

Python CODE

```
class infix_to_prefix:
    precedence = {'^': 5, '*': 4, '/': 4, '+': 3, '-': 3, '(': 2, ')': 1}
    items = []
    size = -1

    def init(self):
        self.items = []
        self.size = -1

    def push(self, value):
        self.items.append(value)
        self.size += 1

    def pop(self):
        if self.isempty():
            return 0
        else:
            self.size -= 1
            return self.items.pop()

    def isempty(self):
        if(self.size == -1):
            return True
        else:
            return False

    def seek(self):
        if self.isempty():
            return False
        else:
            return self.items[self.size]

    def isOperand(self, i):
        if i.isalpha() or i in '1234567890':
            return True
        else:
            return False
```

```
def reverse(self, expr):
```

```
    rev = ""
```

```
    for i in expr:
```

```
        if i == '(':
```

```
            i = ')'
```

```
        elif i == ')':
```

```
            i = '('
```

```
        rev = i+rev
```

```
    return rev
```

```
def infixtoprefix(self, expr):
```

```
    prefix = ""
```

```
    print("\nPrefix expression after every iteration is:")
```

```
    for i in expr:
```

```
        if(len(expr) % 2 == 0):
```

```
            print("Incorrect infix expr")
```

```
            return False
```

```
        elif(self.isOperand(i)):
```

```
            prefix += i
```

```
        elif(i in '+-*/^'):
```

```
            while(len(self.items) and self.precedence[i] < self.precedence[self.seek()]):
```

```
                prefix += self.pop()
```

```
            self.push(i)
```

```
        elif i == '(':
```

```
            self.push(i)
```

```
        elif i == ')':
```

```
            o = self.pop()
```

```
            while o != '(':
```

```
                prefix += o
```

```
            o = self.pop()
```

```
    print(prefix)
```

```
    # end of for
```

```
    while len(self.items):
```

```
        if(self.seek() == '('):
```

```
            self.pop()
```

```
        else:
```

```
            prefix += self.pop()
```

```
            print(prefix)
```

```
    return prefix
```

```
class infix_to_postfix:
    precedence = {'^': 5, '*': 4, '/': 4, '+': 3, '-': 3, '(': 2, ')': 1}
    items = []
    size = -1

    def init(self):
        self.items = []
        self.size = -1

    def push(self, value):
        self.items.append(value)
        self.size += 1

    def pop(self):
        if self.isempty():
            return 0
        else:
            self.size -= 1
            return self.items.pop()

    def isempty(self):
        if(self.size == -1):
            return True
        else:
            return False

    def seek(self):
        if self.isempty():
            return False
        else:
            return self.items[self.size]

    def isOperand(self, i):
        if i in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':
```

```

        return True
    else:
        return False

def infixtopostfix(self, expr):
    postfix = ""
    print("\nPostfix expression after every iteration is:")
    for i in expr:
        if(len(expr) % 2 == 0):
            print("Incorrect infix expr")
            return False
        elif(self.isOperand(i)):
            postfix += i
        elif(i in '+-*/^'):
            while(len(self.items) and self.precedence[i] <= self.precedence[self.seek()]):
                postfix += self.pop()
            self.push(i)
        elif i == '(':
            self.push(i)
        elif i == ')':
            o = self.pop()
            while o != '(':
                postfix += o
            o = self.pop()
    print(postfix)
    # end of for
    while len(self.items):
        if(self.seek() == '('):
            self.pop()
        else:
            postfix += self.pop()
    return postfix

```

```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])
```

```

def generate3AC(pos):
    print("\n\n--- 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

```

```

s = infix_to_postfix()
expr = input('Enter the expression : ')
result = s.infixtopostfix(expr)
if (result != False):
    print("\nThe postfix expr of :", expr, "is", result)
    generate3AC(result)

```

```

k = infix_to_prefix()
rev = ""
rev = k.reverse(expr)
result = k.infixtoprefix(rev)
if (result != False):

    prefix = k.reverse(result)
    print("\n\nThe prefix expr of :", expr, "is", prefix)

```

IMPLEMENTATION

```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:\Users\Shivam\Desktop\STUDY MATERIAL\Compiler Design\Lab\Infix Postfix.py
Enter the expression : A+B*C/D-E

Postfix expression after every iteration is:
A
A
AB
AB
ABC
ABC*
ABC*D
ABC*D/+
ABC*D/+E

The postfix expr of : A+B*C/D-E is ABC*D/+E-

--- THREE ADDRESS CODE GENERATION ---
t1 := B * C
t2 := t1 / D
t3 := A + t2
t4 := t3 - E

Prefix expression after every iteration is:
E
E
ED
ED
EDC
EDC
EDCB
EDCB*/
EDCB*/A
EDCB*/A+
EDCB*/A+-

The prefix expr of : A+B*C/D-E is --A/*BCDE
>>> |
```

Ln: 42 Col: 4

RESULT

Code was successfully implemented and the output was verified.