Compiler Design Lab

CSS651

Group – 2

Members

Saharsh Ananta Jaiswal – 18CS8061

Pudi Pavan Kumar – 18CS8062

Ayesha Uzma – 18CS8063

Rithik Sureka – 18CS8064

Ravupalli Harsha Vardhan – 18CS8065

Assignment: 3

Implement one Operator Precedence Parser assuming conventional precedence of operators for the construction of precedence table.

Software Used: Code::Blocks(IDE)

Theory:

An operator-precedence parser is a simple shift-reduce parser that is capable of parsing a subset of LR(1) grammars. More precisely, the operator-precedence parser can parse all LR(1) grammars where two consecutive non-terminals and epsilon never appear in the right-hand side of any rule.

For example, most calculators use operator precedence parsers to convert from the human-readable infix notation relying on order of operations to a format that is optimized for evaluation such as Reverse Polish notation (RPN).Edsger Dijkstra's shunting yard algorithm is commonly used to implement operator precedence parsers.

There are two techniques for figuring out what precedence relations should hold between a couple of terminals:

- Utilize the ordinary associativity and precedence of the operator.
- The second technique for choosing operator-precedence relations is first to build an unambiguous sentence structure for the language, punctuation that mirrors the right associativity and precedence in its parse trees.

This parser depends on the accompanying three precedence relations: $\lessdot$, $\doteq$, $\gtrdot$

a $\lessdot$ b This implies a "yields precedence to" b.

a $\gtrdot$ b This implies an "overshadows" b.

a $\doteq$ b This implies a "has precedence as" b

Disadvantages:

1. It is hard to handle tokens like the minus sign (-), which has two different precedence (depending on whether it is unary or binary).
2. It is applicable only to a small class of grammars.

Code:

