SOMAIYA VIDYAVIHAR UNIVERSITY
Somaiya School of Basic and Applied Sciences

Somaiya TRUST

Practical Journal

for

Modern Compiler Design

Submitted by

Mr. Atharv Uday Soman

FY MSc CS Part I (Sem I)

Roll No: 32

Seat No: 44030725026

Submitted to: Ms. Ekta Gala

Academic Year: 2025 - 2026

Department of Computer Science,
Somaiya School of Basic and Applied Sciences
(SSBAS)

Somaiya Vidyavihar University

Index

Practical 01: Grammar Conversion

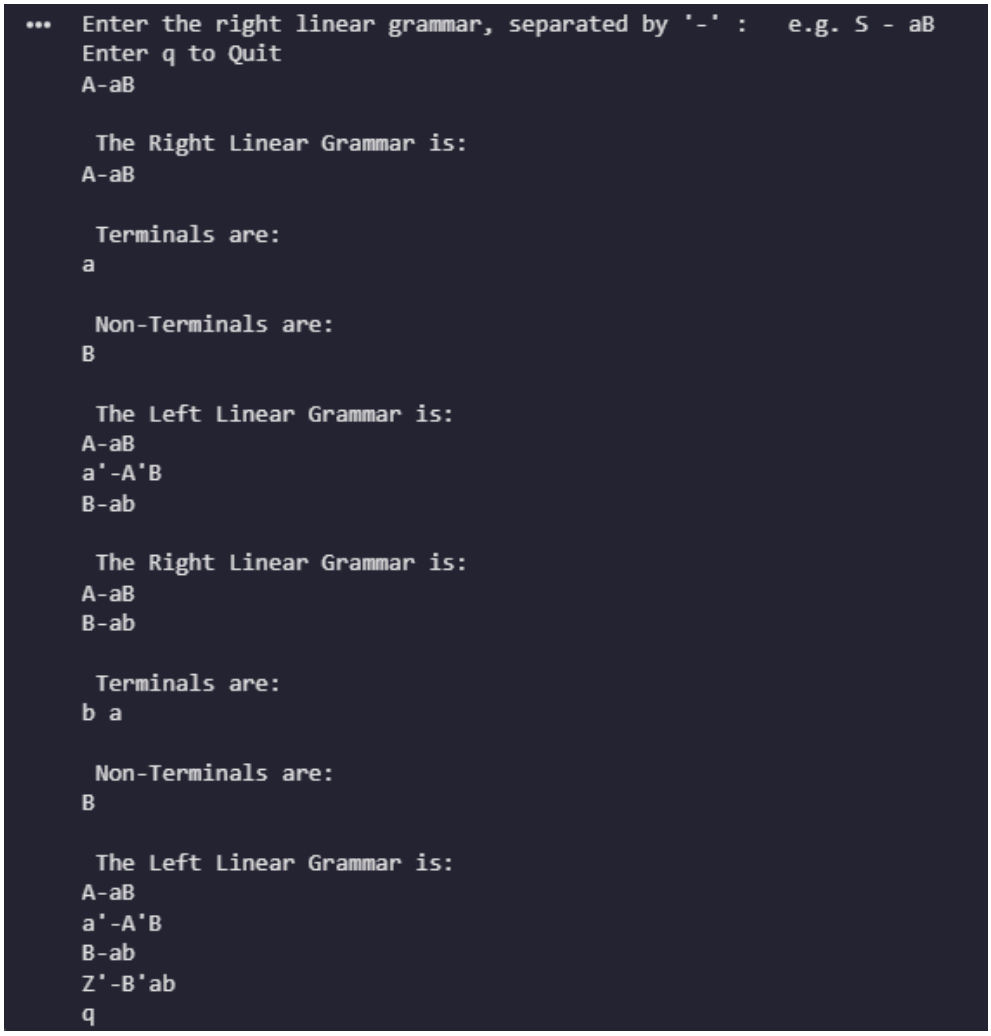
Aim - Converting Right Linear Grammar (RLG) to Left Linear Grammar (LLG)
Question: Convert Right Linear Grammar to Left Linear Grammar
A --> aB
Convert to A --> Ba

Source Code:

```
def main():
  terminals = set()
  non_terminals  = set()
  right_linear_grammar = []

  print("Enter the right linear grammar, separated by '-' :   e.g. S - aB")
  print("Enter q to Quit")
  while True:
    rule = input().strip()
    if rule == 'q':
      break
    right_linear_grammar.append(rule)
    print('\n The Right Linear Grammar is: ')
    for rule in right_linear_grammar:
      print(rule)
    print('\n Terminals are: ')
    for rule in right_linear_grammar:
      for char in rule[2:]:
        if char.islower():
          terminals.add(char)
    print(" ".join(terminals))
    print("\n Non-Terminals are: ")
    for rule in right_linear_grammar:
      for char in rule[2:]:
        if char.isupper():
          non_terminals.add(char)
    print(" ".join(non_terminals))
    print("\n The Left Linear Grammar is: ")
```

```
    for rule in right_linear_grammar:
      print(rule)
      left_side = rule[0]
      right_side = rule[2:]
      if left_side.isupper() and right_side[-1].isupper():
        reversed_right = right_side[:-1][::-1]
        print(f"{reversed_right[0]}'-{left_side}'{reversed_right[1:]}{right_side[-1]}")
      else:
        print(f"Z'-{left_side}'{right_side}")
if __name__ == "__main__":
  main()
```

Output:



Practical 02: NFA to DFA Transitions

Aim - Understanding and Demonstrating NFA to DFA Transitions using Python.

Source Code:

```
def epsilon_closure(states, transitions):
  stack = list(states)
  closure = set(states)

  while stack:
    state = stack.pop()
    for next_state in transitions[state].get('',[]):
      if next_state not in closure:
        closure.add(next_state)
        stack.append(next_state)

  return closure

def nfa_to_dfa(nfa_states, alphabet, nfa_transitions, start_state, accept_states):
    alphabet = [a for a in alphabet if a != '']
    dfa_states = []
    dfa_transitions = {}
    dfa_start = frozenset(epsilon_closure({start_state}, nfa_transitions))
    unmarked_states = [dfa_start]
    dfa_states.append(dfa_start)
    dfa_accept_states = set()

    while unmarked_states:
        current = unmarked_states.pop()
        dfa_transitions[current] = {}

        for symbol in alphabet:
            next_states = set()
            for nfa_state in current:
                next_states.update(nfa_transitions.get(nfa_state,{}).get(symbol,[]))

                # Take Epsilon Closure of the result
                closure = epsilon_closure(next_states, nfa_transitions)
```

```
            closure = frozenset(closure)

            if closure not in dfa_states:
                dfa_states.append(closure)
                unmarked_states.append(closure)

            dfa_transitions[current][symbol] = closure

        for dfa_state in dfa_states:
            if any(state in accept_states for state in dfa_state):
                dfa_accept_states.add(dfa_state)

    return dfa_states, alphabet, dfa_transitions, dfa_start, dfa_accept_states

def print_dfa(dfa_states, alphabet, dfa_transitions, start_state, accept_states):
  print('\nDFA States:')
  for state in dfa_states:
    print(set(state))

  print('\nStart State')
  print(set(start_state))

  print('\nAccept States:')
  for state in accept_states:
    print(set(state))

  print('\nTransitions:')
  for state in dfa_transitions:
    for symbol in dfa_transitions[state]:
      print(f'{set(state)} -- {symbol} --> {set(dfa_transitions[state][symbol])}')


nfa_states = {'q0', 'q1','q2'}
alphabet = ['0', '1']
nfa_transitions = {
    'q0' : {'0' : {'q0', 'q1'},'1': {'q2'}},
    'q1' : {'1' : {'q2'}},
    'q2' : {}
}
```

```
start_state = 'q0'
accept_states = {'q2'}

dfa_states, dfa_alphabet, dfa_transitions, dfa_start, dfa_accept_states =
nfa_to_dfa(nfa_states, alphabet, nfa_transitions, start_state, accept_states)

print_dfa(dfa_states, dfa_alphabet, dfa_transitions, dfa_start, dfa_accept_states)
```

Output:

```
...
    DFA States:
    {'q0'}
    {'q1', 'q0'}
    {'q2'}
    set()

    Start State
    {'q0'}

    Accept States:
    {'q2'}

    Transitions:
    {'q0'} -- 0 --> {'q1', 'q0'}
    {'q0'} -- 1 --> {'q2'}
    {'q2'} -- 0 --> set()
    {'q2'} -- 1 --> set()
    {'q1', 'q0'} -- 0 --> {'q1', 'q0'}
    {'q1', 'q0'} -- 1 --> {'q2'}
```

Practical 03: Lexical Analysis

Aim - Demonstrating and understanding Lexical Analysis to split your program into tokens using python.

