Name: Shahbaz Jahan Section: CSE-1 Roll No.: 160120733050

EXPERIMENT NO.: 01

AIM

Program to count number of characters, words and lines from a file.

DESCRIPTION

Given a file, i.e., a text file. Count the total number of characters, words and lines present in that file.

ALGORITHM

Step 1: Define c = 0, w = 0, and l = 1
Step 2: Read given file line by line.
Step 3: for each character in line; do
c = c + 1
If character is space; then
w = w + 1
Step 4: l = l + 1
Step 5: Repeat steps 2 to 4, until EOF (End Of File) is reached.
Step 6: Write c, w and l to the output
Step 7: End

PROGRAM/CODE

```
FILE_PATH = "./text.txt"
def main() -> None:
   c, l, w = 0, 0, 0
   with open(FILE_PATH, "r") as f:
      for line in f.readlines():
         c += len(line)
         w += len(line.split())
         l += 1
   print(f"Number of char: {c}, line: {l} and word: {w}")
if __name__ == "__main__":
   print("160120733301")
   exit(main() or 0)
```

OUTPUT

```
● ~/.../lab/cd-lab1-0117 ↳  python finfo.py
  160120733301
  Number of char: 2464, line: 9 and word: 360
```

CONCLUSION

Hence, Python script to obtain number of characters, words and lines has been implemented successfully.

EXPERIMENT NO.: 02

AIM

Program to identify different tokens and token types from a file.

DESCRIPTION

Given a file, i.e., a text file. Identify the tokens and there types(i.e., identifier, keyword, etc) present in that file.

ALGORITHM

Step 1: Read the given file character by character.

Step 2: Capture the lexeme into buffer, which are separated deliminators.

Step 3: Calculate the line number for each lexeme.

Step 4: Classify each lexeme as token(i.e., String, Constant, etc).

Step 5: Write the discovered tokens to output.

Step 6: Repeat steps 2 to 5, until EOF (End Of File) is reached.

Step 7: Close the file and End the problem.

PROGRAM/CODE

```
from enum import Enum

FILE_PATH = "./test.txt"
class TokType(Enum):
   String = 1
   Constant = 2
   Special = 3

def tokens(data: list):
   toks = []
   cl = 0
   for line in data:
      cl += 1
      for tok in line.split():
         try:
            int(tok)
            toks.append((cl, tok, TokType.Constant))
         except ValueError:
            if tok.isalpha():
               toks.append((cl, tok, TokType.String))
            else:
               toks.append((cl, tok, TokType.Special))
   return toks
```

```python
def main() -> None:
    data = []
    with open(FILE_PATH, "r") as f:
        data = f.readlines()
    toks = tokens(data)
    print(f"Total token(s): {len(toks)}")
    print("Tokens:-")
    for tok in toks:
        print(f"line: {tok[0]} \"{tok[1]}\"\tis\t{tok[2]}")

if __name__ == "__main__":
    print("160120733301")
    exit(main() or 0)
```

OUTPUT

```
~/.../lab/cd-lab1-0117 ᛋ  python toks.py
160120733301
Total token(s): 9
Tokens:-
line: 1 "this"  is      TokType.String
line: 1 "is"    is      TokType.String
line: 1 "1"     is      TokType.Constant
line: 1 "st"    is      TokType.String
line: 1 "CD"    is      TokType.String
line: 1 "lab"   is      TokType.String
line: 1 "."     is      TokType.Special
line: 2 "line"  is      TokType.String
line: 2 "one"   is      TokType.String
```

CONCLUSION

Hence, Python script to obtain tokens and token types from a file has been implemented successfully.

EXPERIMENT NO.: 03

AIM

Program to implement lexical analyzer without using lex tool.

DESCRIPTION

Given a file, i.e., a text file. Implement Lexical analyzer without using a lex tool present in that file. Lexical analysis, lexing or tokenization is the process of converting a sequence of characters into a sequence of lexical tokens. A program that performs lexical analysis may be termed a lexer, tokenizer, or scanner, although scanner is also a term for the first stage of a lexer.

ALGORITHM

Step 1: Read the given file character by character.

Step 2: Capture the lexeme into buffer, which are separated deliminators.

Step 3: Calculate the line number and column number for each lexeme.

Step 4: Classify each lexeme as token(i.e., Keyword, Identifier, Literal, etc).

Step 5: Write the discovered tokens to output.

Step 6: Repeat steps 2 to 5, until EOF (End Of File) is reached.

Step 7: Close the file and End the problem.

PROGRAM/CODE

```
from enum import Enum

FILE_PATH = "./hello.c"

K_WORDS = ["int", "float", "char", "return", "double", "break", "continue", "if", "else", "for", "while",
"do", "include"]

class TkType(Enum):
    Ident = 0
    Literal = 1
    Symbol = 2
    Const = 3
    Keyword = 4

def toks(data: str):
    a = data
    tks = []
    buf = []
    n = len(data)
    i = 0
    l, c = 1, 1
```

```
    while i < n:
        if a[i] == ' ' or a[i] == '\t':
            c += 1
            i += 1
            continue
        if a[i] == '\n':
            l += 1
            c = 1
            i += 1
            continue
        buf.clear()
        if a[i].isalpha():
            while i < n and a[i].isalpha():
                buf.append(a[i])
                i += 1
                c += 1
            i -= 1
            if "".join(buf) in K_WORDS:
                tks.append(((l, c - len(buf)), "".join(buf), TkType.Keyword))
            else:
                tks.append(((l, c - len(buf)), "".join(buf), TkType.Ident))
        elif a[i].isdigit():
            while i < n and a[i].isdigit():
                buf.append(a[i])
                i += 1
                c += 1
            i -= 1
            tks.append(((l, c - len(buf)), "".join(buf), TkType.Const))
        elif a[i] == "'":
            i += 1
            while i < n and a[i] != "'":
                buf.append(a[i])
                i += 1
                c += 1
            tks.append(((l, c - len(buf)), "'" + "".join(buf) + "'", TkType.Literal))
        else:
            tks.append(((l, c), a[i], TkType.Symbol))
        i += 1
        c += 1
    return tks
```

```python
def main() -> None:
    data = ""
    with open(FILE_PATH, "r") as f:
        data = f.read()
    tks = toks(data)
    print(f"TokenType\tline:col:Token")
    for tk in tks:
        print(f"{tk[2]}\t{tk[0][0]}:{tk[0][1]}:{tk[1]}")

if __name__ == "__main__":
    print("160120733301")
    exit(main() or 0)
```

OUTPUT

```
~/.../cd-lab1-0117/exp ᛋ  python toks.py
160120733301
TokenType       line:col:Token
TkType.Symbol   1:1:#
TkType.Keyword  1:2:include
TkType.Symbol   1:10:<
TkType.Ident    1:11:stdio
TkType.Symbol   1:17:.
TkType.Ident    1:18:h
TkType.Symbol   1:20:>
TkType.Keyword  3:1:int
TkType.Ident    3:6:main
TkType.Symbol   3:11:(
TkType.Symbol   3:12:)
TkType.Symbol   3:14:{
TkType.Ident    4:3:printf
TkType.Symbol   4:10:(
TkType.Literal  4:11:"Hello, World!"
TkType.Symbol   4:25:)
TkType.Symbol   4:26:;
TkType.Keyword  5:3:return
TkType.Const    5:11:0
TkType.Symbol   5:13:;
TkType.Symbol   6:1:}
```

CONCLUSION

Hence, Program to implement lexical analyzer with a lex tool has been implemented successfully.

EXPERIMENT NO.: 04

AIM

Write a lex program to display the count of vowels and consonants in a word.

DESCRIPTION

Given a file, i.e., a text file. Count the total number of vowels and consonants present in that word or line. Ignore all the special character and white spaces.

ALGORITHM

Step 1: Read the given file character by character.

Step 2: Define vowel = 0, consonant = 0

Step 3: Apply Regular expressions for identify Vowels or Consonants.

