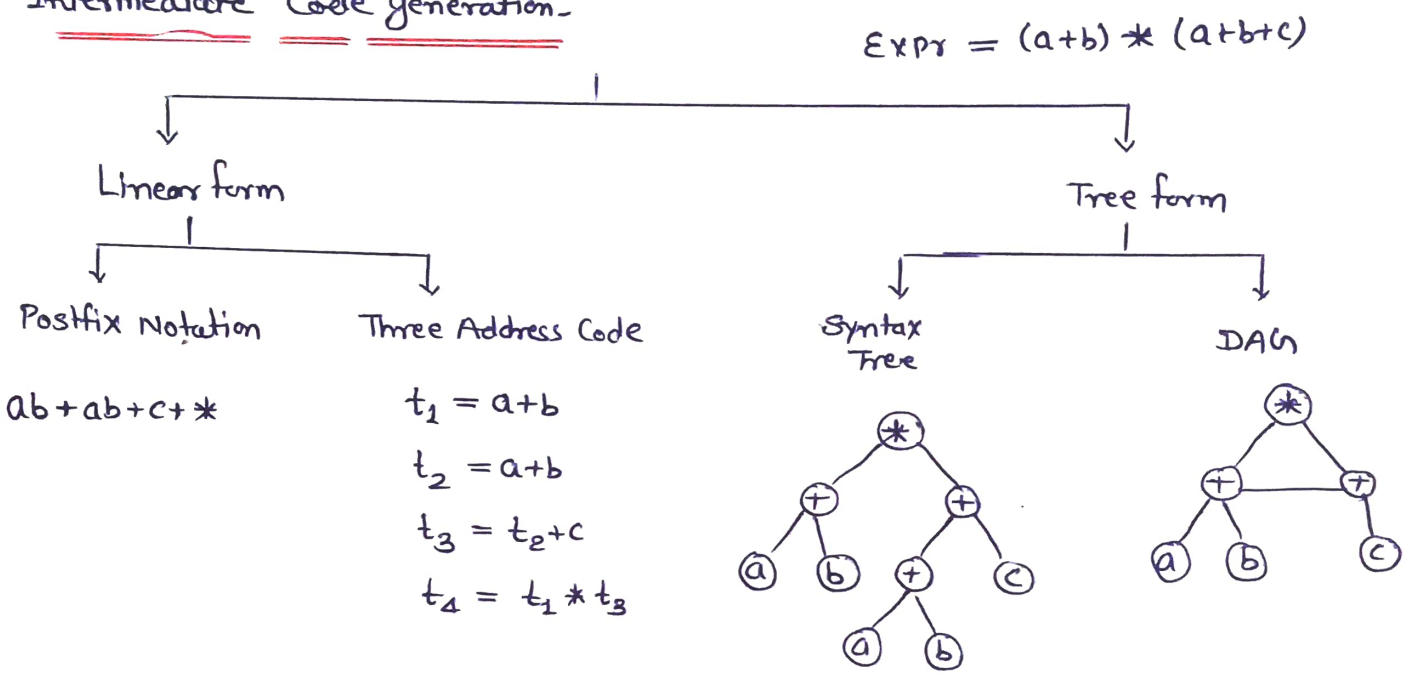
- Heap Allocation-

1) Allocation and Deallocation will be done at anytime based on user requirement
2) Recursion supported
3) Dynamic data structure supported.

* Intermediate Code generation-



- Types of Three Address Code-

1) $x = y \text{ op } z$
2) $x = \text{op } z$
3) $x = y$
4) if (x rel_op y) goto L
5) goto L
6) $A[i] = x$
   $y = A[i]$
7) $x = *p$
   $y = \&x$

- Representation of Three Address Code -

1) Quadruples 2) Triples 3) Indirected Triples

1) Quadruples :-

$x = -(a*b) + (c*d+e)$

$t_1 = a*b$

$t_2 = -t_1$

$t_3 = c*d$

$t_4 = t_3+e$

$t_5 = t_2+t_4$

	Operator	arg1	arg2	result
0	*	a	b	t_1
1	-	t_1		t_2
2	*	c	d	t_3
3	+	t_3	e	t_4
4	+	t_2	t_4	t_5

✓ Allows the optimizers to easily re-position the sub-expression.