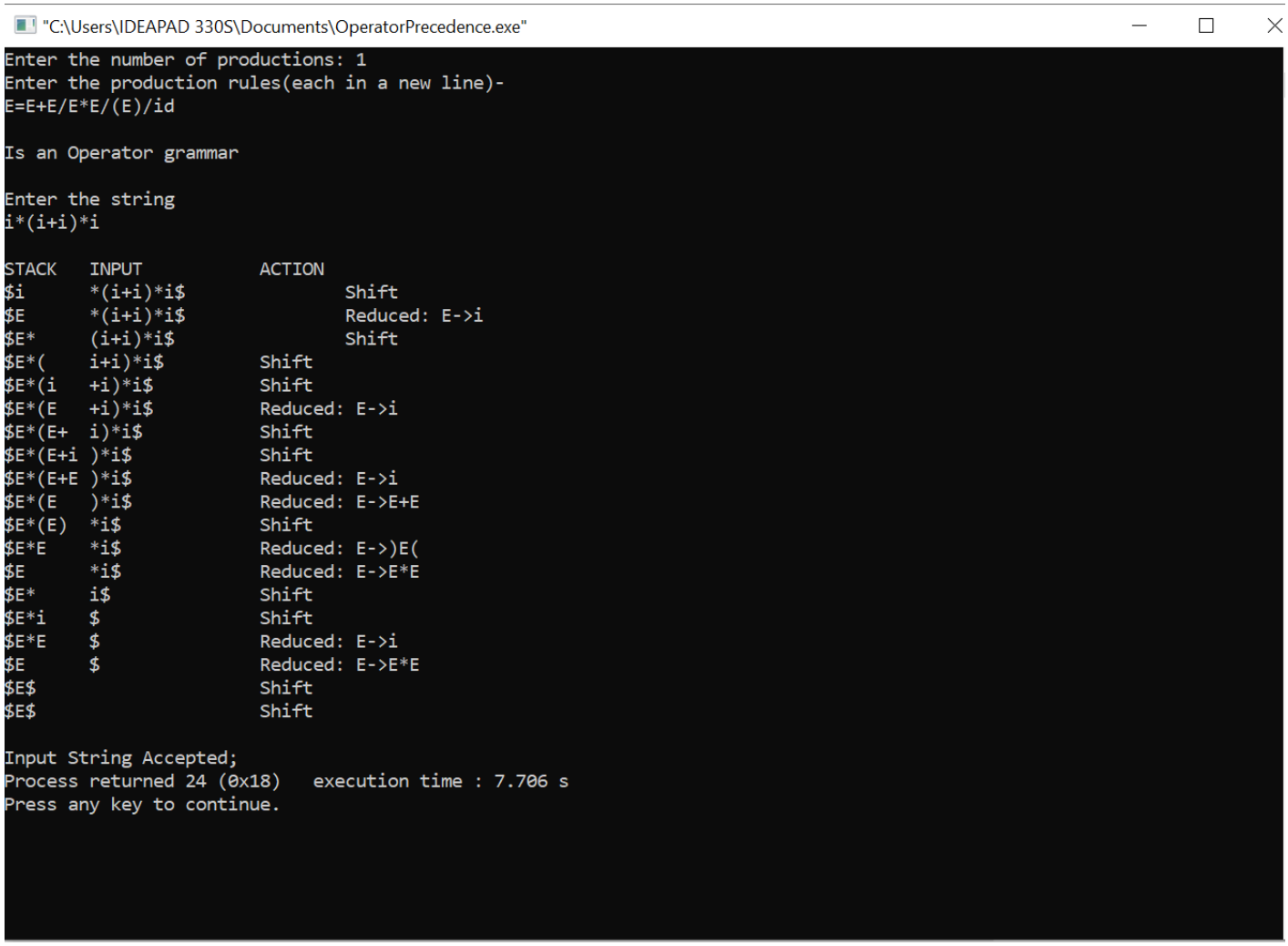
Start here | OperatorPrecedence.c

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

char *input;
int i=0,n;
char lasthandle[6],stack[50],handles[][5]={")E(","E*E","E+E","i","E^E"};
char gram[10][20];
//(E) becomes )E( when pushed to stack

int top=0,l;
char prec[9][9]={

                            /*input*/

            /*stack    +    -    *    /    ^    i    (    )    $  */

            /*  + */  '>', '>','<','<','<','<','<','>','>',

            /*  - */  '>', '>','<','<','<','<','<','>','>',

            /*  * */  '>', '>','>','>','<','<','<','>','>',

            /*  / */  '>', '>','>','>','<','<','<','>','>',

            /*  ^ */  '>', '>','>','>','<','<','<','>','>',

            /*  i */  '>', '>','>','>','>','e','e','>','>',

            /*  ( */  '<', '<','<','<','<','<','<','>','e',

            /*  ) */  '>', '>','>','>','>','e','e','>','>',

            /*  $ */  '<', '<','<','<','<','<','<','<','>',

                };

int getindex(char c)
{
switch(c)
    {
    case '+':return 0;
    case '-':return 1;
    case '*':return 2;
    case '/':return 3;
    case '^':return 4;
    case 'i':return 5;
    case '(':return 6;
    case ')':return 7;
    case '$':return 8;
    }
}

// function f to exit from the loop
// if given condition is not true
void f()
{
    printf("\nNot operator grammar\n");
    exit(0);
}

int shift()
{
stack[++top]=*(input+i++);
stack[top+1]='\0';
```

```
}


int reduce()
{
int i,len,found,t;
for(i=0;i<5;i++)//selecting handles
    {
    len=strlen(handles[i]);
    if(stack[top]==handles[i][0]&&top+1>=len)
        {
        found=1;
        for(t=0;t<len;t++)
            {
            if(stack[top-t]!=handles[i][t])
                {
                found=0;
                break;
                }
            }
        if(found==1)
            {
            stack[top-t+1]='E';
            top=top-t+1;
            strcpy(lasthandle,handles[i]);
            stack[top+1]='\0';
            return 1;//successful reduction
            }
        }
   }
return 0;
}



void dispstack()
{
int j;
for(j=0;j<=top;j++)
    printf("%c",stack[j]);