Step 4: If Step 2 found a match for Vowels; then
vowel = vowel + 1
Else if Step 2 found a match for Real; then
consonant = consonant + 1

Step 5: Write vowel and consonant to the output

Step 6: End

PROGRAM/CODE

```
%{
   int vowel = 0, cons = 0;
%}

%%
[aeiouAEIOU] { vowel++; }
[A-z] { cons++; }
\n { return 0; }
. {}
%%

int yywrap() {}

int main() {
   printf("160120733301\n");
   yylex();
   printf("Count of Vowels: %d\n", vowel);
   printf("Count of Consonants: %d\n", cons);
   return 0;
}
```

OUTPUT

```
~/.../lab/cd-lab2401 ᛋ  flex vandc.l
~/.../lab/cd-lab2401 ᛋ  cc -lfl lex.yy.c
~/.../lab/cd-lab2401 ᛋ  ./a.out
160120733301
Hello, World!
Count of Vowels: 3
Count of Consonants: 7
```

CONCLUSION

Hence, lex program to count vowels and consonants has been implemented successfully.

EXPERIMENT NO.: 05

AIM

Write a lex program to find given number is integer or a real number.

DESCRIPTION

Given a file, i.e., a text file. Find Integer or Real number present in that word or line. Ignore all the special character and white spaces.

ALGORITHM

Step 1: Read the given file character by character.

Step 2: Apply Regular expressions to identify Integer or Real Number.

Step 3: If Step 2 found a match for Integer; then
Write 'Given String is Integer'
Else if Step 2 found a match for Real; then
Write 'Given String is Real Number'
Else
Write 'Invalid String'
Step 4: End

PROGRAM/CODE

```
%{
  #include<stdio.h>
%}
%%
[0-9]*(\.|\.e|e)[0-9]+ { printf("\"%s\" is a Real Number\n", yytext); }
[0-9]+ { printf("\"%s\" is an Integer\n", yytext); }
\n { return 0; }
. {}
%%

int yywrap() {}

int main() {
  printf("160120733301\n");
  yylex();
  return 0;
}
```

OUTPUT

```
~/.../lab/cd-lab2401 ᛋ  flex intorreal.l
~/.../lab/cd-lab2401 ᛋ  cc -lfl lex.yy.c
~/.../lab/cd-lab2401 ᛋ  ./a.out
160120733301
301.2
"301.2" is a Real Number
~/.../lab/cd-lab2401 ᛋ  ./a.out
160120733301
301
"301" is an Integer
```

CONCLUSION

Hence, lex program to find given number is integer or a real number has been implemented successfully.

EXPERIMENT NO.: 06

AIM

Write a lex program to capitalize the characters.

DESCRIPTION

Given a file, i.e., a text file. To capitalize characters present in that word or line.

ALGORITHM

Step 1: Read the given file character by character.

Step 2: Apply Regular expressions to identify Lowercase letters.

Step 3: if the current encoding format is UTF-8, then subtract 32 from the character to obtain the lower case character. else use the mapping take to obtain the lower case character.

Step 4: End

PROGRAM/CODE

```
%{
  #include<stdio.h>
%}
%%
[a-z] { printf("%c", yytext[0] - 32); }
\n { printf("\n"); return 0; }
%%

int yywrap() {}

int main() {
  printf("160120733301\n");
  yylex();
  return 0;
}
```

OUTPUT

```
~/.../lab/cd-lab2401 ↳  flex cap.l
~/.../lab/cd-lab2401 ↳  cc -lfl lex.yy.c
~/.../lab/cd-lab2401 ↳  ./a.out
160120733301
Hello, World!
HELLO, WORLD!
```

CONCLUSION

Hence, lex program to capitalize characters has been implemented successfully.

EXPERIMENT NO.: 07

AIM

Write a lex program to accept words starting with a or A letter.

DESCRIPTION

Given a file, i.e., a text file. To accept words starting with a or A present in that word or line. In Reg Ex a word is generally defined as \b\w, where \b denotes the white space and \w denotes a word.

ALGORITHM

Step 1: Read the given file character by character.

Step 2: Apply Regular expressions to identify a | A is present in the start of a word.

Step 3: If Step 2 found a match; then
Write 'Given String is Accepted'
Else
Write 'Given String is not Accepted'

Step 4: End

PROGRAM/CODE

```
%{
  #include<stdio.h>
  int c = 0;
%}
%%
^[aA][a-zA-Z]+ { c++; }
\n { return 0; }
. {}
%%
int yywrap() {}
int main() {
  printf("160120733301\n");
  yylex();
  if (c > 0) {
    printf("Accepted!\n");
  } else {
    printf("Not Accepted!\n");
  }
  return 0;
}
```

OUTPUT

```
● ~/.../lab/cd-lab0702 ᛋ  flex starts.l
● ~/.../lab/cd-lab0702 ᛋ  cc -lfl lex.yy.c
● ~/.../lab/cd-lab0702 ᛋ  ./a.out
  160120733301
  apple
  Accepted!
● ~/.../lab/cd-lab0702 ᛋ  ./a.out
  160120733301
  banana
  Not Accepted!
```

CONCLUSION

Hence, lex program to accept word starting with a or A has been implemented successfully.

EXPERIMENT NO.: 08

AIM

Write a lex program to design token separator.

DESCRIPTION

Given a file, i.e., a text file. To separate tokens present in word or line(s). A separator is one or two tokens that separate some language features from other language features.

ALGORITHM

Step 1: Read the given file character by character.

Step 2: Apply Regular expressions for the different set tokens.

Step 3: Write the discovered tokens to output.

Step 4: Repeat steps 2 and 3, until EOF (End Of File) is reached.

Step 5: Close the file and End the problem.

PROGRAM/CODE

```
%{
        #include<stdio.h>
        int l = 1;
%}
delim [ \t\b]
ws {delim}*
ident [A-Za-z][A-Za-z0-9]*
op [\+\-\*/%=]
special [;\{\}\[\]\(\)<>]
%%
{ws}(int|return|include) { printf("%d\t\"%s\"\t\t\tKeyword\n", l, yytext); }
{ident} { printf("%d\t\"%s\"\t\t\tIdentifier\n", l, yytext); }
{op} { printf("%d\t\"%s\"\t\t\tOperator\n", l, yytext); }
{special} { printf("%d\t\"%s\"\t\t\tSpecial\n", l, yytext); }
\n { l++; }
. {}
%%

int main() {
   printf("160120733301\n");
   extern FILE *yyin;
   printf("LineNumber\tLexme\t\tToken\n");
   yyin = fopen("hello.c", "r");
   yylex();
   return 0;
}
```

OUTPUT

```
~/.../lab/cd-lab0702 ↰  flex parse.l
~/.../lab/cd-lab0702 ↰  cc -lfl lex.yy.c
~/.../lab/cd-lab0702 ↰  ./a.out
160120733301
LineNumber      Lexme           Token
1       "include"                       Keyword
1       "<"                     Special
1       "stdio"                 Identifier
1       "h"                     Identifier
1       ">"                     Special
3       "int"                   Keyword
3       "main"                  Identifier
3       "("                     Special
3       ")"                     Special
3       "{"                     Special
4       "printf"                        Identifier
4       "("                     Special
4       "Hello"                 Identifier
4       "World"                 Identifier
4       ")"                     Special
4       ";"                     Special
5       "       return"                 Keyword
5       ";"                     Special
6       "}"                     Special
```

CONCLUSION

Hence, lex program to design token separator has been implemented successfully.

EXPERIMENT NO.: 09

AIM

Write a program to implement FIRST function.

DESCRIPTION

.Let α be a string of tokens and nonterminals. FIRST(α) is the set of all tokens t so that α ⇒* tβ for some string β. Additionally, FIRST(α) contains ε if α ⇒* ε. That is, FIRST(α) contains all tokens that can begin α, plus ε if α can be erased.

ALGORITHM

For each nonterminal N, start with FIRST(N) = {} and add members as necessary until no more members can be added.

To compute first(t) and first (N)

1. If t is a token then FIRST(t) = {t}.

2. If N is a nonterminal, find all productions with N on the left-hand side. For each such production N → Y1Y2…Yn (n ≥ 0), do the following.

   a. Add all tokens in FIRST(Y1) to FIRST(N).

   b. For k = 2, … n−1
      if ε is in every one of FIRST(Yj), for j = 1, …, k−1,
      then add all tokens in FIRST(Yj) to FIRST(N).
   c. If ε is in FIRST(Yj) for j = 1, …, n, then add ε to FIRST(N).

The next step is to define FIRST(α) for every string of tokens and nonterminals α.

