ANNAMALAI UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B. E. (COMPUTER SCIENCE AND ENGINEERING)

VI - SEMESTER

CSCP607 – COMPILER DESIGN LAB MANUAL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
B.E. (COMPUTER SCIENCE AND ENGINEERING)

VISION:

To provide a congenial ambience for individuals to develop and blossom as academically superior, socially conscious and nationally responsible citizens.

MISSION:

- Impart high quality computer knowledge to the students through a dynamic scholastic environment wherein they learn to develop technical, communication and leadership skills to bloom as a versatile professional.
- Develop life-long learning ability that allows them to be adaptive and responsive to the changes in career, society, technology, and environment.
- Build student community with high ethical standards to undertake innovative research and development in thrust areas of national and international needs.
- Expose the students to the emerging technological advancements for meeting the demands of the industry.

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

PEO	PEO Statements
PEO1	To prepare the graduates with the potential to get employed in the right role and/or become entrepreneurs to contribute to the society.
PEO2	To provide the graduates with the requisite knowledge to pursue higher education and carry out research in the field of Computer Science.
PEO3	To equip the graduates with the skills required to stay motivated and adapt to the dynamically changing world so as to remain successful in their career.
PEO4	To train the graduates to communicate effectively, work collaboratively and exhibit high levels of professionalism and ethical responsibility.

CSCP607	COMPILER DESIGN LAB	L	T	P	C
		0	0	3	1.5

Course Objectives:

- To make the students to understand the different stages of compiler.
- To impart programming skills needed to develop a compiler.

LIST OF EXERCISES
(The exercises are to be done in python language)

1) a) Implementation of Lexical Analyzer
   b) Implementation of Lexical Tool
2) Conversion of Regular Expression to NFA
3) Elimination of Left Recursion
4) Left Factoring the given grammar
5) Computation of First and Follow Sets
6) Implementation of Recursive Descent Parser
7) Implementation of Shift Reduce Parser
8) Implementation of Intermediate Code Generator

Course Outcomes:

At the end of this course, the students will be able to

1. Apply the programming knowledge skill to design and develop a compiler.
2. Understand and implement lexical analyzer, syntax analyzer, code optimizer and code generator.
3. Demonstrate an ability to listen and answer the viva questions related to programming skills needed for solving real-world problems in Computer Science and Engineering.

Mapping of Course Outcomes with Programme Outcomes												
	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
CO1	2	2	-	-	-	-	-	-	-	-	-	-
CO2	2	2	3	3	-	-	-	-	-	-	-	-
CO3	2	2	-	-	-	-	-	-	-	2	-	2

Rubric for CO3

Rubric for CO3 in Laboratory Courses					
Rubric	**Distribution of 10 Marks for CIE/SEE Evaluation Out of 40/60 Marks**				
	Up To 2.5 Marks	**Up To 5 Marks**	**Up To 7.5 Marks**	**Up To 10 marks**	**Up To 2.5 Marks**
Demonstrate an ability to listen and answer the viva questions related to programming skills needed for solving real-world problems in Computer Science and Engineering.	Poor listening and communication skills. Failed to relate the programming skills needed for solving the problem.	Showed better communication skill by relating the problem with the programming skills acquired but the description showed serious errors.	Demonstrated good communication skills by relating the problem with the programming skills acquired with few errors.	Demonstrated excellent communication skills by relating the problem with the programming skills acquired and have been successful in tailoring the description.	Demonstrate an ability to listen and answer the viva questions related to programming skills needed for solving real-world problems in Computer Science and Engineering.

List of Exercises

EX.NO: 01(a) DATE:	Implementation of Lexical Analyzer

Aim:

To implement Lexical Analysis for given example text file using python coding.

Algorithm:

Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High- level input program into a sequence of Tokens. This sequence of tokens is sent to the parser for syntax analysis. Lexical Analysis can be implemented with the Deterministic finite Automata.

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages The types of token are identifier, numbers, operators, keywords, special symbols etc.

Following are the examples of tokens:

Keywords: Examples-for, while, if etc.
Identifier: Examples-Variable name, function name, etc. Operators: Examples '+', '++', '-' etc.
Separators: Examples ', ' ';' etc.

