Exercise – 4: Construction of Predictive Parsing Table and Parsing the sentence.

Aim: To constuct the predicitve parser and parse the given sentence using top-down parsing.

Theory:

Predictive parser is a recursive descent parser without backtracking that follows top-down parsing approach. Non-recursive grammar is given as an input to the predictive parser. The main components of the predictive parser are: stack, input buffer, parsing table and parsing algorithm. In this exercise, the parsing table is constructed from the given grammar rules and is used to parse the sentence for the syntactic checking of the source code. If the parsing is successful, the intermediate code is generated; otherwise an error message is displayed and the compiling process is terminated. The stack has the non-terminal that is expanded to match its contents with the input string. The parsing table is row-wise indexed by the non-terminal and column-wise indexed by the terminal. The cells in the parsing table is a rule by which the expansion of the rule occurs. If tmultiple rules are present in the cell, the grammar is ambiguous and has to be reworked.

Steps:

1. First read the grammar and identify terminal and nonterminal along with rule head.
2. Identifiy the First and Follow.
3. For each rule place the rule in the predictive parsing table row indexed by non-terminal associated with the rule head and column indexed by the first of this non-terminal.
4. If only epsilon is there in the first of the non-terminal place the rule in the predictive parsing table row indexed by non-terminal associated with the rule head and column indexed by the follows of this non-terminal.

5. Read Sentence and separate them into tokens.
6. Start with the start symbol and parse the sentence by repeateadly looking at the predictive parsing and replacing the non-terminal in the left hand side of the rule by its righthandside and matching with the tokens in the senetence.
7. Report any error if empty entries are encountered in the parsing table during the parsing process.

Modules:

- Reading the grammar rules
    - `readRules`
- Find the first and follow of the non-terminals
    - `computeAllFirst`
    - `computeFirst`
-
    - `computeAllFollow`
- Construct predictive parsing table
    - `constructTable`
- Reading the sentence to be parsed
- `readInput`
- Recognize the sentence
    - `recognize`
- Other modules for displaying the table and intermediate results, checking the duplicates, mapping of the non-terminal and rule number etc

Algorithm:

- Refer to section 4.4.3 and look at the algorithm *constructPredictiveTable, first, follow* for the construction of the predictive parsing table.
- Refer to the algorithm *Predictive_parser* in section 4.4.3 for the recognition of the sentence using predictive parser.

Data Structure and Prototype Declarations:

```
#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 MAXNTS 20
#define MAXTS 20
#define RH 1            //Rule Head
#define RB 2            //Rule Body
#define UNKNOWN -1
#define T 1
#define NT 2
#define testc {printf("\nTest and continue\n");}
#define teste {printf("\nTest and exit..\n");exit(-1);}
#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;    //maximum number of terminals
	char **names;
	First(int);
	void initFirst();
};

class PredParser
{

	int noRules;
	int noNTs;        //Number of non-terminals
	int noTs;         //Number of terminals
	char **input;
	char startSymbol[MAXNAMESIZE];
```

```
        int noInWords;
        int **parsingTable;
        char nts[MAXRULES][MAXNAMESIZE];
        char uniqueNTs[MAXNTS][MAXNAMESIZE]; //List of non-terminals
        char uniqueTs[MAXTS][MAXNAMESIZE];   //List of terminals
        int **ntRuleNumberMapping;
        List *rules;
        char ***first;
        char ***follow;
        int *firstCount;
        int *followCount;
        int *rulesVisited;
        char **leftHeadNames;
        char ***grammarSymbols;  //Grammar symbols in the right hand of rule
        int **grammarSymbolsType; //Grammar symbols type
        int *grammarSymbolCount;
public:
        PredParser();
        void readRules(FILE *);
        void readInput(FILE *);
        void dispInput();
        void constructTable();
        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();
        int indexOfNT(char *str);
```

```
	int indexOfT(char *str);
	void computeAllFirst();
	First computeFirst(char *);
	void computeAllFollow();
	void initRulesVisited();
	void dispRules();
	void dispNTs();
	void dispTs();
	void dispFirst();
	void dispFollow();
	void dispParsingTable();
	int recognize();
};
```

Source Code listing

The implementtion of the predictive parser is as follows:

```
PredParser::PredParser()
{
	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 PredParser::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 PredParser::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 PredParser::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 PredParser::computeAllFirst()
{
    First fst(noTs);
    int i,j;
    printf("%d is the nunber of Non-Terminals\n",noNTs);
    //for(i=0;i<noNTs;i++)
    for(i=0;i<noNTs;i++)
    {
        fst.initFirst();
        initRulesVisited();
        fst=computeFirst(uniqueNTs[i]);
        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 PredParser::initRulesVisited()
{
	int i;
	for(i=0;i<noRules;i++)
		rulesVisited[i]=0;
}

int PredParser::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 PredParser::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 PredParser::computeFirst(char nt[MAXNAMESIZE])
{
        int ntIdx,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)
      {
            //printf("%s...\n",uniqueNTs[ntIdx]);
            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 PredParser::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;
      //printf("%s is the start symbol\n",startSymbol);
      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 PredParser::isMemberInUniqueNTs(char *str)
{
      int i,ret; // = noNTs;

      for(i=0;i<noNTs;i++)
      {
            //printf("str = %s\n",str);
            ret=strcmp(str,uniqueNTs[i]);
            if(ret == 0)
                  return 1;
      }
      return 0;
}

int PredParser::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 PredParser::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 PredParser::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 PredParser::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 PredParser::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 PredParser::dispNTs()
{
```

```
	int i;
	printf("List of Non-terminals\n");
	for(i=0;i<noNTs;i++)
		printf("%s\t",uniqueNTs[i]);
	printf("\n");
}

void PredParser::dispTs()
{
	int i;
	printf("List of Terminals\n");
	for(i=0;i<noTs;i++)
		printf("%s\t",uniqueTs[i]);
	printf("\n");
}

void PredParser::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 PredParser::readRules(FILE *fp)
{
	char str[20],prevstr[20];
	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);
		//strcpy(prevstr,str);
```

```
if(strcmp(str,":") ==0)
{
	prevPtr->next->tOrNt = NT;
	prevPtr->next->rhOrRb = RH;
	//strcpy(nts[noRules],prevstr);
	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);
            //printf("ret=%d\n",ret);
            if(ret != 0)
            {
                  listPtr->tOrNt=NT;
                  //printf("Non Terminal %s..\n",listPtr->name);
            }
            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;
	}
}
```

```
	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 PredParser::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]);
		printf("Outside: Terminal = %s word = %s\n",name,word);
		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)
		{
```

```
printf("Terminal = %s word = %s\n",name,word);
      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)
            {
                  printf("EPS..Rule No. = %d Non-Terminal = %s
            word = %s\n",rno,name,word);
                  s.pop();
            }
            else
            {
                  s.pop();
```

```
                        printf("Inside else...Rule No. = %d Non-
                  Terminal = %s word = %s\n",rno,name,word);
                        printf("grammarSymbolCount =
                        %d\n",grammarSymbolCount[rno]);
                        for(i=grammarSymbolCount[rno]-1;i>=0;i--)
                        {
                              printf("....%s
                              ",grammarSymbols[rno][i]);
                              s.push(grammarSymbols[rno][i]);
                        }
                  }

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

void PredParser::readInput(FILE *fp)
{
      char str[MAXNAMESIZE];
      noInWords = 0;
      while(!feof(fp))
      {
```

```
            fscanf(fp,"%s",str);
            strcpy(input[noInWords],str);
printf("%s ..\n",str);
            noInWords++;
      }
      noInWords--;
}

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

void PredParser::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");
      }
}
```

Main function for constructing the parsing table and recognizing the sentence:

```
int main()
{
      int parsingStatus;
      PredParser pParser;
      FILE *fp,*fp1;
      char fname[30];
      fp=fopen("rules.txt","r");
      if(NULL == fp)
      {
            printf("%s:Error in file open.. Could not read rules from
            file\n");
            exit(-1);
      }
      fp1=fopen("input.txt","r");
      if(NULL == fp1)
      {
            printf("%s:Error in file open.. Could not read sentence from
            file\n");
            exit(-1);
      }
```

```
	pParser.readRules(fp);
	pParser.readInput(fp1);
	pParser.dispInput();
	pParser.constructTable();
	pParser.dispNTs();
	pParser.dispTs();
	pParser.dispFirst();
	pParser.dispFollow();
	pParser.dispRules();
	pParser.dispParsingTable();
	pParser.dispNTRuleNumberMapping();
	parsingStatus = pParser.recognize();
	if(parsingStatus == 1)
		printf("Successful Parsing\n");
	else
		printf("Error in Parsing\n");

	return 1;
}
```

Output

The grammar specification (non-left recursive) is given as follows:

E : T E1 ;

E1 : + T E1 ;

E1 : eps ;

T : F T1 ;

T1 : * F T1 ;

T1 : eps ;

F : (E) ;

F : id ;

F : const ;

The output of the predictive parser along with the intermediate results are as follows:

List of Non-terminals

E E1 T T1 F

List of Terminals

+ * ( ) id const #

First(E):->(id const

First(E1):->+ eps

First(T):->(id const

First(T1):->* eps

First(F):->(id const

Follow(E):->#)

Follow(E1):->#)

Follow(T):->+ #)

Follow(T1):->+ #)

Follow(F):->* + #)

Given Grammar with rule number:

0: E--> T E1

1: E1--> + T E1

2: E1--> eps

3: T--> F T1

4: T1--> * F T1

5: T1--> eps

6: F--> (E)

7: F--> id

8: F--> const

Parsing Table

NT	**+**	*****	**(**	**)**	**id**	**const**	**#**
E	-	-	0	-	0	0	-
E1	1	-	-	2	-	-	2

T	-	-	3	-	3	3	-
T1	5	4	-	5	-	-	5
F	-	-	6	-	7	8	-

---

*All the entries in the parsing table are rule number to be refered during the parsing. For efficient storage, the rule names are labled and dealt inside the program.

Non-terminal and Rule number mapping

E->0

E1->1 2

T->3

T1->4 5

F->6 7 8

The code is tested with the following arithemetic expression:

(id + id * id + id + id * id * id) #

This expression is parsed sucessfully.

Exercises for the students:

- Check the validity of the parsing table
- Use this parsing table to test some other arithmetic expression

---