To compute FIRST(α) for strings α

1. FIRST(ε) = {ε}

2. If α begins with token t then FIRST(α) = {t}.

3. If α = Nβ starts with nonterminal N, then

   a. If FIRST(N) does not contain ε then FIRST(α) = FIRST(N).

   b. If FIRST(N) contains ε then FIRST(α) = FIRST(N) ∪ FIRST(β).

PROGRAM/CODE

```
E = 'Є'
is_terminal = lambda x: not x.isupper()

def first(head: str, prods: dict):
    if is_terminal(prods[head][0][0]):
        ret = [p[0][0] for p in prods[head] if is_terminal(p[0][0])]
        for p in prods[head]:
            if E in p: ret.append(E)
        return ret
    return first(prods[head][0][0], prods)

def main():
    prods = {}
    p = int(input("Enter no.of prods: "))
    for i in range(1, p+1):
        buf = input(f"{i}: ")
        head, body = [x.strip() for x in buf.split("->")]
        body = [[y for y in x.strip().split(" ")] for x in body.split("|")]
        prods[head] = body
    for (k, _) in prods.items():
        print(f'FIRST({k})', first(k, prods))

if __name__ == "__main__":
    print("160120733301")
    exit(main() or 0)
```

OUTPUT

```
~/.../lab/cd-lab1402 ϟ  python first.py
160120733301
Enter no.of prods: 5
1: E  -> T E'
2: E' -> + T E' | Є
3: T  -> F T'
4: T' -> * F T' | Є
5: F  -> ( E ) | i
FIRST(E) ['(', 'i']
FIRST(E') ['+', 'Є']
FIRST(T) ['(', 'i']
FIRST(T') ['*', 'Є']
FIRST(F) ['(', 'i']
```

CONCLUSION

Hence, the program to implement FIRST function has been executed successfully.

EXPERIMENT NO.: 10

AIM

Write a program to implement FOLLOW function.

DESCRIPTION

Let N be a nonterminal of G. FOLLOW(N) is a set of tokens, possibly additionally including special symbol $ indicating end-of-input. FOLLOW(N) contains all tokens t so that there exists a sentential form that contains substring N t. That is, t is in FOLLOW(N) provided t can follow N in a derivation. Additionally, FOLLOW(N) contains $ if N there exists a sentential form that ends on N.

ALGORITHM

Start with FOLLOW(N) = {} for every nonterminal N. Then perform the following steps until none of the FOLLOW sets can be enlarged any more.

1. Add $ to FOLLOW(S), where S is the start nonterminal.

2. If there is a production A → αBβ, then add every token that is in FIRST(β) to FOLLOW(B). (Do not add ε to FOLLOW(B).

3. If there is a production A → αB, then add all members of FOLLOW(A) to FOLLOW(B). (If t can follow A, then there must be a sentential form βAtγ Using production A → αB gives sentential form βαBtγ, where B is followed by t.

4. If there is a production A → αBβ where FIRST(β) contains ε, then add all members of FOLLOW(A) to FOLLOW(B). (Reasoning is like rule 3. Just erase β.)

PROGRAM/CODE

```
E = 'Є'
is_terminal = lambda x: not x.isupper()
prods = {}
first_set = {}
follow_set = {}

def first(head: str, prods: dict):
    if is_terminal(prods[head][0][0]):
        ret = [p[0][0] for p in prods[head] if is_terminal(p[0][0])]
        for p in prods[head]:
            if E in p: ret.append(E)
        return ret
    return first(prods[head][0][0], prods)
```

```
def follow(rel: dict):
    global E, prods, first_set, follow_set
    for cur in rel.keys():
        if cur == list(rel.keys())[0]:
            follow_set[list(rel.keys())[0]] = ['$']
        for h in rel[cur]:
            for prod in prods[h]:
                if cur in prod:
                    idx = prod.index(cur)
                    if idx+1 == len(prod):
                        if h != cur:
                            follow_set[cur].extend(follow_set[h])
                    elif is_terminal(prod[idx+1]):
                        follow_set[cur].append(prod[idx+1])
                    else:
                        f = first_set[prod[idx+1]]
                        if E in f:
                            f.remove(E)
                            follow_set[cur].extend(f)
                            if cur != h:
                                follow_set[cur].extend(follow_set[h])


def main():
    global prods, first_set, follow_set
    p = int(input("Enter no.of prods: "))
    for i in range(1, p+1):
        buf = input(f"{i}: ")
        head, body = [x.strip() for x in buf.split("->")]
        body = [[y for y in x.strip().split(" ")] for x in body.split("|")]
        prods[head] = body
    for (k, _) in prods.items():
        first_set[k] = first(k, prods)
    rel = {k:[x[0] for x in prods.items()
                    if any([k in y for y in x[1]])]
                        for k in prods.keys()}
    follow_set = {k: [] for k in prods.keys()}
    follow(rel)
    for (head, body) in follow_set.items():
        print(f"FOLLOW({head})", body)

if __name__ == "__main__":
    print("160120733301")
    exit(main() or 0)
```

OUTPUT

```
~/.../lab/cd-lab1402 ↳  python follow.py
160120733301
Enter no.of prods: 5
1: E  -> T E'
2: E' -> + T E' | Є
3: T  -> F T'
4: T' -> * F T' | Є
5: F  -> ( E ) | i
FOLLOW(E) ['$', ')']
FOLLOW(E') ['$', ')']
FOLLOW(T) ['+', '$', ')']
FOLLOW(T') ['+', '$', ')']
FOLLOW(F) ['*', '+', '$', ')']
```

CONCLUSION

Hence, the program to implement FOLLOW function has been executed successfully.

EXPERIMENT NO.: 11

AIM

Write a program to implement LL(1) parsing table.

DESCRIPTION

LL(1) Parsing: Here the 1st L represents that the scanning of the Input will be done from Left to Right manner and the second L shows that in this parsing technique we are going to use Left most Derivation Tree. And finally, the 1 represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

Essential conditions to check first are as follows:

- The grammar is free from left recursion.
- The grammar should not be ambiguous.
- The grammar has to be left factored in so that the grammar is deterministic grammar.

ALGORITHM

Step 1: First check all the essential conditions mentioned above and go to step 2.

Step 2: Calculate First() and Follow() for all non-terminals.

Step 3: For each production A –> α. (A tends to alpha)

1. Find First(α) and for each terminal in First(α), make entry A –> α in the table.
2. If First(α) contains ε (epsilon) as terminal, then find the Follow(A) and for each terminal in Follow(A), make entry A –> ε in the table.
3. If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> ε in the table for the $.

PROGRAM/CODE

```
E = 'Є'
is_terminal = lambda x: not x.isupper()
prods = {}
first_set = {}
follow_set = {}
ll1_table = {}

def first(head: str, prods: dict):
    if is_terminal(prods[head][0][0]):
        ret = [p[0][0] for p in prods[head] if is_terminal(p[0][0])]
        for p in prods[head]:
            if E in p: ret.append(E)
        return ret
    return first(prods[head][0][0], prods)
```

```
def follow(rel: dict):
    global E, prods, first_set, follow_set
    for cur in rel.keys():
        if cur == list(rel.keys())[0]:
            follow_set[list(rel.keys())[0]] = ['$']
        for h in rel[cur]:
            for prod in prods[h]:
                if cur in prod:
                    idx = prod.index(cur)
                    if idx+1 == len(prod):
                        if h != cur:
                            follow_set[cur].extend(follow_set[h])
                    elif is_terminal(prod[idx+1]):
                        follow_set[cur].append(prod[idx+1])
                    else:
                        f = first_set[prod[idx+1]].copy()
                        if E in f:
                            f.remove(E)
                            follow_set[cur].extend(f)
                            if cur != h:
                                follow_set[cur].extend(follow_set[h])

def ll1():
    global E, prods, first_set, follow_set, ll1_table
    for (k, v) in prods.items():
        for t in first_set[k]:
            if t == E:
                for b in follow_set[k]:
                    ll1_table[k][b] = {k: E}
            else:
                is_t_in_body = [pd for pd in range(len(v)) if t in v[pd]]
                if not is_t_in_body:
                    ll1_table[k][t] = {k: v}
                else:
                    ll1_table[k][t] = {k: v[is_t_in_body[0]]}
```

```
def main():
    global prods, first_set, follow_set, ll1_table
    p = int(input("Enter no.of prods: "))
    for i in range(1, p+1):
        buf = input(f"{i}: ")
        head, body = [x.strip() for x in buf.split("->")]
        body = [[y for y in x.strip().split(" ")] for x in body.split("|")]
        prods[head] = body
    for (k, _) in prods.items():
        first_set[k] = first(k, prods)
    rel = {k:[x[0] for x in prods.items()
                    if any([k in y for y in x[1]])]
                        for k in prods.keys()}
    follow_set = {k: [] for k in prods.keys()}
    follow(rel)
    terms = set(['$'])
    for prod in prods.values():
        for l in prod:
            for el in l:
                if is_terminal(el):
                    terms.add(el)
    ll1_table = {r: {c: None for c in terms} for r in first_set.keys()}
    ll1()
    print("LL(1) Parsing Table:-")
    for (r, c) in ll1_table.items():
        for (h, b) in c.items():
            print(r, h, b, sep="\t")
        print()

if __name__ == "__main__":
    print("160120733301")
    exit(main() or 0)
```