The algorithm for lexical analysis is as follows:
1. Read the input expression.
2. If the input is a keyword, store it as a keyword.
3. If the input is an operator, store it as operator.
4. If the input is a delimiter, store it as a delimiter.
5. Check whether input is a sequence of alphabets and/or digits then store it an identifier.
6. If the input is a sequence of digits, then store it as a number.

Program:

```
import re

patterns = {
    'IMPORTS' : r'<stdio.h>|<conio.h>|<stdlib.h>',
    'STRING' : r'\".*\"',
    'KEYWORD' : r'#include|if|else|for|break|int|float|void|String|char|double',
    'FUNCTION' : r'printf|scanf|clrscr|getch',
    'FLOAT' : r'\d+\.\d+',
    'INT' : r'\d+',
    'OPERATOR' : r'\+?\+|-|\*|/|=|==|<|>',
    'ID' : r'[a-zA-z_][a-zA-Z0-9_]*',
    'LPARAN' : r'\(',
    'RPARAN' : r'\)',
    'SEPRATOR' : r'[;:,]',
    'LBRACE' : r'\{',
    'RBRACE' : r'\}',
```

```
def lex_anz(input):
    tokens = []

    regex_patt = '|'.join(f'(?P<{tok}>{patterns[tok]})' for tok in patterns)
    for match in re.finditer(regex_patt, input):
        tok_type = match.lastgroup
        tok_val = match.group()
        tokens.append((tok_type, tok_val))
    return tokens

code = open('text.cpp').read()
result = lex_anz(code)
for t, v in result: print(f'{v} -> {t}')
```

Input:

```
#include <stdio.h>

void main(){
  int x = 3;
  if ( x < 10 ) {
    printf("hello world!");
  }
}
```

Output:

```
#include -> KEYWORD
<stdio.h> -> IMPORTS
void -> KEYWORD
main -> ID
( -> LPARAN
) -> RPARAN
{ -> LBRACE
int -> KEYWORD
x -> ID
= -> OPERATOR
3 -> INT
; -> SEPRATOR
if -> KEYWORD
( -> LPARAN
x -> ID
< -> OPERATOR
10 -> INT
) -> RPARAN
{ -> LBRACE
printf -> FUNCTION
( -> LPARAN
"hello world!" -> STRING
) -> RPARAN
; -> SEPRATOR
} -> RBRACE
} -> RBRACE
```

Result:

Thus, the python program to implement the Lexical Analyzer is executed successfully and verified.

EX.NO: 01(b) DATE:	Implementation of Lexical Tool

Aim:

To implement Lexical Analyzer using Lexical Tool in python coding.

Algorithm:

1. Define the set of tokens or lexemes for the programming language to be processed. These can be keywords, identifiers, operators, literals, etc. Store them in a dictionary or a list for easy access.
2. Read the input source code file to be processed.
3. Initialize a cursor or pointer to the beginning of the input source code.
4. Create a loop that iterates through the source code, character by character.
5. Implement a finite state machine (FSM) to recognize tokens. The FSM can have states representing different types of tokens, such as "keyword", "identifier", "operator", etc.
6. For each character encountered in the source code, update the FSM state based on the current character and the current state. If the current state is a final state, store the recognized token and reset the FSM state to the initial state. If the current state is not a final state and the current character does not transition to any valid state, then raise a lexical error.
7. Continue the loop until the end of the input source code is reached.
8. Output the recognized tokens along with their corresponding lexeme value or token type.

Program:

```
import ply.lex as lex

tokens = (
   'IMPORTS',
   'STRING',
   'KEYWORD',
   'FUNCTION',
   'FLOAT',
   'INT',
   'OPERATOR',
   'ID',
   'LPARAN',
   'RPARAN',
   'SEPRATOR',
   'LBRACE',
   'RBRACE',
)

t_IMPORTS = r'<stdio.h>|<conio.h>|<stdlib.h>'
t_STRING = r'\".*\"'
t_KEYWORD = r'\#include|if|else|for|break|int|float|void|String|char|double|while|do'
t_FUNCTION = r'printf|scanf|clrscr|getch'
t_FLOAT = r'\d+\.\d+'
t_INT = r'\d+'
t_OPERATOR = r'\+|-|\*|/|=|==|<|>|>=|<='
t_ID = r'[a-zA-z_][a-zA-Z0-9_]*'
t_LPARAN = r'\('
```

```
t_RPARAN = r'\)'
t_SEPRATOR = r'[;:,]'
t_LBRACE = r'\{'
t_RBRACE = r'\}'
t_ignore = r' |\t'

def t_NEWLINE(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

lexer = lex.lex()

code = open('text.cpp').read()

lexer.input(code)

while True:
    tok = lexer.token()
    if not tok:
        break
    print(tok)
```

