SOEN 6431
Software Comprehension and Maintenance

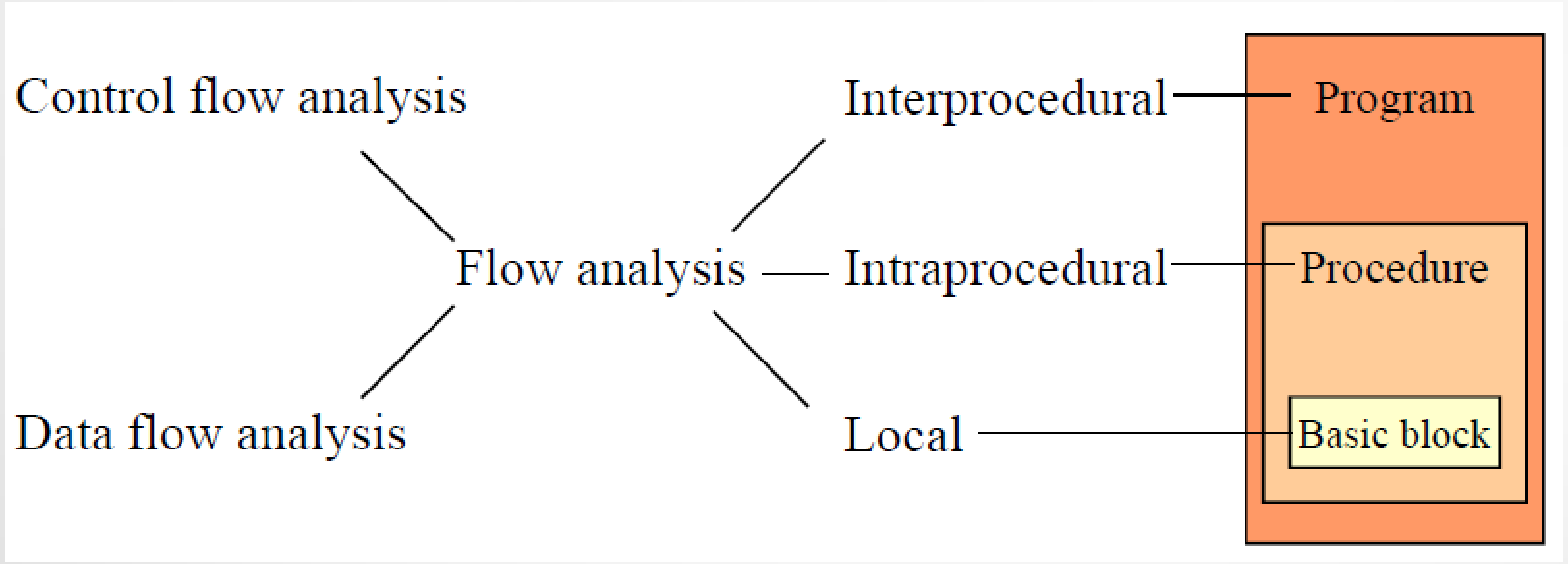
Week 5

Control flow analysis

Outline

- Flow Analysis
- Approaches for Control-Flow Analysis
- Basic Block
- Dominators
- Finding Loops
- Reducibility
- Interval Analysis
- Structure Analysis

Flow Analysis



- **Control Flow Analysis:** determine the control structure of a program and build a Control Flow Graph.
- **Data Flow Analysis:** determine the flow of scalar values and build Data Flow Graphs.
- *Solution to the* ***Flow Analysis Problem****: propagate data flow information along a flow graph.*

Levels of Analysis

(in order of increasing detail & complexity)

- **Local** (single-block) [1960's]
  - Straight-line code
  - Simple to analyze; limited impact
- **Intraprocedural** [1970's – today]
  - Whole procedure
  - Dataflow & dependence analysis
- **Interprocedural** [late 1970's – today]
  - Whole-program analysis
  - Tricky:
    - *Very* time and space intensive
    - Hard for some PL's (e.g., Java)

Optimization = Analysis + Transformation

- Key analyses:
  - Control-flow
    - if-statements, branches, loops, procedure calls
  - Data-flow
    - definitions and uses of variables

- Any unused generality that would ordinarily result from un-optimized compilation can be stripped away. Thus, less efficient but more general mechanisms are replaced with more efficient, specialized ones.

- Representations:
  - Control-flow graph
  - Control-dependence graph
  - Def/use, use/def chains
  - SSA (Static Single Assignment)

Approaches for Control-Flow Analysis

- There are two main approaches to control-flow analysis of single routines

- ***Dominance analysis*** – use dominators to discover loops. It is sufficient for use by optimizers that do data-flow analysis by iteration, or that concentrate their attention strictly on the loops in a routine.

- ***Interval analysis*** – includes a series of methods that analyze the overall structure of the routine and that decompose it into nested regions called *intervals*.
    - The most sophisticated variety of interval analysis, called **structural analysis**, classifies essentially all the control-flow structures in a routine.

Approaches for Control-Flow Analysis (cont.)

- The dominator-based approach
  - The least time-intensive to implement
  - Sufficient to provide the information needed to perform most of the optimizations.
- Advantages of the interval-based approaches
  - Faster at performing the actual data-flow analyses.
  - Easier to update already computed data-flow information in response to changes so that no need to recompute from scratch.
  - Structure analysis is particularly easy to perform certain control-flow transformations.
- Both approaches start by determining the basic blocks that make up the routine and then constructing its flow graph.

Basic Blocks

- A basic block is a sequence of *consecutive* intermediate language statements in which flow of control can only *enter at the beginning* and *leave at the end*.
- Only the last statement of a basic block can be a branch statement and only the first statement of a basic block can be a target of a branch.
- In some frameworks, procedure calls may occur within a basic block.

Basic Block Partitioning Algorithm

1. Identify leader statements (i.e. the first statements of basic blocks) by using the following rules:

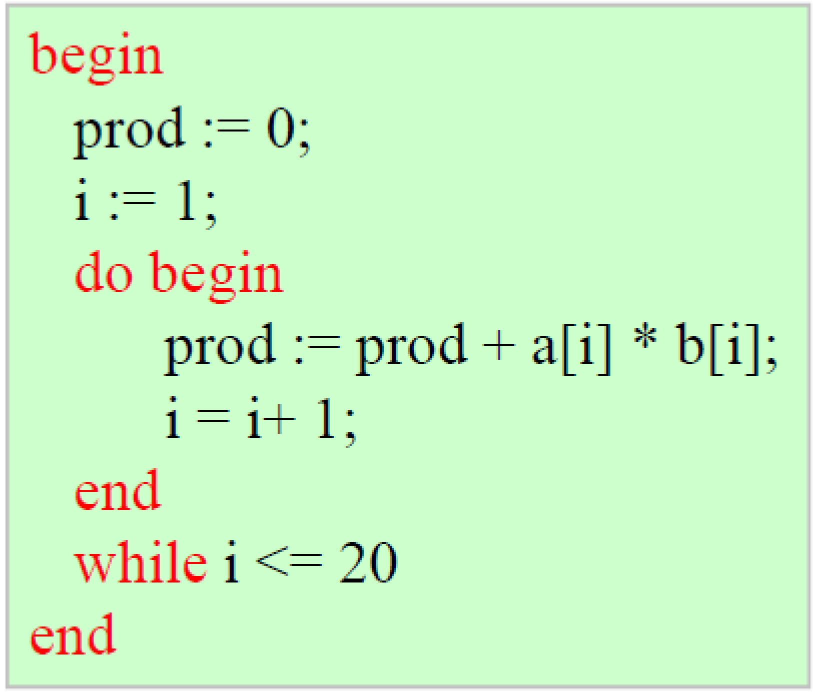
(i) The *first statement* in the program is a leader

(ii) Any statement that is the *target of a branch* statement is a leader (for most intermediate languages these are statements with an associated label)

(iii) Any statement that *immediately follows a branch or return statement* is a leader

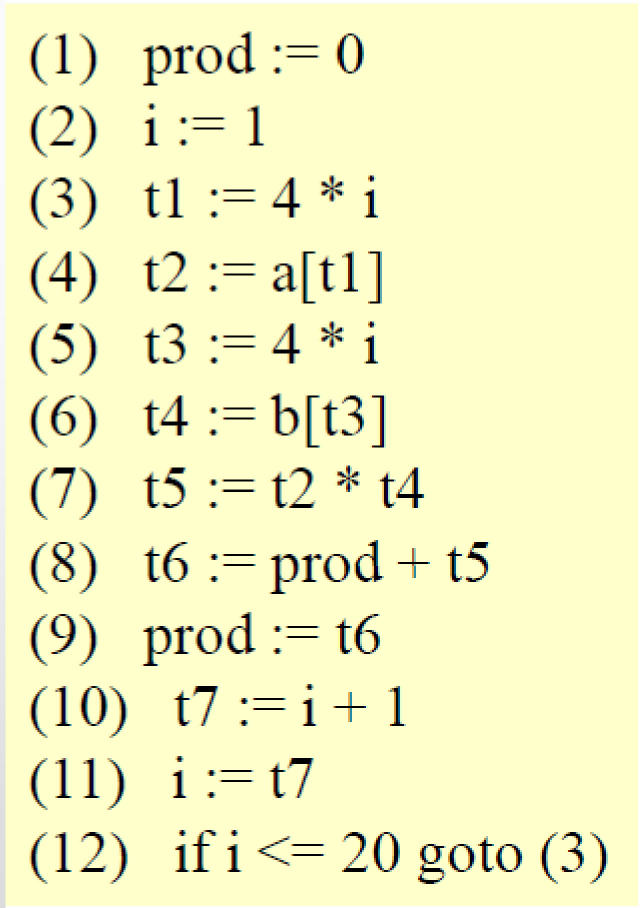
Example: Finding Leaders

The following code computes the inner product of two vectors.



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

Source code



```
(1)   prod := 0
(2)   i := 1
(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)
```

Three-address code

Example: Finding Leaders

The following code computes the inner product of two vectors.

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

Source code

Rule (i)

```
(1)   prod := 0
(2)   i := 1
(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)
(13)  …
```

Three-address code

Example: Finding Leaders

The following code computes the inner product of two vectors.

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

Source code

Rule (i)

Rule (ii)

```
(1)   prod := 0
(2)   i := 1
(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)
(13)  …
```

Three-address code

Example: Finding Leaders

The following code computes the inner product of two vectors.

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

Source code

```
Rule (i)     (1)   prod := 0
             (2)   i := 1
Rule (ii)    (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)
Rule (iii)   (13)  …
```

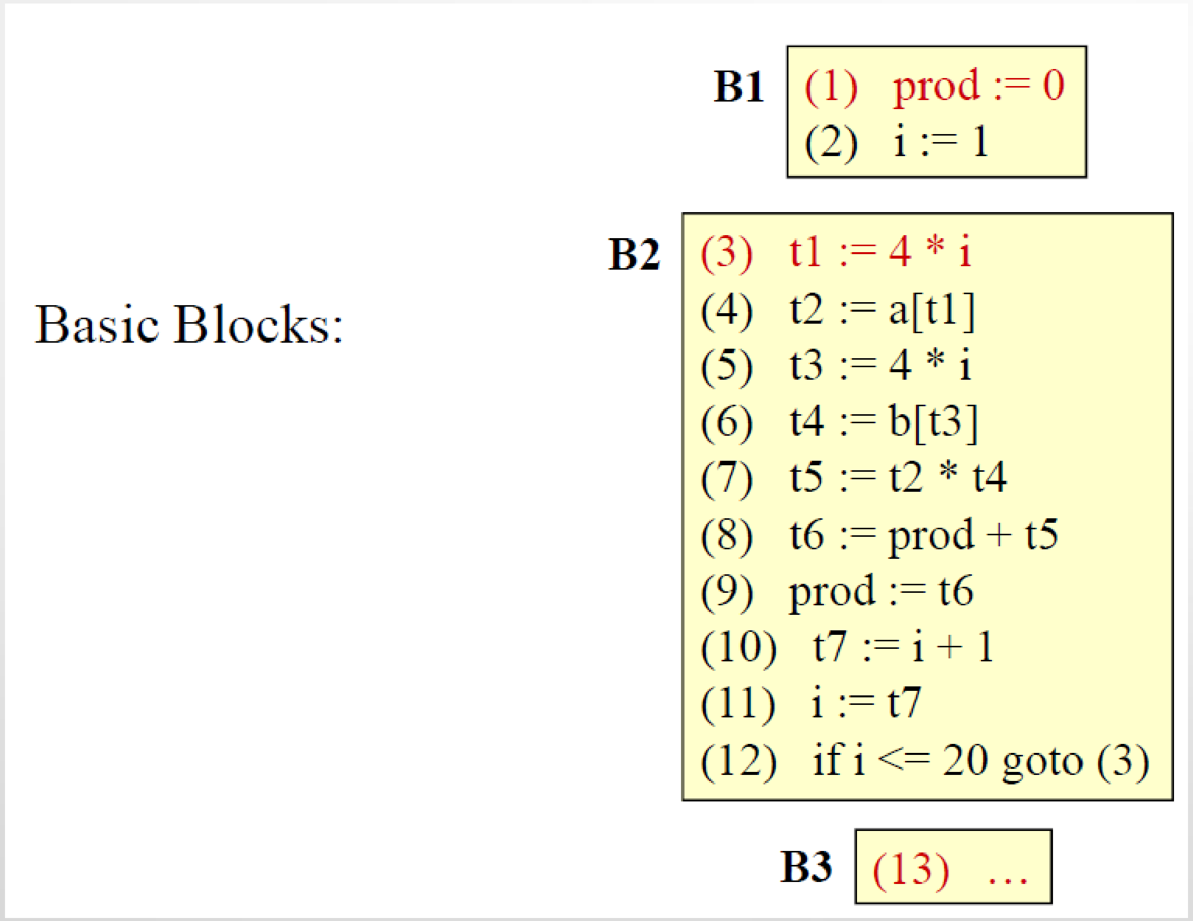
Three-address code

Forming the Basic Blocks

Now that we know the leaders, how do we form the basic blocks associated with each leader?

2. The basic block corresponding to a leader consists of the leader, plus all statements up to *but not including* the next leader or up to the end of the program.

Example: Forming the Basic Blocks



Basic Blocks:

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)
```

B3
```
(13)  …
```

Control Flow Graph (CFG)

- A ***control flow graph*** (CFG), or simply a flow graph, is a directed multigraph in which:
  - (i) the nodes are basic blocks; and
  - (ii) the edges represent flow of control (branches or fall-through execution).

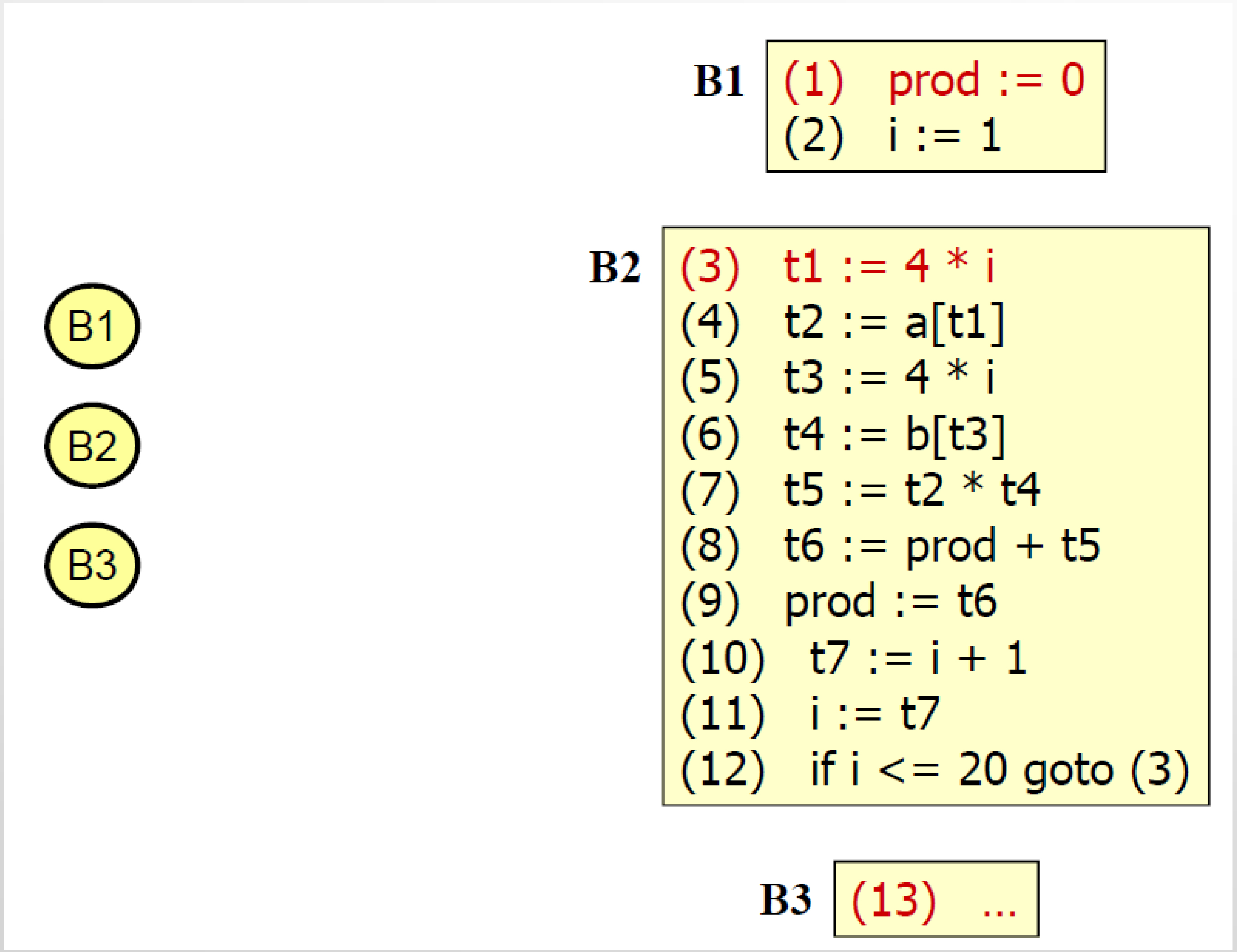
- The basic block whose leader is the first intermediate language statement is called the ***start node***.

- In a CFG, we have no information about the data. Therefore an edge in the CFG means that the program ***may*** take that path.

Control Flow Graph (CFG)

- There is a directed edge from basic block B1 to basic block B2 in the CFG if:
  - (1) There is a branch from the last statement of B1 to the first statement of B2, or
  - (2) Control flow can fall through from B1 to B2 because:
    - (i) B2 immediately follows B1, and
    - (ii) B1 does not end with an unconditional branch

Example: Control Flow Graph Formation

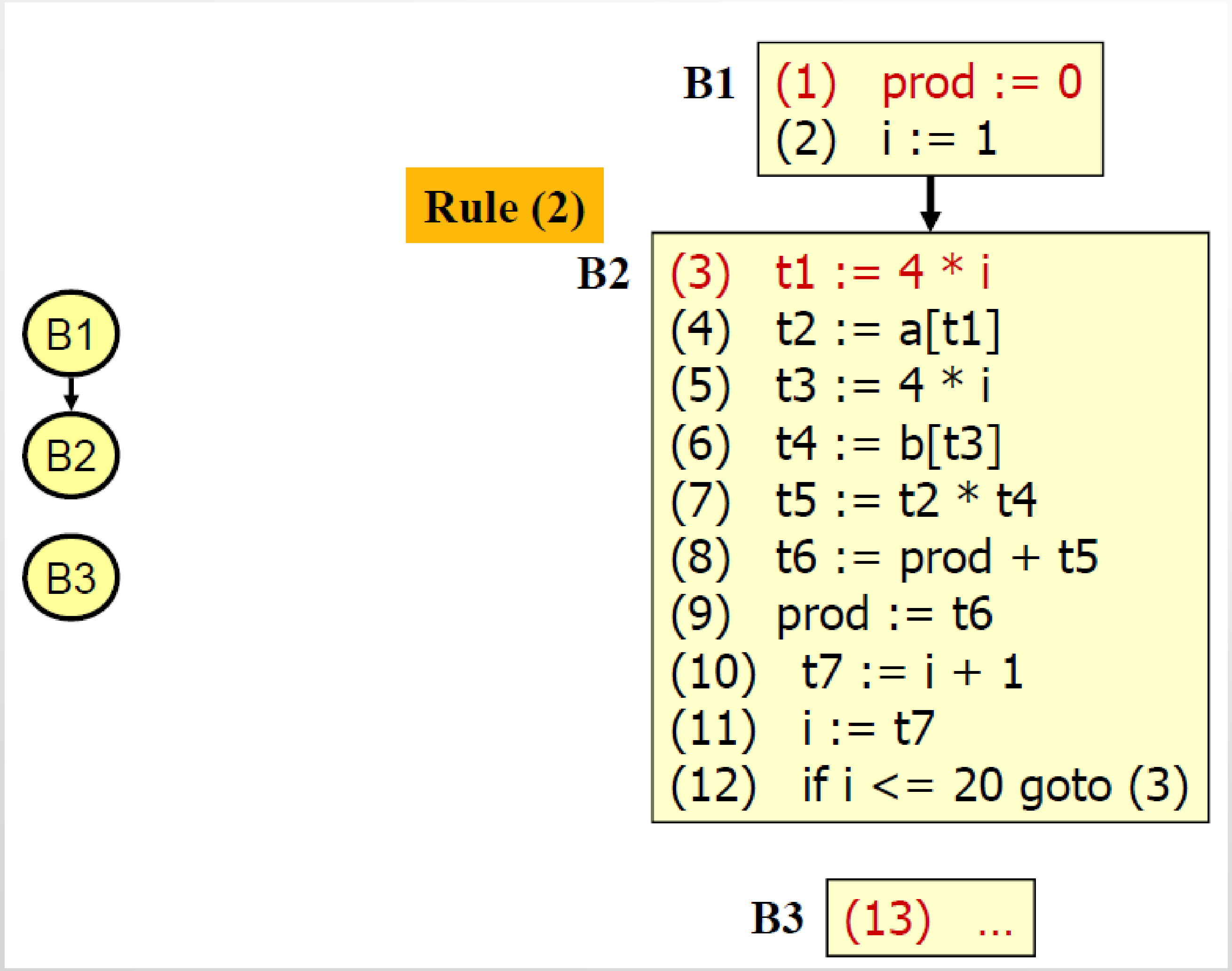


```
B1  (1)  prod := 0
    (2)  i := 1

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)

B3  (13)  ...
```

Example: Control Flow Graph Formation

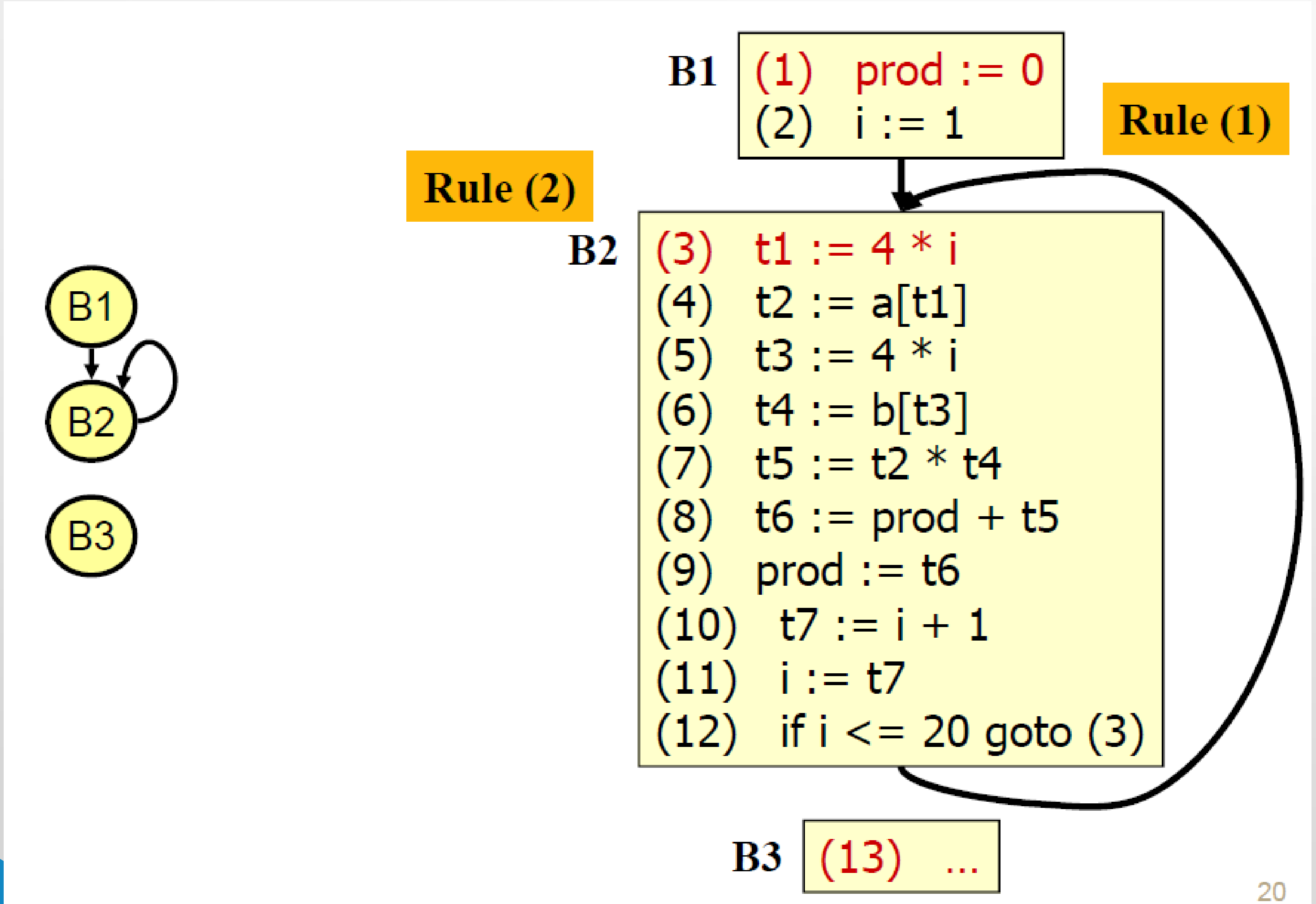


B1
```
(1)   prod := 0
(2)   i := 1
```

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)
```

