Three Address Code → used by the optimizing compilers

Two conditions to follow :-

1. Each instructions should contain atmost 3 addresses (operands)
2. Atmost 1 operator on R.H.S

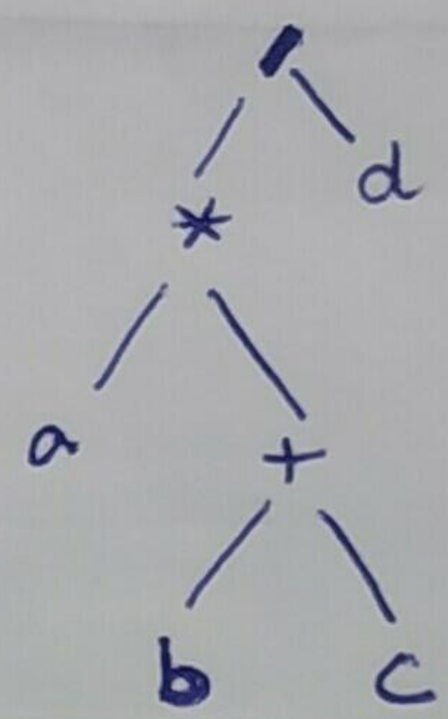
→ Three Address Code is represented in three ways :-

1. Quadruple
2. Triple
3. Indirect Triple

Forms of Intermediate Code

1. Syntax Trees or Abstract Syntax Tree
2. ~~Post fix~~
2. Postfix Notat.n
3. Three Address code

(1) a * (b + c) / d



(2) (a+b) * c

✓ ab+c*

a+(b*c)

✓ abc*+

(a-b) * (c/d)

✓ ab-cd/*

(3) # In three address code, the given expression is broken down into several separate instructions. # These instructions can be easily translate into Assembly language.

It has at most three operands. It is a combination of assignment and a binary operator.

Ex:1
Let the instruction is

$x + y * z$

(It is not a three address code instruction)

- Three operands ✓
- Two operators ✗

Translate it into TAC represent.ⁿ

(RHS only one operator in TAC)

$x + y * z$ // consider precedence while generating TAC

$$\left.\begin{aligned} t1 &= y * z \\ t2 &= x + t1 \end{aligned}\right\}$$ t1 and t2 are temporary names generated by the compiler.

---

① Quadruple / It consists of 4 fields

1. operator
2. argument 1
3. argument 2
4. result

} Each instruction is represented in this form

op	arg 1	arg 2	result

Ex 2 Generate the TAC for the assignment

using Quadruples

$$a = b * -c + b * -c ;$$

(1)

step 1 Create TAC

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

// Instructions like unary minus ($x = -y$) do not use arg2.

$t_3 = b * t_1$

$t_4 = t_2 + t_3$

$a = t_5$ // assignment statement is implied

step 2 Create Quadruple Table

	op	arg1	arg 2	result
(0)	-	c		t_1
(1)	*	b	t_1	t_2
(2)	-	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5

Drawbacks in Quadruple

1. To many temporary variables
2. We need to store all temporary variables in symbol table (requires more amount of memory)

In order to overcome these drawbacks a triple representation is used.

---

(2) Triple

It consists of 3 fields

1. operator
2. argument 1
3. argument 2

operator	argument 1	argument 2

Ex:3 Triple for the assignment

$$a = b * -c + b * -c ;$$

Step 1 Create TAC (same as Quadruple)

Step 2 Create Triple table

addr	Op	arg 1	arg 2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(3)

Major advantage: No need of Temporary variables. (with less amount of memory instructions can be executed)

(3) Indirect Triple

- Triple with an extra table is used.
- Extra table contains pointer to the Triple.

Ex:4 An indirect triple representation of TAC for assignment is:

$$a = b * -c + b * -c$$

step 1 Create TAC

step 2 Create Triple Table

step 3 create Extra Table

Instruction

100	(0)
101	(1)
102	(2)
103	(3)
104	(4)

// Using an array instruction to list pointers to triple in desired order.

