**Practical-1**

**Aim:- Implementation of finite automata and string validation.**

**Code:-**

|  |
| --- |
| #include<stdio.h>  #include<stdlib.h>  struct node{  int id\_num;  int st\_val;  struct node \*link0;  struct node \*link1;  };  struct node \*start, \*q, \*ptr;  int vst\_arr[100], a[10];  int main(){  printf("prctical : 1 \n");  printf("Implementation of finite automata and string validation\n");  printf("Niraj Italiya\n");  printf("190130107041\n\n");  int count, i, posi, j;  char n[10];  printf("\n");  printf("Enter the number of states in the m/c:");  scanf("%d",&count);  q=(struct node \*)malloc(sizeof(struct node)\*count);  for(i=0;i<count;i++){  (q+i)->id\_num=i;  printf("State Machine::%d\n",i);  printf("Next State if i/p is 0:");  scanf("%d",&posi);  (q+i)->link0=(q+posi);  printf("Next State if i/p is 1:");  scanf("%d",&posi);  (q+i)->link1=(q+posi);  printf("Is the state final state(0/1)?");  scanf("%d",&(q+i)->st\_val);}  printf("Enter the Initial State of the m/c:");  scanf("%d",&posi);  start=q+posi;  printf("\n");  while(1){  printf("\n");  printf("Perform String Check(0/1):");  scanf("%d",&j);  if(j){  ptr=start;  printf("Enter the string of inputs:");  scanf("%s",n);  posi=0;  while(n[posi]!='\0'){  a[posi]=(n[posi]-'0');  //printf("%c\n",n[posi]);  //printf("%d",a[posi]);  posi++;  }  i=0;  printf("The visited States of the m/c are:");  do{  vst\_arr[i]=ptr->id\_num;  if(a[i]==0){  ptr=ptr->link0;  }  else if(a[i]==1){  ptr=ptr->link1;  }  else{  printf("iNCORRECT iNPUT\n");  return;  }  printf("[%d]",vst\_arr[i]);  i++;  }while(i<posi);  printf("\n");  printf("Present State:%d\n",ptr->id\_num);  printf("String Status:: ");  if(ptr->st\_val==1)  printf("String Accepted\n");  else  printf("String Not Accepted\n");  }  else  return 0;  }  printf("\n");  return 0;  } |

**Output** :

|  |
| --- |
| C:\Users\Bhavya Kansara\Documents\CD_niraj\1_a.PNG |

**Practical-2**

**Aim:-** **Introduction to Lex Tool**

**Lex:-**

Lex is a program that generates lexical analyzer.

It is used with YACC parser generator. The lexical analyzer is a program that transforms an input stream into a sequence of tokens.

It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

Lex is a program that automatically generates code for scanners.

Input: a description of the tokens in the form of regular expressions, together with the actions to be taken when each expression is matched.

Output: a text file with C source code defining a procedure yylex() that is a table implementing the DFA for the regular expressions.

**THE FUNCTION OF LEX IS AS FOLLOWS:**

Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.

Finally C compiler runs the lex.yy.c program and produces an object program a.out.

a.out is a lexical analyzer that transforms an input stream into a sequence of tokens.

**LEX FILE FORMAT**

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

{ definitions }

%%

{ rules }

%%

{ user subroutines }

Definitions include declarations of constant, variable and regular definitions.

Rules define the statement of form p1 {action1} p2 {action2}....pn {action}.

Where pi describes the regular expression and action1 describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

**Practical-5**

**Aim:- Implementation of Recursive Descent Parser without backtracking**

**Input: The string to be parsed.**

**Output: Whether string parsed successfully or not.**

**Explanation: Students have to implement the recursive procedure for RDP for a typical grammar. The production no. are displayed as they are used to derive the string.**

