

COMPILER DESIGN LAB

BTCSE-602

2020-2024



Submitted to: Mr. Tabrej Ahmed Khan Sir

[signature]

Submitted By: Snigdha Singh

Enroll: 2020-310-217

Std: B.TECH CSE,6TH SEM

Sec: D

**Jamia Hamdard
Department Of Computer Science & Engineering
School Of Engineering Sciences & Technology
New Delhi-110062**

S.No.	ASSIGNMENT	REMARKS
1.	Write a program to recognize keywords and identifiers.	
2.	Write a program to print the total characters, letters, digits, line, special character in a given input file.	
3.	Write a program to compute first and follow of given grammar in C.	
4.	Write short note on YACC and BISON.	
5.	Write a YACC program of a calculator performing “+,-,*,/” operations.	
6.	Write a Program to generate parse tree.	
7.	Write a Program to eliminate left factoring from grammar	

ASSIGNMENT-1

Q1. Program to generate a LEXICAL ANALYSER.

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

%%
int|float|char|break {printf("This is Keyword");}
[A-Za-z][A-Za-z0-9]* {printf("This is Identifier");}
[0-9]+ {printf("This is digit");}
\".*\" {printf("This is a String");}
. {printf("Not Valid");}
%%
int main()
{
yylex();
}
```

OUTPUT

```
int
This is Keyword
float
This is Keyword
hi
This is Identifier
123
This is digit
"fml"
This is a String
```

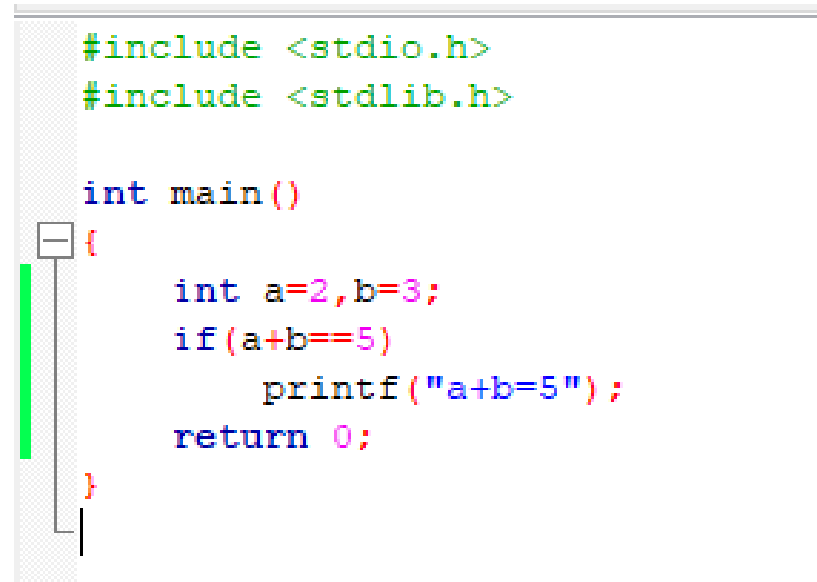
ASSIGNMENT-2

Q2. Write a program to print the total characters, letters, digits, line, special character in a given input file.

```
%{  
#include<stdio.h>  
  
%}  
  
identifier[A-Za-z][A-Za-z0-9]*  
  
  
%%  
  
#.*\> printf("%s is a preprocessor directive\n",yytext);  
  
int|char|float|while|for|do|if|else|switch|break|default|return|printf printf("%s is a  
keyword\n",yytext);  
  
{identifier}\(\) printf("%s is a function\n",yytext);  
  
\{ printf("begining of a block\n");  
  
\} printf("end of a block");  
  
\\+|\\-|\\*|\\/ printf("%s is arithmetic operator\n",yytext);  
  
= printf("%s is assignment operator\n",yytext);  
  
\\<=|\\>=|\\<|\\>|\\== printf("%s is a relational operator\n",yytext);  
  
[0-9]+ printf("%s is number\n",yytext);  
  
[A-Za-z][A-Za-z0-9]* printf("%s is an identifier",yytext);  
  
