CONSTRUCTION OF LALR PARSING TABLE

A PROJECT REPORT

Submitted by

PARTH GOYAL – 16BCE0834

HARSH SINGHAL – 16BCE0855

POOJA GUPTA - 16BCE0863

EASHAN SINGH GILL - 16BCE0899

THEORY OF COMPUTATION AND COMPILER DESIGN - CSE2002

PROJECT SUPERVISOR

PROF. M. ANAND

SCHOOL OF COMPUTER SCIENCE ENGINEERING

NOVEMBER 2017

VIT UNIVERSITY VELLORE, 632014

CERTIFICATE

Certified that this project report "**CONSTRUCTION OF LALR PARSING TABLE**" is the bonafide work of "**PARTH GOYAL, HARSH SINGHAL, POOJA GUPTA, EASHAN SINGH GILL**" who carried out the project work under my supervision.

DATE **SIGNATURE**

Acknowledgement

We would like to thank our guide, Prof. M. Anand for the time, support and knowledge he has granted us. I would like to also thank the Head of the Department Prof. Senthilkumar R. and the Dean Dr. Arunkumar T. This project would not have been possible without their guidance. I am highly grateful to VIT University for providing a platform to achieve academic success. I would also like to express my gratitude to my loved ones who have supported me throughout the whole process.

Contents

Abstract

The project Construction of LALR Parsing Table deals with the C code for the construction of the predictive parsing table of a LALR parser. LALR parsers are basically CLR (Canonical LR) parser only but with reduces states. A LALR(1) parsing table is built from the configurating sets in the same way as canonical LR(1); the lookaheads determine where to place reduce actions. The reduction of states happen when there is no difference in 2 states except the look ahead, these 2 states can be combined. Our main objective is to write a C code for the same.

Introduction

A LALR(1) parsing table is built from the configurating sets in the same way as canonical LR(1); the lookaheads determine where to place reduce actions. In fact, if there are no mergable states in the configuring sets, the LALR(1) table will be identical to the corresponding LR(1) table and we gain nothing.

In the common case, however, there will be states that can be merged and the LALR table will have fewer rows than LR. The LR table for a typical programming language may have several thousand rows, which can be merged into just a few hundred for LALR. Due to merging, the LALR(1) table seems more similar to the SLR(1) and LR(0) tables, all three have the same number of states (rows), but the LALR may have fewer reduce actions—some reductions are not valid if we are more precise about the lookahead.

Thus, some conflicts are avoided because an action cell with conflicting actions in SLR(1) or LR(0) table may have a unique entry in an LALR(1) once some erroneous reduce actions have been eliminated.

Theory

SLR, LALR and LR parsers can all be implemented using exactly the same table-driven machinery.
Fundamentally, the parsing algorithm collects the next input token T, and consults the current state S (and associated lookahead, GOTO, and reduction tables) to decide what to do:
SHIFT: If the current table says to SHIFT on the token T, the pair (S,T) is pushed onto the parse stack, the state is changed according to what the GOTO table says for the current token (e.g, GOTO(T)), another input token T' is fetched, and the process repeats

REDUCE: Every state has 0, 1, or many possible reductions that might occur in the state. If the parser is LR or LALR, the token is checked against lookahead sets for all valid reductions for the state. If the token matches a lookahead set for a reduction for grammar rule G = R1 R2 .... Rn, a stack reduction and shift occurs: the semantic action for G is called, the stack is popped n (from Rn) times, the pair (S,G) is pushed onto the stack, the new state S' is set to GOTO(G), and the cycle repeats with the same token T. If the parser is an SLR parser, there is at most one reduction rule for the state and so the reduction action can be done blindly without searching to see which reduction applies. It is useful for an SLR parser to know if there is a reduction or not; this is easy to tell if each state explicitly records the number of reductions associated with it, and that count is needed for the L(AL)R versions in practice anyway.

ERROR: If neither SHIFT nor REDUCE is possible, a syntax error is declared.

Innovative Idea: If searched on the internet we get codes for checking if the given string is parsable or not given the parser table as the input. But in out project we are trying to make the parser table given the string as the input.