B3
```
(13)   ...
```

Example: Control Flow Graph Formation

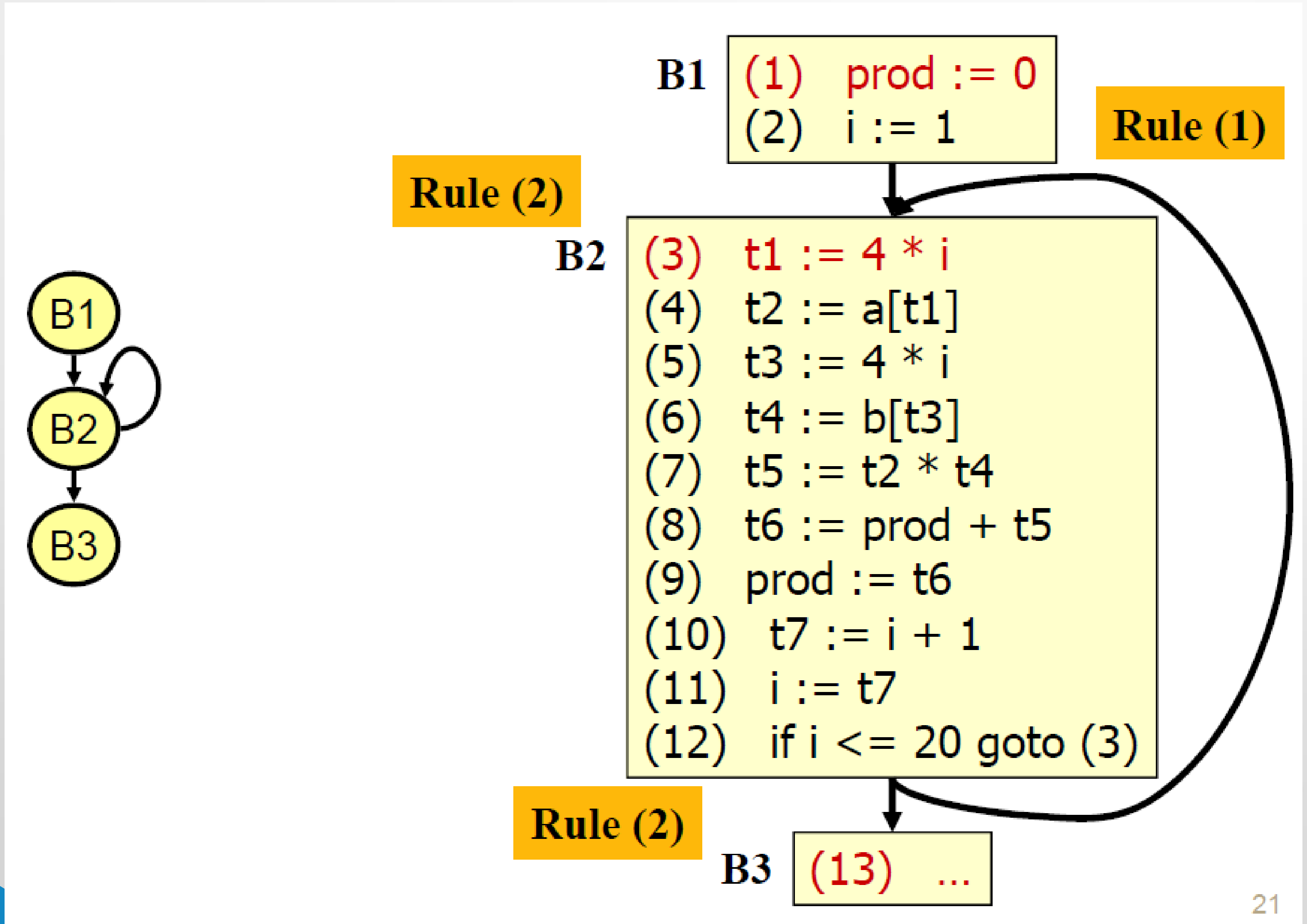


B1
```
(1)  prod := 0
(2)  i := 1
```

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)
```

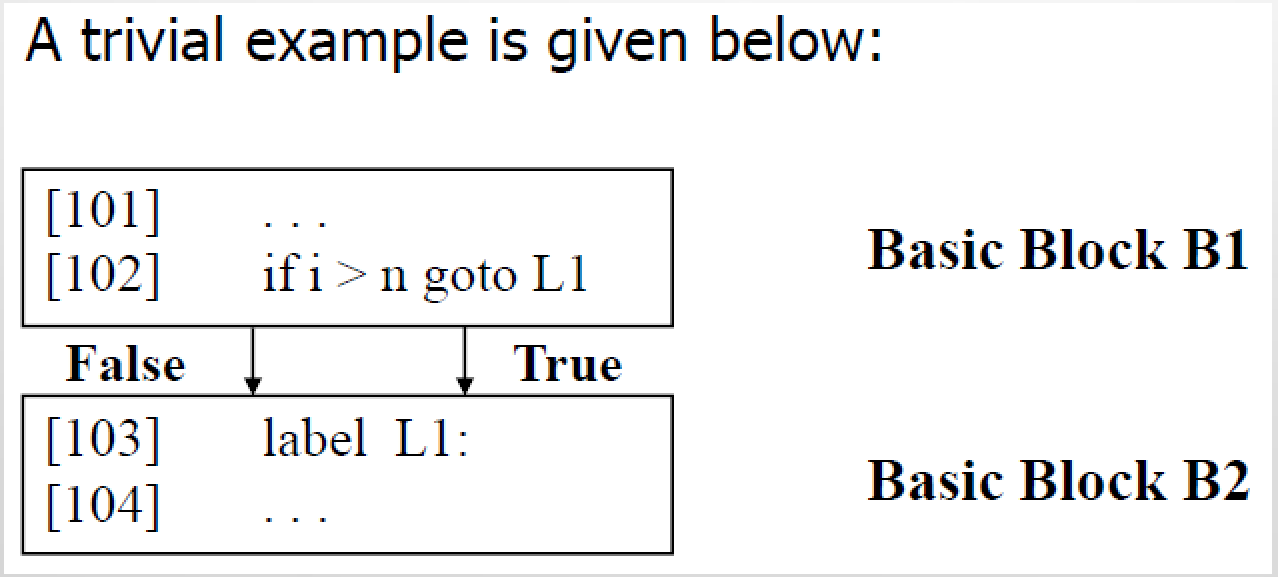
B3
```
(13)  ...
```

Example: Control Flow Graph



CFGs are Multigraphs

- Note: there may be multiple edges from one basic block to another in a CFG.
- Therefore, in general the CFG is a multigraph.
- The edges are distinguished by their condition labels.


A trivial example is given below:

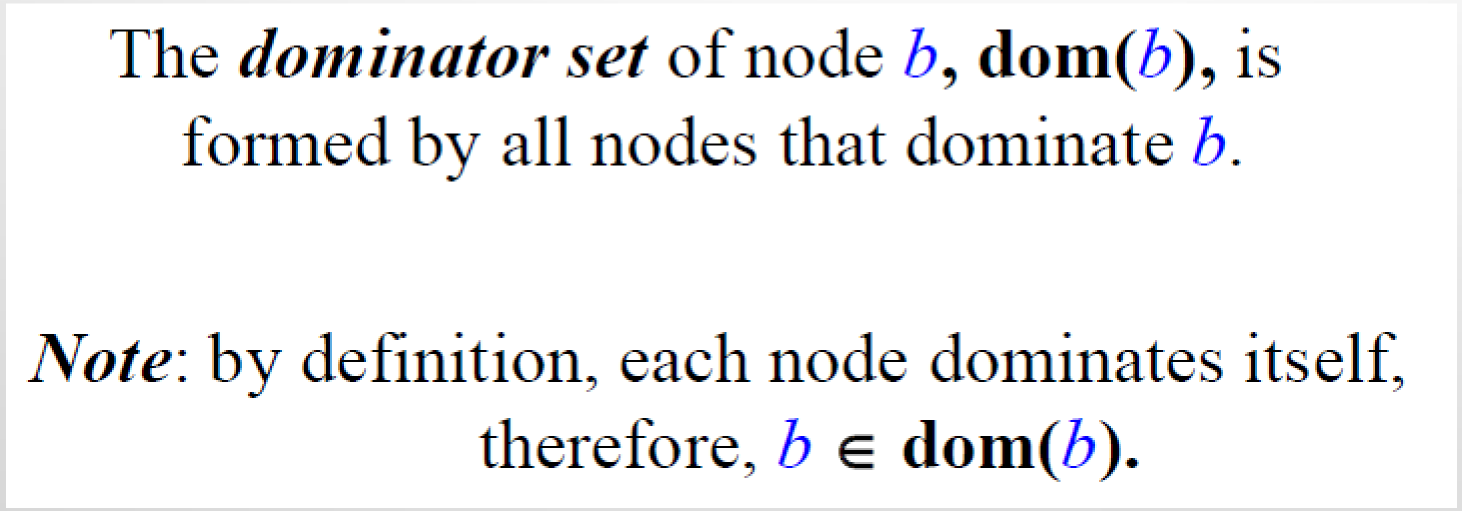
Identifying loops

Question: Given the control flow graph of a procedure, how can we identify loops?

Answer: We use the concept of dominance.

Dominators

- A node *a* in a CFG ***dominates*** a node *b* if every path from the start node to node *b* goes through *a*. We say that node *a* is a ***dominator*** of node *b*.



The ***dominator set*** of node *b*, **dom(*b*)**, is formed by all nodes that dominate *b*.

Note: by definition, each node dominates itself, therefore, $b \in$ **dom(*b*)**.

Domination Relation

Definition: Let $G = (N, E, s)$ denote a flowgraph, where:

N: set of vertices

E: set of edges, $E \subseteq N \times N$

s: starting node.

and let $a \in N$, $b \in N$.

1. a **dominates** b, written $a \leq b$, if
   every path from s to b contains a.

2. a **properly/strictly dominates** b, written $a < b$, if
   $a \leq b$ and $a \neq b$.

3. a **directly (immediately) dominates** b, written $a <_{\mathrm{d}} b$ if:

   $a < b$ and

   there is no $c \in N$ such that $a < c < b$.

Domination Properties

- Dominance has three properties:
  - *Reflexive*: by definition, each node dominates itself, therefore, $a \leq a$.
  - *Transitive*: if $a \leq b$ and $b \leq c$, then $a \leq c$
  - *Antisymmetric*: if $a \leq b$ and $b \leq a$, then $b = a$

An Example

Domination relation:

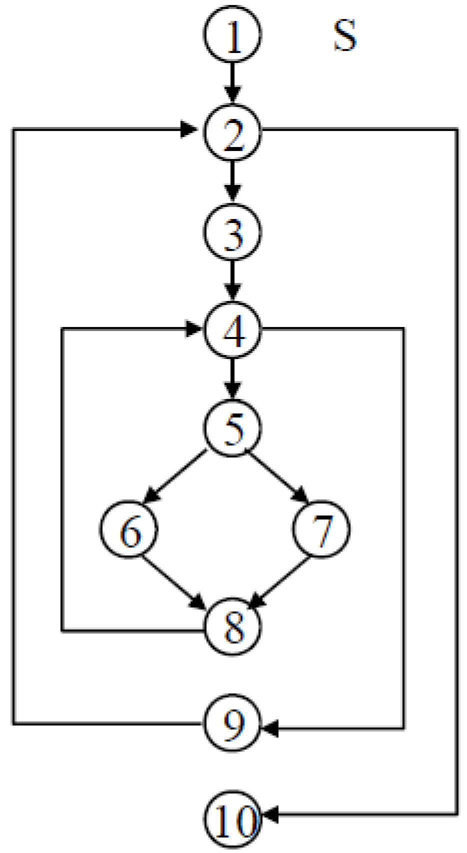
{ (1, 1), (1, 2), (1, 3), (1,4) …
(2, 3), (2, 4), …
(2, 10)
}

Direct Domination:

$1 <_d 2, 2 <_d 3, \ldots$

Dominator Sets:

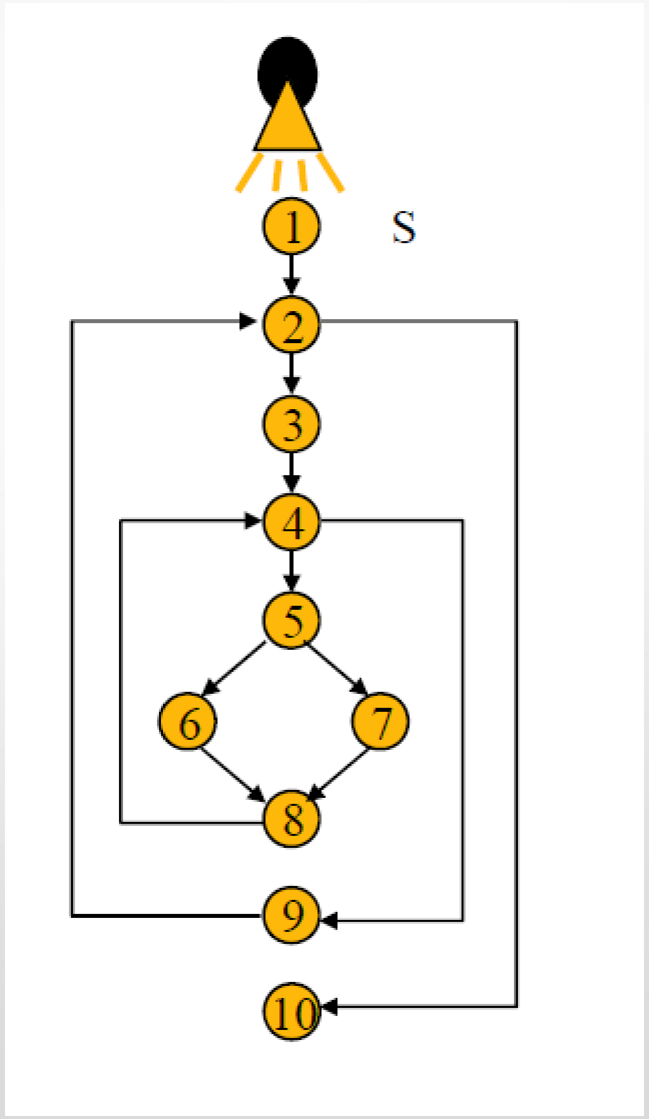
DOM(1) = {1}
DOM(2) = {1, 2}
DOM(3) = {1, 2, 3}
DOM(10) = {1, 2, 10)



Dominance Intuition

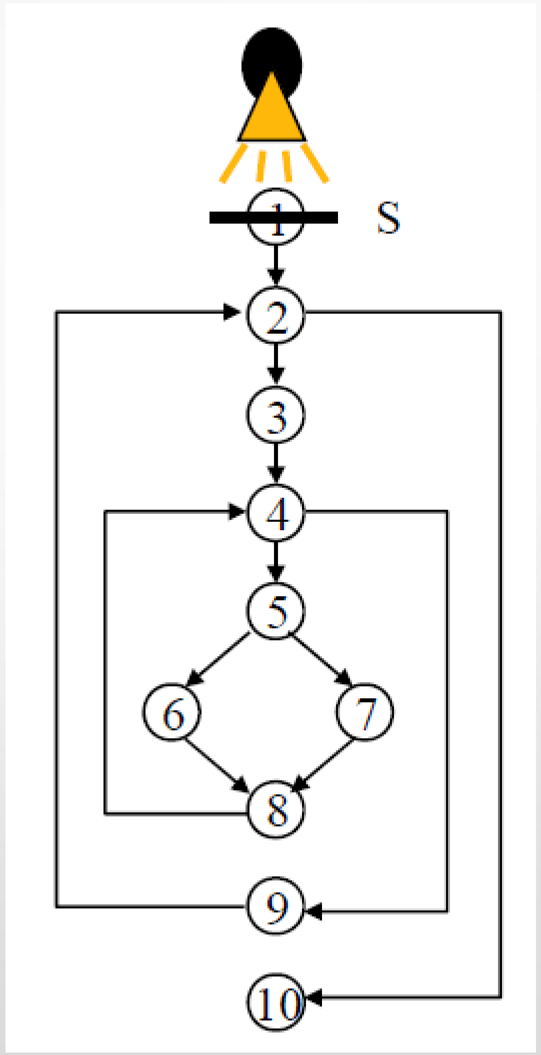
- Imagine a source of light at the start node, and that the edges are optical fibers

- To find which nodes are dominated by a given node a, place an opaque barrier at a and observe which nodes became dark.



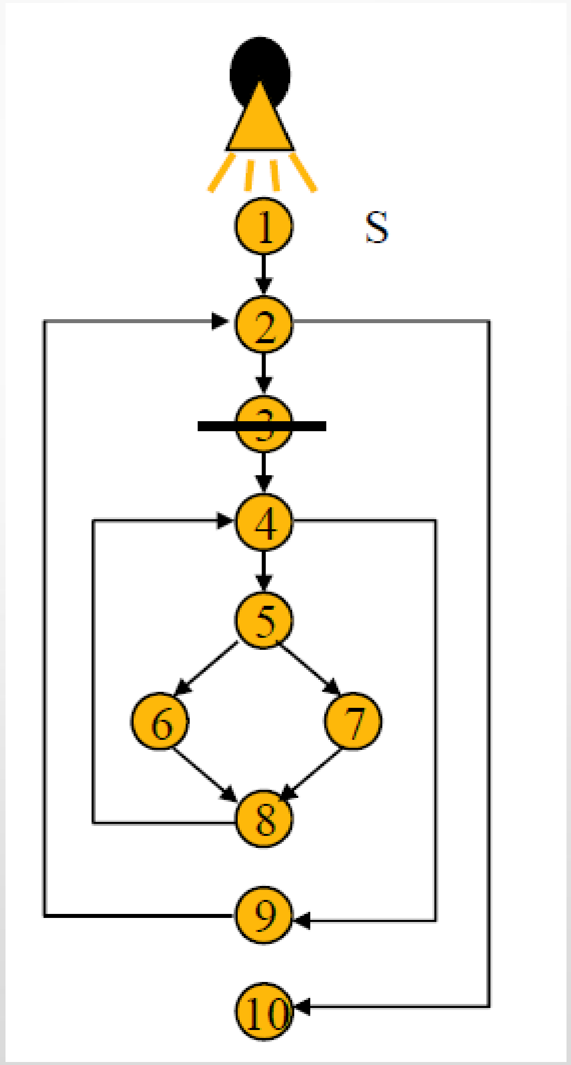
Dominance Intuition

- The start node dominates all nodes in the flowgraph.



Dominance Intuition

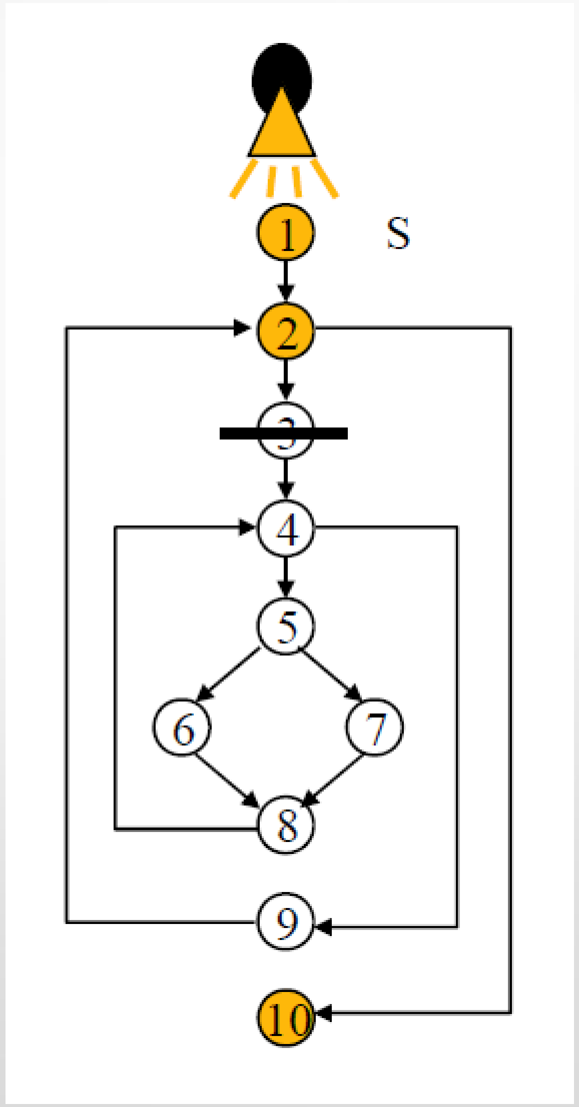
- Which nodes are dominated by node 3?



Dominance Intuition

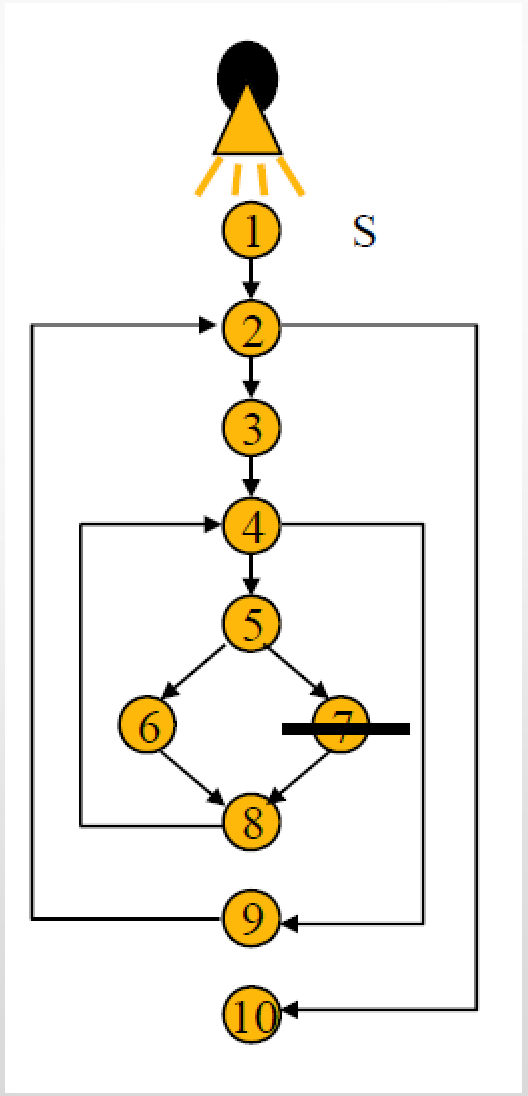
- Which nodes are dominated by node 3?

- Node 3 dominates nodes 3, 4, 5, 6, 7, 8, and 9.



Dominance Intuition

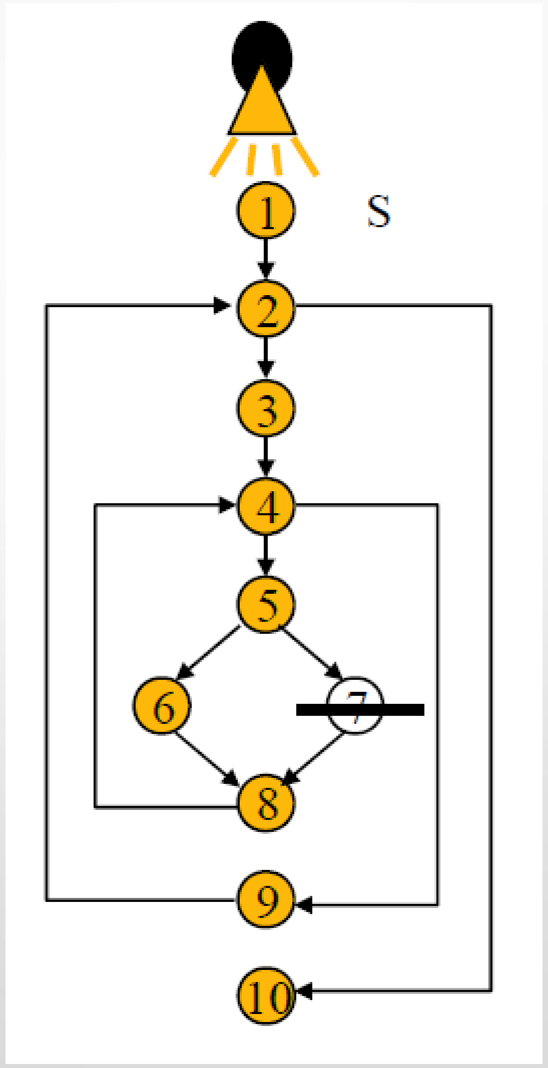
- Which nodes are dominated by node 7?



Dominance Intuition

- Which nodes are dominated by node 7?

- Node 7 only dominates itself.



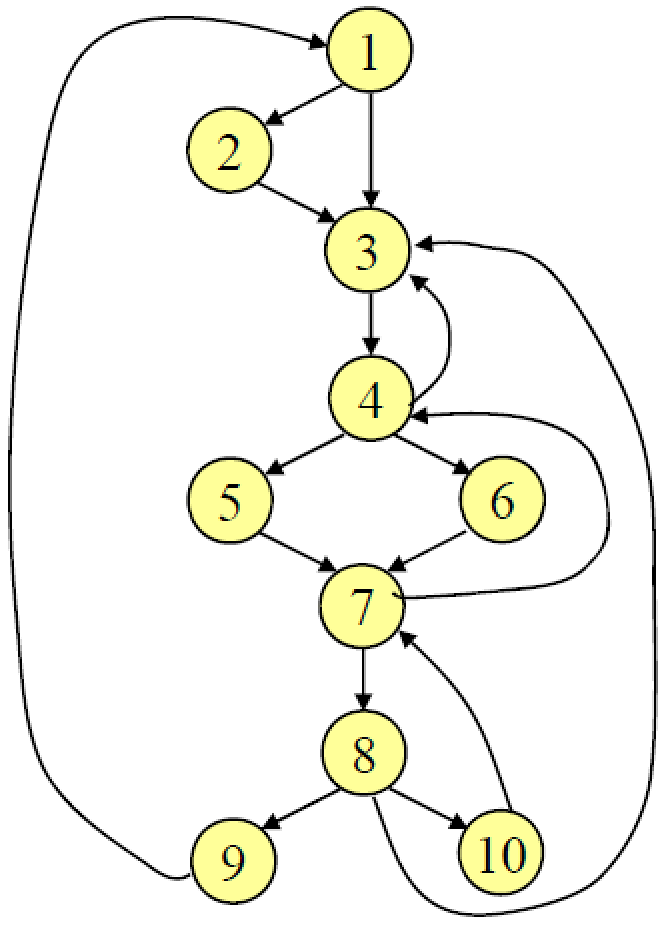
Second Approach

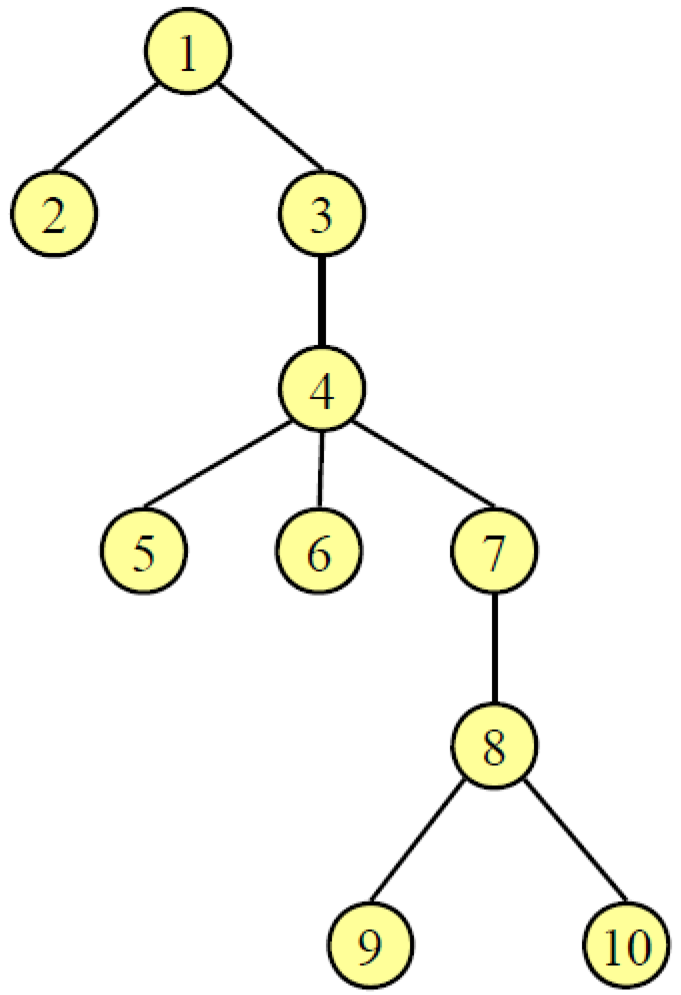
- The second approach to compute dominators was developed by Lengauer and Tarjan [LT79]
  - More complicated than the first approach
  - Running significantly faster on all but the smallest flowgraphs
