SLR PARSING ALGORITHM

FINAL REVIEW REPORT

Submitted by

Arpit Khurana (15BCE0353)

Abhinav Singhania(15BCE0432)

Prepared For

THEORY OF COMPILATION AND COMPILER DESIGN

(CSE2002) – PROJECT COMPONENT

Submitted To

ANNAPURNA J.

School of Computer Science and Engineering

VIT®
UNIVERSITY
(Estd. u/s 3 of UGC Act 1956)
VELLORE ▪ CHENNAI
www.vit.ac.in

School of Computer Science and Engineering

CERTIFICATE

The project report entitled “CONSTRUCTION OF DFA AND SLR PARSING ALGORITHM” is prepared and submitted by **ARPIT KHURANA (15BCE0353)** AND **ABHINAV SINGHANIA (15BCE0432).** It has been found satisfactory in terms of scope, quality and presentation as partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science and Engineering** in VIT University, India.

Guide

(Name & Signature)

Internal Examiner **External Examiner (Name &**
Signature) **(Name & Signature)**

ACKNOWLEDGEMENT

It is our privilege to express our sincerest regards to our project coordinator, ANNAPURNA J. for their valuable inputs, able guidance, encouragement, whole-hearted cooperation and constructive criticism throughout the duration of our project.

I deeply express our sincere thanks to our Head of Department and Dean for encouraging and allowing us to present the project on the topic "CONSTRUCTION OF DFA AND SLR PARSING ALGORITHM" For THEORY AND COMPILATION and COMPILER DESIGN.

I take this opportunity to thank all our lecturers who have directly or indirectly helped our project. We pay our respects and love to our parents and all other family members and friends for their love and encouragement throughout our career. Last but not the least we express our thanks to our friends for their cooperation and support.

ABSTRACT

This project comprises of implementation of one of the most important concepts in compiler construction. In this, we have implemented the construction of DFA in bottom-up SLR parser algorithm and also the implementation of parsing algorithm in which we parse a string for a well-defined context free grammar and check if it belongs to that particular grammar or not, in C programming language. **SLR(1)** parsers is the part of the compiler in which it uses same LR(0) configuration sets and have the same table structure and parser operation. The difference comes in assigning table actions, where we are going to use one token of lookahead to help arbitrate among the conflicts.

In parsing algorithm, theoretically we use only one stack to check if the string given as an input whether it belongs to the given grammar or not. Initially it starts with zero and then we parse it character by character and at the end is a dollar sign.But to implement the same in c language, we have taken two stacks system as one stack contain only the character part and the other contain only the numeric part of the string. At last, we have taken a multilevel array to define a particular grammar. The stack always contains a set of symbols until the input has been matched and only the start state remains - therefore a right sentential derivation can always be found for the given string, if the table is correctly designed without any conflicts. Hence, SLR parsing is correct.

INTRODUCTION: -

LR(0) is the simplest technique in the LR family. Although that makes it the easiest to learn, these parsers are too weak to be of practical use for anything but a very limited set of grammars. The fundamental limitation of LR(0) is the zero, meaning no lookahead tokens are used. It is a stifling constraint to have to make decisions using only what has already been read, without even glancing at what comes next in the input. If we could peek at the next token and use that as part of the decision-making, we will find that it allows for a much larger class of grammars to be parsed.

SLR(1) parsers use the same LR(0) configurating sets and have the same table structure and parser operation. The difference comes in assigning table actions, where we are going to use one token of lookahead to help arbitrate among the conflicts. Kinds of conflicts encountered in LR(0) parsing was the reduce actions that cause the grief. A state in an LR(0) parser can have at most one reduce action and cannot have both shift and reduce instructions. Since a reduce is indicated for any completed item, this dictates that each completed item must be in a state by itself. The simple improvement that SLR(1) makes on the basic LR(0) parser is to reduce only if the next input token is a member of the follow set of the non-terminal being reduced. When filling in the table, we don't assume a reduce on all inputs as we did in LR(0), we selectively choose the reduction only when the next input symbols in a member of the follow set.