Source Code:

```
# Defining Keywords
import re

KEYWORDS = ("if", "else", "while", "return", "int", "float", "char", "for","elif")
OPERATORS = ("+", "-" , "*" , "/" , "==", "!==", "<", ">", "<=", ">=")
DELIMITERS = (";", ",", ".", "(", ")", "{", "}", "[", "]")
MAX_IDENTIFIER_LENGTH = 30

def remove_comments(code):
  code = re.sub(r"//.*", "", code)
  code = re.sub(r"/\*.*?\*/", "", code, flags = re.DOTALL)
  return code

def tokenize(code):
  code = remove_comments(code)
  code = re.sub(r"[ \t]+", " ", code)
  code = re.sub(r"\n+", "\n", code)

  tokens = re.findall(r"[a-zA-Z_]\w*|\d+|==|!==|<=|>=|[{}();,=<>+\-*/]", code)

  for token in tokens:
    if token in KEYWORDS:
      print(f"{token} -> Keyword")
    elif token in OPERATORS:
      print(f"{token} -> Operator")
    elif token in DELIMITERS:
      print(f"{token} -> Delimiter")
    elif re.match(r"^\d+$", token):
      print(f"{token} -> Number")
    elif re.match(r"^[a-zA-Z_]\w*$", token):
      if len(token) > MAX_IDENTIFIER_LENGTH:
        print(f"{token} -> Identifier (too long)")
```

```
    else:
        print(f"{token} -> Identifier")
    else:
        print(f"{token} -> Unknown Error")
```

Output:

```
int -> Keyword
main -> Identifier
( -> Delimiter
) -> Delimiter
{ -> Delimiter
int -> Keyword
a -> Identifier
= -> Unknown Error
10 -> Number
; -> Delimiter
int -> Keyword
b -> Identifier
= -> Unknown Error
20 -> Number
; -> Delimiter
int -> Keyword
c -> Identifier
= -> Unknown Error
a -> Identifier
+ -> Operator
b -> Identifier
; -> Delimiter
return -> Keyword
0 -> Number
; -> Delimiter
} -> Delimiter
```

Practical 04: Comments Identification

Aim - Identifying comments in a given program using Python.

Source Code:

```
import re
def is_comment_line(line):
  line = line.strip()
  if line.startswith("//") or line.startswith("#"):
    return True
  elif line.startswith("/*") or line.endswith("*/"):
    return True
  else:
    return False
lines =[
    "//This is a single line comment",
    "# This is a comment in python",
    " /* Multi Line Comment */",
    " int x = 0; // This is a variable declaration",
    "return 0;",
    "/* Comment with leading space",
    "code line"
]
for line in lines:
  if is_comment_line(line):
    print(f"{line} -> Comment")
  else:
    print(f"{line} -> Not a comment")
```

Output:

```
//This is a single line comment -> Comment
# This is a comment in python -> Comment
 /* Multi Line Comment */ -> Comment
 int x = 0; // This is a variable declaration -> Not a comment
return 0; -> Not a comment
/* Comment with leading space -> Comment
code line -> Not a comment
```

Practical 05: String Recognization

Aim - Identifying strings using re module in Python.

Source Code:

```
import re
def recognize_string(s):
    if re.fullmatch(r'a',s):
        return "'a' pattern matched"
    elif re.fullmatch(r'a*b+',s):
        return "'a*b+' pattern matched"
    elif re.fullmatch(r'abb',s):
        return "'abb' pattern matched"
    else:
        return "No pattern matched"
test_strings = [
    "a", "b", "abb", "aab", "aaab", "abc", "aaaaba", "aaaaaab", "ababaa", "aaa",
"bb"
]
for s in test_strings:
    result = recognize_string(s)
    print(f"'{s}' -> {result}")
```

Output:

```
'a' -> 'a' pattern matched
'b' -> 'a*b+' pattern matched
'abb' -> 'a*b+' pattern matched
'aab' -> 'a*b+' pattern matched
'aaab' -> 'a*b+' pattern matched
'abc' -> No pattern matched
'aaaaba' -> No pattern matched
'aaaaaab' -> 'a*b+' pattern matched
'ababaa' -> No pattern matched
'aaa' -> No pattern matched
'bb' -> 'a*b+' pattern matched
```

Practical 06: Symbol Table

Aim - Demonstrating and Implementing Symbol Table using Python.

Source Code:

```
class SymbolTable:
    def __init__(self):
        # Initialiazing an empty symbol table
        self.table = {}

    def insert(self, name, attributes):
        self.table[name] = attributes
        print(f"Symbol '{name}' inserted/updated with attributes: {attributes}")

    def lookup(self, name):
        if name in self.table:
            print(f"Symbol '{name}' found. Attributes: {self.table[name]}")
            return self.table[name]
        else:
            print(f"Symbol '{name}' not found in the symbol table.")
            return None

    def delete(self,name):
        if name in self.table:
            del self.table[name]
            print(f"Symbol '{name}' deleted successfully!")
            return True
        else:
            print(f"Symbol '{name}' not found.")
            return False

    def display(self):
        if not self.table:
            print('Symbol Table is empty.')
            return
        print("\n_____ Symbol Table Contents _____\n")
        for name, attributes in self.table.items():
            print(f"Name: {name} , Attributes: {attributes}")
```

```
        print('_____\n')
if __name__=='__main__':
    st = SymbolTable()

    # Inserting
    st.insert('x', {'type': 'int', 'scope': 'global', 'value': 10})
    st.insert('my_function', {'type': 'function', 'parameters': 2 , 'return_type': 'void'})
    st.insert('y', {'type': 'float', 'scope': 'local', 'value': 3.14})

    # Display Table
    st.display()

    # Lookup Symbols
    st.lookup('x')
    st.lookup('y')
    st.lookup('my_function')
    st.lookup('non_existent_var')

    # Update/Modify a symbol
    st.insert('x', {'type': 'int', 'scope':'global', 'value': 20, 'is_const': True})
    st.display()

    # Delete a Symbol
    st.delete('x')
    st.display()

    # Attempting to delete non-existent symbol
    st.delete('non_existent_var')
```

Output:

```
Symbol 'x' inserted/updated with attributes: {'type': 'int', 'scope': 'global', 'value': 10}
Symbol 'my_function' inserted/updated with attributes: {'type': 'function', 'parameters': 2, 'return_type': 'void'}
Symbol 'y' inserted/updated with attributes: {'type': 'float', 'scope': 'local', 'value': 3.14}

__________ Symbol Table Contents __________

Name: x , Attributes: {'type': 'int', 'scope': 'global', 'value': 10}
Name: my_function , Attributes: {'type': 'function', 'parameters': 2, 'return_type': 'void'}
Name: y , Attributes: {'type': 'float', 'scope': 'local', 'value': 3.14}
__________________________________________

Symbol 'x' found. Attributes: {'type': 'int', 'scope': 'global', 'value': 10}
Symbol 'y' found. Attributes: {'type': 'float', 'scope': 'local', 'value': 3.14}
Symbol 'my_function' found. Attributes: {'type': 'function', 'parameters': 2, 'return_type': 'void'}
Symbol 'non_existent_var' not found in the symbol table.
Symbol 'x' inserted/updated with attributes: {'type': 'int', 'scope': 'global', 'value': 20, 'is_const': True}

__________ Symbol Table Contents __________

Name: x , Attributes: {'type': 'int', 'scope': 'global', 'value': 20, 'is_const': True}
Name: my_function , Attributes: {'type': 'function', 'parameters': 2, 'return_type': 'void'}
Name: y , Attributes: {'type': 'float', 'scope': 'local', 'value': 3.14}
__________________________________________

Symbol 'x' deleted successfully!

__________ Symbol Table Contents __________

Name: my_function , Attributes: {'type': 'function', 'parameters': 2, 'return_type': 'void'}
Name: y , Attributes: {'type': 'float', 'scope': 'local', 'value': 3.14}
__________________________________________

Symbol 'non_existent_var' not found.
```

Practical 07: First and Follow Sets

Aim - Demonstrating and implementing First and Follow Sets in Python.