{identifier}\\([[0-9]*\\)]? printf("%s is an array ",yytext);  
  
\\\".*\\\" printf("%s is a string",yytext);  
  
\\( ECHO; printf("\\n");  
  
\\(\\;)? ECHO; printf("\\n");  
  
.;  
  
%%  
  
  
int main(int argc,char**argv){
```

```
if(argc<2){
    printf("usage:%s input\n",argv[0]);
    return 1;
}
yyin=fopen(argv[1],"r");
if(!yyin){
    printf("could not open %s\n",argv[1]);
    return 1;
}
yylex();
return 0;
}
```

C - code



```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a=2,b=3;
    if(a+b==5)
        printf("a+b=5");
    return 0;
}
```

OUTPUT

```
#include <stdio.h> is a preprocessor directive
#include <stdlib.h> is a preprocessor directive

int is a keyword
main() is a function

begining of a block

    int is a keyword
    a is an identifier= is assignment operator
    2 is number
    ,b is an identifier= is assignment operator

    if is a keyword
    (
    a is an identifier+ is arithmetic operator
    b is an identifier== is a relational operator
    5 is number
    )

    printf is a keyword
    (
    "a+b=5" is a string);

    return is a keyword

end of a block
shaid_ahmed40955K70P_UWV6777: /opt/d/ubaid/study/cplusplus/assignment_1/page_26
```

ASSIGNMENT-3

Q3. Write a program to compute first and follow of given grammar in C

```
// C program to calculate the First and
```

```
// Follow sets of a given grammar
```

```
#include <ctype.h>
```

```
#include <stdio.h>
```

```
#include <string.h>
```

```
// Functions to calculate Follow
```

```
void followfirst(char, int, int);
```

```
void follow(char c);
```

```
// Function to calculate First
```

```
void findfirst(char, int, int);
```

```
int count, n = 0;
```

```
// Stores the final result
```

```
// of the First Sets
```

```
char calc_first[10][100];
```

```
// Stores the final result
```

```
// of the Follow Sets
```

```
char calc_follow[10][100];
```

```
int m = 0;
```

```

// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char** argv)
{
    int jm = 0;
    int km = 0;
    int i, choice;
    char c, ch;
    count = 8;

    // The Input grammar
    strcpy(production[0], "X=TnS");
    strcpy(production[1], "X=Rm");
    strcpy(production[2], "T=q");
    strcpy(production[3], "T=#");
    strcpy(production[4], "S=p");
    strcpy(production[5], "S=#");
    strcpy(production[6], "R=om");
    strcpy(production[7], "R=ST");

    int kay;
    char done[count];

```



```
int ptr = -1;
```

```
// Initializing the calc_first array
```

```
for (k = 0; k < count; k++) {  
    for (kay = 0; kay < 100; kay++) {  
        calc_first[k][kay] = '!';  
    }  
}
```

```
int point1 = 0, point2, xxx;
```

```
for (k = 0; k < count; k++) {
```

```
    c = production[k][0];
```

```
    point2 = 0;
```

```
    xxx = 0;
```

```
    // Checking if First of c has
```

```
    // already been calculated
```

```
    for (kay = 0; kay <= ptr; kay++)
```

```
        if (c == done[kay])
```

```
            xxx = 1;
```

```
    if (xxx == 1)
```

```
        continue;
```

```
    // Function call
```

```
    findfirst(c, 0, 0);
```

```
    ptr += 1;
```

```

// Adding c to the calculated list
done[ptr] = c;
printf("\n First(%c) = { ", c);
calc_first[point1][point2++] = c;

// Printing the First Sets of the grammar
for (i = 0 + jm; i < n; i++) {
    int lark = 0, chk = 0;

    for (lark = 0; lark < point2; lark++) {

        if (first[i] == calc_first[point1][lark]) {
            chk = 1;
            break;
        }
    }
    if (chk == 0) {
        printf("%c, ", first[i]);
        calc_first[point1][point2++] = first[i];
    }
}
printf("}\n");
jm = n;
point1++;
}
printf("\n");

```

