Exercise 5: Construct the LR(0) parsing table from the given gramamr rules.

Aim: To construct the LR(0) items and construct the **action** and **goto** table.

Theory:

LR parser is a bottom-up parser. L stands for left to right and R stands for the rightmost derivations in reverse. LR parser starts processing the given sentence of the language and keeps reducing the sentence till the start symbol is obtained. This parser identies the handle of the grammar in the sentence and replace it with the right hand side of the rule (non-terminal). LR parser consists of: (i) input buffer, (ii) stack, (iii) parsing table and (iv) parsing algorthm. The parsing table is constructed from the given grammar and is used for the recognition of the given sentence. The contents of the input buffering is the input sentence to be parsed. The contents of the stack is grammar symbol and the current state of the parse. The state on the top of the parser reflects the current status of the parser and appropriate actions such as shift, reduce, accept and error are taken

Steps:

1. Read the grammar rules (augmented grammar rules)
2. Compute First and Follow of the non-terminals
3. Create appropriate data structure for hodling the rules allocate space in the memory

   (we have used Boolean vector to store the presence of the rules in the set of items and position vector to store the information about the dot position in the rule. This representation simplifies the storage process)
4. Start with augmented start symbol and find the closure all the possible items (item refers to a grammar rule and set of items refer to the possible derived items from rule)

5. Construct the set of items by deriving from the initail set of items by making transition for each non-terminal and terminal.
6. Place the transitions from one set of items to another state for the non-terminal in the gotos table and terminal transition in the actions table (for shift action).
7. If dot is at the last position of the rule, it is the rule for reduce action. Place this reduce action in the actions table row indexed by the current state and coloumn indexed by the follow of the rule head.
8. All empty entries are error.
9. Place accept state when the dot appears at the end of the symbol

Modules:

Reading grammar rules and allocate space in the memory

Computation of First and Follow (as done in predictive parsing table)

Subset construction

Display modules

Other modules for checking duplicates state

Algorithm:

Refer section 4.5.3 for the algorithms:

```
ClosureLR0

GotoLR0

ConstructLR0ItemsSet

ConsturctSLRTable
```

Hints on the development:

The process involves the construction of the set of items. Hence, large numbers of rules are replicated with the dot positions at different locations. Hence in our program, we have represented the LR(0) item by two vectors (the dimesion of the vector is the number of rules in the given problem). The first vector used is a binary vector which tells wheher the rule is present in the given LR(0) items. Another vector shows the position of the dot in the rule. There can be case where one rule is associated with two positions in LR(0) item. This exercise illustrates such a case. Hence a structure called POSITION is ued to hold two dot position (first and second). We have used the word *items* to indicate all the LR(0) items. The word *item* is used to represent any one LR(0) item. The position represents the place at which dot is postioned to represent the state of parsing.

Data Structures and Prototype Declaration:

```
#include<stdio.h>
#include<string.h>
#include<malloc.h>
#include<stdlib.h>
#define MAXNAMESIZE 20
#define MAXSTACKSIZE 100
#define SENTENCESIZE 30
#define MAXRULES 100
#define MAXITEMS 100
#define MAXNTS 20
#define MAXTS 20
#define RH 1            //Rule Head
#define RB 2            //Rule Body
```

```
#define UNKNOWN -1
#define T 1
#define NT 2
#include "sstack.h"
#include "sstack.cc"
typedef struct LNode
{
      char name[MAXNAMESIZE];
      int tOrNt;   //terminal or Non-terminal
      int rhOrRb;  //rule head or rule body
//    int dot;    //for LR parser
        struct LNode *next;
} LNode;


typedef LNode *PtrToLNode ;
typedef PtrToLNode List;

class First
{
public:
      int ntIdx;
      int count;
      int maxTs;
      char **names;
      First(int);
      void initFirst();
};
```

```
typedef struct POSITIONS
{
      int first;
      int second;
}POSITIONS;

typedef struct POSITION
{
      int first;
      int second;
}POSITION;

typedef struct ACTION
{
      char action;
      int state;
}ACTION;

class LRParser
{

      int noRules;
      int noItems;
      int noNTs;
      int noTs;
      int **gotos;
      ACTION **actions;
      char **input;
      int itemsCount;
```

```
        char startSymbol[MAXNAMESIZE];
        int noInWords;
        int **parsingTable;
        char nts[MAXRULES][MAXNAMESIZE];
        char uniqueNTs[MAXNTS][MAXNAMESIZE];
        char uniqueTs[MAXTS][MAXNAMESIZE];
        int **ntRuleNumberMapping;
        int **items;
        int *item;
        POSITIONS **positions;
        POSITION *position;
        List *rules;
        char ***first;
        char ***follow;
        int *firstCount;
        int *followCount;
        int *rulesVisited;
        int *itemsVisited;
        char **leftHeadNames;
        char ***grammarSymbols;
        int **grammarSymbolsType;
        int *grammarSymbolCount;
    public:
        LRParser();
        void initItem();
        int validRule(int *);
        void initItemsProcessed();
        int isRulePresent(int,POSITION);
        int isItemPresent(int *itm, POSITION *posi);
```

```
void readRules(FILE *);

void readInput(FILE *);

void dispInput();

void constructTable();

void constructSetOfItems();

int isMemberInUniqueNTs(char *str);

int isMemberInUniqueTs(char *str);

int isPresent(int ntIdx,char *name, int count);

int isPresentFollow(int ntIdx,char *name, int count);

void addRuleIndexToNT(int ruleNo,char *ntName);

void dispNTRuleNumberMapping();

void dispActions();

void dispGotos();

int indexOfNT(char *str);

int indexOfT(char *str);

void computeAllFirst();

First computeFirst(char *);

void computeAllFollow();

void closure(int rno,int p);

void initRulesVisited();

int allItemsAreVisited();

void itemsProcessed();

void dispRules();

void dispNTs();

void dispTs();

void dispFirst();

void dispFollow();

void dispGrammarSymbols();

void dispParsingTable();
```

```
        void dispItems();
        void dispItem(int *);
        void dispPosition(POSITION *);
        int recognize();
};
```

