# Run-time Environments - 4

Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

# Outline of the Lecture

- What is run-time support? (in part 1)
- Parameter passing methods (in part 1)
- Storage allocation (in part 2)
- Activation records (in part 2)
- Static scope and dynamic scope (in part 3)
- Passing functions as parameters (in part 3)
- Heap memory management (in part 3)
- Garbage Collection

# Problems with Manual Deallocation

- Memory leaks
  - Failing to delete data that cannot be referenced
  - Important in long running or nonstop programs
- Dangling pointer dereferencing
  - Referencing deleted data
- Both are serious and hard to debug
- Solution: automatic garbage collection

# Garbage Collection

- Reclamation of chunks of storage holding objects that can no longer be accessed by a program
- GC should be able to determine types of objects
  - Then, size and pointer fields of objects can be determined by the GC
  - Languages in which types of objects can be determined at compile time or run-time are type safe
    - Java is type safe
    - C and C++ are not type safe because they permit type casting, which creates new pointers
    - Thus, any memory location can be (theoretically) accessed at any time and hence cannot be considered inaccessible

# Reachability of Objects

- The *root set* is all the data that can be accessed (reached) directly by a program without having to dereference any pointer
- Recursively, any object whose reference is stored in a field of a member of the root set is also reachable
- New objects are introduced through object allocations and add to the set of reachable objects
- Parameter passing and assignments can propagate reachability
- Assignments and ends of procedures can terminate reachability

# Reachability of Objects

- Similarly, an object that becomes *unreachable* can cause more objects to become unreachable
- A garbage collector periodically finds all unreachable objects by one of the two methods
  - Catch the transitions as reachable objects become unreachable
  - Or, periodically locate all reachable objects and infer that all *other* objects are unreachable

# Reference Counting Garbage Collector

- This is an approximation to the first approach mentioned before
- We maintain a count of the references to an object, as the mutator (program) performs actions that may change the reachability set
- When the count becomes zero, the object becomes unreachable
- Reference count requires an extra field in the object and is maintained as below