OUTPUT

```
~/.../lab/cd-lab1402 ↳  python ll1.py
160120733301
Enter no.of prods: 5
1: E  -> T E'
2: E' -> + T E' | Є
3: T  -> F T'
4: T' -> * F T' | Є
5: F  -> ( E ) | i
LL(1) Parsing Table:-
E       )       None
E       *       None
E       i       {'E': [['T', 'E'']]}
E       +       None
E       (       {'E': [['T', 'E'']]}
E       $       None

E'      )       {'E'': 'Є'}
E'      *       None
E'      i       None
E'      +       {'E'': ['+', 'T', 'E'']}
E'      (       None
E'      $       {'E'': 'Є'}

T       )       None
T       *       None
T       i       {'T': [['F', 'T'']]}
T       +       None
T       (       {'T': [['F', 'T'']]}
T       $       None

T'      )       {'T'': 'Є'}
T'      *       {'T'': ['*', 'F', 'T'']}
T'      i       None
T'      +       {'T'': 'Є'}
T'      (       None
T'      $       {'T'': 'Є'}

F       )       None
F       *       None
F       i       {'F': ['i']}
F       +       None
F       (       {'F': ['(', 'E', ')']}
F       $       None
```

CONCLUSION

Hence, the program to implement LL(1) parsing table has been executed successfully.

EXPERIMENT NO.: 12

AIM

Write a lex program to find given number is odd or even.

DESCRIPTION

Given a file, i.e., a text file. To find given number is odd or even.

ALGORITHM

Step 1: Read the given file character by character.

Step 2: Apply Regular expressions for the numbers.

Step 3: Write the number as even or odd to output.

Step 4: Repeat steps 2 and 3, until EOF (End Of File) is reached.

Step 5: Close the file and End the problem.

PROGRAM/CODE

```
%{
#include<stdio.h>
#include<stdlib.h>
int number = 0;
%}

%%
[0-9]+ { number = atoi(yytext); }
\n { return 0; }
. {}
%%

int yywrap() {}

int main() {
   printf("160120733301\n");
   yylex();
   if (number % 2)
      printf("%d is Odd Number!\n", number);
   else
      printf("%d is Even Number!\n", number);
   return 0;
}
```

OUTPUT

```
● ~/.../lab/lab-2802 ᛋ  flex oddeven.l
● ~/.../lab/lab-2802 ᛋ  cc -lfl lex.yy.c
● ~/.../lab/lab-2802 ᛋ  ./a.out
  160120733301
  2
  2 is Even Number!
● ~/.../lab/lab-2802 ᛋ  ./a.out
  160120733301
  3
  3 is Odd Number!
```

CONCLUSION

Hence, lex program to find given number is even or odd has been implemented successfully.

EXPERIMENT NO.: 13

AIM

Write a lex program to identify characters other than alphabets.

DESCRIPTION

Given a file, i.e., a text file. To identify characters other than alphabets. i.e., white space, special symbols, etc should be identified.

ALGORITHM

Step 1: Read the given file character by character.

Step 2: Apply Regular expressions for the identifying other than alphabets and number.

Step 3: Write the identified characters to output.

Step 4: Repeat steps 2 and 3, until EOF (End Of File) is reached.

Step 5: Close the file and End the problem.

PROGRAM/CODE

```
%{
#include<stdio.h>
%}

%%
\n { return 0; }
[^A-Za-z] { printf("%s\n", yytext); }
. {}
%%

int yywrap() {}

int main() {
   printf("160120733301\n");
   printf("Special Character:-\n");
   yylex();
   return 0;
}
```

OUTPUT

```
● ~/.../lab/lab-2802 ↳  flex specialchar.l
● ~/.../lab/lab-2802 ↳  cc -lfl lex.yy.c
● ~/.../lab/lab-2802 ↳  ./a.out
  160120733301
  Special Character:-
  Hello, World!
  ,

  !
```

CONCLUSION

Hence, lex program to identified characters other than alphabets has been implemented successfully.

EXPERIMENT NO.: 14

AIM

Write a lex program to add line number to the statements.

DESCRIPTION

Given a file, i.e., a text file. To add line number to the statements.

ALGORITHM

Step 1: Read the given file character by character.

Step 2: Apply Regular expressions for the statements / lines.

Step 3: Write the line number and statements to output.

Step 4: Repeat steps 2 and 3, until EOF (End Of File) is reached.

Step 5: Close the file and End the problem.

PROGRAM/CODE

```
%{
#include<stdio.h>
int line = 1;
%}

%%
^.* { printf("%d: %s", line++, yytext); }
. {}
%%

int yywrap() {}

int main(int argc, char** argv) {
   printf("160120733301\n");
   if (argc < 2) {
      printf("E: Expected file path!\n");
      return 1;
   }
   extern FILE *yyin;
   yyin = fopen(argv[1], "r");
   yylex();
   return 0;
}
```

OUTPUT

```
● ~/.../lab/lab-2802 ↳  flex addlines.l
● ~/.../lab/lab-2802 ↳  cc -lfl lex.yy.c
⊗ ~/.../lab/lab-2802 ↳  ./a.out
  160120733301
  E: Expected file path!
● ~/.../lab/lab-2802 ↳  ./a.out text.txt
  160120733301
  1: Hello
  2: two
  3: three
```

CONCLUSION

Hence, lex program to add line number to the statements has been implemented successfully.

EXPERIMENT NO.: 15

AIM

Write a program to implement Recursive Descent Parser.

DESCRIPTION

A recursive descent parser is a type of top-down parser that uses a recursive procedure to analyze the input based on a set of rules or grammar. The parser starts at the root of the grammar and recursively calls itself to evaluate the input and match it against the grammar rules. Each non-terminal symbol in the grammar is associated with a parsing method, which is called recursively to process the input according to the corresponding grammar rule. The parser advances through the input by consuming tokens and matching them against the grammar rules until a complete parse tree is constructed or an error is encountered. Recursive descent parsing is simple and easy to understand but may suffer from issues like left-recursion and backtracking.

ALGORITHM

1. Define a grammar

Define the grammar that will be used for parsing. This can be done in the form of a set of production rules.

2. Define parsing functions for each non-terminal symbol

For each non-terminal symbol in the grammar, define a function that will parse that symbol. These functions should take in the current position in the input string and return the result of the parse (e.g. a syntax tree).

3. Implement tokenization

Before parsing can begin, the input string needs to be split into individual tokens. These tokens will be used to match against the non-terminal symbols in the grammar.

4. Begin parsing

Start parsing by calling the function that corresponds to the starting non-terminal symbol of the grammar. This function will call other parsing functions as needed to complete the parse.

5. Match tokens

As each non-terminal symbol is parsed, it will need to match against the appropstring. If a match fails, the parser can return an error.

6. Return result

Once the parsing is complete, the parser should return the final result of the parse (e.g. the syntax tree).

7. Handle ambiguities

Recursive descent parsers can be ambiguous (i.e. a grammar may have more than one valid parse tree for a given input string). If the grammar is ambiguous, the parsing functions will

need to be modified to handle the ambiguity (e.g. by considering different possible parse trees).