The implementation of all the modules are as follows:

```
LRParser::LRParser()
{
        int i;
        rules = (List*)malloc(sizeof(List) * MAXRULES);
        if(rules == NULL) {printf("Memory allocation problem\n");exit(-1);}
        for(i=0;i<MAXRULES;i++)
        {
                rules[i]=(LNode*)malloc(sizeof(LNode));
                if(rules[i] == NULL) { printf("error in memory allocation for
                hash table\n");exit(-1);}
                rules[i]->next = NULL;
        }
}

void LRParser::addRuleIndexToNT(int ruleNo,char *ntName)
{
        int idx,j;
        idx=indexOfNT(ntName);
        if(idx != -1)
                for(j=0;j<noRules;j++)
                        if(ntRuleNumberMapping[idx][j]==-1)
```

```
                  {
                        ntRuleNumberMapping[idx][j]=ruleNo;
                        return;
                  }
}

void LRParser::dispNTRuleNumberMapping()
{
      int i,j;
      for(i=0;i<noNTs;i++)
      {
            printf("%s->",uniqueNTs[i]);
            for(j=0;j<noRules;j++)
                  if(ntRuleNumberMapping[i][j]==-1)
                  {
                        break;
                  }
                  else
                        printf("%d ",ntRuleNumberMapping[i][j]);
            printf("\n");
      }
}
void LRParser::initItem()
{
      int i;
      for(i=0;i<noRules;i++)
      {
            item[i] = 0;
            position[i].first=-1;
```

```
        position[i].second=-1;
    }
}

int LRParser::isRulePresent(int rno,POSITION pos)
{
    return (item[rno]==1 && (position[rno].first == pos.first &&
position[rno].second == pos.second));
}
int LRParser::isItemPresent(int *itm, POSITION *posi)
{
    int i,j;
    int found=-1;
    for(i=0;i<noItems;i++)
    {
        for(j=0;j<noRules;j++)
        {
        //  if(items[i][j] != itm[j]);
            if(items[i][j] != itm[j] || ( positions[i][j].first !=
        posi[j].first || positions[i][j].second !=
    posi[j].second))
            {
                found=-1;
                break;
            }
        }
        if( j==noRules)
            return i;
```

```
      }
      return found;
}

void  LRParser::closure(int rno,int pos)
{
      POSITION tp;
      int i,j,ntIdx,idx,rc=0;
      if(grammarSymbolsType[rno][pos] != NT)
      {
            return;
      }
      else
      {
            ntIdx = indexOfNT(grammarSymbols[rno][pos]);
            while((idx=ntRuleNumberMapping[ntIdx][rc]) != -1)
            {
                  rc++;
                  tp.first=0;
                  tp.second = -1;
                  if(isRulePresent(idx,tp))
                        continue;
                  else
                  {
                        item[idx] = 1;
                        if(position[idx].first == -1)
                        {
                              position[idx].first = 0;
                              position[idx].second = -1;
                        }
                        else
                              position[idx].second = 0;
                        closure(idx,0);
                  }
            }
      }
      return;
}

void LRParser::dispItem(int *itm)
{
      int j;
      for(j=0;j<noRules;j++)
            printf("%d ",itm[j]);
      printf("\n");
}

void LRParser::dispPosition(POSITION *posi)
{
      int j;
      printf("\n");
      for(j=0;j<noRules;j++)
            printf("%d %d\t",posi[j].first,posi[j].second);
      printf("\n");
}
void LRParser::dispItems()
{
      int i,j,k;
      printf("There are %d  set of Items are constructed\n",noItems);
      for(i=0;i<=noItems;i++)
      {
            printf("ITEM : I%d\n",i);
            for(j=0;j<noRules;j++)
            {
                  if(items[i][j])
                  {
                        printf("\t%s-->",leftHeadNames[j]);
                        for(k=0;k<grammarSymbolCount[j];k++)
                        {
                              if(k==positions[i][j].first ||
                        k==positions[i][j].second)
                                    printf(".%s ",grammarSymbols[j][k]);
                              else
                                    printf("%s ",grammarSymbols[j][k]);
                        }
                        if(k==positions[i][j].first ||
                        k==positions[i][j].second)
                              printf(".");
                        printf("\n");
                  }
            }
            printf("\n");
      }
}


int LRParser::allItemsAreVisited()
```

```
{
      int i;
      for(i=0;i<noItems;i++)
      {
            if(itemsVisited[i] == 0)
                  return 0;
      }
      return 1;
}

int LRParser::validRule(int *itm)
{
      int i,sum=0;
      for(i=0;i<noRules;i++)
            sum = sum + itm[i];
      return sum;
}

void LRParser::constructSetOfItems()
{
      int i,ii,j,rno=0,p=0,pos=0,existingItemIdx=-1;
      int itemIdx,tIdx,ntIdx,ni,ti,tIdxTemp,ntIdxTemp;
      char
ntName[MAXNAMESIZE],tName[MAXNAMESIZE],ntNameTemp[MAXNAMESIZE],tNameTemp[MAXNAMESIZE];
      printf("ConstructSetOfItems\n");
      computeAllFirst();
      computeAllFollow();
      initRulesVisited();
```

```
initItem();
position[rno].first=0;
position[rno].second=-1;
item[rno] = 1;
closure(rno,p);
if((existingItemIdx=isItemPresent(item,position)) ==-1)
{
      for(j=0;j<noRules;j++)
      {
            items[noItems][j] = item[j];
            positions[noItems][j].first = position[j].first;
            positions[noItems][j].second = position[j].second;
      }
}
do
{
      initItemsProcessed();
      itemIdx = 0;
      do
      {
            if(itemsVisited[itemIdx] == 1)
                  continue;
            for(ni = 1;ni < noNTs;ni++)
            {
                  strcpy(ntName,uniqueNTs[ni]);
                  ntIdx = indexOfNT(ntName);
                  initItem();
                  for(rno=0;rno<noRules;rno++)
                  {
                        if(items[itemIdx][rno] != 0)
