "yet another compiler compiler," suggests, many people were writing parser generators at the time. Johnson's tool combined a firm theoretical foundation from parsing work by D. E. Knuth, which made its parsers extremely reliable, and a convenient input syntax. These made it extremely popular among users of Unix systems, although the restrictive license under which Unix was distributed at the time limited its use outside of academia and the Bell System.
In about 1985, Bob Corbett, a graduate student at the University of California, Berkeley, re-implemented yacc using somewhat improved internal algorithms, which evolved into Berkeley yacc. Since his version was faster than Bell's yacc

and was distributed under the flexible Berkeley license, it quickly became the most popular version of yacc. Richard Stallman of the Free Software Foundation (FSF) adapted Corbett's work for use in the GNU project, where it has grown to include a vast number of new features as it has evolved into the current version of bison.

Bison is now maintained as a project of the FSF and is distributed under the GNU Public License. In 1975, Mike Lesk and summer intern Eric Schmidt wrote lex, a lexical analyzer generator, with most of the programming being done by Schmidt. They saw it both as a standalone tool and as a companion to Johnson's yacc. Lex also became quite popular, despite being relatively slow and buggy. (Schmidt nonetheless went on to have a fairly successful career in the computer industry where he is now the CEO of Google.) In about 1987, Vern Paxson of the Lawrence Berkeley Lab took a version of lex written in rat for (an extended Fortran popular at the time) and translated it into C, calling it flex, for "Fast Lexical Analyzer Generator." Since it was faster and more reliable than AT&T lex and, like Berkeley yacc, available under the Berkeley license, it has completely supplanted the original lex. Flex is now a SourceForge project, still under the Berkeley license.

A flex program consists of three sections, separated by %% lines. The first section contains declarations and option settings. The second section is a list of patterns and actions, and the third section is C code that is copied to the generated scanner, usually small routines related to the code in the actions. In the declaration section, code inside of %{ and %} is copied through verbatim near the beginning of the generated C source file. In this case it just sets up variables for lines, words, and characters.

The commands for executing the LEX program are:

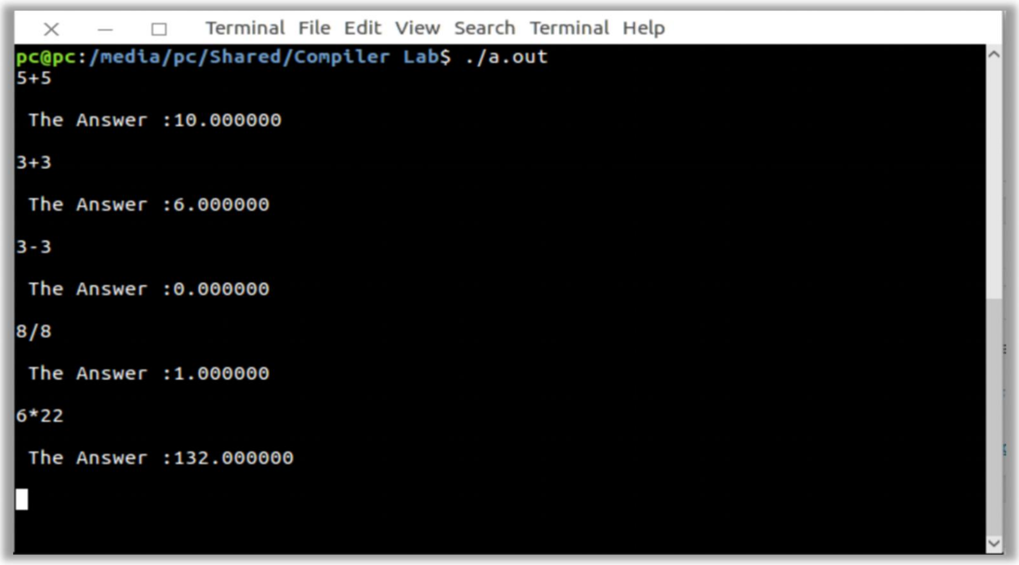
1. **lex abc.l (abc is the file name)**
2. **cc lex.yy.c -efl**
3. **./a.out**

Practice Questions:

1. Write a program to construct a small calculator that can add, subtract, multiply and divide in LEX.

```
% {
      int op = 0,i;
     float a, b;
% }

dig [0-9]+|([0-9]*)"."([0-9]+)
add "+"
sub "-"
mul "*"
div "/"
pow "^"
ln \n
%%

/* digi() is a user defined function */
{dig} {digi();}
{add} {op=1;}
{sub} {op=2;}
{mul} {op=3;}
{div} {op=4;}
{pow} {op=5;}
{ln} {printf("\n The Answer :%f\n\n",a);}

%%
digi()
{
 if(op==0)

/* atof() is used to convert
      - the ASCII input to float */
 a=atof(yytext);

 else
 {
 b=atof(yytext);

 switch(op)
 {
   case 1:a=a+b;
    break;

   case 2:a=a-b;
   break;

   case 3:a=a*b;
   break;

   case 4:a=a/b;
   break;
```

```
  case 5:for(i=a;b>1;b--)
  a=a*i;
  break;
 }
op=0;
 }
}

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

 yylex();
}

yywrap()
 {
 return 1;
 }
```

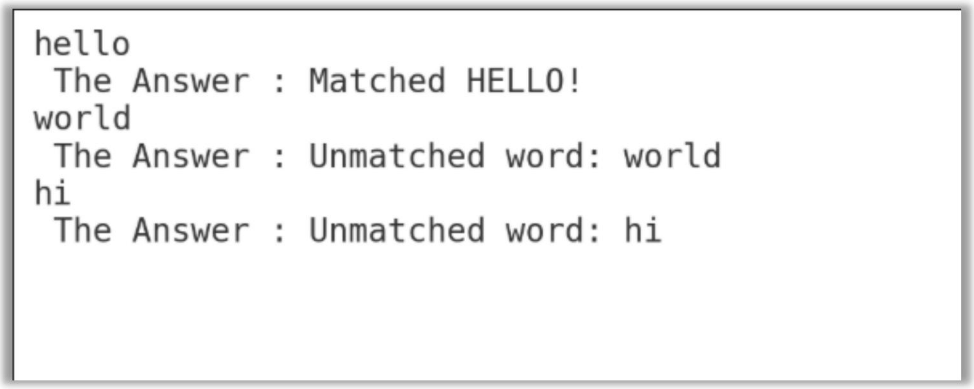


```
pc@pc:/media/pc/Shared/Compiler Lab$ ./a.out
5+5

 The Answer :10.000000

3+3

 The Answer :6.000000

3-3

 The Answer :0.000000

8/8

 The Answer :1.000000

6*22

 The Answer :132.000000

```

2. Write a program to match a string in LEX.

```
%{
#include <stdio.h>
%}

%%
hello    { printf("Matched HELLO!\n"); }
[a-zA-Z]+  { printf("Unmatched word: %s\n", yytext); }
.|\n     { /* ignore other characters */ }
%%

int main() {
   yylex();
```

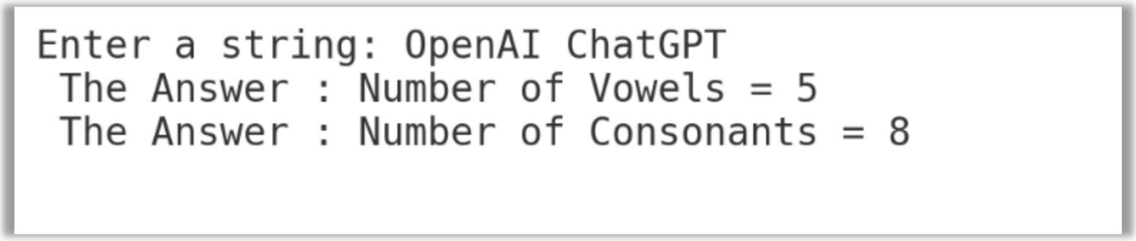
```
    return 0;
}
```



```
hello
 The Answer : Matched HELLO!
world
 The Answer : Unmatched word: world
hi
 The Answer : Unmatched word: hi
```

3.Write a LEX program to count the number of vowels and consonants in a given string.

```
%{
#include <stdio.h>
int vowels = 0, consonants = 0;
%}

%%
[aeiouAEIOU]    { vowels++; }
[a-zA-Z]        { consonants++; }
.|\n            { /* ignore everything else */ }
%%

int main() {
    printf("Enter a string: ");
    yylex();
    printf("\n The Answer : Number of Vowels = %d\n", vowels);
    printf(" The Answer : Number of Consonants = %d\n",
consonants);
    return 0;
}
```



```
Enter a string: OpenAI ChatGPT
 The Answer : Number of Vowels = 5
 The Answer : Number of Consonants = 8
```

Dated:

EXPERIMENT-2

Aim: Write a program to check whether a string belongs to the grammar or not.