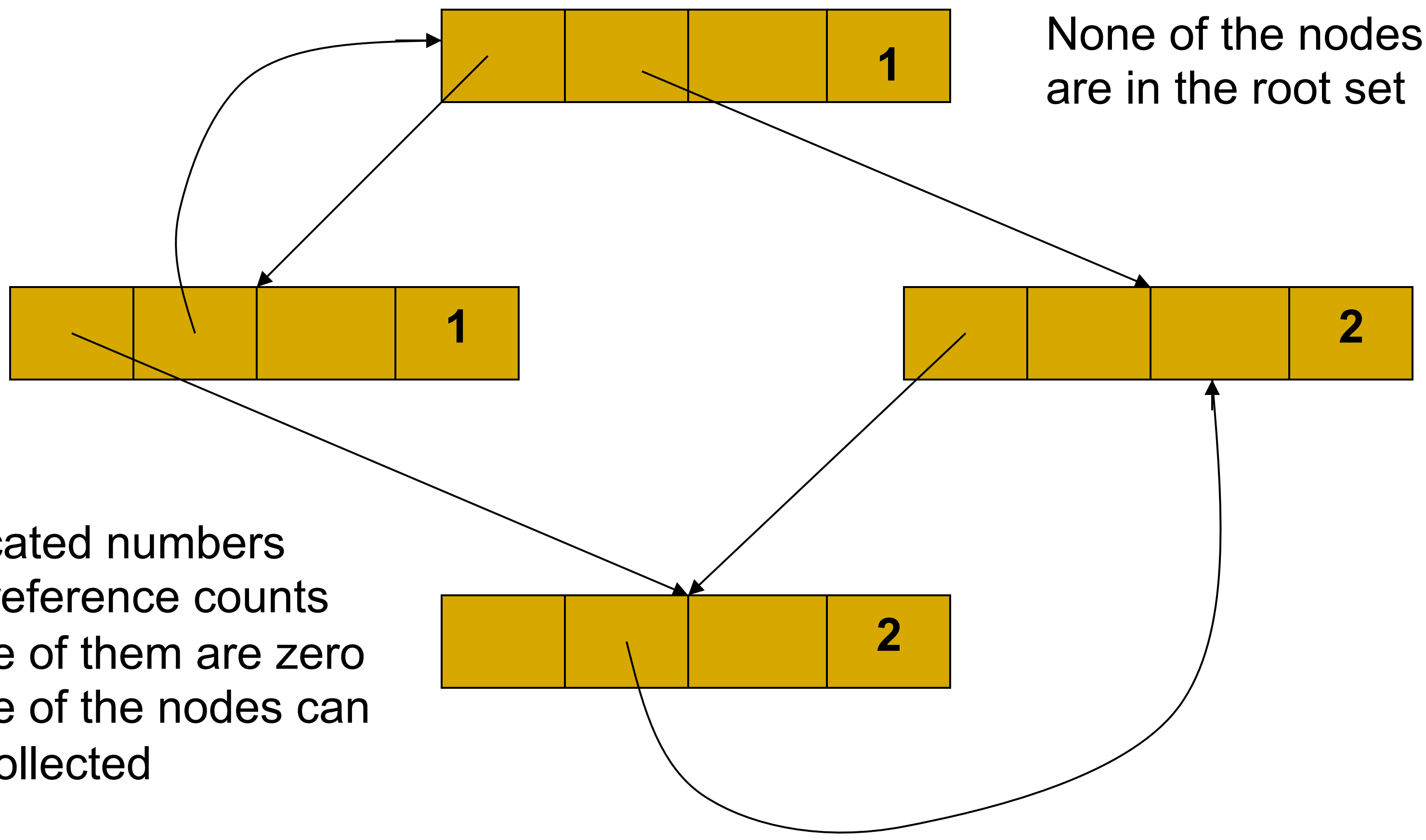
# Maintaining Reference Counts

- *New object allocation.* ref_count=1 for the new object
- *Parameter passing.* ref_count++ for each object passed into a procedure
- *Reference assignments.* For u:=v, where u and v are references, ref_count++ for the object *v, and ref_count-- for the object *u
- *Procedure returns.* ref_count-- for each object pointed to by the local variables
- *Transitive loss of reachability.* Whenever ref_count of an object becomes zero, we must also decrement the ref_count of each object pointed to by a reference within the object

# Reference Counting GC: Disadvantages and Advantages

- High overhead due to reference maintenance
- Cannot collect unreachable cyclic data structures (ex: circularly linked lists), since the reference counts never become zero
- Garbage collection is incremental
  - overheads are distributed to the mutator's operations and are spread out throughout the life time of the mutator
- Garbage is collected immediately and hence space usage is low
- Useful for real-time and interactive applications, where long and sudden pauses are unacceptable

# Unreachable Cyclic Data Structure




None of the nodes are in the root set

Indicated numbers are reference counts
None of them are zero
None of the nodes can be collected

# Mark-and-Sweep Garbage Collector

- Memory recycling steps
  - Program runs and requests memory allocations
  - GC traces and finds reachable objects
  - GC reclaims storage from unreachable objects
- Two phases
  - Marking reachable objects
  - Sweeping to reclaim storage
- Can reclaim unreachable cyclic data structures
- Stop-the-world algorithm

