Experiment 1

Aim: Implementation of Lexical Analyzer

Theory: A lexical analyzer, also known as a lexer or tokenizer, is a fundamental component of a compiler or interpreter. Its primary task is to break down the source code into a sequence of tokens, which are meaningful units such as keywords, identifiers, operators, and literals. The process of tokenizing the source code is known as lexical analysis.

Code:

```
#include<bits/stdc++.h>
using namespace std;

string arr[] = {
  "void",
  "using",
  "namespace",
  "int",
  "include",
  "iostream",
  "std",
  "main",
  "cin",
  "cout",
  "return",
  "float",
  "double",
  "string"
};

string test_code = "#include <stdio.h> "
"void main ( ) "
"{ "
"int x = 6 ; "
"int y = 4 ; "
"x = x + y ; "
"printf(\"%d\", x); "
"}";
istringstream iss(test_code);
string s;
bool isKeyword(string a);

int main() {
```

```
  cout << "Drish (RA2011030030026)\n\n";
  while (iss >> s) {
    if (s == "+" || s == "-" || s == "" || s == "/" || s == "^" || s ==
"&&" || s == "||" || s == "=" || s == "==" || s == "&" || s == "|" || s
== "%" || s == "++" || s == "--" || s == "+=" || s == "-=" || s == "/="
|| s == "=" || s == "%=") {
      cout << s << " is an operator\n";
      s = "";
    } else if (isKeyword(s)) {
      cout << s << " is a keyword\n";
      s = "";
    } else if (s == "(" || s == "{" || s == "[" || s == ")" || s == "}"
|| s == "]" || s == "<" || s == ">" || s == "()" || s == ";" || s == "<<"
|| s == ">>" || s == "," || s == "#") {
      cout << s << " is a symbol\n";
      s = "";
    } else if (s == "\n" || s == " " || s == "") {
      s = "";
    } else if (isdigit(s[0])) {
      int x = 0;
      if (!isdigit(s[x++])) {
        continue;
      } else {
        cout << s << " is a constant\n";
        s = "";
      }
    } else {
      cout << s << " is an identifier\n";
      s = "";
    }
  }
  return 0;
}

bool isKeyword(string a) {
  for (int i = 0; i < 14; i++) {
    if (arr[i] == a) {
      return true;
    }
  }
  return false;
}
```

Output:

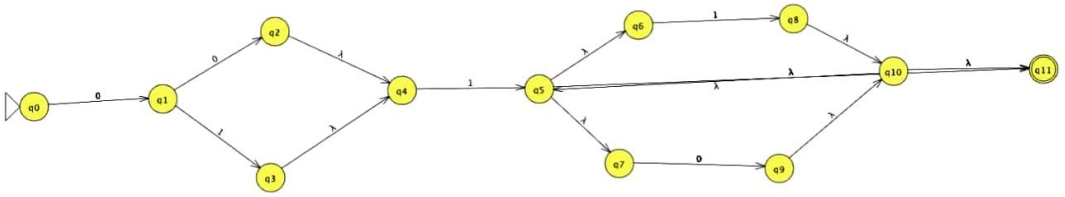
```
Drish (RA2011030030026)

#include is an identifier
<stdio.h> is an identifier
void is a keyword
main is a keyword
( is a symbol
) is a symbol
{ is a symbol
int is a keyword
x is an identifier
= is an operator
6 is a constant
; is a symbol
int is a keyword
y is an identifier
= is an operator
4 is a constant
; is a symbol
x is an identifier
= is an operator
x is an identifier
+ is an operator
y is an identifier
; is a symbol
printf("%d", is an identifier
x); is an identifier
} is a symbol
```

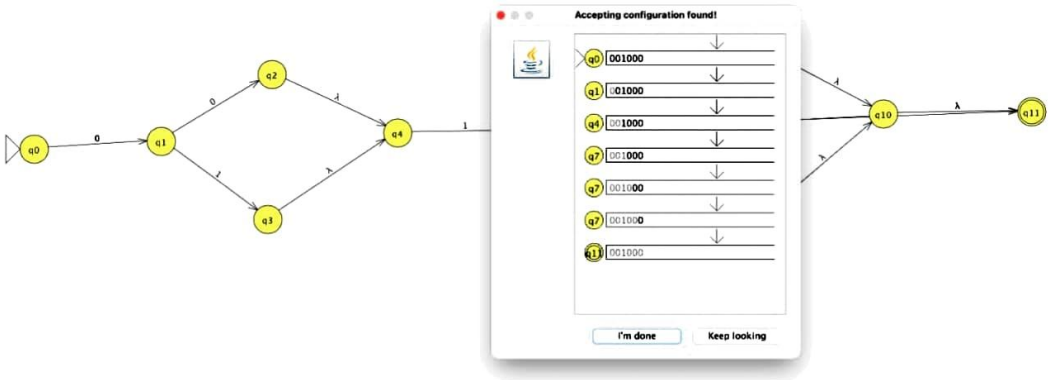
Experiment 2

Aim: Conversion of Regular Expression to NFA

Theory: Nondeterministic Finite Automaton (NFA) and Regular Expressions are key concepts in formal language theory. An NFA is a model that recognizes regular languages, comprising states, transitions, an initial state, and final states. It allows multiple transitions on the same input symbol and epsilon transitions. Regular expressions, on the other hand, are symbolic representations of patterns used for pattern matching and text searching. They employ metacharacters and constructs to define pattern sets. NFAs and regular expressions are closely related, as regular expressions can be converted into equivalent NFAs and NFAs can be transformed into DFAs. They provide powerful tools for pattern recognition and text processing, enabling efficient searching and manipulation of strings based on predefined patterns.

Output:





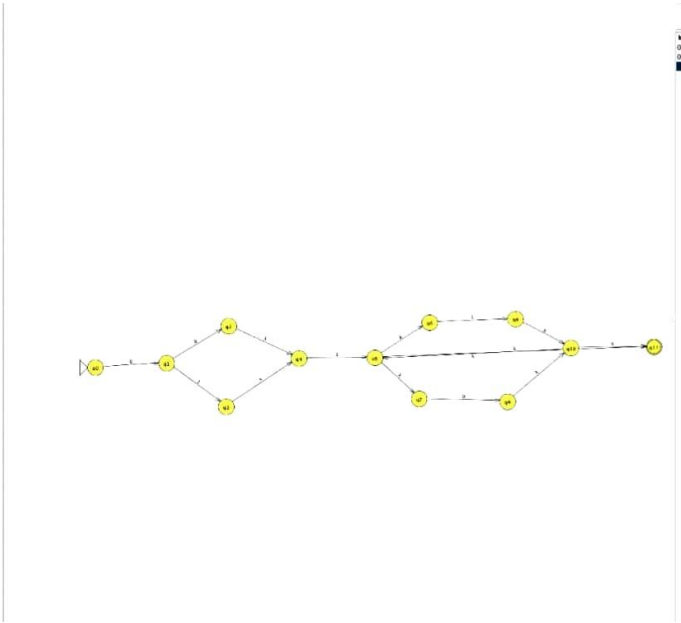
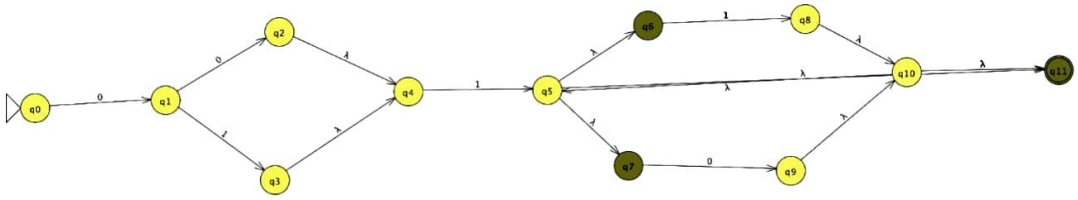
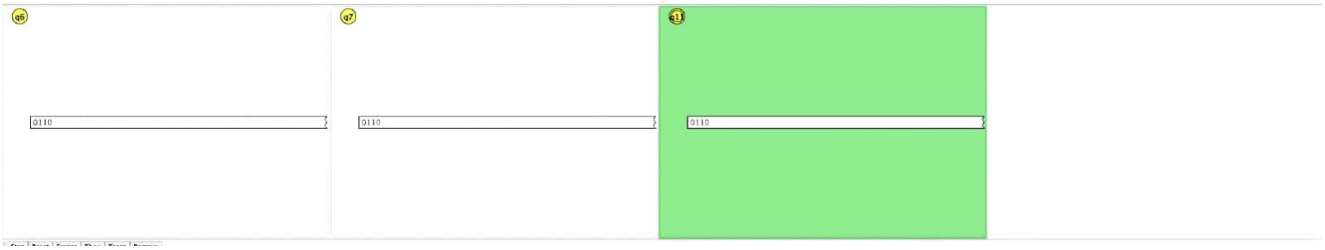