```
printf("-----"
```

```
    "\n\n");
```

```
char donee[count];
```

```
ptr = -1;
```

```
// Initializing the calc_follow array
```

```
for (k = 0; k < count; k++) {
```

```
    for (kay = 0; kay < 100; kay++) {
```

```
        calc_follow[k][kay] = '!';
```

```
    }
```

```
}
```

```
point1 = 0;
```

```
int land = 0;
```

```
for (e = 0; e < count; e++) {
```

```
    ck = production[e][0];
```

```
    point2 = 0;
```

```
    xxx = 0;
```

```
    // Checking if Follow of ck
```

```
    // has already been calculated
```

```
    for (kay = 0; kay <= ptr; kay++)
```

```
        if (ck == donee[kay])
```

```
            xxx = 1;
```

```
    if (xxx == 1)
```

```
        continue;
```

```
    land += 1;
```

```

// Function call
follow(ck);

ptr += 1;


// Adding ck to the calculated list
donee[ptr] = ck;

printf(" Follow(%c) = { ", ck);

calc_follow[point1][point2++] = ck;


// Printing the Follow Sets of the grammar
for (i = 0 + km; i < m; i++) {
    int lark = 0, chk = 0;

    for (lark = 0; lark < point2; lark++) {
        if (f[i] == calc_follow[point1][lark]) {
            chk = 1;
            break;
        }
    }

    if (chk == 0) {
        printf("%c, ", f[i]);

        calc_follow[point1][point2++] = f[i];
    }
}

printf(" }\n\n");

km = m;

point1++;

```

```
}  
}
```

```
void follow(char c)
```

```
{
```

```
    int i, j;
```

```
    // Adding "$" to the follow
```

```
    // set of the start symbol
```

```
    if (production[0][0] == c) {
```

```
        f[m++] = '$';
```

```
    }
```

```
    for (i = 0; i < 10; i++) {
```

```
        for (j = 2; j < 10; j++) {
```

```
            if (production[i][j] == c) {
```

```
                if (production[i][j + 1] != '\0') {
```

```
                    // Calculate the first of the next
```

```
                    // Non-Terminal in the production
```

```
                    followfirst(production[i][j + 1], i,
```

```
                        (j + 2));
```

```
                }
```

```
            if (production[i][j + 1] == '\0'
```

```
                && c != production[i][0]) {
```

```
                // Calculate the follow of the
```

```
                // Non-Terminal in the L.H.S. of the
```

```
                // production
```

```

        follow(production[i][0]);
    }
}
}
}
}
}

```

```
void findfirst(char c, int q1, int q2)
{
    int j;

    // The case where we
    // encounter a Terminal
    if (!(isupper(c))) {
        first[n++] = c;
    }

    for (j = 0; j < count; j++) {
        if (production[j][0] == c) {
            if (production[j][2] == '#') {
                if (production[q1][q2] == '\0')
                    first[n++] = '#';
                else if (production[q1][q2] != '\0'
                        && (q1 != 0 || q2 != 0)) {
                    // Recursion to calculate First of New
                    // Non-Terminal we encounter after
                    // epsilon
                    findfirst(production[q1][q2], q1,
```

```

        (q2 + 1));
    }
    else
        first[n++] = '#';
    }
    else if (!isupper(production[j][2])) {
        first[n++] = production[j][2];
    }
    else {
        // Recursion to calculate First of
        // New Non-Terminal we encounter
        // at the beginning
        findfirst(production[j][2], j, 3);
    }
}
}
}

```

```

void followfirst(char c, int c1, int c2)

```

```

{
    int k;

    // The case where we encounter
    // a Terminal
    if (!isupper(c))
        f[m++] = c;
    else {

```

```

int i = 0, j = 1;
for (i = 0; i < count; i++) {
    if (calc_first[i][0] == c)
        break;
}