In the SLR(1) parser, it is allowable for there to be both shift and reduce items in the same state as well as multiple reduce items. The SLR(1) parser will be able to determine which action to take as long as the follow sets are disjoint.

SLR(1) Grammars :

A grammar is SLR(1) if the following two conditions hold for each configurating set: 1. For any item A –> u•xv in the set, with terminal x, there is no complete item B –> w• in that set with x in Follow(B). In the tables, this translates no shift-reduce conflict on any state. This means the successor function for x from that set either shifts to a new state or reduces, but not both. 2. For any two complete items A –> u• and B –> v• in the set, the follow sets must be disjoint, e.g. Follow(A) ∩ Follow(B) is empty. This translates to no reduce-reduce conflict on any state. If more than one non-terminal could be reduced from this set, it must be possible to uniquely determine which using only one token of lookahead.

SLR(1) Limitations :

The SLR(1) technique still leaves something to be desired, because we are not using all the information that we have at our disposal. When we have a completed configuration (i.e., dot at the end) such as X –> u•, we know that this corresponds to a situation in which we have u as a handle on top of the stack which we then can reduce, i.e., replacing u by X. We allow such a reduction whenever the next symbol is in Follow(X). However, it may be that we should not reduce for every symbol in Follow(X), because the symbols below u on the stack preclude u being a handle for reduction in this case. In other words, SLR(1) states only tell us about the sequence on top of the stack, not what is below it on the stack. We may need to divide an SLR(1) state into separate states to differentiate the possible means by which that sequence has appeared on the stack. By carrying more information in the state, it will allow us to rule out these invalid reductions.

METHODOLOGY:

Constructing the Parse Table

The actions are placed in the Goto table.

1. For each transition arc between Si and Sj, with label X:

a. Enter j in the cell (Si, X) (i.e. enter the end state in the row of the start state under the arc label) b. If the label is a terminal, precede with a 's'

2. For each state with double circle (a reduce state):

a. Find the completed rules in the state's closure (those with the dot after the last symbol)

b. For each of these, find the Follow set of the LHS symbol.

c. For each terminal in the Follow set of the LHS, place an entry 'r' followed by the rule number (e.g., r5 if it is rule 5). 3. In place of the action reduce for the topmost rule (here E' :- E $), put the action 'accept'.

Using the Parse Table for SLR(1)

Initially we have:

1. The input vector: a vector of the input tokens, terminated by token $

2. Next token pointer: pointer to the next input token to be processed, initially pointing at the first token.

3. The Stack: initially we place S0 on the stack.

Recursively:

1. **Apply action**: Apply the action given in the cell indexed by current state and next input token.
    - If shift, move the next token onto the top of the stack, and move the pointer to the next token.
    - If reduce, look up the rule given in the action, and remove n*2 items from the stack (where n is the number of symbols on the RHS of the rule). Then place the LHS of the production on the stack.
    - If accept, then finish parsing, we have a finished analysis.

2. **Determine the Goto State:** The top of the stack now contains a state and a symbol (terminal or nonterminal). In the parse table, look up the row for that state, and the column for the symbol.

- If there just a number there, take this as the new state number.

- If there is “s” followed by a number, take the number as the new state number.
- If there is “r” followed by a number, check your parse table, you made an error in drawing it.
- If the cell is empty, check your parse table, you made an error in drawing it

3. **Change the State:**

• put the new state number on the top of the stack.

This is the theoretical way of applying the SLR parser. All the parts of the algorithms will be same except that while implementing the SLR parser in C language we cannot declare an array which can take both value and so due to this we have taken a system of two stacks, one which contains all the characters while the other that takes all the numerical part.