Source Code:

```
from collections import defaultdict
def compute_first(grammar):
    first = defaultdict(set)
    changed = True

    #Initializing first sets for terminals
    for non_terminal, productions in grammar.items():
        for production in productions:
            if not production:
                first[non_terminal].add('ε')
            elif not production[0].isupper():
                first[production[0]].add(production[0])

    while changed:
        changed = False
        for non_terminal, productions in grammar.items():
            for production in productions:
                current_first_size = len(first[non_terminal])

                # Rule 1: X -> a... (a is a terminal)
                if production and not production[0].isupper():
                    first[non_terminal].add(production[0])

                # Rule 2: X -> Y1 Y2 ... Yn
                else:
                    can_derive_eplison = True
                    for symbol in production:
                        if symbol not in first:
                            first[symbol].add(symbol)

                        first[non_terminal].update(first[symbol] - {'ε'})
                        if 'ε' not in first[symbol]:
                            can_derive_eplison = False
```

```
                    break
                if can_derive_eplison and production:
                    first[non_terminal].add('ε')

            if len(first[non_terminal]) != current_first_size:
                changed = True

    return first

def compute_follow(grammar,start_symbol,first_sets):
  follow=defaultdict(set)
  follow[start_symbol].add('$') #Rule 1: add $ to follow of start symbol
  changed=True

  while changed:
    changed=False
    for non_terminal, productions in grammar.items():
      for production in productions:
        for i , symbol in enumerate(production):
          if symbol.isupper():
            curren_follow_size=len(follow[symbol])

            if i + 1 < len(production):
              next_symbol=production[i+1]
              follow[symbol].update(first_sets[next_symbol]-{'ε'})
              if 'ε' in first_sets[next_symbol]:
                follow[symbol].update(follow[non_terminal])
            elif i + 1 == len(production) and symbol != non_terminal:
              follow[symbol].update(follow[non_terminal])
            if len(follow[symbol]) != curren_follow_size:
              changed=True

  return follow
if __name__ == "__main__":
  grammar = {
      'E':['TE\' '],
      'E\'':['+TE\' ','ε'],
      'T':['FT\' '],
      'T\'':['*FT\' ','ε'],
```

```
    'F':['(E)','id']
}

start_state='E'
first_set= compute_first(grammar)

print("FIRST sets: ")
for nt, s in first_set.items():
  print(f"FIRST({nt}) = {s}")

follow_set = compute_follow(grammar, start_state, first_set)

print("\nFOLLOW sets: ")
for nt, s in follow_set.items():
  print(f"FOLLOW({nt}) = {s}")
```

Output:

```
FIRST sets:
FIRST(+) = {'+'}
FIRST(ε) = {'ε'}
FIRST(*) = {'*'}
FIRST(() = {'('}
FIRST(i) = {'i'}
FIRST(E) = {'T', 'F', '(', 'i'}
FIRST(T) = {'T', 'F', '(', 'i'}
FIRST(E') = {'+', 'ε'}
FIRST(F) = {'(', 'F', 'i'}
FIRST(T') = {'*', 'ε'}

FOLLOW sets:
FOLLOW(E) = {'$'}
FOLLOW(T) = {'T', 'F', 'i', '('}
FOLLOW(F) = {'T', 'F', 'i', '('}
```

Practical 08: LL1

Aim - Understanding and Implementing LL1 Parser in Python.

Source Code:

```
#ll(1)
class Production:
    def __init__(self, head, body):
        self.head = head
        self.body = body

    def __repr__(self):
        return f"{self.head} -> {' '.join(self.body)}"

class LL1Parser:
    def __init__(self, grammar_rules, start_symbol):
        self.grammar = grammar_rules
        self.start_symbol = start_symbol
        self.non_terminals = set(p.head for p in self.grammar)
        self.terminals = self.get_terminals()
        self.first_sets = self.compute_first_sets()
        self.follow_sets = self.compute_follow_sets()
        self.parsing_table = self.build_parsing_table()

    def get_terminals(self):
        terminals = set()
        for prod in self.grammar:
            for symbol in prod.body:
                if symbol not in self.non_terminals and symbol != 'epsilon':
                    terminals.add(symbol)
        terminals.add('$')
        return sorted(list(terminals))

    def compute_first_sets(self):
        first_sets = {nt: set() for nt in self.non_terminals}
        first_sets['epsilon'] = {'epsilon'}
        while True:
            updated = False
```

```
        for prod in self.grammar:
            head = prod.head
            body = prod.body
            if body[0] in self.terminals:
                if body[0] not in first_sets[head]:
                    first_sets[head].add(body[0])
                    updated = True
            elif body[0] == 'epsilon':
                if 'epsilon' not in first_sets[head]:
                    first_sets[head].add('epsilon')
                    updated = True
            else:
                for symbol in body:
                    before_size = len(first_sets[head])
                    first_sets[head].update(first_sets[symbol] - {'epsilon'})
                    if len(first_sets[head]) > before_size:
                        updated = True
                    if 'epsilon' not in first_sets[symbol]:
                        break
                else:
                    if 'epsilon' not in first_sets[head]:
                        first_sets[head].add('epsilon')
                        updated = True
        if not updated:
            break
    return first_sets

def compute_follow_sets(self):
    follow_sets = {nt: set() for nt in self.non_terminals}
    follow_sets[self.start_symbol].add('$')
    while True:
        updated = False
        for prod in self.grammar:
            head = prod.head
            body = prod.body
            for i, symbol in enumerate(body):
                if symbol in self.non_terminals:
                    remaining = body[i+1:]
                    if remaining:
```