PROPOSED ARCHITECTURE:

Pseudocode

1. Construct C={I_0,I_1……..,I_n}, the collection of sets of LR(1) items for G'.
2. State I of the parser is constructed from I_i. The parsing actions for state I are determined as follows :

a) If [A → α. a β, b] is in I_i, and goto(Ii, a) = Ij, then set action[i,a] to "shift j." Here, a is required to be a terminal.

b) If [A → α., a] is in I_i, A ≠ S', then set action[i,a] to "reduce A → α."

c) If [S'→S.,$] is in I_i, then set action[i ,$] to "accept."

If a conflict results from above rules, the grammar is said not to be LR(1), and the algorithm is said to be fail.

3. The goto transition for state i are determined as follows: If goto(I_i , A)= I_j ,then goto[i,A]=j.
4. All entries not defined by rules (2) and (3) are made "error."
5. The initial state of the parser is the one constructed from the set containing item [S'→.S, $].

C-code

The code for checking if a string is parsable in LALR or not.

```
#include<omp.h>
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
void push(char *,int *,char);
char stacktop(char *);
void isproduct(char,char);
int isterminal(char);
int isnonterminal(char);
int isstate(char);
void error();
void isreduce(char,char);
char pop(char *,int *);
void printtable(char *,int *,char [],int);
void rep(char [],int);
struct action
{
   char row[6][5];
};
const struct action A[12]={
   {"sf","emp","emp","se","emp","emp"},
   {"emp","sg","emp","emp","emp","acc"},
   {"emp","rc","sh","emp","rc","rc"},
   {"emp","re","re","emp","re","re"},
   {"sf","emp","emp","se","emp","emp"},
   {"emp","rg","rg","emp","rg","rg"},
   {"sf","emp","emp","se","emp","emp"},
   {"sf","emp","emp","se","emp","emp"},
   {"emp","sg","emp","emp","sl","emp"},
   {"emp","rb","sh","emp","rb","rb"},
   {"emp","rb","rd","emp","rd","rd"},
   {"emp","rf","rf","emp","rf","rf"}
};
struct gotol
{
   char r[3][4];
};
const struct gotol G[12]={
   {"b","c","d"},
   {"emp","emp","emp"},
   {"emp","emp","emp"},
   {"emp","emp","emp"},
   {"i","c","d"},
   {"emp","emp","emp"},
   {"emp","j","d"},
   {"emp","emp","k"},
   {"emp","emp","emp"},
   {"emp","emp","emp"},
};
```

```
char terminal[6]={'i','+','*',')','(','$'};
char nonterminal[3]={'E','T','F'};
char states[12]={'a','b','c','d','e','f','g','h','m','j','k','l'};
char stack[100];
int top=-1;
char temp[10];
struct grammar
{
   char left;
   char right[5];
};
const struct grammar rl[6]={
   {'E',"e+T"},
   {'E',"T"},
   {'T',"T*F"},
   {'T',"F"},
   {'F',"(E)"},
   {'F',"i"},
};
void main()
{
   char inp[80],x,p,dl[80],y,bl='a';
   int i=0,j,k,l,n,m,c,len;
   printf(" Enter the input :");
   scanf("%s",inp);
   len=strlen(inp);
   inp[len]='$';
   inp[len+1]='\0';
   push(stack,&top,bl);
   printf("\n stack \t\t\t input");
   printtable(stack,&top,inp,i);
   do
   {
      x=inp[i];
      p=stacktop(stack);
      isproduct(x,p);
      if(strcmp(temp,"emp")==0)
      error();
      if(strcmp(temp,"acc")==0)
      break;
      else
      {
         if(temp[0]=='s')
         {
            push(stack,&top,inp[i]);
            push(stack,&top,temp[1]);
            i++;
         }
         else
         {
            if(temp[0]=='r')
            {
               j=isstate(temp[1]);
               strcpy(temp,rl[j-2].right);
```

```
                dl[0]=rl[j-2].left;
                dl[1]='\0';
                n=strlen(temp);
                for(k=0;k<2*n;k++)
                pop(stack,&top);
                for(m=0;dl[m]!='\0';m++)
                push(stack,&top,dl[m]);
                l=top;
                y=stack[l-1];
                isreduce(y,dl[0]);
                for(m=0;temp[m]!='\0';m++)
                push(stack,&top,temp[m]);
            }
        }
      }
      printtable(stack,&top,inp,i);
   }
   while(inp[i]!='\0');
   if(strcmp(temp,"acc")==0)
   printf(" \n accept the input ");
   else
   printf(" \n do not accept the input ");
   getch();
}
void push(char *s,int *stackpointer,char item)
{
   if(*stackpointer==100)
   printf(" stack is full ");
   else
   {
      *stackpointer=*stackpointer+1;
      s[*stackpointer]=item;
   }
}
char stacktop(char *s)
{
   char i;
   i=s[top];
   return i;
}
void isproduct(char x,char p)
{
   int k,l;
   k=isterminal(x);
   l=isstate(p);
   strcpy(temp,A[l-1].row[k-1]);
}
int isterminal(char x)
{
   int i;
   for(i=0;i<6;i++)
   if(x==terminal[i])
   return (i+1);
   return 0;
```

```
}
int isnonterminal(char x)
{
   int i;
   for(i=0;i<3;i++)
   if(x==nonterminal[i])
   return (i+1);
   return 0;
}
int isstate(char p)
{
   int i;
   for(i=0;i<12;i++)
   if(p==states[i])
   return (i+1);
   return 0;
}
void error()
{
   printf(" error in the input ");
   exit(0);
}
void isreduce(char x,char p)
{
   int k,l;
   k=isstate(x);
   l=isnonterminal(p);
   strcpy(temp,G[k-1].r[l-1]);
}
char pop(char *s,int *stackpointer)
{
   char item;
   if(*stackpointer==-1)
   printf(" stack is empty ");
   else
   {
      item=s[*stackpointer];
      *stackpointer=*stackpointer-1;
   }
   return item;
}
void printtable(char *t,int *p,char inp[],int i)
{
   int r;
   printf("\n");
   for(r=0;r<=*p;r++)
   rep(t,r);
   printf("\t\t\t");
   for(r=i;inp[r]!='\0';r++)
   printf("%c",inp[r]);
}
void rep(char t[],int r)
{
   char c;
```