PROGRAM/CODE

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

#define SUCCESS 1
#define FAILED 0

int E(), Edash(), T(), Tdash(), F();

const char *cursor;
char string[64];

int main() {
   puts("Enter the string");
   scanf("%s", string);
   cursor = string;
   puts("");
   puts("Input     Action");
   if (E() && *cursor == '\0') {
      puts("String is successfully parsed");
      return 0;
   } else {
      puts("Error in parsing String");
      return 1;
   }
}

int E() {
   printf("%-16s E -> T E'\n", cursor);
   if (T()) {
      if (Edash())
         return SUCCESS;
      else
         return FAILED;
   } else
      return FAILED;
}

int Edash() {
   if (*cursor == '+') {
      printf("%-16s E' -> + T E'\n", cursor);
      cursor++;
      if (T()) {
         if (Edash())
            return SUCCESS;
```

```
        else
            return FAILED;
      } else
        return FAILED;
   } else {
      printf("%-16s E' -> $\n", cursor);
      return SUCCESS;
   }
}

int T() {
   printf("%-16s T -> F T'\n", cursor);
   if (F()) {
      if (Tdash())
         return SUCCESS;
      else
         return FAILED;
   } else
      return FAILED;
}

int Tdash() {
   if (*cursor == '*') {
      printf("%-16s T' -> * F T'\n", cursor);
      cursor++;
      if (F()) {
         if (Tdash())
            return SUCCESS;
         else
            return FAILED;
      } else
         return FAILED;
   } else {
      printf("%-16s T' -> $\n", cursor);
      return SUCCESS;
   }
}

int F() {
   if (*cursor == '(') {
      printf("%-16s F -> ( E )\n", cursor);
      cursor++;
      if (E()) {
         if (*cursor == ')') {
            cursor++;
            return SUCCESS;
         } else
            return FAILED;
```

```
    } else
        return FAILED;
  } else if (*cursor == 'i') {
    cursor++;
    printf("%-16s F -> i\n", cursor);
    return SUCCESS;
  } else
    return FAILED;
}
```

OUTPUT

```
~/.../lab/cd-lab2103 ↳  cc rdp.c && ./a.out
Enter the string
i+i

Input           Action
i+i             E -> T E'
i+i             T -> F T'
+i              F -> i
+i              T' -> $
+i              E' -> + T E'
i               T -> F T'
                F -> i
                T' -> $
                E' -> $
String is successfully parsed
```

CONCLUSION

Hence, The program to implement Recursive Descent Parser has been implemented successfully.

EXPERIMENT NO.: 16

AIM

Write a program to find LR(0) Items for a given grammar.

DESCRIPTION

LR(0) items are the potential states that a LR(0) parser could be in while parsing a given grammar. They consist of a production rule, a dot to indicate the current position in the rule, and a lookahead symbol. The lookahead symbol is the next input symbol that the parser needs to read before making a shift or reduce decision. The LR(0) items represent all possible configurations of the parser's stack during parsing. The LR(0) items do not take into account any future input symbols beyond the lookahead symbol, and hence, the parser may make incorrect decisions in certain situations. However, LR(0) items are useful for generating LR(0) parsing tables and provide a starting point for constructing more sophisticated parsing algorithms.

ALGORITHM

Input: A grammar in BNF form

Output: A list of all LR(0) items for the grammar

1. Start by adding an item for the start symbol S, with a dot at the beginning: S -> .A

2. For each production A -> alpha in the grammar, add an item with a dot at the beginning: A -> .alpha

3. For each item A -> alpha. B beta, where B is not epsilon, add an item with a dot after B and beta: B -> beta.

4. Repeat step 3 for all new items added in step 3 until no new items are added.

5. Return the list of all items found in steps 1-4.

PROGRAM/CODE

```
def findlr0(lhs, rhs, lr0):
    for i in range(len(rhs)+1):
        x = lhs + '->' + rhs[:i] + '.' + rhs[i:]
        lr0.append(x)

n = int(input("Enter the number of productions: "))
arr = []
for i in range(n):
    arr.append(str(input()))
lr0 = []
for i in range(len(arr)):
    ip = arr[i]
    lhs, rhs = ip.split("->")
    productions = list(rhs.split("|"))
    for prod in productions:
```

```
    findlr0(lhs, prod, lr0)
print("LR(0) items for given production: ")
for i in range(len(lr0)):
  print(i,"->",lr0[i])
```

OUTPUT

```
~/.../lab/cd-lab2103 ↳  python lr0.py
Enter the number of productions: 3
T->T*F
F->i
T->e
LR(0) items for given production:
0 -> T->.T*F
1 -> T->T.*F
2 -> T->T*.F
3 -> T->T*F.
4 -> F->.i
5 -> F->i.
6 -> T->.e
7 -> T->e.
```

CONCLUSION

Hence, The program to find LR(0) items for a given grammar has been implemented successfully.

EXPERIMENT NO.: 17

AIM

To recognize a valid arithmetic expression that uses operator +, -, *, % using lex and yacc tool.

DESCRIPTION

In this program, we try to recognize a valid arithmetic expression that uses operator +, -, *, % using lex and yacc tool.

ALGORITHM

Step-1: Start
Step-2: Create a lex file
Step-3: Declare the regular expressions
[a-zA-Z_][a-zA-Z_0-9]* return id;
[0-9]+(\.[0-9]*)? return num;
[+/*] return op;
. return yytext[0];
\n return 0;
Step-4: Create a Yacc file
Step-5: valid = 1
Step-6: Declare required variables
start : id '=' s ';'
s : id x
| num x
| '-' num x
| '(' s ')' x
;
x : op s
| '-' s
|
;
Step-7: Read the expression from user
Step-8: if(valid) Print “Valid Expression”
Step-9: else Print “Invalid Expression”
Step-10: End

PROGRAM/CODE

arith.l

```
%{
#include "y.tab.h"
%}
%%
[a-zA-Z_][a-zA-Z_0-9]* return id;
[0-9]+(\.[0-9]*)?     return num;
[+*\/]              return op;
.                 return yytext[0];
```

```
\n                return 0;
%%
int yywrap(){ return 1; }
```

arith.y

```
%{
#include<stdio.h>
int valid=1;
%}
%token num id op
%%
start : id '=' s ';'s : id x
   | num x
   | '-' num x
   | '(' s ')' x
   ;
x : op s
   | '-' s
   |
   ;
%%

int yyerror() {
 valid=0;
 printf("\nInvalid expression!\n");
 return 0;
}

int main() {
 printf("\nEnter the expression:\n");
 yyparse();
 if(valid)
   printf("\nValid expression!\n");
 return 0;
}
```

OUTPUT

```
● ~/.../lab/cd-lab0404 ↳  bison -dy arith.y && flex arith.l
● ~/.../lab/cd-lab0404 ↳  cc lex.yy.c y.tab.c --no-warnings
● ~/.../lab/cd-lab0404 ↳  ./a.out

  Enter the expression:
  a=b+c;

  Valid expression!
● ~/.../lab/cd-lab0404 ↳  ./a.out

  Enter the expression:
  a=bc

  Invalid expression!
```

CONCLUSION

Hence, The program to validate arithmetic expression using Yacc tool has been implemented successfully.

EXPERIMENT NO.: 18

AIM

To implement Yacc program to check valid variable followed by letter or digits.

DESCRIPTION

In this program, we try to implement Yacc program to check valid variable followed by letter or digits.

ALGORITHM

Step 1: Start the program.
Step 2: Reading an expression .
Step 3: Checking the validating of the given variable according to the rule using Yacc .
Step 4: Using expression rule print the result of the given values
Step 5: Stop the program.

PROGRAM/CODE

digit.l

```
%{
#include "y.tab.h"
%}
%%
[a-zA-Z_][a-zA-Z_0-9]* return letter;
[0-9] return digit;
. return yytext[0];
\n return 0;
%%
int yywrap(){ return 1; }
```

digit.y

```
%{
 #include<stdio.h>
 int yylex(void);
 int yyerror();
 int valid = 1;
%}
%token digit letter
%%
start : letter s
s :    letter s
    | digit s
    |
    ;
%%
int yyerror(){
 printf("\nIts not a identifier!\n");
```