- Alstrup and Lauridesn [AL96] describe a technique for incrementally updating a dominator tree as the control flow of a procedure is modified.

A Dominator Tree

- A dominator tree is a useful way to represent the dominance relation.
- In a dominator tree the start node **s** is the root, and each node **d** dominates only its descendants in the tree.

A Dominator Tree (Example)





Finding Loops

- ***Motivation***: Programs spend most of the execution time in loops, therefore there is a larger payoff for optimizations that exploit loop structure.
- How do we identify loops in a flow graph?
- The goal is to create an uniform treatment for program loops written using different loop structures (e.g. while, for) and loops constructed out of goto’s.
- ***Basic idea***: Use a general approach based on analyzing graph-theoretical properties of the CFG.

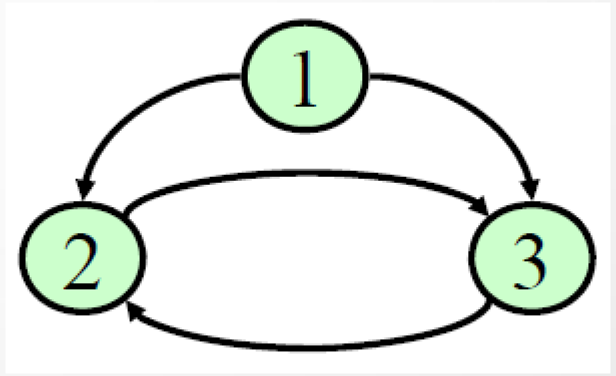
Definition

- A strongly-connected component (SCC) of a flowgraph G = (N, E, s) is a subgraph G′ = (N′, E′, s′) in which there is a path from each node in N′ to every node in N′.

- A strongly-connected component G′ = (N′, E′, s′) of a flowgraph G = (N, E, s) is a loop with entry s′ if s′ dominates all nodes in N′.

Example

- In the flow graph below, do nodes 2 and 3 form a loop?

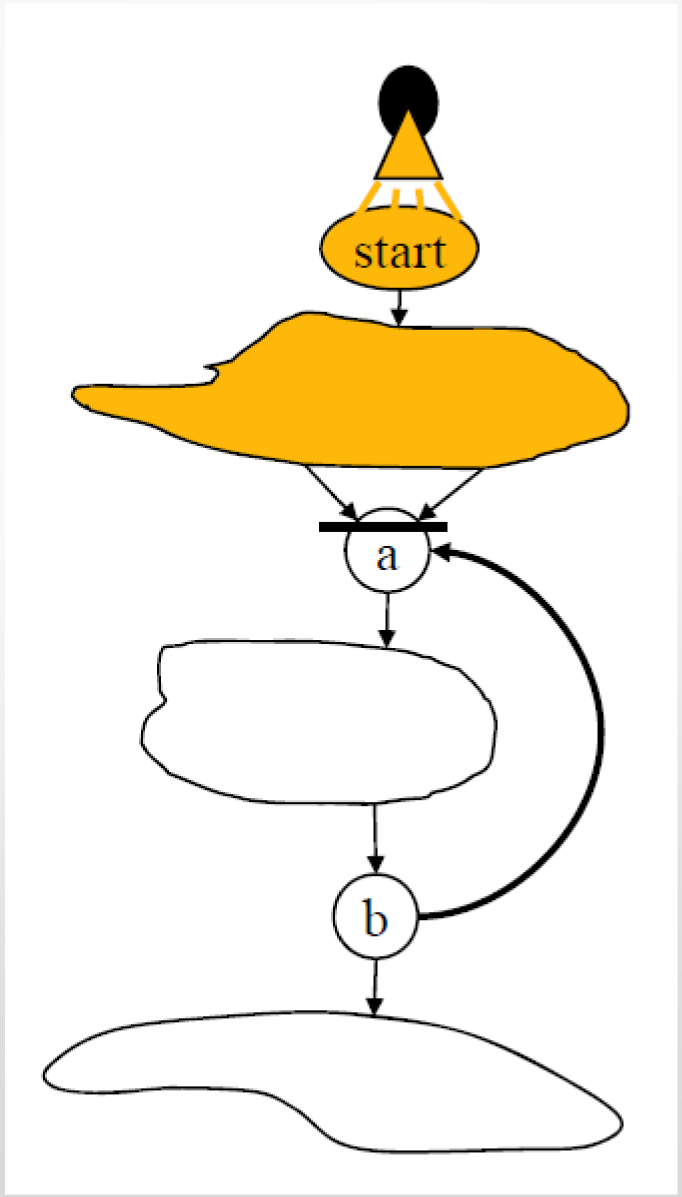


- Nodes 2 and 3 form a strongly connected component, but they are not a loop. Why?

- No node in the subgraph dominates all the other nodes, therefore this subgraph is not a loop.

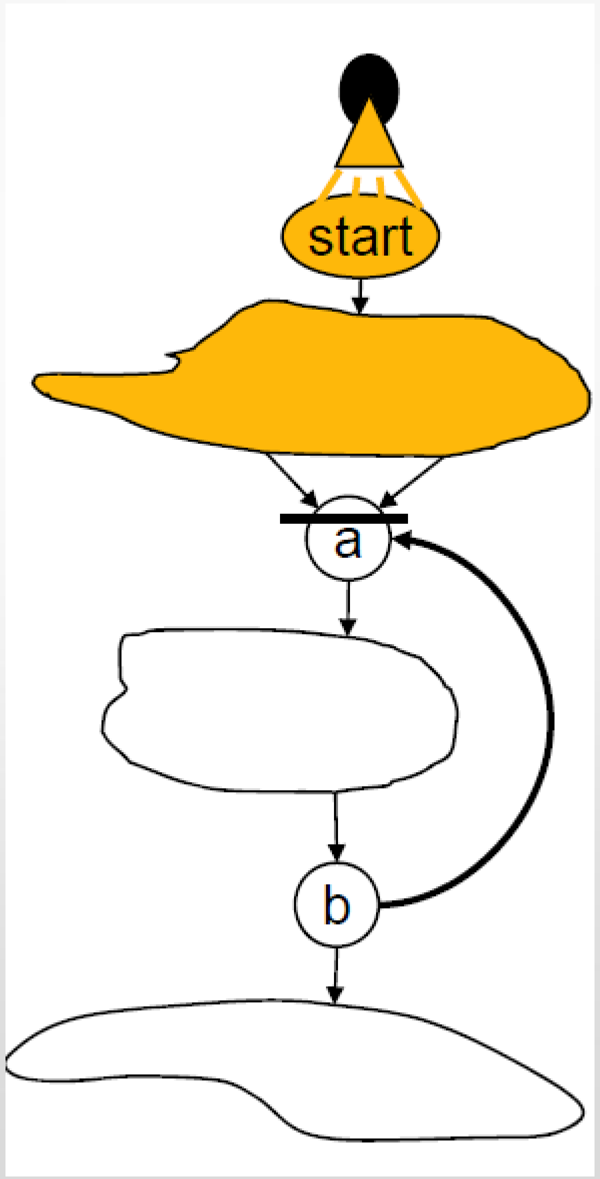
How to Find Loops?

- Look for “back edges”
- An edge (b,a) of a flowgraph G is a back edge if a dominates b, a < b.



Natural Loops

Given a back edge (b,a), a natural loop associated with (b,a) with entry in node a is the subgraph formed by a plus all nodes that can ***reach*** b ***without going through*** a.



Natural Loops

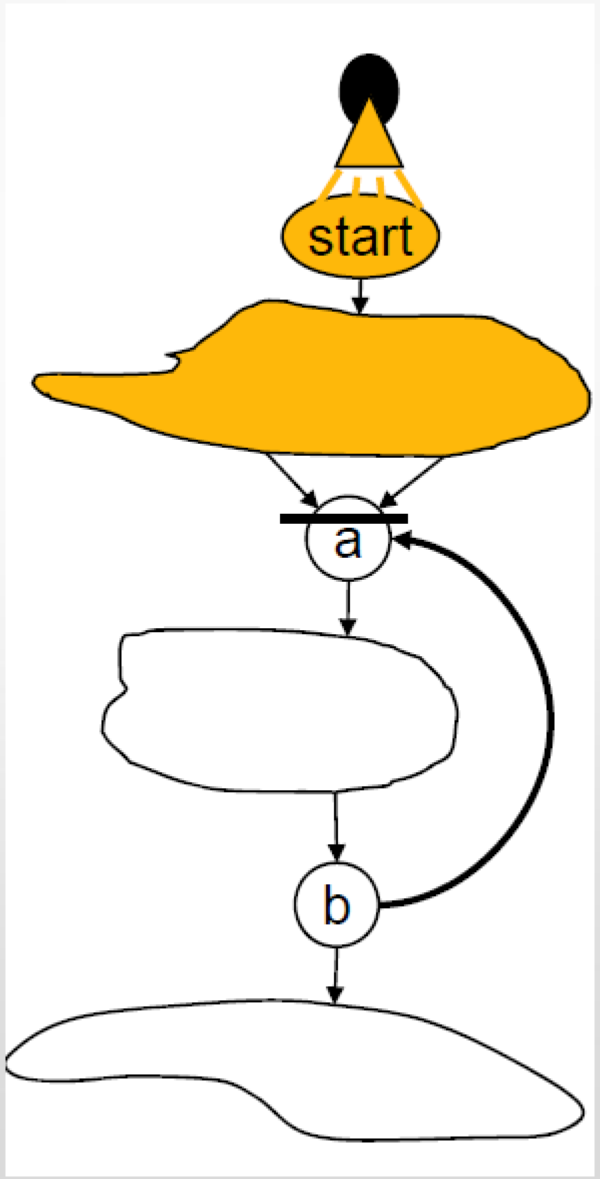
One way to find natural loops is:

1) find a back edge (b,a)

2) find the nodes that are

dominated by a.

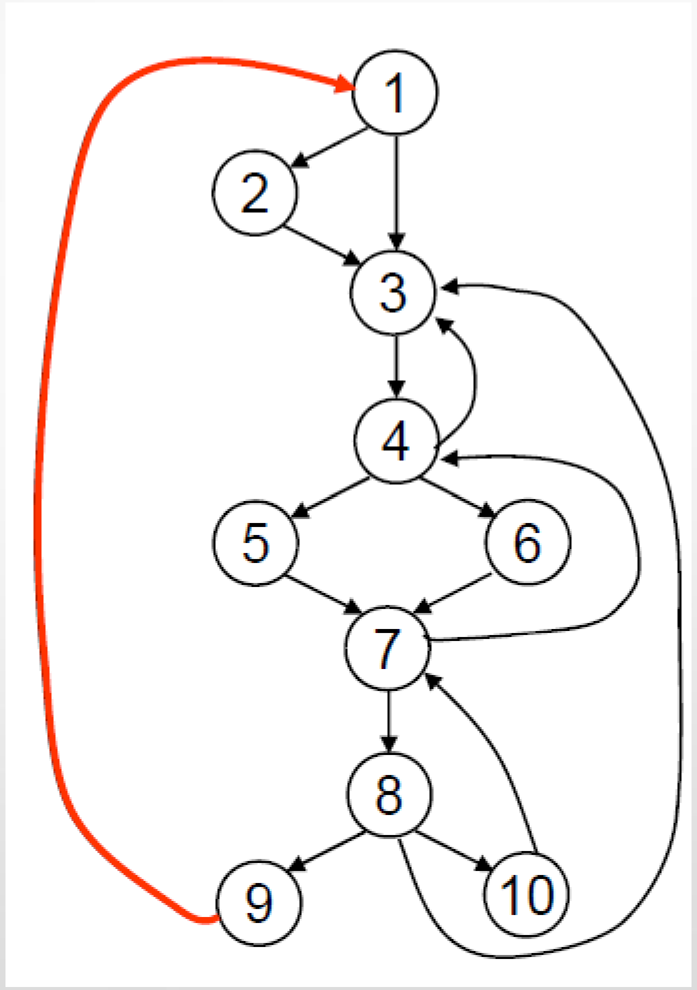
3) look for nodes that can reach b among the nodes dominated by a.



An Example

Find all back edges in this graph and the natural loop associated with each back edge

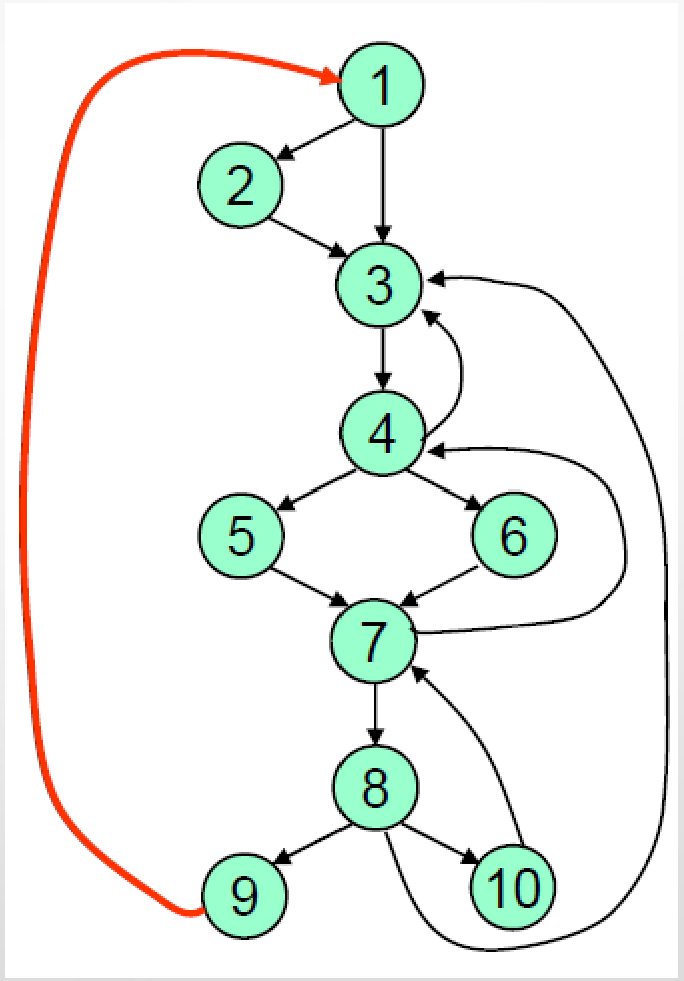
(9,1)



An Example

Find all back edges in this graph and the natural loop associated with each back edge

(9,1)	Entire graph

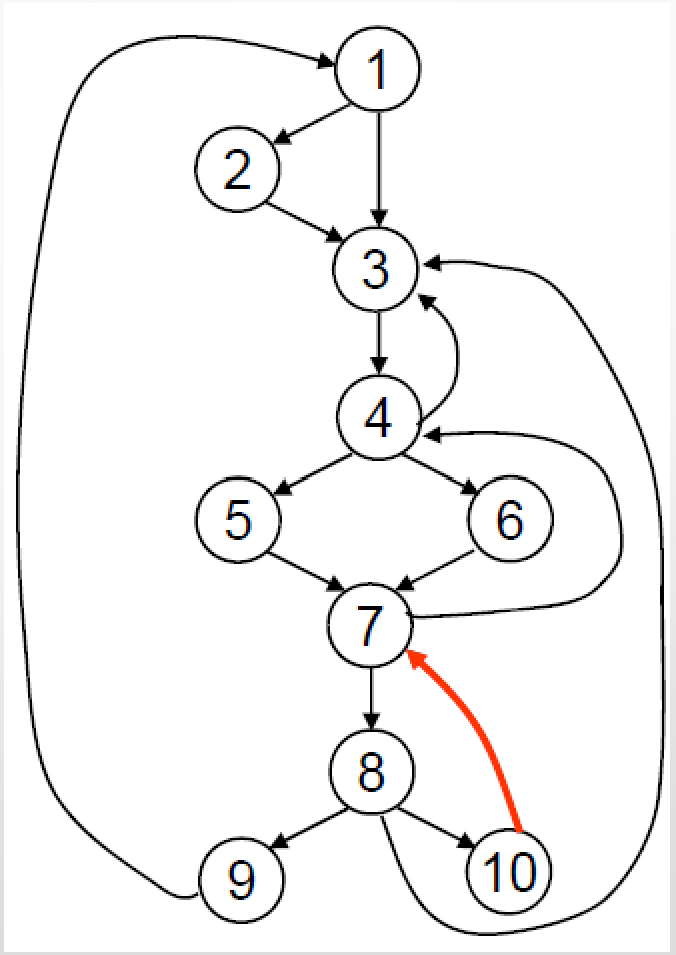


An Example

Find all back edges in this graph and the natural loop associated with each back edge

(9,1)   Entire graph

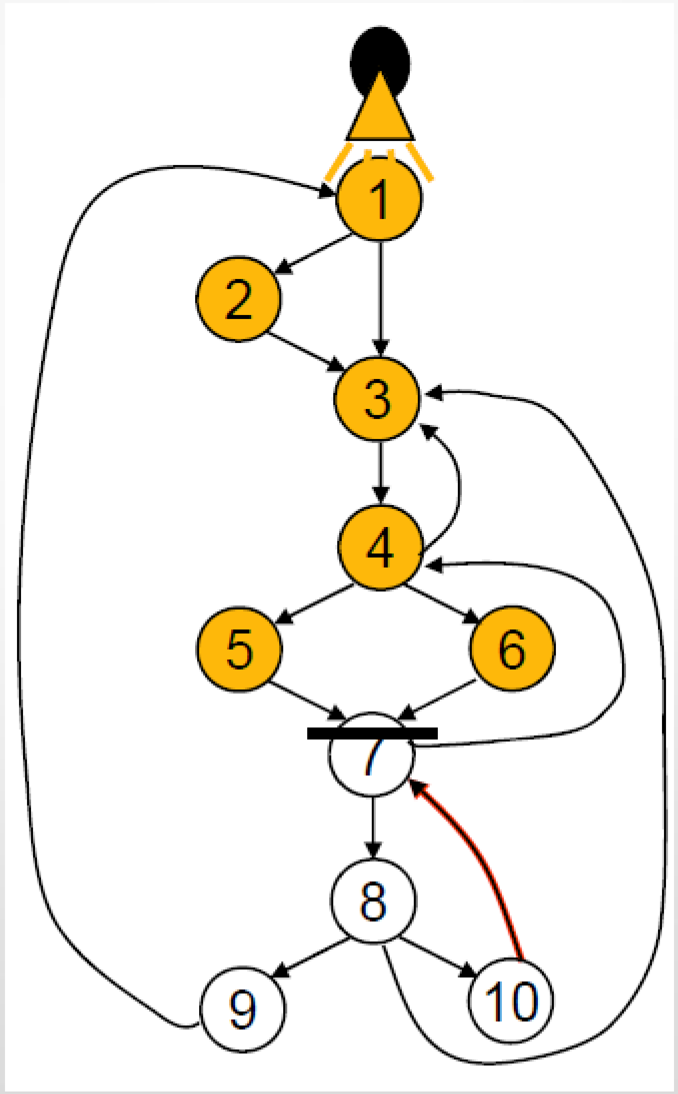
(10,7)



An Example

Find all back edges in this graph and the natural loop associated with each back edge

(9,1)	Entire graph
(10,7)	

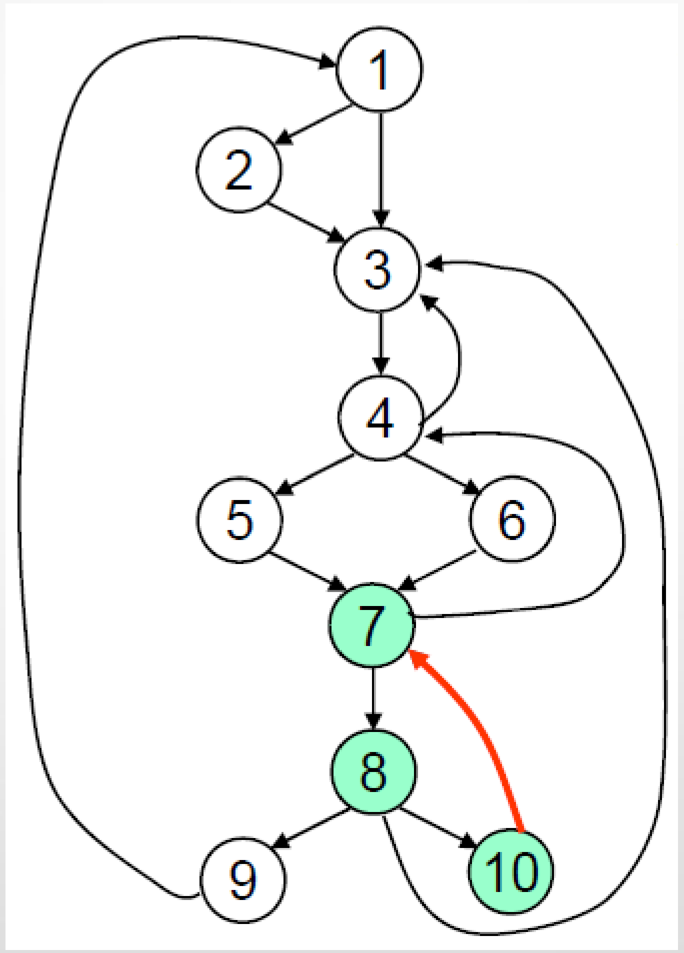


An Example

Find all back edges in this graph and the natural loop associated with each back edge

(9,1)   Entire graph

(10,7) {7,8,10}



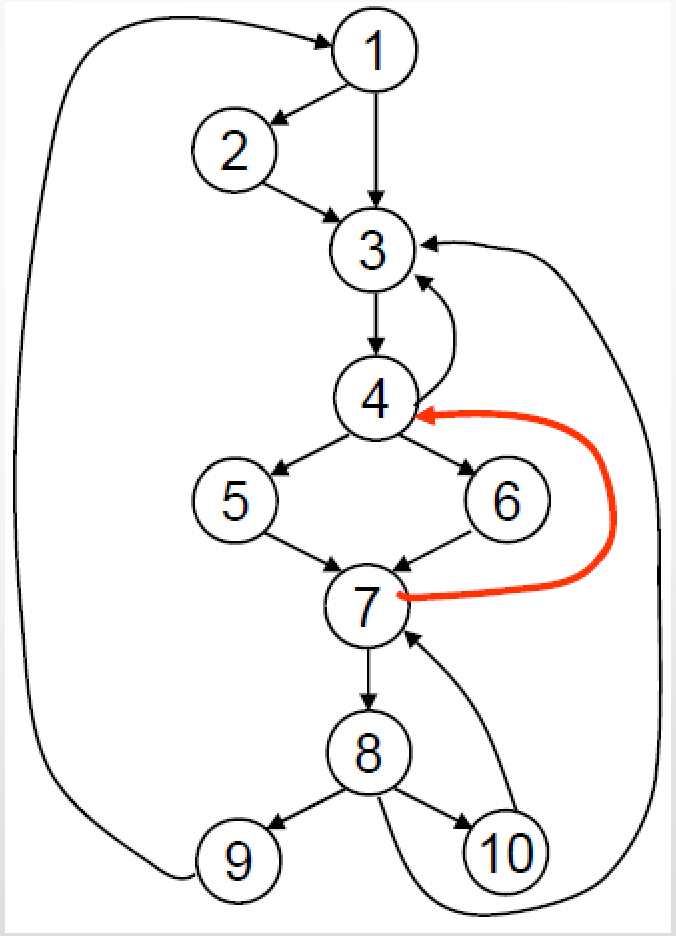
An Example

Find all back edges in this graph and the natural loop associated with each back edge

(9,1) Entire graph

(10,7) {7,8,10}

(7,4)



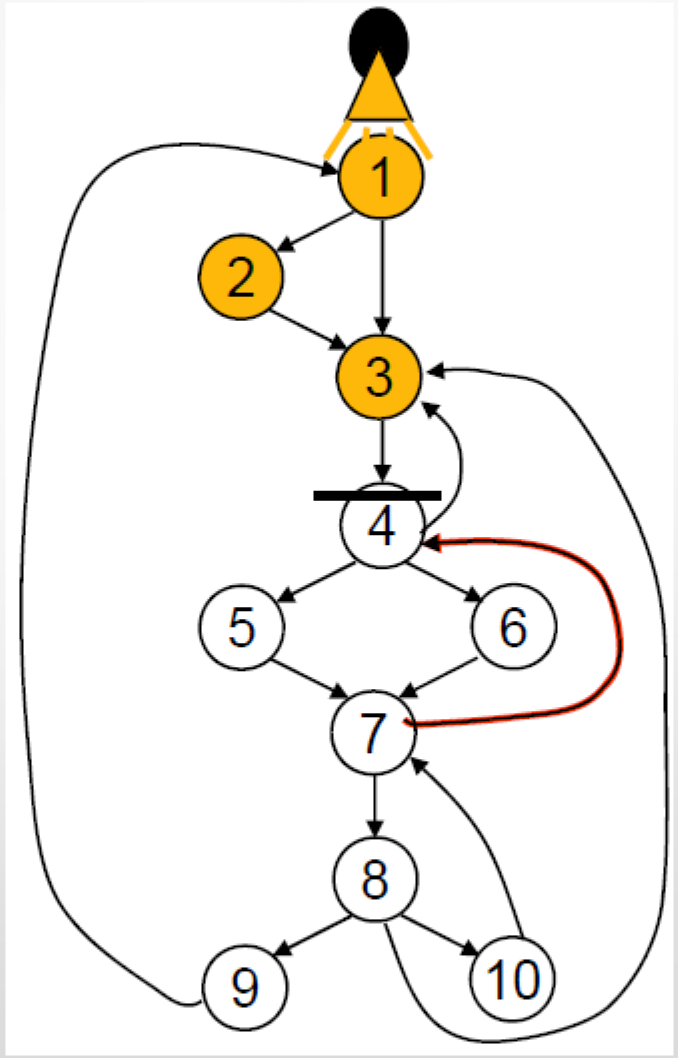
An Example

Find all back edges in this graph and the natural loop associated with each back edge

(9,1) Entire graph

(10,7) {7,8,10}

(7,4)



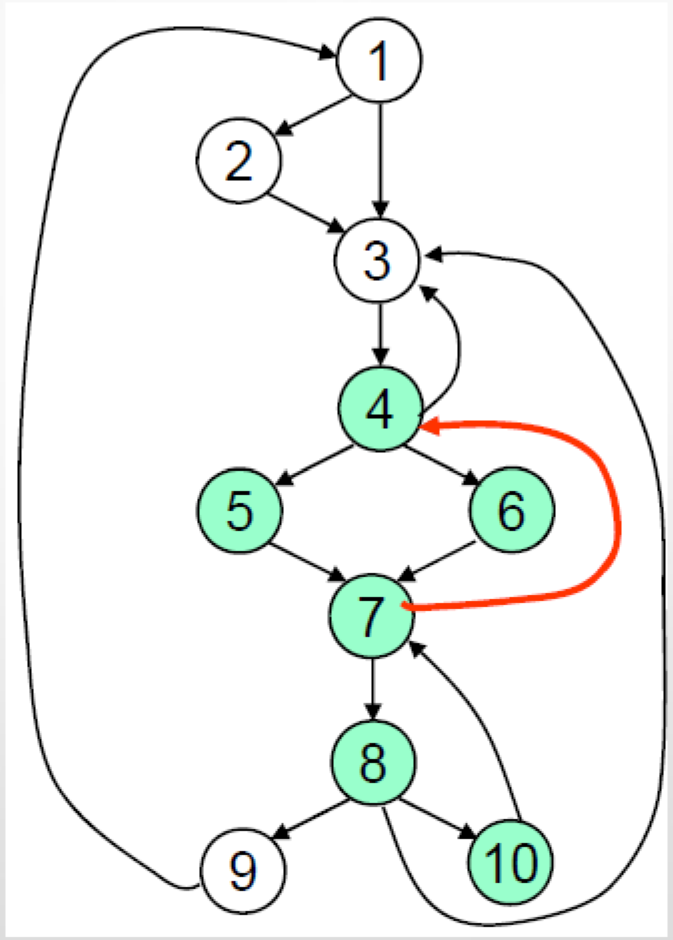
An Example

Find all back edges in this graph and the natural loop associated with each back edge

(9,1)   Entire graph