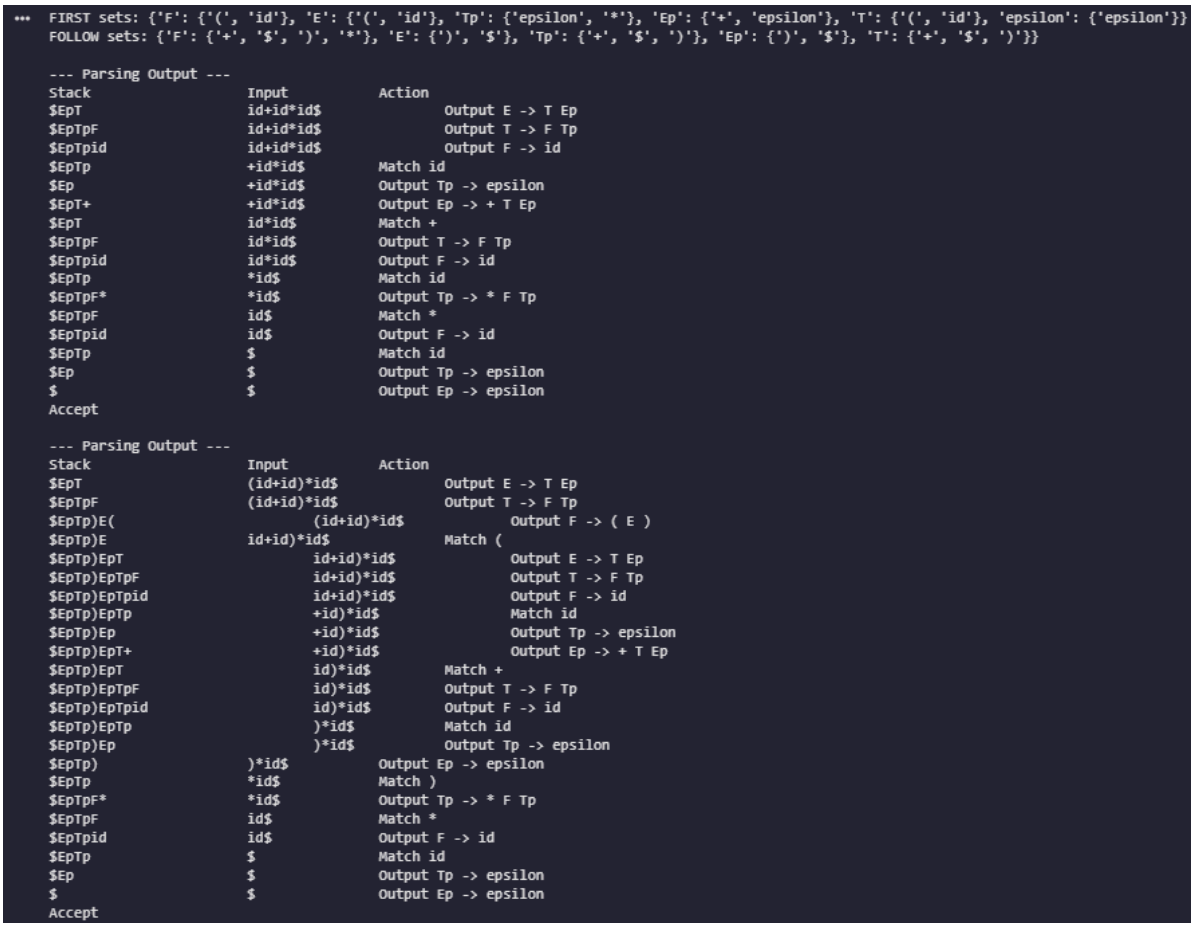
```
                    first_of_remaining = self.get_first_of_string(remaining)
                    before_size = len(follow_sets[symbol])
                    follow_sets[symbol].update(first_of_remaining - {'epsilon'})
                    if len(follow_sets[symbol]) > before_size:
                        updated = True
                if not remaining or 'epsilon' in self.get_first_of_string(remaining):
                    before_size = len(follow_sets[symbol])
                    follow_sets[symbol].update(follow_sets[head])
                    if len(follow_sets[symbol]) > before_size:
                        updated = True
        if not updated:
            break
    return follow_sets

def get_first_of_string(self, symbol_list):
    first_of_str = set()
    for symbol in symbol_list:
        if symbol in self.terminals:
            first_of_str.add(symbol)
            break
        else:
            first_of_str.update(self.first_sets[symbol] - {'epsilon'})
            if 'epsilon' not in self.first_sets[symbol]:
                break
    else:
        first_of_str.add('epsilon')
    return first_of_str

def build_parsing_table(self):
    table = {}
    for nt in self.non_terminals:
        for t in self.terminals:
            table[(nt, t)] = None

    for prod in self.grammar:
        head = prod.head
        body = prod.body
        first_of_body = self.get_first_of_string(body)
        for terminal in first_of_body:
```

```
            if terminal != 'epsilon':
                table[(head, terminal)] = prod
        if 'epsilon' in first_of_body:
            for terminal in self.follow_sets[head]:
                table[(head, terminal)] = prod
    return table

  def parse(self, input_tokens):
    input_tokens = input_tokens + ['$']
    stack = ['$']
    stack.append(self.start_symbol)

    print("\n--- Parsing Output ---")
    print("Stack\t\t\tInput\t\tAction")

    while stack:
        stack_top = stack[-1]
        input_token = input_tokens[0]

        if stack_top == input_token:
            if stack_top == '$':
                print("Accept")
                return True
            else:
                stack.pop()
                input_tokens.pop(0)
                print(f"{''.join(stack)}\t\t\t{''.join(input_tokens)}\t\tMatch
{input_token}")

        elif stack_top in self.non_terminals:
            production = self.parsing_table.get((stack_top, input_token))
            if production:
                stack.pop()
                if production.body != ['epsilon']:
                    for sym in reversed(production.body):
                        stack.append(sym)
                print(f"{''.join(stack)}\t\t\t{''.join(input_tokens)}\t\tOutput
{production}")
            else:
```

```
                print(f"Error: No production for ({stack_top}, {input_token})")
                return False
            else:
                print(f"Error: Unexpected symbol {stack_top}")
                return False
        return False

def tokenize(input_string):
    tokens = []
    i = 0
    while i < len(input_string):
        if input_string[i].isspace():
            i += 1
            continue
        if input_string[i].isalpha():
            id_str = ''
            while i < len(input_string) and input_string[i].isalnum():
                id_str += input_string[i]
                i += 1
            tokens.append(id_str)
        else:
            tokens.append(input_string[i])
            i += 1
    return tokens

if __name__ == "__main__":
    grammar_rules = [
        Production('E', ['T', 'Ep']),
        Production('Ep', ['+', 'T', 'Ep']),
        Production('Ep', ['epsilon']),
        Production('T', ['F', 'Tp']),
        Production('Tp', ['*', 'F', 'Tp']),
        Production('Tp', ['epsilon']),
        Production('F', ['(', 'E', ')']),
        Production('F', ['id'])
    ]
    start_symbol = 'E'

    parser = LL1Parser(grammar_rules, start_symbol)
```

```
print("FIRST sets:", parser.first_sets)
print("FOLLOW sets:", parser.follow_sets)

input_string1 = "id+id*id"
tokens1 = tokenize(input_string1)
parser.parse(tokens1)

input_string2 = "(id+id)*id"
tokens2 = tokenize(input_string2)
parser.parse(tokens2)
```

Output:



```
FIRST sets: {'F': {'(', 'id'}, 'E': {'(', 'id'}, 'Tp': {'epsilon', '*'}, 'Ep': {'+', 'epsilon'}, 'T': {'(', 'id'}, 'epsilon': {'epsilon'}}
FOLLOW sets: {'F': {'+', '$', ')', '*'}, 'E': {')', '$'}, 'Tp': {'+', '$', ')'}, 'Ep': {')', '$'}, 'T': {'+', '$', ')'}}

--- Parsing Output ---
Stack                   Input           Action
$EpT                    id+id*id$               Output E -> T Ep
$EpTpF                  id+id*id$               Output T -> F Tp
$EpTpid                 id+id*id$               Output F -> id
$EpTp                   +id*id$         Match id
$Ep                     +id*id$         Output Tp -> epsilon
$EpT+                   +id*id$         Output Ep -> + T Ep
$EpT                    id*id$          Match +
$EpTpF                  id*id$          Output T -> F Tp
$EpTpid                 id*id$          Output F -> id
$EpTp                   *id$            Match id
$EpTpF*                 *id$            Output Tp -> * F Tp
$EpTpF                  id$             Match *
$EpTpid                 id$             Output F -> id
$EpTp                   $               Match id
$Ep                     $               Output Tp -> epsilon
$                       $               Output Ep -> epsilon
Accept

--- Parsing Output ---
Stack                   Input           Action
$EpT                    (id+id)*id$             Output E -> T Ep
$EpTpF                  (id+id)*id$             Output T -> F Tp
$EpTp)E(                        (id+id)*id$             Output F -> ( E )
$EpTp)E                 id+id)*id$              Match (
$EpTp)EpT                       id+id)*id$              Output E -> T Ep
$EpTp)EpTpF                     id+id)*id$              Output T -> F Tp
$EpTp)EpTpid                    id+id)*id$              Output F -> id
$EpTp)EpTp                      +id)*id$                Match id
$EpTp)Ep                        +id)*id$                Output Tp -> epsilon
$EpTp)EpT+                      +id)*id$                Output Ep -> + T Ep
$EpTp)EpT                       id)*id$         Match +
$EpTp)EpTpF                     id)*id$         Output T -> F Tp
$EpTp)EpTpid                    id)*id$         Output F -> id
$EpTp)EpTp                      )*id$           Match id
$EpTp)Ep                        )*id$           Output Tp -> epsilon
$EpTp)                  )*id$           Output Ep -> epsilon
$EpTp                   *id$            Match )
$EpTpF*                 *id$            Output Tp -> * F Tp
$EpTpF                  id$             Match *
$EpTpid                 id$             Output F -> id
$EpTp                   $               Match id
$Ep                     $               Output Tp -> epsilon
$                       $               Output Ep -> epsilon
Accept
```