2) Triples :-

- Temporary variables are not used
- Reference to the instructions are made

$x = -(a*b) + (c*d+e)$

	Operator	arg1	arg2
0	*	a	b
1	-	(0)	
2	*	c	d
3	+	(2)	e
4	+	(1)	(3)

3) Indirect Triples :-

- Enhancement over Triples
- Uses an additional Instruction Array to list the pointers to the triples in desired order.
- Pointers are used to store result
- ✓ Allows repositioning

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

- **Backpatching :-** This is a Compiler design technique used to resolve forward references in a single pass during ICG.
  - It handles Control-flow Constructs (Eg.- If, While) by generating code with empty placeholders for target labels which are filled later, once the correct address are known.

1) if (a < b) then t = 1 else t = 0

Generate Three Address Code-

1) if (a < b) goto 4
2) t = 0
3) goto 5
4) t = 1
5) END / HALT / NEXT

2) if (a<b) && (c<d) then t=1 else t=0

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

3)

```
for(i=1; i≤n ; i++)
{
    x = a+b*c;
}
```

1) i = 1
2) if (i ≤ n) goto 4
3) goto 9
4) t_1 = b * c
5) t_2 = a + t_1
6) x = t_2
7) i = i+1
8) goto 2
9) HALT / NEXT

#4)
```
switch (i)
{
  Case 1 :
      x1 = a1 + b1 * c1 ;
      break;
  Case 2 :
      x2 = a2 + b2 * c2 ;
      break;
  default :
      x3 = a3 + b3 * c3
      break;
}
```

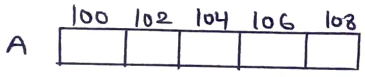
1) if (i == 1) goto 7
2) if (i == 2) goto 11
3) $t_1 = b_3 * c_3$
4) $t_2 = a_3 + t_1$
5) $x_3 = t_2$
6) HALT / NEXT

Case 1
7) $t_1 = b_1 * c_1$
8) $t_2 = a_1 + t_1$
9) $x_1 = t_2$
10) goto 6

Case 2
11) $t_1 = b_2 * c_2$
12) $t_2 = a_2 + t_2$
13) $x_2 = t_2$
14) goto 6

#5)
```
int A[10], B[10]
int x = 0, i;
for(i=0; i<10; i++)
{
   x = x + A[i] * B[i]
}
```

Integer → 2 Bytes



A[i] = BA + (i − lb) * c

BA : Base Address
lb : Lower bound
c : element size

1) x = 0
2) i = 0
3) if (i ≥ 10) goto 15
4) t_1 = base address of A
5) $t_2 = i * 2$
6) $t_3 = t_1[t_2]$
7) t_4 = base Address of B
8) $t_5 = i * 2$
9) $t_6 = t_4[t_5]$
10) $t_7 = t_3 * t_6$
11) $t_8 = x + t_7$
12) $x = t_8$
13) $i = i + 1$
14) goto 3
15) HALT / Next

- for 2D Array - $A[i][j] = BA + (i \times N_c + j) * W$
  ↳ No. of columns

- Procedure Calls-

function - $f(a_1, a_2, a_3, a_4)$

⇒ Param a_1
Param a_2
Param a_3
Param a_4
Call f, 4

Call <function_name>, <number of parameters>

We can use Push operation (stack style) to represent function.
↳ then - Call <function_name>

E.g. -

```
int i, max, sum;
i=0; sum=0;
While (i<10) {
    max = find_max(1);
    A[i] = max;
    sum = sum + max;
    i = i+1;
}
Print_int(sum);
```

1) i = 0
2) sum = 0
3) L_1 : if i<10 goto 5
4) goto 15
5) Param i
6) t_1 = Call find_max, 1
7) max = t_1
8) t_2 = i * 4 (integer = 4B)
9) A[t_2] = max
10) t_3 = sum + max
11) sum = t_3
12) t_4 = i+1
13) i = t_4
14) goto L_1/3
15) Param sum
16) Call Print_Int, 1
17) HALT / NEXT