}



void dispinput()
{
int j;
for(j=i;j<l;j++)
    printf("%c",*(input+j));
}

void check_operatorGrammar()
{
    // Here using flag variable set to 0, considers grammar is not operator grammar
    int i, j = 2, flag = 0;
    char c;
    for (i = 0; i < n; i++) {
        c = gram[i][2];

        while (c != '\0') {

            if (gram[i][3] == '+' || gram[i][3] == '-'
                || gram[i][3] == '*' || gram[i][3] == '/')
                flag = 1;
             else {
                flag = 0;
```

```
                f();
            }

            if (c == '$') {
                flag = 0;
                f();
            }
            c = gram[i][++j];
        }
    }
    if (flag == 1)
        printf("\nIs an Operator grammar\n");
}


void main()
{
    int j;
    // taking number of productions from user
    printf("Enter the number of productions: ");
    scanf("%d", &n);
    printf("Enter the production rules(each in a new line)- \n");
    for (int i = 0; i < n; i++)
        scanf("%s", gram[i]);
    check_operatorGrammar();
    input=(char*)malloc(50*sizeof(char));
    printf("\nEnter the string\n");
    scanf("%s",input);
    input=strcat(input,"$");
    l=strlen(input);
    strcpy(stack,"$");
    printf("\nSTACK \tINPUT\t\tACTION");
    while(i<=l)
        {
        shift();
        printf("\n");
        dispstack();
        printf("\t");
        dispinput();
        printf("\t\tShift");
        if(prec[getindex(stack[top])][getindex(input[i])]=='>')
            {
            while(reduce())
                {
                printf("\n");
                dispstack();
                printf("\t");
                dispinput();
                printf("\t\tReduced: E->%s",lasthandle);
                }
            }
        }

    if(strcmp(stack,"$E$")==0)
        printf("\n\nInput String Accepted;");
    else
        printf("\nNot Accepted;");
    }

```

Output:


"C:\Users\IDEAPAD 330S\Documents\OperatorPrecedence.exe"
Enter the number of productions: 1
Enter the production rules(each in a new line)-
E=E+E/E*E/(E)/id
Is an Operator grammar
Enter the string
i*(i+i)*i
STACK INPUT ACTION
$i *(i+i)*i$ Shift
$E *(i+i)*i$ Reduced: E->i
$E* (i+i)*i$ Shift
$E*(i+i)*i$ Shift
$E*(i +i)*i$ Shift
$E*(E +i)*i$ Reduced: E->i
$E*(E+ i)*i$ Shift
$E*(E+i)*i$ Shift
$E*(E+E)*i$ Reduced: E->i
$E*(E)*i$ Reduced: E->E+E
$E*(E) *i$ Shift
$E*E *i$ Reduced: E->)E(
$E *i$ Reduced: E->E*E
$E* i$ Shift
$E*i $ Shift
$E*E $ Reduced: E->i
$E $ Reduced: E->E*E
E Shift
E Shift
Input String Accepted;
Process returned 24 (0x18) execution time : 7.706 s
Press any key to continue.

Assignment: 4

Implement one LL(1) parser. The grammar for the parser is fixed and the input is/are the sentence(s) to be parsed and the output is the sequence of production used.

Software Used: Code::Blocks(IDE)

Theory:

A top-down parser that uses a one-token look-ahead is called an LL(1) parser.

- The first L indicates that the input is read from left to right.
- The second L says that it produces a left-to-right derivation.
- And the 1 says that it uses one look-ahead token. (Some parsers look ahead at the next 2 tokens, or even more than that.)

To construct the Parsing table, we have two functions:

1: First(): If there is a variable, and from that variable if we try to drive all the strings then the beginning *Terminal Symbol* is called the first.
2: Follow(): What is the *Terminal Symbol* which follow a variable in the process of derivation.
Now, after computing the First and Follow set for each *Non-Terminal symbol* we have to construct the Parsing table. In the table Rows will contain the Non-Terminals and the column will contain the Terminal Symbols.
All the **Null Productions** of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of First set.

Following are some of the assumptions I have made :-

1. Epsilon is represented by '#'.
2. Productions are of the form A=B , where 'A' is a single Non-Terminal and 'B' can be any combination of Terminals and Non- Terminals.
3. The L.H.S. of the first production rule is the start symbol.
4. Grammer is not left recursive.
5. Each production of a non terminal is entered on a different line.
6. Only Upper Case letters are Non-Terminals and everything else is a terminal.
7. Do not use '!' or '$' as they are reserved for special purposes.
8. All input Strings have to end with a '$'

Code:

Start here | LL-1 Parser.c