FIRST sets: {'F': {'(', 'id'}, 'E': {'(', 'id'}, 'Tp': {'epsilon', '*'}, 'Ep': {'+', 'epsilon'}, 'T': {'(', 'id'}, 'epsilon': {'epsilon'}}
FOLLOW sets: {'F': {'+', '$', ')', '*'}, 'E': {')', '$'}, 'Tp': {'+', '$', ')'}, 'Ep': {')', '$'}, 'T': {'+', '$', ')'}}

--- Parsing Output ---

Stack	Input	Action
$EpT	id+id*id$	Output E -> T Ep
$EpTpF	id+id*id$	Output T -> F Tp
$EpTpid	id+id*id$	Output F -> id
$EpTp	+id*id$	Match id
$Ep	+id*id$	Output Tp -> epsilon
$EpT+	+id*id$	Output Ep -> + T Ep
$EpT	id*id$	Match +
$EpTpF	id*id$	Output T -> F Tp
$EpTpid	id*id$	Output F -> id

```
$EpTp                   *id$            Match id
$EpTpF*                 *id$            Output Tp -> * F Tp
$EpTpF                  id$             Match *
$EpTpid                 id$             Output F -> id
$EpTp                   $               Match id
$Ep             $               Output Tp -> epsilon
$               $               Output Ep -> epsilon
Accept

--- Parsing Output ---
Stack           Input           Action
$EpT            (id+id)*id$             Output E -> T Ep
$EpTpF                  (id+id)*id$             Output T -> F Tp
$EpTp)E(                (id+id)*id$             Output F -> ( E )
$EpTp)E                 id+id)*id$              Match (
$EpTp)EpT               id+id)*id$              Output E -> T Ep
$EpTp)EpTpF                     id+id)*id$              Output T -> F Tp
$EpTp)EpTpid                    id+id)*id$              Output F -> id
$EpTp)EpTp                      +id)*id$                Match id
$EpTp)Ep                +id)*id$                Output Tp -> epsilon
$EpTp)EpT+                      +id)*id$                Output Ep -> + T Ep
$EpTp)EpT               id)*id$                 Match +
$EpTp)EpTpF                     id)*id$                 Output T -> F Tp
$EpTp)EpTpid                    id)*id$                 Output F -> id
$EpTp)EpTp                      )*id$           Match id
$EpTp)Ep                )*id$           Output Tp -> epsilon
$EpTp)                  )*id$           Output Ep -> epsilon
$EpTp                   *id$            Match )
$EpTpF*                 *id$            Output Tp -> * F Tp
$EpTpF                  id$             Match *
$EpTpid                 id$             Output F -> id
$EpTp                   $               Match id
$Ep             $               Output Tp -> epsilon
$               $               Output Ep -> epsilon
Accept
```

Practical 09: LR0

Aim - Understanding and Implementing LR0 Parser in Python.

Source Code:

```
class Production:
    def __init__(self, head, body):
        self.head = head
        self.body = body

# Represents an LR(0) item
class Item:
    def __init__(self, production, dot_position):
        self.production = production
        self.dot_position = dot_position

    def __eq__(self, other):
        return (self.production.head == other.production.head and
        self.production.body == other.production.body and
        self.dot_position == other.dot_position)

    def __hash__(self):
        return hash((self.production.head, tuple(self.production.body),
self.dot_position))

# LR(0) Parser Class
class LR0Parser:
    def __init__(self, grammar):
        self.grammar = grammar
        self.augmented_grammar = self.augment_grammar()
        self.terminals = self.get_terminals()
        self.non_terminals = self.get_non_terminals()
        self.states = []
        self.action_table = {}
        self.goto_table = {}
        self.production_map = self.create_production_map()

    def augment_grammar(self):
```

```
        start_symbol = self.grammar[0].head
        augmented_production = Production("S'", [start_symbol])
        return [augmented_production] + self.grammar

    def get_terminals(self):
        terminals = set()
        for prod in self.grammar:
            for symbol in prod.body:
                if not symbol.isupper() and symbol not in ["ϵ", "$"]:
                    terminals.add(symbol)
        terminals.add('$')
        return sorted(list(terminals))

    def get_non_terminals(self):
        non_terminals = set()
        for prod in self.grammar:
            non_terminals.add(prod.head)
        return sorted(list(non_terminals))

    def create_production_map(self):
        return {f"R{i}": prod for i, prod in enumerate(self.grammar)}

    def closure(self, item_set):
        closure_set = set(item_set)
        while True:
            new_items_added = False
            items_to_add = set()
            for item in closure_set:
                if item.dot_position < len(item.production.body):
                    symbol_after_dot = item.production.body[item.dot_position]
                    if symbol_after_dot in self.non_terminals:
                        for prod in self.augmented_grammar:
                            if prod.head == symbol_after_dot:
                                new_item = Item(prod, 0)
                                if new_item not in closure_set and new_item not in
items_to_add:
                                    items_to_add.add(new_item)
                                    new_items_added = True
            if not new_items_added:
```

```
            break
        closure_set.update(items_to_add)
    return frozenset(closure_set)


def goto(self, item_set, symbol):
    next_item_set = set()
    for item in item_set:
        if item.dot_position < len(item.production.body):
            if item.production.body[item.dot_position] == symbol:
                new_item = Item(item.production, item.dot_position + 1)
                next_item_set.add(new_item)
    return self.closure(next_item_set)

def build_parsing_table(self):
    # Initial state
    initial_item = Item(self.augmented_grammar[0], 0)
    initial_state = self.closure({initial_item})
    self.states.append(initial_state)

    queue = [initial_state]
    state_map = {initial_state: 0}

    while queue:
        current_state = queue.pop(0)
        current_state_idx = state_map[current_state]

        # Process shifts and gotos
        all_symbols = self.terminals[:-1] + self.non_terminals
        for symbol in all_symbols:
            next_state = self.goto(current_state, symbol)
            if next_state:
                if next_state not in state_map:
                    state_map[next_state] = len(self.states)
                    self.states.append(next_state)
                    queue.append(next_state)

                next_state_idx = state_map[next_state]
                if symbol in self.terminals:
```

```
                    # Shift action
                    self.action_table[(current_state_idx, symbol)] =
f"S{next_state_idx}"
                else:
                    # Goto action
                    self.goto_table[(current_state_idx, symbol)] = next_state_idx

            # Process reductions
            for item in current_state:
                if item.dot_position == len(item.production.body):
                    if item.production.head == "S'":
                        # Accept action
                        self.action_table[(current_state_idx, '$')] = "accept"
                    else:
                        # Reduce action for all terminals
                        prod_idx = self.augmented_grammar.index(item.production) - 1
                        for terminal in self.terminals:
                            if (current_state_idx, terminal) in self.action_table:
                                # Conflict detected (Shift/Reduce)
                                print(f"Warning: Shift/Reduce conflict in state
{current_state_idx} on terminal '{terminal}'")
                                # This simple parser just warns and keeps the shift action.
                            elif (current_state_idx, terminal) in self.action_table and
self.action_table[(current_state_idx, terminal)].startswith('R'):
                                # Conflict detected (Reduce/Reduce)
                                print(f"Warning: Reduce/Reduce conflict in state
{current_state_idx} on terminal '{terminal}'")
                                # This simple parser just warns.
                            else:
                                self.action_table[(current_state_idx, terminal)] = f"R{prod_idx
+ 1}"

    def parse(self, input_string):
        stack = [0]
        input_string = list(input_string) + ['$']

        print("\nParsing Table:")
        self.print_parsing_table()
```

```
        print("\nParsing Input:", "".join(input_string))
        print("Stack\t\t\tInput\t\tAction")

        while True:
            current_state = stack[-1]
            next_input = input_string[0]

            action = self.action_table.get((current_state, next_input))

            print(f"{stack}\t\t\t{input_string}\t\t{action}")

            if action is None:
                print("Error: No valid action found.")
                return False
            elif action.startswith('S'):
                # Shift
                new_state = int(action[1:])
                stack.append(next_input)
                stack.append(new_state)
                input_string.pop(0)
            elif action.startswith('R'):
                # Reduce
                prod_idx = int(action[1:])
                prod_to_reduce = self.grammar[prod_idx - 1]

                # Pop symbols from stack
                for _ in range(len(prod_to_reduce.body) * 2):
                    stack.pop()

                new_non_terminal = prod_to_reduce.head
                current_state_after_pop = stack[-1]

                # Push non-terminal and new state
                goto_state = self.goto_table.get((current_state_after_pop,
new_non_terminal))
                if goto_state is None:
                    print("Error: Invalid GOTO transition during reduction.")
                    return False
```

```
                stack.append(new_non_terminal)
                stack.append(goto_state)
            elif action == "accept":
                print("Accept: Input string successfully parsed.")
                return True
            else:
                print("Error: Unknown action.")
                return False

    def print_parsing_table(self):
        terminals_header = self.terminals
        non_terminals_header = self.non_terminals

        header = ["State"] + terminals_header + non_terminals_header

        print("\t".join(header))
        print("-" * (8 * len(header)))

        for i in range(len(self.states)):
            row = [str(i)]
            for terminal in terminals_header:
                row.append(self.action_table.get((i, terminal), ""))
            for non_terminal in non_terminals_header:
                row.append(self.goto_table.get((i, non_terminal), ""))
            print("\t".join(map(str, row)))

# Main execution
if __name__ == "__main__":
    grammar_rules = [
        Production('E', ['E', '+', 'T']),
        Production('E', ['T']),
        Production('T', ['T', '*', 'F']),
        Production('T', ['F']),
        Production('F', ['(', 'E', ')']),
        Production('F', ['id'])
    ]

    parser = LR0Parser(grammar_rules)
    parser.build_parsing_table()
```

```
input_string_to_parse = "id+id*id"
parser.parse(input_string_to_parse)

print("\n--- Another Test ---")
input_string_to_parse = "(id+id)*id"
parser.parse(input_string_to_parse)
```

