Unit – 5
Code Generation
(PART-I)

Mrs. Ruchi Sharma
ruchi.sharma@bkbiet.ac.in

Topics to be covered

- Issues in code generation
- Target machine
- Basic block and flow graph
- Transformation on basic block
- Next use information
- Register allocation and assignment
- DAG representation of basic block
- Generation of code from DAG

Code Generation

- Properties of Target code:
  1. Correctness
  2. High quality
  3. Efficient use of resources of target code
  4. Quick code generation

Issues in Code Generation

Input to code generator

- Input to the code generator consists of the intermediate representation of the source program.
- Types of intermediate language are:
  1. Postfix notation
  2. Quadruples
  3. Syntax trees or DAGs
- The detection of semantic error should be done before submitting the input to the code generator.
- The code generation phase requires complete error free intermediate code as an input.

Target program

- The output may be in form of:
  1. Absolute machine language: Absolute machine language program can be placed in a memory location and immediately execute.
  2. Relocatable machine language: The subroutine can be compiled separately. A set of relocatable object modules can be linked together and loaded for execution.
  3. Assembly language: Producing an assembly language program as output makes the process of code generation easier, then assembler is require to convert code in binary form.

Memory management

- Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator.
- We assume that a name in a three-address statement refers to a symbol table entry for the name.
- From the symbol table information, a relative address can be determined for the name in a data area.

Instruction selection

- Example: the sequence of statements

  a := b + c

  d := a + e

- would be translated into

  MOV b, R0

  ADD c, R0

  MOV R0, a

  MOV a, R0

  ADD e, R0

  MOV R0, d

- Here the fourth statement is redundant, so we can eliminate that statement.

Register allocation

- The use of registers is often subdivided into two sub problems:
- During register allocation, we select the set of variables that will reside in registers at a point in the program.
- During a subsequent register assignment phase, we pick the specific register that a variable will reside in.
- Finding an optimal assignment of registers to variables is difficult, even with single register value.
- Mathematically the problem is NP-complete.

Choice of evaluation

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.
- Picking a best order is another difficult, NP-complete problem.

Approaches to code generation

- The most important criterion for a code generator is that it produces correct code.
- The design of code generator should be in such a way so it can be implemented, tested, and maintained easily.

Target Machine

Target Machine

- We will assume our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps.
- The underlying computer is a byte-addressable machine with n general-purpose registers, $R_0, R_1, \ldots, Rn$
- The two address instruction of the form: *op source, destination*
- It has following opcodes:

  MOV (move source to destination)

  ADD (add source to destination)

  SUB (subtract source to destination)

Instruction Cost

- The address modes together with the assembly language forms and associated cost as follows:

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k +contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1

- The instruction cost can be computed as one plus cost associated with the source and destination addressing modes given by “extra cost”.

Instruction Cost

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k +contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1

- Calculate cost for following:

MOV B,R0
ADD C,R0
MOV R0,A	

Instruction Cost

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k +contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1

- Calculate cost for following:

MOV *R1 ,*R0 MOV *R1 ,*R0	

Basic Block & Flow Graph

Basic Blocks

- A 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 except at the end.
- The following sequence of three-address statements forms a basic block:

```
t1 := a*a
t2 := a*b
t3 := 2*t2
t4 := t1+t3
t5 := b*b
t6 := t4+t5
```

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements.

Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

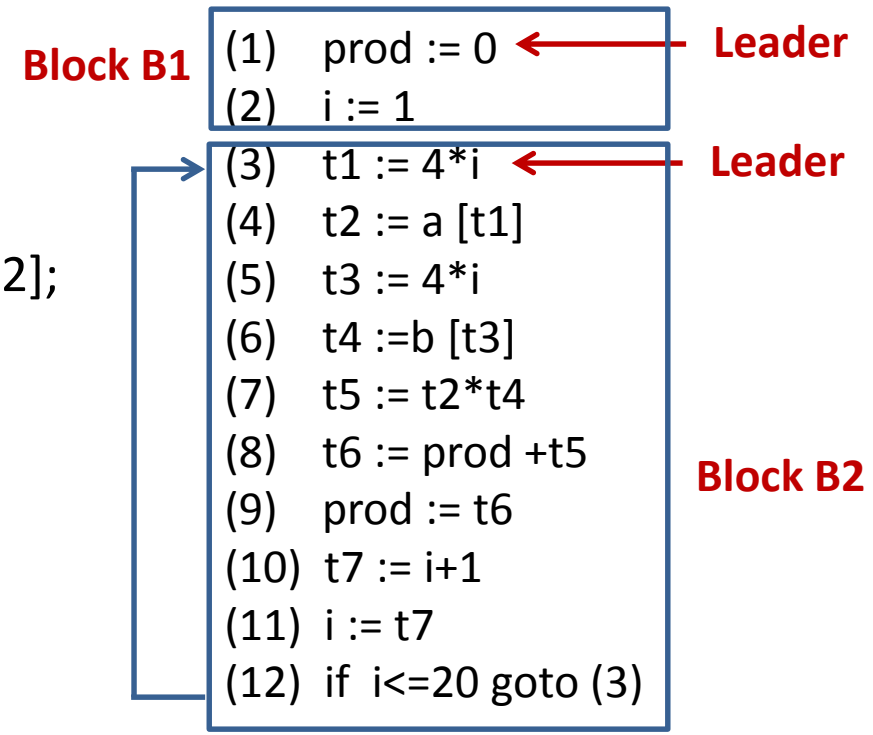
1. *We first determine the set of **leaders**, for that we use the following rules:*
    - I. *The first statement is a leader.*
    - II. *Any statement that is the target of a conditional or unconditional goto is a leader.*
    - III. *Any statement that immediately follows a goto or conditional goto statement is a leader.*
2. *For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.*

Basic block identification

- We may use the following algorithm to find the basic blocks in a program:
- Search header statements of all the basic blocks from where a basic block starts:
  - First statement of a program.
  - Statements that are target of any branch (conditional/unconditional).
  - Statements that follow any branch statement.
- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.
- Basic blocks are important concepts from both code generation and optimization point of view.

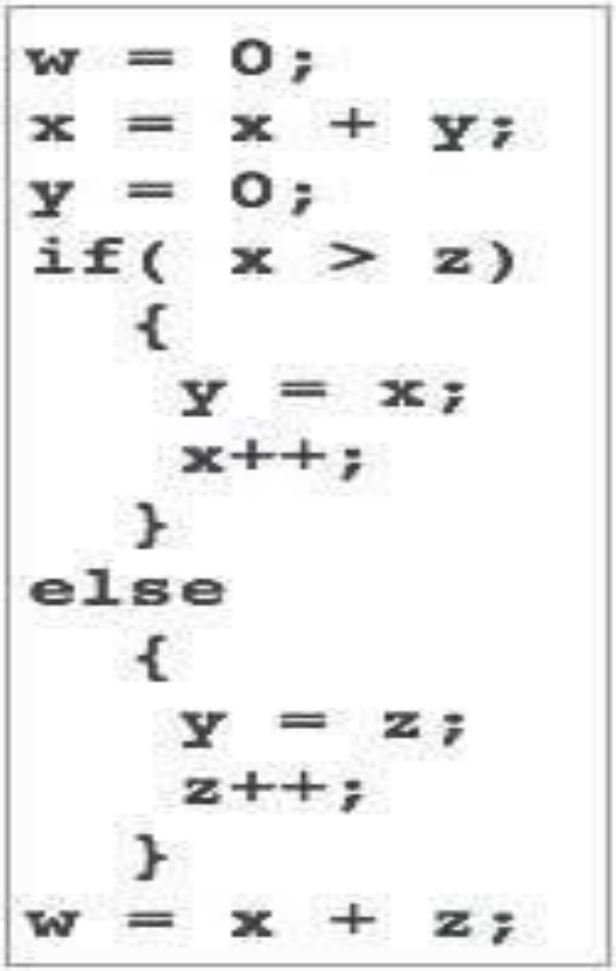
Example: Partition into basic blocks

```
begin
        prod := 0;
              i := 1;
        do
              prod := prod + a[t1] * b[t2];
              i := i+1;
         while  i<= 20
end
```