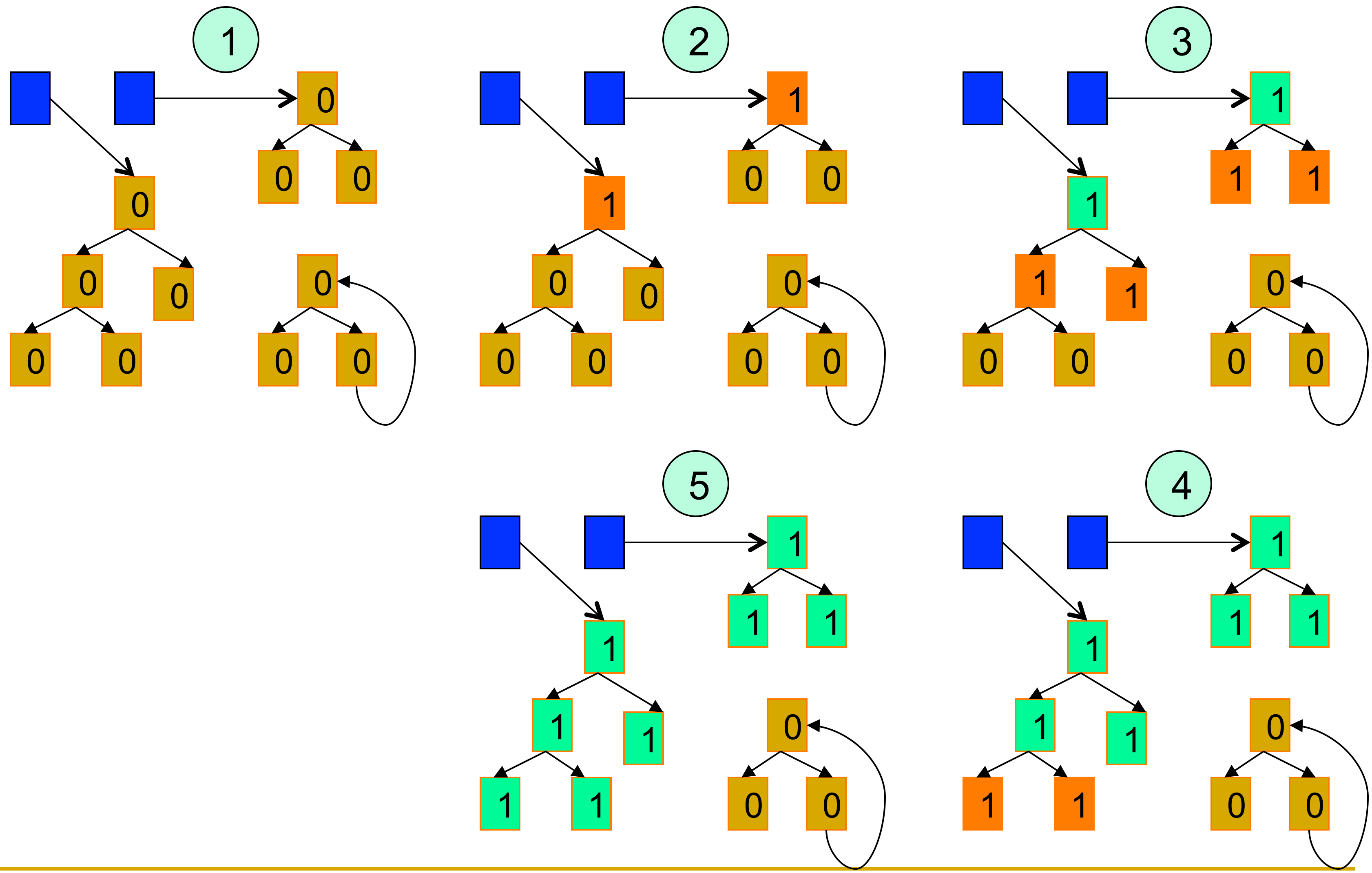
# Mark-and-Sweep Algorithm - Mark

/* marking phase */

1. Start scanning from root set, mark all reachable objects (set reached-bit = 1), place them on the list Unscanned
2. while (Unscanned ≠ Φ) do

```
{ object o = delete(Unscanned);
  for (each object o1 referenced in o) do
    { if (reached-bit(o1) == 0)
        { reached-bit(o1) = 1; place o1 on Unscanned;}
    }
}
```