**Code:**

|  |
| --- |
| #include<stdio.h>  #include<ctype.h>  #include<string.h>  char input[100];  char prod[100][100];  int pos=-1,l,st=-1;  char id,num;  void E();  void T();  void F();  void advance();  void Td();  void Ed();  void advance()  {  pos++;  if(pos<l)  {  if(input[pos]>='0'&& input[pos]<='9')  {  num=input[pos];  id='\0';  }  if((input[pos]>='a' || input[pos]>='A')&&(input[pos]<='z' || input[pos]<='Z'))  {id=input[pos];  num='\0';  }  }  }  void E()  {  strcpy(prod[++st],"E->TE'");  T();  Ed();  }  void Ed()  {  int p=1;  if(input[pos]=='+')  {  p=0;  strcpy(prod[++st],"E'->+TE'");  advance();  T();  Ed();  }  if(input[pos]=='-')  { p=0;  strcpy(prod[++st],"E'->-TE'");  advance();  T();  Ed();  }  // Recursive Descent Parser  if(p==1)  { strcpy(prod[++st],"E'->null"); } }  void T()  { strcpy(prod[++st],"T->FT'");  F();  Td();  }  void Td()  {  int p=1;  if(input[pos]=='\*')  {  p=0;  strcpy(prod[++st],"T'->\*FT'");  advance();  F();  Td();  }  if(input[pos]=='/')  { p=0;  strcpy(prod[++st],"T'->/FT'");  advance();  F();  Td();  }  if(p==1)  strcpy(prod[++st],"T'->null");  }  void F()  {  if(input[pos]==id) {  strcpy(prod[++st],"F->id");  advance(); }  if(input[pos]=='(')  {  strcpy(prod[++st],"F->(E)");  advance();  E();  if(input[pos]==')') {  strcpy(prod[++st],"F->(E)");  advance(); }  }  if(input[pos]==num)  {  strcpy(prod[++st],"F->num");  advance();  }  }  int main()  {  printf("prctical : 5 \n");  printf("Implementation of Recursive Descent Parser without backtracking \nInput: The string to be parsed.\nOutput: Whether string parsed successfully or not. Explanation:\nStudents have to implement the recursive procedure for RDP for a typical grammar. The\nproduction no. are displayed as they are used to derive the string\n");  printf("Niraj Italiya\n");  printf("190130107041\n\n");  int i;  printf("Enter Input String :- ");  scanf("%s",input);  l=strlen(input);  input[l]='$';  advance();  E();  if(pos==l)  {  printf("String Accepted\n");  for(i=0;i<=st;i++)  {  printf("%s\n",prod[i]);  }  }  else  {  printf("String rejected\n");  }  return 0;  } |

**Output**:

|  |
| --- |
| C:\Users\Bhavya Kansara\Documents\CD_niraj\5.PNG |

**Practical-6**

**Aim:- Finding “First” set**

**Input: The string consists of grammar symbols.**

**Output: The First set for a given string.**

**Explanation: The student has to assume a typical grammar. The program when run will ask for the string to be entered. The program will find the First set of the given string**

**Code:**

|  |
| --- |
| #include<stdio.h>  #include<ctype.h>  void FIRST(char );  int count,n=0;  char prodn[10][10], first[10];  main()  {  printf("prctical : 6 \n");  printf("Finding 'First' set\nInput: The string consists of grammar symbols\nOutput: The First set for a given string.\nExplanation:\nThe student has to assume a typical grammar. The program when run will ask for the string\nto be entered. The program will find the First set of the given string\n");  printf("Niraj Italiya\n");  printf("190130107041\n\n");  int i,choice;  char c,ch;  printf("How many productions ? :");  scanf("%d",&count);  printf("Enter %d productions epsilon= $ :\n\n",count);  for(i=0;i<count;i++)  scanf("%s%c",prodn[i],&ch);  do  {  n=0;  printf("Element :");  scanf("%c",&c);  FIRST(c);  printf("\n FIRST(%c)= { ",c);  for(i=0;i<n;i++)  printf("%c ",first[i]);  printf("}\n");  printf("press 1 to continue : ");  scanf("%d%c",&choice,&ch);  }  while(choice==1);  }  void FIRST(char c)  {  int j;  if(!(isupper(c)))first[n++]=c;  for(j=0;j<count;j++)  {  if(prodn[j][0]==c)  {  if(prodn[j][2]=='$') first[n++]='$';  else if(islower(prodn[j][2]))first[n++]=prodn[j][2];  else FIRST(prodn[j][2]);  }  }  } |