Table Text Size

Input	Result
001	Accept
011	Accept
	Reject



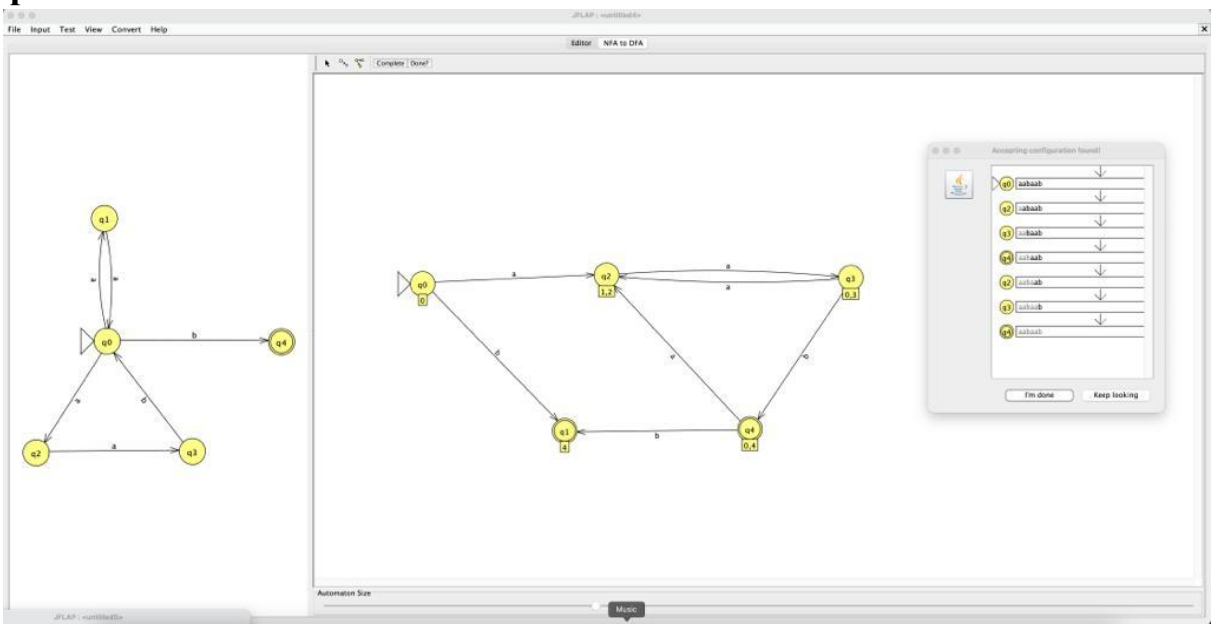


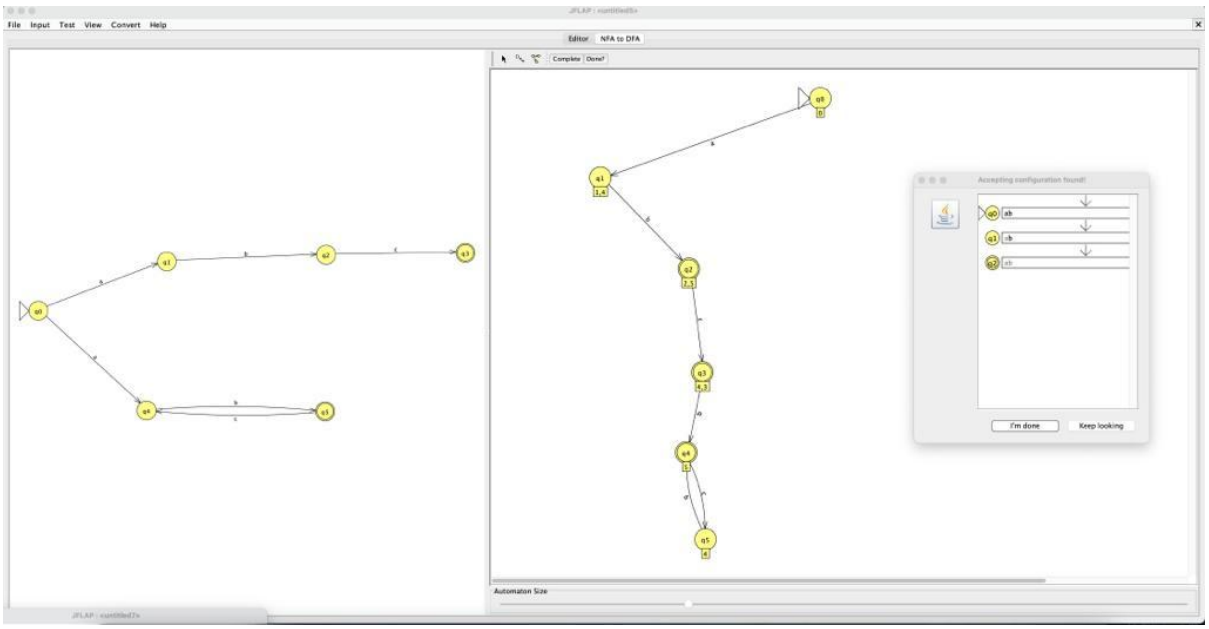
Experiment 3

Aim: Conversion from NFA to DFA

Theory: NFAs and DFAs are models of computation used in formal language theory. An NFA consists of states, transitions, an initial state, and one or more final states. It allows multiple transitions on the same symbol and epsilon transitions. In contrast, a DFA has unique transitions on each symbol from every state. NFAs are more expressive but potentially ambiguous, while DFAs are deterministic and unambiguous. They both recognize regular languages and can be converted into each other using specific algorithms. NFAs provide flexibility but may require more states, while DFAs offer simplicity and efficient processing. Understanding and using NFAs and DFAs are fundamental in formal language theory for analyzing and processing regular languages.

Output:





Experiment 4

Aim: Elimination of Ambiguity Left Recursion and Left Factoring

Theory: Left recursion and left factoring are techniques used to eliminate ambiguity in grammars. Left recursion occurs when a non-terminal can directly or indirectly produce a derivation that starts with itself, leading to parsing conflicts. To eliminate left recursion, we create new non-terminals and modify productions. For example, given a production A -> Aα, we create a new non-terminal A' and rewrite it as A -> αA'. Left factoring addresses common prefixes in alternatives of a production, which can cause parsing ambiguity. It involves splitting the production into two or more productions, factoring out the common prefix. These techniques are crucial for building unambiguous grammars and efficient parsing algorithms. By removing ambiguity, parsers can determine the correct interpretation of the input and facilitate language processing.