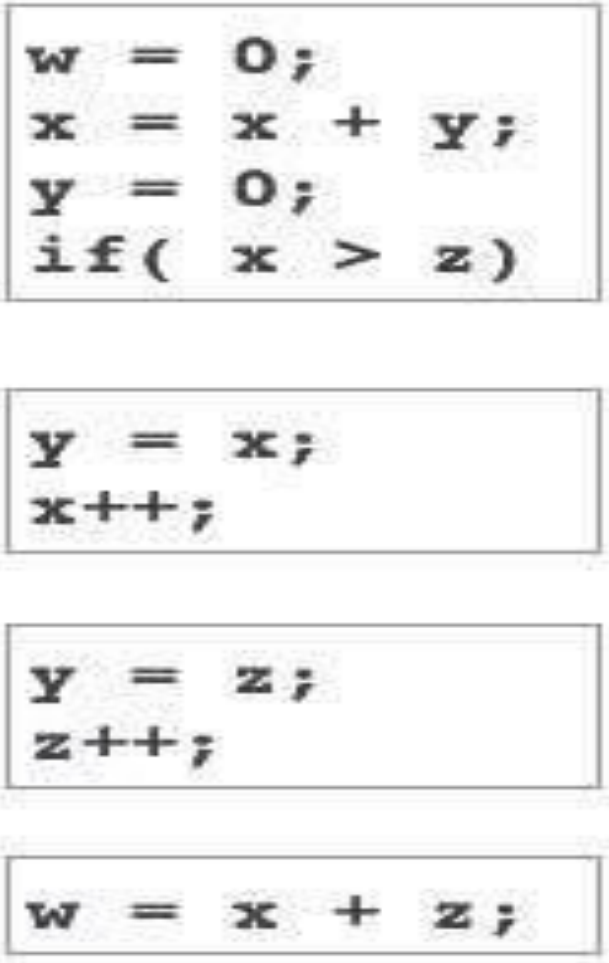


Three Address Code

Conversion from Source code to Basic Block



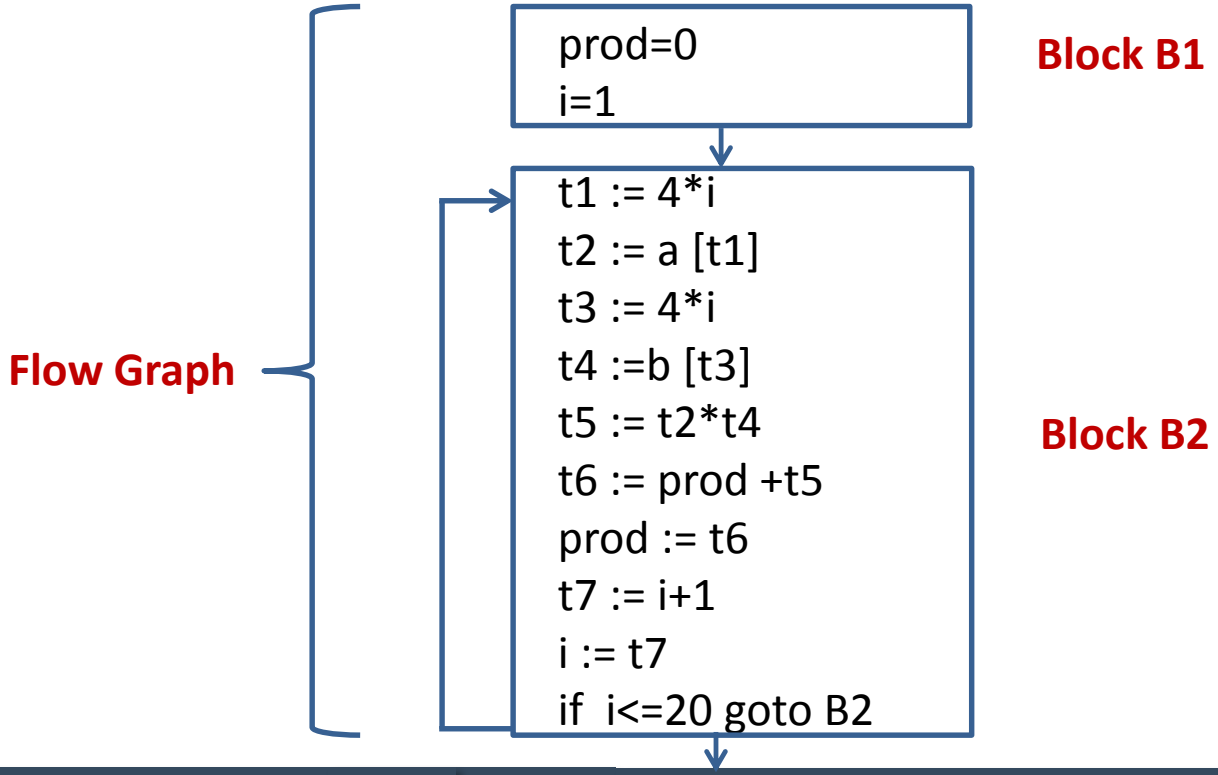
Source Code



Basic Blocks

Flow Graph

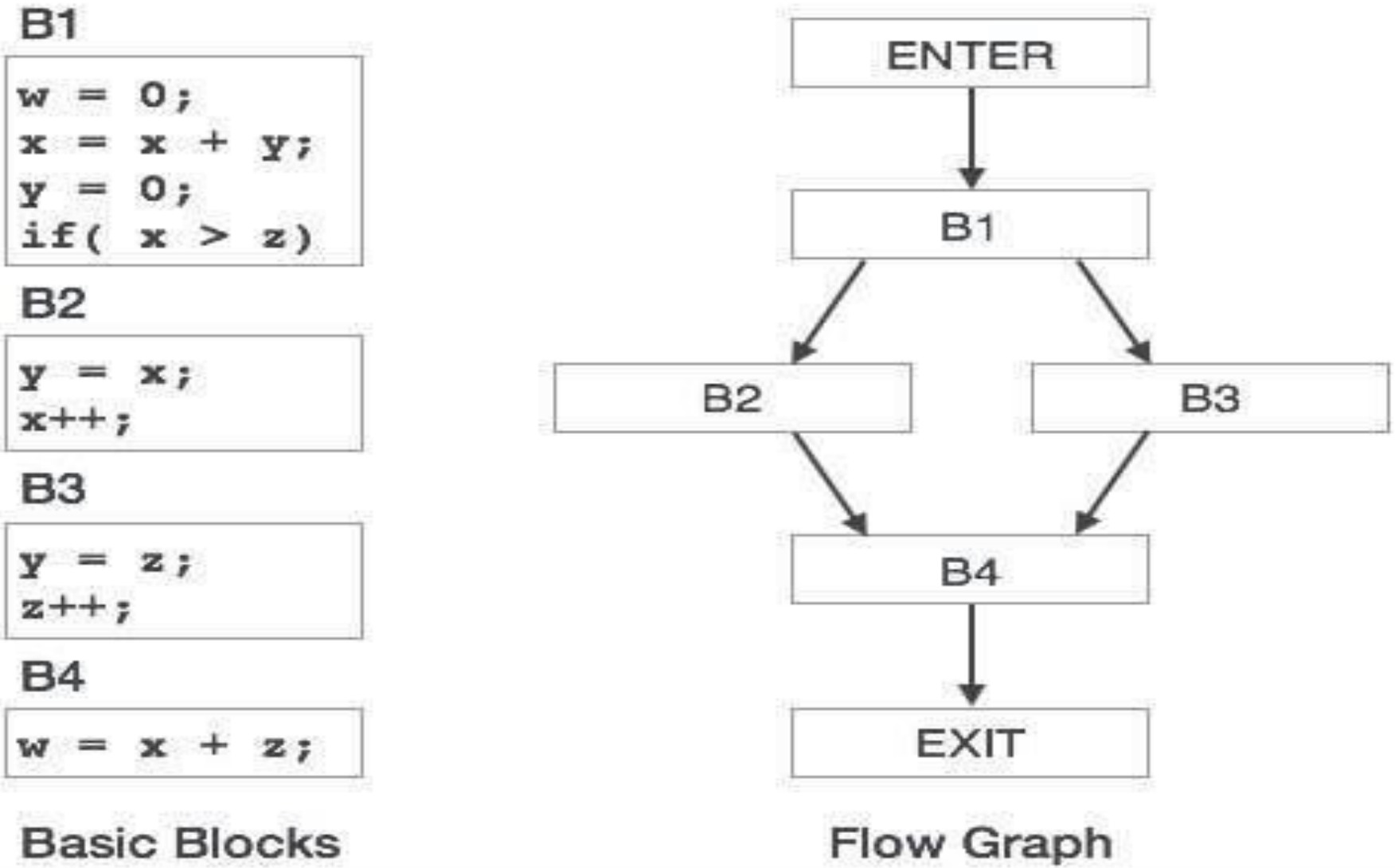
- We can add flow-of-control information to the set of basic blocks making up a program by constructing a direct graph called a flow graph.
- Nodes in the flow graph represent computations, and the edges represent the flow of control.
- Example of flow graph for following three address code:



Conversion from Basic Block to Control Flow

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.