**Output**:

|  |
| --- |
| C:\Users\Bhavya Kansara\Documents\CD_niraj\6.PNG |

**Practical-7**

**Aim:- Generate 3-tuple intermediate code for given infix expression**

**Code:-**

|  |
| --- |
| #include<stdio.h>  #include<conio.h>  #include<string.h>  int i=1,j=0,no=0,tmpch=90;  char str[100],left[15],right[15];  void findopr();  void explore();  void fleft(int);  void fright(int);  struct exp  {  int pos;  char op;  }  k[15];  void main()  {  printf("prctical : 7 \n");  printf("Generate 3-tuple intermediate code for given infix expression\n");  printf("Niraj Italiya\n");  printf("190130107041\n\n");  printf("\t\tINTERMEDIATE CODE GENERATION\n\n");  printf("Enter the Expression :");  scanf("%s",str);  printf("The intermediate code:\t\tExpression\n");  findopr();  explore();  getch();  }  void findopr() {  for(i=0;str[i]!='\0';i++)  if(str[i]==':'  {  k[j].pos=i  ;  k[j++].op=':'  ;  }  for(i=0;str[i]!='\0';i++  )  if(str[i]=='/'  )  {  k[j].pos=i  ;  k[j++].op='/'  ;  }  for(i=0;str[i]!='\0';i++  )  if(str[i]=='\*'  )  {  k[j].pos=i  ;  k[j++].op='\*'  ;  }  for(i=0;str[i]!='\0';i++  )  if(str[i]=='+'  )  {  k[j].pos=i  ;  k[j++].op='+'  ;  }  for(i=0;str[i]!='\0';i++  )  if(str[i]=='-')  {  k[j].pos=i  ;  k[j++].op='-';  }  }  void explore(  )  {  i=1  ;  while(k[i].op!='\0'  )  {  fleft(k[i].pos);  fright(k[i].pos);  str[k[i].pos]=tmpch--;  printf("\t%c := %s%c%s\t\t",str[k[i].pos],left,k[i].op,right);  for(j=0;j <strlen(str);j++)  if(str[j]!='$')  printf("%c",str[j]);  printf("\n");  i++;  }  fright(-1);  if(no==0)  {  fleft(strlen(str));  printf("\t%s := %s",right,left);  getch();  exit(0);  }  printf("\t%s := %c",right,str[k[--i].pos]);  getch();  }  void fleft(int x)  {  int w=0,flag=0;  x--;  while(x!= -1 &&str[x]!= '+' &&str[x]!='\*'&&str[x]!='='&&str[x]!='\0'&&str[x]!='-'&&str[x]!='/'&&str[x]!=':')  {  if(str[x]!='$'&& flag==0)  {  left[w++]=str[x];  left[w]='\0';  str[x]='$';  flag=1;  }  x--;  }  }  void fright(int x)  {  int w=0,flag=0;  x++;  while(x!= -1 && str[x]!= '+'&&str[x]!='\*'&&str[x]!='\0'&&str[x]!='='&&str[x]!=':'&&str[x]!='-'&&str[x]!='/')  {  if(str[x]!='$'&& flag==0)  {  right[w++]=str[x];  right[w]='\0';  str[x]='$';  flag=1;  }  x++;}} |