Input:

```
#include <stdio.h>

void main(){
  int x = 3;
  if ( x < 10 ) {
    printf("hello world!");
  }
}
```

Output:

```
LexToken(KEYWORD,'#include',1,0)
LexToken(IMPORTS,'<stdio.h>',1,9)
LexToken(KEYWORD,'void',3,20)
LexToken(ID,'main',3,25)
LexToken(LPARAN,'(',3,29)
LexToken(RPARAN,')',3,30)
LexToken(LBRACE,'{',3,31)
LexToken(KEYWORD,'int',4,35)
LexToken(ID,'x',4,39)
LexToken(OPERATOR,'=',4,41)
LexToken(INT,'3',4,43)
LexToken(SEPRATOR,';',4,44)
LexToken(KEYWORD,'if',5,48)
```

```
LexToken(LPARAN,'(',5,51)
LexToken(ID,'x',5,53)
LexToken(OPERATOR,'<',5,55)
LexToken(INT,'10',5,57)
LexToken(RPARAN,')',5,60)
LexToken(LBRACE,'{',5,62)
LexToken(FUNCTION,'printf',6,69)
LexToken(LPARAN,'(',6,75)
LexToken(STRING,'"hello world!"',6,76)
LexToken(RPARAN,')',6,90)
LexToken(SEPRATOR,';',6,91)
LexToken(RBRACE,'}',7,95)
LexToken(RBRACE,'}',8,97)
```

Result:

Thus, the python program to implement the Lexical Analyzer using Lexical Tool is executed successfully and verified

EX.NO: 02 DATE:	Conversion of Regular Expression to NFA

Aim:

To write a python program to convert the given Regular Expression to NFA.
‘;

Algorithm:

Thompson’s Construction of an NFA from a Regular Expression:

Input: A regular expression r over the alphabet.
Output: An NFA N accepting L(r)

METHOD: Begin by parsing r into its constituent subexpressions. The rules for constructing an NFA consist of the following basics rules.

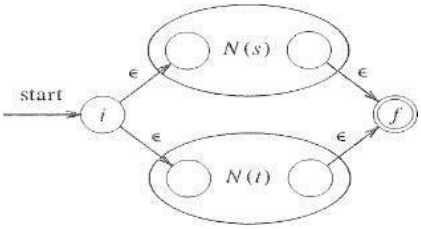
For expression e construct the NFA,

Here, i is a new state, the start state of this NFA, and f is another new state, the accepting state for the NFA.

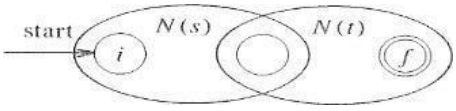
For any subexpressions, construct the NFA,

INDUCTION: Suppose N(s) and N (t) are NFA's for regular expressions s and t, respectively.

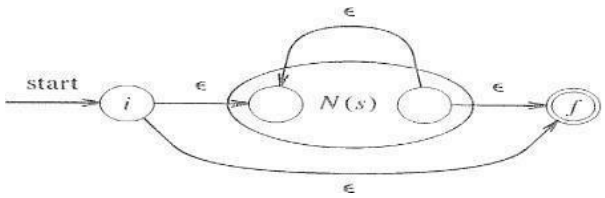
a) For the regular expression s|t ,



b) For the regular expression st,



c) For the regular expression S*,



d) Finally, suppose r = (s), then L(r) = L(s), and we can use the NFA, N(s) as N(r)

