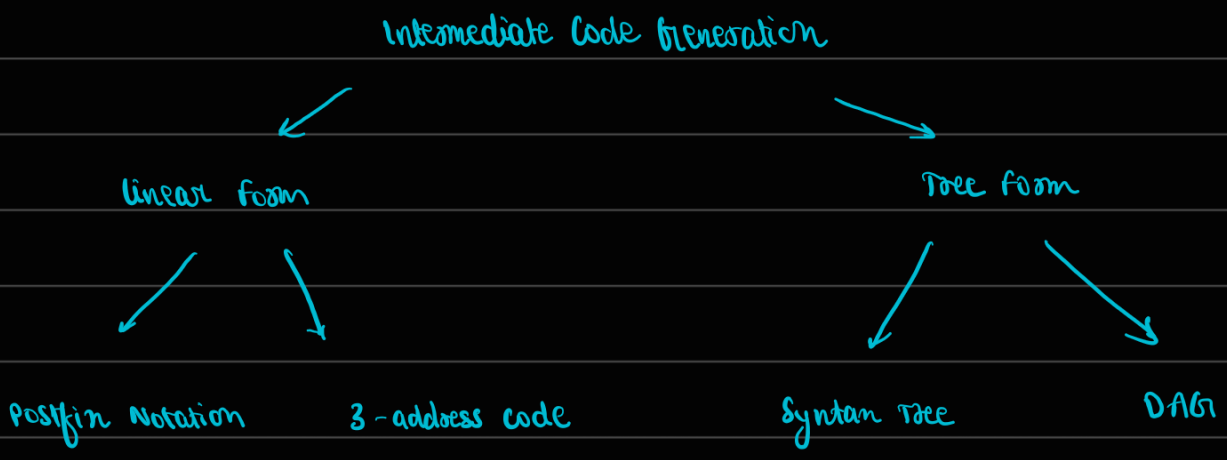
Intermediate Code Generation



Eg. Input : (a+b) * (a+b+c)

- Postfix notation : ab+ab+c+*
- 3-address code : $t_1 = a+b$; $t_2 = a+b$; $t_3 = t_2+c$; $t_4 = t_1 * t_3$
- Syntax tree :
- Directed Acyclic Graph (DAG) :

- Types of 3-address code : (for a C-like language)

OP : Operator

REL OP : Relational operator

- 1 $x = y$ OP z
- 2 $x =$ OP z
- 3 $x = y$
- 4 goto LINE (unconditional goto)
- 5 if (x REL OP y) goto LINE (conditional goto)
- 6 A[i] = x and y = A[i]
- 7 $x = *y$ and $z = \&x$ (reference & de-reference)

- Representation of 3-address codes :

Eg. Input : $-(a*b) + (c*d+e)$

3-address code : $t_1 = a*b$ $t_3 = c*d$ $t_5 = t_2 + t_4$

$t_2 = -t_1$ $t_4 = t_3 + e$

Quadruples :

	Opr.	op1	op2	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

Triples :

	Opr.	op1	op2
0	*	a	b
1	-	(0)	
2	*	c	d
3	+	(2)	e
4	+	(1)	(3)

Indirect Triples :

Instructions (line nos.)		
100	(0)	Reference to triples
101	(1)	
102	(2)	
103	(3)	
104	(4)	

- Quadruples :
  - More space required
  - Operations CAN BE MOVED. meaning (2) can be computed before (0), etc. if necessary for optimization.

- Triples :
  - less space required
  - operations CAN NOT BE MOVED

- Indirect Triplets :
  - need to store 2 memory references (line no. & a pointer to the corresponding entry in the triplets table or something similar)
  - Operations CAN BE MOVED

Back - Patching

- leaving the labels empty and filling them later is called back-patching. We leave the labels empty because we might not know the labels at that point of intermediate code generation

- Conditionals :

Eg. (1) if (a < b) {	L1 : if (a < b) goto L4
t = 1	L2 : t = 0
} else {	L3 : goto L5 (corresponding to else)
t = 0	L4 : t = 1
}	L5 : [corresponds to out of if-else block]

② if ((a<b) && $\bullet_1$
$\bullet_2$ (c<d))
{
t = 1
} else {
t = 0
}
	L_1: if (a<b) goto L_4
	L_2: t = 0 (If $\bullet_1$ is false)
	L_3: goto L_7 (If $\bullet_1$ is false)
	L_4: if (c<d) goto L_6 (If $\bullet_1$ is true)
	L_5: goto L_2 (If $\bullet_2$ is false)
	L_6: t = 1 (If $\bullet_2$ is true)
	L_7:

[we can have either if (a<b) or if (a≥b).
Just change the corresponding gotos / statements]

- Switch case :

eg. ① switch (i) {

case 1 :

$x_1 = a_1 + b_1 * c_1$

break

case 2 :

$x_2 = a_2 + b_2 * c_2$

break

default :

$x_3 = a_3 + b_3 * c_3$

break

}

PTO

L_1 : if (i == 1) goto $C_1 = L_7$

L_2 : if (i == 2) goto $C_2 = L_8$

L_3 : $t_1 = b_2 * c_3$

L_4 : $t_2 = a_3 + t_1$

L_5 : $x_3 = t_2$

L_6 :

C_1 : L_7 : $t_1 = b_1 * c_1$

L_8 : $t_2 = a_1 + t_1$

L_9 : $x_1 = t_2$

L_{10} : goto L_6

C_2 : L_{11} : $t_1 = b_2 * c_2$

L_{12} : $t_2 = a_2 + t_2$

L_{13} : $x_2 = t_2$

L_{14} : goto L_6

These C_1 & C_2 are labels. we can use these to mimic functions (sort of, not exactly)

- **loops :**

Eg. ① `for (int i = 0 ; i ≤ n ; i++) {`

`x = a + b * c`

`}`

L_1 : $i = 0$

L_2 : if ($i \leq n$) goto L_4

L_3 : goto L_9

L_4 : $t_1 = b * c$

L_5 : $t_2 = a + t_1$

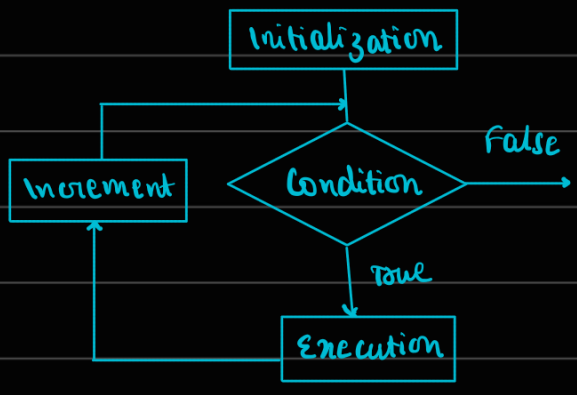
L_6 : $x = t_2$

L_7 : $i = i + 1$

L_8 : goto L_2

L_9

Flow chart of for-loop:



- **1D Arrays:**

eg. ①
```
int A[10], B[10];
int x = 0, i;
for (i = 0; i < 10; i++) {
    x = x + A[i] * B[i]
}
```

Address calculation:

Addr. (A[i]) = Base address + (i - lower bound (lb)) × size of element in bytes (c)

Here, lb = 0 and c = 2 (bytes)

```
L1 : x = 0
L2 : i = 0
L3 : if (i > 10) goto L15
L4 : t1 = Base address of A
L5 : t2 = t1 * 2 (size of element in bytes)
L6 : t3 = A[t2]
L7 : t4 = Base address of B
L8 : t5 = t4 * 2
L9 : t6 = B[t5]
L10 : t7 = t3 * t6
L11 : t8 = x + t7
L12 : x = t8
L13 : i = i + 1
L14 : goto L3
L15 :
```

- **2D Arrays :**

N_r : no. of rows L_r : lower bound of row BA : Base address

N_c : no. of columns L_c : lower bound of col. C : Size of element

- Row major order (RMO) :

$$Addr[A[i][j]] = BA + ((i - L_r) \times N_c + (j - L_c)) \times C$$

- Column major order (RMO) :

$$Addr[A[i][j]] = BA + ((j - L_c) \times N_r + (i - L_r)) \times C$$

Usually, if L_c or L_r is not given, assume $L_r = L_c = 0$.

Eg. ① $x = A[i][j]$ (assuming RMO)

$t_1 = BA$ of A

$t_2 = i - L_r$

$t_3 = t_2 \times N_c$

$t_4 = j - L_c$

$t_5 = t_3 + t_4$

$t_6 = t_5 \times C$

$t_7 = t_1 + t_6$

$t_8 = A[t_7]$

$x = t_8$

Eg. ① For a C program accessing X[i][j][k], the following intermediate code is generated by a compiler : (Assume mat size of an integer is 32-bits & mat of char 8 bits) (GATE - 2014)

```
t0 = i * 1024
t1 = j * 32
t2 = k * 4
t3 = t1 + t0
t4 = t3 + t2
t5 = X[t4]
```

A) X is declared as int X[32][32][8]

B) " " int X[4][1024][32]

C) " " char X[4][32][8]

D) " " char X[32][16][2]

Soln: 3D Array Addressing : (RMO : left to right)

N_1 : no. of rows ; N_2 : no. of cols. ; N_3 : no. of channels

w : size of elements in bytes

$$\text{Addr}(A[i][j][k]) = [(i \times N_2 \times N_3) + (j \times N_3) + (k)] \times w$$

Checking the options :

A) int X[32][32][8] : $[(i \times 32 + 32) + (j \times 8) + k] \times 4$

$= [(i \times 1024) + (j \times 32) + (k \times 4)]$

$(i \times 1024)$ → t_0 ; $(j \times 32)$ → t_1 ; $(k \times 4)$ → t_2

t_0, t_1 → t_3 ; t_3, t_2 → t_4

$t_5 = X[t_4]$ ⟸ t_4

⇒ A is correct

NOTE: 3D Array Addressing (C no: Right to Left)

$$Addr(A[i][j][k]) = [(k \times N_2 \times N_1) + (j \times N_1) + i] \times w$$

Directed Acyclic Graph

- AST & DAG are similar with one major difference: If a node is already computed, DAG will use that node whereas AST will not & end up recomputing it. Thus, a DAG not only represents exprs. succintly but also gives compiler important clues regarding the generation of efficient code to evaluate expressions
- DAG for an expr. identifies its common sub-exprs. (exprs. that occur more than once)
- Many important techniques for local optimization begin by transforming basic blocks into a DAG.

- The DAG for a basic block is constructed as follows:

  1. There is a node representing the initial values of each variable in the basic block
  2. Each instruction 's' in the basic block has a node associated to it. Its children are the nodes that correspond to the last definition of the operands used by 's'
  3. Each node 'N' is labelled by the operator applied at 's' and the list of variables and temporaries for which it is the last definition within the basic block
  4. A node is an OUTPUT NODE if its variables are LIVE ON EXIT

i.e. they will be used by another block in the control flow graph

Eg. ① L_1: $t_1 = 4 * i$

L_2: $t_2 = a[t_1]$

L_3: $t_3 = 4 * i$

L_4: $t_4 = b[t_3]$

L_5: $t_5 = t_2 * t_4$

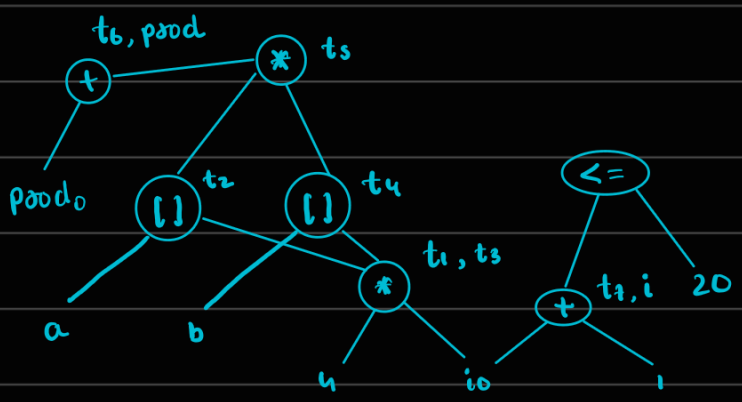
L_6: $t_6 = prod + t_5$

L_7: $prod = t_6$

L_8: $t_7 = i + 1$

L_9: $i = t_7$

L_{10}: if $(i \leq 20)$ goto L_1

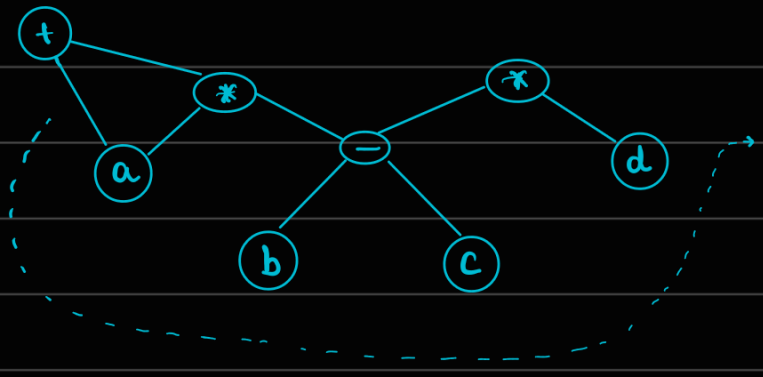


These i_0, $prod_0$ are labels. We need them since the values of i and prod changes and we need to differentiate b/w the different values.

↙

This is called Static Single Assignment (SSA)

② $(a + a * (b - c)) + ((b - c) * d)$



We can do this since the value of a is not changing

Whenever we see 2 parents for a node, that means that it is a common sub expr. used in 2 places. Similar case applies if a node has 'N' parent i.e. it is a common sub expr. used in 'N' places

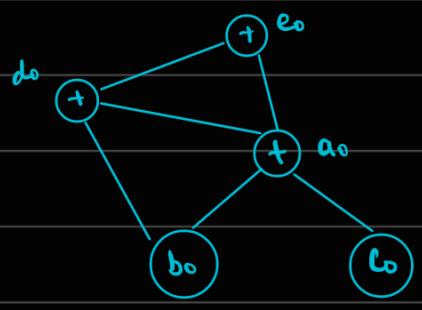
③ Basic block :

$a = b + c$
$d = b + a$
$e = d + a$

$\Rightarrow$

$a_0 = b_0 + c_0$
$d_0 = b_0 + a_0$
$e_0 = d_0 + a_0$

[Change the subscript $\Leftrightarrow$ if the variable changes its value]



Create a new node / re-use an existing one. That decision is made by looking at the subscript of the variable

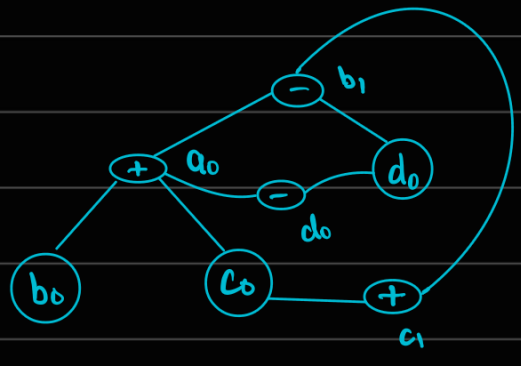
④ Basic block :

$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$

$\Rightarrow$

$a_0 = b_0 + c_0$
$b_1 = a_0 - d_0$
$c_1 = b_1 + c_0$
$d_1 = a_0 - d_0$

The value of b here and here are diff. hence, subscript changes. Similar case for c & d.

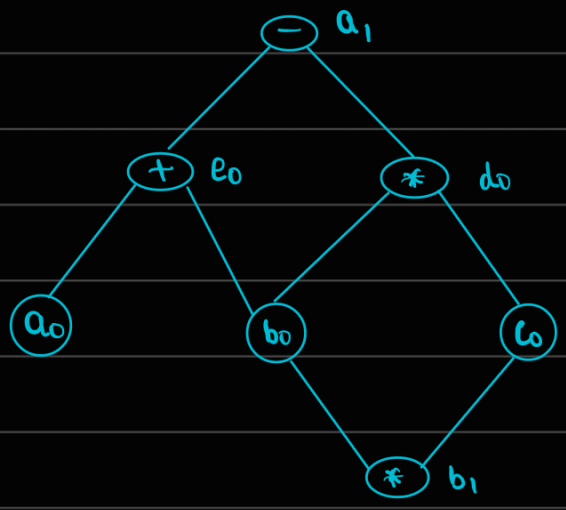


⑤ Basic Block :

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

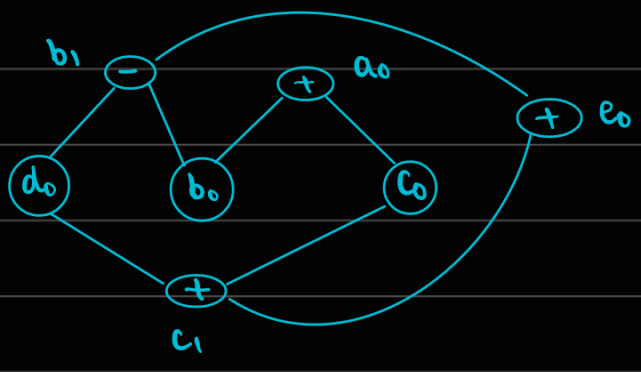
$\Rightarrow$

$d_0 = b_0 * c_0$
$e_0 = a_0 + b_0$
$b_1 = b_0 * c_0$
$a_1 = e_0 - d_0$



(6) Basic Blocks :

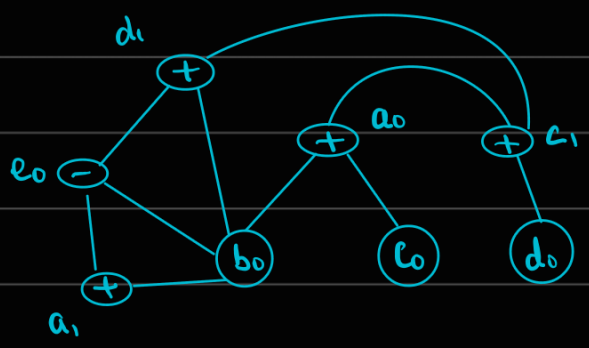
$$
\begin{aligned}
a &= b + c \\
b &= b - d \\
c &= c + d \\
e &= b + c
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
a_0 &= b_0 + c_0 \\
b_1 &= b_0 - d_0 \\
c_1 &= c_0 + d_0 \\
e_0 &= b_1 + c_1
\end{aligned}
$$



(7) Basic Blocks :

$$
\begin{aligned}
a &= b + c \\
c &= a + d \\
d &= b + c \\
e &= d - b \\
a &= e + b
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
a_0 &= b_0 + c_0 \\
c_1 &= a_0 + d_0 \\
d_1 &= b_0 + c_1 \\
e_0 &= d_1 - b_0 \\
a_1 &= e_0 + b_0
\end{aligned}
$$

DO NOT SIMPLIFY UNLESS STATED OTHERWISE



min. num. of edges = 8

min. num. of nodes = 10

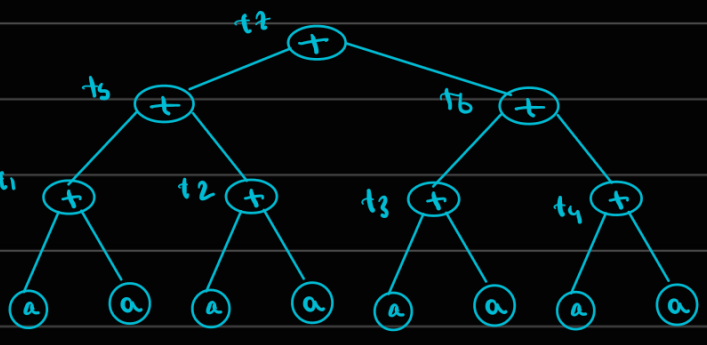
(8) $n = (((a+a)+(a+a))+((a+a)+(a+a)))$

t_1 t_2 t_3 t_4

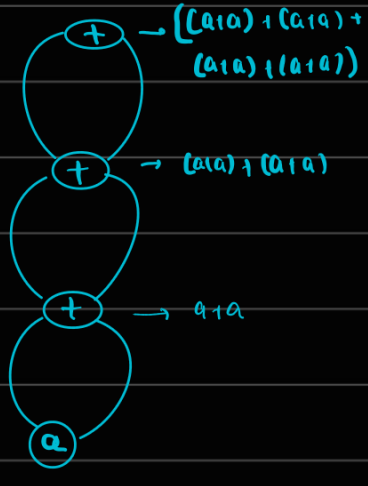
t_5 t_6

t_7

AST:



DAG:



- Applications of DAG:

  - 1 we can automatically detect common sub expressions
  - 2 we can determine the statements that compute the values, which could be used outside the block
  - 3 we can determine which identifiers have their values used in the block.