}

CODE: -

Construction of dfa

```
#include<stdio.h>

#include<string.h>

#include<conio.h>

int i,j,k,m,n=0,o,p,ns=0,tn=0,rr=0,ch=0;

char
read[15][10],gl[15],gr[15][10],temp,templ[15],tempr[15][10],*ptr,temp2[5],dfa[15][15];

struct states

{

   char lhs[15],rhs[15][10];

   int n;

}I[15];

int compstruct(struct states s1,struct states s2)

{

   int t;

   if(s1.n!=s2.n)

      return 0;

   if( strcmp(s1.lhs,s2.lhs)!=0 )

      return 0;

   for(t=0;t<s1.n;t++)

      if( strcmp(s1.rhs[t],s2.rhs[t])!=0 )

         return 0;

   return 1;

}

void moreprod()

{
```

```
int r,s,t,l1=0,rr1=0;
char *ptr1,read1[15][10];

for(r=0;r<I[ns].n;r++)
{
    ptr1=strchr(I[ns].rhs[l1],'.');
    t=ptr1-I[ns].rhs[l1];
    if( t+1==strlen(I[ns].rhs[l1]) )
    {
        l1++;
        continue;
    }
    temp=I[ns].rhs[l1][t+1];
    l1++;
    for(s=0;s<rr1;s++)
        if( temp==read1[s][0] )
            break;
    if(s==rr1)
    {
        read1[rr1][0]=temp;
        rr1++;
    }
  else
        continue;

    for(s=0;s<n;s++)
    {
        if(gl[s]==temp)
```

```
            {
                I[ns].rhs[I[ns].n][0]='.';
                I[ns].rhs[I[ns].n][1]=NULL;
                strcat(I[ns].rhs[I[ns].n],gr[s]);
                I[ns].lhs[I[ns].n]=gl[s];
                I[ns].lhs[I[ns].n+1]=NULL;
                I[ns].n++;
            }
        }
    }
}

void canonical(int l)
{
    int t1;
    char read1[15][10],rr1=0,*ptr1;
    for(i=0;i<I[l].n;i++)
    {
        temp2[0]='.';
        ptr1=strchr(I[l].rhs[i],'.');
        t1=ptr1-I[l].rhs[i];
        if( t1+1==strlen(I[l].rhs[i]) )
            continue;

        temp2[1]=I[l].rhs[i][t1+1];
        temp2[2]=NULL;

        for(j=0;j<rr1;j++)
    if( strcmp(temp2,read1[j])==0 )
        break;
if(j==rr1)
{
    strcpy(read1[rr1],temp2);
    read1[rr1][2]=NULL;
    rr1++;
}
else
    continue;

for(j=0;j<I[0].n;j++)
{
    ptr=strstr(I[l].rhs[j],temp2);
    if( ptr )
    {
        templ[tn]=I[l].lhs[j];
        templ[tn+1]=NULL;
        strcpy(tempr[tn],I[l].rhs[j]);
        tn++;
    }
}

for(j=0;j<tn;j++)
{
    ptr=strchr(tempr[j],'.');
    p=ptr-tempr[j];
    tempr[j][p]=tempr[j][p+1];
    tempr[j][p+1]='.';
    I[ns].lhs[I[ns].n]=templ[j];