**Output**:

|  |
| --- |
| C:\Users\Bhavya Kansara\Documents\CD_niraj\7.PNG |

**Practical-8**

**Aim:- Extract Predecessor and Successor from given Control Flow Graph**

**Code:**

|  |
| --- |
| #include<iostream>  using namespace std;  // BST Node  struct Node  {  int key;  struct Node \*left, \*right;  };  // This function finds predecessor and successor of key in BST.  // It sets pre and suc as predecessor and successor respectively  void findPreSuc(Node\* root, Node\*& pre, Node\*& suc, int key)  {  // Base case  if (root == NULL) return ;  // If key is present at root  if (root->key == key)  {  // the maximum value in left subtree is predecessor  if (root->left != NULL)  {  Node\* tmp = root->left;  while (tmp->right)  tmp = tmp->right;  pre = tmp ;  }  // the minimum value in right subtree is successor  if (root->right != NULL)  {  Node\* tmp = root->right ;  while (tmp->left)  tmp = tmp->left ;  suc = tmp ;  }  return ;  }  // If key is smaller than root's key, go to left subtree  if (root->key > key)  {  suc = root ;  findPreSuc(root->left, pre, suc, key) ;  }  else // go to right subtree  {  pre = root ;  findPreSuc(root->right, pre, suc, key) ;  }  }  // A utility function to create a new BST node  Node \*newNode(int item)  {  Node \*temp = new Node;  temp->key = item;  temp->left = temp->right = NULL;  return temp;  }  /\* A utility function to insert a new node with given key in BST \*/  Node\* insert(Node\* node, int key)  {  if (node == NULL) return newNode(key);  if (key < node->key)  node->left = insert(node->left, key);  else  node->right = insert(node->right, key);  return node;  }  // Driver program to test above function  int main()  {  cout << "prctical : 8 \n";  cout << "Extract Predecessor and Successor from given Control Flow Graph\n";  cout << "Niraj Italiya\n";  cout << "190130107041\n\n";  int key = 75; //Key to be searched in BST  Node \*root = NULL;  root = insert(root, 50);  insert(root, 30);  insert(root, 20);  insert(root, 40);  insert(root, 70);  insert(root, 60);  insert(root, 80);  Node\* pre = NULL, \*suc = NULL;  findPreSuc(root, pre, suc, key);  if (pre != NULL)  cout << "Predecessor is " << pre->key << endl;  else  cout << "No Predecessor";  if (suc != NULL)  cout << "Successor is " << suc->key;  else  cout << "No Successor";  return 0;  } |

**Output**:

|  |
| --- |
| C:\Users\Bhavya Kansara\Documents\CD_niraj\8.PNG |

**Practical-9**

**Aim:- Introduction to YACC and generate Calculator Program**

**Introduction to YACC**

YACC (Yet Another Compiler Compiler) is a tool used to generate parser.

• A parser is a program that checks whether a given input meets a grammatical specification.

• YACC translates a given Context-free grammar in .y file and converts it to c implementation by creating y.tab.c and y.tab.h file.

We can get yacc with bison.

**$ sudo apt install bison**

**Practical-10**

**Aim:- Finding “Follow” set**

**Input: The string consists of grammar symbols.**

**Output: The Follow set for a given string.**

**Explanation: The student has to assume a typical grammar. The program when run will ask for the string to be entered. The program will find the Follow set of the given string.**

**Code:**

|  |
| --- |
| #include<stdio.h>  #include<string.h>  int n,m=0,p,i=0,j=0;  char a[10][10],f[10];  void follow(char c);  void first(char c);  int main()  {  printf("prctical : 10 \n");  printf("Finding “Follow” set\nInput: The string consists of grammar symbols\nOutput: The Follow set for a given string\nExplanation:\nThe student has to assume a typical grammar. The program when run will ask for the string\nto be entered. The program will find the Follow set of the given string.\n");  printf("Niraj Italiya\n");  printf("190130107041\n\n");  int i,z;  char c,ch;  printf("Enter the no.of productions:");  scanf("%d",&n);  printf("Enter the productions(epsilon=$):\n");  for(i=0;i<n;i++)  scanf("%s%c",a[i],&ch);  do  {  m=0;  printf("Enter the element whose FOLLOW is to be found:");  scanf("%c",&c);  follow(c);  printf("FOLLOW(%c) = { ",c)  ;  for(i=0;i<m;i++  )  printf("%c ",f[i])  ;  printf(" }\n")  ;  printf("Do you want to continue(0/1)?")  ;  scanf("%d%c",&z,&ch)  ;  }  while(z==1)  ;  }  void follow(char c  )  {  if(a[0][0]==c)f[m++]='$'  ;  for(i=0;i<n;i++  )  {  for(j=2;j<strlen(a[i]);j++  )  {  if(a[i][j]==c  )  {  if(a[i][j+1]!='\0')first(a[i][j+1]);  if(a[i][j+1]=='\0'&&c!=a[i][0]  )  follow(a[i][0])  ;  }  }}}  void first(char c  )  {  int k  ;  if(!(isupper(c)))f[m++]=c  ;  for(k=0;k<n;k++  )  {  if(a[k][0]==c  )  {  if(a[k][2]=='$') follow(a[i][0])  ;  else if(islower(a[k][2]))f[m++]=a[k][2];  else first(a[k][2]);  }  }  } |