```c
#include<stdio.h>
#include<ctype.h>
#include<string.h>

void followfirst(char , int , int);
void findfirst(char , int , int);
void follow(char c);

int count,n=0;
char calc_first[10][100];
char calc_follow[10][100];
int m=0;
char production[10][10], first[10];
char f[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;
    printf("How many productions ? :");
    scanf("%d",&count);
    printf("\nEnter %d productions in form A=B where A and B are grammar symbols :\n\n",count);
    for(i=0;i<count;i++)
    {
        scanf("%s%c",production[i],&ch);
    }
    int kay;
    char done[count];
    int ptr = -1;
    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;
        for(kay = 0; kay <= ptr; kay++)
            if(c == done[kay])
                xxx = 1;
        if (xxx == 1)
            continue;
        findfirst(c,0,0);
        ptr+=1;
        done[ptr] = c;
        printf("\n First(%c)= { ",c);
        calc_first[point1][point2++] = c;
        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;
    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;
        for(kay = 0; kay <= ptr; kay++)
            if(ck == donee[kay])
                xxx = 1;
        if (xxx == 1)
            continue;
        land += 1;
        follow(ck);
        ptr+=1;
        donee[ptr] = ck;
        printf(" Follow(%c) = { ",ck);
        calc_follow[point1][point2++] = ck;
        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++;
    }
    char ter[10];
    for(k=0;k<10;k++){
        ter[k] = '!';
    }
    int ap,vp,sid = 0;
    for(k=0;k<count;k++){
        for(kay=0;kay<count;kay++){
            if(!isupper(production[k][kay]) && production[k][kay]!= '#' &&
                production[k][kay] != '=' && production[k][kay] != '\0')
            {
                vp = 0;
                for(ap = 0;ap < sid; ap++)
                {
                    if(production[k][kay] == ter[ap])
                    {
                        vp = 1;
                        break;
                    }
                }
                if(vp == 0){
                    ter[sid] = production[k][kay];
```