```
 valid=0;
 return 0;
}
int main() {
 printf("Enter a name to tested for identifier: ");
 yyparse();
 if(valid)
  printf("\nIt is a identifier!\n");
 return 0;
}
```

OUTPUT

```
● ~/.../lab/cd-lab0404 ᛋ  bison -dy digit.y && flex digit.l
● ~/.../lab/cd-lab0404 ᛋ  cc lex.yy.c y.tab.c --no-warnings
● ~/.../lab/cd-lab0404 ᛋ  ./a.out
  Enter a name to tested for identifier: abc

  It is a identifier!
● ~/.../lab/cd-lab0404 ᛋ  ./a.out
  Enter a name to tested for identifier: 12l

  Its not a identifier!
```

CONCLUSION

Hence, The program to validate Identifier using Yacc tool has been implemented successfully.

EXPERIMENT NO.: 19

AIM

To implement Yacc program for implementing a calculator for computing the given expression using semantic rules of the YACC tool and LEX.

DESCRIPTION

In this program, we try to implement Yacc program for implementing a calculator for computing the given expression using semantic rules of the YACC tool and LEX.

ALGORITHM

Step 1: A Yacc source program has three parts as follows:
Declarations %% translation rules %% supporting C routines
Step 2: Declarations Section: This section contains entries that:
i. Include standard I/O header file.
ii. Define global variables.
iii. Define the list rule as the place to start processing.
iv. Define the tokens used by the parser. v. Define the operators and their precedence.
Step 3: Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.
Step 4: Programs Section: The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the yacc library when processing this file.
Step 5: Main- The required main program that calls the yyparse subroutine to start the program.
Step 6: yyerror(s) -This error-handling subroutine only prints a syntax error message.
Step 7: yywrap -The wrap-up subroutine that returns a value of 1 when the end of input occurs. The calc.lex file contains include statements for standard input and output, as programmar file information if we use the -d flag with the yacc command. The y.tab.h file contains definitions for the tokens that the parser program uses.
Step 8: calc.lex contains the rules to generate these tokens from the input stream.

PROGRAM/CODE

calc.l

```
%{
#include<stdio.h>
#include "y.tab.h";
extern int yylval;
%}
%%
[0-9]+ { yylval=atoi(yytext); return NUMBER; }
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap() { return 1; }
```

calc.y

```
%{
   #include<stdio.h>
   int yylex(void);
   int yyerror();
   int flag=0;
%}
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%
ArithmeticExpression: E{
        printf("\nResult=%d\n",$$);
        return 0;
       };
E:E'+'E {$$=$1+$3;}
 |E'-'E {$$=$1-$3;}
 |E'*'E {$$=$1*$3;}
 |E'/'E {$$=$1/$3;}
 |E'%'E {$$=$1%$3;}
 |'('E')' {$$=$2;}
 | NUMBER {$$=$1;}
;
%%
void main() {
  printf("\nEnter Any Arithmetic Expression\n");
  yyparse();
}
int yyerror() { return 0; }
```

OUTPUT

```
● ~/.../lab/cd-lab0404 ᛋ  bison -dy calc.y && flex calc.l
● ~/.../lab/cd-lab0404 ᛋ  cc lex.yy.c y.tab.c --no-warnings
● ~/.../lab/cd-lab0404 ᛋ  ./a.out

  Enter Any Arithmetic Expression
  3*10*10+1

  Result=301
```

CONCLUSION

Hence, The program to evaluate arithmetic expression using Yacc tool has been implemented successfully.

EXPERIMENT NO.: 20

AIM

To implement Yacc program to check whether given string is accepted or not.

DESCRIPTION

In this program, we try to implement Yacc program to check whether given string is accepted or not.

ALGORITHM

Step 1: Start the program.
Step 2: Reading an expression .
Step 3: Checking the validating of the given String according to the rule for given grammar using Yacc.
Step 4: Using expression rule print the result of the given values
Step 5: Stop the program.

PROGRAM/CODE

accept.l

```
%{
#include "y.tab.h";
%}
%%
[aA] return A;
[bB] return B;
\n return NL;
. return yytext[0];
%%
int yywrap() { return 1; }
```

accept.y

```
%{
#include<stdio.h>
#include<stdlib.h>
int yylex(void);
int yyerror(char* msg);
%}
%token A B NL
%%
stmt: S NL { printf("valid string\n"); exit(0); }
;
S: A S B |
;
%%
int yyerror(char *msg) {
 printf("invalid string\n");
 exit(0);
```

```
}

void main() {
 printf("enter the string\n");
 yyparse();
}
```

OUTPUT

```
● ~/.../lab/cd-lab0404 ↳  bison -dy accept.y && flex accept.l
● ~/.../lab/cd-lab0404 ↳  cc lex.yy.c y.tab.c --no-warnings
● ~/.../lab/cd-lab0404 ↳  ./a.out
  enter the string
  ab
  valid string
● ~/.../lab/cd-lab0404 ↳  ./a.out
  enter the string
  aab
  invalid string
```

CONCLUSION

Hence, The program to check given string is valid or not using Yacc tool has been implemented successfully.

EXPERIMENT NO.: 21

AIM

To implement a program for Symbol table management.

DESCRIPTION

In this program, we try to implement a program for Symbol table management.

ALGORITHM

Step 1: Start the program for performing insert, display, delete, search and modify option in symbol table
Step 2: Define the structure of the Symbol Table
Step 3: Enter the choice for performing the operations in the symbol Table
Step 4: If the entered choice is 1, search the symbol table for the symbol to be inserted. If the symbol is
already present, it displays “Duplicate Symbol”. Else, insert the symbol and the corresponding address in
the symbol table.
Step 5: If the entered choice is 2, the symbols present in the symbol table are displayed.
Step 6: If the entered choice is 3, the symbol to be deleted is searched in the symbol table.
Step 7: If it is not found in the symbol table it displays “Label Not found”. Else, the symbol is deleted.
Step 8: If the entered choice is 5, the symbol to be modified is searched in the symbol table.

PROGRAM/CODE

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

void main() {
 int i=0,j=0,x=0,n;
 void *p,*add[5];
 char ch,srch,b[15],d[15],c;
 printf("Expression terminated by $:");
 while((c=getchar())!='$') {
  b[i]=c;
  i++;
 }
 n=i-1;
 printf("Given Expression:");
 i=0;
 while(i<=n) {
  printf("%c",b[i]);
  i++;
 }
```

```
 printf("\n Symbol Table\n");
 printf("Symbol \t addr \t type");
 while(j<=n) {
  c=b[j];
  if(isalpha(toascii(c))) {
   p=malloc(c);
   add[x]=p;
   d[x]=c;
   printf("\n%c \t %d \t identifier\n",c,p);
   x++;
   j++;
  }
  else if (isdigit(c)) {
   p=malloc(c);
   add[x]=p;
   d[x]=c;
   printf("\n%c \t %d \t Constant\n",c,p);
   x++;
   j++;
  }
  else {
   ch=c;
   if(ch=='+'||ch=='-'||ch=='*'||ch=='=')
   {
    p=malloc(ch);
    add[x]=p;
    d[x]=ch;
    printf("\n %c \t %d \t operator\n",ch,p);
    x++;
    j++;
   }
  }
 }
}
```

OUTPUT

```
● ~/.../lab/cd-lab0404 ᛋ  cc symtable.c && ./a.out
Expression terminated by $:x=1+2$
Given Expression:x=1+2
 Symbol Table
Symbol   addr    type
x        -1623962944     identifier

 =       -1623962816     operator

1        -1623962736     Constant

 +       -1623962672     operator

2        -1623962608     Constant
```

CONCLUSION

Hence, The program to implement Symbol table has been implemented successfully.

EXPERIMENT NO.: 22

AIM

To implement language to an intermediate form.

DESCRIPTION

In this program, we try to implement language to an intermediate form.