Conversion from Basic Block to Control Flow



Transformation on Basic Blocks

Transformation on Basic Blocks

- A number of transformations can be applied to a basic block without changing the set of expressions computed by the block.
- Many of these transformations are useful for improving the quality of the code.
- Types of transformations are:
    1. Structure preserving transformation
    2. Algebraic transformation

Structure Preserving Transformations

- Structure-preserving transformations on basic blocks are:
  1. Common sub-expression elimination
  2. Dead-code elimination
  3. Renaming of temporary variables
  4. Interchange of two independent adjacent statements

Common sub-expression elimination

- Consider the basic block,

  a:= b+c

  b:= a-d

  c:= b+c

  d:= a-d

- The second and fourth statements compute the same expression, hence this basic block may be transformed into the equivalent block:

  a:= b+c

  b:= a-d

  c:= b+c

  d:= b

Dead-code elimination

- Suppose x is dead, that is, never subsequently used, at the point where the statement $\boldsymbol{x := y + z}$ appears in a basic block.
- Above statement may be safely removed without changing the value of the basic block.

Renaming of temporary variables

- Suppose we have a statement

  t:=b+c, where t is a temporary variable.

- If we change this statement to

  u:= b+c, where u is a new temporary variable,

- Change all uses of this instance of t to u, then the value of the basic block is not changed.
- In fact, we can always transform a basic block into an equivalent block in which each statement that defines a temporary defines a new temporary.
- We call such a basic block a *normal-form* block.

Interchange of two independent adjacent statements

- Suppose we have a block with the two adjacent statements,

  t1:= b+c

  t2:= x+y

- Then we can interchange the two statements without affecting the value of the block if and only if neither x nor y is $t1$ and neither b nor c is $t2$.
- A normal-form basic block permits all statement interchanges that are possible.

Algebraic Transformation

- Countless algebraic transformation can be used to change the set of expressions computed by the basic block into an algebraically equivalent set.
- The useful ones are those that simplify expressions or replace expensive operations by cheaper one.
- Example: x=x+0 or x=x*1 can be eliminated.

Next Use Information

Computing Next Uses

- The 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 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.

Storage for Temporary Names

- For the distinct names each time a temporary is needed. And each time a space gets allocated for each temporary.
- To have optimization in the process of code generation we pack two temporaries into the same location if they are not live simultaneously.
- Consider three address code as,

```
t1=a*a
t2=a*b
t3=4*t2
t4=t1+t3
t5=b*b
t6=t4+t5
```

Register and Address Descriptors

- The code generator algorithm uses descriptors to keep track of register contents and addresses for names.
- **Address descriptor** stores the location where the current value of the name can be found at run time. The information about locations can be stored in the symbol table and is used to access the variables.
- **Register descriptor** is used to keep track of what is currently in each register. The register descriptor shows that initially all the registers are empty. As the generation for the block progresses the registers will hold the values of computation.

Compilation of expressions

Compilation of expressions

- The major issues in code generation for expressions are as follows:
  1. Determination of an evaluation order for the operators in an expression.
  2. Selection of instruction to be used in target code.
  3. Use of registers.

A toy code generator for expressions

- A toy code generator has to track both the registers (for availability) and addresses (location of values) while generating the code.
- For both of them, the following two descriptors are used:
- The code generator uses the notation of an operand descriptor to maintain type, length and addressability information for each operand.
- It uses a register descriptor to maintain information about what operand or partial result would be contained in a CPU register during execution of the generated code.

Operand descriptors

- An operand descriptor has the following fields:
  1. Attributes: Contains the subfields type, and length.
  2. Addressability: Specifies where the operand is located, and how it can be accessed. It has two subfields:
     I. Addressability code: Takes the values 'M' (operand is in memory), and 'R' (operand is in register).
     II. Address: Address of a CPU register or memory word.
- Example:

a*b

MOVER AREG, A

MULT AREG, B

Attribute	Addressability

Operand_descriptor[1]

Operand_descriptor[2]

Operand_descriptor[3]

Register descriptors

- A register descriptor has two fields:
  1. Status: Contains the code free or occupied to indicate register status.
  2. Operand descriptor: If status = occupied, this field contains the descriptor for the operand contained in the register.
- The register descriptor for AREG after generating code for a*b would be:

Status	Operand descriptor #
Occupied	#3

- This indicates that register AREG contains the operand described by descriptor #3.

A Simple Code Generator

• A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.

• **For example: consider the three-address statement a := b+c It can have the following sequence of codes:**

ADD Rj, Ri Cost = 1
(or)
ADD c, Ri Cost = 2
(or)
MOV c, Rj Cost = 3
ADD Rj, Ri

Register and Address Descriptors:

• **A register descriptor** is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.

• **An address descriptor** stores the location where the current value of the name can be found at run time.

A Code-generation Algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block.

For each three-address statement of the form x : = y op z, perform the following actions:

1. Invoke a function getreg to determine the location L where the result of the **computation y op z should be stored.**

2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register.
If the value of y is not already in L, generate the instruction MOV y' , L to place a copy of y in L.

3. Generate the instruction OP z' , L where z' is a current location of z. Prefer a register to a memory location if z is in both.
Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, **after execution of x : = y op z , those registers will no longer contain y or z**

Generating Code for Assignment Statemer

The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:

Code sequence for the example is:

```
t : = a - b
u : = a - c
v:= t+ u
d : = v + u
```

with d live at the end.

Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u : = a - c	MOV a , R1 SUB c , R1	R0 contains t R1 contains u	t in R0 u in R1
v:=t+ u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0

d : = v + u	ADD R1, R0
	
	MOV R0, d
		R0 contains d	d in R0
			d in R0 and memory

Generating Code for Assignment Statements

The assignment statement d:= (a-b) + (a-c) + (a-c)

can be translated into the following sequence of three address code:

t:= a-b

u:= a-c

v:= t +u

d:= v+u

Statement	Code Generated	Register descriptor Register empty	Address descriptor
t:= a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u:= a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
v:= t + u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R1
d:= v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed **assignment a:= b[i] and a[i]:= b**

Statements	Code Generated	Cost
a : = b[i]	MOV b(Ri), R	2
a[i] : = b	MOV b, a(Ri)	3

Generating Code for Pointer Assignments & Conditional Statement

- The table shows the code sequences generated for the pointer assignments a : = *p and *p : = a

Statements	Code Generated	Cost
a : = *p	MOV *Rp, a	2
*p : = a	MOV a, *Rp	2

Generating Code for Conditional Statements

Statement	Code
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */
x : = y +z	MOV y, R0

Register Allocation & Assignment

Register Allocation & Assignment

- Efficient utilization of registers is important in generating good code.
- There are four strategies for deciding what values in a program should reside in a registers and which register each value should reside.
- Strategies are:
    1. Global Register Allocation
    2. Usage Count
    3. Register assignment for outer loop
    4. Register allocation for graph coloring

Global Register Allocation

- Global register allocation strategies are:
- The global register allocation has a strategy of storing 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 are not already allocated may be used to hold values local to one block.
- In certain languages like C or Bliss programmer can do the register allocation by using register declaration to keep certain values in register for the duration of the procedure.
- Example:

```
{
  register int  x;
}
```