Three-Address Instruction form

(1) Assignments instructions of the form $x = y \; op \; z$

where op is binary operator

Ex $x = y + z$, $x = y > z$, $x = y \;\&\&\; z$

(2) Assignment of the form $x = op \; y$ where

op is unary operator

Ex $x = -y$, $x = ++y$, $x = !y$

(3) Copy instructions of the form $x = y$ where value of y is assigned to x

Ex
```
x = 10;
y = x;
```

(4) Unconditional Jump goto L

```
Ex  goto L;      // goto 100;
    L: statements to be executed
```

// break ?
// continue ?
// exit ?

(5) Conditional Jump in the form

(a) if x relop y goto L (8)

(b) if x goto L_1 else goto L_2

6. If x > y goto L

   If x goto L_1 else goto L_2

(6) Procedure calls and returns are implemented using following

```
param x
y = call P, n
return y
```

(7) Address and pointer assignments of the form

```
x = &y
x = *y
*x = y
```

(8) Indexed Copy instructions of the form

```
x = y[i]
x[i] = y
```

Construct Quadruples, triples, indirect triples for the statement

$$\underbrace{(a+b)}_{1} * \underbrace{(c+d)}_{2} - (a+b+c)$$

① TAC (Three Address Code) for above statement

$$t_1 = a + b$$

$$t_2 = c + d$$

$$t_3 = t_1 * t_2$$

$$t_4 = t_1 + c$$

$$t_5 = t_3 - t_4$$

② Quadruple

	op	arg 1	arg 2	Result
(0)	+	a	b	t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	+	t_1	c	t_4
(4)	−	t_3	t_4	t_5

(3) Triple

	op	arg1	arg2
(0)	+	a	b
(1)	+	c	d
(2)	*	(0)	(1)
(3)	+	(0)	c
(4)	−	(2)	(3)

(4) Indirect Triple

instructions

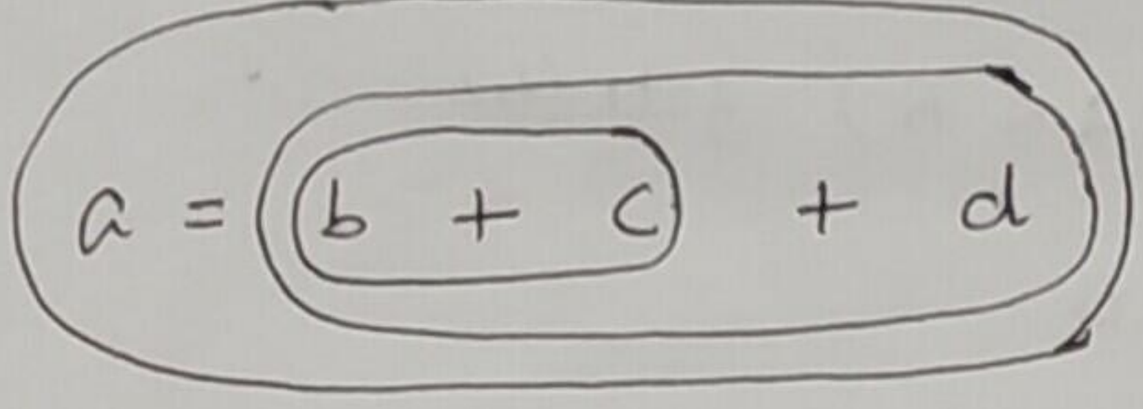
pointer	Address
100	(0)
101	(1)
102	(2)
103	(3)
104	(4)

(I) $a = b + c + d$

(II) $-(a * b) + (c + d) - (a + b + c + d)$

TAC

Sol^n I



$\Rightarrow$

$t_1 = b + c$

$t_2 = t_1 + d$

$a = t_2$

Sol^n II

$-(a * b) + (c + d) - (a + b + c + d)$

$t_1 = a * b$

$t_2 = \text{uminus } t1$

$t_3 = c + d$

$t_4 = t_2 + t_3$

$t_5 = a + b$

$t_6 = t_5 + t_3$

$t_7 = t_4 - t_6$

Uniary operator has higher precedence over binary operators

III rd Generate TAC for

If A < B
then 1
else 0

Soln

(1) if (A < B) goto (4)
(2) $T_1 = 0$
(3) goto (5)
(4) $T_1 = 1$
(5)

IV th Generate TAC for "If a < b and c < d then t = 1
else t = 0"

(1) if (a < b) goto (3)
(2) goto (4)
(3) if (c < d) goto (6)
(4) t = 0
(5) goto (7)
(6) t = 1
(7)