// Including the First set of the
// Non-Terminal in the Follow of
// the original query
while (calc_first[i][j] != '!') {
    if (calc_first[i][j] != '#') {
        f[m++] = calc_first[i][j];
    }
    else {
        if (production[c1][c2] == '\0') {
            // Case where we reach the
            // end of a production
            follow(production[c1][0]);
        }
        else {
            // Recursion to the next symbol
            // in case we encounter a "#"
            followfirst(production[c1][c2], c1,
                c2 + 1);
        }
    }
}
j++;

```



```
    }  
  }  
}
```

OUTPUT

```
First(X) = { q, n, o, p, #, }
```

```
First(T) = { q, #, }
```

```
First(S) = { p, #, }
```

```
First(R) = { o, p, q, #, }
```

```
Follow(X) = { $, }
```

```
Follow(T) = { n, m, }
```

```
Follow(S) = { $, q, m, }
```

```
Follow(R) = { m, }
```

ASSIGNMENT-4

Q4) Write short note on YACC and BISON.

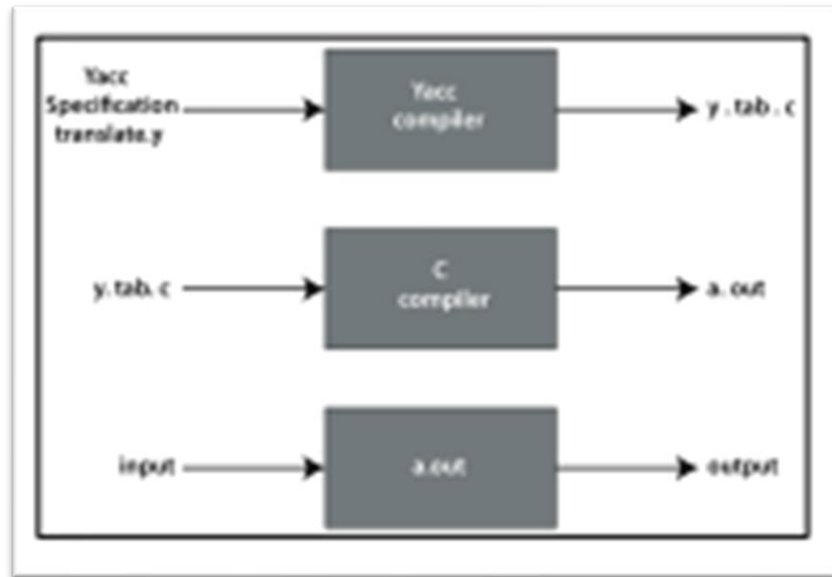
The `yyparse()` function reads the tokens, performs the actions and returns to the main when it reaches the end of the file or when an error occurs. It return 00 if the parsing is successful, and 11 if it is unsuccessful.

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 definition 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.

For Compiling YACC program:

- Write lex program in a file `file.l` and yacc in a file `file.y`
- Open Terminal and Navigate to the Directory where you have saved the files.
- Type `lex file.l`
- Type `yacc file.y`
- Type `cc.lex.yy.c.y.tab.h -ll`



BISON

- This is a general-purpose parser generator used to convert a grammar description for a LALR(1) context-free grammar into a parser which is written in C and parses the grammar.
- Bison parsers are bottom-up parsers that is they start from the leaf nodes of a tree and work their way upward until a root node is arrived at.
- It is an upgrade from yacc and one of the most commonly used LALR tools.

WORKING OF BISON

- Bison is built to be used with C code and thus generates code in C.
- First we give it a grammar specification file which we name using the .y extension then invoke bison on the .y file and it in turn creates y.tab.y and y.tab.c files with hundreds of C code implementing a LALR(1) parser for the specified grammar together with code for the specified actions.
- This file also provides an extern function yyparser that parses a valid sentence. We compile it as usual using C compile.
- The generated parser by default will read from standard input and write to standard output.
- Here are the steps in list form:

- Here are the steps in list form:
 1. Bison textFile.y-ay.tab.c file containing C code for a parser is created.
 2. gcc -c y.tab.c – compile parser code normally using gcc C compiler.
 3. Gcc -o parse y.tab.o lex.yy.o -ll -ly – we link the parser, scanner and libraries.
 4. ./parse – we execute the parser that reads from standard input.

The bison input file format is similar to the yacc file format therefore we shall not repeat it here.

ASSIGNMENT-5

Q5. Write a YACC program of a calculator performing simple operations.

YACC CODE