Usage count

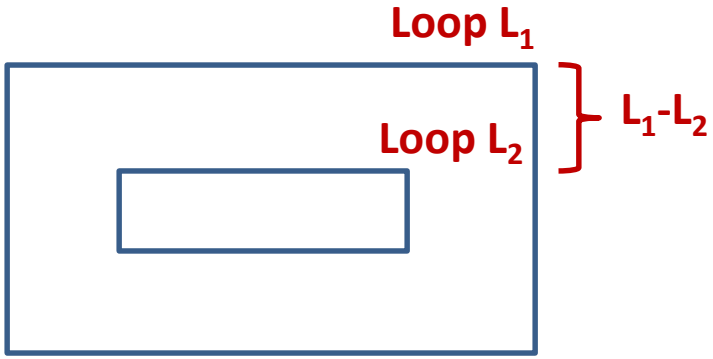
- The usage count is the count for the use of some variable x in some register used in any basic block.
- The usage count gives the idea about how many units of cost can be saved by selecting a specific variable for global register allocation.
- The approximate formula for usage count for the Loop L in some basic block B can be given as,

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

- Where $use(x, B)$ is number 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 $live(x) = 0$.

Register assignment for outer loop

- Consider that there are two loops $L1$ is outer loop and $L2$ is an inner loop, and allocation of variable a is to be done to some register.



- Following criteria should be adopted for register assignment for outer loop,
- If a is allocated in loop $L2$ then it should not be allocated in $L1 - L2$.
- If a is allocated in $L1$ and it is not allocated in $L2$ then store a on entrance to $L2$ and load a while leaving $L2$.
- If a is allocated in $L2$ and not in $L1$ then load a on entrance of $L2$ and store a on exit from $L2$.

Register allocation for graph coloring

- The graph coloring works in two passes. The working is as given below:
- In the first pass the specific machine instruction is selected for register allocation. For each variable a symbolic register is allocated.
- In the second pass the register inference graph is prepared.
- In register inference graph each node is a symbolic registers and an edge connects two nodes where one is live at a point where other is defined.
- Then a graph coloring technique is applied for this register inference graph using k-color.
- The k-colors can be assumed to be number of assignable registers.
- In graph coloring technique no two adjacent nodes can have same color. Hence in register inference graph using such graph coloring principle each node is assigned the symbolic registers so that no two symbolic registers can interfere with each other with assigned physical registers.

DAG Representation of Basic Block

- DAG is a very useful data structure for implementing transformations on **Basic Blocks**
- A DAG is constructed for optimizing the basic block.
- A DAG is usually constructed using **Three Address Code**
- Transformations such as dead code elimination and common sub expression elimination are then applied.

Properties of DAG

- Reachability relation forms a partial order in DAGs.
- Both transitive closure & transitive reduction are uniquely defined for DAGs.
- **Topological Orderings** are defined for DAGs

Construction of DAGs

- Following rules are used for the construction of DAGs-
- **Rule-01:**
  - In a DAG,
  - Interior nodes always represent the operators.
  - Exterior nodes (leaf nodes) always represent the names, identifiers or constants.
- **Rule-02:**
  - While constructing a DAG,
  - A check is made to find if there exists any node with the same value.
  - A new node is created only when there does not exist any node with the same value.
  - This action helps in detecting the common sub-expressions and avoiding the re-computation of the same.
- **Rule-03:**
  - The assignment instructions of the form x:=y are not performed unless they are necessary.

Algorithm: DAG Construction

We assume the three address statement could of following types:

Case (i) x:=y op z

Case (ii) x:=op y

Case (iii) x:=y

With the help of following steps the DAG can be constructed.

- Step 1: If y is undefined then create node(y). Similarly if z is undefined create a node (z)
- Step 2:

  Case(i) create a node(op) whose left child is node(y) and node(z) will be the right child. Also check for any common sub expressions.

  Case (ii) determine whether is a node labeled op, such node will have a child node(y).

  Case (iii) node n win be node(y).
- Step 3: Delete x from list of identifiers for node(x). Append x to the list of attached identifiers for node n found in 2.

PRACTICE PROBLEMS BASED ON DIRECTED ACYCLIC GRAPHS Problem-01:

Consider the following expression and construct a DAG for it-

(a + b) x (a + b + c)

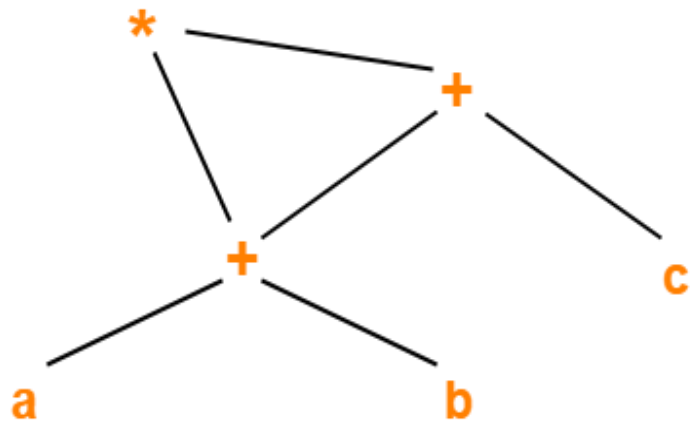
Solution-

Three Address Code for the given expression is-

T1 = a + b

T2 = T1 + c

T3 = T1 x T2



Directed Acyclic Graph

From the constructed DAG, we observe-
The common sub-expression (a+b) has been expressed into a single node in the DAG.
The computation is carried out only once and stored in the identifier T1 and reused later.

This illustrates how the construction scheme of a DAG identifies the common sub-expression and helps in eliminating its re-computation later.

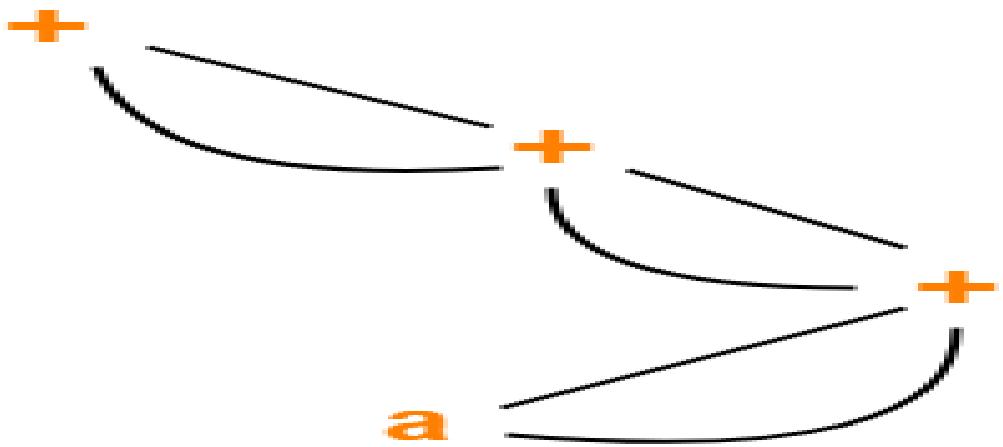
PRACTICE PROBLEMS BASED ON DIRECTED ACYCLIC GRAPHS Problem-02:

Consider the following expression and construct a DAG for it-

(((a + a) + (a + a)) + ((a + a) + (a + a)))

Solution-

Directed Acyclic Graph for the given expression is-



Directed Acyclic Graph

PRACTICE PROBLEMS BASED ON DIRECTED ACYCLIC GRAPHS Problem-03:

Consider the following block and construct a DAG for it-

(1) a = d x c

(2) d = b

(3) e = d x c

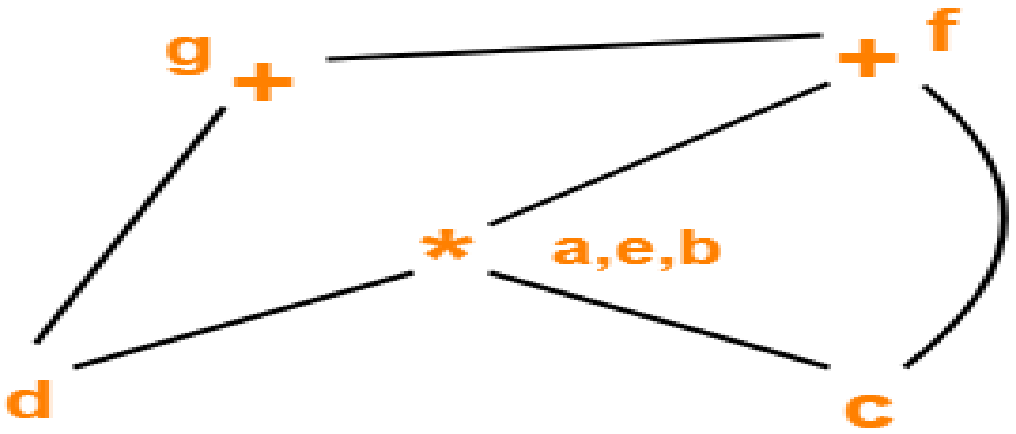
(4) b = e

(5) f = b + c

(6) g = f + d

Solution-

Directed Acyclic Graph for the given block is-

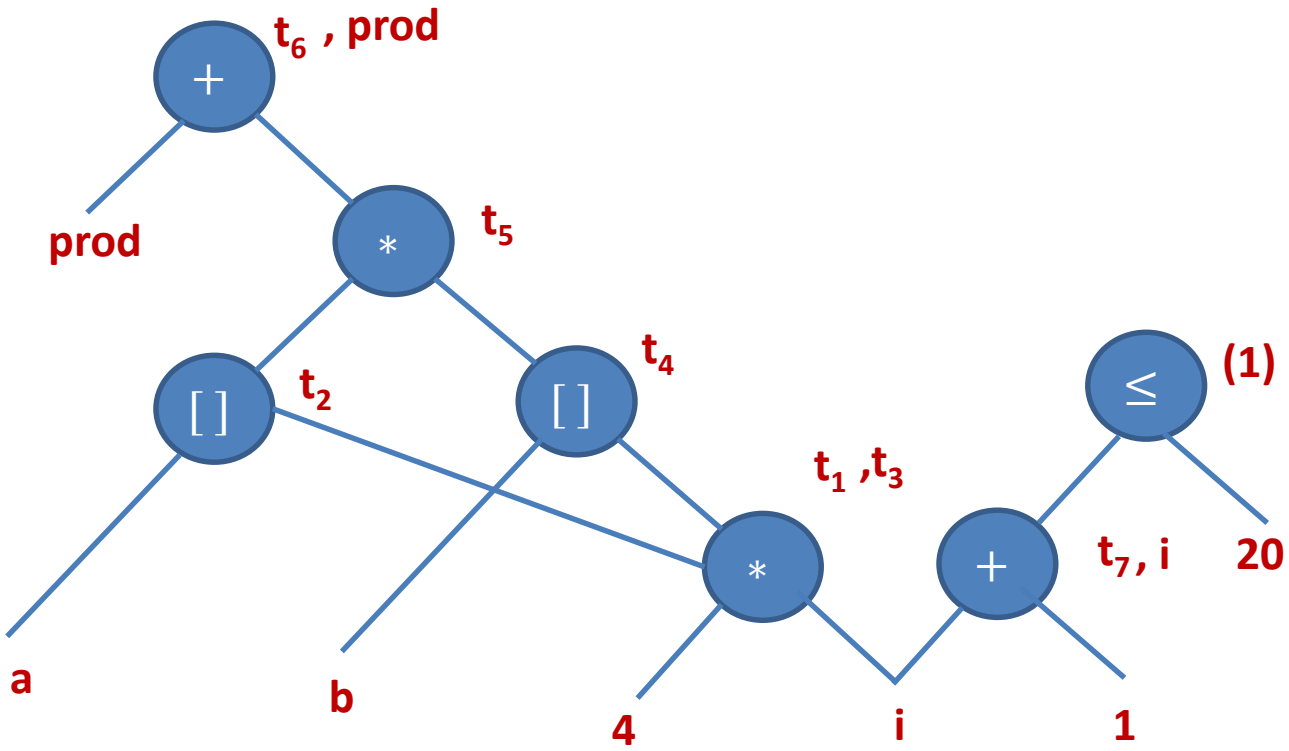


Directed Acyclic Graph

PRACTICE PROBLEMS BASED ON DIRECTED ACYCLIC GRAPHS Problem-04:

Example:

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



PRACTICE PROBLEMS BASED ON DIRECTED ACYCLIC GRAPHS Problem-05:

Consider the following basic block-

B10:

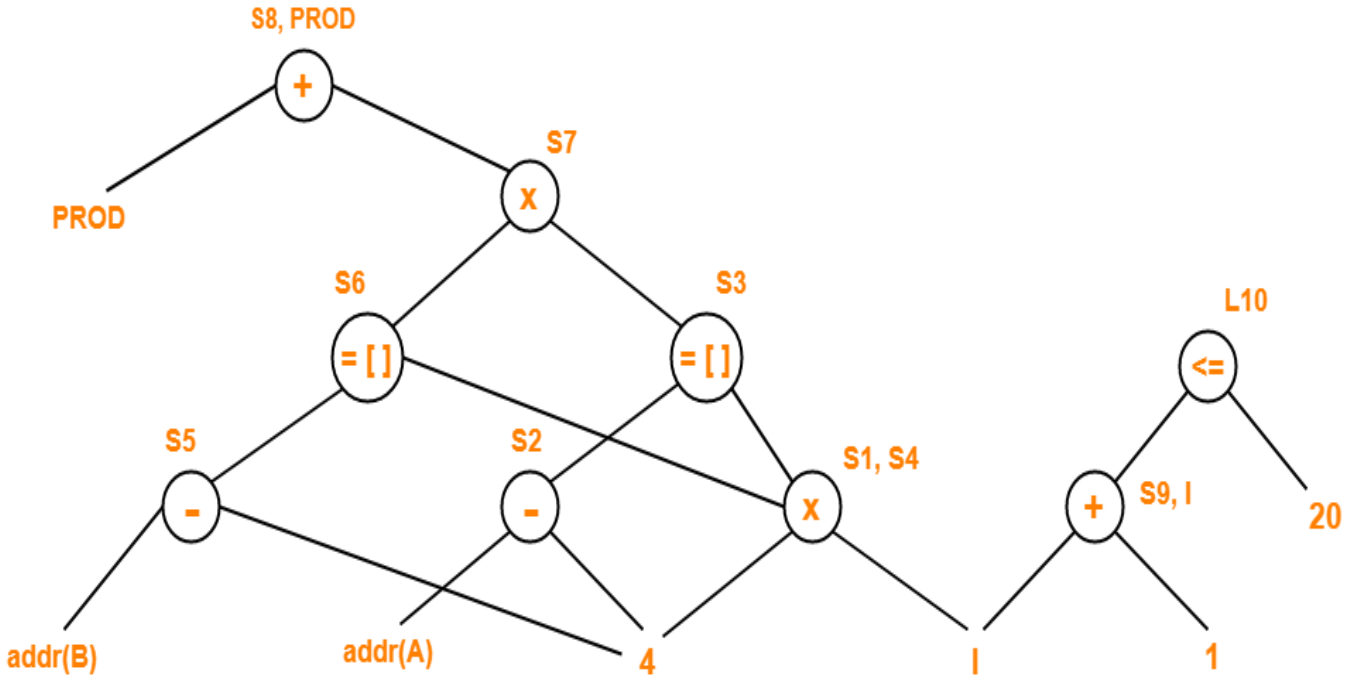
S1 = 4 x I
S2 = addr(A) – 4
S3 = S2[S1]
S4 = 4 x I
S5 = addr(B) – 4
S6 = S5[S4]
S7 = S3 x S6
S8 = PROD + S7
PROD = S8
S9 = I + 1
I = S9
If I <= 20 goto L10

A. Draw a directed acyclic graph and identify local common sub-expressions.

B. After eliminating the common sub-expressions, re-write the basic block.

Solution-

Directed Acyclic Graph for the given basic block is-



Directed Acyclic Graph

PRACTICE PROBLEMS BASED ON DIRECTED ACYCLIC GRAPHS Problem-05: Continued

In this code fragment,

- **4 x I is a common sub-expression**. Hence, we can eliminate because **S1 = S4.**
- We can optimize **S8 = PROD + S7 and PROD = S8 as PROD = PROD + S7.**
- We can optimize **S9 = I + 1 and I = S9 as I = I + 1.**
- After eliminating **S4, S8 and S9,** we get the following basic block-

B10:

S1 = 4 x I

S2 = addr(A) – 4

S3 = S2[S1]

S5 = addr(B) – 4

S6 = S5[S1]

S7 = S3 x S6

PROD = PROD + S7

I = I + 1

If I <= 20 goto L10

Applications of DAG

- The DAGs are used in following:
  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 the common sub-expressions and not performing the assignment of the form x:=y unless and until it is a must.

Generation of Code from DAGs

Generation of Code from DAGs

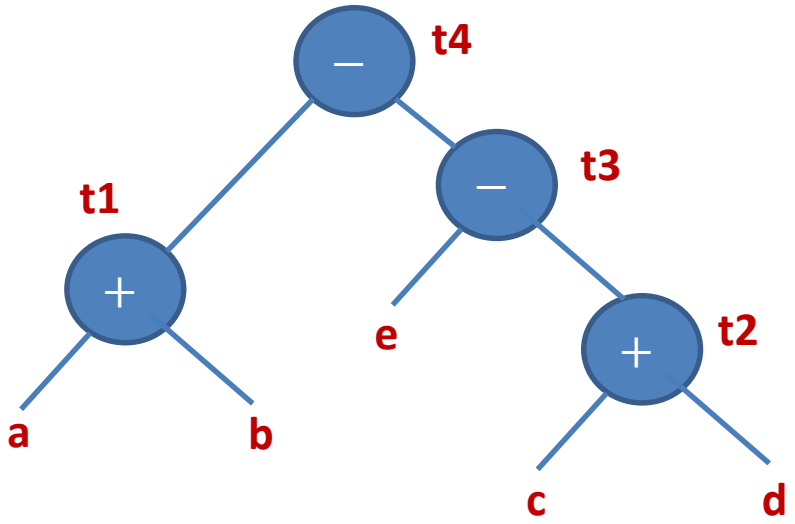
- Methods generating code from DAGs are:
  1. Rearranging Order
  2. Heuristic ordering
  3. Labeling algorithm

Rearranging Order

- The order of three address 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.
- Example:

t1:=a+b
t2:=c+d
t3:=e-t2
t4:=t1-t3

Three Address Code



Example: Rearranging Order

t1:=a+b
t2:=c+d
t3:=e-t2
t4:=t1-t3

Three Address Code

Re-arrange →

t2:=c+d
t3:=e-t2
t1:=a+b
t4:=t1-t3

Three Address Code

MOV a, R0
ADD b, R0
MOV c, R1
ADD d, R1
MOV R0, t1
MOV e, R0
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4

Assembly Code

MOV c, R0
ADD d, R0
MOV e, R1
SUB R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
MOV R0, t4

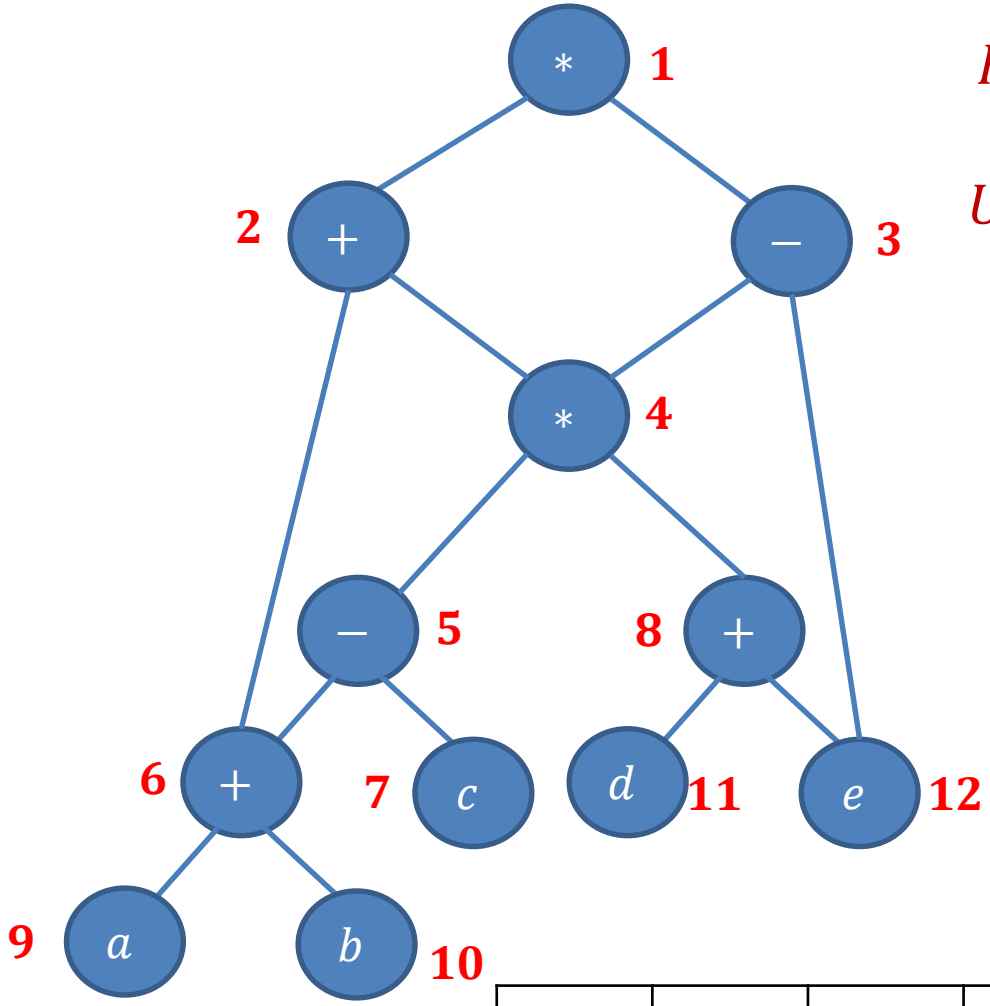
Assembly Code

Algorithm: Heuristic Ordering

Obtain all the interior nodes. Consider these interior nodes as unlisted nodes.

```
while(unlisted interior nodes remain)
    {
        pick up an unlisted node n, whose parents have been listed
        list n;
        while(the leftmost child m of n has no unlisted parent AND is not leaf)
                        {
                        List m;
                        n=m;
                        }
    }
```

Example: Heuristic Ordering



$$Interior\ nodes = 1\ 2\ 3\ 4\ 5\ 6\ 8$$

$$Unlisted\ nodes = \cancel{1}\ \cancel{2}\ 3\ 4\ 5\ 6\ 8$$

Pick up an unlisted node, whose parents have been listed

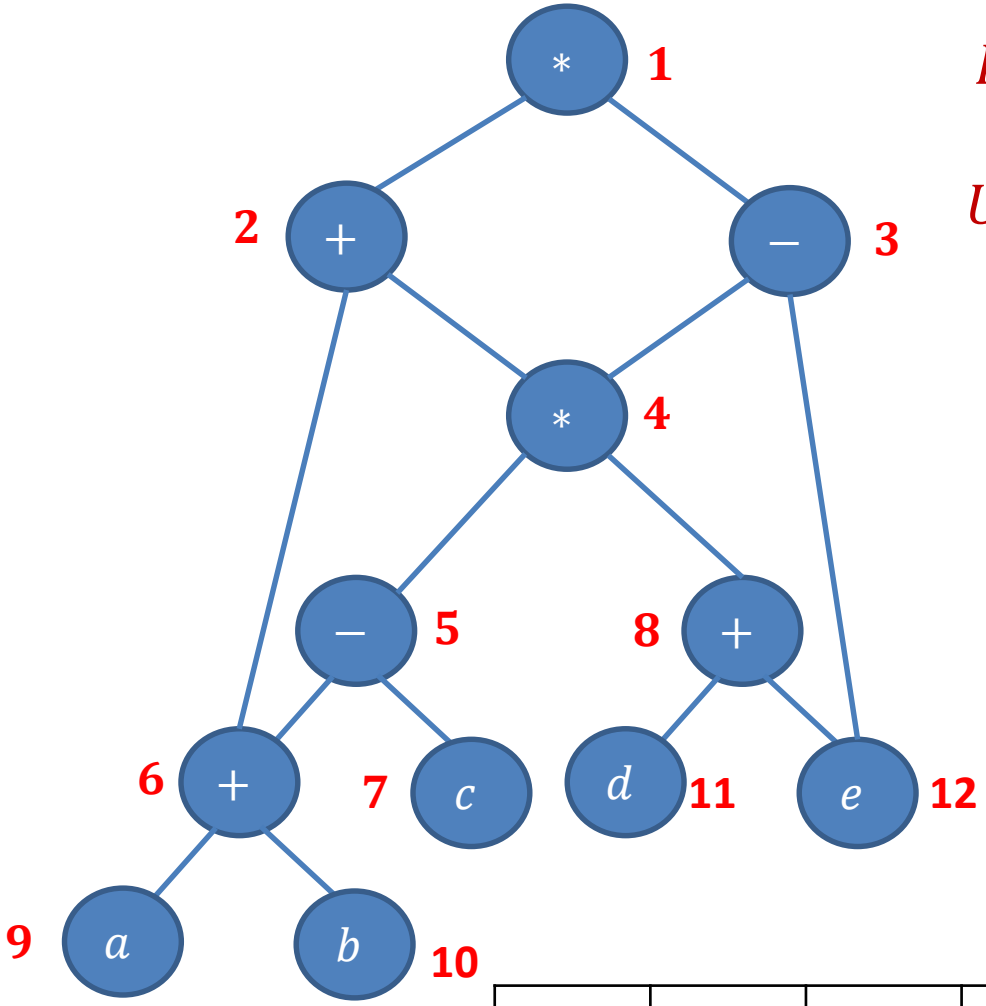
1

Leftchild of 1 = 2

Parent 5 is not listed so list 2 list 6

Listed Node							

Example: Heuristic Ordering



$$Interior\ nodes = 1\ 2\ 3\ 4\ 5\ 6\ 8$$

$$Unlisted\ nodes = 1\ 2\ 3\ 4\ 5\ 6\ 8$$

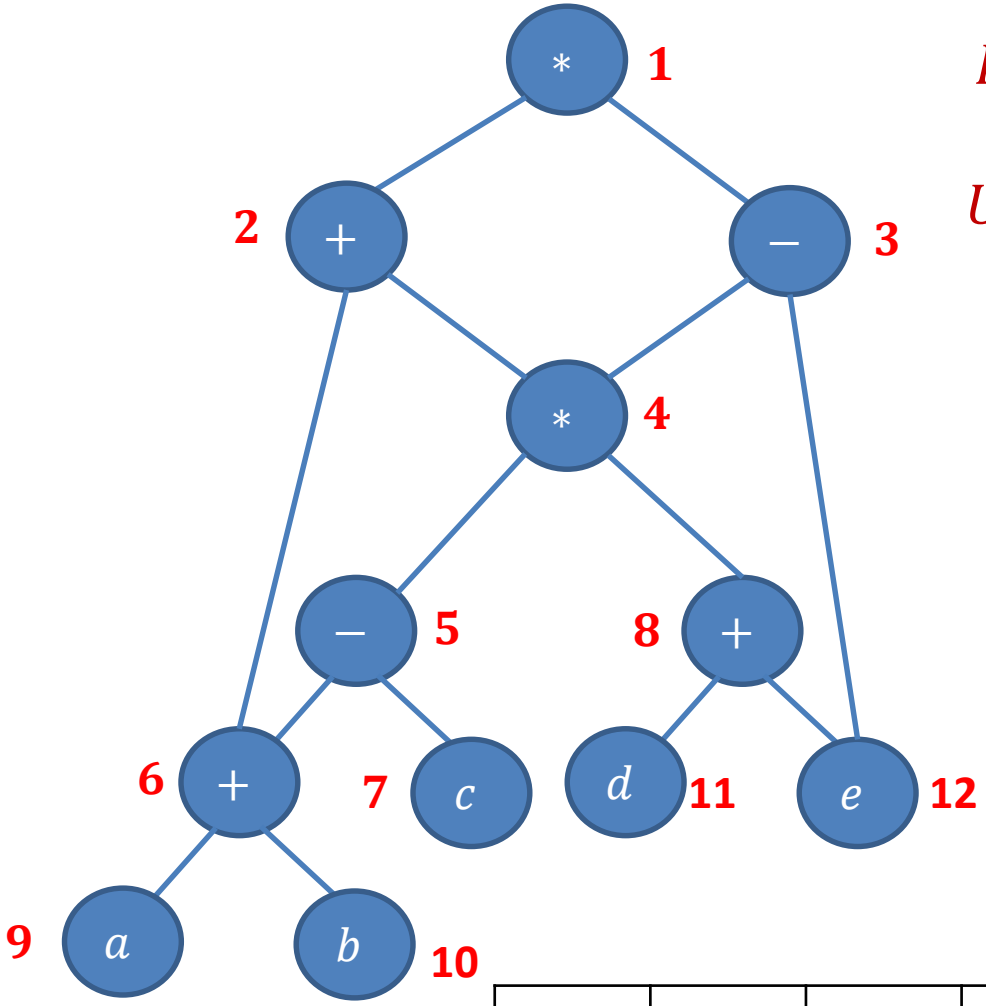
Pick up an unlisted node, whose parents have been listed

Right child of 3 = 4

Parent 2, 3 already listed in List

Listed Node	1	2					

Example: Heuristic Ordering



$$Interior\ nodes = 1\ 2\ 3\ 4\ 5\ 6\ 8$$

$$Unlisted\ nodes = \cancel{1}\ \cancel{2}\ \cancel{3}\ \cancel{4}\ \cancel{5}\ \cancel{6}\ 8$$

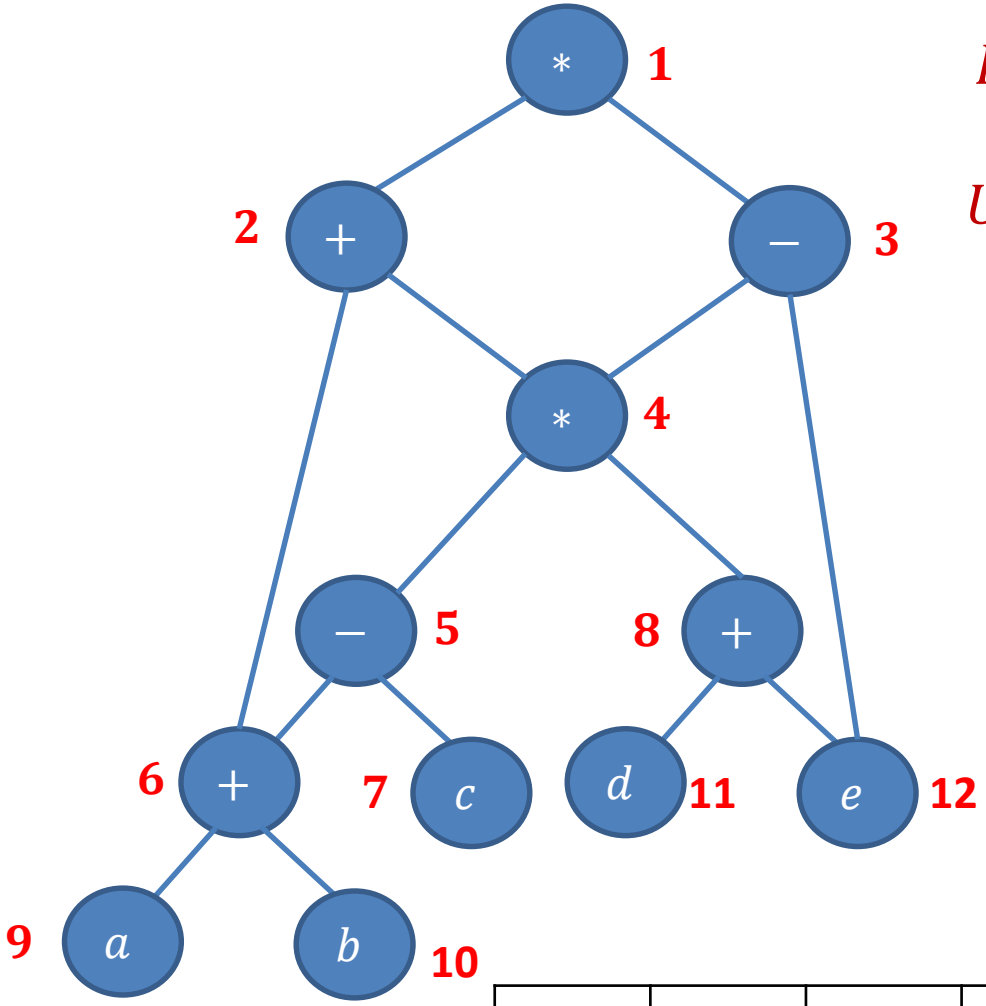
Pick up an unlisted node, whose parents have been listed

Leftchild of 4 = 6

Parent 2,4 is listed so list 6

Listed Node	1	2	3	4			

Example: Heuristic Ordering



$$Interior\ nodes = 1\ 2\ 3\ 4\ 5\ 6\ 8$$

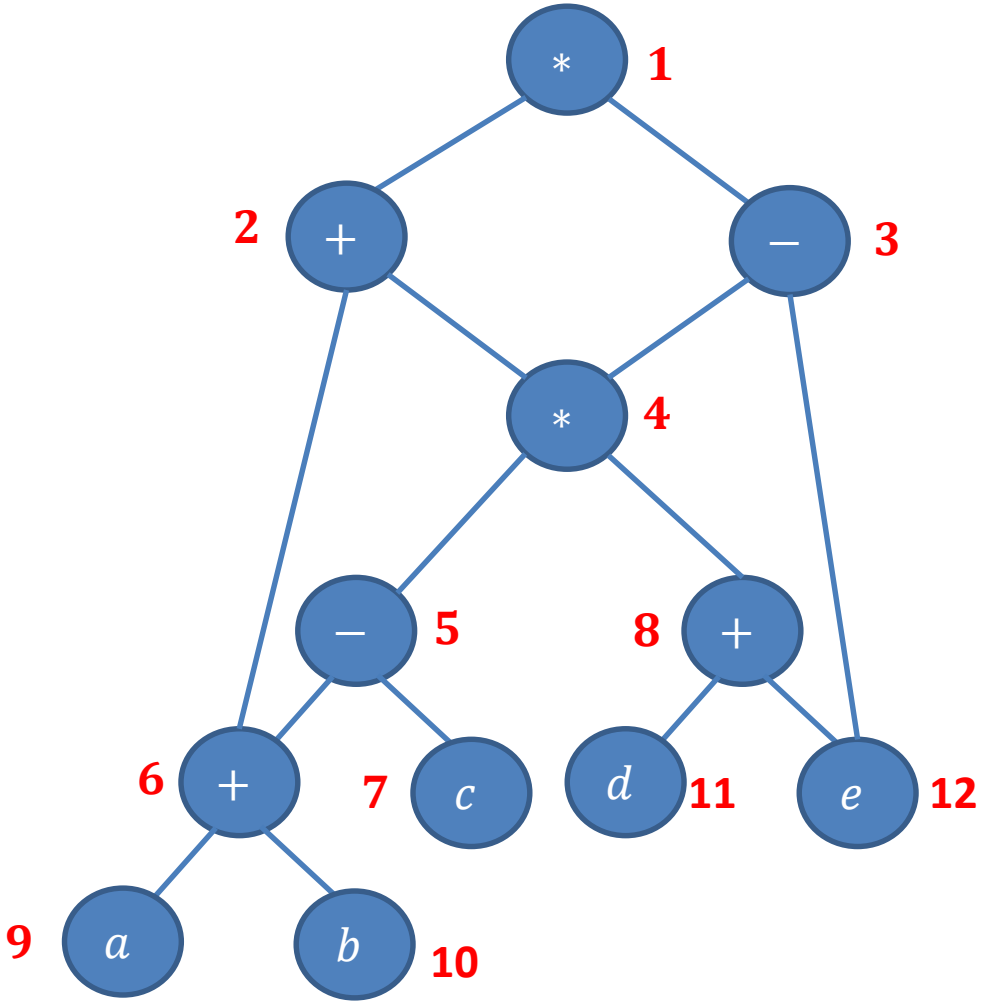
$$Unlisted\ nodes = \cancel{1\ 2\ 3\ 4\ 5\ 6\ 8}$$

Pick up an unlisted node, whose parents have been listed

Rightchild of 4 = 8
Parent 4 is listed so list 8

Listed Node	1	2	3	4	5	6	

Example: Heuristic Ordering



Listed Node	1	2	3	4	5	6	8

Reverse Order for three address code = 8 6 5 4 3 2 1

t8=d+e
t6=a+b
t5=t6-c
t4=t5*t8
t3=t4-e
t2=t6+t4
t1=t2*t3

Optimal Three Address code

Labeling Algorithm

- The labeling algorithm generates the optimal code for given expression in which minimum registers are required.
- Using labeling algorithm the labeling can be done to tree by visiting nodes in bottom up order.
- For computing the label at node n with the label L1 to left child and label L2 to right child as,

$$Label\ (n)\ =\ \max(L1, L2)\ if\ L1\ not\ equal\ to\ L2$$
$$Label(n)\ =\ L1 + 1\ if\ L1 = L2$$

- We start in bottom-up fashion and label left leaf as 1 and right leaf as 0.

Example: Labeling Algorithm

Example:

t1:=a+b

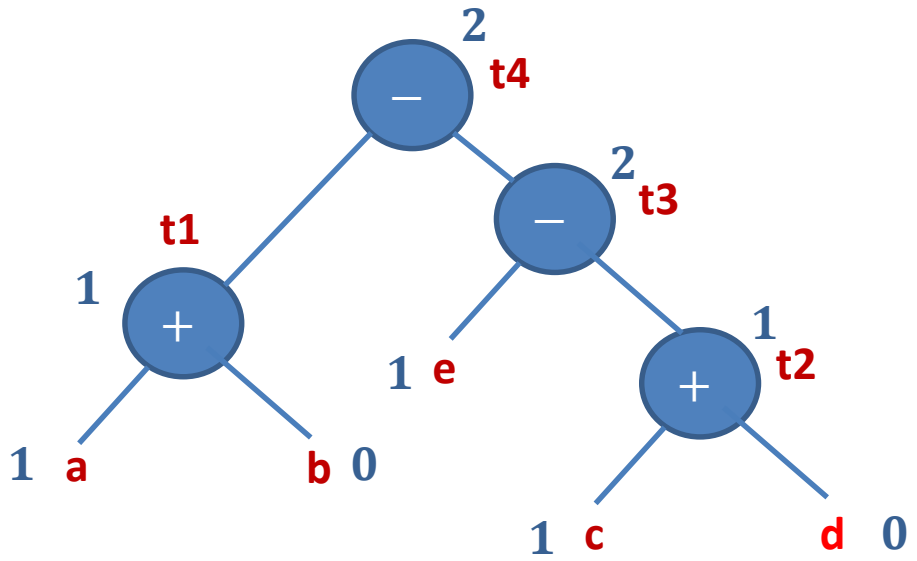
t2:=c+d

t3:=e-t2

t4:=t1-t3

Three Address Code

$$Label(n) = \begin{cases} Max(l1, l2) & if\ l1 \neq l2 \\ L1 + 1 & if\ l1 = l2 \end{cases}$$



$$postorder\ traversal = a\ b\ t1\ e\ c\ d\ t2\ t3\ t4$$

End of Unit-5(Part-1)