Code:

```
#include<bits/stdc++.h>

struct ProductionRule {
  std::string nonTerminal;
  std::vector < std::string > expansions;
};

void eliminateLeftRecursion(std::vector < ProductionRule > & grammar) {
  std::vector < ProductionRule > newGrammar;

  for (const auto & rule: grammar) {
    std::vector < std::string > recursiveExpansions;
    std::vector < std::string > nonRecursiveExpansions;

    for (const auto & expansion: rule.expansions) {
      if (expansion[0] == rule.nonTerminal[0]) {
        recursiveExpansions.push_back(expansion.substr(1));
      } else {
        nonRecursiveExpansions.push_back(expansion);
      }
    }

    if (!recursiveExpansions.empty()) {
      std::string newNonTerminal = rule.nonTerminal + "'";
      std::string newExpansion;

      for (const auto & recursiveExp: recursiveExpansions) {
```

```
        newExpansion += recursiveExp + newNonTerminal + "|";
      }
      newExpansion += "epsilon";

      newGrammar.push_back({
        rule.nonTerminal,
        nonRecursiveExpansions
      });
      newGrammar.push_back({
        newNonTerminal,
        {
          newExpansion
        }
      });
    } else {
      newGrammar.push_back(rule);
    }
  }
  grammar = newGrammar;
}
void eliminateLeftFactoring(std::vector < ProductionRule > & grammar) {
  std::vector < ProductionRule > newGrammar;

  for (const auto & rule: grammar) {
    std::vector < std::string > expansions = rule.expansions;

    while (!expansions.empty()) {
      std::string currentExpansion = expansions[0];
      std::vector < std::string > commonPrefixes;

      for (size_t i = 1; i < expansions.size(); ++i) {
        size_t j = 0;
        while (j < currentExpansion.length() && j <
expansions[i].length() && currentExpansion[j] == expansions[i][j]) {
          ++j;
        }
        commonPrefixes.push_back(currentExpansion.substr(0, j));
      }

      if (!commonPrefixes.empty()) {

        std::string longestPrefix = *
std::max_element(commonPrefixes.begin(), commonPrefixes.end(),
          [](const std::string & prefix1,
            const std::string & prefix2) {
```

```
            return prefix1.length() < prefix2.length();
          });

        std::string newNonTerminal = rule.nonTerminal + "'";
        std::vector < std::string > newExpansions;

        for (const auto & expansion: expansions) {
          if (expansion.substr(0, longestPrefix.length()) ==
longestPrefix) {

newExpansions.push_back(expansion.substr(longestPrefix.length()));
          } else {
            newExpansions.push_back(expansion);
          }
        }

        newExpansions.push_back(newNonTerminal);

        newGrammar.push_back({
          rule.nonTerminal,
          newExpansions
        });
      } else {
        newGrammar.push_back(rule);
      }

      expansions.erase(expansions.begin());
    }
  }

  grammar = newGrammar;
}
int main() {
  std::cout << "Drish (RA2011030030026)\n\n";
  std::vector < ProductionRule > recgrammar = {
    {
      "S",
      {
        "S+A",
        "A"
      }
    },
    {
      "A",
      {
```

```
      "a"
    }
  }
};

std::cout << "Original Grammar:" << std::endl;
for (const auto & rule: recgrammar) {
  std::cout << rule.nonTerminal << " -> ";
  for (const auto & expansion: rule.expansions) {
    std::cout << expansion << " | ";
  }
  std::cout << std::endl;
}

eliminateLeftRecursion(recgrammar);

std::cout << "Grammar after eliminating left recursion:" << std::endl;
for (const auto & rule: recgrammar) {
  std::cout << rule.nonTerminal << " -> ";
  for (const auto & expansion: rule.expansions) {
    std::cout << expansion << " | ";
  }
  std::cout << std::endl;
}

std::vector < ProductionRule > facgrammar = {
  {
    "S",
    {
      "S+A",
      "S-B",
      "A"
    }
  },
  {
    "A",
    {
      "a",
      "b"
    }
  }
};

std::cout << "Original Grammar:" << std::endl;
for (const auto & rule: facgrammar) {
```

```
    std::cout << rule.nonTerminal << " -> ";
    for (const auto & expansion: rule.expansions) {
      std::cout << expansion << " | ";
    }
    std::cout << std::endl;
  }
  eliminateLeftFactoring(facgrammar);

  std::cout << "Grammar after eliminating left factoring:" << std::endl;
  for (const auto & rule: facgrammar) {
    std::cout << rule.nonTerminal << " -> ";
    for (const auto & expansion: rule.expansions) {
      std::cout << expansion << " | ";
    }
    std::cout << std::endl;
  }
  return 0;
}
```

Output:

```
Drish (RA2011030030026)

Original Grammar:
S -> S+A | A |
A -> a |
Grammar after eliminating left recursion:
S -> A |
S' -> +AS'|epsilon |
A -> a |
Original Grammar:
S -> S+A | S-B | A |
A -> a | b |
Grammar after eliminating left factoring:
S -> +A | -B | A | S' |
S -> S-B | A | S' |
S -> S+A | S-B | A |
A -> a | b | A' |
A -> a | b |
```

Experiment 5

Aim: FIRST and FOLLOW computation

Theory: FIRST and FOLLOW sets are used in parsing and grammar analysis. The FIRST set of a non-terminal or sequence of symbols is the set of terminals that can appear as the first symbol(s) of a derivation. The FOLLOW set of a non-terminal is the set of terminals that can appear immediately after occurrences of the non-terminal. The FIRST set is computed by considering terminals, non-terminals, and sequences of symbols, while the FOLLOW set is determined by the position of the non-terminal in productions. These sets are important in parsing algorithms to resolve conflicts and make parsing decisions. They ensure efficient and accurate analysis of the grammar, aiding in parsing and language processing.

Code:

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>

void followfirst(char, int, int);
void follow(char c);

void findfirst(char, int, int);

int count, n = 0;

char calc_first[10][100];

char calc_follow[10][100];
int m = 0;

char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char ** argv) {
    printf("Drish (RA2011030030026)\n\n");
    int jm = 0;
    int km = 0;
    int i, choice;
    char c, ch;
```

```
count = 8;

strcpy(production[0], "E=TR");
strcpy(production[1], "R=+TR");
strcpy(production[2], "R=#");
strcpy(production[3], "T=FY");
strcpy(production[4], "Y=*FY");
strcpy(production[5], "Y=#");
strcpy(production[6], "F=(E)");
strcpy(production[7], "F=i");

int kay;
char done[count];
int ptr = -1;

for (k = 0; k < count; k++) {
    for (kay = 0; kay < 100; kay++) {
        calc_first[k][kay] = '!';
    }
}
int point1 = 0, point2, xxx;

for (k = 0; k < count; k++) {
    c = production[k][0];
    point2 = 0;
    xxx = 0;
    for (kay = 0; kay <= ptr; kay++)
        if (c == done[kay])
            xxx = 1;
    if (xxx == 1)
        continue;
    findfirst(c, 0, 0);
    ptr += 1;
    done[ptr] = c;
    printf("\n First(%c) = { ", c);
    calc_first[point1][point2++] = c;

    for (i = 0 + jm; i < n; i++) {
        int lark = 0, chk = 0;

        for (lark = 0; lark < point2; lark++) {
            if (first[i] == calc_first[point1][lark]) {
                chk = 1;
                break;
            }
```