**Output**:

|  |
| --- |
| C:\Users\Bhavya Kansara\Documents\CD_niraj\10.PNG |

**Practical-12**

**Aim:- Implement a C program for constructing LL (1) parsing.**

**Code:**

|  |
| --- |
| def removeLeftRecursion(rulesDiction):  store = {}    for lhs in rulesDiction:    alphaRules = []  betaRules = []    allrhs = rulesDiction[lhs]  for subrhs in allrhs:  if subrhs[0] == lhs:  alphaRules.append(subrhs[1:])  else:  betaRules.append(subrhs)    if len(alphaRules) != 0:    lhs\_ = lhs + "'"  while (lhs\_ in rulesDiction.keys()) \  or (lhs\_ in store.keys()):  lhs\_ += "'"    for b in range(0, len(betaRules)):  betaRules[b].append(lhs\_)  rulesDiction[lhs] = betaRules    for a in range(0, len(alphaRules)):  alphaRules[a].append(lhs\_)  alphaRules.append(['#'])    store[lhs\_] = alphaRules    for left in store:  rulesDiction[left] = store[left]  return rulesDiction      def LeftFactoring(rulesDiction):  newDict = {  for lhs in rulesDiction:  allrhs = rulesDiction[lhs]  temp = dict()  for subrhs in allrhs:  if subrhs[0] not in list(temp.keys()):  temp[subrhs[0]] = [subrhs]  else:  temp[subrhs[0]].append(subrhs)  new\_rule = []  for term\_key in temp:  from temp for term\_key  allStartingWithTermKey = temp[term\_key]  if len(allStartingWithTermKey) > 1:  lhs\_ = lhs + "'"  while (lhs\_ in rulesDiction.keys()) \  or (lhs\_ in tempo\_dict.keys()):  lhs\_ += "'"  new\_rule.append([term\_key, lhs\_])  ex\_rules = []  for g in temp[term\_key]:  ex\_rules.append(g[1:])  tempo\_dict[lhs\_] = ex\_rules  else:  new\_rule.append(allStartingWithTermKey[0])  newDict[lhs] = new\_rule  for key in tempo\_dict:  newDict[key] = tempo\_dict[key]  return newDict  def first(rule):  global rules, nonterm\_userdef, \  term\_userdef, diction, firsts  if len(rule) != 0 and (rule is not None):  if rule[0] in term\_userdef:  return rule[0]  elif rule[0] == '#':  return '#'  if len(rule) != 0:  if rule[0] in list(diction.keys()):  fres = []  rhs\_rules = diction[rule[0]]  for itr in rhs\_rules:  indivRes = first(itr)  if type(indivRes) is list:  for i in indivRes:  fres.append(i)  else:  fres.append(indivRes)  if '#' not in fres:  return fres  else:  newList = []  fres.remove('#')  if len(rule) > 1:  ansNew = first(rule[1:])  if ansNew != None:  if type(ansNew) is list:  newList = fres + ansNew  else:  newList = fres + [ansNew]  else:  newList = fres  return newList  fres.append('#')  return fres  def follow(nt):  global start\_symbol, rules, nonterm\_userdef, \  term\_userdef, diction, firsts, follows  solset = set()  if nt == start\_symbol:  solset.add('$')    for curNT in diction:  rhs = diction[curNT]  for subrule in rhs:  if nt in subrule:  while nt in subrule:  index\_nt = subrule.index(nt)  subrule = subrule[index\_nt + 1:]  if len(subrule) != 0:  res = first(subrule)  if '#' in res:  newList = []  res.remove('#')  ansNew = follow(curNT)  if ansNew != None:  if type(ansNew) is list:  newList = res + ansNew  else:  newList = res + [ansNew]  else:  newList = res  res = newList  else:  if nt != curNT:  res = follow(curNT)    if res is not None:  if type(res) is list:  for g in res:  solset.add(g)  else:  solset.add(res)  return list(solset)      def computeAllFirsts():  global rules, nonterm\_userdef, \  term\_userdef, diction, firsts  for rule in rules:  k = rule.split("->")  k[0] = k[0].strip()  k[1] = k[1].strip()  rhs = k[1]  multirhs = rhs.split('|')  for i in range(len(multirhs)):  multirhs[i] = multirhs[i].strip()  multirhs[i] = multirhs[i].split()  diction[k[0]] = multirhs    print(f"\nRules: \n")  for y in diction:  print(f"{y}->{diction[y]}")  print(f"\nAfter elimination of left recursion:\n")    diction = removeLeftRecursion(diction)  for y in diction:  print(f"{y}->{diction[y]}")  print("\nAfter left factoring:\n")    diction = LeftFactoring(diction)  for y in diction:  print(f"{y}->{diction[y]}")    for y in list(diction.keys()):  t = set()  for sub in diction.get(y):  res = first(sub)  if res != None:  if type(res) is list:  for u in res:  t.add(u)  else:  t.add(res)    firsts[y] = t    print("\nCalculated firsts: ")  key\_list = list(firsts.keys())  index = 0  for gg in firsts:  print(f"first({key\_list[index]}) "  f"=> {firsts.get(gg)}")  index += 1      def computeAllFollows():  global start\_symbol, rules, nonterm\_userdef,\  