    I[ns].lhs[I[ns].n+1]=NULL;
    strcpy(I[ns].rhs[I[ns].n],tempr[j]);
    I[ns].n++;
}

moreprod();
for(j=0;j<ns;j++)
{
 //if ( memcmp(&I[ns],&I[j],sizeof(struct states))==1 )
    if( compstruct(I[ns],I[j])==1 )
    {
        I[ns].lhs[0]=NULL;
        for(k=0;k<I[ns].n;k++)
            I[ns].rhs[k][0]=NULL;
        I[ns].n=0;
        dfa[l][j]=temp2[1];
        break;
    }
}
if(j<ns)
{
    tn=0;
    for(j=0;j<15;j++)
    {
        templ[j]=NULL;
        tempr[j][0]=NULL;
```

```
            }
            continue;
        }

        dfa[l][j]=temp2[1];
        printf("\n\nI%d :",ns);
        for(j=0;j<I[ns].n;j++)
            printf("\n\t%c -> %s",I[ns].lhs[j],I[ns].rhs[j]);
        ns++;
        tn=0;
        for(j=0;j<15;j++)
        {
            templ[j]=NULL;
            tempr[j][0]=NULL;
        }
    }
}

int main()
{
    FILE *f;
    int l;

    for(i=0;i<15;i++)
    {
        I[i].n=0;
        I[i].lhs[0]=NULL;
        I[i].rhs[0][0]=NULL;
    dfa[i][0]=NULL;
}
f=fopen("tab6.txt","r");
while(!feof(f))
{
    fscanf(f,"%c",&gl[n]);
    fscanf(f,"%s\n",gr[n]);
    n++;}

printf("THE GRAMMAR IS AS FOLLOWS\n");
for(i=0;i<n;i++)
    printf("\t\t\t\t%c -> %s\n",gl[i],gr[i]);

I[0].lhs[0]='Z';
strcpy(I[0].rhs[0],".S");
I[0].n++;
l=0;
for(i=0;i<n;i++)
{
    temp=I[0].rhs[l][1];
    l++;
    for(j=0;j<rr;j++)
        if( temp==read[j][0] )
            break;
    if(j==rr)
    {
        read[rr][0]=temp;
        rr++;
        }
        else
            continue;
       for(j=0;j<n;j++)
  {
            if(gl[j]==temp)
            {
                I[0].rhs[I[0].n][0]='.';
                strcat(I[0].rhs[I[0].n],gr[j]);
                I[0].lhs[I[0].n]=gl[j];
                I[0].n++;
            }
        }
    }
    ns++;

    printf("\nI%d :\n",ns-1);
    for(i=0;i<I[0].n;i++)
        printf("\t%c -> %s\n",I[0].lhs[i],I[0].rhs[i]);

    for(l=0;l<ns;l++)
        canonical(l);

    printf("\t\t\tDFA TABLE IS AS FOLLOWS\n\n\n");
    for(i=0;i<ns;i++)
    {
        printf("I%d : ",i);
        for(j=0;j<ns;j++)
```

```
            if(dfa[i][j]!='\0')

                printf("'%c'->I%d | ",dfa[i][j],j);

        printf("\n\n\n");

    }


    printf("\n\n\n\t\tPRESS ANY KEY TO EXIT");

    getch();

}
```