Output:

```
Warning: Shift/Reduce conflict in state 4 on terminal '*'
Warning: Shift/Reduce conflict in state 9 on terminal '*'

Parsing Table:
State   $       (       )       *       +       id      E       F       T
------------------------------------------------------------------------------
0               S1                                      2       3       4
1               S1                                      5       3       4
2       accept                          S6
3       R4      R4      R4      R4      R4      R4
4       R2      R2      R2      S7      R2      R2
5                       S8              S6
6               S1                                              3       9
7               S1                                              10
8       R5      R5      R5      R5      R5      R5
9       R1      R1      R1      S7      R1      R1
10      R3      R3      R3      R3      R3      R3

Parsing Input: id+id*id$
Stack                   Input           Action
[0]                     ['i', 'd', '+', 'i', 'd', '*', 'i', 'd', '$']           None
Error: No valid action found.

--- Another Test ---

Parsing Table:
State   $       (       )       *       +       id      E       F       T
------------------------------------------------------------------------------
0               S1                                      2       3       4
1               S1                                      5       3       4
2       accept                          S6
3       R4      R4      R4      R4      R4      R4
4       R2      R2      R2      S7      R2      R2
5                       S8              S6
6               S1                                              3       9
7               S1                                              10
8       R5      R5      R5      R5      R5      R5
9       R1      R1      R1      S7      R1      R1
10      R3      R3      R3      R3      R3      R3

Parsing Input: (id+id)*id$
Stack                   Input           Action
[0]                     ['(', 'i', 'd', '+', 'i', 'd', ')', '*', 'i', 'd', '$']         S1
[0, '(', 1]                     ['i', 'd', '+', 'i', 'd', ')', '*', 'i', 'd', '$']
Error: No valid action found.
```

Practical 10: Infix to Postfix

Aim - Demonstrating and Implementing a Python Program to convert Infix to Postfix Expression

Source Code:

```
def precedence(op):
    if op == '+' or op == '-':
        return 1
    elif op == '*' or op == '/':
        return 2
    elif op == '^':
        return 3
    else:
        return 0

class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)
        print(f"Pushed to stack: {item}")

    def pop(self):
        if not self.is_empty():
            popped = self.items.pop()
            print(f"Popped from stack: {popped}")
            return popped
        return None

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        return None

    def is_empty(self):
        return len(self.items) == 0
```

```
def infix_to_postfix(expression):
    stack = Stack()
    postfix = ''

    for char in expression:
        if char.isalnum():
            postfix += char
            print(f"Added operand to postfix: {char}")
        elif char == '(':
            stack.push(char)
        elif char == ')':
            while not stack.is_empty() and stack.peek() != '(':
                postfix += stack.pop()
            stack.pop()
            print("Popped '(' from stack")
        else:
            while (not stack.is_empty() and precedence(char) <=
precedence(stack.peek())):
                postfix += stack.pop()
            stack.push(char)

    while not stack.is_empty():
        postfix += stack.pop()

    return postfix

if __name__ == "__main__":
    infix_expr = "a+b*(c^d-e)^(f+g*h)-i"
    print(f"Infix expression: {infix_expr}")
    postfix_expr = infix_to_postfix(infix_expr)
    print(f"Postfix expression: {postfix_expr}")
```

Output:

```
Infix expression: a+b*(c^d-e)^(f+g*h)-i
Added operand to postfix: a
Pushed to stack: +
Added operand to postfix: b
Pushed to stack: *
```

Pushed to stack: (
Added operand to postfix: c
Pushed to stack: ^
Added operand to postfix: d
Popped from stack: ^
Pushed to stack: -
Added operand to postfix: e
Popped from stack: -
Popped from stack: (
Popped '(' from stack
Pushed to stack: ^
Pushed to stack: (
Added operand to postfix: f
Pushed to stack: +
Added operand to postfix: g
Pushed to stack: *
Added operand to postfix: h
Popped from stack: *
Popped from stack: +
Popped from stack: (
Popped '(' from stack
Popped from stack: ^
Popped from stack: *
Popped from stack: +
Pushed to stack: -
Added operand to postfix: i
Popped from stack: -
Postfix expression: abcd^e-fgh*+^*+i-

Image of the Output:

```
Infix expression: a+b*(c^d-e)^(f+g*h)-i
Added operand to postfix: a
Pushed to stack: +
Added operand to postfix: b
Pushed to stack: *
Pushed to stack: (
Added operand to postfix: c
Pushed to stack: ^
Added operand to postfix: d
Popped from stack: ^
Pushed to stack: -
Added operand to postfix: e
Popped from stack: -
Popped from stack: (
Popped '(' from stack
Pushed to stack: ^
Pushed to stack: (
Added operand to postfix: f
Pushed to stack: +
Added operand to postfix: g
Pushed to stack: *
Added operand to postfix: h
Popped from stack: *
Popped from stack: +
Popped from stack: (
Popped '(' from stack
Popped from stack: ^
Popped from stack: *
Popped from stack: +
Pushed to stack: -
Added operand to postfix: i
Popped from stack: -
Postfix expression: abcd^e-fgh*+^*+i-
```

Practical 11: Operator Precedence

Aim - Understanding and Demonstrating Operator Precedence using Python.