a)
S -> aS
S -> Sb
S -> ab
String of the form : aab

b)
S -> aSa
S -> bSb
S -> a
S -> b
The Language generated is : All Odd Length Palindromes

c)
S -> aSbb
S -> abb
The Language generated is : anb2n , where n > 1

d)
S -> aSb
S -> ab
The Language generated is : anbn, where n > 0

Theory:

A Context-Free Grammar (CFG) is a formal system used to define the structure of languages. A CFG is represented by four components: a set of variables (non-terminals), a set of terminals, a set of production rules, and a start symbol. The grammar rules are applied recursively to generate strings of the language until only terminals remain. CFGs are powerful enough to represent recursive patterns such as balanced parentheses, palindromes, and equal numbers of symbols.

(a) Grammar:
S → aS | Sb | ab
This grammar generates strings that begin with one or more occurrences of the symbol `a` and end with exactly one symbol `b`. The recursive rule `S → aS` allows repetition of `a` on the left side, while the rule `S → Sb` ensures that the final `b` occurs at the end of the string. The terminating rule `S → ab` provides the base case. Thus, the language generated is L = { $a^n b$ | n ≥ 1 }. Example strings are `ab`, `aab`, `aaab`.

(b) Grammar:
S → aSa | bSb | a | b
This grammar produces palindromes of odd length. The recursive rules ensure symmetry by placing the same symbol at both ends, while the productions `S → a` and `S → b` provide the middle character that terminates the recursion. Therefore, the

grammar generates all odd-length palindromes over {a, b}. Example strings are `a`, `b`, `aba`, `bab`, `ababa`.

(c) Grammar:
S → aSbb | abb
Here, the base rule `S → abb` creates the smallest valid string, consisting of one `a` followed by two `b`s. The recursive rule `S → aSbb` adds one more `a` to the left and two more `b`s to the right in every step. As a result, the number of `b`s is always double the number of `a`s. The language generated is L = { $a^n b^{2n}$ | n > 1 }. Example strings are `abb`, `aabbbb`, `aaabbbbbb`.

(d) Grammar:
S → aSb | ab
This grammar generates equal numbers of `a`s and `b`s in balanced order. The base case `S → ab` represents the smallest valid string with one `a` and one `b`. The recursive rule `S → aSb` adds an `a` at the beginning and a `b` at the end in every step, maintaining balance. Thus, the grammar generates the language L = { $a^n b^n$ | n > 0 }. Example strings are `ab`, `aabb`, `aaabbb`.

Conclusion:
The above grammars demonstrate the expressive power of CFGs. Grammar (a) generates strings of the form $a^n b$, grammar (b) generates odd-length palindromes, grammar (c) generates strings of the form $a^n b^{2n}$, and grammar (d) generates strings of the form $a^n b^n$. These examples highlight how CFGs can represent recursive, symmetric, and dependent structures in formal languages.

Source Code :