```
                    sid ++;
                }
            }
        }
    }
    ter[sid] = '$';
    sid++;
    printf("\n\t\t\t\t\t\t\t\t The LL(1) Parsing Table for the above grammer :-");
    printf("\n\t\t\t\t\t\t\t^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
    printf("\n\t\t\t==========================================================================================");
    printf("=========================================\n");
    printf("\t\t\t\t|\t");
    for(ap = 0;ap < sid; ap++){
        printf("%c\t\t",ter[ap]);
    }
    printf("\n\t\t\t==========================================================================================");
    printf("=========================================\n");
    char first_prod[count][sid];
    for(ap=0;ap<count;ap++){
        int destiny = 0;
        k = 2;
        int ct = 0;
        char tem[100];
        while(production[ap][k] != '\0'){
            if(!isupper(production[ap][k])){
                tem[ct++] = production[ap][k];
                tem[ct++] = '_';
                tem[ct++] = '\0';
                k++;
                break;
            }
            else{
                int zap=0;
                int tuna = 0;
                for(zap=0;zap<count;zap++){
                    if(calc_first[zap][0] == production[ap][k]){
                        for(tuna=1;tuna<100;tuna++){
                            if(calc_first[zap][tuna] != '!'){
                                tem[ct++] = calc_first[zap][tuna];
                            }
                            else
                                break;
                        }
                    break;
                    }
                }
                tem[ct++] = '_';
            }
            k++;
        }
        int zap = 0,tuna;
        for(tuna = 0;tuna<ct;tuna++){
            if(tem[tuna] == '#'){
                zap = 1;
            }
            else if(tem[tuna] == '_'){
                if(zap == 1){
                    zap = 0;
                }
                else
                    break;
            }
            else{
                first_prod[ap][destiny++] = tem[tuna];
            }
        }
    }
    char table[land][sid+1];
```

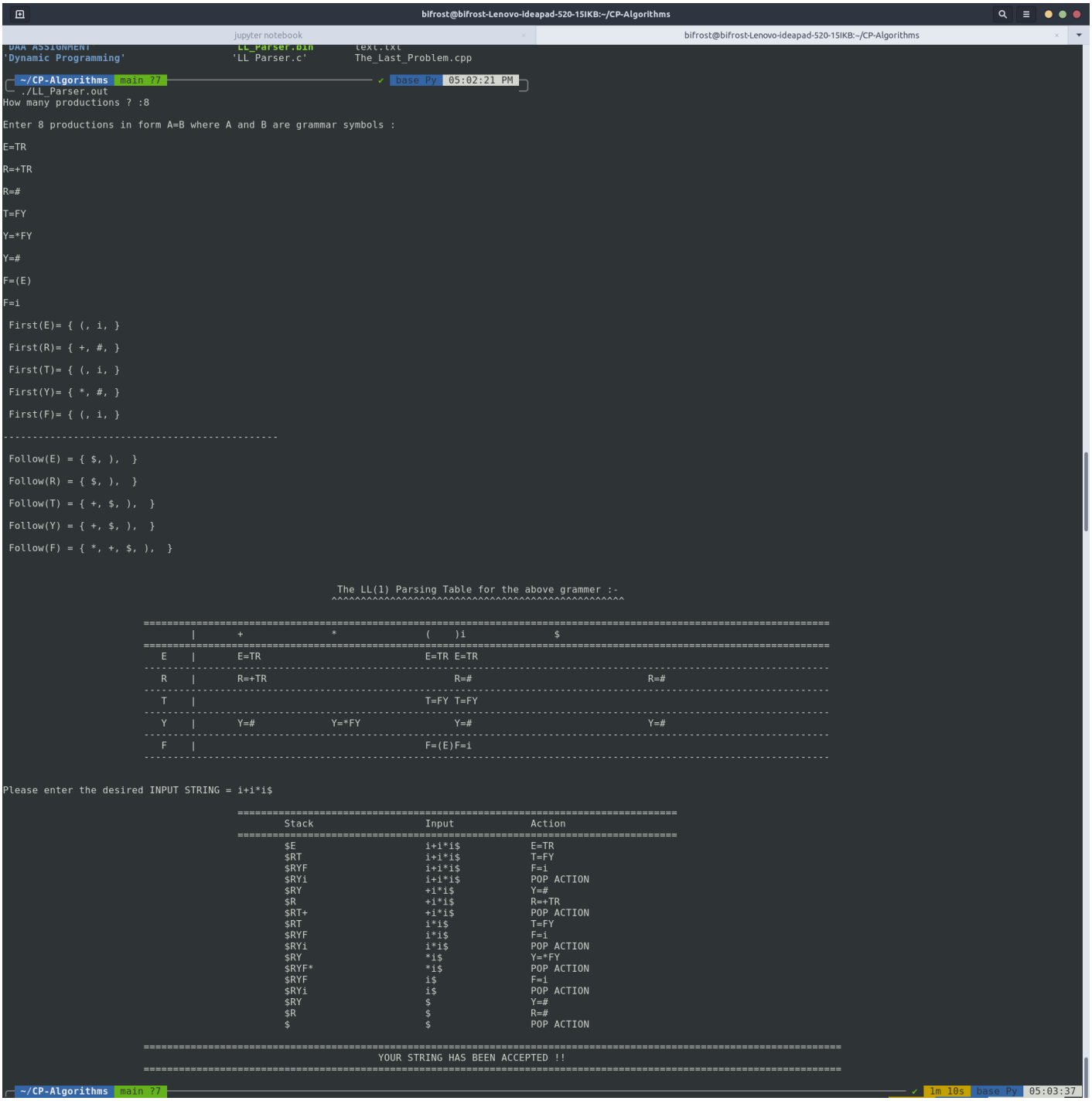
```
    ptr = -1;
    for(ap = 0; ap < land ; ap++){
        for(kay = 0; kay < (sid + 1) ; kay++){
            table[ap][kay] = '!';
        }
    }
    for(ap = 0; ap < count ; ap++){
        ck = production[ap][0];
        xxx = 0;
        for(kay = 0; kay <= ptr; kay++)
            if(ck == table[kay][0])
                xxx = 1;
        if (xxx == 1)
            continue;
        else{
            ptr = ptr + 1;
            table[ptr][0] = ck;
        }
    }
    for(ap = 0; ap < count ; ap++){
        int tuna = 0;
        while(first_prod[ap][tuna] != '\0'){
            int to,ni=0;
            for(to=0;to<sid;to++){
                if(first_prod[ap][tuna] == ter[to]){
                    ni = 1;
                }
            }
            if(ni == 1){
                char xz = production[ap][0];
                int cz=0;
                while(table[cz][0] != xz){
                    cz = cz + 1;
                }
                int vz=0;
                while(ter[vz] != first_prod[ap][tuna]){
                    vz = vz + 1;
                }
                table[cz][vz+1] = (char)(ap + 65);
            }
            tuna++;
        }
    }
    for(k=0;k<sid;k++){
        for(kay=0;kay<100;kay++){
            if(calc_first[k][kay] == '!'){
                break;
            }
            else if(calc_first[k][kay] == '#'){
                int fz = 1;
                while(calc_follow[k][fz] != '!'){
                    char xz = production[k][0];
                    int cz=0;
                    while(table[cz][0] != xz){
                        cz = cz + 1;
                    }
                    int vz=0;
                    while(ter[vz] != calc_follow[k][fz]){
                        vz = vz + 1;
                    }
                    table[k][vz+1] = '#';
                    fz++;
                }
                break;
            }
        }
    }
    for(ap = 0; ap < land ; ap++){
        printf("\t\t\t   %c\t|\t",table[ap][0]);
        for(kay = 1; kay < (sid + 1) ; kay++){
            if(table[ap][kay] == '!')
```