```
        }
        if (chk == 0) {
            printf("%c, ", first[i]);
            calc_first[point1][point2++] = first[i];
        }
    }
    printf("}\n");
    jm = n;
    point1++;
}
printf("\n");
printf("-----------------------------------------------\n\n");
char donee[count];
ptr = -1;

for (k = 0; k < count; k++) {
    for (kay = 0; kay < 100; kay++) {
        calc_follow[k][kay] = '!';
    }
}
point1 = 0;
int land = 0;
for (e = 0; e < count; e++) {
    ck = production[e][0];
    point2 = 0;
    xxx = 0;

    for (kay = 0; kay <= ptr; kay++)
        if (ck == donee[kay])
            xxx = 1;

    if (xxx == 1)
        continue;
    land += 1;

    follow(ck);
    ptr += 1;

    donee[ptr] = ck;
    printf(" Follow(%c) = { ", ck);
    calc_follow[point1][point2++] = ck;

    for (i = 0 + km; i < m; i++) {
        int lark = 0, chk = 0;
        for (lark = 0; lark < point2; lark++) {
```

```
                if (f[i] == calc_follow[point1][lark]) {
                    chk = 1;
                    break;
                }
            }
            if (chk == 0) {
                printf("%c, ", f[i]);
                calc_follow[point1][point2++] = f[i];
            }
        }
        printf(" }\n\n");
        km = m;
        point1++;
    }
}

void follow(char c) {
    int i, j;

    if (production[0][0] == c) {
        f[m++] = '$';
    }
    for (i = 0; i < 10; i++) {
        for (j = 2; j < 10; j++) {
            if (production[i][j] == c) {
                if (production[i][j + 1] != '\0') {
                    followfirst(production[i][j + 1], i, (j + 2));
                }

                if (production[i][j + 1] == '\0' && c !=
production[i][0]) {
                    follow(production[i][0]);
                }
            }
        }
    }
}

void findfirst(char c, int q1, int q2) {
    int j;
    if (!(isupper(c))) {
        first[n++] = c;
    }
    for (j = 0; j < count; j++) {
        if (production[j][0] == c) {
```

```
            if (production[j][2] == '#') {
                if (production[q1][q2] == '\0')
                    first[n++] = '#';
                else if (production[q1][q2] != '\0' &&
                    (q1 != 0 || q2 != 0)) {

                    findfirst(production[q1][q2], q1, (q2 + 1));
                } else
                    first[n++] = '#';
            } else if (!isupper(production[j][2])) {
                first[n++] = production[j][2];
            } else {
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

void followfirst(char c, int c1, int c2) {
    int k;

    if (!(isupper(c)))
        f[m++] = c;
    else {
        int i = 0, j = 1;
        for (i = 0; i < count; i++) {
            if (calc_first[i][0] == c)
                break;
        }
        while (calc_first[i][j] != '!') {
            if (calc_first[i][j] != '#') {
                f[m++] = calc_first[i][j];
            } else {
                if (production[c1][c2] == '\0') {

                    follow(production[c1][0]);
                } else {
                    followfirst(production[c1][c2], c1, c2 + 1);
                }
            }
            j++;
        }
    }
}
```

Output:

```
Drish (RA2011030030026)

 First(E) = { (, i, }
 First(R) = { +, #, }
 First(T) = { (, i, }
 First(Y) = { *, #, }
 First(F) = { (, i, }

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

 Follow(E) = { $, ),  }
 Follow(R) = { $, ),  }
 Follow(T) = { +, $, ),  }
 Follow(Y) = { +, $, ),  }
 Follow(F) = { *, +, $, ),  }
```

Experiment 6

Aim: Predictive Parsing Table

Theory: A predictive parser is a top-down parsing technique that uses a predictive parsing table to decide which production to apply based on the current input symbol and the stack. The parsing table is constructed using the FIRST and FOLLOW sets of the grammar. The table guides the parser's decisions by matching the non-terminal and input symbol. If there are no conflicts or ambiguities in the table, the parser can predictively choose the right production without backtracking. This approach enables efficient parsing without requiring trial and error. The predictive parsing table serves as a roadmap for the parser, ensuring a systematic and deterministic parsing process.

Code:

```
#include<bits/stdc++.h>
using namespace std;
char prol[7][10] = {
  "S",
  "A",
  "A",
  "B",
  "B",
  "C",
  "C"
};
char pror[7][10] = {
  "A",
  "Bb",
  "Cd",
  "aB",
  "@",
  "Cc",
  "@"
};
char prod[7][10] = {
  "S->A",
  "A->Bb",
  "A->Cd",
  "B->aB",
  "B->@",
  "C->Cc",
  "C->@ "
};
char first[7][10] = {
```

```
  "abcd",
  "ab",
  "cd",
  "a@",
  "@",
  "c@",
  "@"
};
char follow[7][10] = {
  "$",
  "$",
  "$",
  "a$",
  "b$",
  "c$",
  "d$"
};
char table[5][6][10];
int numr(char c) {
  switch (c) {
  case 'S':
    return 0;
  case 'A':
    return 1;
  case 'B':
    return 2;
  case 'C':
    return 3;
  case 'a':
    return 0;
  case 'b':
    return 1;
  case 'c':
    return 2;
  case 'd':
    return 3;
  case '$':
    return 4;
  }
  return (2);
}

int main() {
  printf("Drish (RA2011030030026)\n\n");
  int i, j, k;
```

```
  for (i = 0; i < 5; i++)
    for (j = 0; j < 6; j++)
      strcpy(table[i][j], " ");
  printf("\nThe following is the predictive parsing table for the
following grammar: \n ");
  for (i = 0; i < 7; i++) {
    printf("%s\n", prod[i]);
  }
  printf("\nPredictive parsing table is\n");
  fflush(stdin);
  for (i = 0; i < 7; i++) {
    k = strlen(first[i]);
    for (j = 0; j < 10; j++)
      if (first[i][j] != '@')
        strcpy(table[numr(prol[i][0]) + 1][numr(first[i][j]) + 1],
prod[i]);
  }
  for (i = 0; i < 7; i++) {
    if (strlen(pror[i]) == 1) {
      if (pror[i][0] == '@') {
        k = strlen(follow[i]);
        for (j = 0; j < k; j++)
          strcpy(table[numr(prol[i][0]) + 1][numr(follow[i][j]) + 1],
prod[i]);
      }
    }
  }
  strcpy(table[0][0], " ");
  strcpy(table[0][1], "a");
  strcpy(table[0][2], "b");
  strcpy(table[0][3], "c");
  strcpy(table[0][4], "d");
  strcpy(table[0][5], "$");
  strcpy(table[1][0], "S");
  strcpy(table[2][0], "A");
  strcpy(table[3][0], "B");
  strcpy(table[4][0], "C");
  printf("\n---------------------------------------------------------\n");

for (i = 0; i < 5; i++)
    for (j = 0; j < 6; j++) {
      printf("%-10s", table[i][j]);
      if (j == 5)
```

```
printf("\n----------------------------------------------------------\n");
    }
  return 0;
}
```

Output:

```
Drish (RA2011030030026)


The following is the predictive parsing table for the following grammar:
S->A
A->Bb
A->Cd
B->aB
B->@
C->Cc
C->@