(10,7) {7,8,10}

(7,4) {4,5,6,7,8,10}



An Example

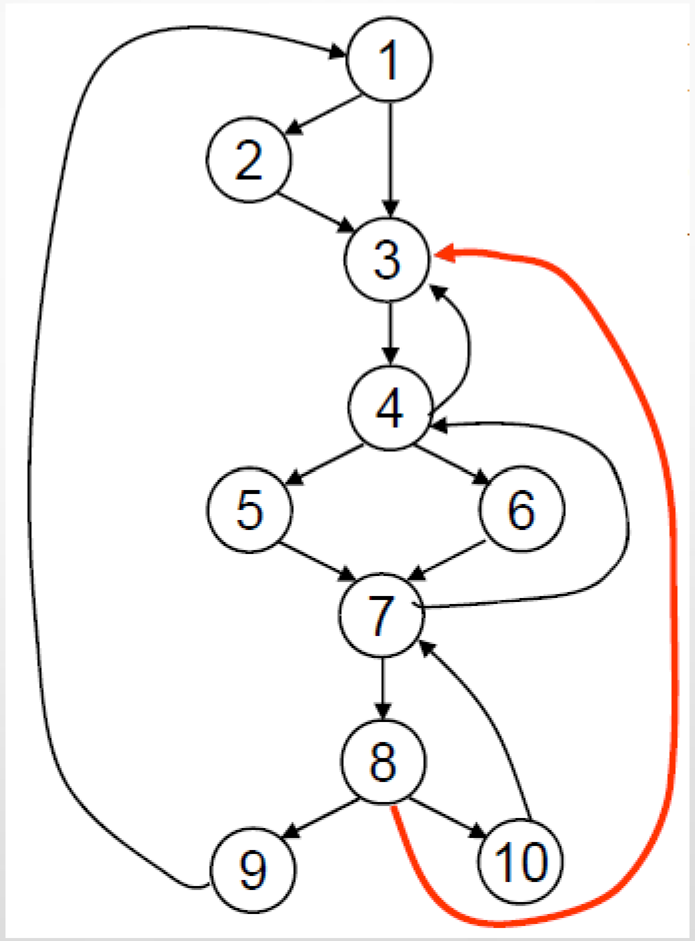
Find all back edges in this graph and the natural loop associated with each back edge

(9,1) Entire graph

(10,7) {7,8,10}

(7,4) {4,5,6,7,8,10}

(8,3)



An Example

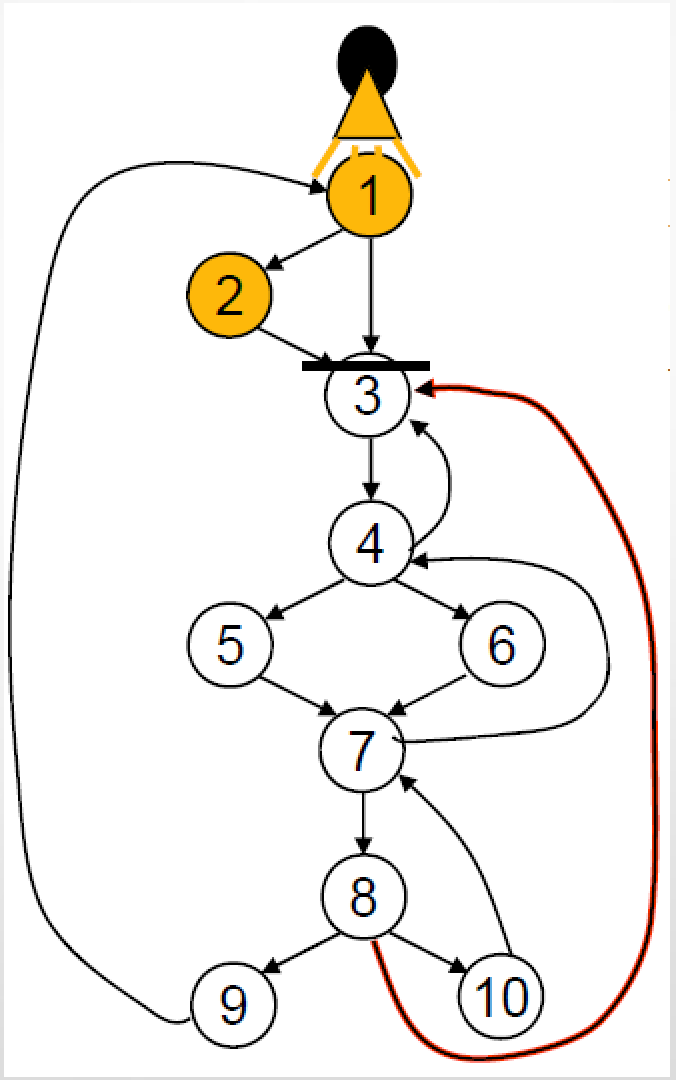
Find all back edges in this graph and the natural loop associated with each back edge

(9,1) Entire graph

(10,7) {7,8,10}

(7,4) {4,5,6,7,8,10}

(8,3)



An Example

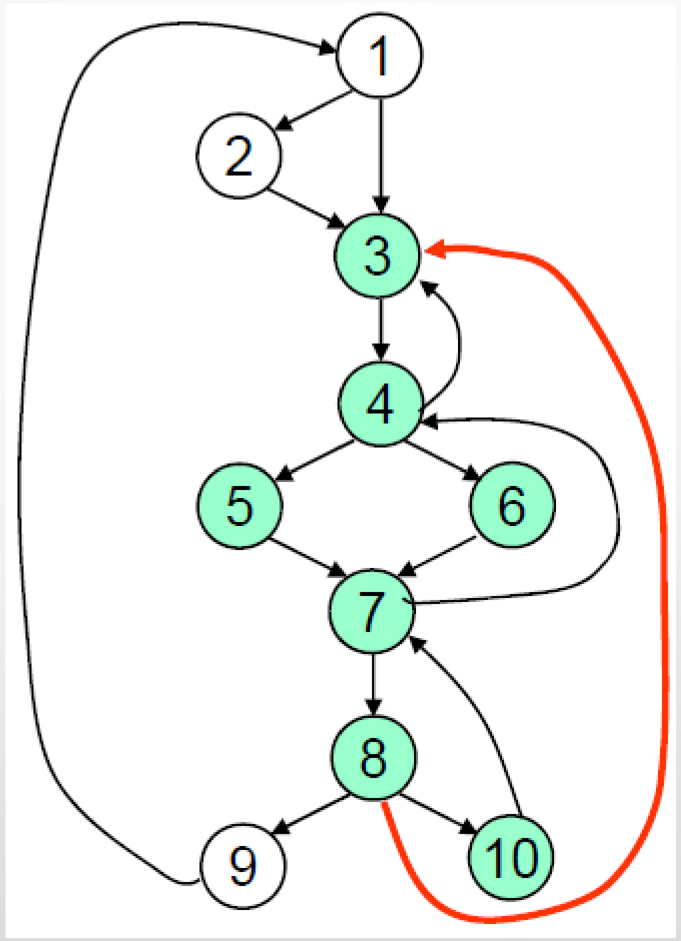
Find all back edges in this graph and the natural loop associated with each back edge

(9,1)   Entire graph

(10,7) {7,8,10}

(7,4) {4,5,6,7,8,10}

(8,3) {3,4,5,6,7,8,10}



An Example

Find all back edges in this graph and the natural loop associated with each back edge

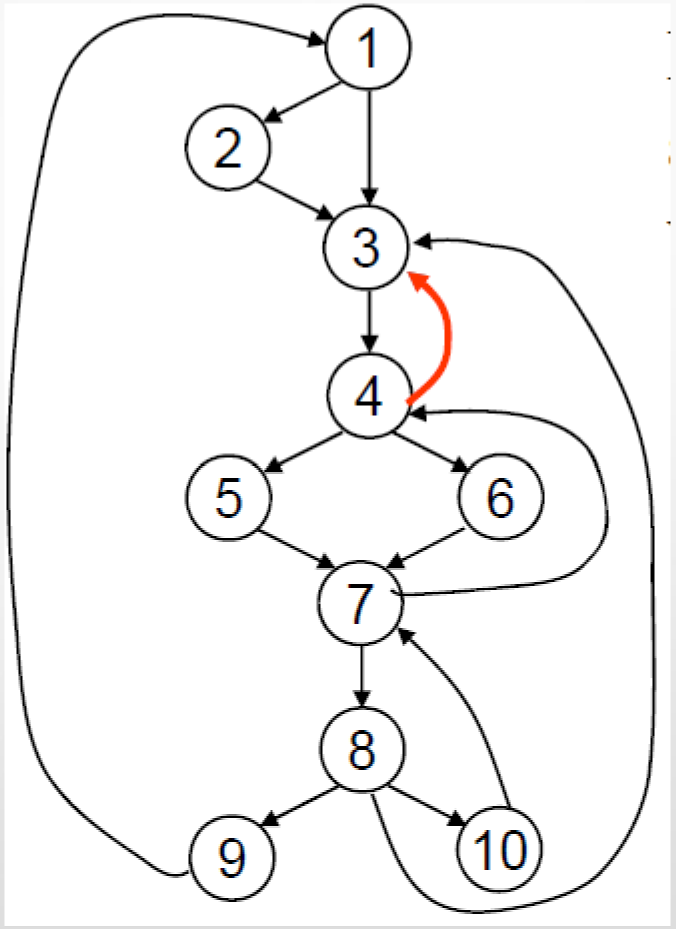
(9,1) Entire graph

(10,7) {7,8,10}

(7,4) {4,5,6,7,8,10}

(8,3) {3,4,5,6,7,8,10}

(4,3)



An Example

Find all back edges in this graph and the natural loop associated with each back edge

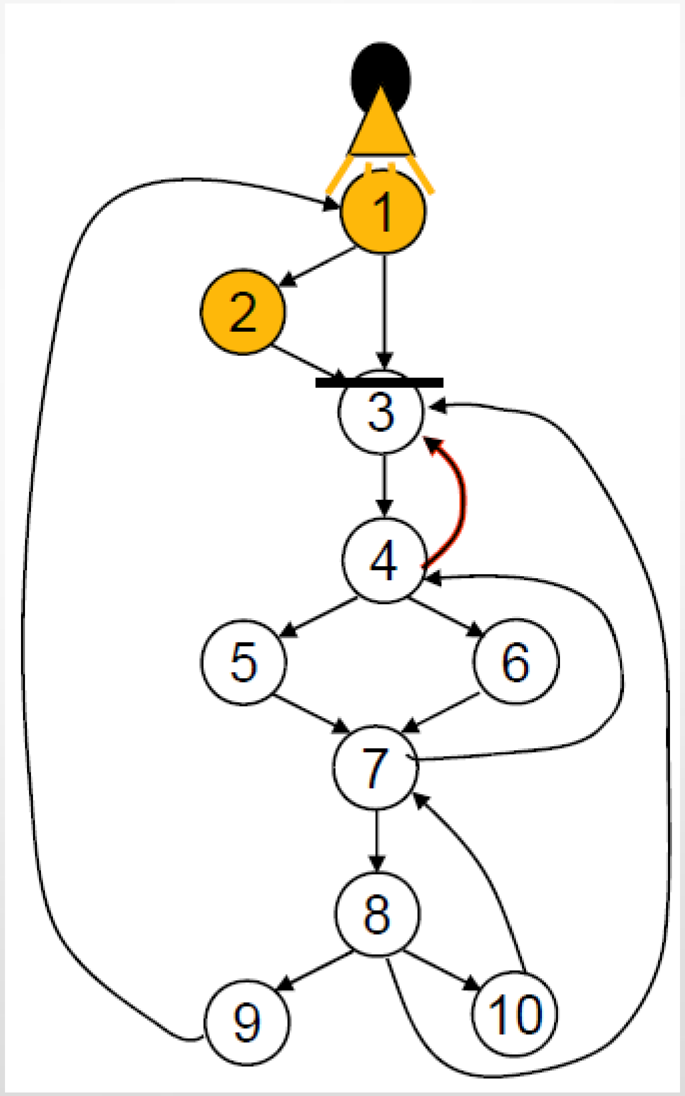
(9,1) Entire graph

(10,7) {7,8,10}

(7,4) {4,5,6,7,8,10}

(8,3) {3,4,5,6,7,8,10}

(4,3)



An Example

Find all back edges in this graph and the natural loop associated with each back edge

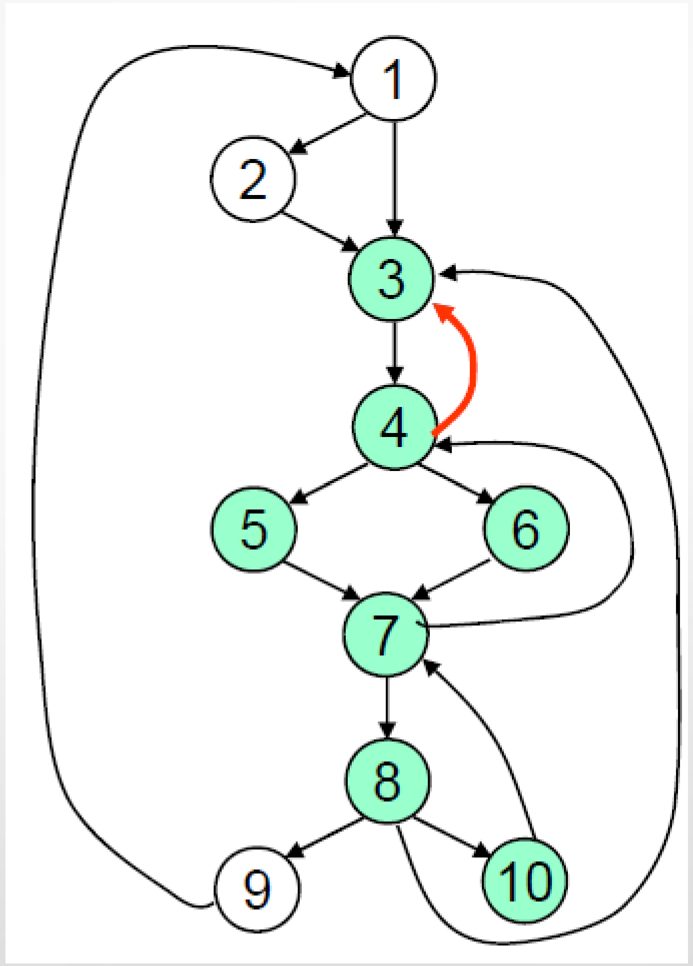
(9,1) Entire graph

(10,7) {7,8,10}

(7,4) {4,5,6,7,8,10}

(8,3) {3,4,5,6,7,8,10}

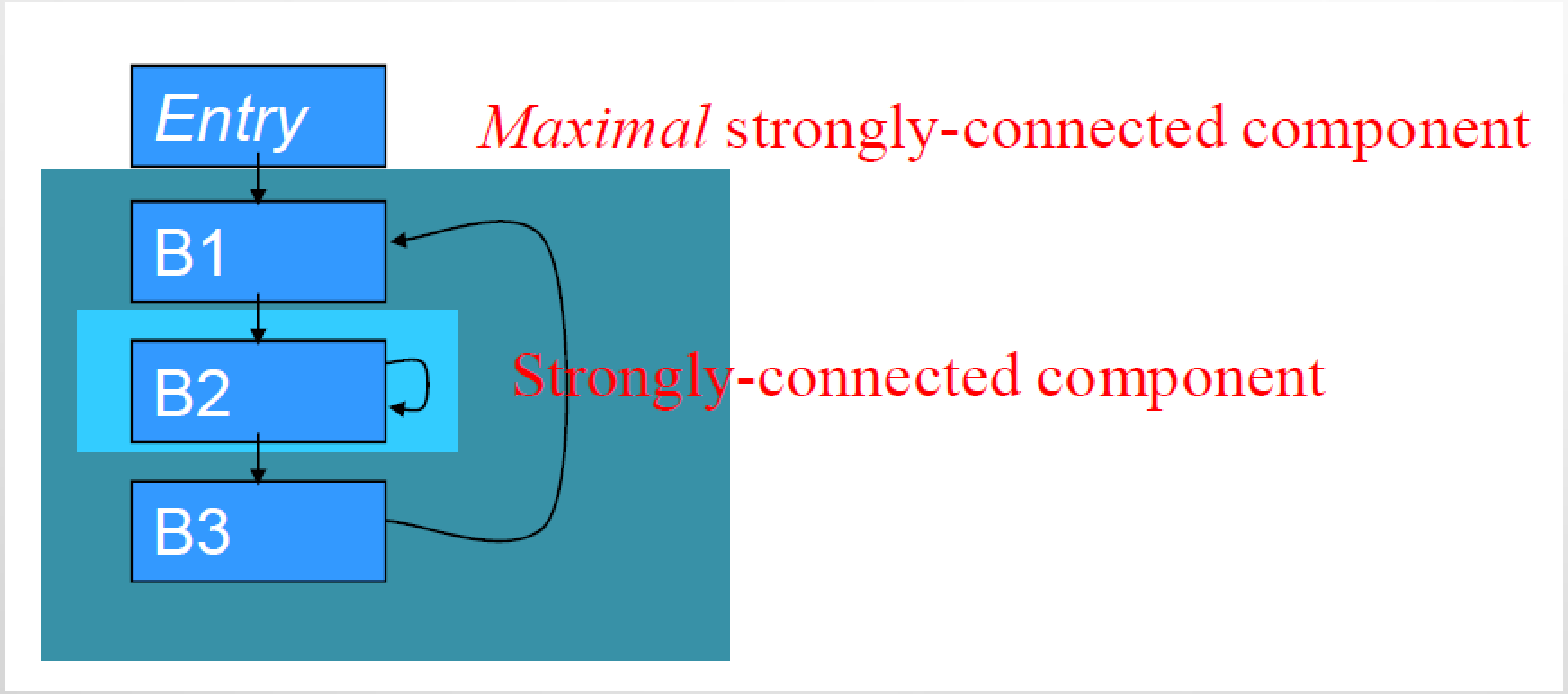
(4,3) {3,4,5,6,7,8,10}



General Looping Structure

- Most general loop form = **strongly-connected component (SCC)**:
    - subgraph S such that every node in S is reachable from every other node by path including only edges in S
- **Maximal SCC**:
    - S is maximal SCC if it is the largest SCC that contains S.
- Tarjan's algorithm [Tar72]:
    - Computes all maximal SCCs
    - Linear-time (in number of nodes and edges)

SCC Example


Entry
B1
B2
B3
Maximal strongly-connected component
Strongly-connected component

Regions

- A ***region*** is a set of nodes N that include a ***header*** with the following properties:

  (i) the header must dominate all the nodes in the region;

  (ii) All the edges between nodes in N are in the region (except for some edges that enter the header);

- A ***loop*** is a special region that had the following additional properties:

  (i) it is strongly connected;

  (ii) All back edges to the header are included in the loop;

- Typically we are interested on studying the data flow into and out of regions. For instance, which definitions reach a region.

Reducibility

- **Natural loops**:
  - no jumps into middle of loop
  - entirely disjoint or nested

- **Reducible**:
  - A flowgraph considered to be *reducible* (or say *well-structured*) if applying a sequence of transformations ultimately reduces it to a single node.
  - A flowgraph is reducible *iff* all loops in it are natural

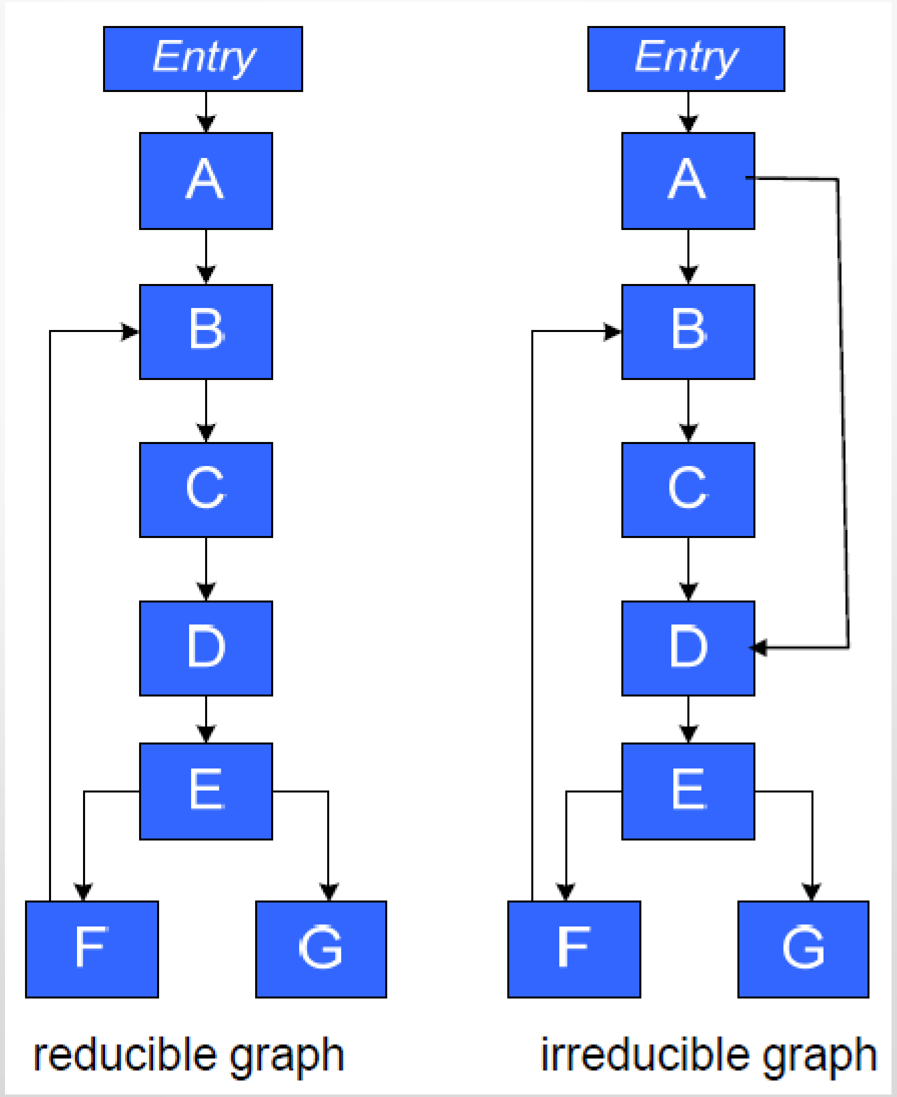
- Certain control-flow patterns (called ***improper regions***) make flowgraphs irreducible. In general, they are multiple-entry strongly connected components of a flowgraph.

Reducibility Example

- Some languages only permit procedures with reducible flowgraphs (e.g., Java)
- "GOTO Considered Harmful":

introduces irreducibility

  - FORTRAN



Interval Analysis in Control Flow-Analysis

- In control-flow analysis, interval analysis refers to
  - Dividing the flowgraph into regions of various sorts