```
                printf("\t\t");
            else if(table[ap][kay] == '#')
                printf("%c=#\t\t",table[ap][0]);
            else{
                int mum = (int)(table[ap][kay]);
                mum -= 65;
                printf("%s\t\t",production[mum]);
            }
        }
        printf("\n");
        printf("\t\t\t----------------------------------------------------------------------");
        printf("-----------------------------------------------------");
        printf("\n");
    }
    int j;
    printf("\n\nPlease enter the desired INPUT STRING = ");
    char input[100];
    scanf("%s%c",input,&ch);
    printf("\n\t\t\t\t\t\t");
    printf("================================================================================\n");
    printf("\t\t\t\t\t\tStack\t\t\tInput\t\t\tAction");
    printf("\n\t\t\t\t\t\t");
    printf("================================================================================\n");
    int i_ptr = 0,s_ptr = 1;
    char stack[100];
    stack[0] = '$';
    stack[1] = table[0][0];
    while(s_ptr != -1){
        printf("\t\t\t\t\t\t");
        int vamp = 0;
        for(vamp=0;vamp<=s_ptr;vamp++){
            printf("%c",stack[vamp]);
        }
        printf("\t\t\t");
        vamp = i_ptr;
        while(input[vamp] != '\0'){
            printf("%c",input[vamp]);
            vamp++;
        }
        printf("\t\t\t");
        char her = input[i_ptr];
        char him = stack[s_ptr];
        s_ptr--;
        if(!isupper(him)){
            if(her == him){
                i_ptr++;
                printf("POP ACTION\n");
            }
            else{
                printf("\nString Not Accepted by LL(1) Parser !!\n");
                exit(0);
            }
        }
        else{
            for(i=0;i<sid;i++){
                if(ter[i] == her)
                    break;
            }
            char produ[100];
            for(j=0;j<land;j++){
                if(him == table[j][0]){
                    if (table[j][i+1] == '#'){
                        printf("%c=#\n",table[j][0]);
                        produ[0] = '#';
                        produ[1] = '\0';
                    }
                    else if(table[j][i+1] != '!'){
                        int mum = (int)(table[j][i+1]);
                            mum -= 65;
                            strcpy(produ,production[mum]);
                            printf("%s\n",produ);
```