Program:

```
import re

t = 0; f = 1

def nodret(ip):
    global t, f
    e = u'\u03b5'
    nodes = []
    if re.match(re.compile(r'^[a-z]$'), ip):
        nodes = [
            (t, t + 1, ip)
        ]
        t += 1
    elif re.match(re.compile(r'^[a-z]\*$'), ip):
        nodes = [
            (t, t + 1, e),
            (t, t + 3, e),
            (t + 1, t + 2, ip[0]),
            (t + 2, t + 1, e),
            (t + 2, t + 3, e)
        ]
        t += 3
    elif re.match(re.compile(r'^[a-z]\/[a-z]$'), ip):
        nodes = [
            (t, t + 1, e),
            (t, t + 3, e),
            (t + 1, t + 2, ip[0]),
            (t + 3, t + 4, ip[2]),
            (t + 2, t + 5, e),
            (t + 4, t + 5, e),
        ]
        t += 5
    else:
        print("please enter basic expressions(linear combination of a, a*, a/b, a b)")
        f = 0
    return nodes

def tab_gen(v):
    ips = list(set([e for e1, e2, e in v]))
```

```
    ips.sort()

    a = [[[] for j in range(len(ips))] for i in range(t)]
    for s, d, i in v:
        a[s][ips.index(i)].append(d)
    print('state', end="")
    for x in ips:
        print(f'\t{x}', end='')

    print('\n', '-' * (len(ips) * 10))
    for i in range(t):
        print(f'{i}', end='')
        for j in range(len(ips)):
            print(f'\t{a[i][j]}', end='')
        print()
    print(f'State {t} is the final state')

ip = input("enter regex(leave space between characters): ")

nodes = []
for ch in ip.split():
    nodes += nodret(ch)
if f:
    tab_gen(nodes)
```

Input and output:

```
enter regex(leave space between characters): a* b* c/d
state   a       b       c       d       ε
 ---------------------------------------------------
0       []      []      []      []      [1, 3]
1       [2]     []      []      []      []
2       []      []      []      []      [1, 3]
3       []      []      []      []      [4, 6]
4       []      [5]     []      []      []
5       []      []      []      []      [4, 6]
6       []      []      []      []      [7, 9]
7       []      []      [8]     []      []
8       []      []      []      []      [11]
9       []      []      []      [10]    []
10      []      []      []      []      [11]
State 11 is the final state
```

Result:

Thus, the python program for construction of NFA table from Regular Expression is executed successfully and tested with various samples.

EX.NO: 03 DATE:	Elimination of Left Recursion

Aim:

To write a python program to implement Elimination of Left Recursion for given sample grammer.

Algorithm:

A grammar is left recursive if it has a nonterminal A such that there is a derivation A => Aα for some string. Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.

For each production rule `x` in the list of productions `p`:

- Initialize empty lists `alpha` and `beta`.
- Separate the productions of `x` into `alpha` (left recursive) and `beta` (non-left recursive) based on whether the production starts with the name of the non-terminal or not.
- If `alpha` is not empty:
  - Modify the right-hand side of productions in `beta` by appending the non-terminal's prime symbol.
  - Modify the right-hand side of productions in `alpha` by appending the non-terminal's prime symbol and epsilon.
  - Update the production rules for `x` with the modified `beta` productions.
  - Add new production rules for the non-terminal's prime symbol with the modified `alpha` productions to the list of productions `p`.

Program:

```
e=u'\u03b5'
p=[]

class Prod:
  def __init__(self, name, products):
    self.name=name
    self.products=products

  def print(self):
    s = f'{self.name} -> '
    for p in self.products:
      s+= f' {p} |'
    s=s.rstrip('|')
    print(s)

def trans():
    for x in p:
        alpha=[];beta=[]
    for product in x.products:
        if x.name==product[0]:
            alpha.append(product[1:])
        else:
            beta.append(product)
```

```
        if alpha:
            for i in range(len(beta)):
                beta[i]=f"{beta[i]}{x.name}'"
            for i in range(len(alpha)):
                alpha[i]=f"{alpha[i]}{x.name}'"
                alpha.append(e)
                x.products =beta
                p.append(Prod(f"{x.name}'",alpha))


n = int(input("No of production: "))
for i in range(n):
    ip = input(f"Production {i+1}: ")
    name, prods = ip.split(' -> ')
    products = prods.split(' | ')
    p.append(Prod(name, products))

print('Productions:')
for x in p: x.print()
print('Transforming...')
trans()
for x in p: x.print()
```

