Experiment No.9
Study and implementation of lexical analyzer using LEX Tool and parser using YACC Tool.
Date of Performance: 03-03-2024
Date of Submission: 10-03-2024

Aim: Study and implementation of lexical analyzer using LEX Tool and parser using YACC Tool.

Objective: To develop an ability to use modern compiler construction tools for lexical analyzer and parser.

Theory:

LEX: Lex is a program that generates lexical analyzer. It is used with YACC parser generator. The lexical analyzer is a program that transforms an input stream into a sequence of tokens. It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

Structure of a Lex file:

The structure of a Lex file is intentionally similar to that of a yacc file; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

- The **definition** section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
- The **rules** section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.
- The **C code** section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

Example of a Lex file

The following is an example Lex file for the flex version of Lex. It recognizes strings of numbers (positive integers) in the input, and simply prints them out.

```
/*** Definition section ***/

%{
```

```
/* C code to be copied verbatim */
#include <stdio.h>
%}

/* This tells flex to read only one input file */
%option noyywrap

%%
    /*** Rules section ***/

    /* [0-9]+ matches a string of one or more digits */
[0-9]+  {
            /* yytext is a string containing the matched text. */
            printf("Saw an integer: %s\n", yytext);
        }

.|\n    {   /* Ignore all other characters. */   }

%%
/*** C Code section ***/

int main(void)
{
    /* Call the lexer, then quit. */
    yylex();
    return 0;
}
```

If this input is given to flex, it will be converted into a C file, lex.yy.c. This can be compiled into an executable which matches and outputs strings of integers. For example, given the input:

abc123z.!&*2gj6

the program will print:

```
Saw an integer: 123
Saw an integer: 2
Saw an integer: 6
```

YACC: YACC (Yet Another Compiler-Compiler) is a computer program for the Unix operating system developed by Stephen C. Johnson. It is a Look Ahead Left-to-Right (LALR) parser generator, generating a LALR parser (the part of a compiler that tries to make syntactic sense of the source code) based on a formal grammar, written in a notation similar to Backus–Naur Form (BNF).

- **YACC:** YACC stands for Yet Another Compiler Compiler. YACC provides a tool to produce a parser for a given grammar. YACC is a program designed to compile a LALR (1) grammar. The input of YACC is the rule or grammar and the output is a C program.
- **Example Code:**

Input File:
YACC input file is divided in three parts.

/* definitions */

....

%%

/* rules */

....

%%

/* auxiliary routines */

....

Input File: Definition Part:

- The definition part includes information about the tokens used in the syntax definition:
- %token NUMBER

  %token ID
- YACC automatically assigns numbers for tokens, but it can be overridden by

  %token NUMBER 621

- YACC also recognizes single characters as tokens. Therefore, assigned token numbers should no overlap ASCII codes.
- The definition part can include C code external to the definition of the parser and variable declarations, within **%{** and **%}** in the first column.
- It can also include the specification of the starting symbol in the grammar:

  %start nonterminal

Input File: Rule Part:

- The rules part contains grammar definition in a modified BNF form.
- Actions is C code in { } and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() function must call the function yyparse().

Input File:

- If yylex() is not defined in the auxiliary routines sections, then it should be included:

  #include "lex.yy.c"
- YACC input file generally finishes with: .y

Output Files:

- The output of YACC is a file named **y.tab.c**
- If it contains the **main()** definition, it must be compiled to be executable.

- Otherwise, the code can be an external function definition for the function **int yyparse()**
- If called with the **–d** option in the command line, Yacc produces as output a header file **y.tab.h** with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).
- If called with the **–v** option, Yacc produces as output a file **y.output** containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.