# Mark-and-Sweep GC Example - Mark



# Mark-and-Sweep Algorithm - Sweep

- /* Sweeping phase, each object in the heap is inspected only once */

3. Free = Φ;

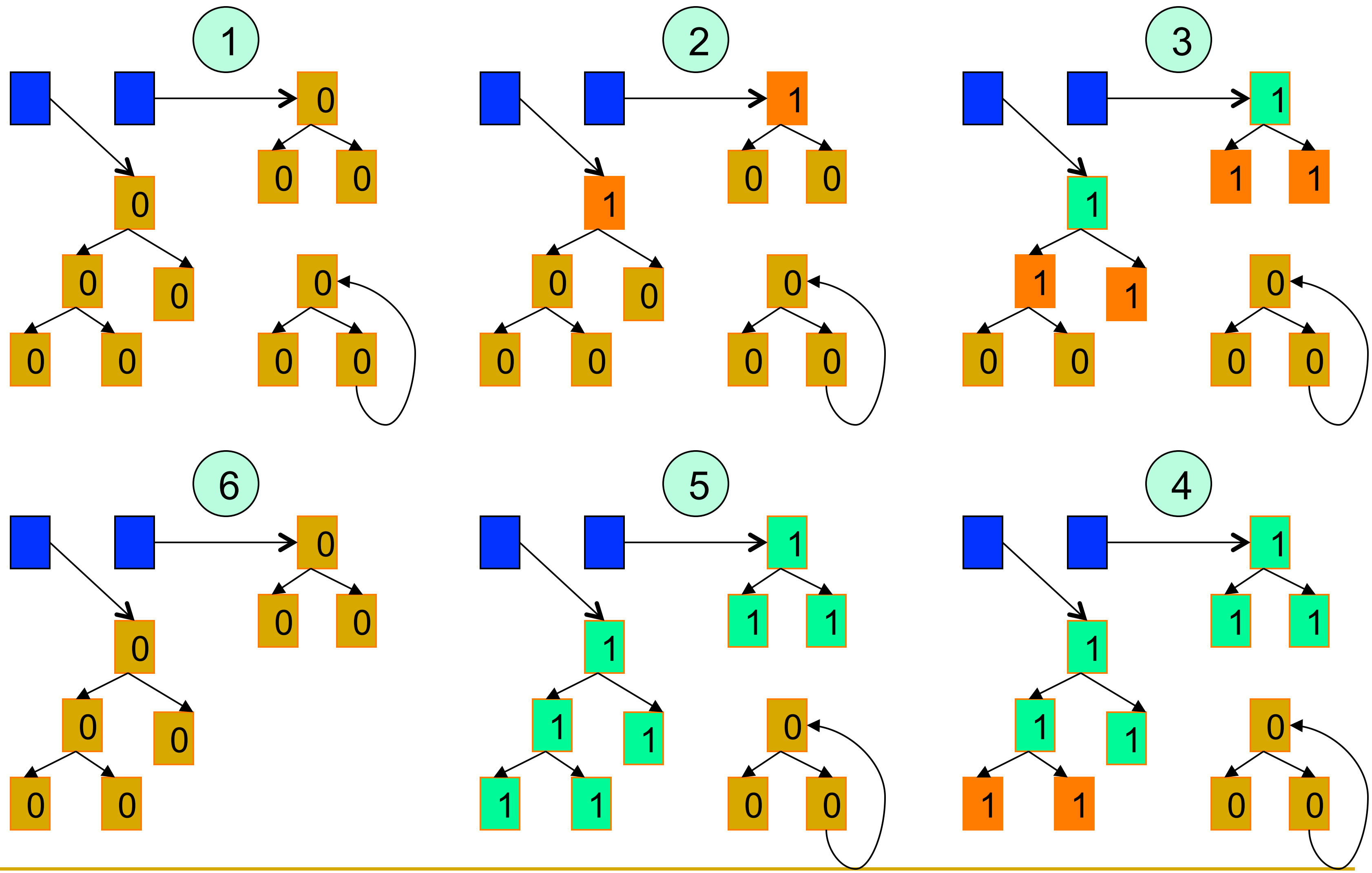
   for (each object o in the heap) do

   { if (reached-bit(o) == 0)    add(Free, o);

   else reached-bit(o) = 0;

   }

# Mark-and-Sweep GC Example - Sweep



# Control-Flow Graph and Local Optimizations - Part 1

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

# Outline of the Lecture

- What is code optimization and why is it needed?
- Types of optimizations
- Basic blocks and control flow graphs
- Local optimizations
- Building a control flow graph
- Directed acyclic graphs and value numbering

# Machine-independent Code Optimization

- Intermediate code generation process introduces many inefficiencies
  - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
- Code optimization removes such inefficiencies and improves code
- Improvement may be time, space, or power consumption
- It changes the structure of programs, sometimes of beyond recognition
  - Inlines functions, unrolls loops, eliminates some programmer-defined variables, etc.
- Code optimization consists of a bunch of heuristics and percentage of improvement depends on programs (may be zero also)
- Optimizations may be classified as *local* and *global*

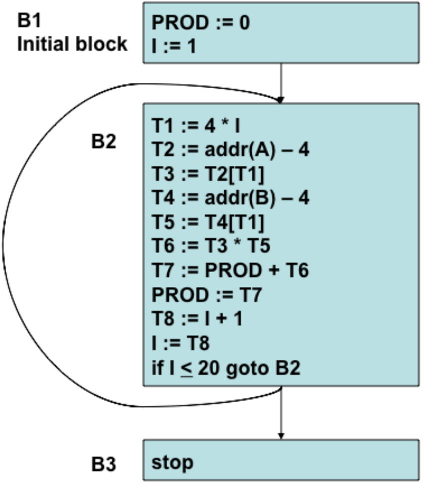
# Local and Global Optimizations

- Local optimizations: within basic blocks
  - Local common subexpression elimination
  - Dead code (instructions that compute a value that is never used) elimination
  - Reordering computations using algebraic laws
- Global optimizations: on whole procedures/programs
  - Global common sub-expression elimination
  - Constant propagation and constant folding
  - Loop invariant code motion
  - Partial redundancy elimination
  - Loop unrolling and function inlining
  - Vectorization and Concurrentization

# Basic Blocks and Control-Flow Graphs

- Basic blocks are sequences of intermediate code with a *single entry* and a single exit
- We consider the quadruple version of intermediate code here, to make the explanations easier
- Control flow graphs show control flow among basic blocks
- Basic blocks are represented as *directed acyclic blocks*(DAGs), which are in turn represented using the value-numbering method applied on quadruples
- Optimizations on basic blocks