Input and Output:

```
No of production: 3
Production 1: E -> E+T | T
Production 2: T -> T*F | F
Production 3: F -> ( E ) | id
Productions:
E ->  E+T | T
T ->  T*F | F
F ->  (E) | id
Transforming...
E ->  TE'
T ->  FT'
F ->  (E) | id
E' ->  +TE' | ε
T' ->  *FT' | ε
```

Result:

Thus, the python program to implement elimination of left recursion is executed successfully and tested with various samples.

EX.NO: 04 DATE:	Left Factoring the given grammar

Aim:

To write a python program to implement Left Factoring for given sample grammar.

Algorithm:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

1. For each nonterminal A, find the longest prefix α common to two or more of its alternatives.

2. If a=∈, there is a non-trivial common prefix and hence replace all of A-productions, A → αβ1| αβ2 |…..| αβn| γ, where γ represents all alternatives that do not begin with α, by A →αA'| γ A' →β1| β2|…… |βn

3. Here, A' is new non terminal. Repeatedly apply this transformation until two alternatives for a non-terminal has common prefix

Program:

```
e = u'\u03b5'
p = []

class Prod:
    def __init__(self, name, products):
        self.name = name
        self.products = products

    def print(self):
        s = f'{self.name} -> '
        for p in self.products:
            s += f' {p} |'
        s = s.rstrip('|')
        print(s)

def trans():
    a = p[0]
    temp = a.products
    temp.sort()
    a.products = []
    while temp:
        group = []
        alpha = ''
        beta = []
        for i in range(1, len(temp)):
```

```
            if temp[0][0] == temp[i][0]:
                group.append(temp[i])
        if group:
            group.insert(0, temp[0])
            temp = [j for j in temp if j not in group]
            for j in range(len(group)):
                group[j] += e
            for c in group[0]:
                f1 = 0
                for j in group:
                    if c != j[0]:
                        f1 = 1
                if f1:
                    beta = group
                    break
                else:
                    alpha += group[0][0]
                    for j in range(len(group)):
                        group[j] = group[j][1:]
            for j in range(len(beta)):
                if beta[j][0] != e:
                    beta[j] = beta[j][:-1]
            a.products.append(alpha + alpha[0] + "'")
            p.append(Prod(alpha[0] + "'", beta))
        else:
            a.products.append(temp[0])
            temp.pop(0)

ip = input(f"Enter production: ")
name, prods = ip.split(' -> ')
products = prods.split(' | ')
p.append(Prod(name, products))

print('Productions:')
for x in p:
    x.print()
print('Transforming...')
trans()
for x in p:
    x.print()
```

Input and Output:

Enter production: A -> ABs | AB | Sed | Swa | p
Productions:
A -> ABs | AB | Sed | Swa | p
Transforming...
A -> ABA' | SS' | p
A' -> ε | s
S' -> ed | wa

Result:

Thus, the python program to implement elimination of left factoring is executed successfully and tested with various samples.

EX.NO: 05 DATE:	Computation of First and Follow Sets

Aim:

To write a python program to Compute First and Follow Sets for given sample grammar.

Algorithm:

1. Initialize an empty list `p` to store production rules.
2. Define the `Prod` class with attributes `name`, `products`, `first`, and `follow`.
3. Define a function `is_terminal` to check if a symbol is a terminal.
4. Define a function `find_prod` to find a production by name.
5. Define a function `calc_first` to calculate the `first` set for each non-terminal symbol.
6. Define a function `calc_follow` to calculate the `follow` set for each non-terminal symbol.
7. Define a function `find_follow` to find the `follow` set for a given non-terminal symbol.
8. Define a function `first` to retrieve the `first` set for a given non-terminal symbol.
9. Define a function `follow` to retrieve the `follow` set for a given non-terminal symbol.
10. Accept input for the number of productions `n`.
    a. For each production:
    b. Input the production rule in the format `A -> B1 | B2 | ... | Bn`.
    c. Split the input to extract the non-terminal symbol `name` and the list of productions `prods`.
    d. Split the list of productions `prods` into individual productions and create a `Prod` object for each non-terminal symbol.
11. Calculate the `first` and `follow` sets for each non-terminal symbol using `calc_first` and `calc_follow` functions.
12. Print the `first` and `follow` sets for each non-terminal symbol.