  - Consolidating each region into a new node (called *abstract node*)
  - Replacing the edges entering or leaving the region with edges entering or leaving the corresponding abstract node

- A flowgraph resulting from such transformations is called an *abstract flowgraph*.

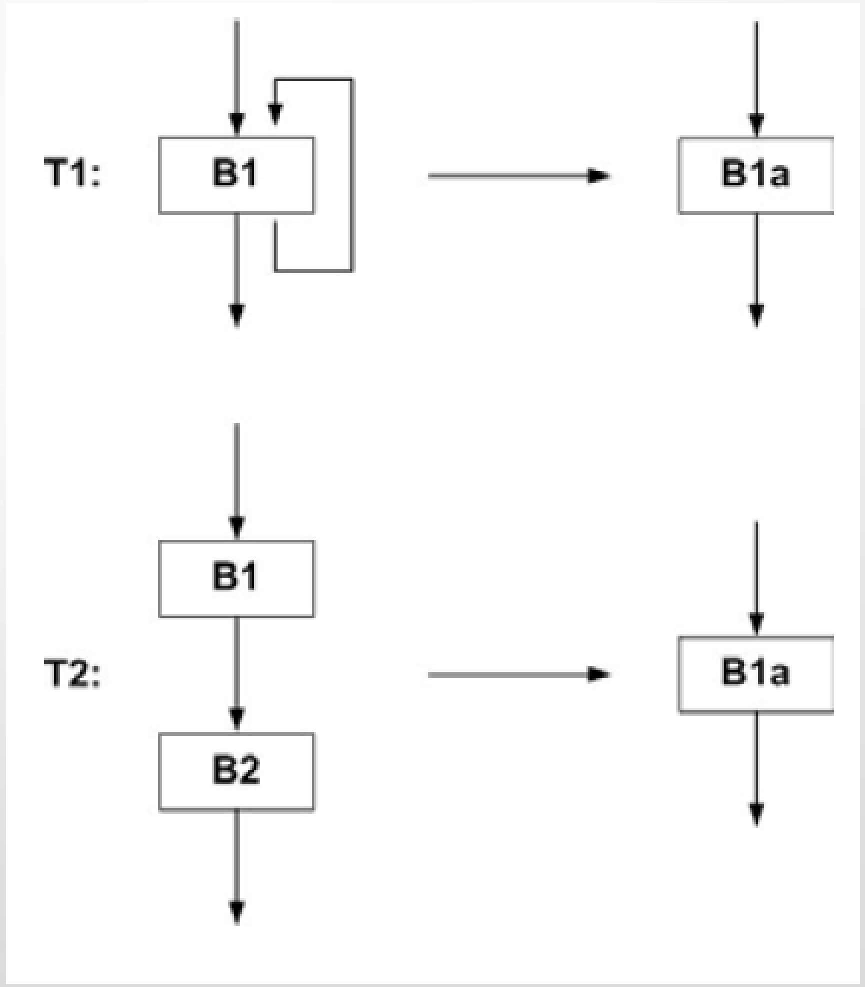
- The result of applying a sequence of such transformations produces a *control tree*.

Control Tree

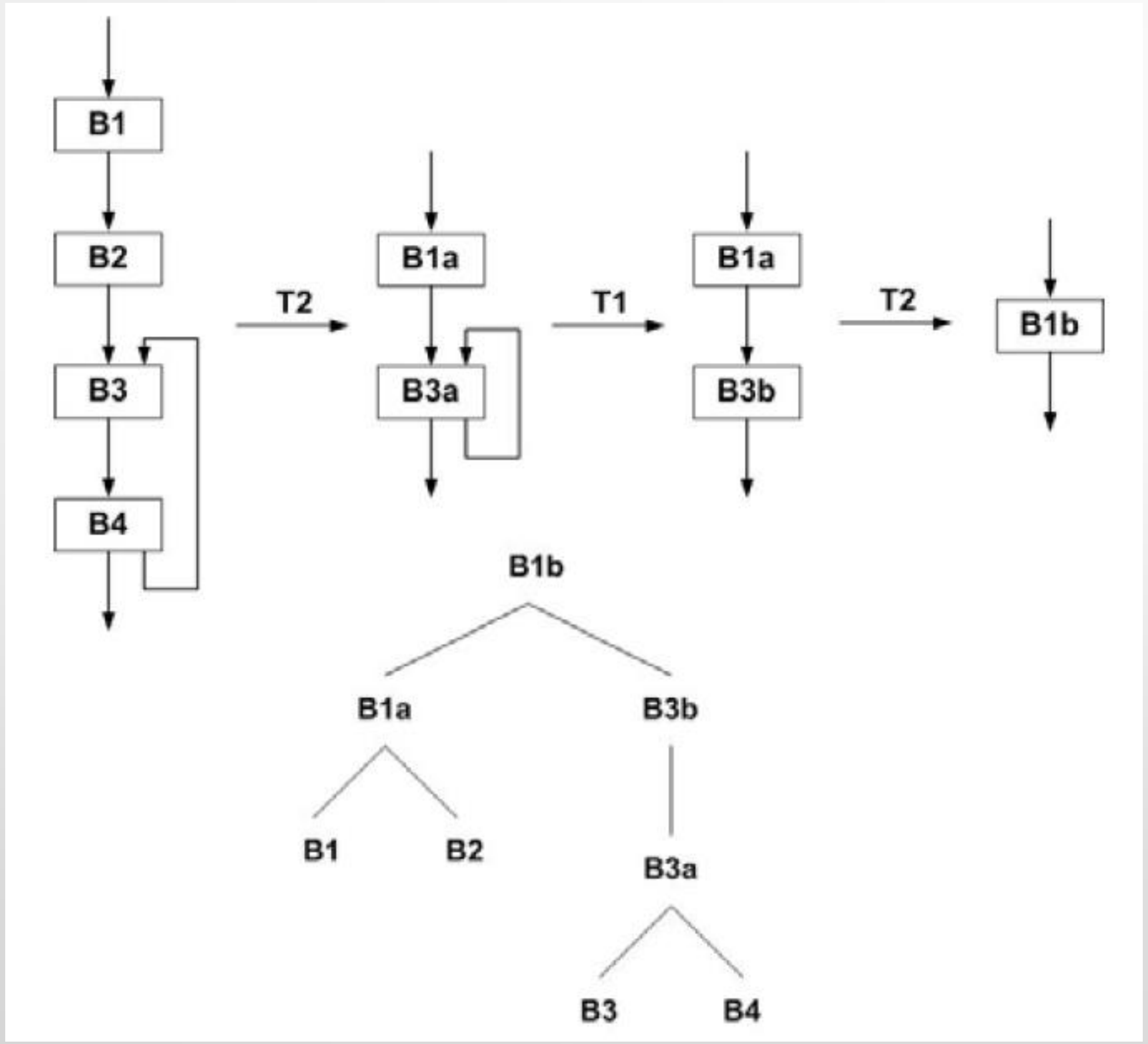
- A control tree is defined as follows:
  1. The root of the control tree is an abstract graph representing the original flowgraph.
  2. The leaves of the control tree are individual basic blocks.
  3. The nodes between the root and the leaves are abstract nodes representing regions of the flowgraph.
  4. The edges of the tree represent the relationship between each abstract node and the regions that are its descendants (and that were abstracted to form it).

T1-T2 Transformations

- One of the simplest and historically earliest forms of interval analysis is known as T1-T2 analysis.
  - T1 collapses a one-node self loop to a single node.
  - T2 collapses a sequence of two nodes such that the first is the only predecessor of the second to a single node.



T1-T2 Transformations (cont.)

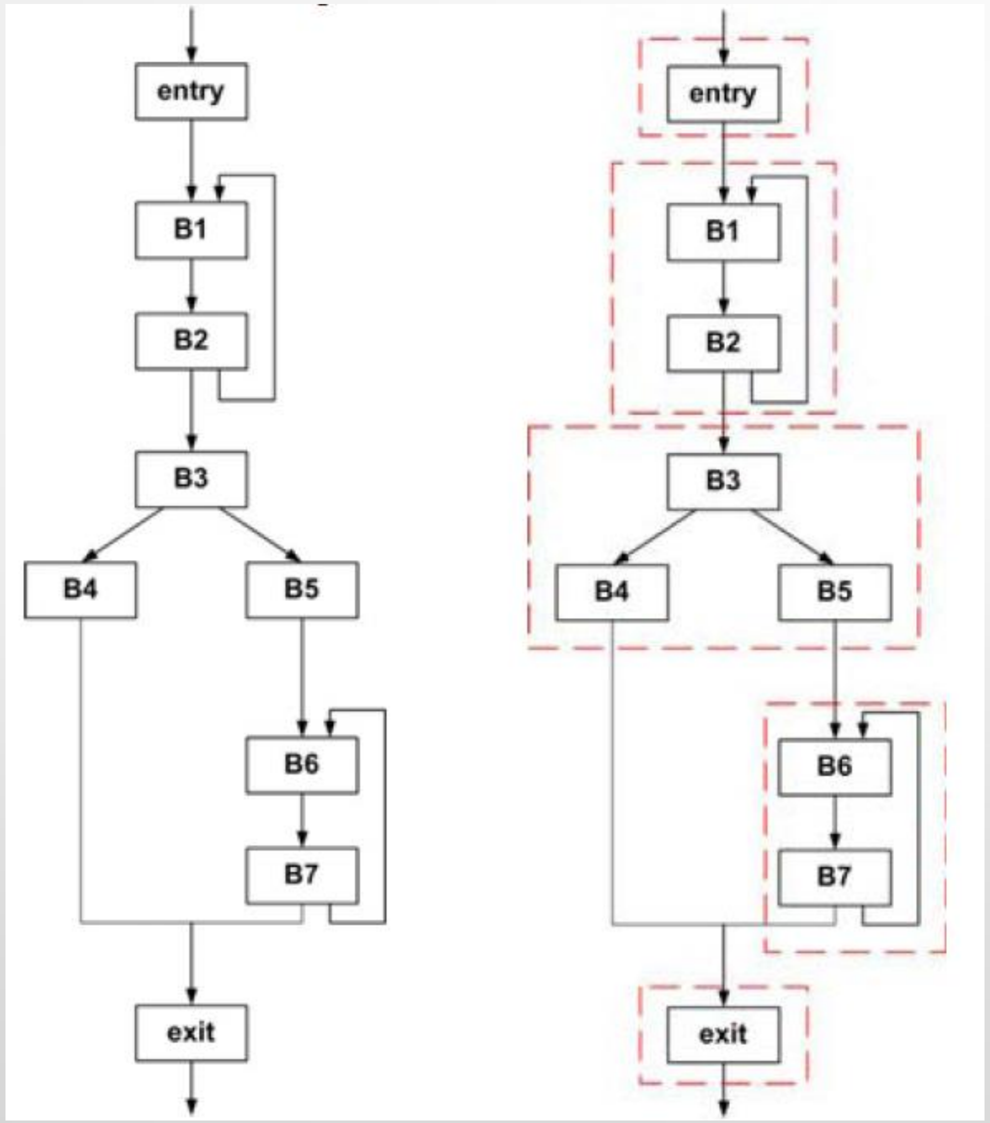


Maximal Interval and Minimal Interval

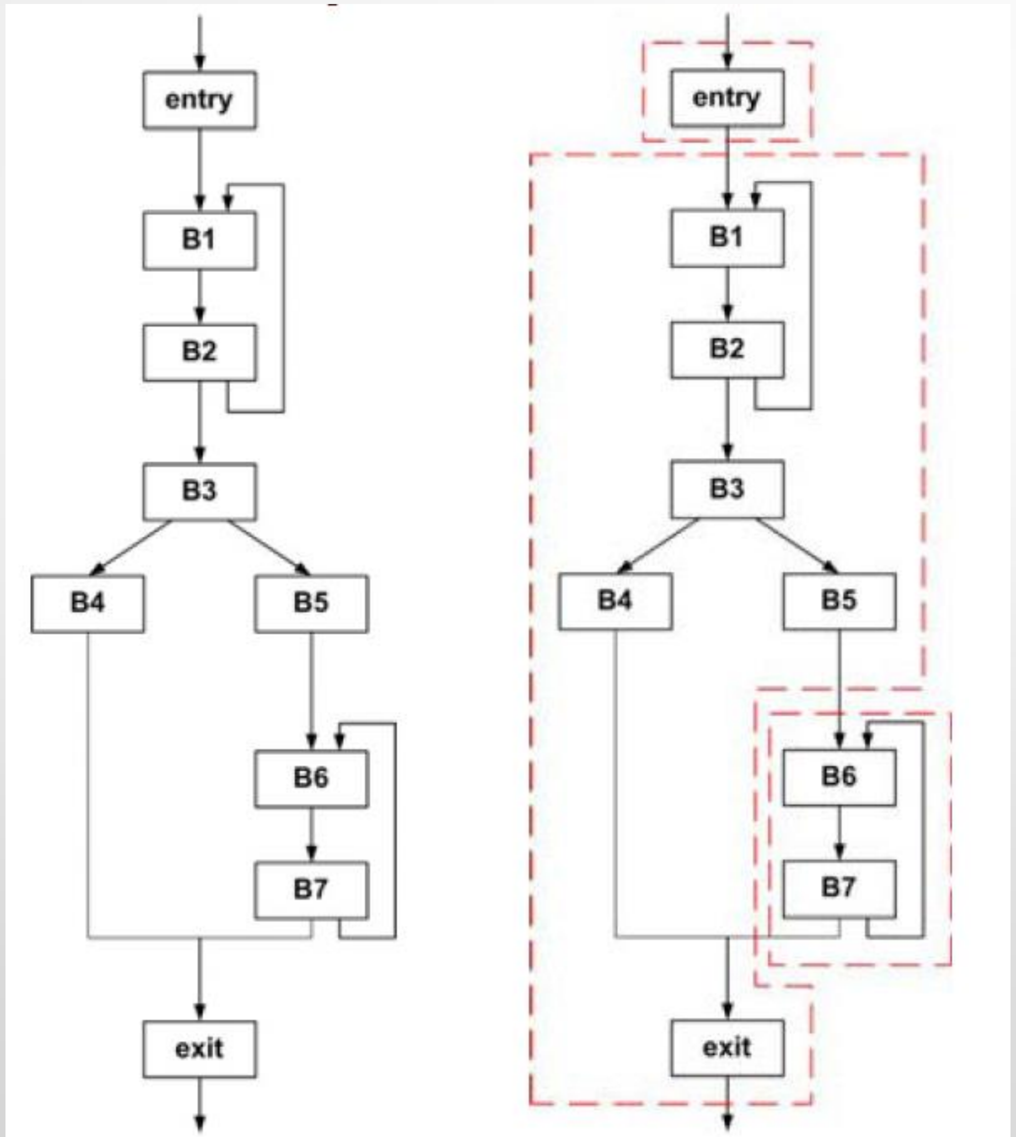
- A maximal interval *IM(h)* with head *h* is the maximal, single-entry subgraph with *h* as its only entry node and with all closed paths in the subgraph containing *h*.
- In essence, a maximal interval is the natural loop with entry *h*, plus some acyclic structure dangling from its exits.

- A minimal interval *I* is defined to be:
  - a) a natural loop
  - b) or a maximal acyclic subgraph
  - c) or a minimal irreducible region

An Example of Minimal Interval



An Example of Maximal Interval



Outline of Interval Analysis based on Minimal Interval

- The basic steps are as follows:

  1. Perform a postorder traversal of the node set of the flowgraph, looking for loop headers and headers of improper regions.

  2. For each loop header found, construct its natural loop and reduce it to an abstract region of type "*natural loop*".

  3. For each set of entries of an improper region, construct the minimal strongly connected component of the flowgraph containing all the entries and reduce it to an abstract region of type "*improper region*".

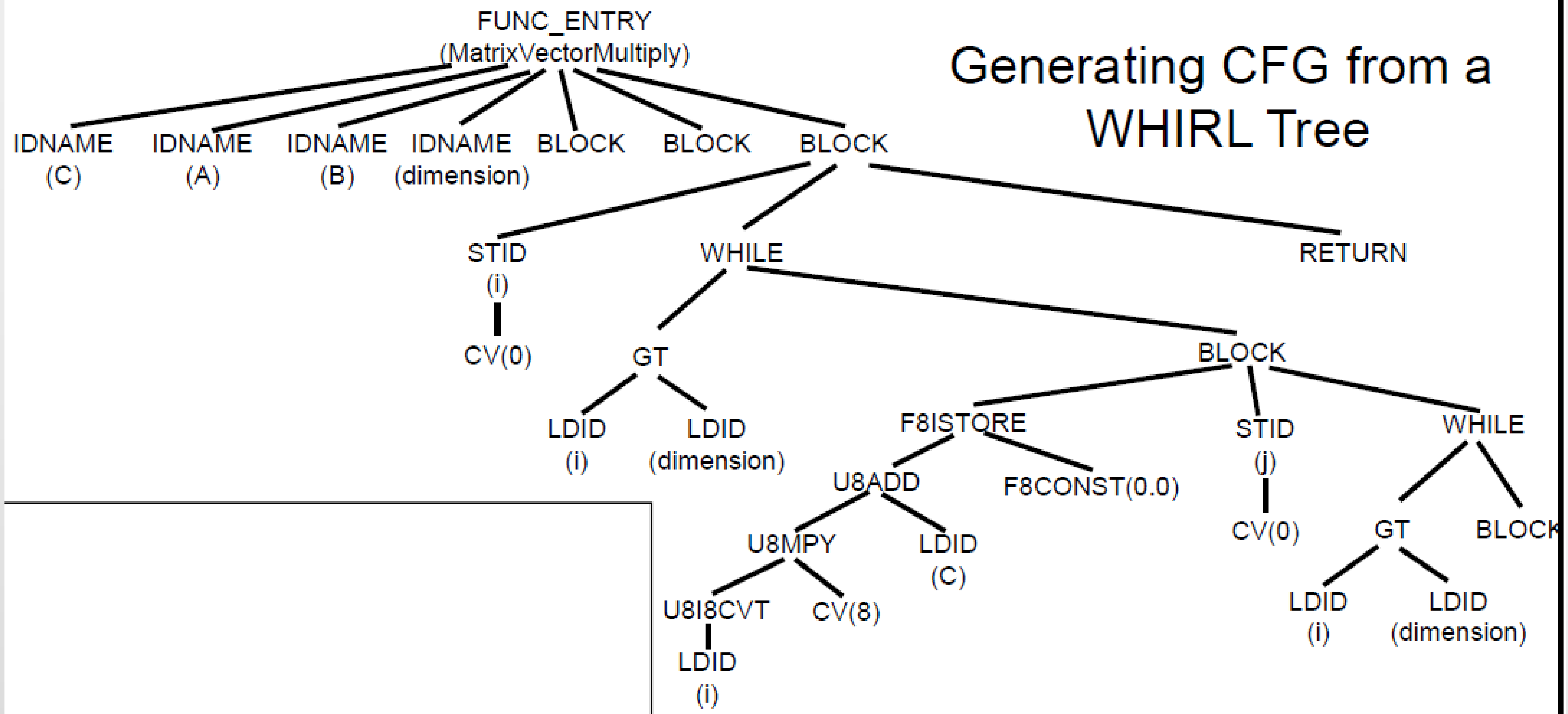
  4. For the entry node and for each immediate descendant of a node in a natural loop or in an irreducible region, construct the maximal acyclic graph with that node as its root; if the resulting graph has more than one node in it, reduce it to an abstract region of type "*acyclic region*".

  5. Iterate this process until it terminates.

Structural Analysis

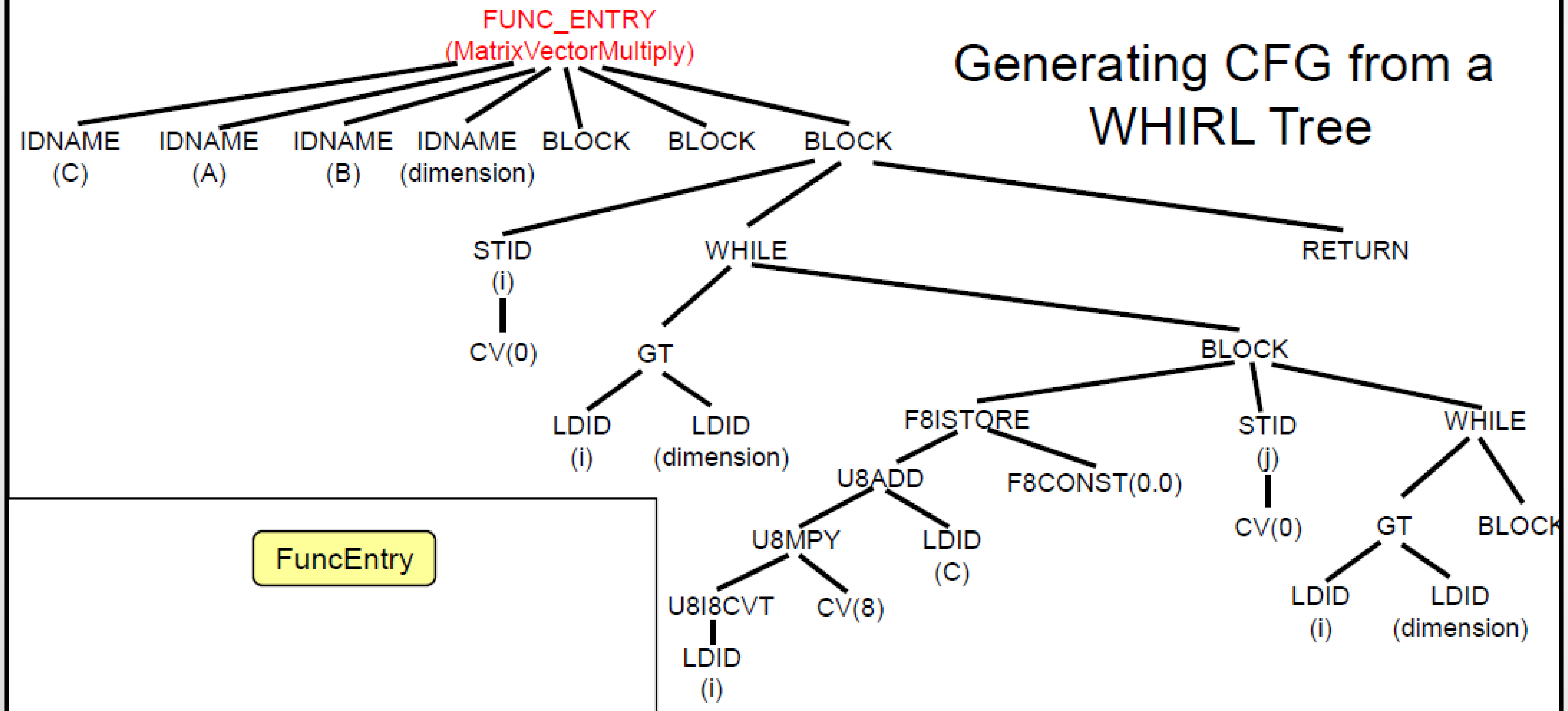
- Interval Analysis (e.g., in the form of minimal intervals) concentrates on identifying the loops in the flowgraph without classifying other types of control structures.
- Structural analysis is a more refined form of interval analysis.
- Compared to basic interval analysis, it identifies many more types of control structures than just loops, forming each into a region. As a result, it provides a basis for doing very efficient data-flow analysis.

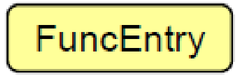
Generating CFG from a WHIRL Tree



```
void MatrixVectorMultiply(double *C, double *A, double *B, int dimension)
{
 int i, j;
 for(i=0 ; i<dimension ; i++)
  {
   C[i] = 0.0;
   for(j=0 ; j<dimension ; j++)
    C[i] = C[i] + A[i*dimension+j]*B[j];
  }
}
```

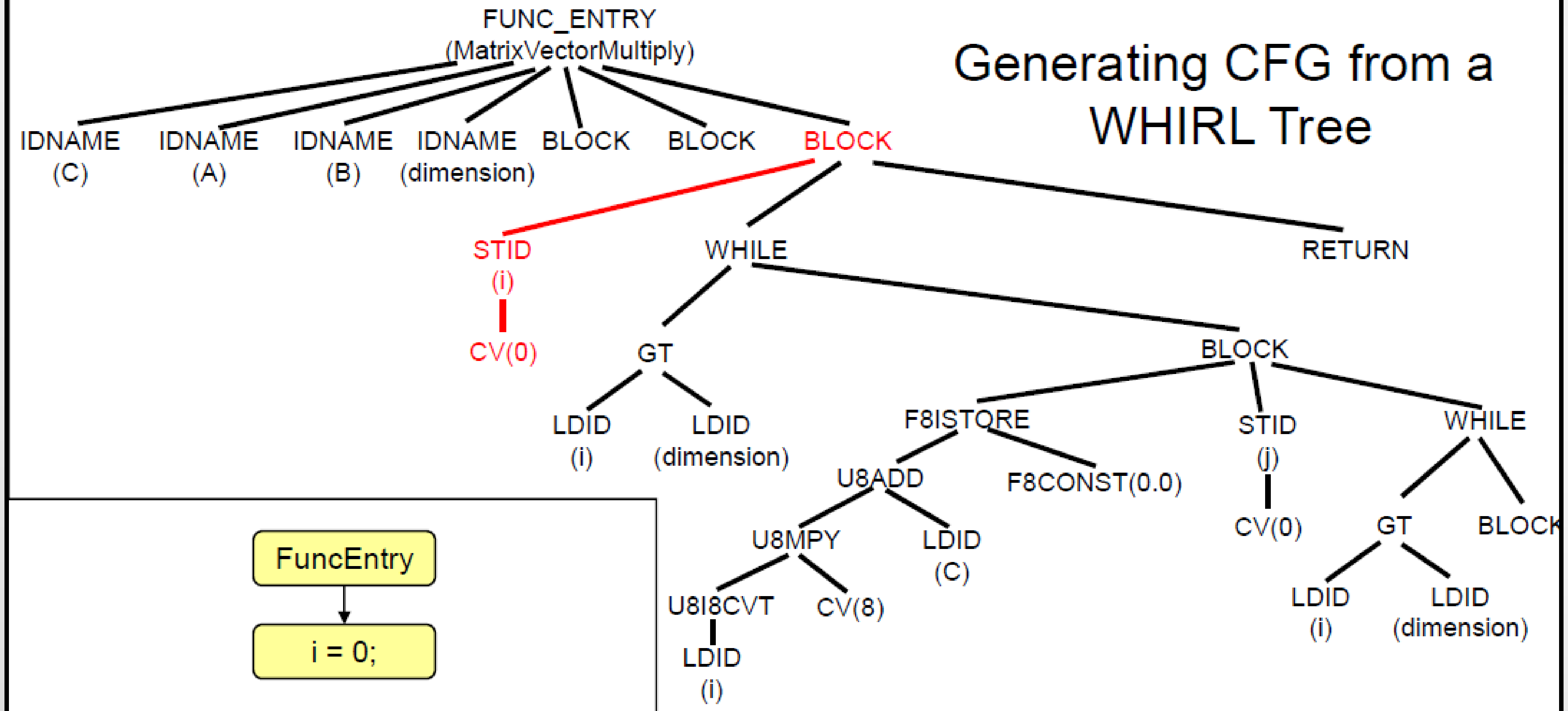
Generating CFG from a WHIRL Tree

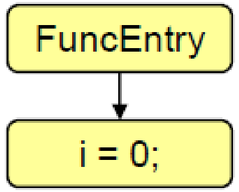