ALGORITHM

Step-1: Start
Step-2: Define a structure three
Step-3: f1=fopen("sum.txt","r")
Step-4: f2=fopen("out.txt","w")
Step-5: while(fscanf(f1,"%s",s[len].data)!=EOF)
    Step-5.1: len++
Step-6: itoa(j,d1,7)
Step-7: strcat(d2,d1)
Step-8: strcpy(s[j].temp,d2)
Step-9: strcpy(d1,"")
Step-10: strcpy(d2,"t")
Step-11: if(!strcmp(s[3].data,"+"))
    Step-11.1: Print(s[j].temp,s[i+2].data,s[i+4].data)
    Step-11.2: j++
Step-12: else if(!strcmp(s[3].data,"-"))
    Step-12.1: Print(s[j].temp,s[i+2].data,s[i+4].data)
    Step-12.2: j++
Step-13: for(i=4;i<len-2;i+=2)
    Step-13.1: itoa(j,d1,7)
    Step-13.2: strcat(d2,d1)
    Step-13.3: strcpy(s[j].temp,d2)
    Step-13.4: if(!strcmp(s[i+1].data,"+"))
        Step-13.4.1: Print(s[j].temp,s[j-1].temp,s[i+2].data)
    Step-13.5: else if(!strcmp(s[i+1].data,"-"))
        Step-13.5.1: Print(s[j].temp,s[j-1].temp,s[i+2].data)
    Step-13.6: strcpy(d1,"")
    Step-13.7: strcpy(d2,"t")
    Step-13.8: j++
Step-14: Print(s[0].data,s[j-1].temp)
Step-15: fclose(f1)
Step-16: fclose(f2)
Step-17: End

PROGRAM/CODE

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

```
struct three {
  char data[10],temp[7];
} s[30];

int main() {
  char d1[7],d2[7]="t";
  int i=0,j=1,len=0;
  FILE *f1,*f2;
  f1=fopen("input.txt","r");
  f2=fopen("output.txt","w");
  while(fscanf(f1,"%s",s[len].data)!=EOF) len++;
  itoa(j,d1,7);
  strcat(d2,d1);
  strcpy(s[j].temp,d2);
  strcpy(d1,"");
  strcpy(d2,"t");
  if(!strcmp(s[3].data,"+")) {
    fprintf(f2,"%s=%s+%s",s[j].temp,s[i+2].data,s[i+4].data);
    j++;
  }
  else if(!strcmp(s[3].data,"-")) {
    fprintf(f2,"%s=%s-%s",s[j].temp,s[i+2].data,s[i+4].data);
    j++;
  }
  for(i=4;i<len-2;i+=2) {
    itoa(j,d1,7);
    strcat(d2,d1);
    strcpy(s[j].temp,d2);
    if(!strcmp(s[i+1].data,"+"))
      fprintf(f2,"\n%s=%s+%s",s[j].temp,s[j-1].temp,s[i+2].data);
    else if(!strcmp(s[i+1].data,"-"))
      fprintf(f2,"\n%s=%s-%s",s[j].temp,s[j-1].temp,s[i+2].data);
    strcpy(d1,"");
    strcpy(d2,"t");
    j++;
  }
  fprintf(f2,"\n%s=%s",s[0].data,s[j-1].temp);
  fclose(f1);
  fclose(f2);
}
```

OUTPUT

```
~/.../lab/cd-lab0404 ᛋ  cat input.txt
output = in1 + in2 + in3 - in4
~/.../lab/cd-lab0404 ᛋ  cat output.txt
t1=in1+in2
t2=t1+in3
t3=t2-in4
output=t3
```

CONCLUSION

Hence, The program to generate intermediate form has been implemented successfully.

EXPERIMENT NO.: 23

AIM

To implement Yacc program to check for relational operator.

DESCRIPTION

In this program, we try to implement Yacc program to check for relational operator.

ALGORITHM

Step 1: Start the program.
Step 2: Reading an expression .
Step 3: Checking the validating of the given expression for relational operator according to the rule using Yacc .
Step 4: Using expression rule print the result of the given values
Step 5: Stop the program.

PROGRAM/CODE

relop.l

```
%{
#include "y.tab.h"
%}
%%
[0-9]+       { yylval = atoi(yytext); return NUM; }
"=="         { return EQ; }
"!="         { return NEQ; }
"<"          { return LT; }
">"          { return GT; }
"<="         { return LTE; }
">="         { return GTE; }
[ \t]        ;
\n           ;
.            { return yytext[0]; }
%%
int yywrap() { return 1; }
```

relop.y

```
%{
#include <stdio.h>
%}

%token NUM
%token EQ NEQ LT GT LTE GTE
%left EQ NEQ LT GT LTE GTE

%start expr
%%
expr: NUM { printf("Expression: %d\n", $1); }
```

```
  | expr EQ expr { printf("Expression: %d == %d\n", $1, $3); }
  | expr NEQ expr { printf("Expression: %d != %d\n", $1, $3); }
  | expr LT expr { printf("Expression: %d < %d\n", $1, $3); }
  | expr GT expr { printf("Expression: %d > %d\n", $1, $3); }
  | expr LTE expr { printf("Expression: %d <= %d\n", $1, $3); }
  | expr GTE expr { printf("Expression: %d >= %d\n", $1, $3); }
  ;
%%
int main() {
  yyparse();
  printf("Sucess");
  return 0;
}

void yyerror(const char *s) { printf("Error: %s\n", s); }
```

OUTPUT

```
● ~/.../lab/cd-lab1104 ᛋ  bison -dy relop.y && flex relop.l
● ~/.../lab/cd-lab1104 ᛋ  cc lex.yy.c y.tab.c --no-warnings
⊗ ~/.../lab/cd-lab1104 ᛋ  ./a.out
  5!=3
  Expression: 5
  Expression: 3
  Expression: 5 != 3
  ^C
● ~/.../lab/cd-lab1104 ᛋ  ./a.out
  2=1
  Expression: 2
  Error: syntax error
```

CONCLUSION

Hence, The program to implement relational operators using Yacc tool has been implemented successfully.

EXPERIMENT NO.: 24

AIM

To implement a program for code generation into target code(assembly language).

DESCRIPTION

In this program, we try to implement a program for code generation into target code.

ALGORITHM

Step 1:Start.
Step 2:Open the input file "exe.txt" in read mode using fopen and store the file pointer in f1.
Step 3:Open the output file "exe1.txt" in write mode using fopen and store the file pointer in f2.
Step 4:Read data from the input file using fscanf and store it in the data array of the s structure until the end of file (EOF) is reached, while incrementing the len variable for each data read.
Step 5:Loop through the s structure array from index 0 to len:
a. Check if the current data in s[i].data is equal to "=".
b. If yes, then write corresponding assembly language instructions to the output file f2 based on the values of s[i+1].data, s[i+2].data, and s[i+3].data in the structure array s.
Step 6:print the result from the file.
Step 7: Stop.

PROGRAM/CODE

```
#include<stdio.h>
#include<string.h>
struct three {
 char data[10],temp[7];
}s[30];


int main() {
 char *d1,*d2;
 char c;
 int i=0,len=0;
 FILE *f1,*f2,*f3;
 f1=fopen("input.txt","r");
 f2=fopen("output.txt","w");
 while(fscanf(f1,"%s",s[len].data)!=EOF) len++;
 for(i=0;i<=len;i++) {
  if(!strcmp(s[i].data,"=")) {
   fprintf(f2,"\nLDA\t%s",s[i+1].data);
   if(!strcmp(s[i+2].data,"+"))
    fprintf(f2,"\nADD\t%s",s[i+3].data);
   if(!strcmp(s[i+2].data,"-"))
    fprintf(f2,"\nSUB\t%s",s[i+3].data);
   fprintf(f2,"\nSTA\t%s",s[i-1].data);
```

```
  }
 }
 fclose(f1);
 fclose(f2);
 f3=fopen("output.txt","r");
 while((c=fgetc(f3))!=EOF)
  printf("%c",c);
 fclose(f3);
 return 0;
}
```

Input.txt

t1 = in1 + in2
t2 = t1 + in3
t3 = t2 - in4
out = t3

OUTPUT

```
C:\Users\Admin\Downloads>gcc AL.c

C:\Users\Admin\Downloads>a.exe

LDA     in1
ADD     in2
STA     t1
LDA     t1
ADD     in3
STA     t2
LDA     t2
SUB     in4
STA     t3
LDA     t3
STA     out
```

CONCLUSION

Hence, The program to generate target code has been implemented successfully.

EXPERIMENT NO.: 25

AIM

To implement a program for code optimization.

DESCRIPTION

In this program, we try to implement a program for code optimization.