Program:

```
import re

p=[]
class Prod:
    def __init__(self, name, products):
        self.name=name
        self.products=products
        self.first=[]
        self.follow=[]

def is_terminal(s):
    if re.match(re.compile('^[A-Z]$'),s):
        return False
    else:
        return True

#find production by name:
def find_prod(name):
    for x in p:
        if x.name == name:
            return x
```

```
def first(name):
    for x in p:
        if name == x.name:
            return x.first

def follow(name):
    for x in p:
        if name == x.name:
            return x.follow

def calc_first():
    for i in reversed(range(len(p))):
        for x in p[i].products:
            if is_terminal(x[0]):
                p[i].first.append(x[0])
            else:
                f = find_prod(x[0]).first
                p[i].first.extend(f)
                c=1
                while 'e' in f:
                    if is_terminal(x[c]):
                        f=x[c]
                    else:
                        f=find_prod(x[c]).first

                    p[i].first.extend(f)
                    c+=1
                    if c == len(x): break
        p[i].first = list(set(p[i].first))

def calc_follow():
    p[0].follow.append('$')
    for x in p:
        find_follow(x)

def find_follow(x):
    for y in p:
        for pr in y.products:
            for c in range(len(pr)):
                if pr[c] == x.name:
                    if c+1 >= len(pr):
                        x.follow.extend(y.follow)
                    elif is_terminal(pr[c+1]):
                        x.follow.append(pr[c+1])
                    elif 'e' not in first(pr[c+1]):
                        x.follow.extend(first(pr[c+1]))
                    elif follow(pr[c+1]):
                        x.follow.extend(first(pr[c+1]) + follow(pr[c+1]))
                    else:
                        x.follow.extend(first(pr[c+1]) + find_follow(find_prod(pr[c+1])))
                    x.follow =  list(set(x.follow)-{'e'})
        return x.follow
n = int(input("No of production: "))
print("Epsilon = e")
```

```
for i in range(n):
    ip = input(f"Production {i+1}: ")
    name, prods = ip.split(' -> ')
    products = prods.split(' | ')
    p.append(Prod(name, products))

calc_first()
calc_follow()

#print first and follow
for x in p: print(f'first({x.name}) = {x.first}')
for x in p: print(f'follow({x.name}) = {x.follow}')
```

Input and Output:

```
No of production: 5
Epsilon = e
Production 1: E -> TX
Production 2: X -> +TX | e
Production 3: T -> FY
Production 4: Y -> *FY | e
Production 5: F -> (E) | i
first(E) = ['(', 'i']
first(X) = ['e', '+']
first(T) = ['(', 'i']
first(Y) = ['e', '*']
first(F) = ['(', 'i']
follow(E) = [')', '$']
follow(X) = [')', '$']
follow(T) = ['+', ')', '$']
follow(Y) = ['+', ')', '$']
follow(F) = ['+', '*', ')', '$']
```

Result:

Thus, the python program to implement the Computation of First and Follow sets is executed successfully and tested with various samples.

EX.NO: 06 DATE:	Implementation of Recursive Descent Parser

Aim:

To write a Python program that uses a Recursive Descent Parser to check if a given string is valid according to a specified grammar.

Algorithm:

Recursive descent parsing is a top-down parsing technique for context-free grammars that uses recursive functions to parse the input string. Each non-terminal symbol in the grammar has a corresponding parsing function. The parsing functions are called recursively to parse the input string, and they check if the current input character matches the expected symbol. Recursive descent parsing is simple and widely used, but left recursion in the grammar can cause infinite recursion and must be eliminated.