                        {
                              pos = positions[itemIdx][rno].first;
                              if(pos != -1)
                              {
                                    if(pos ==
                                    grammarSymbolCount[rno])
                                    {

                                          continue;
                                    }

if(strcmp(grammarSymbols[rno][pos],ntName) != 0)
                                    {
                                          continue;
                                          //goto the next rule
                                    }
                                    pos = pos+1;
                                    if(position[rno].first == -1)
                                    {
                                          position[rno].first = pos;
                                    }
                                    else
                                          position[rno].second = pos;
                                    item[rno] = 1;
                              if(grammarSymbolsType[rno][pos] == NT)
                                    {
                                          closure(rno,pos);
                                    }
```

```
                    }
                    pos = positions[itemIdx][rno].second;
                    if(pos != -1)
                    {
                          if(pos ==
                    grammarSymbolCount[rno])
                          {

                                continue;
                          }

        if(strcmp(grammarSymbols[rno][pos],ntName) != 0)
                          {
                                continue;
                    }
                          pos = pos+1;
                          if(position[rno].first == -1)
                          {
                                position[rno].first = pos;
                          }
                          else
                                position[rno].second = pos;
                          item[rno] = 1;
                    if(grammarSymbolsType[rno][pos] == NT)
                          {
                                closure(rno,pos);
                          }
                    }
              }
        }
        //Non-Terminal Processing
        if(validRule(item) )
        {
              existingItemIdx =
              isItemPresent(item,position);
              if(existingItemIdx == -1)
              {
                    noItems++;
                    for(j=0;j<noRules;j++)
                    {
                          items[noItems][j] = item[j];
                          positions[noItems][j].first =
                          position[j].first;
                          positions[noItems][j].second =
                          position[j].second;
                    }
                    gotos[itemIdx][ntIdx] = noItems;
              }
              else
                    gotos[itemIdx][ntIdx] =
                    existingItemIdx;
        }

  }

  for(ti = 0;ti < noTs;ti++)
  {
        strcpy(tName,uniqueTs[ti]);
```

```
tIdx = indexOfT(tName);
initItem();
for(rno=0;rno<noRules;rno++)
{
      if(items[itemIdx][rno] != 0)
      {
            pos = positions[itemIdx][rno].first;
            if(pos != -1)
            {
                  if(pos ==
                  grammarSymbolCount[rno])
                  {


      strcpy(ntNameTemp,leftHeadNames[rno]);
                        ntIdxTemp  =
                  indexOfNT(ntNameTemp);
                        if(ntIdxTemp != 0)
                              for(ii =
            0;ii<followCount[ntIdxTemp];ii++)
                              {
                                    tIdxTemp =
            indexOfT(follow[ntIdxTemp][ii]);

actions[itemIdx][tIdxTemp].action = 'r';

      actions[itemIdx][tIdxTemp].state = rno;
                              }
                              continue;
                  }


if(strcmp(grammarSymbols[rno][pos],tName) != 0)
                  {
                        continue;
                  }
                  pos = pos+1;
                  item[rno] = 1;
                  if(position[rno].first == -1)
                  {
                        position[rno].first = pos;
                  }
                  else
                        position[rno].second = pos;
            if(grammarSymbolsType[rno][pos] == NT)
                  {
                        closure(rno,pos);
                  }
            }
            pos = positions[itemIdx][rno].second;
            if(pos != -1)
            {
                  if(pos ==
            grammarSymbolCount[rno])
                  {

            strcpy(ntNameTemp,leftHeadNames[rno]);
```

```
                        ntIdxTemp  =
                  indexOfNT(ntNameTemp);
                        if(ntIdxTemp != 0)
                              for(ii =
            0;ii<followCount[ntIdxTemp];ii++)
                              {
                                    tIdxTemp =
      indexOfT(follow[ntIdxTemp][ii]);

actions[itemIdx][tIdxTemp].action = 'r';

actions[itemIdx][tIdxTemp].state = rno;
                              }
                              continue;
                  }

if(strcmp(grammarSymbols[rno][pos],tName) != 0)
                  {
                        continue;
                  }
                  pos = pos+1;
                  item[rno] = 1;
                  if(position[rno].first == -1)
                  {
                        position[rno].first = pos;
                  }
                  else
                        position[rno].second = pos;
            if(grammarSymbolsType[rno][pos] == NT)
```

```
                        {
                              closure(rno,pos);
                        }
                  }
            }
      }
//Terminal Processing;
if(validRule(item) )
{