```
%{
    /* Definition section */
    #include<stdio.h>
    int flag=0;
}%

%token NUMBER

%left '+' '-'
%left '*' '/' '%'
%left '(' ')'

/* Rule Section */
%%

ArithmeticExpression: E{

    printf("\nResult=%d\n", $$);

    return 0;

};
E: E '+' E {$$=$1+$3;}
| E '-' E {$$=$1-$3;}
| E '*' E {$$=$1*$3;}
| E '/' E {$$=$1/$3;}
| E '%' E {$$=$1%$3;}
| '(' E ')' {$$=$2;}
| NUMBER {$$=$1;}

;
```

```
%%
```

```
//driver code
```

```
void main()
```

```
{
```

```
    printf("\nEnter Any Arithmetic Expression which  
           can have operations Addition,  
           Subtraction, Multiplication, Division,  
           Modulus and Round brackets:\n");
```

```
    yyparse();
```

```
    if(flag==0)
```

```
        printf("\nEnter arithmetic expression is Valid\n\n");
```

```
}
```

```
void yyerror()
```

```
{
```

```
    printf("\nEnter arithmetic expression is Invalid\n\n");
```

```
    flag=1;
```

```
}
```

LEX CODE

```
%{
```

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

```
    #include<stdio.h>
```

```
    #include "y.tab.h"
```

```
    extern int yylval;
```

```
%}
```

```
/* Rule Section */
```

```
%%
```

```
[0-9]+ {
```

```
    yylval=atoi(yytext);
```

```
    return NUMBER;
```

```
}
```

```
[\t] ;
```

```
[\n] return 0;
```

```
. return yytext[0];
```

```
%%
```

```
int yywrap()  
{  
    return 1;  
}
```

```
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:  
4+5  
Result=9  
Entered arithmetic expression is Valid  
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out  
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:  
10-5  
Result=5  
Entered arithmetic expression is Valid  
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out  
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:  
10+5-  
Entered arithmetic expression is Invalid  
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out  
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:  
10/5  
Result=2  
Entered arithmetic expression is Valid  
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out  
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:  
(2+5)*3  
Result=21  
Entered arithmetic expression is Valid  
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out  
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:  
(2*4)+  
Entered arithmetic expression is Invalid
```

Assignment-6

Q6)WAP to generate a parse tree

```
#include <iostream>

#include <cstdlib>

#include <stack>

#include <sstream>

#include <string>

#include <vector>

#include <algorithm>

using namespace std;
```

```
class BinaryTree {

private:

    string key;

    BinaryTree *leftChild;

    BinaryTree *rightChild;

public:

    BinaryTree(string rootObj){

        this->key = rootObj;

        this->leftChild = NULL;

        this->rightChild = NULL;

    }

    void insertLeft(string newNode){

        if (this->leftChild == NULL){

            this->leftChild = new BinaryTree(newNode);
```



```
    }  
    else {  
        BinaryTree *t = new BinaryTree(newNode);  
        t->leftChild = this->leftChild;  
        this->leftChild = t;  
    }  
}
```

```
void insertRight(string newNode){  
    if (this->rightChild == NULL){  
        this->rightChild = new BinaryTree(newNode);  
    }  
    else {  
        BinaryTree *t = new BinaryTree(newNode);  
        t->rightChild = this->rightChild;  
        this->rightChild = t;  
    }  
}
```

```
BinaryTree *getRightChild(){  
    return this->rightChild;  
}
```

```
BinaryTree *getLeftChild(){  
    return this->leftChild;  
}
```

```
void setRootVal(string obj){  
    this->key = obj;  
}
```

```

    }

    string getRootVal(){
        return this->key;
    }
};