Input grammar:

E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> ε | i

1. Define functions `match(a)`, `F()`, `Tx()`, `T()`, `Ex()`, and `E()` to handle the grammar rules.
2. Initialize `s` as the input string converted to a list and `i` as the current index.
3. In `match(a)`, check if the current character matches `a` and increment `i` if it does.
4. In `F()`, check if the current character is '(' and if so, recursively check E and match ')'; if not, check if the current character is 'i'.
5. In `Tx()`, if the current character is '*', recursively check F and then Tx(); otherwise, return True.
6. In `T()`, recursively check F and then Tx().
7. In `Ex()`, if the current character is '+', recursively check T and then Ex(); otherwise, return True.
8. In `E()`, recursively check T and then Ex().
9. Call `E()` to check if the input string satisfies the grammar rules.
10. If `E()` returns True and `i` reaches the end of the input string, print "String is accepted".
11. If `E()` returns True but `i` does not reach the end of the input string, print "String is not accepted".
12. If `E()` returns False, print "String is not accepted".

Program:

```
print("Recursive Descent Parsing For following grammar\n")
print("E->TE'\nE'->+TE'/@\nT->FT'\nT'->*FT'/@\nF->(E)/i\n")
print("Enter the string want to be checked\n")
global s
s=list(input())
global i
i=0
def match(a):
    global s
    global i
    if(i>=len(s)):
        return False
    elif(s[i]==a):
        i+=1
        return True
```

```
    else:
        return False
def F():
    if(match("(")):
        if(E()): return match(")")
        else: return False
    else: return match("i")

def Tx():
    if(match("*")):
        if(F()): return Tx()
        else: return False
    else:
        return True
def T():
    if(F()): return Tx()
    else: return False
def Ex():
    if(match("+")):
        if(T()): return Ex()
        else: return False
    else: return True
def E():
    if(T()): return Ex()
    else: return False
if(E()):
    if(i==len(s)): print("String is accepted")
    else: print("String is not accepted")
else: print("string is not accepted")
```

Output:

```
Recursive Descent Parsing For following grammar

E->TE'
E'->+TE'/@
T->FT'
T'->*FT'/@
F->(E)/i
Enter the string want to be checked
i+i*(i)
String is accepted
```

Result:

Thus, the python program to check if a given string is valid using Recursive Descent Parser is executed successfully and tested with various samples

EX.NO: 07 DATE:	Implementation of Shift Reduce Parsing Algorithm

Aim:

To write a python program to implement the shift-reduce parsing algorithm.

Algorithm:

Shift-reduce parsing is a bottom-up parsing technique for context-free grammars that involves shifting input symbols onto a stack and reducing stack symbols using production rules. It is used in LR parsing, SLR parsing, and LALR parsing algorithms. The parsing process continues until the entire input string has been parsed or an error is detected. If the parsing process ends with the stack containing only the start symbol and the input buffer being empty, the input string is accepted; otherwise, it is rejected.
A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages The types of tokens are identifier, numbers, operators, keywords, special symbols etc.

While the input buffer is not empty:

1. For each production in the start symbol's right-hand side:
   If the production is in the stack, replace it with the start symbol and print a reduction action.
2. If the input buffer has more than one character:
   Add the first character of the input buffer to the stack and shift it to the right.
3. If the stack is equal to the end-of-string symbol followed by the start symbol, check if the input buffer is also empty.
   If the input buffer is empty, print "Accepted".
   If the input buffer is not empty, print "Rejected" and break the loop.

Program:

```
gram = {
   "S": ["S+S", "S*S",'S-S','(S)', "id"]
}
start = "S"
inp = "(id+id)$"
stack = "$"
print(f'{"Stack": <15}' + "|" + f'{"Input Buffer": <15}' + "|" + 'Parsing Action')
print(f'{"-":-<50}')
while True:
   i = 0
   for i in range (len(gram[start])):
      if gram[start][i] in stack:
         stack = stack.replace(gram[start][i], start)
         print(f'{stack: <15}' + "|" + f'{inp: <15}' + "|" + f'Reduce > {gram[start][i]}')
   if len(inp) > 1:
      stack += inp[0]
      inp = inp[1:]
      print(f'{stack: <15}' + "|" + f'{inp: <15}' + "|" + 'Shift')
   if stack == ("$" + start):
      if inp == '$':
         print(f'{stack: <15}' + "|" + f'{inp: <15}' + "|" + 'Accepted')
      else:
         print(f'{stack: <15}' + "|" + f'{inp: <15}' + "|" + 'Rejected')
```

```
        break
```