Predictive parsing table is

------------------------------------------------------------
          a         b         c         d         $
------------------------------------------------------------
S         S->A      S->A      S->A      S->A
------------------------------------------------------------
A         A->Bb     A->Bb     A->Cd     A->Cd
------------------------------------------------------------
B         B->aB     B->@      B->@                B->@
------------------------------------------------------------
C                             C->@      C->@      C->@
------------------------------------------------------------
```

Experiment 7

Aim: Shift Reduce Parsing

Theory: A shift-reduce parser is a bottom-up parsing algorithm that processes input by shifting symbols onto a stack and reducing them according to grammar rules. It maintains a stack and an input buffer. The parser performs shift operations when input matches the stack top and reduces when a stack sequence matches a production's right-hand side. A parsing table guides actions based on the stack symbol and input symbol. The process continues until the end of input or an error. Successful parsing occurs when the start symbol is reduced and the input is fully consumed. Shift-reduce parsing is efficient but may face conflicts in ambiguous grammars, which can be resolved using precedence rules or conflict resolution techniques.

Code:

```
#include<bits/stdc++.h>

using namespace std;
int k = 0, z = 0, i = 0, j = 0, c = 0;
char a[16], ac[20], stk[15], act[10];
void check();
int main() {
  printf("Drish (RA2011030030026)\n\n");
  printf("GRAMMAR is \n E->E+E \n E->E*E \n E->(E) \n E->id");
  printf("\n\nEnter input string ");
  scanf("%s", a);
  c = strlen(a);
  strcpy(act, "SHIFT->");
  printf("STACK \t INPUT \tCOMMENT");
  printf("$ \t%s$\n", a);
  for (k = 0, i = 0; j < c; k++, i++, j++) {
    if (a[j] == 'i' && a[j + 1] == 'd') {
      stk[i] = a[j];
      stk[i + 1] = a[j + 1];
      stk[i + 2] = '\0';
      a[j] = ' ';
      a[j + 1] = ' ';
      printf("\n$%s\t%s$\t%sid", stk, a, act);
      check();
    } else {
      stk[i] = a[j];
      stk[i + 1] = '\0';
      a[j] = ' ';
      printf("\n$%s\t%s$\t%ssymbols", stk, a, act);
```

```
      check();
    }
  }
}
void check() {
  strcpy(ac, "REDUCE TO E");
  for (z = 0; z < c; z++)
    if (stk[z] == 'i' && stk[z + 1] == 'd') {
      stk[z] = 'E';
      stk[z + 1] = '\0';
      printf("\n$%s\t%s$\t%s", stk, a, ac);
      j++;
    }
  for (z = 0; z < c; z++)
    if (stk[z] == 'E' && stk[z + 1] == '+' && stk[z + 2] == 'E') {
      stk[z] = 'E';
      stk[z + 1] = '\0';
      stk[z + 2] = '\0';
      printf("\n$%s\t%s$\t%s", stk, a, ac);
      i = i - 2;
    }
  for (z = 0; z < c; z++)
    if (stk[z] == 'E' && stk[z + 1] == '*' && stk[z + 2] == 'E') {
      stk[z] = 'E';
      stk[z + 1] = '\0';
      stk[z + 1] = '\0';
      printf("\n$%s\t%s$\t%s", stk, a, ac);
      i = i - 2;
    }
  for (z = 0; z < c; z++)
    if (stk[z] == '(' && stk[z + 1] == 'E' && stk[z + 2] == ')') {
      stk[z] = 'E';
      stk[z + 1] = '\0';
      stk[z + 1] = '\0';
      printf("\n$%s\t%s$\t%s", stk, a, ac);
      i = i - 2;
    }
}
```

Output:

```
Drish (RA2011030030026)
GRAMMAR is
 E->E+E
 E->E*E
 E->(E)
 E->id

Enter input string id+id*id
STACK       INPUT     COMMENT$   id+id*id$

$id     +id*id$  SHIFT->id
$E      +id*id$  REDUCE TO E
$E+      id*id$  SHIFT->symbols
$E+id      *id$  SHIFT->id
$E+E       *id$  REDUCE TO E
$E         *id$  REDUCE TO E
$E*         id$  SHIFT->symbols
$E*id         $  SHIFT->id
$E*E          $  REDUCE TO E
$E            $  REDUCE TO E
```

Experiment 8

Aim: Computation of LEADING and TRAILING

Theory: LEADING and TRAILING sets are used to determine the possible first and last terminals of strings derived from non-terminals in a context-free grammar. The LEADING set of a non-terminal consists of terminals that can be the first symbol in any derivation. It includes terminals in productions starting with terminals and non-terminals with computed LEADING sets. The TRAILING set represents terminals that can be the last symbol of any derivation. It includes terminals in productions ending with terminals and non-terminals with computed TRAILING sets. These sets are crucial for parsing, conflict resolution, and constructing parsing tables. By computing LEADING and TRAILING sets, we gain insights into valid lookahead symbols during parsing, enabling efficient analysis and handling of grammatical structures.

Code:

```
#include<bits/stdc++.h>
using namespace std;
int nt, t, top = 0;
char s[50], NT[10], T[10], st[50], l[10][10], tr[50][50];
int searchnt(char a) {
    int count = -1, i;
    for (i = 0; i < nt; i++) {
        if (NT[i] == a)
            return i;
    }
    return count;
}
int searchter(char a) {
    int count = -1, i;
    for (i = 0; i < t; i++) {
        if (T[i] == a)
            return i;
    }
    return count;
}
void push(char a) {
    s[top] = a;
    top++;
}
char pop() {
    top--;
    return s[top];
```

```
}

void installl(int a, int b) {
    if (l[a][b] == 'f') {
        l[a][b] = 't';
        push(T[b]);
        push(NT[a]);
    }
}
void installt(int a, int b) {
    if (tr[a][b] == 'f') {
        tr[a][b] = 't';
        push(T[b]);
        push(NT[a]);
    }
}
int main() {
    cout << "Drish (RA2011030030026)\n\n";
    int i, s, k, j, n;
    char pr[30][30], b, c;
    cout << "Enter the no of productions:\n";
    cin >> n;
    cout << "Enter the productions one by one\n";
    for (i = 0; i < n; i++)
        cin >> pr[i];
    nt = 0;
    t = 0;
    for (i = 0; i < n; i++) {
        if ((searchnt(pr[i][0])) == -1)
            NT[nt++] = pr[i][0];
    }
    for (i = 0; i < n; i++) {
        for (j = 3; j < strlen(pr[i]); j++) {
            if (searchnt(pr[i][j]) == -1) {
                if (searchter(pr[i][j]) == -1)
                    T[t++] = pr[i][j];
            }
        }
    }
    for (i = 0; i < nt; i++) {
        for (j = 0; j < t; j++)
            l[i][j] = 'f';
    }
    for (i = 0; i < nt; i++) {
        for (j = 0; j < t; j++)
```

```
            tr[i][j] = 'f';
    }
    for (i = 0; i < nt; i++) {
        for (j = 0; j < n; j++) {
            if (NT[(searchnt(pr[j][0]))] == NT[i]) {
                if (searchter(pr[j][3]) != -1)
                    installl(searchnt(pr[j][0]), searchter(pr[j][3]));
                else {
                    for (k = 3; k < strlen(pr[j]); k++) {
                        if (searchnt(pr[j][k]) == -1) {
                            installl(searchnt(pr[j][0]),
searchter(pr[j][k]));
                            break;
                        }
                    }
                }
            }
        }
    }
    while (top != 0) {
        b = pop();
        c = pop();
        for (s = 0; s < n; s++) {
            if (pr[s][3] == b)
                installl(searchnt(pr[s][0]), searchter(c));
        }
    }
    for (i = 0; i < nt; i++) {
        cout << "Leading[" << NT[i] << "]" << "\t{";
        for (j = 0; j < t; j++) {
            if (l[i][j] == 't')
                cout << T[j] << ",";
        }
        cout << "}\n";
    }
    top = 0;
    for (i = 0; i < nt; i++) {
        for (j = 0; j < n; j++) {
            if (NT[searchnt(pr[j][0])] == NT[i]) {
                if (searchter(pr[j][strlen(pr[j]) - 1]) != -1)
                    installt(searchnt(pr[j][0]),
searchter(pr[j][strlen(pr[j]) - 1]));
                else {
                    for (k = (strlen(pr[j]) - 1); k >= 3; k--) {
                        if (searchnt(pr[j][k]) == -1) {
```