BinaryTree *buildParseTree(string fpexp){
    string buf;
    stringstream ss(fpexp);
    vector<string> fplist;
    while (ss >> buf){
        fplist.push_back(buf);
    }
    stack<BinaryTree*> pStack;
    BinaryTree *eTree = new BinaryTree("");
    pStack.push(eTree);
    BinaryTree *currentTree = eTree;

    string arr[] = {"+", "-", "*", "/"};
    vector<string> vect(arr,arr+(sizeof(arr)/ sizeof(arr[0])));

    string arr2[] = {"+", "-", "*", "/", ""};
    vector<string> vect2(arr2,arr2+(sizeof(arr2)/ sizeof(arr2[0])));

    for (unsigned int i = 0; i<fplist.size(); i++){

        if (fplist[i] == "("){
            currentTree->insertLeft("");

```

```

    pStack.push(currentTree);
    currentTree = currentTree->getLeftChild();
}

else if (find(vect.begin(), vect.end(), fplist[i]) != vect.end()){
    currentTree->setRootVal(fplist[i]);
    currentTree->insertRight("");
    pStack.push(currentTree);
    currentTree = currentTree->getRightChild();
}

else if (fplist[i] == ""){
    currentTree = pStack.top();
    pStack.pop();
}

else if (find(vect2.begin(), vect2.end(), fplist[i]) == vect2.end()) {
    try {
        currentTree->setRootVal(fplist[i]);
        BinaryTree *parent = pStack.top();
        pStack.pop();
        currentTree = parent;
    }

    catch (string ValueError ){
        cerr <<"token " << fplist[i] << " is not a valid integer"<<endl;
    }
}
}

```

```
    return eTree;
}

void postorder(BinaryTree *tree){
    if (tree != NULL){
        postorder(tree->getLeftChild());
        postorder(tree->getRightChild());
        cout << tree->getRootVal() << endl;
    }
}

int main() {

    BinaryTree *pt = buildParseTree("( ( 10 + 5 ) * 3 )");

    postorder(pt);

    return 0;
}
```

OUTPUT

```
10
5
+
3
*
```

Assignment-7

Q7)WAP to eliminate left factoring from a grammar

```
#include<iostream>

#include<string>

using namespace std;

int main()

{ string ip,op1,op2,temp;

  int sizes[10] = {};

  char c;

  int n,j,l;

  cout<<"Enter the Parent Non-Terminal : ";

  cin>>c;

  ip.push_back(c);

  op1 += ip + "'->";

  ip += "->";

  op2+=ip;

  cout<<"Enter the number of productions : ";

  cin>>n;

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

  { cout<<"Enter Production "<<i+1<<" : ";

    cin>>temp;

    sizes[i] = temp.size();

    ip+=temp;

    if(i!=n-1)

      ip += "|";

  }

  cout<<"Production Rule : "<<ip<<endl;

  for(int i=0,k=3;i<n;i++)
```

```

{
    if(ip[0] == ip[k])
    {
        cout<<"Production "<<i+1<<" has left recursion."<<endl;
        if(ip[k] != '#')
        {
            for(l=k+1;l<k+sizes[i];l++)
                op1.push_back(ip[l]);
            k=l+1;
            op1.push_back(ip[0]);
            op1 += "\\'|";
        }
    }
    else
    {
        cout<<"Production "<<i+1<<" does not have left recursion."<<endl;
        if(ip[k] != '#')
        {
            for(j=k;j<k+sizes[i];j++)
                op2.push_back(ip[j]);
            k=j+1;
            op2.push_back(ip[0]);
            op2 += "\\'|";
        }
        else
        {
            op2.push_back(ip[0]);
            op2 += "\\''";
        }
    }
}
}
}

```

```
op1 += "#";  
cout<<op2<<endl;  
cout<<op1<<endl;  
return 0;}
```

OUTPUT

```
/tmp/KOC0Vh8MJH.o  
Enter the Parent Non-Terminal : U  
Enter the number of productions : 3  
Enter Production 1 : Ub  
Enter Production 2 : aid  
Enter Production 3 : #  
Production Rule : U->Ub|aid|#  
Production 1 has left recursion.  
Production 2 does not have left recursion.  
Production 3 does not have left recursion.  
U->aidU'|U'  
U' ->bU'|#  
|
```