Source Code:

```
from tabulate import tabulate

def get_precedence(operator):
    precedence = {
        '(': 'Highest',
        ')': 'Highest',
        '**': 'High',
        '*': 'High', '/': 'High', '//': 'High', '%': 'High',
        '+': 'Medium', '-': 'Medium',
        '<<': 'Low', '>>': 'Low',
        '&': 'Low',
        '^': 'Low',
        '|': 'Lowest'
    }
    return precedence.get(operator, 'Unknown')

def show_operator_precedence(operators):
    precedence_table = []
    for op in operators:
        precedence_table.append((op, get_precedence(op)))

    print("\nOperator Precedence Table:")
    print(tabulate(precedence_table, headers=["Operator", "Precedence"],
tablefmt="grid"))

def extract_operators(expression):
    operators = ['+', '-', '*', '/', '(', ')', '**', '%', '//', '<<', '>>', '&', '^', '|']
    used_operators = []
    for op in operators:
        if op in expression:
            used_operators.append(op)
    return used_operators
```

```
def solve_expression():
    z = 10
    a = 5
    b = 2
    d = 3
    x = 8
    y = 4

    expression = "(z + (a / b) * (b / d)) + (x / y)"
    print(f"\nSolving Expression: {expression}")

    result = eval(expression)
    print(f"Result of expression: {result}\n")
    return expression

def evaluate_expression_step_by_step():
    z = 10
    a = 5
    b = 2
    d = 3
    x = 8
    y = 4

    trace = []

    expr = f"({z} + ({a} / {b}) * ({b} / {d})) + ({x} / {y})"
    trace.append(f"Original expression: {expr}")

    a_div_b = a / b
    b_div_d = b / d
    x_div_y = x / y
    expr_step1 = f"({z} + {a_div_b} * {b_div_d}) + {x_div_y}"
    trace.append(f"After evaluating divisions: {expr_step1}")

    mult = a_div_b * b_div_d
    expr_step2 = f"({z} + {mult}) + {x_div_y}"
    trace.append(f"After evaluating multiplication: {expr_step2}")

    inside_paren = z + mult
```

```
    expr_step3 = f"{inside_paren} + {x_div_y}"
    trace.append(f"After adding inside parentheses: {expr_step3}")

    result = inside_paren + x_div_y
    trace.append(f"Final Result: {result}")

    print("\nStep-by-Step Evaluation:")
    for step in trace:
        print(step)

def main():
    expression = solve_expression()
    used_operators = extract_operators(expression)
    show_operator_precedence(used_operators)

    print("\nDetailed Precedence for Operators:")
    for op in used_operators:
        print(f"Operator: {op} | Precedence: {get_precedence(op)}")

    evaluate_expression_step_by_step()

main()
```

Output:

```
...
    Solving Expression: (z + (a / b) * (b / d)) + (x / y)
    Result of expression: 13.666666666666666


    Operator Precedence Table:
    +------------+--------------+
    | Operator   | Precedence   |
    +============+==============+
    | +          | Medium       |
    +------------+--------------+
    | *          | High         |
    +------------+--------------+
    | /          | High         |
    +------------+--------------+
    | (          | Highest      |
    +------------+--------------+
    | )          | Highest      |
    +------------+--------------+

    Detailed Precedence for Operators:
    Operator: + | Precedence: Medium
    Operator: * | Precedence: High
    Operator: / | Precedence: High
    Operator: ( | Precedence: Highest
    Operator: ) | Precedence: Highest

    Step-by-Step Evaluation:
    Original expression: (10 + (5 / 2) * (2 / 3)) + (8 / 4)
    After evaluating divisions: (10 + 2.5 * 0.6666666666666666) + 2.0
    After evaluating multiplication: (10 + 1.6666666666666665) + 2.0
    After adding inside parentheses: 11.666666666666666 + 2.0
    Final Result: 13.666666666666666
```

Practical 12: Syntax Trees

Aim - Understanding and Implementing Syntax Trees in Python.

Source Code:

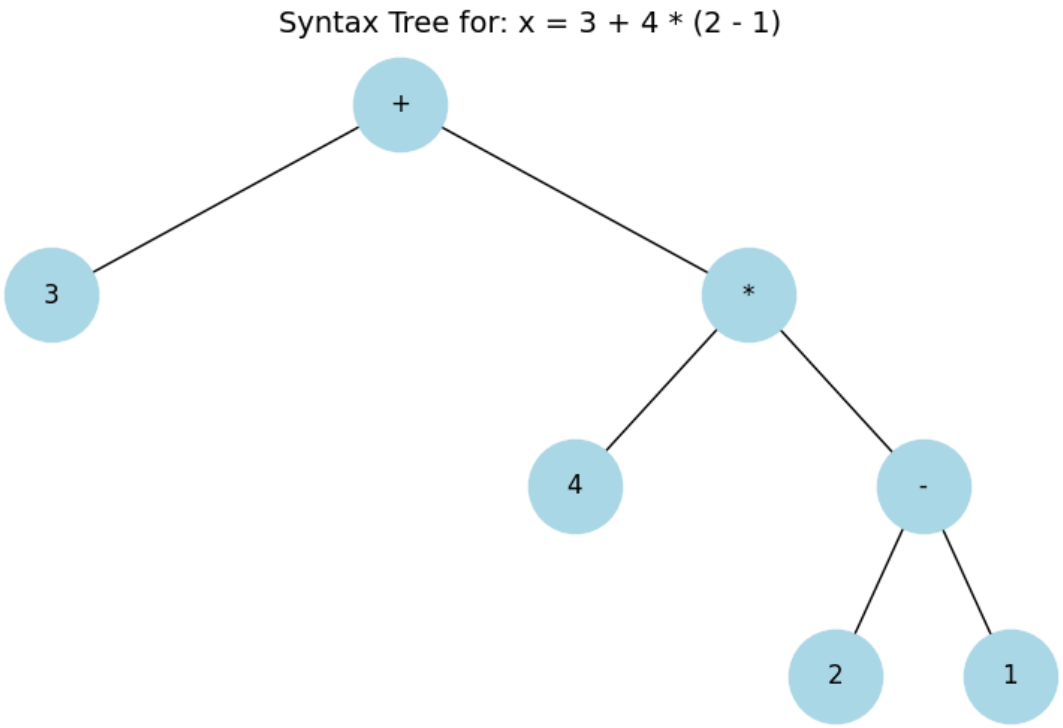
```
import matplotlib.pyplot as plt
import networkx as nx

class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def is_operator(c):
    return c in "+-*/"

def build_syntax_tree(expression):
    tokens = list(expression.replace(" ", ""))
    output = []
    operators = []

    def precedence(op):
        if op in '+-':
            return 1
        if op in '*/':
            return 2
        return 0

    def apply_operator():
        op = operators.pop()
        right = output.pop()
        left = output.pop()
        node = Node(op)
        node.left = left
        node.right = right
        output.append(node)
```

```
    i = 0
    while i < len(tokens):
        token = tokens[i]
        if token.isdigit():
            num = token
            while i + 1 < len(tokens) and tokens[i + 1].isdigit():
                i += 1
                num += tokens[i]
            output.append(Node(num))
        elif token == '(':
            operators.append(token)
        elif token == ')':
            while operators and operators[-1] != '(':
                apply_operator()
            operators.pop()
        elif is_operator(token):
            while (operators and operators[-1] != '(' and
                   precedence(operators[-1]) >= precedence(token)):
                apply_operator()
            operators.append(token)
        i += 1

    while operators:
        apply_operator()

    return output[0]

def visualize_tree(root, expr_label):
    G = nx.DiGraph()
    def add_nodes_edges(node):
        if node:
            G.add_node(node.value + str(id(node)), label=node.value)
            if node.left:
                G.add_node(node.left.value + str(id(node.left)), label=node.left.value)
                G.add_edge(node.value + str(id(node)), node.left.value +
str(id(node.left)))
                add_nodes_edges(node.left)
            if node.right:
```

```
            G.add_node(node.right.value + str(id(node.right)),
label=node.right.value)
            G.add_edge(node.value + str(id(node)), node.right.value +
str(id(node.right)))
            add_nodes_edges(node.right)

    add_nodes_edges(root)

    pos = hierarchy_pos(G, root.value + str(id(root)))
    labels = nx.get_node_attributes(G, 'label')

    plt.figure(figsize=(10,6))
    plt.title(f"Syntax Tree for: {expr_label}", fontsize=14)
    nx.draw(G, pos=pos, labels=labels, with_labels=True, arrows=False,
          node_size=2000, node_color='lightblue', font_size=12)
    plt.show()

def hierarchy_pos(G, root, width=1., vert_gap=0.2, vert_loc=0, xcenter=0.5):
    pos = {}
    def _hierarchy_pos(G, root, left, right, vert_loc):
        pos[root] = ((left + right) / 2, vert_loc)
        neighbors = list(G.successors(root))
        if neighbors:
            dx = (right - left) / len(neighbors)
            nextx = left
            for neighbor in neighbors:
                _hierarchy_pos(G, neighbor, nextx, nextx + dx, vert_loc - vert_gap)
                nextx += dx
    _hierarchy_pos(G, root, 0, width, vert_loc)
    return pos

inp = input("Enter expression (e.g. x=3 + 4 * (2 - 1)): ").strip()
if '=' in inp:
    var, expr = map(str.strip, inp.split('=', 1))
else:
    var, expr = "", inp

tree = build_syntax_tree(expr)
visualize_tree(tree, inp)
```

```
if var:
    print(f"{var} = {expr}")
else:
    print(expr)
```

Output:



Practical 13: Directed Acyclic Graph (DAG)

Aim - Generation of three address code and DAG for the given arithmetic expression.