existingItemIdx= isItemPresent(item,position);
      if(existingItemIdx == -1)
      {
            noItems++;
            for(j=0;j<noRules;j++)
            {
                  items[noItems][j] = item[j];
positions[noItems][j].first = position[j].first;
positions[noItems][j].second = position[j].second;
            }
            actions[itemIdx][tIdx].state = noItems;
            actions[itemIdx][tIdx].action = 's';
      }
      else
      {
actions[itemIdx][tIdx].state = existingItemIdx;
            actions[itemIdx][tIdx].action = 's';


      }
}
```

```
                  }
                  itemsVisited[itemIdx] = 1;
                  itemIdx++;
            } while (itemIdx <= noItems);
      } while(!allItemsAreVisited());
}

void LRParser::constructTable()
{
      int i,j,ntIdx,tIdx;
      LNode *listPtr=NULL;
      computeAllFirst();
      computeAllFollow();

      for(i=0;i<noRules;i++)
      {
            if(rules[i]->next->next != NULL)
            {
                  ntIdx = indexOfNT(rules[i]->next->name);
                  listPtr = rules[i]->next->next;
                  if(listPtr->tOrNt == T)
                  {
                        tIdx = indexOfT(listPtr->name);
                        parsingTable[ntIdx][tIdx] = i;
                  }
                  else
                  {
                        for(j=0;j<firstCount[ntIdx];j++)
                        {
                              tIdx = indexOfT(first[ntIdx][j]);
                              parsingTable[ntIdx][tIdx] = i;
                        }
                  }
                  if(strcmp(rules[i]->next->next->name,"eps") == 0)
            {
                        for(j=0;j<followCount[ntIdx];j++)
                        {
                              tIdx = indexOfT(follow[ntIdx][j]);
                              parsingTable[ntIdx][tIdx] = i;
                        }
                  }
            }
      }
}

void LRParser::computeAllFirst()
{
      First fst(noTs);
      int i,j,ret,ntIdx;
      for(i=0;i<noNTs;i++)
      {
            fst.initFirst();
            initRulesVisited();
            fst=computeFirst(uniqueNTs[i]);
            //fst=computeFirst(listPtr->name);
            for(j=0;j<fst.count;j++)
                  if(!isPresent(i,fst.names[j],firstCount[i]))
                  {
                        strcpy(first[i][firstCount[i]],fst.names[j]);
                        firstCount[i] = firstCount[i]+1;
                  }
      }
}

void LRParser::initRulesVisited()
{
      int i;
      for(i=0;i<noRules;i++)
            rulesVisited[i]=0;
}

void LRParser::initItemsProcessed()
{
      int i;
      for(i=0;i<noItems;i++)
            itemsVisited[i]=0;
}

int LRParser::isPresent(int ntIdx,char *name, int count)
{
      int i, ret;
      for(i=0;i<count;i++)
      {
            ret = strcmp(first[ntIdx][i],name);
            if(ret == 0)
                  return 1;
      }
      return 0;
}

int LRParser::isPresentFollow(int ntIdx,char *name, int count)
{
      int i, ret;
      for(i=0;i<count;i++)
      {
            ret = strcmp(follow[ntIdx][i],name);
            if(ret == 0)
                  return 1;
      }
      return 0;
}

First LRParser::computeFirst(char nt[MAXNAMESIZE])
{
      int ntIdx,ntIdx1,i,rc=0;
      LNode *listPtr;
      First fst(noTs);
      ntIdx = indexOfNT(nt);
      fst.ntIdx = ntIdx;
      fst.initFirst();
      fst.ntIdx = ntIdx;
      while(ntRuleNumberMapping[ntIdx][rc] != -1)
      {

            if(rulesVisited[ntRuleNumberMapping[ntIdx][rc]]!=0)
            {
```

```
            rc++;

            continue;

      }

      rulesVisited[ntRuleNumberMapping[ntIdx][rc]]=1;

      listPtr = rules[ntRuleNumberMapping[ntIdx][rc]]->next->next;

      if(listPtr->tOrNt == T)

      {

            strcpy(fst.names[rc], listPtr->name);

      }

      else if(listPtr->tOrNt == NT)

      {

            ntIdx = indexOfNT(listPtr->name);

            fst=computeFirst(listPtr->name);

            for(i=0;i<fst.count;i++)

                  if(!isPresent(ntIdx,fst.names[i],firstCount[ntIdx]))

                  {


strcpy(first[ntIdx][firstCount[ntIdx]],fst.names[i]);

                        firstCount[ntIdx] = firstCount[ntIdx]+1;

                  }

            while(fst.count == 1 && (strcmp(fst.names[0],"eps")==0)

      && listPtr->next != NULL)

            {

                  listPtr=listPtr->next;

                  if(listPtr->tOrNt == T)


      if(!isPresent(ntIdx,fst.names[ntIdx],firstCount[ntIdx]))

                  {
                        strcpy(first[ntIdx][firstCount[ntIdx]],listPtr-
>name);
                              firstCount[ntIdx] = firstCount[ntIdx]+1;
                        }
                        fst=computeFirst(listPtr->name);
                  }
                  return fst;
            }
            rc++;
      }
      fst.count = rc;
      return fst;
 }

void LRParser::computeAllFollow()
{
      int i,j,ntIdx,followNTIdx,lastNTIdx,idx;
      int lastTOrNTType;
      LNode *listPtr=NULL, *lastNT=NULL,*prevPtr=NULL;
      char followNT[MAXNAMESIZE];
      char ruleHeadName[MAXNAMESIZE],lastNTName[MAXNAMESIZE];
      strcpy(startSymbol,rules[0]->next->name);
      ntIdx = indexOfNT(startSymbol);
      strcpy(follow[ntIdx][followCount[ntIdx]],"#");
      followCount[ntIdx] = followCount[ntIdx] + 1;
      for(i=0;i<noRules;i++)
      {
            listPtr = rules[i]->next->next;
            while(listPtr != NULL)
            {
                  if(listPtr->tOrNt == NT && listPtr->next != NULL &&
            listPtr->next->tOrNt == T)
                  {
                        ntIdx = indexOfNT(listPtr->name);
            if(!isPresent(ntIdx,listPtr->next->name,followCount[ntIdx]))
                        {

            strcpy(follow[ntIdx][followCount[ntIdx]],listPtr->next->name);
                        followCount[ntIdx] = followCount[ntIdx]+1;
                        }