* DAG (Directed Acyclic Graph) :-

#1) $x = (((a+a) + (a+a)) + ((a+a) + (a+a)))$

t_1 t_2 t_3 t_4 t_5 t_6

$t_1 = a+a$

$t_2 = a+a$

$t_3 = t_1+t_2$

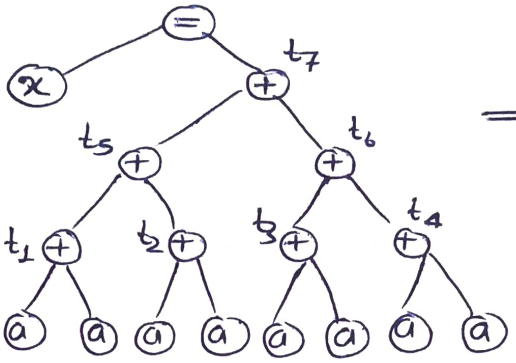
$t_4 = a+a$

$t_5 = a+a$

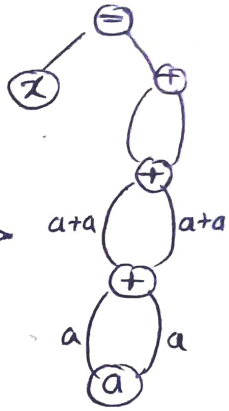
$t_6 = t_4+t_5$

$t_7 = t_3+t_6$

$x = t_7$

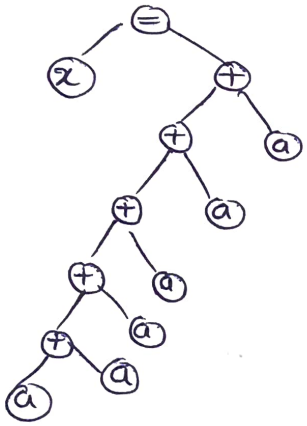


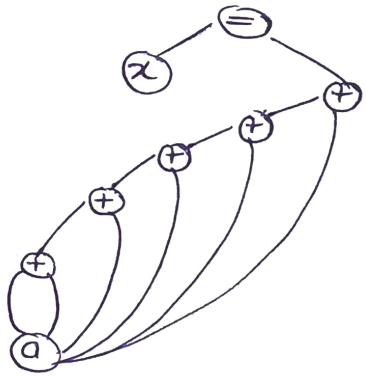
Syntax Tree



DAG

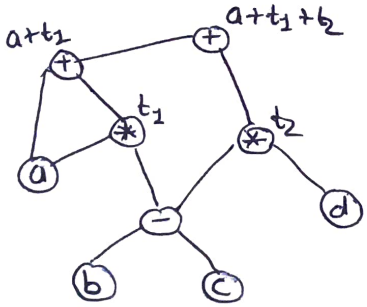
#2) $x = a+a+a+a+a+a$





DAG

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



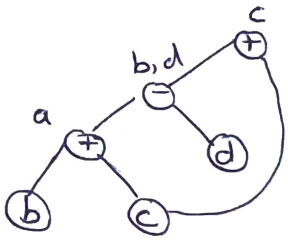
DAG

#4) $a = b+c$

$b = a-d$

$c = b+c$

$d = a-d$



DAG

#5) $a = b+c$

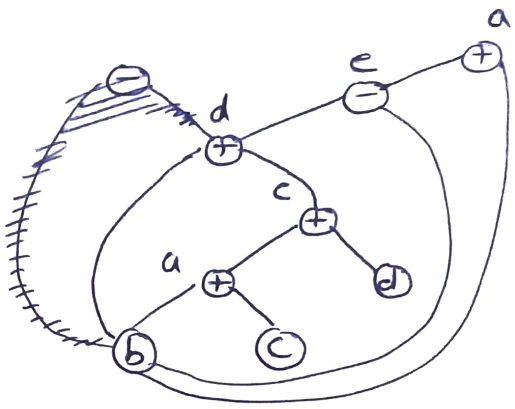
$c = a+d$

$d = b+c$

$e = d-b$

$a = e+b$

find min Number of Nodes and edges



# of Nodes = 8
# of Edges = 10
} Not minimum

→ first minimize expr -

Put values in ~~Expr~~ Expr from bottom to top

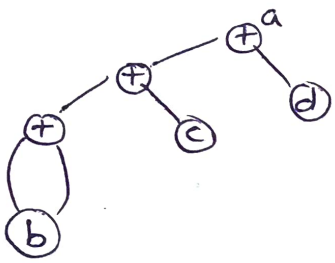
$a = d - b + b \quad (e = d-b)$

$a = d$

$a = b+c \quad (d = b+c)$

$a = b+a+d \quad (c = a+d)$

$a = b+b+c+d \quad (a = b+c)$



# of Nodes = 6
# of Edges = 6