```
                            installt(searchnt(pr[j][0]),
searchter(pr[j][k]));
                            break;
                        }
                    }
                }
            }
        }
    }
    while (top != 0) {
        b = pop();
        c = pop();
        for (s = 0; s < n; s++) {
            if (pr[s][3] == b)
                installt(searchnt(pr[s][0]), searchter(c));
        }
    }
    for (i = 0; i < nt; i++) {
        cout << "Trailing[" << NT[i] << "]" << "\t{";
        for (j = 0; j < t; j++) {
            if (tr[i][j] == 't')
                cout << T[j] << ",";
        }
        cout << "}\n";
    }
}
```

Output:

```
Drish (RA2011030030026)

Enter the no of productions: 6
Enter the productions one by one :
E->E+T
E->T
T->T*F
T->F
F->(E)
F->i
Leading[E] {+,*,(,i,}
Leading[T] {*,(,i,}
Leading[F] {(,i,}
Trailing[E]     {+,*,),i,}
Trailing[T]     {*,),i,}
Trailing[F]     {),i,}
```

Experiment 9

Aim: Computation of LR(0) Items

Theory: Computation of LR(0) items involves constructing LR(0) item sets that represent states in a deterministic finite automaton (DFA) for bottom-up parsing. LR(0) items are productions augmented with a dot to indicate the current parsing position. The process includes the closure operation, which recursively adds items based on grammar rules, and the goto operation, which determines transitions between item sets. Item sets are built through closure and goto operations until no new sets can be generated. The resulting item sets form the basis for constructing the LR(0) parsing table. This table determines shift, reduce, and accept actions based on the current state and input symbol. Computation of LR(0) items and the resulting item sets is essential for constructing efficient LR(0) parsers that can recognize viable prefixes and parse context-free grammars.

Code:

```
#include<bits/stdc++.h>

using namespace std;

char prod[20][20], listofvar[26] = "ABCDEFGHIJKLMNOPQR";
int novar = 1, i = 0, j = 0, k = 0, n = 0, m = 0, arr[30];
int noitem = 0;

struct Grammar {
  char lhs;
  char rhs[8];
}
g[20], item[20], clos[20][10];

int isvariable(char variable) {
  for (int i = 0; i < novar; i++)
    if (g[i].lhs == variable)
      return i + 1;
  return 0;
}
void findclosure(int z, char a) {
  int n = 0, i = 0, j = 0, k = 0, l = 0;
  for (i = 0; i < arr[z]; i++) {
    for (j = 0; j < strlen(clos[z][i].rhs); j++) {
      if (clos[z][i].rhs[j] == '.' && clos[z][i].rhs[j + 1] == a) {
        clos[noitem][n].lhs = clos[z][i].lhs;
        strcpy(clos[noitem][n].rhs, clos[z][i].rhs);
```

```
        char temp = clos[noitem][n].rhs[j];
        clos[noitem][n].rhs[j] = clos[noitem][n].rhs[j + 1];
        clos[noitem][n].rhs[j + 1] = temp;
        n = n + 1;
      }
    }
  }
  for (i = 0; i < n; i++) {
    for (j = 0; j < strlen(clos[noitem][i].rhs); j++) {
      if (clos[noitem][i].rhs[j] == '.' &&
isvariable(clos[noitem][i].rhs[j + 1]) > 0) {
        for (k = 0; k < novar; k++) {
          if (clos[noitem][i].rhs[j + 1] == clos[0][k].lhs) {
            for (l = 0; l < n; l++)
              if (clos[noitem][l].lhs == clos[0][k].lhs &&
strcmp(clos[noitem][l].rhs, clos[0][k].rhs) == 0)
                break;
            if (l == n) {
              clos[noitem][n].lhs = clos[0][k].lhs;
              strcpy(clos[noitem][n].rhs, clos[0][k].rhs);
              n = n + 1;
            }
          }
        }
      }
    }
  }
  arr[noitem] = n;
  int flag = 0;
  for (i = 0; i < noitem; i++) {
    if (arr[i] == n) {
      for (j = 0; j < arr[i]; j++) {
        int c = 0;
        for (k = 0; k < arr[i]; k++)
          if (clos[noitem][k].lhs == clos[i][k].lhs &&
strcmp(clos[noitem][k].rhs, clos[i][k].rhs) == 0)
            c = c + 1;
        if (c == arr[i]) {
          flag = 1;
          goto exit;
        }
      }
    }
  }
  exit: ;
```

```
  if (flag == 0)
    arr[noitem++] = n;
}

int main() {
  cout << "Drish (RA2011030030026)\n\n";
  cout << "ENTER THE PRODUCTIONS OF THE GRAMMAR(0 TO END) :\n";
  do {
    cin >> prod[i++];
  } while (strcmp(prod[i - 1], "0") != 0);
  for (n = 0; n < i - 1; n++) {
    m = 0;
    j = novar;
    g[novar++].lhs = prod[n][0];
    for (k = 3; k < strlen(prod[n]); k++) {
      if (prod[n][k] != '|')
        g[j].rhs[m++] = prod[n][k];
      if (prod[n][k] == '|') {
        g[j].rhs[m] = '\0';
        m = 0;
        j = novar;
        g[novar++].lhs = prod[n][0];
      }
    }
  }
  for (i = 0; i < 26; i++)
    if (!isvariable(listofvar[i]))
      break;
  g[0].lhs = listofvar[i];
  char temp[2] = {
    g[1].lhs,
    '\0'
  };
  strcat(g[0].rhs, temp);
  cout << "\n\n Augumented grammar \n";
  for (i = 0; i < novar; i++)
    cout << endl << g[i].lhs << "->" << g[i].rhs << " ";

  for (i = 0; i < novar; i++) {
    clos[noitem][i].lhs = g[i].lhs;
    strcpy(clos[noitem][i].rhs, g[i].rhs);
    if (strcmp(clos[noitem][i].rhs, "ε") == 0)
      strcpy(clos[noitem][i].rhs, ".");
    else {
      for (int j = strlen(clos[noitem][i].rhs) + 1; j >= 0; j--)
```

```
        clos[noitem][i].rhs[j] = clos[noitem][i].rhs[j - 1];
      clos[noitem][i].rhs[0] = '.';
    }
  }
  arr[noitem++] = novar;
  for (int z = 0; z < noitem; z++) {
    char list[10];
    int l = 0;
    for (j = 0; j < arr[z]; j++) {
      for (k = 0; k < strlen(clos[z][j].rhs) - 1; k++) {
        if (clos[z][j].rhs[k] == '.') {
          for (m = 0; m < l; m++)
            if (list[m] == clos[z][j].rhs[k + 1])
              break;
          if (m == l)
            list[l++] = clos[z][j].rhs[k + 1];
        }
      }
    }
    for (int x = 0; x < l; x++)
      findclosure(z, list[x]);
  }
  cout << "\n THE SET OF ITEMS ARE \n\n";
  for (int z = 0; z < noitem; z++) {
    cout << "\n I" << z << "\n\n";
    for (j = 0; j < arr[z]; j++)
      cout << clos[z][j].lhs << "->" << clos[z][j].rhs << "\n";

  }

}
```

Output:

```
Drish (RA2011030030026)

ENTER THE PRODUCTIONS OF THE GRAMMAR(0 TO END) :
E->E+T
E->T
T->T*F
T->F
F->(E)
F->i
0
```

```
Augumented grammar

A->E
E->E+T
E->T
T->T*F
T->F
F->(E)
-->
 THE SET OF ITEMS ARE

 I0

A->.E
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
-->.