PARSING OF STRING: -

```
#include<stdio.h>

#include<string.h>

int action[][6][2]={

        {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},

        {{-1,-1},{100,6},{-1,-1},{-1,-1},{-1,-1},{102,102}},

        {{-1,-1},{101,2},{100,7},{-1,-1},{101,2},{101,2}},

        {{-1,-1},{101,4},{101,4},{-1,-1},{101,4},{101,4}},

        {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},

        {{-1,-1},{101,6},{101,6},{-1,-1},{101,6},{101,6}},

        {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},

        {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},

        {{-1,-1},{100,6},{-1,-1},{-1,-1},{100,1},{-1,-1}},

        {{-1,-1},{101,1},{100,7},{-1,-1},{101,1},{101,1}},

        {{-1,-1},{101,3},{101,3},{-1,-1},{101,3},{101,3}},

        {{-1,-1},{101,5},{101,5},{-1,-1},{101,5},{101,5}}

        };//Axn Table

int gotto[12][3]={1,2,3,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,2,3,-1,-1,-1,

   -1,9,3,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};  //GoTo table

int state[10];

char string[10];

int statetop=-1,stringtop=-1,i;

void statepush(int k)

{

  if(statetop<9)

    state[++statetop]=k;

}
```

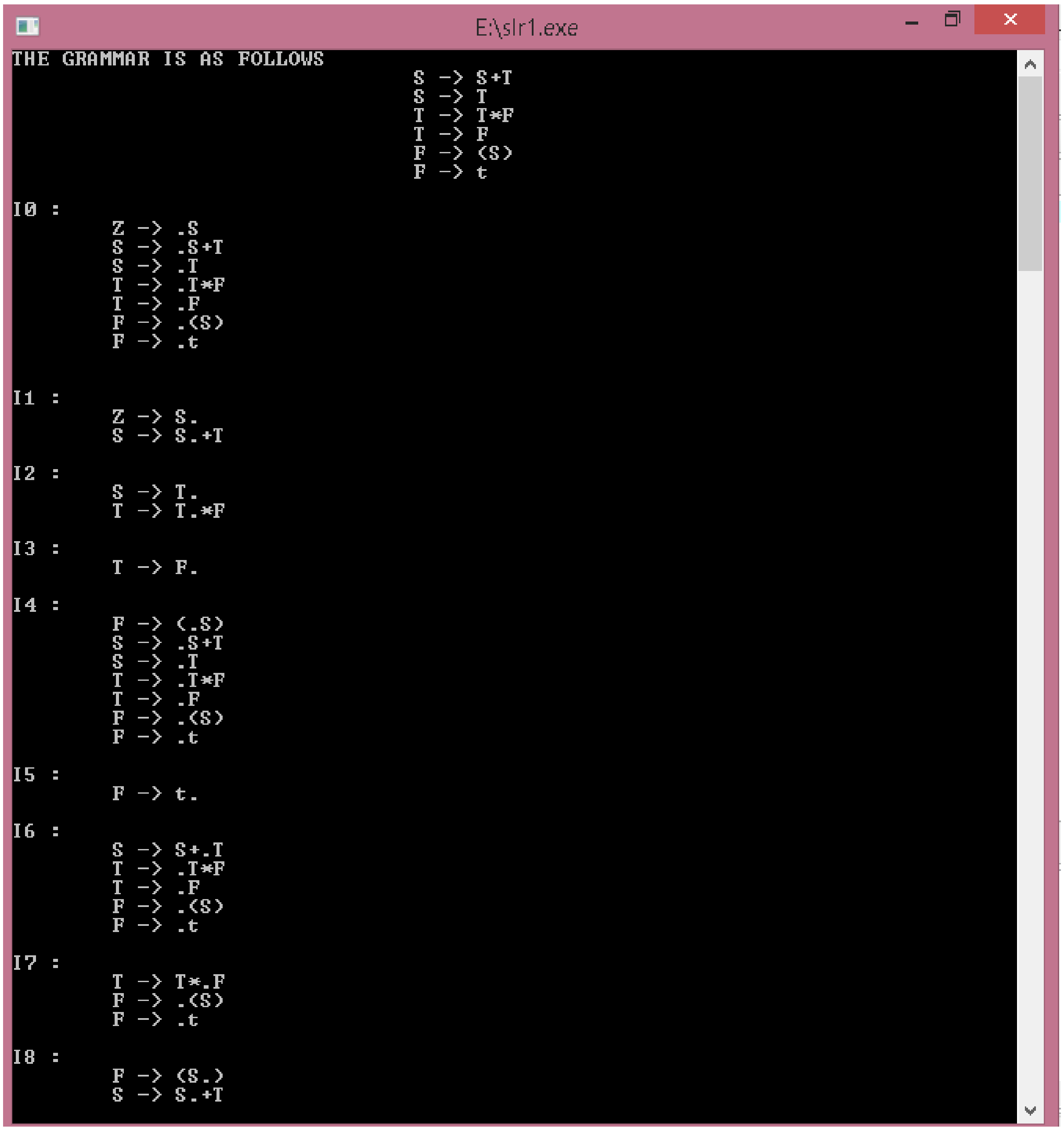
```
void stringpush(char k)
{
  if(stringtop<9)
    string[++stringtop]=k;
}
char topofstack()
{
   return state[statetop];
}
void statepop()
{
   if(statetop>=0)
     statetop--;
}
void stringpop()
{
   if(stringtop>=0)
     string[stringtop--]='\0';
}

void display()
{
  for(i=0;i<=statetop;i++)