                  }
                  else if(listPtr->tOrNt == NT && listPtr->next != NULL &&
listPtr->next->tOrNt == NT)
                  {
                        ntIdx = indexOfNT(listPtr->name);
                        strcpy(followNT,listPtr->next->name);
                        followNTIdx = indexOfNT(followNT);
                        for(j=0;j<firstCount[followNTIdx];j++)
                        {
                              if(  strcmp(first[followNTIdx][j],"eps") != 0)

if(!isPresentFollow(ntIdx,first[followNTIdx][j],followCount[ntIdx]))
                                    {

      strcpy(follow[ntIdx][followCount[ntIdx]],first[followNTIdx][j]);
                              followCount[ntIdx] = followCount[ntIdx]+1;
                                    }
                        }
                  }
                  listPtr = listPtr->next;
            }

      }
      for(i=0;i<noRules;i++)
      {
            listPtr = rules[i]->next;
            ntIdx = indexOfNT(listPtr->name);
            strcpy(ruleHeadName,listPtr->name);
            prevPtr = listPtr;
            lastNT=listPtr->next;
            lastTOrNTType = lastNT->tOrNt;
            strcpy(lastNTName,lastNT->name);
            while(listPtr->next != NULL)
            {
                  lastNT = listPtr->next;
                  lastTOrNTType = lastNT->tOrNt;
                  strcpy(lastNTName,lastNT->name);
                  prevPtr = listPtr;
                  listPtr = listPtr->next;
            }
            if(strcmp(ruleHeadName,lastNTName) != 0 && lastTOrNTType == NT)
            {
                  idx = indexOfNT(lastNT->name);
                  if(isPresent(idx,"eps",firstCount[idx]))
                  {
                        lastNTIdx = indexOfNT(prevPtr->name);
                        for(j=0;j<followCount[ntIdx];j++)
      if(!isPresentFollow(lastNTIdx,follow[ntIdx][j],followCount[lastNTIdx]))
                              {

      strcpy(follow[lastNTIdx][followCount[lastNTIdx]],follow[ntIdx][j]);
                                    followCount[lastNTIdx]                  =
followCount[lastNTIdx]+1;
                              }
                  }
            }
      }
      for(i=0;i<noRules;i++)
      {
            listPtr = rules[i]->next;
            ntIdx = indexOfNT(listPtr->name);
            strcpy(ruleHeadName,listPtr->name);
            lastNT=listPtr->next;
            lastTOrNTType = lastNT->tOrNt;
            strcpy(lastNTName,lastNT->name);
            while(listPtr->next != NULL)
            {
                  lastNT = listPtr->next;
                  lastTOrNTType = lastNT->tOrNt;
                  strcpy(lastNTName,lastNT->name);
                  listPtr = listPtr->next;
            }
            if(strcmp(ruleHeadName,lastNTName) != 0 && lastTOrNTType == NT)
            {
                  lastNTIdx = indexOfNT(lastNTName);
                  for(j=0;j<followCount[ntIdx];j++)

      if(!isPresentFollow(lastNTIdx,follow[ntIdx][j],followCount[lastNTIdx]))
                        {

      strcpy(follow[lastNTIdx][followCount[lastNTIdx]],follow[ntIdx][j]);
                        followCount[lastNTIdx] = followCount[lastNTIdx]+1;
                        }
            }
      }

}

int LRParser::isMemberInUniqueNTs(char *str)
{
      int i,ret;

      for(i=0;i<noNTs;i++)
      {
            ret=strcmp(str,uniqueNTs[i]);
            if(ret == 0)
                  return 1;
      }
      return 0;
}

int LRParser::indexOfNT(char *str)
{
      int i,ret=-1; // = noNTs;
```

```
        for(i=0;i<noNTs;i++)
        {
              ret=strcmp(str,uniqueNTs[i]);
              if(ret == 0)
                    return i;
        }
        return -1;
}

int LRParser::indexOfT(char *str)
{
        int i,ret=-1; // = noNTs;
        for(i=0;i<noTs;i++)
        {
              ret=strcmp(str,uniqueTs[i]);
              if(ret == 0)
                    return i;
        }
        return -1;
}

int LRParser::isMemberInUniqueTs(char *str)
{
        int i,ret;
        for(i=0;i<noTs;i++)
        {
              ret=strcmp(str,uniqueTs[i]);
              if(ret == 0)
                    return 1;
```

```
      }
      return 0;
}

void LRParser::dispFirst()
{
      int i,j;
      for(i=0;i<noNTs;i++)
      {
            printf("First(%s):->",uniqueNTs[i]);
            for(j=0;j<noTs+1;j++)
                  printf("%s ",first[i][j]);
            printf("\n");
      }
      printf("\n");
}

void LRParser::dispFollow()
{
      int i,j;
      for(i=0;i<noNTs;i++)
      {
            printf("Follow(%s):->",uniqueNTs[i]);
            for(j=0;j<noTs+1;j++)
                  printf("%s ",follow[i][j]);
            printf("\n");
      }
      printf("\n");
}
```

```
void LRParser::dispNTs()

{

      int i;

      printf("List of Non-terminals\n");

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

            printf("%s\t",uniqueNTs[i]);

      printf("\n");

}


void LRParser::dispTs()

{

      int i;

      printf("List of Terminals\n");

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

            printf("%s\t",uniqueTs[i]);

      printf("\n");

}


void LRParser::dispParsingTable()

{

      int i,j;


      printf("Parsing Table\n");

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

      printf("NT\t");

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

            printf("%s\t ", uniqueTs[j]);

      printf("\n----------------------------------------------------- \n");
```

```
    for(i=0;i<noNTs;i++)
    {
        printf("%s\t",uniqueNTs[i]);
        for(j=0;j<noTs;j++)
            if(parsingTable[i][j] == -1)
                printf("-\t");
            else
                printf("%d\t ",parsingTable[i][j]);
        printf("\n");
    }
    printf("\n---------------------------------------------------- \n");
}

void LRParser::readRules(FILE *fp)
{
    char str[20],prevstr[20];
    int soiIdx;
    List tp;