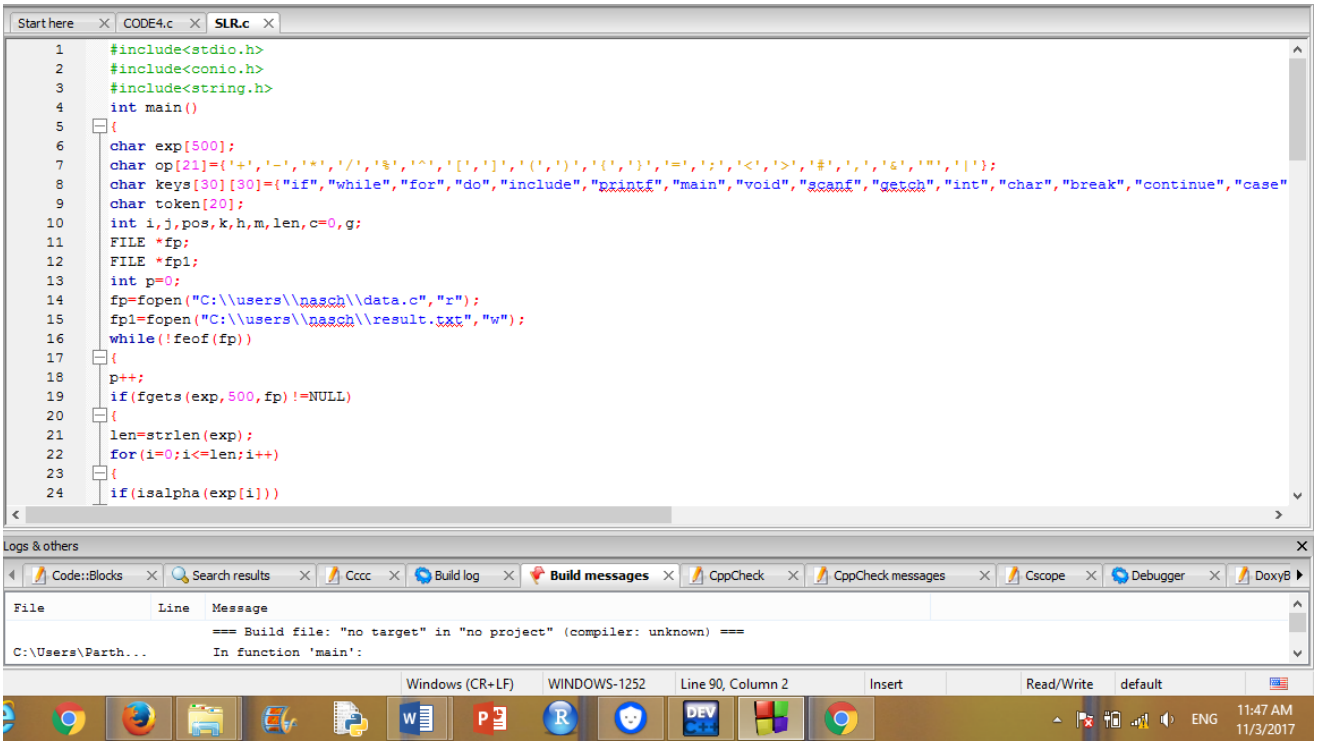
```
   c=t[r];
   switch(c)
   {
      case 'a': printf("0");
      break;
      case 'b': printf("1");
      break;
      case 'c': printf("2");
      break;
      case 'd': printf("3");
      break;
      case 'e': printf("4");
      break;
      case 'f': printf("5");
      break;
      case 'g': printf("6");
      break;
      case 'h': printf("7");
      break;
      case 'm': printf("8");
      break;
      case 'j': printf("9");
      break;
      case 'k': printf("10");
      break;
      case 'l': printf("11");
      break;
      default: printf("%c",t[r]);
      break;
   }
}
```

