Lab No. 05 – An Advanced Syntax Analyzer for Custom Syntax

In this experiment we will design a user defined syntax for creating the execution of a "FOR LOOP". In order to execute a for statement we need to first build a lexical analyzer that can accept the keywords and syntaxes of the loop and then build our own syntax analyzer to assign an action for the custom syntax.

Step 1: Build the Lexical Analyzer –

```
1  %{
2  #include "y.tab.h"
3  extern int yylval;
4  %}
5  %%
6  "for" {return FOR;}
7  "print" {return PRINT;}
8  [0-9]+ {yylval = atoi(yytext); return NUMBER;}
9  [a-z] {yylval = atoi(yytext-'a');return TOKEN;}
10 [ \t]+ {} /* ignore whitespace */
11 '\n' {return 0;} /* logical EOF */
12 . {return yytext[0];}
13 %%
```

Code Listing 1 – "LEX.L" – A Lexical Analyzer that accepts certain syntaxes

Explanation of the Lexical Analyzer:

- **Line no. 8** – A generalized regular expression to accept any possible integer numbers. This is required for generating the start and increment values of a standard for loop.
- **Line no. 6-7** – Accepts the syntaxes of **"FOR"** and **"PRINT"** command. Note that, we will print values in our custom syntax using **"PRINT"** instead of **"PRINTF()"**.
- **Line no. 9** – A very simple REGEX that accepts all possible alphabets and sends the index **"TOKEN"** which is obtained by subtracting the ASCII value of 'a' from the accepted character.
- **Line no. 12** – A newline will terminate the code.
- **Line no. 3** – As we will send integer numbers so we just need an integer form of "YYLVAL".

Step 2: Build the Syntax Analyzer –

```
1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5  void yyerror(const char *s){
6        fprintf(stderr,"%s\n",s);
7  }
8  extern FILE *yyin;
9  int yylex();
10 int count_start;
11 int count_end;
12 int var=0;
13 %}
```

```
14 %token NUMBER TOKEN PRINT FOR
15 %%
16 for_statement : for_cond statement { } ‘;’
17               ;
18 for_cond : token '(' expression ')' { int count = 0;}
19          ;
20 expression: TOKEN '=' NUMBER ':' NUMBER {count_start=$3;count_end=$5;
                                            var=$1;
                                           }
21           ;
22 statement: command TOKEN { if(var==$2){
23                               for(var=count_start;var<=count_end;var++){
24                                     printf("%d ",var);
25                               }
26                            }
                             }
27          ;
28 command: PRINT { }
29 ;
30 token: FOR { }
31 ;
32 %%
33 int main(){
34       FILE *file;
35       file = fopen("code.c", "r") ;
36       if (!file) {
37             printf("couldnot open file");
38             exit (1);
39       }
40       else {
41             yyin = file;
42       }
43       yyparse();
44 }
```

Code Listing 2 – “YAC.Y” – The Syntax Analyzer for our custom for loop

Explanation of the Syntax Analyzer:

- **Line no. 8 –** We need this to take file input stream.
- **Line no. 9** – Called the YYLEX function to initiate the input taking process.
- **Line no. 10-12 –** Initialized some variables to store the start value and end value and a counter.
- **Line no. 16-30** – Implemented the simple **CFG** that can recognize any for loops that follow the form as follows **“for(variable = start_value: end_value) print variable”**.
- **Line no. 20** – Initialized the counter start and end values along with the variable.
- **Line no. 24** – Printed each value of the variable in each iteration.
- **Line no 57 –** Called the YYPARSE() function to start parsing the valid lexemes according to our specified grammar.
- **Note that this grammar only recognizes a “FOR” keyword and a “PRINT” command and no other functions of standard C language.**

Step 3 – Preparing the input file:

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

```
for(a=1:15) print a;
```

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 -lm`**
[**"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, "**-ly**" is to add the BISON library and **"-lm"** to add the math library to the compiler.]
iv. **`./output_name`** [To run the output file.]

Tasks for LAB 05:

1. Create a Syntax Analyzer that can use custom syntax for cases like:
    a. Calculate the value and print each iterative value of syntax like -
    "`for(a=1+2:7-2) print a;`". The desired output will be: **3 4 5**
    b. Execute a while loop that has the structure as follows:
    "`while(1<=a<=1+9) print a;`". The desired output will be **1 2 3 4 5 6 7 8 9 10**