Name: Adwait Hegde
Roll No: 2019130019
TE Comp (Batch-A)

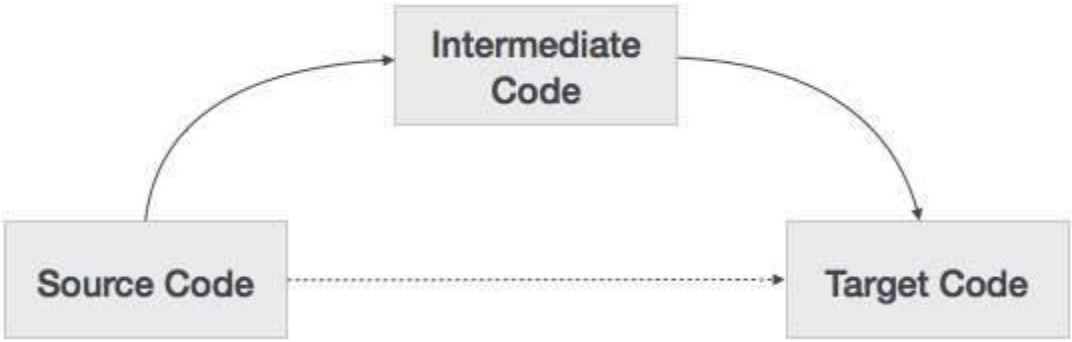
EXPERIMENT 9

Aim:

To Intermediate code generation
Three address code from postfix notation use quadruple representation

Theory:

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.



Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

For example:

```
a = b + c * d;
```

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;
r2 = b + r1;
a = r2
```

r being used as registers in the target program.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms: quadruples and triples.

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg_1	arg_2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg_1	arg_2
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Indirect Triples

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Code:

```
import pandas as pd

# + - * / & ^ =
# x a b + c & d - e * =

operators = ['+', '-', '*', '/', '&', '^', '=']

inp = input('Enter the postfix expression: ')
print()

if ' ' not in inp:
    inp = [i for i in inp]
else:
    inp = inp.split()


qtab = []
stack = []

t = 1

for i in inp:
    args = ['','']
    if i in operators:
        if i != '&':
            args[0] = stack.pop()
            args[1] = stack.pop()
        elif i == '&':
            args[1] = stack.pop()
        stack.append('t'+str(t))
        t+=1
        tab = [i, args[1],args[0],stack[-1]]
        print(' operator encountered-> ',tab)
        qtab.append(tab)
    else:
        stack.append(i)
    print(' STACK=>',stack)

print('\n\n ----------------TABLE----------------')

index = ['']*len(qtab)
print(pd.DataFrame(qtab, columns=['opr', 'arg1', 'arg2','res']))
```

Result:

Output:

```
PS D:\SPIT\SEM 6\CC Lab> py exp9.py
Enter the postfix expression: x a b + c & d - e * =

 STACK=> ['x']
 STACK=> ['x', 'a']
 STACK=> ['x', 'a', 'b']
 operator encountered->  ['+', 'a', 'b', 't1']
 STACK=> ['x', 't1']
 STACK=> ['x', 't1', 'c']
 operator encountered->  ['&', 'c', '', 't2']
 STACK=> ['x', 't1', 't2']
 STACK=> ['x', 't1', 't2', 'd']
 operator encountered->  ['-', 't2', 'd', 't3']
 STACK=> ['x', 't1', 't3']
 STACK=> ['x', 't1', 't3', 'e']
 operator encountered->  ['*', 't3', 'e', 't4']
 STACK=> ['x', 't1', 't4']
 operator encountered->  ['=', 't1', 't4', 't5']
 STACK=> ['x', 't5']


 ----------------TABLE----------------
  opr arg1 arg2 res
0   +    a    b  t1
1   &    c       t2
2   -   t2    d  t3
3   *   t3    e  t4
4   =   t1   t4  t5
```

Conclusion:

In this Experiment, I learned about the concept of Intermediate code generation and implemented three address code from postfix notation use quadruple representation.