 I1

A->E.
E->E.+T

 I2

E->T.
T->T.*F

 I3

T->F.

 I4

F->(.E)
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)

 I5
```

```
E->E+.T
T->.T*F
T->.F
F->.(E)

 I6

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

 I7

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

 I8

E->E+T.
T->T.*F

 I9

T->T*F.

 I10

F->(E).
```

Experiment 10

Aim: Intermediate Code Generation: Postfix, Prefix

Theory: Intermediate code generation transforms source code into a lower-level representation before code generation and optimization. Postfix notation, also known as Reverse Polish Notation (RPN), represents expressions with operators appearing after operands. It eliminates the need for parentheses and explicitly indicates the order of operations. Prefix notation, also called Polish Notation, represents expressions with operators preceding operands. It allows unambiguous parsing without parentheses or precedence rules. Both notations facilitate stack-based evaluation and efficient code generation. The choice between postfix and prefix notation depends on the specific compiler or interpreter. Intermediate code generation in postfix or prefix form provides a compact, machine-independent representation that simplifies subsequent optimization and translation into target code.

Code:

Infix to PostFix:

```
#include<bits/stdc++.h>
#define SIZE 100

char stack[SIZE];
int top = -1;

void push(char item) {
  if (top >= SIZE - 1) {
    printf("\nStack Overflow.");
  } else {
    top = top + 1;
    stack[top] = item;
  }
}
char pop() {
  char item;
  if (top < 0) {
    printf("stack under flow: invalid infix expression");
    getchar();
    exit(1);
  } else {
    item = stack[top];
    top = top - 1;
    return (item);
  }
}
```

```
int is_operator(char symbol) {
  if (symbol == '^' || symbol == '*' || symbol == '/' || symbol == '+' ||
symbol == '-') {
    return 1;
  } else {
    return 0;
  }
}
int precedence(char symbol) {
  if (symbol == '^') {
    return (3);
  } else if (symbol == '*' || symbol == '/') {
    return (2);
  } else if (symbol == '+' || symbol == '-') {
    return (1);
  } else {
    return (0);
  }
}

void InfixToPostfix(char infix_exp[], char postfix_exp[]) {
  int i, j;
  char item;
  char x;
  push('(');
  strcat(infix_exp, ")");
  i = 0;
  j = 0;
  item = infix_exp[i];

  while (item != '\0') {
    if (item == '(') {
      push(item);
    } else if (isdigit(item) || isalpha(item)) {
      postfix_exp[j] = item;
      j++;
    } else if (is_operator(item) == 1) {
      x = pop();
      while (is_operator(x) == 1 && precedence(x) >= precedence(item)) {
        postfix_exp[j] = x;
        j++;
        x = pop();
      }
      push(x);
```

```
      push(item);
    } else if (item == ')') {
      x = pop();
      while (x != '(') {
        postfix_exp[j] = x;
        j++;
        x = pop();
      }
    } else {
      printf("\nInvalid infix Expression.\n");
      getchar();
      exit(1);
    }
    i++;

    item = infix_exp[i];
  }
  if (top > 0) {
    printf("\nInvalid infix Expression.\n");
    getchar();
    exit(1);
  }
  if (top > 0) {
    printf("\nInvalid infix Expression.\n");
    getchar();
    exit(1);
  }
  postfix_exp[j] = '\0';
}
int main() {
  printf("Drish (RA2011030030026)\n\n");
  char infix[SIZE], postfix[SIZE];
  printf("ASSUMPTION: The infix expression contains single letter
variables and single digit constants only.\n");
  printf("\nEnter Infix expression : ");
  scanf("%s", infix);

  InfixToPostfix(infix, postfix);
  printf("Postfix Expression: ");
  puts(postfix);

  return 0;
}
```

PostFix to Prefix:

```
#include<bits/stdc++.h>
#define MAX 20
char str[MAX], stack[MAX];
int top = -1;

void push(char c) {
  stack[++top] = c;
}

char pop() {
  return stack[top--];
}

int checkIfOperand(char ch) {
  return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

int isOperator(char x) {
  switch (x) {
  case '+':
  case '-':
  case '/':
  case '*':
    return 1;
  }
  return 0;
}

void postfixToprfix() {
  int n, i, j = 0;
  char c[20];
  char a, b, op;
  printf("Enter the postfix expression: ");
  scanf("%s", str);
  n = strlen(str);
  for (i = 0; i < MAX; i++)
    stack[i] = '\0';
  printf("Prefix expression is:\t");

  for (i = n - 1; i >= 0; i--) {
    if (isOperator(str[i])) {
      push(str[i]);
    } else {
      c[j++] = str[i];
```

```
      while ((top != -1) && (stack[top] == '#')) {
        a = pop();
        c[j++] = pop();
      }
      push('#');
    }
  }
  c[j] = '\0';
  i = 0;
  j = strlen(c) - 1;
  char d[20];

  while (c[i] != '\0') {
    d[j--] = c[i++];
  }
  printf("%s\n", d);
}
int main() {
  printf("Drish (RA2011030030026)\n\n");
  postfixToprfix();
  return 0;
}
```

Output:

Infix to PostFix

```
Drish (RA2011030030026)

ASSUMPTION: The infix expression contains single letter variables and
single digit constants only.

Enter Infix expression : A+B^C/D
Postfix Expression: ABC^D/+
```

PostFix to Prefix

```
Drish (RA2011030030026)

Enter the postfix expression: ABC^D/+
Prefix expression is: B+C/^D
```

Experiment 11

Aim: Intermediate Code Generation: Quadruple, Triple, Indirect Triple

Theory: Intermediate code generation involves transforming source code into a lower-level representation before further optimization and code generation. Quadruples, triples, and indirect triples are common forms of intermediate code used in this process. Quadruples use four fields to represent an operation, including the operator, two operands, and a result. Triples simplify this representation by omitting the result field, while indirect triples add a pointer to the next operation. These intermediate code forms provide compact and efficient representations, capturing essential information for evaluation and facilitating optimization. They serve as an intermediate step between the source code and target code, enabling easier analysis and translation. The choice of intermediate code representation depends on the specific requirements and goals of the compiler or interpreter, balancing factors such as code size, efficiency, and flexibility.

Code:

```
#include<bits/stdc++.h>

int main() {
  printf("Drish (RA2011030030026)\n\n");
  char line[20];
  int s[20];
  int t = 1;
  int i = 0;
  printf("Enter String : ");
  scanf("%s", line);
  for (i = 0; i < 20; i++)
    s[i] = 0;
  printf("op\ta1\ta2\tres\n");
  for (i = 2; line[i] != '\0'; i++) {
    if (line[i] == '/' || line[i] == '*') {
      printf("\n");
      if (s[i] == 0) {
        if (s[i + 1] == 0) {
          printf(":=\t%c\t\t t%d\n", line[i + 1], t);
          s[i + 1] = t++;
        }
        printf("%c\t", line[i]);
        (s[i - 1] == 0) ? printf("%c\t", line[i - 1]): printf("t%d\t",
s[i - 1]);
        printf("t%d \t t%d", s[i + 1], t);
        s[i - 1] = s[i + 1] = t++;
        s[i] = 1;
```

