Lab No. 03 – Introduction to a Syntax Analyzer

Creating a Syntax Analyzer:

In order to create a Syntax Analyzer we need the following:

1. A Lexical Analyzer that matches tokens to specified patterns and produces valid lexemes for processing. The Lexical Analyzer is built with **FLEX**.
2. A Syntax Analyzer module built in the package **BISON**.
3. Compiling and running the code.

Step 1: Build the Lexical Analyzer –

To build a Lexical Analyzer we need the package FLEX and some patterns to start with. Suppose we want to only recognize simple integer numbers and some operators to do some mathematical operations with the inputs. So we can use a lexical analyzer as follows:

```
1  %{
2  #include "y.tab.h"
3  extern int yylval;
4  %}
5  %%
6  [0-9]+ { yylval = atoi(yytext); return NUMBER;}
7  [ \t]+ {} /* ignore whitespace */
8  \n {return 0;} /* logical EOF */
9  . {return yytext[0];}
10 %%
```

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

Explanation of the Lexical Analyzer:

- **Line no. 2** – We need the "order of tokens" that is specified in the syntax analyzer. So, we have added the grammar as a library file in our Lexical Analyzer.
- **Line no. 3** – We need to pass valid lexemes to the next phase i.e. the Syntax Analyzer. So, we have created a global and external variable called "yylval" and passed the value through it.
- **Line no. 6** – A very simple REGEX that accepts all possible integer numbers and passes the values through a token called "NUMBER" after transforming the string to an integer.
- **Line no. 8** – We have specified a NEWLINE character to mark the end of inputs.
- **Line no. 9** – We have passed all possible operators like +,-.* or / as singular characters.

We can create custom patterns or REGEX to recognize any tokens that are required for the specific program or compiler. For example, to use the variable types int, char, double or float, there are specific tokens and patterns that are defined in the Lexical Analyzer of the C compiler.

Step 2: Build the Syntax Analyzer –

To build a Syntax Analyzer we need some context free grammar or set of rules that the inputs need to follow after being recognized by a Lexical Analyzer. So, we will implement the grammar and their corresponding actions in a file built with the help of the package BISON. In our first example, let's use the following Syntax Analyzer.

```
1  %{
2        #include <stdio.h>
3         yylex();
4  %}
5  %token NUMBER
6  %%statements:  expressions { printf("= %d\n",$1); }
7                 ;
8  expressions:   expressions '+' NUMBER { $$ = $1 + $3; }
9                 | expressions '-' NUMBER { $$ = $1 - $3; }
10                | NUMBER { $$ = $1; }
11                ;
12 %%
13 int main() {
14       yyparse();
15 }
```

Code Listing 2 – "YAC.Y" – A Syntax Analyzer that can do addition and subtraction

Explanation of the Syntax Analyzer:

- **Line no. 2 –** We need the **STDIO** library to use printf() function or standard C functions.
- **Line no. 4** – We called the yylex() function to initiate the Lexical Analyzer.
- **Line no. 5 –** Declared the token "**NUMBER**" that we need from the Lexical Analyzer.
- **Line no. 6-11** – Implemented the simple **CFG** that can recognize simple addition and subtraction operations.
- **Line no. 6** – Once we have recognized the "**ROOT**" of our syntax tree, we printed the value of the tree.
- **Line no. 8** – Recognizes any production that has an **EXPRESSIONS** at the beginning, a **"+"** at the middle and a **NUMBER** at the end of the string. The whole production can now be addressed using index of the terms used in the production. And the resultant is stored in **$$** variable that indicates the **EXPRESSIONS** token. Here, **$1** indicates the second **EXPRESSIONS** token and **$3** indicates the **NUMBER** token. And consequently **$2** would be the **"+"** symbol.
- **Line no. 9** - Recognizes any production that has an **EXPRESSIONS** at the beginning, a **"-"** at the middle and a **NUMBER** at the end of the string. Stored the resultant in **EXPRESSIONS** token using the **$$** variable by subtracting **$3** from **$1**.
- **Line not. 10** – If there is a single number in an **EXPRESSIONS** then the value of the **NUMBER** itself, becomes the value of the **EXPRESSIONS**.
- **Line no 14 –** Called the YYPARSE() function to start parsing the valid lexemes according to our specified grammar.

Here, in our example we have demonstrated only two operations, addition and subtraction. In order to recognize and execute other operations we need to expand and elaborate the required grammar for those operations. For example, to include a multiplication or division operation we can simply add their grammar to the existing one.

Step 3 – 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.]

Taking the input from a file:

In order to take input from a file we need to read the code file and parse it accordingly. For example, to use file input we can change the code of Syntax Analyzer as follows:

```
%{
#include <stdio.h>
%}
%token NUMBER
%%
int yylex();
statements: expressions { printf("= %d\n",$1); }
      ;
expressions: expressions '+' NUMBER { $$ = $1 + $3; }
      | expressions '-' NUMBER { $$ = $1 - $3; }
      | NUMBER { $$ = $1; }
      ;
int main(){
      FILE *file;
      file = fopen("code.c", "r") ;
      if (!file) {
            printf("couldnot open file");
            exit (1);
      }
      else {
            yyin = file;
      }
      yyparse();
}
```

Tasks for LAB 02:

1. Create a Syntax Analyzer that can:
    a. Is the grammar ambiguous? If yes, then transform the grammar so that correct precedence and associativity is followed.
    b. Transform the code so that, the operations can be done on double numbers too.