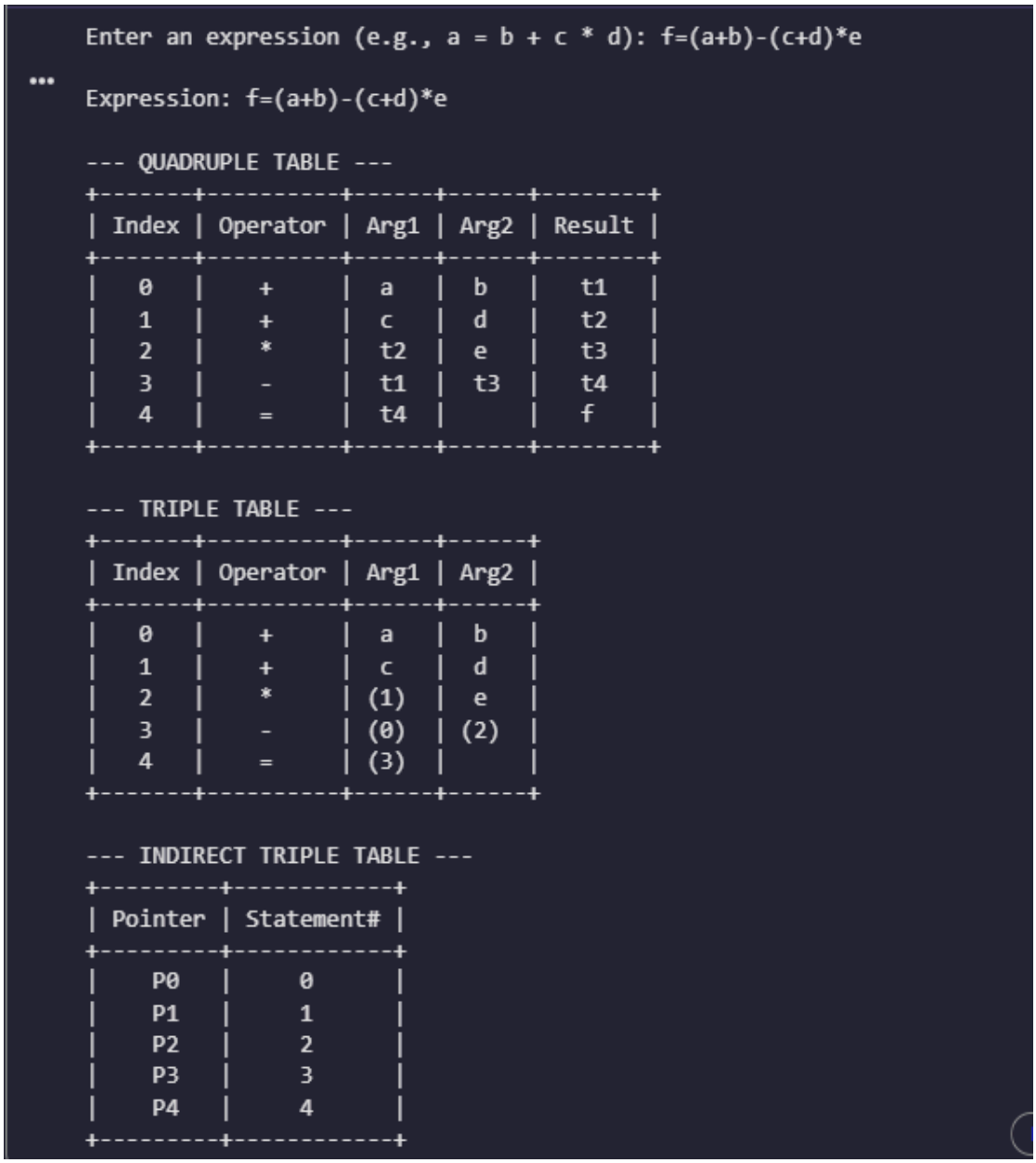
Source Code:

```
import re
from prettytable import PrettyTable

def generate_TAC(expr):
    expr = expr.replace(" ", "")
    temp_count = 1
    stack = []
    tac = []
    precedence = {'^': 3, '*': 2, '/': 2, '+': 1, '-': 1}
    tokens = re.findall(r'[a-zA-Z]+|\d+|[\+\-\*\/\^\=\(\)]', expr)

    if '=' in tokens:
        lhs = tokens[0]
        tokens = tokens[2:]
    else:
        lhs = None

    def get_temp():
        nonlocal temp_count
        t = f"t{temp_count}"
        temp_count += 1
        return t

    def process(op, op2, op1):
        t = get_temp()
        tac.append((op, op1, op2, t))
        stack.append(t)

    def infix_to_TAC(tokens):
        op_stack = []
        val_stack = []
        for token in tokens:
```

```
            if re.match(r'[a-zA-Z0-9]+', token):
                val_stack.append(token)
            elif token == '(':
                op_stack.append(token)
            elif token == ')':
                while op_stack and op_stack[-1] != '(':
                    op = op_stack.pop()
                    op2 = val_stack.pop()
                    op1 = val_stack.pop()
                    process(op, op2, op1)
                    val_stack.append(stack.pop())
                op_stack.pop()
            else:
                while (op_stack and op_stack[-1] != '(' and
                       precedence[op_stack[-1]] >= precedence[token]):
                    op = op_stack.pop()
                    op2 = val_stack.pop()
                    op1 = val_stack.pop()
                    process(op, op2, op1)
                    val_stack.append(stack.pop())
                op_stack.append(token)

        while op_stack:
            op = op_stack.pop()
            op2 = val_stack.pop()
            op1 = val_stack.pop()
            process(op, op2, op1)
            val_stack.append(stack.pop())

        return val_stack[0]

    final_temp = infix_to_TAC(tokens)
    if lhs:
        tac.append(('=', final_temp, '', lhs))

    return tac

def display_tables(expr):
    tac = generate_TAC(expr)
```

```
    quad = PrettyTable(['Index', 'Operator', 'Arg1', 'Arg2', 'Result'])
    for i, row in enumerate(tac):
        quad.add_row([i] + list(row))

    triples = PrettyTable(['Index', 'Operator', 'Arg1', 'Arg2'])
    for i, (op, arg1, arg2, res) in enumerate(tac):
        arg1_ref = f"({[r[3] for r in tac].index(arg1)})" if arg1 in [r[3] for r in tac] else arg1
        arg2_ref = f"({[r[3] for r in tac].index(arg2)})" if arg2 in [r[3] for r in tac] else arg2
        triples.add_row([i, op, arg1_ref, arg2_ref])

    indirect = PrettyTable(['Pointer', 'Statement#'])
    for i in range(len(tac)):
        indirect.add_row([f"P{i}", i])

    result = f"\nExpression: {expr}\n\n--- QUADRUPLE TABLE ---\n{quad}\n\n---
TRIPLE TABLE ---\n{triples}\n\n--- INDIRECT TRIPLE TABLE ---\n{indirect}"
    print(result)

def main():
    expr = input("Enter an expression (e.g., a = b + c * d): ")
    display_tables(expr)

if __name__ == "__main__":
    main()
```

Output:


Enter an expression (e.g., a = b + c * d): f=(a+b)-(c+d)*e
Expression: f=(a+b)-(c+d)*e
--- QUADRUPLE TABLE ---
Index	Operator	Arg1	Arg2	Result
0	+	a	b	t1
1	+	c	d	t2
2	*	t2	e	t3
3	-	t1	t3	t4
4	=	t4		f
--- TRIPLE TABLE ---
Index	Operator	Arg1	Arg2
0	+	a	b
1	+	c	d
2	*	(1)	e
3	-	(0)	(2)
4	=	(3)	
--- INDIRECT TRIPLE TABLE ---
Pointer	Statement#
P0	0
P1	1
P2	2
P3	3
P4	4

Practical 14: YACC

Aim -

Source Code:
Output: