Backpatching

- It is the activity of filling up unspecified information of labels using appropriate semantics actions in during the code generation process.

Eg: $x<100 \;||\; y>200 \;\&\&\; x!=y$

(|| : logical OR; && : logical AND → if both condition is true then result is true otherwise false)

- we know what is logical OR, if any one of the expression is true then the entire result is true whereas logical AND means if all the expressions are true then only the result is true otherwise the result is false.
- Let $x<100$ guide at 100.

100: if $x<100$ then ____ (blank specifies there is some label here.)

if $x<100$ then entire result is true but we dont know the label for true result.

→ if $x<100$ is false then we have to evaluate $y>200$ but we dont know what is the address of this instruction. Let 101 be the address of next instruction.

101: goto ____
102: if $y>200$ then ____
103: goto ____
104: if $x!=y$ then ____.
105: goto ____
106: true
107: false

- In the process of intermediate code generation we may not know all the labels in the first pass so in order to overcome this problem we need second pass, so that 2nd pass is nothing but backpatching.
- Backpatching is the process of filling these missing labels, so these missing labels are filled with the help of backpatch
- so let us fill the labels in the 2nd pass.
- so in the first pass we don't know what are the labels here

- In order to fill these labels we use backpatching.

2nd pass

100 : if x<100 then 106
101 : goto 102
102 : if y>200 then 104
103 : goto 107
104 : if x != y then 106
105 : goto 107
106 : true
107 : false.

→ Let us write the translation rule.

logical OR

$B \rightarrow B_1 \;||\; M\, B_2$ (|| → logical OR operator; it has 2 arguments; B - boolean expression; B_1 & B_2)

{

Backpatch (B_1.falselist, M.instr);

M - marker NT. it is a NT. it can be epsilon, so epsilon means it is nothing. M gives address of B_2 instr. address of B_2 is specified with help of M here.

$B.truelist = merge(B_1.truelist, B_2.truelist)$;

$B.falselist = B_2.falselist$.

}

B.truelist contains list of addresses of true stm.
B.falselist contains list of addresses of false stm.

logical AND

$B \rightarrow B_1 \;\&\&\; M\, B_2$

{

Backpatch (B_1.truelist, M.instruction);

$B.truelist = B_2.truelist$;

$B.falselist = merge(B_1.falselist, B_2.falselist)$;

}

③ $B \rightarrow ! B_1$

{ B.truelist = B_1.falselist ;
B.falselist = B_1.truelist ;
}

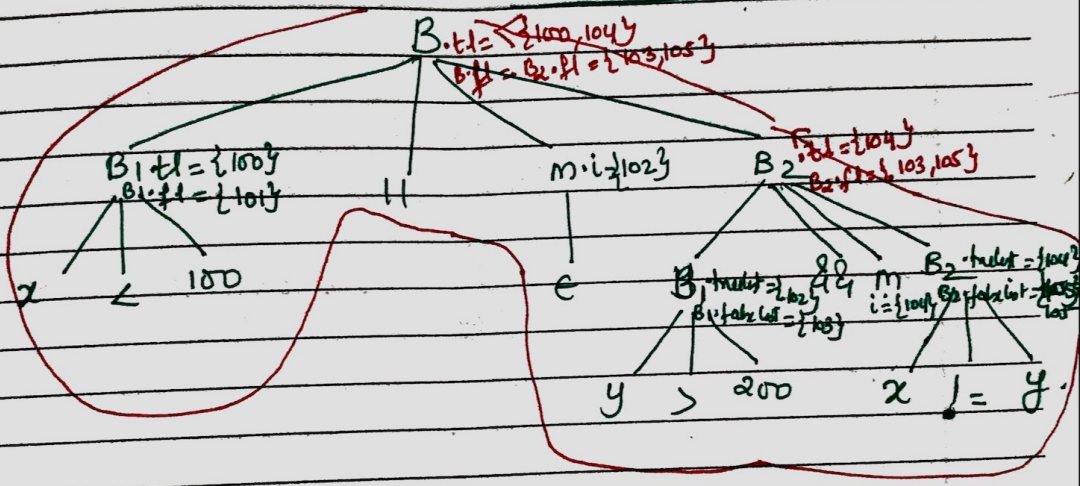
④ $B \rightarrow (B_1)$

{ B.tl = B_1.tl ;
B.fl = B_1.tl ;
}.

⑤ $M \rightarrow \epsilon$

{ m.instr = nextinstr ;
}.

Annotated parse tree.



Basic Blocks

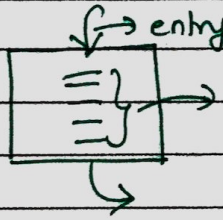
- The basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching.

Eg:-
```
t1 := a*5
t2 := t1+7
t3 := t2-5
t4 := t1+t3
t5 := t2+b.
```

- A basic block is a collection of statements which are always executed in a sequential manner ie nothing but one one one.
- Let us assume that we have 11 statements:

```
1.  PROD = 0
2.  I = 1
3.  T2 = addr(A) - 4
4.  T4 = addr(B) - 4
5.  T1 = 4*i          leader
6.  T3 = T2[T1]
7.  PROD = PROD + T3  leader
8.  I = I + 1
9.  If I <= 20 goto (5)
10. j = j+1           leader
11. K = K+1
```

- Any basic block mainly contains one entry point and one exit point. So let us assume that this is a block.



- So first the control will be at the 1st stm. ie stm to be executed.
- Then the control will be at the 2nd stm.
- Likewise all the stms will be executed.

- Once all the stm in a block are executed then the control exits from the block.
- Any basic block contains one entry point as well as one exit point.
- * Basic block should not contain condihonal control statements as well as unconditional control statements in the middle of the block.

condihonal control stm → simple if, if-else - etc, else if ladder, switch, while, do-while.

unconditional control stm → break, continue, exit etc.

→ We can have condihonal control stm as the first stm (unconditional control stm) as the last stm of the blk. &
But we should not have condihonal control stm and unconditional control stm in the middle of the block. ie intermediate level of the block.

Eg: $x = a+b+c$. → This is not three address code. Convert it to TAC instn.

```
t1 = a+b
t2 = t1+c
x = t2
```

⇒ is a basic blk.
⇒ if u place these stm in basic blk all instn will be executed one by one.

* Purpose of forming basic block is for register allocation and for optimization.

Algorithm for partitioning 3 Address code into Basic block.

Rule 1) Determining the leaders.

a) First stm of a block is always treated as a leader.

b) The target of conditional (or) unconditional jump stm is a leader.

c) The statement following ~~leader~~ conditional or unconditional jump is a leader.

Rule 2) Determining the basic blocks - starts at leader & ends before the next leader.

- beginning at the leader statements and ends before the next leader.

* We can take stms in any manner that is our choice.
* main intension is to partition into basic blk.

1. PROD = 0 - leader.
2. I = 1
3. T2 = addr(A) - 4
4. T4 = addr(B) - 4
5. T1 = 4 * i - leader.
6. T3 = T2[T1]
7. PROD = PROD + T3 - leader.
8. I = I + 1
9. If I <= 20 goto (5) → conditional jump target stm is 5.
10. j = j + 1 - leader. - if stm following conditn or uncondtn jump.
11. K = K + 1.
12. If j <= 5 goto (7)
13. i = i + j - leader. rule 1(c).

Leaders stm: 1, 5, 7, 10, 13

B1:
```
PROD = 0
I = 1
T2 = addr(A) - 4
T4 = addr(B) - 4.
```

B2:
```
T1 = 4 * i
T3 = T2[T1]
```

```
PROD = PROD + T3
I = I + 1
If I <= 20 goto B2
```

B3

```
j = j + 1
K = K + 1
If j <= 5 goto B3
```

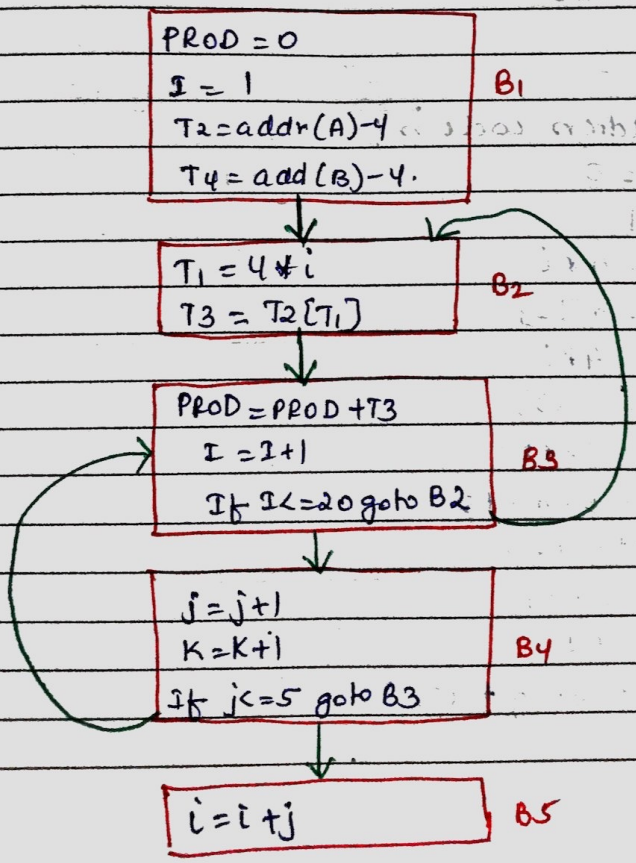
B4

```
i = i + j
```

B5

Flow Graph

- A flow graph is a directed graph in which the flow control information is added to the graph or basic blocks. directed graph means there are some directions from one node to another.
- A flow graph is a collection of nodes where we can have some edges from one node to another node.
- Here the basic blocks becomes the nodes of flow graph.
- So if we take previous example we have 5 nodes ie B1, B2, B3, B4, B5.
- The edges will specify the flow of information. from one block to another block.
- If there is any reference from B1 to B2 then we can say that B2 block immediately follows B1.
- If there is a reference B5 to B4 then we can say that B4 immediately follows B5.
- In previous eg B1 is called initial block because it contains the first leader.



Loop

- Loop is a collection of nodes in the flow graph such that
i) All such nodes are strongly connected. That means always there is a path from any node to any other node within that loop.
ii) The collection of nodes has unique entry. That means there is only one path from a node outside the loop to the node inside the loop.
iii) The loop that contains no other loop is called inner loop.

Eg.

```
Begin
    prod := 0
    i = 1;
    do begin
        prod := prod + a[i] * b[i];
        i = i + 1;
    end
    while i <= 20
end.
```

Three address code is :

B1
1. prod := 0
2. i := 1

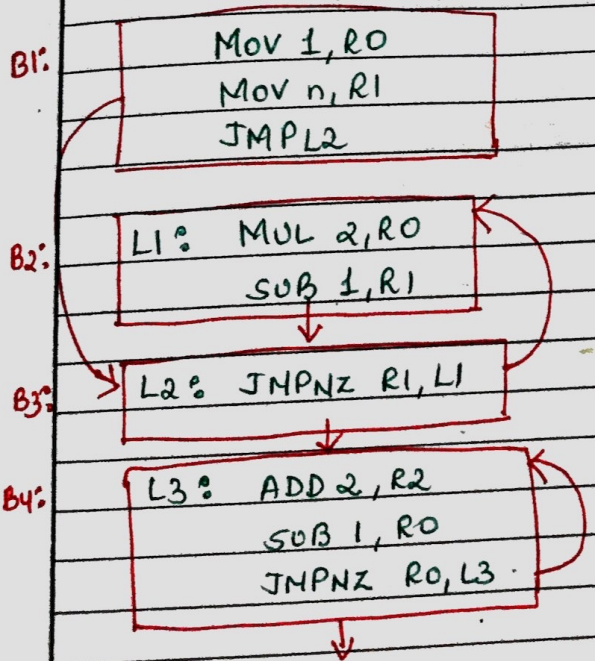
B2
3. t1 := 4 * i
4. t2 := a[t1]
5. t3 := 4 * i
6. t4 := b[t3]
7. t5 := t2 * t4
8. t6 := prod + t5
9. prod := t6
10. t7 := i + 1
11. i := t7
12. if i <= 20 goto (3)

B1
1. prod := 0
2. i := 1

B2
3. t_1 := 4*i
4. t_2 := a[t_1]
5. t_3 := 4*i
6. t_4 := b[t_3]
7. t_5 := t_2 * t_4
8. t_6 := prod + t_5
9. prod := t_6
10. t_7 := i+1
11. i := t_7
12. if i<=20 goto (3)

B_1 is the predecessor of B_2 and B_2 is the successor of B_1.

Loop (Example)



Strongly connected components:
SCC = {{B2, B3}, {B4}}.

Entries:
B3, B4.

to come to B_3 I have to go through B2 only to B_2 is strongly connected.

Transformations on Basic blocks

→ Basic block computes set of expressions.
→ Values of the variables outside the block is decided by the computation inside the block.
→ Two basic blocks are equivalent if they compute the same set of expressions.

Equivalence of Basic blocks

```
b = 0
t1 = a + b
t2 = c * t1
a = t2
```

↓

```
a = c * a
b = 0.
```

```
a = c * a
b = 0
```

↓

```
a = c * a
b = 0.
```

* Next-use Information

- Next-use information is a collection of all the names that are useful for next subsequent statement in a block. The use of a name is defined as follows.
  consider a statement
  $x := i$
  $j := x \text{ op } y$.
  that means the statement j uses value of x.
- The next-use information can be collected by making the backward scan of the programming code in that specific block.
- Suppose the 3-address statement is as given below.
  L: a := b op c then the steps in backward scan are -
  i) The currently found information in symbol table regarding the next use and liveness of a, b and c is associated with the statement L.
  ii) In the symbol table set 'a' to "not live" and "no next use".
  iii) Set 'b' and 'c' to live and next uses of b and c in symbol table.

* Register Allocation and Assignment

- The use of register operands instead of memory operands is always the faster and shorter. Proper use of registers help in generating the good code.
- Certain strategies are adopted by the compiler for register allocation and assignment.
- The most commonly used strategy to register allocation & assignment is to assign specific values to specific registers. For instance for base addresses separate set of registers can be used. similarly for storing the stack pointers again a separate set of registers can be assigned; arithmetic computations can be done using separate set of registers.

Advt:- the design procedure of compiler for code generation becomes simplified.

disadvt:- the design of compiler becomes complicated because of restrictive use of registers.

Strategies used in register allocation & Assignment

1. Global register allocation.
2. Usage count
3. Register assignment for outer loop.
4. Graph coloring for register assignment.

1) Global register allocation

- while generating the code the registers are used to hold the values for the duration of single block.
- All the live variables are stored at the end of each block.
- For the variables that are used consistently we can allocate specific set of registers.
- Allocation of variables to specific registers that is consistent across the block boundaries is called global register allocation.

→ Strategies adopted while doing the global register allocation

* It stores the most frequently used variables in fixed registers throughout the loop.
* Another strategy is to assign some fixed number of global registers to hold the most active values in each inner loop.
* The registers not already allocated may be used to hold # values local to one block.
* In certain languages programmer can do the register allocation by using register declaration.

2) Usage Count

- The usage count is the count for the use of some variable x in some register used in any basic block.
- The usage count gives the idea about how many

units of cost can be saved by selecting a specific variable for global register allocation.

- The approximate formula is given as,

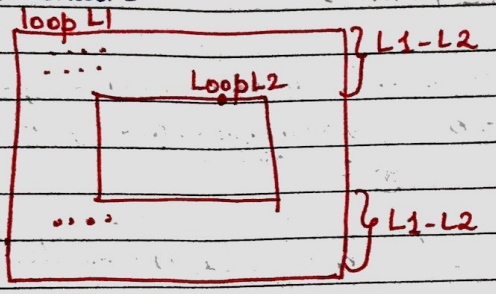
$$\sum_{\text{block } B \text{ in } L} (use(x, B) + 2 * live(x, B))$$

where use(x, B) - no of times x used in block B prior to any definition of x

live(x, B) = 1 if x is live on exit from B, otherwise live(x) = 0.

3) Register Assignment for outer loop

- Consider that there are 2 loops. L1 is outer loop and L2 is an inner loop.
- Allocation of variable a is to be done to some register.
- Consider the scenario.



Nested loop.

⇒ following criteria should be adopted for register assignment for outer loop.

1) If a is allocated in loop L2 then it should not be allocated in L1-L2.
2) If a is allocated in L1 and it is not allocated in L2 then store a on a entrance of to L2 and load a while leaving L2.
3) If a is allocated in L2 and not in L1 then load a on entrance of L2 and store a on exit from L2.

→ Directed Acyclic Graph

DAG Representation of Basic blocks:

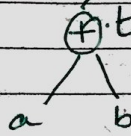
- DAG represents the structure of a basic block.
1) In a DAG, internal node represents operators.
2) Leaf node represents identifiers (names of variables), constants (values).
3) Internal node also represents result of expressions.

Applications of DAG:- → If expression is repeated several times.

1) Determining the common sub expressions.
2) Determining which names are used inside the block and computed outside the block.
3) Determining which statements of the block could have their computed value outside the block.
4) Simplifying the list of Quadruples by eliminating common sub expressions.

Eg: let us consider one block contains only one stm.

$t = a + b$ → internal node also represents result of expression.



+ - operator.
(a/b) - as identifier.
t → result

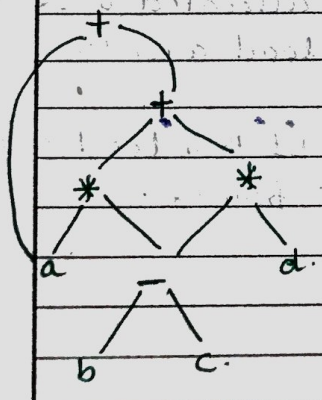
Construction of DAG for a basic block.

Eg: Construct DAG for the expression

$a + a * (b - c) + (b - c) * d$ → right children

(b-c) → left children, internal node

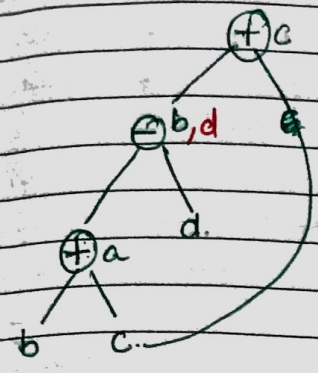
- parenthesis is having higher precedence.
so (b-c) will be evaluated first.



Eg: Construct DAG for the block.

① $a = b + c$ ② $c = b + c$

③ $b = a - d$ ④ $d = a - d$



since a-d is also represented and stored in b it is b,d.

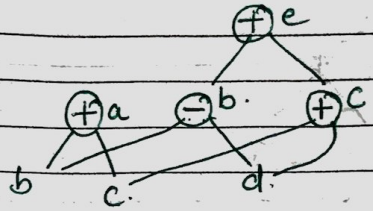
Eg: Construct DAG for the following.

$a = b + c$

$b = b - d$.

$c = c + d$.

$e = b + c$.



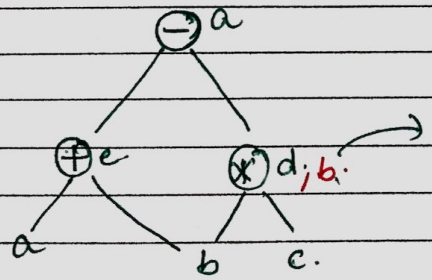
Eg:

$d = b * c$

$e = a + b$

$b = b * c$

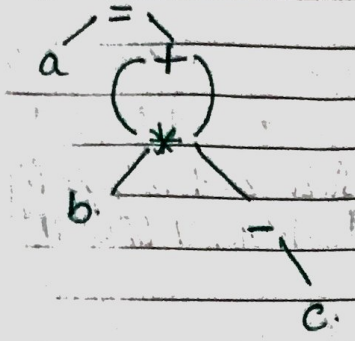
$a = e - d$.



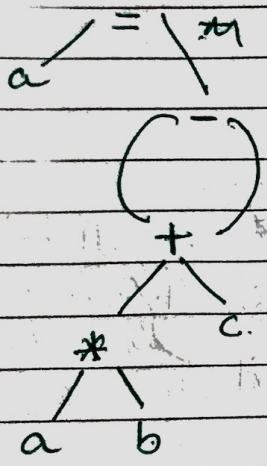
b value doesn't change, hence can be written as d, b.

unary minus / unnary operators have higher priority than * and +.

Eg: a = b * -c + b * -c



Eg: a = (a * b + c) - (a * b + c)

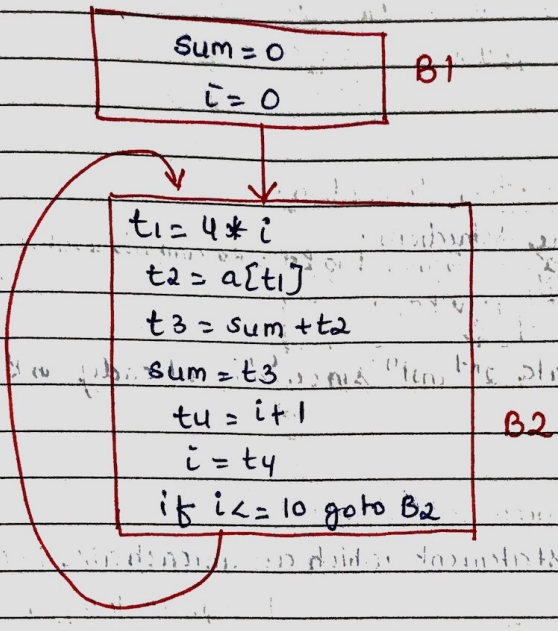


Eg: Consider sum = 0;
for (i = 0; i <= 10; i++)
sum = sum + a[i];

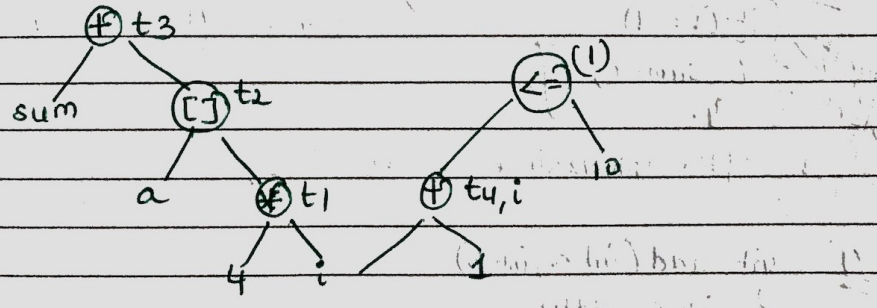
Soln The 3-address code is:

1) sum = 0
2) i = 0
3) $t_1 = 4 * i$
4) $t_2 = a[t_1]$
5) $t_3 = sum + t_2$
6) sum = t_3
7) $t_4 = i + 1$
8) i = t_4
9) if i <= 10 goto (3)

Basic blocks are:



let us consider block B2 for construction of DAG.



DAG for block B2.

Peephole optimization

- instead of examining entire prgm we examine short sequence & if there is some possibility replace by short sequence of instructions or fast instructions.

- This technique is applied to improve the performance of program by examining a short sequence of instructions in a window (peephole) & replace the instructions by a faster or short sequence of instructions.

Characteristics of Peephole optimization

1. Redundant instruction elimination.
2. Flow of control optimization.
3. Algebraic simplification
4. use of machine idioms.
5. Removal of unreachable code.

1) Redundant instruction elimination.

eg: Consider the instructions:

```
Mov R0, a   } a value is moved to R0 - R0 contains a value
Mov a, R0   } Mov R0, a
```

we can eliminate 2nd instn since 'a' is already in R0.

2) Removal of unreachable code.

- eliminate the statement which are unreachable ie never executed.

eg:

```
i = 0;                          i = 0;
if (i == 1)                     throughout the prgm i value should be 0. but what is the condtn i value is 1
{ sum = 0;                      can eliminate this block
}
```

so after optimization code is `i = 0;`

eg:

```
int add (int a, int b)
{ int c = a+b;
  return c;
  printf ("%d", c);   → eliminate.
}                       ↓ is never executed.
```

3) Flow-of-control optimizations - using peephole optimizations.
- unnecessary jumps can be eliminated.

eg:
```
    goto L1
    ...
L1: goto L2
    ...
L2: goto L3
    ...
L3: MOV a, R0
```
Multiple jumps can make the code inefficient. Above code can be replaced by

```
    goto L3
    ...
L1: goto L3
    ...
L2: goto L3
    ...
L3: MOV a, R0
```

4) Algebraic simplifications:

eg: $x = x + 0$ or $x = x * 1$ ⇒ is nothing but x only. By executing these statements result of x won't change.

The above statements can be eliminated because by executing those statements the result 'x' won't changes.

eg: $a = x \wedge 2$ (power operator) replaced with $a = x * x$ ⇒ replace expensive operator with cheapest operator.

eg: $b = y / 8$ (expensive operator) " " $b = y >> 3$ ⇒ division by 2. (right-shift operator)

5) Use of Machine idioms.
- It is the process of using powerful features of CPU instructions.

eg: auto inc/dec features can be used to inc/dec variable.

eg: `a = a + 1` } `inc a;`

`a = a - 1` } `dec a;`

* A Simple code Generator

- Code generation is the last phase of compiler.
→ It generates target code for a sequence of instn.
→ It uses a function getReg() to assign registers to variables.
→ It uses 2 data structures: ① Register descriptor
② Addres descriptor.

which register contains which variable

Register descriptor:- used to keep track of which variable is stored in a register. Initially all registers are empty.

where variable may be stored is reg, memory location, stack etc

Addres descriptor:- used to keep track of location where variable is stored. location may be register, memory address, stack ....

A simple code generator Algorithm

- The following actions are performed by code generator for an instruction $x = y \text{ op } z$. Assumes that L is the location where the output of y op z is to be saved

1) Call function getReg() to get the location of L.
2) Determine the present location of 'y' by consulting Addres descriptor of y. If y is not present in the location L then generate the instruction MOV y', L to copy value of y to L. (MOV memory y, L; y's address descriptor)
3) The present location of z is determined using step 2 and the instruction is generated as op z', L. (z' address descriptor of z)
4) Now L contains the value of y op z, ie Assigned to x. So, if L is a register then update its descriptor that it contains value of x. update Addres descriptor of x to indicate that it is stored in 'L'.
5) If y, z have no future use, then update the descriptors to remove y & z.

eg: $d = (a-b) + (a-c) + (a-c)$

Three address code:- $t_1 = a - b$

$t_2 = a - c$

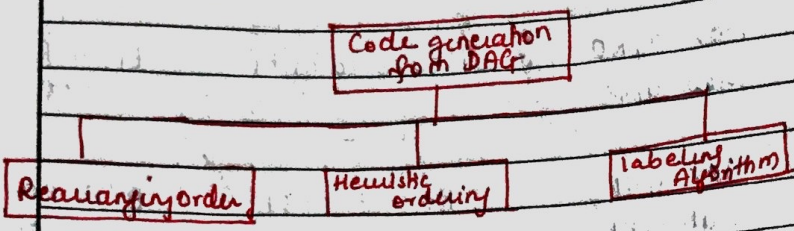
$t_3 = t_1 + t_2$

$d = t_3 + t_2$

Statement	Code Generation	Register Descriptor (which register contains which variable)	Address Descriptor (Locations where the variable is stored)
$t_1 = a - b$	Mov a, R0 (destination register); Sub b, R0	R0 contains t_1 ($a-b$)	t_1 in R0
$t_2 = a - c$	Mov a, R1; Sub c, R1	R0 contains t_1; R1 contains t_2	t_1 in R0; t_2 in R1
$t_3 = t_1 + t_2$	Add R1, R0 (our choice) or R0, R1	R0 contains t_3; R1 contains t_2	t_3 in R0; t_2 in R1
$d = t_3 + t_2$	add R1, R0 (result is available in R0 so mov not required); ~~Mov R0, d~~	R0 contains d; R1 contains t_2	d in R0.

Generating code from DAG

- Generating code from DAG is much simpler than the linear sequence of 3 addres code.
- Using DAG we can rearange some sequence of instructions and generate an efficent code.
- Various algorithms used for generating code from DAG



(1) Rearranging Order.

- The order of 3 addres code affects the cost of the object code being generated.
- By changing the order in which computations are done we can obtain the object code with minimum cost.

Eg:

$t_1 = a + b$

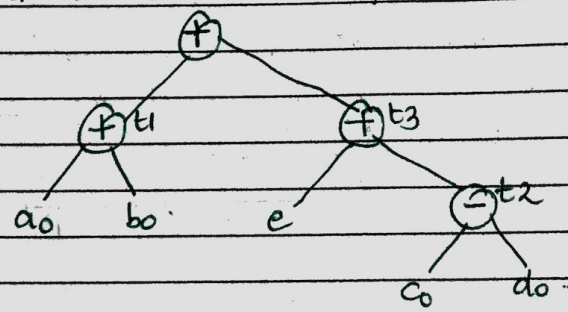
$t_2 = c - d$

$t_3 = e + t_2$

$t_4 = t_1 + t_3$

for the expression $(a+b) + (e + (c+d))$

- A DAG can be constructed for the above sequence.



- The code can be generated by translating the 3 addres code line by line.

```
MOV a, R0
ADD b, R0
MOV c, R1
```

```
SUB d, R1
MOV R0, t1
MOV e, R0
ADD R0, R1
MOV t1, R0
ADD R1, R0
MOV R0, t4.
```

Now if we change the ordering sequence of the above TAC.

$t_2 = c - d$

$t_3 = e + t_2$

$t_1 = a + b$

$t_4 = t_1 + t_3$

then we get an improved code as:

```
MOV c, R0
SUB d, R0
MOV e, R1
ADD R0, R1
MOV a, R0
ADD R1, R0
MOV R0, t4.
```

2) Heuristic Ordering

- The heuristic ordering algorithm is as follows:

(1) Obtain all the interior nodes. consider these interior nodes as unlisted interior nodes.

(2) while (unlisted interior nodes remain)

{

(3) pick up an unlisted node n, whose parents have been listed.

(4) List n;

(5) while (the leftmost child m of n has no unlisted parent AND is not leaf

{

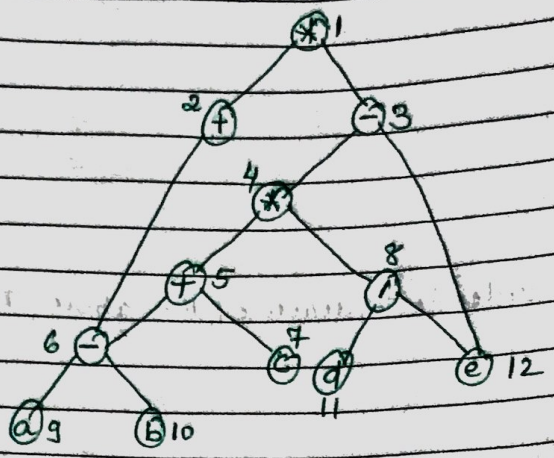
(6) List m;

(7) n = m;

}

}.

- First we will draw a DAG.



- The DAG is first numbered from top to bottom and from left to right. Then consider the unlisted interior nodes 1 2 3 4 5 6 8

Initially the only node with unlisted parent is 1

1. Set n=1 by line 4) of algorithm.

Now left argument of 1 is 2 and parent of 2 is 1 which is listed. Hence list 2.

2. Set n=2 by line 7) of algorithm.

+	~~2~~	3	4	5	6	8

Now we will find the leftmost node of 2 and that is 6. But 6 has unlisted parent 5. Hence we cannot select 6.

We therefore can switch to 3. The parent of 3 is 1 which is listed one. Hence list 3.

Set n=3.

+	~~2~~	~~3~~	4	5	6	8

The left of 3 is 4. As parent of 4 is 3 and that is listed hence list 4. left of 4 is 5 which has listed parent (ie 4) hence list 5. similarly list 6.

+	2	3	~~4~~	5	6	8

As now only 8 is remaining from the unlisted interior nodes we will list it.

Hence the resulting list is 1 2 3 4 5 6 8. Then the order of computation is decided by reversing this list.

We get the order of evaluation as 8 6 5 4 3 2 1. That also means that we have to perform the computations at these nodes in the given order.

$t_8 = d / e$

$t_6 = a - b$

$t_5 = t_6 + c$

$t_4 = t_5 * t_8$

$t_3 = t_4 - e$

$t_2 = t_6 + t_4$

$t_1 = t_2 * t_3$

This gives the optimized code for DAG even though there are any no of registers.