```
                    }
                    else{
                        printf("\nString Not Accepted by LL(1) Parser !!\n");
                        exit(0);
                    }
                }
            }
            int le = strlen(produ);
            le = le - 1;
            if(le == 0)
                continue;
            for(j=le;j>=2;j--){
                s_ptr++;
                stack[s_ptr] = produ[j];
            }
        }
    }
    printf("\n\t\t\t==========================================================================");
    printf("================================================\n");
    if (input[i_ptr] == '\0')
        printf("\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN ACCEPTED !!\n");
    else
        printf("\n\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN REJECTED !!\n");
    printf("\t\t\t==========================================================================");
    printf("================================================\n");
}

void follow(char c)
{
    int i ,j;
    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'){
                    followfirst(production[i][j+1],i,(j+2));
                }
                if(production[i][j+1]=='\0'&&c!=production[i][0]){
                    follow(production[i][0]);
                }
            }
        }
    }
}

void findfirst(char c ,int q1 , int q2)
{
    int j;
    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))
                {
                    findfirst(production[q1][q2], q1, (q2+1));
                }
                else
```

```
                    first[n++]='#';
                }
                else if(!isupper(production[j][2])){
                    first[n++]=production[j][2];
                }
                else {
                    findfirst(production[j][2], j, 3);
                }
            }
        }
    }

void followfirst(char c, int c1 , int c2)
{
    int k;
    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;
        }
        while(calc_first[i][j] != '!')
        {
            if(calc_first[i][j] != '#'){
                f[m++] = calc_first[i][j];
            }
            else{
                if(production[c1][c2] == '\0'){
                    follow(production[c1][0]);
                }
                else{
                     followfirst(production[c1][c2],c1,c2+1);
                 }
             }
             j++;
         }
     }
}
```

Output:


bifrost@bifrost-Lenovo-ideapad-520-15IKB:~/CP-Algorithms
jupyter notebook
bifrost@bifrost-Lenovo-ideapad-520-15IKB:~/CP-Algorithms
'DAA ASSIGNMENT'
'Dynamic Programming'
LL_Parser.bin
'LL Parser.c'
text.txt
The_Last_Problem.cpp
~/CP-Algorithms main ?7
base Py 05:02:21 PM
./LL_Parser.out
How many productions ? :8
Enter 8 productions in form A=B where A and B are grammar symbols :
E=TR
R=+TR
R=#
T=FY
Y=*FY
Y=#
F=(E)
F=i
First(E)= { (, i, }
First(R)= { +, #, }
First(T)= { (, i, }
First(Y)= { *, #, }
First(F)= { (, i, }
Follow(E) = { $,), }
Follow(R) = { $,), }
Follow(T) = { +, $,), }
Follow(Y) = { +, $,), }
Follow(F) = { *, +, $,), }
The LL(1) Parsing Table for the above grammer :-
	+	*	( )i	$
E	E=TR		E=TR E=TR	
R	R=+TR		R=#	R=#
T			T=FY T=FY	
Y	Y=#	Y=*FY	Y=#	Y=#
F			F=(E)F=i	
Please enter the desired INPUT STRING = i+i*i$
Stack Input Action
$E i+i*i$ E=TR
$RT i+i*i$ T=FY
$RYF i+i*i$ F=i
$RYi i+i*i$ POP ACTION
$RY +i*i$ Y=#
$R +i*i$ R=+TR
$RT+ +i*i$ POP ACTION
$RT i*i$ T=FY
$RYF i*i$ F=i
$RYi i*i$ POP ACTION
$RY *i$ Y=*FY
$RYF* *i$ POP ACTION
$RYF i$ F=i
$RYi i$ POP ACTION
$RY $ Y=#
$R $ R=#
$ $ POP ACTION
YOUR STRING HAS BEEN ACCEPTED !!
~/CP-Algorithms main ?7
1m 10s base Py 05:03:37