```
      }
    }
  }
  for (i = 2; line[i] != '\0'; i++) {

    if (line[i] == '+' || line[i] == '-') {
      printf("\n");
      if (s[i] == 0) {
        if (s[i + 1] == 0) {
          printf(":=\t%c\t\t t%d\n", line[i + 1]);
          s[i + 1] = t++;
        }
        printf("%c\t", line[i]);
        (s[i - 1] == 0) ? printf("%c\t", line[i - 1]): printf("t%d\t",
s[i - 1]);
        printf("t%d \t t%d", s[i + 1], t);
        s[i - 1] = s[i + 1] = t++;
        s[i] = 1;
      }
    }
  }
  printf("\n:=\tt%d\t\t%c", t - 1, line[0]);
  return 0;
}
```

Output:

```
Drish (RA2011030030026)

Enter String : X=a*b+b*c
op    a1    a2    res

:=    b           t1
*     a     t1    t2
:=    c           t3
*     b     t3    t4
+     t2    t4    t5
:=    t5         X
```

Experiment 12

Aim: A Simple Code Generator

Theory: A simple code generator is a component of a compiler or interpreter that translates intermediate code or abstract syntax trees into executable machine code or bytecode. It takes an intermediate representation, such as quadruples or triples, and maps operations to target machine instructions. The code generator considers the target machine's architecture, performs register allocation, and generates code by organizing instructions and incorporating control flow constructs. While a simple code generator may not perform extensive optimizations, it can still exploit basic optimization opportunities. It serves as a bridge between the intermediate representation and executable code, facilitating the translation of high-level language constructs into machine-specific instructions.

Code:

```
#include <iostream>
#include <vector>
#include <string>

struct TACInstruction {
  std::string result;
  std::string operand1;
  std::string operand2;
  std::string operation;
};

void generateAssembly(const std::vector < TACInstruction > &
tacInstructions) {
  std::string resultRegister = "";
  for (const TACInstruction & instruction: tacInstructions) {
    if (instruction.operation == "+") {
      if (resultRegister.empty()) {
        resultRegister = instruction.result;
      } else {
        std::cout << "add " << resultRegister << ", " <<
instruction.operand1 << std::endl;
      }
      std::cout << "add " << resultRegister << ", " <<
instruction.operand2 << std::endl;
    } else if (instruction.operation == "-") {
      if (resultRegister.empty()) {
        resultRegister = instruction.result;
      } else {
```

```
        std::cout << "sub " << resultRegister << ", " <<
instruction.operand1 << std::endl;
      }
      std::cout << "sub " << resultRegister << ", " <<
instruction.operand2 << std::endl;
    } else if (instruction.operation == "*") {
      if (resultRegister.empty()) {
        resultRegister = instruction.result;
      } else {
        std::cout << "imul " << resultRegister << ", " <<
instruction.operand1 << std::endl;
      }
      std::cout << "imul " << resultRegister << ", " <<
instruction.operand2 << std::endl;
    } else {
      std::cout << "; Unsupported operation: " << instruction.operation
<< std::endl;
    }
  }
  std::cout << "mov " << resultRegister << ", " <<
tacInstructions.back().result << std::endl;
}

int main() {
  std::cout << "Drish (RA2011030030026)\n\n";
  std::vector < TACInstruction > tacInstructions = {
    {
      "t1",
      "a",
      "b",
      "+"
    },
    {
      "t2",
      "t1",
      "c",
      "*"
    },
    {
      "result",
      "t2",
      "d",
      "-"
    }
  };
```

```
  generateAssembly(tacInstructions);
  return 0;
}
```

Output:

```
Drish (RA2011030030026)
add t1, b
imul t1, t1
imul t1, c
sub t1, t2
sub t1, d
mov t1, result
```

Experiment 13

Aim: Implementation of DAG

Theory: A Directed Acyclic Graph (DAG) is a graph without cycles used in compilers to optimize code generation. It represents expressions and their dependencies. DAGs are constructed from intermediate representations like three-address code by identifying common subexpressions and sharing their nodes. They enable efficient analysis and optimizations like constant folding, copy propagation, and common subexpression elimination. During code generation, the DAG is traversed to generate optimized assembly code, minimizing redundant computations by reusing values stored in the DAG nodes. DAGs provide a compact representation of expressions, enabling efficient analysis and optimization algorithms.

Code:

```
#include<bits/stdc++.h>
using namespace std;
int main() {
  string exp;
  cout << "Drish (RA2011030030026)\n\n";
  cout << "Enter the expression: ";
  cin >> exp;
  int j = 0, k = 0;
  char q;
  for (int i = exp.length() - 1; i > 1; i--) {
    if (islower(exp[i]) || (exp[i] >= 48 && exp[i] <= 57)) {
      cout << j << "->" << exp[i] << endl;
      j++;
    }
  }
  for (int i = exp.length() - 1; i > 1; i--) {
    if (!(islower(exp[i]) || (exp[i] >= 48 && exp[i] <= 57))) {
      cout << j << "->" << exp[i] << k << k + 1 << endl;
      j++;
      k += 2;
    }
  }
  cout << j << "->" << exp[0] << endl;
  j++;
  cout << j << "->" << exp[1] << j - 1 << j - 2 << endl;
  return 0;
}
```

Output:

```
Drish (RA2011030030026)

Enter the expression: a=b+c-5
0->5
1->c
2->b
3->-01
4->+23
5->a
6->=54
```

Experiment 14

Aim: Recursive Descent Parsing

Theory: Recursive Descent Parsing is a top-down parsing technique used for syntax analysis in programming languages. It builds a parse tree by recursively applying production rules. Each non-terminal symbol has a corresponding parsing function. The parser matches input tokens with expected tokens from the grammar, invoking the relevant parsing function. Recursive Descent Parsing is simple and easy to implement, reflecting the grammar structure. It handles LL(k) grammars, with k representing the lookahead tokens. However, it has limitations, including difficulty handling left-recursive grammars and potential backtracking in ambiguous or conflicting situations. To address these issues, preprocessing techniques such as left-factoring and left-recursion elimination can be applied to the grammar. Recursive Descent Parsing is a widely used and effective parsing technique in many programming language compilers and parsers.

Code:

```
#include<bits/stdc++.h>

using namespace std;
int main() {
  cout << "Drish (RA2011030030026)\n\n";
  int flag = 0;
  map < char, vector < string > > rules;
  string exp, test;
  rules['S'].push_back("aAc");
  rules['A'].push_back("cd");
  rules['A'].push_back("d");
  cout << "Enter the string: ";
  cin >> exp;
  string start = "aAc";
  if (start[0] != exp[0])
    cout << "Not Accepted";
  else {
    cout << "S" << endl << start << endl;
    string a = (rules['A'])[0];
    string b = (rules['A'])[1];
    string t;
    t = start[0] + a + start[2];
    cout << t << endl;
    if (t == exp) {
      flag = 1;
      cout << "Accepted";
```

```
    } else {
      cout << start << endl;
      t = start[0] + b + start[2];
      cout << t << endl;
      if (t == exp) {
        flag = 1;
        cout << "Accepted";
      }
    }
  }
  if (flag == 0) cout << "Not accepted";
  return 0;
}
```

Output:

```
Drish (RA2011030030026)

Enter the string: adc
S
aAc
acdc
aAc
adc
Accepted
```