    LNode *listPtr;
    LNode *prevPtr;
    int i,j,ret;
    noRules = 0;
    noNTs = 0;
    noTs = 0;
    listPtr=prevPtr=NULL;
    tp = rules[noRules];
    strcpy(str," ");
    strcpy(prevstr," ");
    while(!feof(fp))
```

```
{

      fscanf(fp,"%s",str);
      if(strcmp(str,":") ==0)
      {
            prevPtr->next->tOrNt = NT;
            prevPtr->next->rhOrRb = RH;
            if(!isMemberInUniqueNTs(prevstr))
            {
                  strcpy(uniqueNTs[noNTs],prevstr);
                  noNTs++;
            }
      }
      else if(strcmp(str,";") == 0)
      {
            noRules++;
            tp = rules[noRules];
      }
      else
      {
            listPtr = (LNode*)malloc(sizeof(LNode));
            if(NULL == listPtr)
            {
                  printf("Error in memory allocations\n");
                  exit(-1);
            }
            listPtr->next = NULL;
            strcpy(listPtr->name, str);
            listPtr->tOrNt = UNKNOWN;
            listPtr->rhOrRb = RB;
            tp->next = listPtr;
            prevPtr = tp;
            tp = tp->next;
            strcpy(prevstr,str);
      }
}
noRules = noRules - 1;
for(i=0;i<noRules;i++)
{
      listPtr = rules[i]->next;
      while(listPtr != NULL)
      {
            ret=isMemberInUniqueNTs(listPtr->name);
            if(ret != 0)
            {
                  listPtr->tOrNt=NT;
            }
            else
            {
                  if(!isMemberInUniqueTs(listPtr->name) &&
            strcmp(listPtr->name,"eps") != 0)
                  {
                        strcpy(uniqueTs[noTs],listPtr->name);
                        noTs++;
                  }
                  listPtr->tOrNt=T;
            }
            listPtr = listPtr->next;
      }
}
strcpy(uniqueTs[noTs],"#");
noTs++;

first = (char ***) malloc(noNTs*sizeof(long int));
follow = (char ***) malloc(noNTs*sizeof(long int));
for (i=0;i<noNTs;i++)
{
      first[i] = (char**) malloc(noTs*sizeof(long int));
      follow[i] = (char**) malloc(noTs*sizeof(long int));
}
for(i=0;i<noNTs;i++)
      for(j=0;j<noTs+1;j++)
      {
            first[i][j] = (char*)malloc(MAXNAMESIZE*sizeof(char));
            follow[i][j] = (char*)malloc(MAXNAMESIZE*sizeof(char));
      }
for(i=0;i<noNTs;i++)
      for(j=0;j<noTs;j++)
      {
            strcpy(first[i][j],"");
            strcpy(follow[i][j],"");
      }
parsingTable = (int**) malloc(noNTs*sizeof(long int));
for(i=0;i<noNTs;i++)
      parsingTable[i] = (int *) malloc(noTs*sizeof(long int));
for(i=0;i<noNTs;i++)
      for(j=0;j<noTs;j++)
```

```
                parsingTable[i][j] = -1;
        ntRuleNumberMapping= (int**) malloc(noNTs*sizeof(long int));
        for(i=0;i<noNTs;i++)
            ntRuleNumberMapping[i] = (int *) malloc(noRules*sizeof(long int));
        for(i=0;i<noNTs;i++)
            for(j=0;j<noRules;j++)
                ntRuleNumberMapping[i][j]=-1;
        for(i=0;i<noRules;i++)
        {
            listPtr = rules[i]->next;
            addRuleIndexToNT(i,listPtr->name);
        }
        firstCount = (int*) malloc(sizeof(long int) * noNTs);
        for(i=0;i<noNTs;i++)
            firstCount[i] = 0;
        followCount = (int*) malloc(sizeof(long int) * noNTs);
        for(i=0;i<noNTs;i++)
            followCount[i] = 0;
        rulesVisited=(int*)malloc(sizeof(long int) * noRules);
        for(i=0;i<noRules;i++)
            rulesVisited[0];
        grammarSymbols = (char ***) malloc(sizeof(long int)*noRules);
        grammarSymbolsType = (int **) malloc(sizeof(long int)*noRules);
        leftHeadNames = (char **) malloc(sizeof(long int) * noRules);
        grammarSymbolCount = (int *) malloc(sizeof(long int)*noRules);
        for(i=0;i<noRules;i++)
        {
```

```
        grammarSymbols[i] = (char **) malloc(sizeof(long int) *
(noNTs+noTs) );
        grammarSymbolsType[i] = (int *) malloc(sizeof(long int) *
(noNTs+noTs) );
        leftHeadNames[i] = (char*) malloc(sizeof(char)*MAXNAMESIZE);
        strcpy(leftHeadNames[i], " ");
        grammarSymbolCount[i] = 0;
    }

    for(i=0;i<noRules;i++)
        for(j=0;j<(noNTs+noTs);j++)
        {
            grammarSymbols[i][j] = (char *)
malloc(sizeof(char)*(MAXNAMESIZE));
            strcpy(grammarSymbols[i][j], " ");
            grammarSymbolsType[i][j]=-1;
        }
    for(i=0;i<noRules;i++)
    {
        listPtr = rules[i]->next->next;
        j=0;
        strcpy(leftHeadNames[i],rules[i]->next->name);
        while(listPtr != NULL)
        {
            strcpy(grammarSymbols[i][j],listPtr->name);
            grammarSymbolsType[i][j] = listPtr->tOrNt;
            grammarSymbolCount[i]++;
            j++;
            listPtr=listPtr->next;
```

```
      }
}
items = (int**) malloc(sizeof(long int) * MAXITEMS);
positions = (POSITIONS**) malloc(sizeof(POSITIONS) * MAXITEMS);
for(i=0;i<MAXITEMS;i++)
{
      items[i] = (int*) malloc(sizeof(long int) * noRules);
      positions[i] = (POSITIONS*) malloc(sizeof(POSITIONS) * noRules);
}
item = (int*) malloc(sizeof(long int) * noRules);
position = (POSITION*) malloc(sizeof(POSITION) * noRules);
for(i=0;i<MAXITEMS;i++)
      for(j=0;j<noRules;j++)
      {
            items[i][j] = 0;
            positions[i][j].first = -1;
            positions[i][j].second = -1;
      }
for(j=0;j<noRules;j++)
{
      item[j] = 0;
      position[j].first = -1;
      position[j].second = -1;
}
gotos   = (int **) malloc(sizeof(long int) * MAXITEMS);
actions = (ACTION **) malloc(sizeof(ACTION) * MAXITEMS);
for(i = 0;i< MAXITEMS;i++)
{
      gotos[i] =   (int *) malloc(sizeof(long int) *noNTs);
```

```
            actions[i] = (ACTION *) malloc(sizeof(ACTION) * noTs);
      }
      itemsVisited = (int*) malloc(sizeof(long int) * MAXITEMS);

      for(i=0;i<MAXITEMS;i++)
      {
            itemsVisited[i] = 0;
            for(j=0;j<noNTs;j++)
                  gotos[i][j]=-1;
            for(j=0;j<noTs;j++)
            {
                  actions[i][j].action =' ';