    printf("%d%c",state[i],string[i]);
}
void displaycur(char p[],int m) //Displays The Present Input String
{
  int l;
  printf("\t\t");
  for(l=m;p[l]!='\0';l++)
    printf("%c",p[l]);
  printf("\n");
}
void error()
{
  printf("Syntax Error");
}
void reduce(int p)
{
   int len,k,ad;
   char source,*destination;
   switch(p)
   {

case 1:destination="E+T";
        source='E';
        break;
 case 2:destination="T";
        source='E';
        break;
 case 3:destination="T*F";
        source='T';
        break;
 case 4:destination="F";
        source='T';
        break;
 case 5:destination="(E)";
        source='F';
        break;
 case 6:destination="i";
        source='F';
        break;
 default:destination="\0";
  source='\0';
  break;
   }



for(k=0;k<strlen(destination);k++)
   {
     statepop();
     stringpop();
   }
   stringpush(source);
   switch(source)
   {
 case 'E':ad=0;
   break;
 case 'T':ad=1;
   break;
 case 'F':ad=2;
   break;
 default: ad=-1;
   break;
```

```
    }
  statepush(gotto[topofstack()][ad]);
}
int main()
{
   int j,st,ic;
   char ip[20]="\0",an;
  // clrscr();
   printf("Enter any String\n");
   scanf("%s",ip);
   statepush(0);
   display();
   printf("\t%s\n",ip);
   for(j=0;ip[j]!='\0';)
   {
 st=topofstack();
 an=ip[j];
 if(an>='a'&&an<='z') ic=0;
 else if(an=='+') ic=1;
 else if(an=='*') ic=2;
 else if(an=='(') ic=3;
 else if(an==')') ic=4;
 else if(an=='$') ic=5;
 else {
    error();
    break;

   if(action[st][ic][0]==100)
```