Example:
Yacc File (.y)

```
%{
   #include <ctype.h>
   #include <stdio.h>
   #define YYSTYPE double /* double type for yacc stack */
%}
%%
 Lines :  Lines S '\n' { printf("OK \n"); }
       |  S '\n'
       |  error '\n' {yyerror("Error: reenter last line:");
                      yyerrok; };
 S     :  '(' S ')'
```

```
        |  '[' S ']'
        |   /* empty */  ;
%%
#include "lex.yy.c"
void yyerror(char * s)
/* yacc error handler */
{
 fprintf (stderr, "%s\n", s);
}
int main(void)
 {
 return yyparse();
 }
```

Lex File (.l)

```
%{
%}
%%
```

```
[ \t]     { /* skip blanks and tabs */ }

\n|.      { return yytext[0]; }

%%
```

For Compiling YACC Program:

1. Write lex program in a file file.l and yacc in a file file.y
2. Open Terminal and Navigate to the Directory where you have saved the files.
3. type lex file.l
4. type yacc file.y
5. type cc lex.yy.c y.tab.h -ll
6. type ./a.out

Code:

1. Lexer File (lexer.l):

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


%%
[0-9]+    { printf("NUMBER: %s\n", yytext); }
[a-zA-Z]+ { printf("IDENTIFIER: %s\n", yytext); }
"="       { printf("ASSIGNMENT: %s\n", yytext); }
"+"       { printf("PLUS: %s\n", yytext); }
"-"       { printf("MINUS: %s\n", yytext); }
"*"       { printf("MULTIPLY: %s\n", yytext); }
"/"       { printf("DIVIDE: %s\n", yytext); }
```

```
[ \t\n]   { /* Ignore whitespace */ }
.         { printf("UNKNOWN: %s\n", yytext); }
%%

int main() {
   yylex();  // Call lexical analyzer
   return 0;
}

int yywrap() {
   return 1;  // End of input
}
```

2. Parser File (parser.y):

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

%token NUMBER
%token PLUS MINUS MULTIPLY DIVIDE
%token LPAREN RPAREN

%%

expr:
     expr PLUS term    { printf("ADD\n"); }
```

```
    | expr MINUS term   { printf("SUBTRACT\n"); }
    | term
    ;

term:
      term MULTIPLY factor   { printf("MULTIPLY\n"); }
    | term DIVIDE factor     { printf("DIVIDE\n"); }
    | factor
    ;

factor:
      NUMBER   { printf("NUMBER: %d\n", $1); }
    | LPAREN expr RPAREN
    ;

%%

int main() {
    printf("Enter an expression:\n");
    yyparse();
    return 0;
}

int yyerror(char *msg) {
    printf("Syntax error: %s\n", msg);
    return 1;
```

}

Output:

```
root@LAB0303:~# nano lexer.l
root@LAB0303:~# nano lexer.l
root@LAB0303:~# flex -o lexer.c lexer.l
root@LAB0303:~# bison -d -o parser.c parser.y
root@LAB0303:~# gcc lexer.c parser.c -o parser -ll
```

```
C:\Users\admin\Desktop\VIvek_SPCC>flex lexer.l

C:\Users\admin\Desktop\VIvek_SPCC>gcc lex.yy.c -o lexer

C:\Users\admin\Desktop\VIvek_SPCC>lexer
x = 10 + 20
IDENTIFIER: x
ASSIGNMENT: =
NUMBER: 10
PLUS: +
NUMBER: 20
```

```
root@LAB0303:~# ./parser
Enter an expression:
3 + 5 + 3
NUMBER: 3
NUMBER: 5
ADD
NUMBER: 3
```

Conclusion:

LEX is a lexical analyzer generator that converts input patterns into tokens, acting as the first phase of a compiler. It uses regular expressions to define patterns and generates C code for token recognition. YACC (Yet Another Compiler Compiler) is a parser generator that processes tokens from LEX and applies context-free grammar (CFG) to construct a syntax tree. The grammar is typically defined in BNF format, specifying rules like:

$S \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

LEX and YACC together automate syntax analysis, making compiler development easier and structured. While LEX focuses on tokenization, YACC ensures syntactic correctness, enabling efficient parsing for language processing tasks.