Lab No. 04 – An Advanced Syntax Analyzer with Symbol Table

Step 1: Build the Lexical Analyzer –

```
1  %{
2  #include "y.tab.h"
3  #include <math.h>
4  extern double vbltable[26];
5  %}
6  %%
7  ([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
8        yylval.dval = atof(yytext); return NUMBER;
9  }
10 [ \t] ; /* ignore whitespace */
11 [a-z] { yylval.vblno = yytext[0] - 'a'; return NAME; }
12 \n|. return yytext[0];
13 %%
```

Code Listing 1 – "LEX.L" – A Lexical Analyzer that accepts Integers, Operators and Variables

Explanation of the Lexical Analyzer:

- **Line no. 7** – A generalized regular expression to accept any possible double numbers. Note that, exponential numbers can also be used using 'e' or 'E' prefix.
- **Line no. 8** – A **dval** property of the union **yylval** is used to store the double value after conversion from array to float and then is passed as a token called **NUMBER**.
- **Line no. 11** – A very simple REGEX that accepts all possible alphabets and sends the index "NAME" which is obtained by subtracting the ASCII value of 'a' from the accepted character.
- **Line no. 12** – As codes can be multiline so NEWLINE is also a valid token.
- **Line no. 4** – This is the simplest form of a **Symbol Table** that can hold values of 26 different variables as the size of the array is just 26.

Step 2: Build the Syntax Analyzer –

```
1  %{
2  #include<stdio.h>
3  #include<stdlib.h>
4  extern FILE *yyin;
5  double vbltable[26] ;
6  void yyerror(const char *c){
7        fprintf(stderr,"%s",c);
8  }
9  int yylex();
10 %}
11 %union{
12       double dval;
13       int vblno;
14 }
15 %token <vblno> NAME
16 %token <dval> NUMBER
17 %type <dval> expression term factor
18 %%
```

```
19 statement_list: statement_list statement ';' '\n'
20               | statement ';' '\n'
21 ;
22 statement: NAME '=' expression
                { vbltable[$1] = $3; printf("%c = %lf\n",$1+'a',$3);}
23          | expression { printf("= %g\n",$1); }
24 ;
25 expression: expression '+' term { $$ = $1 + $3; }
26           | expression '-' term { $$ = $1 - $3; }
27           | term { }
28 ;
29 term: term '*' factor { $$ = $1 * $3; }
30     | term '/' factor
31     {
32       if($3==0.0){
33          yyerror ("divide by zero");
34       }
35       else{
36          $$ = $1 / $3;
37       }
38     }
39     | factor { }
40 ;
41 factor: '-' factor { $$ = -$2; }
42       | '(' expression ')' { $$ = $2; }
43       | NUMBER {$$ = $1;}
44       | NAME { $$ = vbltable[$1]; }
45 ;
46 %%
47 int main(){
48       FILE *file;
49       file = fopen("code.c", "r") ;
50       if (!file) {
51             printf("Could not open file");
52             exit (1);
53       }
54       else {
55             yyin = file;
56       }
57       yyparse();
58 }
```

Code Listing 2 – "YAC.Y" – The Syntax Analyzer

Explanation of the Syntax Analyzer:

- **Line no. 4 –** We need this to store the incoming file stream.
- **Line no. 5** – We are using the same external array that is already declared in the lexical analyzer.
- **Line no. 11-14 –** Declared the union that we need to store double value of the given input.
- **Line no. 25-45** – Implemented the simple **CFG** that can recognize simple addition, subtraction, multiplication, division and unary minus operations.

- **Line no. 19-20** – Here we have specified that the code can be a multiline code where each line ends with a ';' and separated by a newline.
- **Line no. 23** – Simply printed the final result.
- **Line no. 44** – Whenever we are encountering a variable, we access our symbol table i.e **vbltable[26]** and store the value at the corresponding index of the array. For example if we get **"b=10;"** the code will save the value **10** to **vbltable[1]** (as ASCII code of 'b' – ASCII code of 'a' = 1).
- **Line not. 22** – When it's time to print the name of the variable we are adding the index with the ASCII code of 'a' and thus getting back the character equivalent of the corresponding index.
- **Line no 57 –** Called the YYPARSE() function to start parsing the valid lexemes according to our specified grammar.

Step 3 – Preparing the input file:

Create a input file called "code.c" and give an example input as follows:

```
x=12+3;
y=x+13;
z=x+y;
```

Code Listing 3 – Sample input file

Step 4 – Compile and run the code:

Run the following commands in the TERMINAL as per the given order –

i. `lex file_name1.l` [The "**file_name1**" is the name of the LEX file that contains the code.]
ii. `yacc -d file_name2.y` [The –d switch is used to create the header file "y.tab.h" and "**file_name2**" is the BISON file that contains the code.]
iii. `cc lex.yy.c y.tab.c -o output_name -ll -ly`
[**"cc**" is for selecting the C compiler, "**lex.yy.c**" is the output file from the lexical analyzer, "**y.tab.c**" is the output file from the syntax analyzer, "**-o**" is to specify an output file name, "**output_name**" can be given by user, "**-ll**" is to add the FLEX library and "**-ly**" is to add the BISON library to the compiler.]
iv. `./output_name` [To run the output file.]

Tasks for LAB 04:

1. Create a Syntax Analyzer that can use variable storing technique and:
    a. Call functions like exponent, square root, maximum (number 1, number 2) and use their values in calculation of the arithmetic operations.
    b. Use data separation for different data types like integer and double. This means the compiler will only add/subtract integer from integer and double from double, otherwise, it should generate an error.