term\_userdef, diction, firsts, follows  for NT in diction:  solset = set()  sol = follow(NT)  if sol is not None:  for g in sol:  solset.add(g)  follows[NT] = solset    print("\nCalculated follows: ")  key\_list = list(follows.keys())  index = 0  for gg in follows:  print(f"follow({key\_list[index]})"  f" => {follows[gg]}")  index += 1      # create parse table  def createParseTable():  import copy  global diction, firsts, follows, term\_userdef  print("\nFirsts and Follow Result table\n")    # find space size  mx\_len\_first = 0  mx\_len\_fol = 0  for u in diction:  k1 = len(str(firsts[u]))  k2 = len(str(follows[u]))  if k1 > mx\_len\_first:  mx\_len\_first = k1  if k2 > mx\_len\_fol:  mx\_len\_fol = k2    print(f"{{:<{10}}} "  f"{{:<{mx\_len\_first + 5}}} "  f"{{:<{mx\_len\_fol + 5}}}"  .format("Non-T", "FIRST", "FOLLOW"))  for u in diction:  print(f"{{:<{10}}} "  f"{{:<{mx\_len\_first + 5}}} "  f"{{:<{mx\_len\_fol + 5}}}"  .format(u, str(firsts[u]), str(follows[u])))    ntlist = list(diction.keys())  terminals = copy.deepcopy(term\_userdef)  terminals.append('$')  mat = []  for x in diction:  row = []  for y in terminals:  row.append('')  mat.append(row)    grammar\_is\_LL = True  for lhs in diction:  rhs = diction[lhs]  for y in rhs:  res = first(y)  if '#' in res:  if type(res) == str:  firstFollow = []  fol\_op = follows[lhs]  if fol\_op is str:  firstFollow.append(fol\_op)  else:  for u in fol\_op:  firstFollow.append(u)  res = firstFollow  else:  res.remove('#')  res = list(res) +\  list(follows[lhs])  ttemp = []  if type(res) is str:  ttemp.append(res)  res = copy.deepcopy(ttemp)  for c in res:  xnt = ntlist.index(lhs)  yt = terminals.index(c)  if mat[xnt][yt] == '':  mat[xnt][yt] = mat[xnt][yt] \  + f"{lhs}->{' '.join(y)}"  else:  if f"{lhs}->{y}" in mat[xnt][yt]:  continue  else:  grammar\_is\_LL = False  mat[xnt][yt] = mat[xnt][yt] \  + f",{lhs}->{' '.join(y)}"    print("\nGenerated parsing table:\n")  frmt = "{:>12}" \* len(terminals)  print(frmt.format(\*terminals))    j = 0  for y in mat:  frmt1 = "{:>12}" \* len(y)  print(f"{ntlist[j]} {frmt1.format(\*y)}")  j += 1    return (mat, grammar\_is\_LL, terminals)      def validateStringUsingStackBuffer(parsing\_table, grammarll1,  table\_term\_list, input\_string,  term\_userdef,start\_symbol):    print(f"\nValidate String => {input\_string}\n")  if grammarll1 == False:  return f"\nInput String = " \  f"\"{input\_string}\"\n" \  f"Grammar is not LL(1)"  stack = [start\_symbol, '$']  buffer = []  input\_string = input\_string.split()  input\_string.reverse()  buffer = ['$'] + input\_string    print("{:>20} {:>20} {:>20}".  format("Buffer", "Stack","Action"))    while True:  if stack == ['$'] and buffer == ['$']:  print("{:>20} {:>20} {:>20}"  .format(' '.join(buffer),  ' '.join(stack),  "Valid"))  return "\nValid String!"  elif stack[0] not in term\_userdef:  x = list(diction.keys()).index(stack[0])  y = table\_term\_list.index(buffer[-1])  if parsing\_table[x][y] != '':  entry = parsing\_table[x][y]  print("{:>20} {:>20} {:>25}".  format(' '.join(buffer),  ' '.join(stack),  f"T[{stack[0]}][{buffer[-1]}] = {entry}"))  lhs\_rhs = entry.split("->")  lhs\_rhs[1] = lhs\_rhs[1].replace('#', '').strip()  entryrhs = lhs\_rhs[1].split()  stack = entryrhs + stack[1:]  else:  return f"\nInvalid String! No rule at " \  f"Table[{stack[0]}][{buffer[-1]}]."  else:  if stack[0] == buffer[-1]:  print("{:>20} {:>20} {:>20}"  .format(' '.join(buffer),  ' '.join(stack),  f"Matched:{stack[0]}"))  buffer = buffer[:-1]  stack = stack[1:]  else:  return "\nInvalid String! " \  "Unmatched terminal symbols"    sample\_input\_string = None  rules=["S -> A k O",  "A -> A d | a B | a C",  "C -> c",  "B -> b B C | r"]    nonterm\_userdef=['A','B','C']  term\_userdef=['k','O','d','a','c','b','r']  sample\_input\_string="a r k O"    diction = {}  firsts = {}  follows = {}  print("Niraj italiya")  print("190130107041")  computeAllFirsts()  start\_symbol = list(diction.keys())[0]  computeAllFollows()    (parsing\_table, result, tabTerm) = createParseTable()    if sample\_input\_string != None:  validity = validateStringUsingStackBuffer(parsing\_table, result,  tabTerm, sample\_input\_string,  term\_userdef,start\_symbol)  print(validity)  else:  print("\nNo input String detected") |

**Output:-**

|  |
| --- |
|  |

**Practical-12**

**Aim:- Implement a C program to implement LALR parsing.**

**Code:**

|  |
| --- |
| #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 ister(char);  int isnter(char);  int isstate(char);  void error();  void isreduce(char,char);  char pop(char \*,int \*);  void printt(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 ter[6]={'i','+','\*',')','(','$'};  char nter[3]={'E','T','F'};  char states[12]={'a','b','c','d','e','f','g','h','m','j','k','l'};  char stack[100];  int top=-1;  char temp[10];  struct grammar  {  char left;  char right[5];  };  const struct grammar rl[6]={  {'E',"e+T"},  {'E',"T"},  {'T',"T\*F"},  {'T',"F"},  {'F',"(E)"},  {'F',"i"},  };  int main() {  printf("prctical : 12 \n");  printf("Implement a C program to implement LALR parsing.\n");  printf("Niraj Italiya\n");  printf("190130107041\n\n");  char inp[80],x,p,dl[80],y,bl='a';  int i=0,j,k,l,n,m,c,len;  printf(" Enter the input :");  scanf("%s",inp);  len=strlen(inp);  inp[len]='$';  inp[len+1]='\0';  push(stack,&top,bl);  printf("\n stack\t\t\t input");  printt(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]);  }  }  }  printt(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 ");  return 0;  }  void push(char \*s,int \*sp,char item)  {  if(\*sp==100)  printf(" stack is full ");  else  {  \*sp=\*sp+1;  s[\*sp]=item;  }  }  char stacktop(char \*s) {  char i;  i=s[top];  return i;  }  void isproduct(char x,char p) {  int k,l;  k=ister(x);  l=isstate(p);  strcpy(temp,A[l  -1].row[k  -1]);  }  int ister(char x) {  int i;  for(i=0;i<6;i++)  if(x==ter[i])  return i+1;  return 0;  }  int isnter(char x) {  int i;  for(i=0;i<3;i++)  if(x==nter[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=isnter(p);  strcpy(temp,G[k  -1].r[l  -1]);  }  char pop(char \*s,int \*sp) {  char item;  if(\*sp==  -1)  printf(" stack is empty ");  else {  item=s[\*sp];  \*sp=\*sp  -1;  }  return item;  }  void printt(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;  }  } |

