class ExpressionConverter:

# Operator precedence

precedence = {'+': 1, '-': 1, '\*': 2, '/': 2, '^': 3}

def \_\_init\_\_(self, expression):

self.expression = expression

@staticmethod

def is\_operator(c):

return c in ['+', '-', '\*', '/', '^']

@staticmethod

def is\_operand(c):

return c.isalnum()

def infix\_to\_postfix(self):

stack = [] # Stack for operators

postfix = [] # Result

for char in self.expression:

if self.is\_operand(char): # If operand, add to postfix

postfix.append(char)

elif char == '(':

stack.append(char)

elif char == ')':

# Pop until '('

while stack and stack[-1] != '(':

postfix.append(stack.pop())

stack.pop() # Remove '('

else:

# Pop operators with higher precedence

while stack and stack[-1] != '(' and self.precedence[char] <= self.precedence.get(stack[-1], 0):

postfix.append(stack.pop())

stack.append(char)

while stack:

postfix.append(stack.pop()) # Pop remaining operators

return ''.join(postfix)

def infix\_to\_prefix(self):

# Reverse the infix expression and adjust parentheses

reversed\_expression = self.expression[::-1]

adjusted\_expression = reversed\_expression.replace('(', 'temp').replace(')', '(').replace('temp', ')')

# Convert reversed infix to postfix

postfix = ExpressionConverter(adjusted\_expression).infix\_to\_postfix()

# Reverse postfix to get prefix

return postfix[::-1]

def postfix\_to\_infix(self, postfix):

stack = []

for char in postfix:

if self.is\_operand(char):

stack.append(char)

else:

# Pop two operands and create expression

op1 = stack.pop()

op2 = stack.pop()

stack.append(f'({op2}{char}{op1})')

return stack[-1]

def prefix\_to\_infix(self, prefix):

stack = []

for char in reversed(prefix):

if self.is\_operand(char):

stack.append(char)

else:

# Pop two operands and create expression

op1 = stack.pop()

op2 = stack.pop()

stack.append(f'({op1}{char}{op2})')

return stack[-1]

# Example Usage

expression = "a+b\*(c^d-e)^(f+g\*h)-i" # Infix expression

converter = ExpressionConverter(expression)

print("Infix Expression:", expression)

postfix = converter.infix\_to\_postfix()

print("Postfix Expression:", postfix)

prefix = converter.infix\_to\_prefix()

print("Prefix Expression:", prefix)

print("Converted back to Infix from Postfix:", converter.postfix\_to\_infix(postfix))

print("Converted back to Infix from Prefix:", converter.prefix\_to\_infix(prefix))

Task:

{a + [(b \* {c + [d - e]} ) / f] } \* g