# THREE ADDRESS CODE FOR LOOPING ,CONDITIONS ,SWITCH CASE STATEMENTS USING INTERMEDIATE CODE

Shanmukha Sudha Kiran.T-**211FA04003**, RaviKiran.K- **211FA04043**,

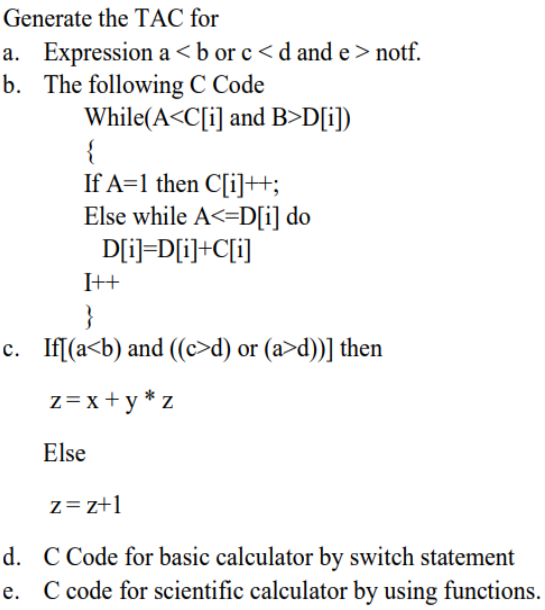
Deepthi .U -**211FA04046**, Sahil Raj - **211FA04677**

**BATCH-17,3$^{Rd}$ B. TECH, CSE BRANCH, VFSTR DEEMED TO BE UNIVERSITY**

## ABSTRACT:

In this paper, we initiate our exploration by introducing the concept of Intermediate Code as a pivotal metric for deriving Three Address Code. The focus lies on generating intermediate code for looping and conditional statements, as well as switch-case structures. To exemplify the application of these concepts, we delve into the intermediate code generation for a switch-case scenario, using a simple calculator as an illustrative example. Through this exercise, our objective is to elucidate the intricate process of determining Three Address Code within the context of practical problem-solving.

## PROBLEM STATEMENT:



Generate the TAC for

a. Expression a < b or c < d and e > notf.

b. The following C Code

```
      While(A<C[i] and B>D[i])
      {
      If A=1 then C[i]++;
      Else while A<=D[i] do
        D[i]=D[i]+C[i]
      I++
      }
```

c. If[(a<b) and ((c>d) or (a>d))] then

```
 z = x + y * z
Else
 z = z+1
```

d. C Code for basic calculator by switch statement

e. C code for scientific calculator by using functions.

## INTRODUCTION:

## INTERMEDIATE CODE:

Intermediate code is a machine-independent representation used during compilation. It acts as a bridge between high-level source code and target machine code. This abstract representation simplifies program analysis and optimization. Examples include three-address code or abstract syntax trees.

Intermediate code can be represented in three ways :

1) Linear Representation
2) Hierarchical Representation
3) Three address code(TAC)

## THREE ADDRESS CODE:

Three Address Code is a low-level intermediate code representation used in compilers to express instructions with at most three operands. Each instruction in this format typically contains an operation, along with two source operands and one destination operand. It simplifies the translation from high-level programming languages to machine code by providing a more manageable and uniform structure for compiler optimizations and code generation.

Three address code can be represented using:

1) Quadtriples
2) Triples
3) Indirect Triples

**ALGORITHM:**

1.start

2.Read two values a,b.

3.Read the operator ,which is used to perform operation.

4. Initialize a variable result to store the calculation result.

5.Maintain the function to calculate Intermediate code.

6.With in function apply basic calculation using operator.

7.For addition (operation == '+'):

- Call the add function with num1 and num2 as arguments.
- Store the returned value in result.

8.For subtraction (operation == '-'):

- Call the subtract function with num1 and num2 as arguments.
- Store the returned value in result.

9.For multiplication (operation == '*'):

- Call the multiply function with num1 and num2 as arguments.
- Store the returned value in result

10. For addition (operation == '/'):

- Call the division function with num1 and num2 as arguments.
- Store the returned value in result.

11.Convert each statement into Intermediate code.

12.Display the resulting code.