Code for construction of parsing table.

```
#include<omp.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
{
char exp[500];
char op[21]={'+','-','*','/','%','^','[',']','(',')','{','}','=',';','<','>','#',',','&','"','|'};
char
keys[30][30]={"if","while","for","do","include","printf","main","void","scanf","getch","int","char","brea
k","continue","case","const","default","else","double","float","void","isaplha","isdigit","strcmp"};
char token[20];
int i,j,pos,k,h,m,len,c=0,g;
FILE *fp;
FILE *fp1;
int p=0;
fp=fopen("C:\\users\\nasch\\Desktop\\data.c","r");
fp1=fopen("C:\\users\\nasch\\Desktop\\result.txt","w");
while(!feof(fp))
```

```
{
p++;
if(fgets(exp,500,fp)!=NULL)
{
len=strlen(exp);
for(i=0;i<=len;i++)
{
if(isalpha(exp[i]))
{
pos=i;
g=i;
while(isalpha(exp[g])||isdigit(exp[g]))
g++;
for(j=0;j<20;j++)
token[j]=NULL;
k=0;
for(j=pos;j<g;j++)
{
token[k]=exp[j];
k++;
}
c=0;
if((strcmp(token,"stdio")==0)||(strcmp(token,"conio")==0)||(strcmp(token,"string")==0))
{
fprintf(fp1,"%s.h is a header file\n",token);
g=g+2;
}
else
{
fprintf(fp1,"%s",token);
for(m=0;m<30;m++)
{
if(strcmp(token,keys[m])==0)
c++;
}
if(c==0)
fprintf(fp1," - It is a identifier\n");
else
fprintf(fp1," - It is a key word\n");
}
i=g-1;
}
else if(isdigit(exp[i]))
{
while(isdigit(exp[i+1]))
{
fprintf(fp1,"%c",exp[i]);
i++;
}
fprintf(fp1,"%c",exp[i]);
fprintf(fp1," - It is a digit\n");
}
else
{
```

```
if(exp[i]=='+'||exp[i]=='-'||exp[i]=='='||exp[i]=='|'||exp[i]=='<'||exp[i]=='>'||exp[i]=='!')
{
i++;
if(exp[i]=='+'||exp[i]=='-'||exp[i]=='='||exp[i]=='|')
fprintf(fp1,"%c%c - It is an operator\n",exp[i-1],exp[i]);
else
{
fprintf(fp1,"%c - It is an operator\n",exp[i-1]);
i--;
}
}
else
{
for(h=0;h<21;h++)
{
if(exp[i]==op[h])
{
fprintf(fp1,"%c - It is an operator\n",op[h]);
break;
}
}
}
}
}
}
}
getch();
return 0;
}
```



Conclusion

We made the code for checking if a string is LALR, which is telling us if the string belongs to the language or not correctly. We also wrote a code for generating the parser table when a parsable string is given as the input.

References

1. https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/140%20LALR%20Parsing.pdf
2. https://www.cs.clemson.edu/course/cpsc827/material/LRk/LR1.pdf