**Output**:

|  |
| --- |
| C:\Users\Bhavya Kansara\Documents\CD_niraj\12.PNG |

**Practical-13**

**Aim:- Implement a C program to implement operator precedence parsing.**

**Code:**

|  |
| --- |
| #include<stdio.h>  #include<conio.h>  #include<string.h>  char \*input;  int i=0;  char  lasthandle[6],stack[50],handles[][5]={")E(","E\*E","E+E","i","E^E"  }; //(E) becomes )E( when pushed to stack  int top=0,l;  char prec[9][9]={  /\*input\*/  /\*stack + - \* / ^ i ( ) $ \*/  /\* + \*/ '>', '>','<','<','<','<','<','>','>',  /\* - \*/ '>', '>','<','<','<','<','<','>','>',  /\* \* \*/ '>', '>','>','>','<','<','<','>','>',  /\* / \*/ '>', '>','>','>','<','<','<','>','>',  /\* ^ \*/ '>', '>','>','>','<','<','<','>','>',  /\* i \*/ '>', '>','>','>','>','e','e','>','>',  /\* ( \*/ '<', '<','<','<','<','<','<','>','e',  /\* ) \*/ '>', '>','>','>','>','e','e','>','>',  /\* $ \*/ '<', '<','<','<','<','<','<','<','>',  };  int getindex(char c)  {  switch(c)  {  case '+':return 0;  case '-':return 1;  case '\*':return 2;  case '/':return 3;  case '^':return 4;  case 'i':return 5;  case '(':return 6;  case ')':return 7;  case '$':return 8;  }  }  int shift()  {  stack[++top]=\*(input+i++);  stack[top+1]='\0';  }  int reduce()  {  int i,len,found,t;  for(i=0;i<5;i++)//selecting handles  {  len=strlen(handles[i]);  if(stack[top]==handles[i][0]&&top+1>=len)  {  found=1;  for(t=0;t<len;t++)  {  if(stack[top-t]!=handles[i][t])  {  found=0;  break;  }  }  if(found==1)  {  stack[top-t+1]='E';  top=top-t+1;  strcpy(lasthandle,handles[i]);  stack[top+1]='\0';  return 1;//successful reduction  }  }  }  return 0;  }  void dispstack()  {  int j;  for(j=0;j<=top;j++)  printf("%c",stack[j]);  }  void dispinput()  {  int j;  for(j=i;j<l;j++)  printf("%c",\*(input+j));  }  void main()  {  printf("prctical : 13 \n");  printf("Implement a C program to implement operator precedence parsing.\n");  printf("Niraj Italiya\n");  printf("190130107041\n\n");  int j;  input=(char\*)malloc(50\*sizeof(char));  printf("\nEnter the string :-\n");  scanf("%s",input);  input=strcat(input,"$");  l=strlen(input);  strcpy(stack,"$");  printf("\nSTACK\tINPUT\tACTION");  while(i<=l)  {  shift();  printf("\n");  dispstack();  printf("\t");  dispinput();  printf("\tShift");  if(prec[getindex(stack[top])][getindex(input[i])]=='>')  {  while(reduce())  {  printf("\n");  dispstack();  printf("\t");  dispinput();  printf("\tReduced: E->%s",lasthandle);  }  }  }  if(strcmp(stack,"$E$")==0)  printf("\nAccepted;");  else  printf("\nNot Accepted;");  getch();  } |

**Output**:

|  |
| --- |
| C:\Users\Bhavya Kansara\Documents\CD_niraj\13.PNG |