                  actions[i][j].state = -1;
            }
      }
      input = (char **) malloc(sizeof(long int)* SENTENCESIZE);
      for(i=0;i<SENTENCESIZE;i++)
      {
            input[i] = (char*) malloc(sizeof(char)*MAXNAMESIZE);
            strcpy(input[i]," ");
      }
      return;
}

int LRParser::recognize()
{
      int i,ntIdx,tIdx,type,rno;
      char name[MAXNAMESIZE],word[MAXNAMESIZE];
      int ip=0;
      SStack s;
      s.push(startSymbol);
      do
      {
            strcpy(name,s.getTop());
            strcpy(word,input[ip]);
            if(isMemberInUniqueTs(name))
                  type = T;
            else if(isMemberInUniqueNTs(name))
                  type = NT;
            else {printf("Problem in deciding Terminal or Non-terminal.. Exit\n");exit(-1);}
            if(type == T || strcmp(name,"#") == 0)
            {
                  if(strcmp(name,word) == 0)
                  {
                        s.pop();
                        ip = ip+1;
                  }
                  else
                  {
                        printf("Error in parsing..\n");
                        exit(-1);
                  }
            }
            else
            {
                  ntIdx = indexOfNT(name);
                  tIdx  = indexOfT(word);
                  if(parsingTable[ntIdx][tIdx] != -1)
                  {
                        rno = parsingTable[ntIdx][tIdx];
                        if(strcmp(rules[rno]->next->next->name,"eps") == 0)
                        {
                              s.pop();
                        }
                        else
                        {
                              s.pop();
                              for(i=grammarSymbolCount[rno]-1;i>=0;i--)
                              {
                                    s.push(grammarSymbols[rno][i]);
                              }
                        }

                  }
                  else
                  {
                        printf("Error in parsing..\n");
                        exit(-1);
                  }
            }
      } while(strcmp(name,"#") != 0);
      return 1;
}

void LRParser::readInput(FILE *fp)
{
        char str[MAXNAMESIZE];
        noInWords = 0;
        while(!feof(fp))
        {
                fscanf(fp,"%s",str);
                strcpy(input[noInWords],str);
                noInWords++;
        }
        noInWords--;
}

void LRParser::dispInput()
{
        int i;
        for(i=0;i<noInWords;i++)
                printf("%s ",input[i]);
        printf("\n");
}

void LRParser::dispRules()
{
        LNode *listPtr;
        //LNode *prevPtr;
        int i;
        printf("\n");
        for(i=0;i<noRules;i++)
        {
                listPtr = rules[i]->next;
                printf("%d: %s--> ",i,listPtr->name);
```

```
            listPtr=listPtr->next;
            while(listPtr != NULL)
            {
                  printf("%s ",listPtr->name);
                  listPtr = listPtr->next;
            }
            printf("\n");
      }
}

void LRParser::dispGrammarSymbols()
{
      int i,j;
      printf("\n");
      for(i=0;i<noRules;i++)
      {
            printf("%s: ",leftHeadNames[i]);
            for(j=0;j<grammarSymbolCount[i];j++)
                  printf("%s ",grammarSymbols[i][j]);
            printf("\n");
      }
      printf("\n");
      for(i=0;i<noRules;i++)
      {
            for(j=0;j<grammarSymbolCount[i];j++)
                  printf("%d ",grammarSymbolsType[i][j]);
            printf("\n");
      }
}
```

```
void LRParser::dispActions()
{
      int i,j;
      printf("\n------------------------------------------------------\n");
      printf("\t");
      for(j=0;j<noTs;j++)
            printf("%s\t",uniqueTs[j]);
      printf("\n------------------------------------------------------\n");
      printf("noOfItems=%d\n",noItems);

      for(i=0;i<=noItems;i++)
      {
            printf("%d:\t",i);
            for(j=0;j<noTs;j++)
            {
                  if(actions[i][j].action != ' ')
                        printf("%c%d
                  \t",actions[i][j].action,actions[i][j].state);
                  else
                        printf("\t");
            }
            printf("\n");
      }
      printf("\n------------------------------------------------------\n");
}

void LRParser::dispGotos()
{
        int i,j;

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

        printf("\t");

        for(j=1;j<noNTs;j++)

                printf("%s\t",uniqueNTs[j]);

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

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

        {

                printf("%d:\t",i);

                for(j=1;j<noNTs;j++)

                {

                        if(gotos[i][j] != -1)

                                printf("%d\t",gotos[i][j]);

                        else

                                printf("\t");

                }

                printf("\n");

        }

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

}

int main()

{

        int parsingStatus;

        LRParser lrParser;

        FILE *fp,*fp1;

        char fname[30];

        fp=fopen("rules1.txt","r");

        if(NULL == fp)

        {
```

```
        printf("Error in file open.. Could not read rules from file\n");
        exit(-1);
}
fp1=fopen("input.txt","r");
if(NULL == fp1)
{
        printf("Error in file open.. Could not read sentence from file\n");
        exit(-1);
}

lrParser.readRules(fp);
lrParser.readInput(fp1);
lrParser.dispInput();
lrParser.dispNTs();
lrParser.dispTs();

lrParser.dispRules();
lrParser.dispParsingTable();
lrParser.dispNTRuleNumberMapping();
lrParser.dispGrammarSymbols();
lrParser.constructSetOfItems();
lrParser.dispFirst();
lrParser.dispFollow();
lrParser.constructTable();
lrParser.dispItems();
lrParser.dispActions();
lrParser.dispGotos();
```

```
      return 1;
}
```