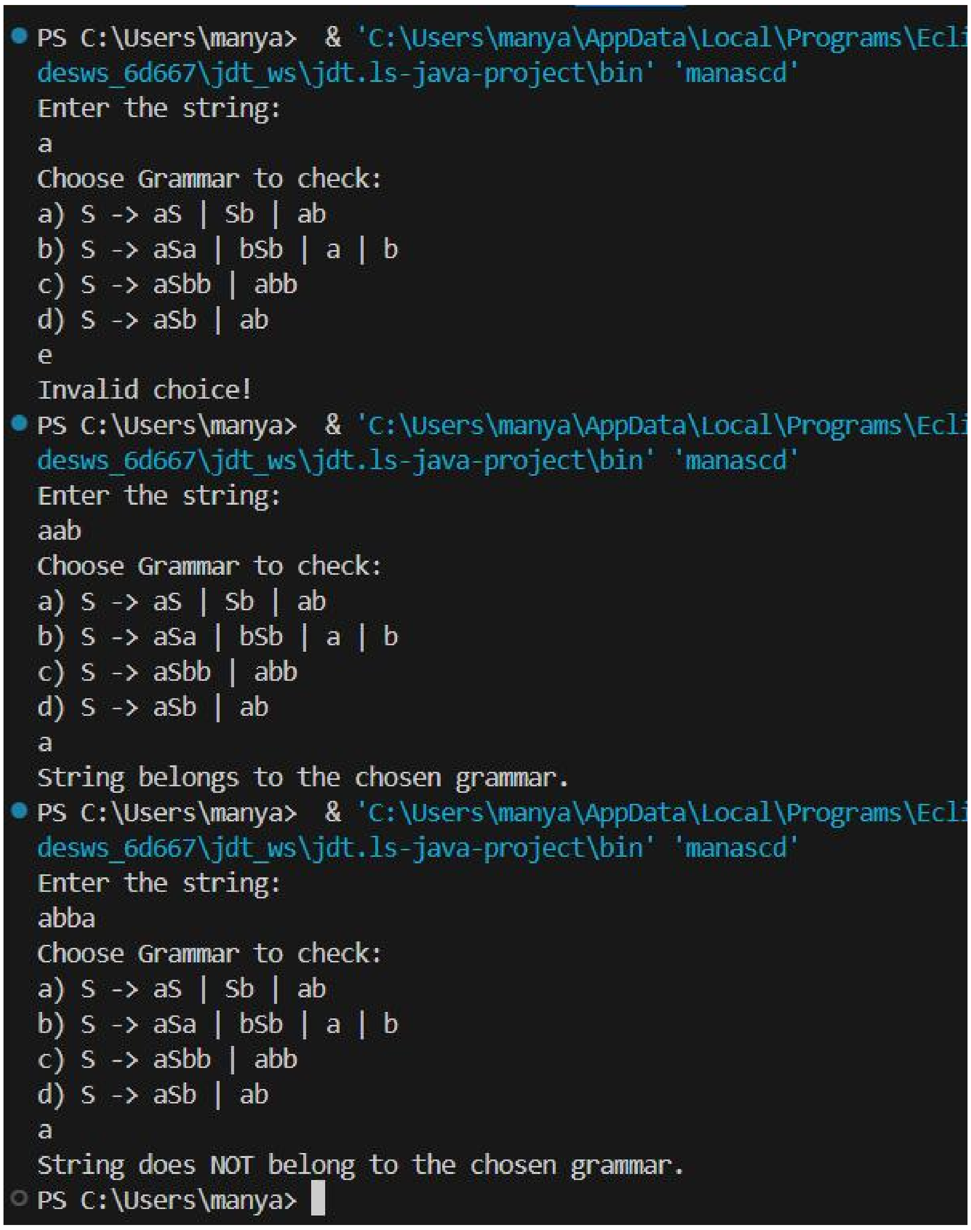
```
import java.util.Scanner;

public class manascd {

static boolean checkGrammarA(String str) {

if (str.length() < 2 || str.charAt(str.length()-1) != 'b') return false;

for (int i = 0; i < str.length()-1; i++) {

if (str.charAt(i) != 'a') return false;

}

return true;

}

static boolean checkGrammarB(String str) {
```

```
int n = str.length();

if (n % 2 == 0) return false;

for (int i = 0; i < n/2; i++) {

if (str.charAt(i) != str.charAt(n-1-i)) return false;

}

return true;

}

static boolean checkGrammarC(String str) {

int countA = 0, countB = 0;

int i = 0;

while (i < str.length() && str.charAt(i) == 'a') {

countA++;

i++;

}

while (i < str.length() && str.charAt(i) == 'b') {

countB++;

i++;

}

return (i == str.length() && countA > 1 && countB == 2*countA);

}

static boolean checkGrammarD(String str) {

int countA = 0, countB = 0;

int i = 0;

while (i < str.length() && str.charAt(i) == 'a') {

countA++;
i++;
}
while (i < str.length() && str.charAt(i) == 'b') {
countB++;
i++;
}
return (i == str.length() && countA > 0 && countA == countB);
}
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.println("Enter the string: ");
String input = sc.nextLine();
System.out.println("Choose Grammar to check: ");
System.out.println("a) S -> aS | Sb | ab");
System.out.println("b) S -> aSa | bSb | a | b");
System.out.println("c) S -> aSbb | abb");
System.out.println("d) S -> aSb | ab");
char choice = sc.next().charAt(0);
boolean result = false;
switch(choice) {
case 'a':
result = checkGrammarA(input);
break;
case 'b':
result = checkGrammarB(input);
```

```
break;
case 'c':
result = checkGrammarC(input);
break;
case 'd':
result = checkGrammarD(input);
break;
default:
System.out.println("Invalid choice!");
return;
}
if(result)
System.out.println("String belongs to the chosen grammar.");
else
System.out.println("String does NOT belong to the chosen grammar.");
}
}
```

Dated:

Output :



```
PS C:\Users\manya>  & 'C:\Users\manya\AppData\Local\Programs\Ecli
desws_6d667\jdt_ws\jdt.ls-java-project\bin' 'manascd'
Enter the string:
a
Choose Grammar to check:
a) S -> aS | Sb | ab
b) S -> aSa | bSb | a | b
c) S -> aSbb | abb
d) S -> aSb | ab
e
Invalid choice!
PS C:\Users\manya>  & 'C:\Users\manya\AppData\Local\Programs\Ecli
desws_6d667\jdt_ws\jdt.ls-java-project\bin' 'manascd'
Enter the string:
aab
Choose Grammar to check:
a) S -> aS | Sb | ab
b) S -> aSa | bSb | a | b
c) S -> aSbb | abb
d) S -> aSb | ab
a
String belongs to the chosen grammar.
PS C:\Users\manya>  & 'C:\Users\manya\AppData\Local\Programs\Ecli
desws_6d667\jdt_ws\jdt.ls-java-project\bin' 'manascd'
Enter the string:
abba
Choose Grammar to check:
a) S -> aS | Sb | ab
b) S -> aSa | bSb | a | b
c) S -> aSbb | abb
d) S -> aSb | ab
a
String does NOT belong to the chosen grammar.
PS C:\Users\manya>
```

Dated:

EXPERIMENT-3

Aim: Write a program to check whether a string include Keywords or not.

Theory:

In Java, keywords are reserved words that have a predefined meaning in the language and cannot be used as identifiers such as variable names, class names, or method names. They form the fundamental building blocks of Java syntax and define the structure, control flow, data types, and access levels within a program. Since keywords are part of the core grammar of Java, their usage is strictly enforced by the compiler, ensuring that code follows a consistent set of rules.

Java provides a total of 53 keywords (including literals like true, false, and null) that are categorized based on their purpose. For example, keywords like class, interface, and enum are used for defining types, while public, private, and protected specify access control. Keywords such as if, else, switch, while, and for are used for decision-making and iteration, controlling the logical flow of the program. Data type keywords like int, float, char, boolean, and double define the kinds of values variables can hold. Similarly, keywords such as static, final, and abstract modify the behavior of classes, methods, and variables.

The keywords try, catch, throw, throws, and finally are dedicated to exception handling, ensuring that runtime errors can be managed systematically. The return keyword specifies the value that a method gives back to the caller, whereas void indicates that a method does not return any value. Java also supports keywords such as extends and implements for inheritance and polymorphism, reflecting the object-oriented nature of the language.

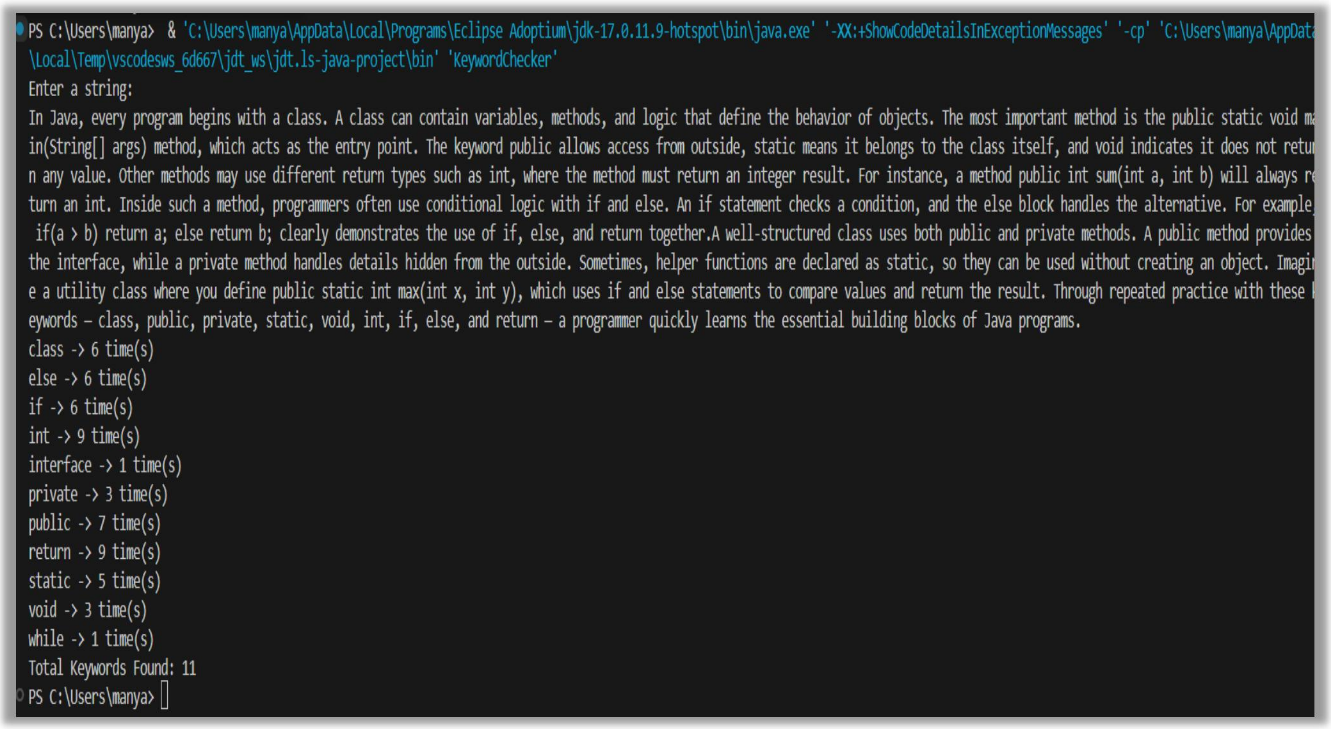
Overall, Java keywords define the syntax rules, programming constructs, and object-oriented features of the language. Understanding the meaning and proper usage of keywords is essential for writing correct and efficient Java programs. Since these words are reserved and case-sensitive, they cannot be redefined or used for any other purpose by the programmer. Mastery of keywords forms the foundation of learning Java and helps in building robust applications.

