**DELHI TECHNOLOGICAL UNIVERSITY**

Bawana Rd, Shahbad Daulatpur Village, Rohini, Delhi, 110042

****

**Compiler Design Lab Book**

Submitted by

Ritvik Nair (2K19/CO/321)

Submitted to

Ms. Dipika Jain

**Table of Contents**

| **No.** | **Aim** | **Page** |
| --- | --- | --- |
| 1 | To understand the Lex and Yacc Tools | 3 |
| 2 | Write a program to eliminate Left Recursion. | 5 |
| 3 | Write a program to perform Left Factoring. | 9 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Experiment 1**

**Aim:** To understand the Lex and Yacc Tools

**Theory:**

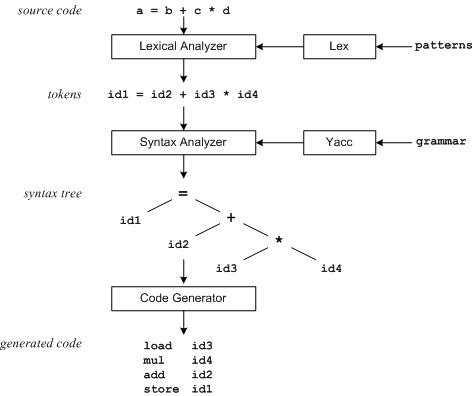
Lex and Yacc tools are used in modern compiler design. Lex stands for Lexical Analysis Generator. Yacc stands for Yet Another Compiler Compiler.

The Lex tool identifies the different patterns(regular expressions) and generates C code for a lexical analyser based on the pattern. The Lexical Analyser then takes in the source code and the Lex output to generate tokens.

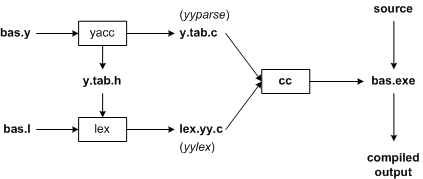
The lex uses a abc.l extension. It outputs a lex.yy.c (yylex) file.

The Yacc tool reads the grammar and generates a parser for the syntax analyser. A syntax tree is generated in the syntax analyser using this and the tokens from Lex and the operators are operated over it using depth-first search.

The yacc uses a abc.y extension. It outputs y.tab.c (yyparse) file and y.tab.h header file. Header is used by lex.



The figure above shows the lex and yacc in a compiler. Lex will read patterns and generate C code for the lexical analyzer. The lexical analyzer will match strings in the input and convert the strings to tokens.



In this image we can see a basic compiler with lex and yacc. 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

cc lex.yy.c y.tab.c -obas.exe # compile/link

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, function yylex, in file lex.yy.c.

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

**Experiment 2**

**Aim:** Write a program to eliminate Left Recursion.

**Theory:**

Left recursion occurs when the leftmost variable of the RHS is the same as the variable of the LHS. Left recursion is eliminated as it is troublesome for top-down parsers.

Eg:

E -> Ea | b

Becomes:

E -> bE’

E’ -> aE’

**Code:**

#include<stdio.h>

#include<string.h>

void LRecursion(char production[][20], int size)

{

int i, j, k, l, n = 0;

for(i = 0; i < size; i++)

{

if(production[i][0] == production[i][3]);

n++;

}

if(n == 0)

{

printf("\nNo Left Recursion");

return;

}

char endPro[size+n][20];

char temp[20];

l = 0;

for(i = 0; i < size; i++)

{

if(production[i][0] == production[i][3])

{

for(j = 0; j < 3; j++)

{

endPro[l][j] = production[i][j];

}

k = j;

endPro[l+1][0] = endPro[l][0];

endPro[l+1][1] = '\'';

endPro[l+1][2] = endPro[l][1];

endPro[l+1][3] = endPro[l][2];

for(j = 4; j < 10; j ++)

{

if(production[i][j] == '|')

{

endPro[l+1][j+1] = endPro[l+1][0];

endPro[l+1][j+2] = endPro[l+1][1];

endPro[l+1][j+3] = '|';

endPro[l+1][j+4] = 'e';

break;

}

endPro[l+1][j] = production[i][j];

}

printf("%c",endPro[l+1][j+4]);

j++;

while(production[i][j] != '\0')

{

endPro[l][k] = production[i][j];

k++;

j++;

}

endPro[l][k] = endPro[l][0];

endPro[l][k+1] = '\'';

l+=2;

}

else

{

j = 0;

k = 0;

while(production[i][j] != '\0')

{

endPro[l][k] = production[i][j];

k++;

j++;

}

l++;

}

}

printf("\nOutput:\n");

for(i = 0; i < size+n; i++)

{

printf("%s\n", endPro[i]);

}

}

int main () {

int size, i;

printf("Enter no of Productions: ");

scanf("%d", &size);

char production[size][20];

printf("\nEnter the Productions:\n");

for(i = 0; i < size; i++)

{

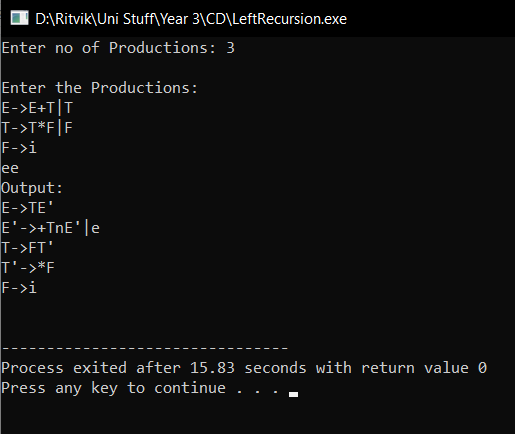
scanf("%s", production[i]);

}

LRecursion(production, size);

}

**Output:**



**Experiment 3**

**Aim:** Write a program to perform Left Factoring.

**Theory:**

Left factoring is a method of removing right recursion. In left factoring the RHS has the same value at the start for multiple productions of the same LHS. To remove this a new LHS variable is created with its productions pointing to the endings of the repeating RHS productions. The repetition is removed from the original production and the new variable is added to the end of the RHS.

**Code:**

#include<stdio.h>

#include<string.h>

int main()

{

char gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];

int i,j=0,k=0,l=0,pos;

printf("Enter Production : A->");

gets(gram);

for(i=0;gram[i]!='|';i++,j++)

part1[j]=gram[i];

part1[j]='\0';

for(j=++i,i=0;gram[j]!='\0';j++,i++)

part2[i]=gram[j];

part2[i]='\0';

for(i=0;i<strlen(part1)||i<strlen(part2);i++)

{

if(part1[i]==part2[i])

{

modifiedGram[k]=part1[i];

k++;

pos=i+1;

}

}

for(i=pos,j=0;part1[i]!='\0';i++,j++)

{

newG

ram[j]=part1[i];

}

newGram[j++]='|';

for(i=pos;part2[i]!='\0';i++,j++)

{

newGram[j]=part2[i];

}

modifiedGram[k]='X';

modifiedGram[++k]='\0';

newGram[j]='\0';

printf("\n A->%s",modifiedGram);

printf("\n X->%s\n",newGram);

}

**Output:**