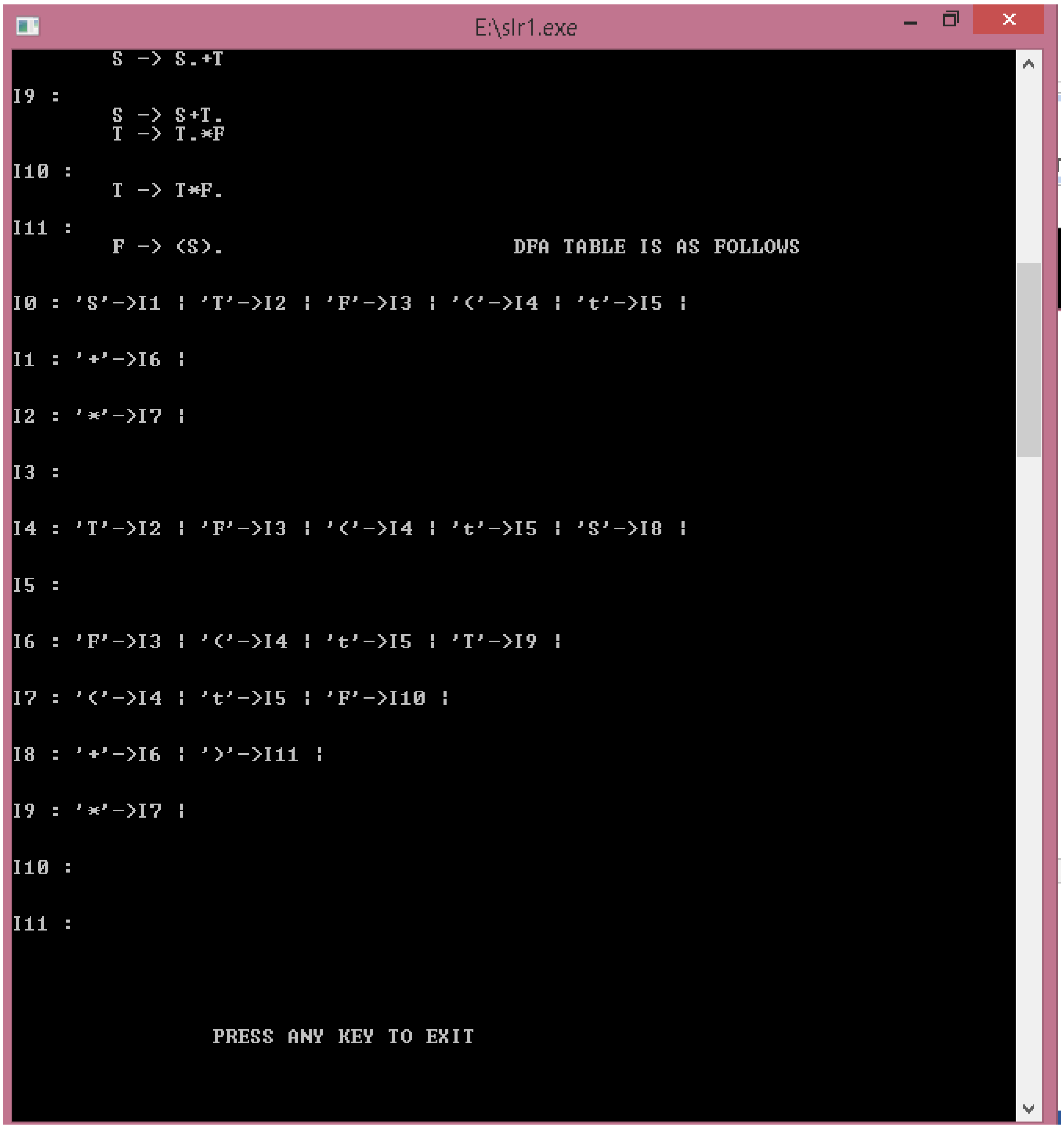
```
    {
      stringpush(an);
      statepush(action[st][ic][1]);
      display();
      j++;
      displaycur(ip,j);
    }
   if(action[st][ic][0]==101)
    {
      reduce(action[st][ic][1]);
      display();
      displaycur(ip,j);
    }
   if(action[st][ic][1]==102)
    {
   printf("Given String is accepted \n");
 //  getch();
      Break;
    }
 /*  else
   {
   printf("Given String is rejected \n");

      break;
    }*/
  }
return 0;
}
```