ALGORITHM

Step 1:Start.
Step 2:Start by defining the structures for op and pr with l and r as members, representing left and right sides of an assignment statement.
Step 3:Take input for the number of values n.
Step 4:Loop through n times and take input for left and right sides of the assignment statements, storing them in the op structure.
Step 5:Print the intermediate code by looping through op and displaying l and r values.
Step 6:Perform dead code elimination by looping through op and checking if the l value is present in the r value of other op structures. If present, store it in pr structure.
Step 7:Print the result of dead code elimination by looping through pr and displaying l and r values.
Step 8:Perform common expression elimination by looping through pr and checking if the r value of one pr structure is a substring of r value of other pr structures. If present, replace the common expression with the l value of the first pr structure.
Step 9:Print the result of common expression elimination by looping through pr and displaying l and r values.
Step 10:Finally, eliminate redundant assignments by looping through pr and checking for duplicate assignments with same l and r values. If found, mark the l value as '\0'.
Print the optimized code by looping through pr and displaying l and r values, excluding the ones with l value as '\0'.
Step 11: Stop.

PROGRAM/CODE

```
#include<stdio.h>
#include<string.h>
struct op {
  char l;
  char r[20];
} op[10],pr[10];


void main() {
  int a,i,k,j,n,z=0,m,q;
  char *p,*l;
  char temp,t;
  char *tem;
  printf("Enter the Number of Values:");
  scanf("%d",&n);
```

```
for(i=0;i<n;i++) {
  printf("left: ");
  scanf(" %c",&op[i].l);
  printf("right: ");
  scanf(" %s",&op[i].r);
}
printf("Intermediate Code\n") ;
for(i=0;i<n;i++) {
  printf("%c=",op[i].l);
  printf("%s\n",op[i].r);
}
for(i=0;i<n-1;i++) {
  temp=op[i].l;
  for(j=0;j<n;j++) {
    p=strchr(op[j].r,temp);
    if(p) {
      pr[z].l=op[i].l;
      strcpy(pr[z].r,op[i].
         r);
      z++;
    }
  }
}
pr[z].l=op[n-1].l;
strcpy(pr[z].r,op[n-1].r);
z++;
printf("\nAfter Dead Code Elimination\n");
for(k=0;k<z;k++) {
  printf("%c\t=",pr[k].l);
  printf("%s\n",pr[k].r);
}
for(m=0;m<z;m++) {
  tem=pr[m].r;
  for(j=m+1;j<z;j++) {
    p=strstr(tem,pr[j].r);
    if(p) {
      t=pr[j].l;
      pr[j].l=pr[m].l;
      for(i=0;i<z;i++) {
        l=strchr(pr[i].r,t) ;
        if(l) {
          a=l-pr[i].r;
          printf("pos: %d\n",a);
          pr[i].r[a]=pr[m].l;
        }
      }
    }
  }
```

```
  }
  printf("Eliminate Common Expression\n");
  for(i=0;i<z;i++) {
    printf("%c\t=",pr[i].l);
    printf("%s\n",pr[i].r);
  }
  for(i=0;i<z;i++) {
    for(j=i+1;j<z;j++) {
      q=strcmp(pr[i].r,pr[j].r);
      if((pr[i].l==pr[j].l)&&!q)
        pr[i].l='\0';
    }
  }
  printf("Optimized Code\n");
  for(i=0;i<z;i++) {
    if(pr[i].l!='\0') {
      printf("%c=",pr[i].l);
      printf("%s\n",pr[i].r);
    }
  }
}
```

OUTPUT

```
● ~/.../lab/cd-lab1104 ᛋ  cc copt.c
● ~/.../lab/cd-lab1104 ᛋ  ./a.out
Enter the Number of Values:5
left: a
right: 9
left: b
right: c+d
left: e
right: c+d
left: f
right: b+e
left: r
right: f
Intermediate Code
a=9
b=c+d
e=c+d
f=b+e
r=f

After Dead Code Elimination
b       =c+d
e       =c+d
f       =b+e
r       =f
pos: 2
Eliminate Common Expression
b       =c+d
b       =c+d
f       =b+b
r       =f
Optimized Code
b=c+d
f=b+b
r=f
```

CONCLUSION

Hence, The program to implement code optimizer has been implemented successfully.

EXPERIMENT NO.: 26

AIM

To implement a parser for small language.

DESCRIPTION

In this program, we try to implement a parser for LISP language.

ALGORITHM

Step 1: Start by defining the grammar rules of the language that the parser will be parsing.
Step 2: Create a lexical analyzer (also known as a lexer or scanner) that will read in the input code and tokenize it according to the grammar rules.
Step 3: Create a parser that will use the tokens generated by the lexer to construct a parse tree. The parse tree represents the structural relationship between the different parts of the code.
Step 4: Use a stack-based approach to parse the input code. The parser will push the tokens onto the stack and use a set of rules to determine how to reduce the input code into a parse tree.
Step 5: Implement error handling mechanisms to detect and recover from syntax errors in the input code.
Step 6: Once the parse tree has been constructed, the parser may do additional processing to generate intermediate code or perform semantic analysis.
Step 7: Finally, the parser may output the result of its processing in some form, such as machine code or a high-level representation of the input code.

PROGRAM/CODE

```
def generate_AST(string):
 number_symbols = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '.', '-']
 ind = 1
 arr_to_return = []
 while ind < len(string):
  char = string[ind]
  if char == "(":
   open_cnt = 1
   closed_cnt = 0
   sub_str = "("
   for c in string[ind + 1:]:
    if c == "(": open_cnt += 1
    if c == ")": closed_cnt += 1
    sub_str += c
    if open_cnt == closed_cnt: break
   arr_to_return.append(generate_AST(sub_str))
   ind += len(sub_str)
  elif char == " " or char == ")":
   ind += 1
  else:
   stop_ind = string.find(" ", ind)
```

```
    if stop_ind == -1:
      stop_ind = string.find(")", ind)
    s = string[ind:stop_ind]
    if all(x in number_symbols for x in list(s)):
      if s.find('-', 1) == -1:
        num = float(s)
        arr_to_return.append(num)
    else:
      arr_to_return.append(s)
    ind = stop_ind + 1
  return arr_to_return


if __name__ == "__main__":
  ip = "(first (list -1.5 (+ 2 3) 9))" # lisp
  print("Input(LISP):", ip)
  print("Output (AST):", generate_AST(ip))
```

OUTPUT

```
~/.../lab/cd-lablast ↳  python lisp_ast.py
Input(LISP): (first (list -1.5 (+ 2 3) 9))
Output (AST): ['first', ['list', -1.5, ['+', 2.0, 3.0], 9.0]]
```

CONCLUSION

Hence, The program to implement parser for small language has been implemented successfully.

EXPERIMENT NO.: 27

AIM

To implement a DFA to identify given string is a keyword or Identifier.

DESCRIPTION

In this program, we try to implement a DFA to identify given string is a keyword or Identifier.

ALGORITHM

Step 1: Start by defining the grammar rules of the for DFA.
Step 2: Read the input String.
Step 3: Pass the input string through DFA.
Step 4: if final state is reached;
Write “Input String is Keyword”
Step 5: Else
Write “Input String is Identifier”
Step 6: Print the Transitions of the DFA.

PROGRAM/CODE

```
DFA = {
  'i': set('n'),
  'n': set('t'),
  't': None
}

def main():
  ip = input("Enter identifier or int keyword: ")
  print(f'Transitions for {ip}')
  c, n = 0, len(ip)
  for lemme in ip:
    c += 1
    if lemme not in DFA.keys():
      print(ip[c-1:], '->', 'Identifier')
      break
    cur = DFA[lemme]
    if cur is None and c == n:
      print(lemme, '->', 'Keyword')
    else:
      print(lemme, '->', cur)

if __name__ == "__main__":
  exit(main() or 0)
```

OUTPUT

```
● ~/.../lab/cd-lablast ᛋ  python dfa.py
Enter identifier or int keyword: int
Transitions for int
i -> {'n'}
n -> {'t'}
t -> Keyword
● ~/.../lab/cd-lablast ᛋ  python dfa.py
Enter identifier or int keyword: intval
Transitions for intval
i -> {'n'}
n -> {'t'}
t -> None
val -> Identifier
```

CONCLUSION

Hence, The program to implement a DFA to identify given string is a keyword or Identifier has been implemented successfully.