Source Code:

```
import java.util.*;

public class KeywordChecker {

static String[] javaKeywords = {

"abstract","assert","boolean","break","byte","case","catch","char","class",

"const","continue","default","do","double","else","enum","extends","final",
"finally","float","for","goto","if","implements","import","instanceof","int",
"interface","long","native","new","package","private","protected","public",
"return","short","static","strictfp","super","switch","synchronized","this",
"throw","throws","transient","try","void","volatile","while","true","false","null"
};
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.println("Enter a string:");
String input = sc.nextLine();
String[] words = input.split("\\W+");
int[] counts = new int[javaKeywords.length];
int total = 0;
for (String word : words) {
for (int i = 0; i < javaKeywords.length; i++) {
if (word.equals(javaKeywords[i])) {
counts[i]++;
}
}
}
for (int i = 0; i < javaKeywords.length; i++) {
if (counts[i] > 0) {
System.out.println(javaKeywords[i] + " -> " + counts[i] + " time(s)");
total++;
}
}
```

```
if (total == 0) {
System.out.println("No Java keywords found in the string.");
} else {
System.out.println("Total Keywords Found: " + total);
}
}
}
```

Output :



```
PS C:\Users\manya>  & 'C:\Users\manya\AppData\Local\Programs\Eclipse Adoptium\jdk-17.0.11.9-hotspot\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\manya\AppDat
\Local\Temp\vscodesws_6d667\jdt_ws\jdt.ls-java-project\bin' 'KeywordChecker'
Enter a string:
In Java, every program begins with a class. A class can contain variables, methods, and logic that define the behavior of objects. The most important method is the public static void ma
in(String[] args) method, which acts as the entry point. The keyword public allows access from outside, static means it belongs to the class itself, and void indicates it does not retur
n any value. Other methods may use different return types such as int, where the method must return an integer result. For instance, a method public int sum(int a, int b) will always re
turn an int. Inside such a method, programmers often use conditional logic with if and else. An if statement checks a condition, and the else block handles the alternative. For example
 if(a > b) return a; else return b; clearly demonstrates the use of if, else, and return together.A well-structured class uses both public and private methods. A public method provides
the interface, while a private method handles details hidden from the outside. Sometimes, helper functions are declared as static, so they can be used without creating an object. Imagi
e a utility class where you define public static int max(int x, int y), which uses if and else statements to compare values and return the result. Through repeated practice with these
eywords – class, public, private, static, void, int, if, else, and return – a programmer quickly learns the essential building blocks of Java programs.
class -> 6 time(s)
else -> 6 time(s)
if -> 6 time(s)
int -> 9 time(s)
interface -> 1 time(s)
private -> 3 time(s)
public -> 7 time(s)
return -> 9 time(s)
static -> 5 time(s)
void -> 3 time(s)
while -> 1 time(s)
Total Keywords Found: 11
PS C:\Users\manya> 
```