13.Stop

**SOURCE CODE:**

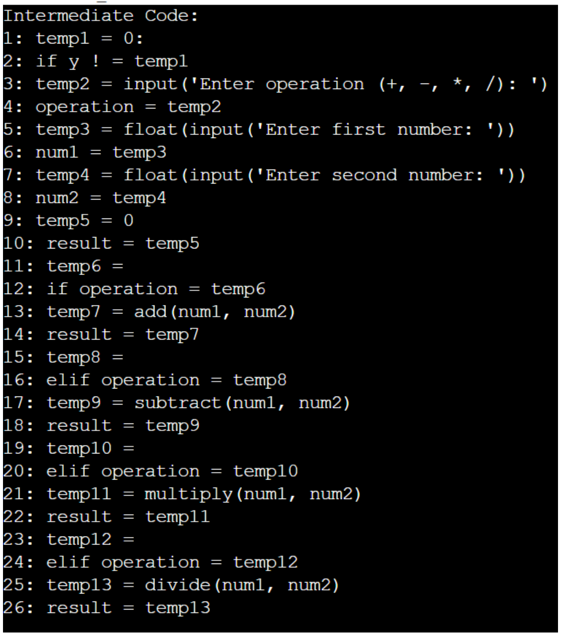
```
def generate_intermediate(cleaned_code):
    final_code = []
    current_temp = 1


    for codeline in cleaned_code:
        if '=' in codeline:   # Assignment statement
            variable, expression = map(str.strip, codeline.split("="))
            temp_var = f"temp{current_temp}"
            final_code.append(f"{temp_var} = {expression}")
            final_code.append(f"{variable} = {temp_var}")
            current_temp += 1
        elif 'if' in codeline:  # If statement
            condition = codeline.split('if')[1].split('goto')[0].strip()
            goto_line = codeline.split('goto')[1].strip('()')
            final_code.append(f"if {condition} goto {goto_line}")
        elif 'goto' in codeline:   # Goto statement
            goto_line = codeline.split('goto')[1].strip('()')
            final_code.append(f"goto {goto_line}")
        elif 'print' in codeline:   # Print statement
```

```
            value = codeline.split('print')[1].strip('()')

final_code.append(f"print({value})")
        elif 'input' in codeline:  # Input statement
            variable, prompt = map(str.strip, codeline.split('='))
            final_code.append(f"{variable} = input({prompt})")
        elif 'return' in codeline:  # Return statement
            value = codeline.split('return')[1].strip()
            final_code.append(f"return {value}")
        elif 'END' in codeline:  # End statement
            final_code.append("END")

    return final_code

# Example code
code = [
    "def add(x, y):",
    "    return x + y",
    "",
    "def subtract(x, y):",
    "    return x - y",
    "",
    "def multiply(x, y):",
    "    return x * y",
    "",
    "def divide(x, y):",
    "    if y != 0:",
    "        return x / y",
    "    else:",
    "        return 'Error: Division by zero'",
    "",
    "# Input reading and function invocation",
    "operation = input('Enter operation (+, -, *, /): ')",
    "num1 = float(input('Enter first number: '))",
    "num2 = float(input('Enter second number: '))",
    "",
    "result = 0",
    "",
    "if operation == '+':",
    "    result = add(num1, num2)",
    "elif operation == '-':",
    "    result = subtract(num1, num2)",
    "elif operation == '*':",
    "    result = multiply(num1, num2)",
    "elif operation == '/':",
    "    result = divide(num1, num2)",
    "else:",
    "    print('Invalid operation')",
    "",
    "# Output the result",
    "print(f'Result: {result}')",
]
```

```
cleaned_code = [line.strip() for line in code]

final_code = generate_intermediate(cleaned_code)


print('\nIntermediate code is:')

for i, line in enumerate(final_code, 1):

    print(f'{i}: {line}')
```