Output

Grammar Specification:

E1 : E ;

E : E + T ;

E : T ;

T : T * F ;

T : F ;

F : (E) ;

F : id ;

The program generates the following results with the intermediate code as follows:

List of Non-terminals

E1 E T F

List of Terminals

+ * ( ) id #

Rules:

```
0: E1--> E

1: E--> E + T

2: E--> T

3: T--> T * F
```

```
4: T--> F

5: F--> ( E )

6: F--> id
```

Mapping the non-terminal and the rules

```
E1->0

E->1 2

T->3 4

F->5 6
```

First

```
First(E1):->( id

First(E):->( id

First(T):->( id

First(F):->( id
```

Follow

```
Follow(E1):->#

Follow(E):->+ ) #

Follow(T):->* + ) #

Follow(F):->* + ) #
```

The set of Items:

```
ITEM : I0

      E1-->.E

      E-->.E + T

      E-->.T

      T-->.T * F

      T-->.F
      F-->.( E )
      F-->.id

ITEM : I1
      E1-->E .
      E-->E .+ T

ITEM : I2
      E-->T .
      T-->T .* F

ITEM : I3
      T-->F .

ITEM : I4
      E-->.E + T
      E-->.T
      T-->.T * F
      T-->.F
      F-->.( .E )
      F-->.id

ITEM : I5
      F-->id .

ITEM : I6
      E-->E + .T
      T-->.T * F
      T-->.F
      F-->.( E )

      F-->.id


ITEM : I7

      T-->T * .F

      F-->.( E )

      F-->.id


ITEM : I8

      E-->E .+ T

      F-->( E .)


ITEM : I9

      E-->E + T .

      T-->T .* F


ITEM : I10

      T-->T * F .


ITEM : I11

      F-->( E ) .
```

Parsing Table - Action

```
------------------------------------------------------

      +     *     (     )     id    #

------------------------------------------------------

noOfItems=11

0:                s4          s5
```

```
1:    s6
2:    r2    s7          r2          r2
3:    r4    r4          r4          r4
4:                            s5
5:    r6    r6          r6          r6
6:                s4          s5
7:                s4          s5
8:    s6                s11
9:    r1    s7          r1          r1
10:   r3    r3          r3          r3
11:   r5    r5          r5          r5

------------------------------------------------------
```

Parsing Table - Goto

```
-----------------------------------
      E     T     F
-----------------------------------
0:    1     2     3
1:
2:
3:
4:    8     2     3
5:
6:          9     3
7:                10
8:
9:
```

```
10:
11:
----------------------------------
```

Exercises for the students

- Verify the construction of the set of items and the LR(0) parsing table.
- In this exercise the parsing table is constructed. Use the parsing table to parse an arithemetic expression as done for the predictive parser. Refer to the algorithm *LR_Parsing* in section 4.5.3.