# Example of Basic Blocks and Control Flow Graph




High level language code:

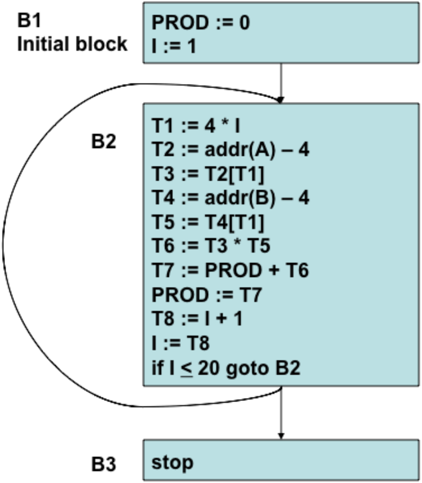
```
{ PROD = 0;
  for ( I = 1; I <= 20; I++)
    PROD = PROD + A[I] * B[I];
}
```

```
PROD := 0
I := 1
T1 := 4 * I
T2 := addr(A) – 4
T3 := T2[T1]
T4 := addr(B) – 4
T5 := T4[T1]
T6 := T3 * T5
T7 := PROD + T6
PROD := T7
T8 := I + 1
I := T8
if I ≤ 20 goto B2
stop
```

# Algorithm for Partitioning into Basic Blocks

1. Determine the set of *leaders*, the first statements of basic blocks
   - The first statement is a leader
   - Any statement which is the target of a conditional or unconditional *goto* is a leader
   - Any statement which immediately follows a *conditional goto* is a leader
2. A leader and all statements which follow it upto but not including the next leader (or the end of the procedure), is the basic block corresponding to that leader
3. Any statements, not placed in a block, can never be executed, and may now be removed, if desired

# Example of Basic Blocks and CFG




High level language code:

```
{ PROD = 0;
  for ( I = 1; I <= 20; I++)
    PROD = PROD + A[I] * B[I];
}
```

```
PROD := 0
I := 1
T1 := 4 * I
T2 := addr(A) – 4
T3 := T2[T1]
T4 := addr(B) – 4
T5 := T4[T1]
T6 := T3 * T5
T7 := PROD + T6
PROD := T7
T8 := I + 1
I := T8
if I ≤ 20 goto B2
stop
```

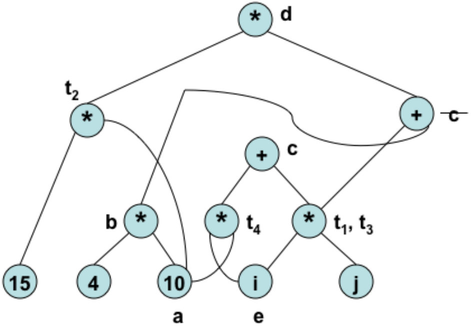
# Control Flow Graph

- The nodes of the CFG are basic blocks
- One node is distinguished as the initial node
- There is a directed edge $B1 \longrightarrow B2$, if B2 can immediately follow B1 in some execution sequence; i.e.,
  - There is a conditional or unconditional jump from the last statement of B1 to the first statement of B2, or
  - B2 immediately follows B1 in the order of the program, and B1 does not end in an unconditional jump
- A basic block is represented as a record consisting of
  1. a count of the number of quadruples in the block
  2. a pointer to the leader of the block
  3. pointers to the predecessors of the block
  4. pointers to the successors of the block

  Note that jump statements point to basic blocks and not quadruples so as to make code movement easy

# Example of a Directed Acyclic Graph (DAG)

1. $a = 10$
2. $b = 4 * a$
3. $t1 = i * j$
4. $c = t1 + b$
5. $t2 = 15 * a$
6. $d = t2 * c$
7. $e = i$
8. $t3 = e * j$
9. $t4 = i * a$
10. $c = t3 + t4$



# Value Numbering in Basic Blocks

- A simple way to represent DAGs is via *value-numbering*
- While searching DAGs represented using pointers etc., is inefficient, *value-numbering* uses hash tables and hence is very efficient
- Central idea is to assign numbers (called value numbers) to expressions in such a way that two expressions receive the same number if the compiler can prove that they are equal for all possible program inputs
- We assume quadruples with binary or unary operators
- The algorithm uses three tables indexed by appropriate hash values:
  *HashTable, ValnumTable,* and *NameTable*
- Can be used to eliminate common sub-expressions, do constant folding, and constant propagation in basic blocks
- Can take advantage of commutativity of operators, addition of zero, and multiplication by one