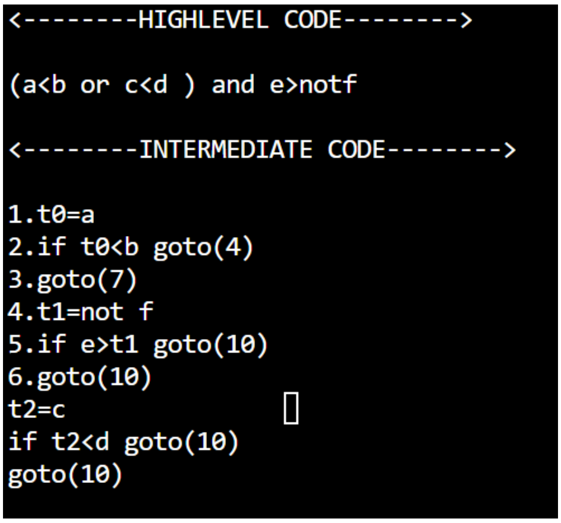
**OUTPUTS:**

**1)**



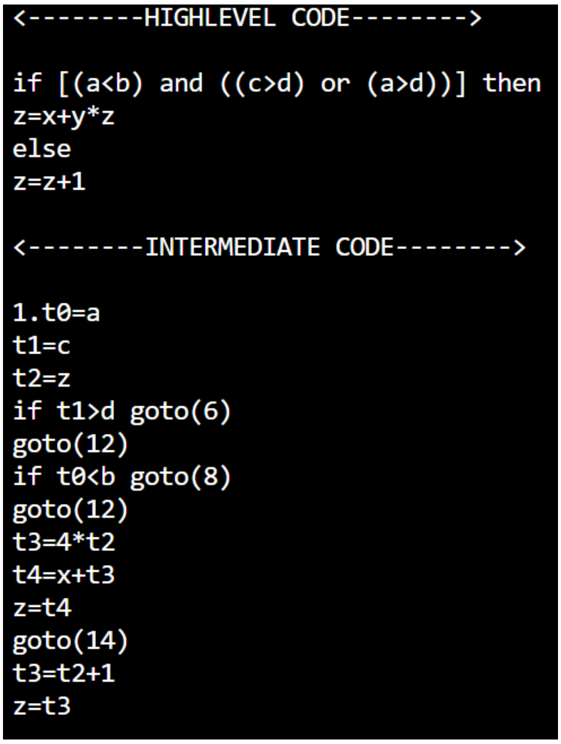
```
Intermediate Code:
1: temp1 = 0:
2: if y ! = temp1
3: temp2 = input('Enter operation (+, -, *, /): ')
4: operation = temp2
5: temp3 = float(input('Enter first number: '))
6: num1 = temp3
7: temp4 = float(input('Enter second number: '))
8: num2 = temp4
9: temp5 = 0
10: result = temp5
11: temp6 =
12: if operation = temp6
13: temp7 = add(num1, num2)
14: result = temp7
15: temp8 =
16: elif operation = temp8
17: temp9 = subtract(num1, num2)
18: result = temp9
19: temp10 =
20: elif operation = temp10
21: temp11 = multiply(num1, num2)
22: result = temp11
23: temp12 =
24: elif operation = temp12
25: temp13 = divide(num1, num2)
26: result = temp13
```

**2)**



```
<--------HIGHLEVEL CODE-------->

(a<b or c<d ) and e>notf

<--------INTERMEDIATE CODE-------->

1.t0=a
2.if t0<b goto(4)
3.goto(7)
4.t1=not f
5.if e>t1 goto(10)
6.goto(10)
t2=c
if t2<d goto(10)
goto(10)
```

**3)**



```
<--------HIGHLEVEL CODE-------->

if [(a<b) and ((c>d) or (a>d))] then
z=x+y*z
else
z=z+1

<--------INTERMEDIATE CODE-------->

1.t0=a
t1=c
t2=z
if t1>d goto(6)
goto(12)
if t0<b goto(8)
goto(12)
t3=4*t2
t4=x+t3
z=t4
goto(14)
t3=t2+1
z=t3
```

**NOTE:**

During the conversion of high-level code to intermediate code, it is essential to minimize the utilization of costly instructions while simultaneously optimizing the usage of memory space.

**REFERENCE:**

https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/

https://gateoverflow.in/174372/intermediate-code

https://www.codingninjas.com/studio/library/intermediate-code-for-procedures

**CONCLUSION:**

In conclusion,The tasks provided cover various aspects of programming, including logical expressions, control flow structures, and Switch case .