RESULTS: -

Construction of dfa

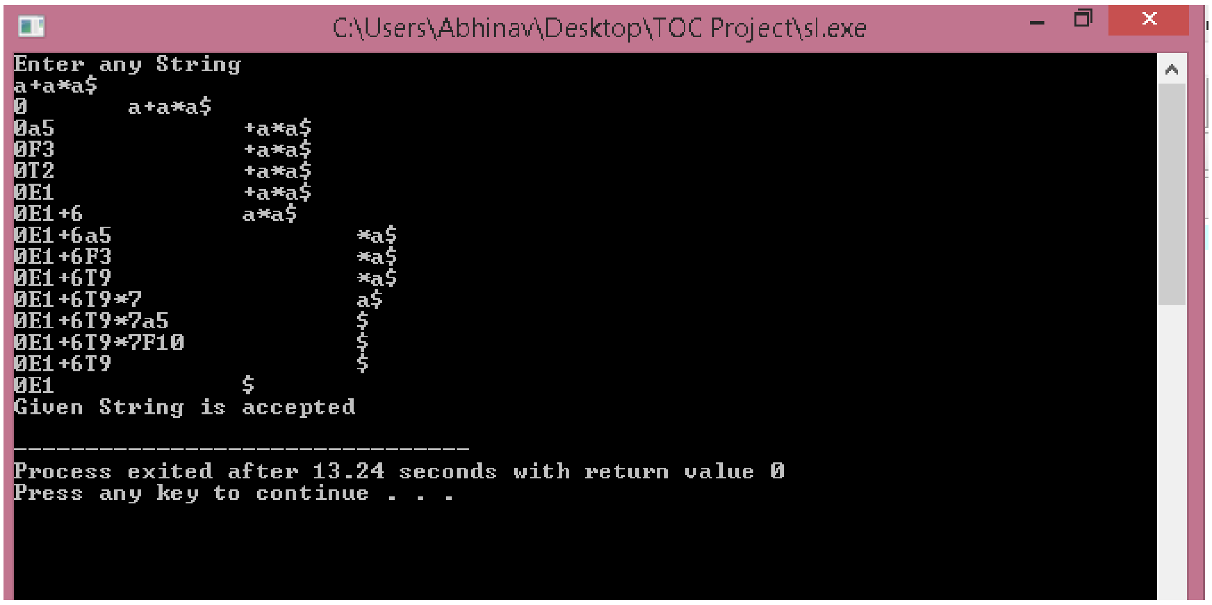


```
THE GRAMMAR IS AS FOLLOWS
                                S -> S+T
                                S -> T
                                T -> T*F
                                T -> F
                                F -> (S)
                                F -> t

I0 :
        Z -> .S
        S -> .S+T
        S -> .T
        T -> .T*F
        T -> .F
        F -> .(S)
        F -> .t


I1 :
        Z -> S.
        S -> S.+T

I2 :
        S -> T.
        T -> T.*F

I3 :
        T -> F.

I4 :
        F -> (.S)
        S -> .S+T
        S -> .T
        T -> .T*F
        T -> .F
        F -> .(S)
        F -> .t

I5 :
        F -> t.

I6 :
        S -> S+.T
        T -> .T*F
        T -> .F
        F -> .(S)
        F -> .t

I7 :
        T -> T*.F
        F -> .(S)
        F -> .t

I8 :
        F -> (S.)
        S -> S.+T
```



```
        S -> S.+T

I9 :
        S -> S+T.
        T -> T.*F

I10 :
        T -> T*F.

I11 :
        F -> (S).                          DFA TABLE IS AS FOLLOWS

I0 : 'S'->I1 ¦ 'T'->I2 ¦ 'F'->I3 ¦ '('->I4 ¦ 't'->I5 ¦

I1 : '+'->I6 ¦

I2 : '*'->I7 ¦

I3 :

I4 : 'T'->I2 ¦ 'F'->I3 ¦ '('->I4 ¦ 't'->I5 ¦ 'S'->I8 ¦

I5 :

I6 : 'F'->I3 ¦ '('->I4 ¦ 't'->I5 ¦ 'T'->I9 ¦

I7 : '('->I4 ¦ 't'->I5 ¦ 'F'->I10 ¦

I8 : '+'->I6 ¦ ')'->I11 ¦

I9 : '*'->I7 ¦

I10 :

I11 :



                PRESS ANY KEY TO EXIT
```

Parsing of string: -



```
Enter any String
a+a*a$
0       a+a*a$
0a5             +a*a$
0F3             +a*a$
0T2             +a*a$
0E1             +a*a$
0E1+6           a*a$
0E1+6a5                 *a$
0E1+6F3                 *a$
0E1+6T9                 *a$
0E1+6T9*7               a$
0E1+6T9*7a5             $
0E1+6T9*7F10            $
0E1+6T9                 $
0E1             $
Given String is accepted

--------------------------------
Process exited after 13.24 seconds with return value 0
Press any key to continue . . .
```

REFRENCES: -

[1] A.V. Aho, J. Hopcroft and J.D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974. 22

[2] Aho, A.V., Johnson, S.C. LR Parsing. ACM Computing Surveys, vol. 6, no. 2, pp. 99-124, 1974.

[3] Aho, A.V., Sethi, R., Ullman, J.D. Compilers: Principles, Techniques and Tools. AddisonWesley, Reading, MA (1986).

[4] Aho, A.V., Ullman, J.D. The Theory of Parsing, Translation and Compiling; vol. 1, Parsing. Prentice Hall, Englewood Clis, NJ (1972).

[5] Dencker, P., D • urre, K., Heuft, J. Optimization of Parser Tables for Portable Compilers. ACM Trans. on Prog. Lang. and Sys., 6,4, 546-572 (1984).