Output:

```
Stack         |Input Buffer   |Parsing Action
-------------------------------------------------
$(            |id+id)$        |Shift
$(i           |d+id)$         |Shift
$(id          |+id)$          |Shift
$(S           |+id)$          |Reduce > id
$(S+          |id)$           |Shift
$(S+i         |d)$            |Shift
$(S+id        |)$             |Shift
$(S+S         |)$             |Reduce > id
$(S+S)        |$              |Shift
$(S)          |$              |Reduce > S+S
$S            |$              |Reduce > (S)
$S            |$              |Accepted
```

Result:

Thus, the python program to implement the shift-reduce parsing algorithm is executed successfully and tested with various samples

EX.NO: 08 DATE:	Implementation of Intermediate Code Generator

Aim:

To write a python program to implement Intermediate Code Generator

Algorithm:

Intermediate code generation is a step-in compiler optimization that involves translating high-level source code into an intermediate representation for further analysis and optimization. This intermediate representation, often in the form of assembly-like instructions, is easier to analyze and optimize than high-level source code.

1. Define the set of operators `OPERATORS` and the precedence dictionary `PRI`.
2. Implement the `infix_to_postfix` function that takes a string formula as input and returns a postfix string.
3. Initialize an empty stack `stack` and an empty output string `output`.
4. Iterate through each character `ch` in the formula.
    4.1. If `ch` is not an operator, append it to the output string.
    4.2. If `ch` is an opening parenthesis, push it onto the stack.
    4.3. If `ch` is a closing parenthesis, pop and output stack elements until an opening parenthesis is reached.
    4.4. If `ch` is an operator, pop and output stack elements with higher or equal precedence until an operator with lower precedence or an opening parenthesis is reached.
5. After the loop, output any remaining elements in the stack.
6. Implement the `generate3AC` function that takes a postfix string `pos` as input and generates three-address code.
    6.1. Initialize an empty expression stack `exp_stack` and a temporary variable counter `t`.
    6.2. Iterate through each character `i` in the postfix string.
        6.2.1. If `i` is not an operator, push it onto the expression stack.
        6.2.2. If `i` is an operator, pop and output the top two elements from the expression stack, perform the operation, and push the result onto the expression stack.
    6.3. After the loop, the expression stack should contain only the final result.
7. Get the input expression from the user and convert it to postfix form using `infix_to_postfix`.
8. Generate three-address code using `generate3AC`.

Program:

```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])
PRI = {'+':1, '-':1, '*':2, '/':2}

def infix_to_postfix(formula):
    stack = []
    output = ''
    for ch in formula:
        if ch not in OPERATORS:
            output += ch
        elif ch == '(':
            stack.append('(')
        elif ch == ')':
            while stack and stack[-1] != '(':
```

```
            output += stack.pop()
        stack.pop()
    else:
        while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:
            output += stack.pop()
        stack.append(ch)
# leftover
while stack:
    output += stack.pop()
print(f'POSTFIX: {output}')
return output

def generate3AC(pos):
    print("### THREE ADDRESS CODE GENERATION ###")
    exp_stack = []
    t = 1

    for i in pos:
        if i not in OPERATORS:
            exp_stack.append(i)
        else:
            print(f't{t} := {exp_stack[-2]} {i} {exp_stack[-1]}')
            exp_stack=exp_stack[:-2]
            exp_stack.append(f't{t}')
            t+=1

expres = input("INPUT THE EXPRESSION: ")
pos = infix_to_postfix(expres)
generate3AC(pos)
```

Input and Output:

```
INPUT THE EXPRESSION: a=3+4*(8-7*(c-b)+2)
POSTFIX: a=3487cb-*-2+*+
### THREE ADDRESS CODE GENERATION ###
t1 := c - b
t2 := 7 * t1
t3 := 8 - t2
t4 := t3 + 2
t5 := 4 * t4
t6 := 3 + t5
```

Result:

Thus, the python program to implement Intermediate Code Generator is executed successfully and tested with various samples