```
void MatrixVectorMultiply(double *C, double *A, double *B, int dimension)
{
 int i, j;
 for(i=0 ; i<dimension ; i++)
  {
   C[i] = 0.0;
   for(j=0 ; j<dimension ; j++)
     C[i] = C[i] + A[i*dimension+j]*B[j];
  }
}
```

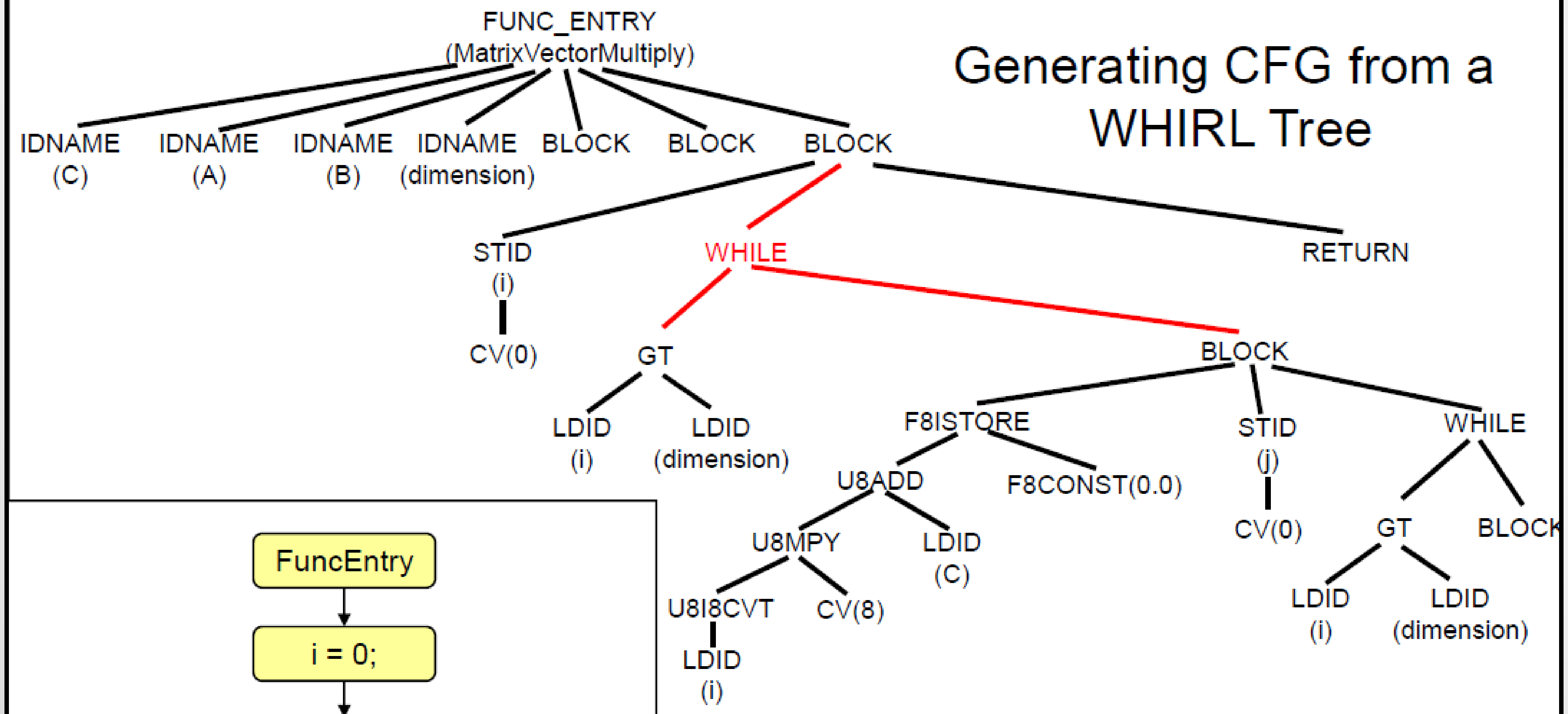
Generating CFG from a WHIRL Tree

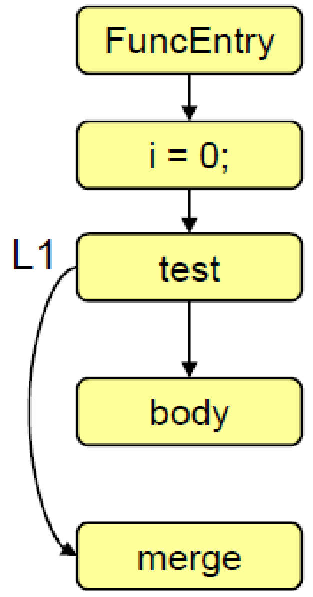




```
void MatrixVectorMultiply(double *C, double *A, double *B, int dimension)
{
 int i, j;
 for(i=0 ; i<dimension ; i++)
   {
     C[i] = 0.0;
     for(j=0 ; j<dimension ; j++)
       C[i] = C[i] + A[i*dimension+j]*B[j];
   }
}
```

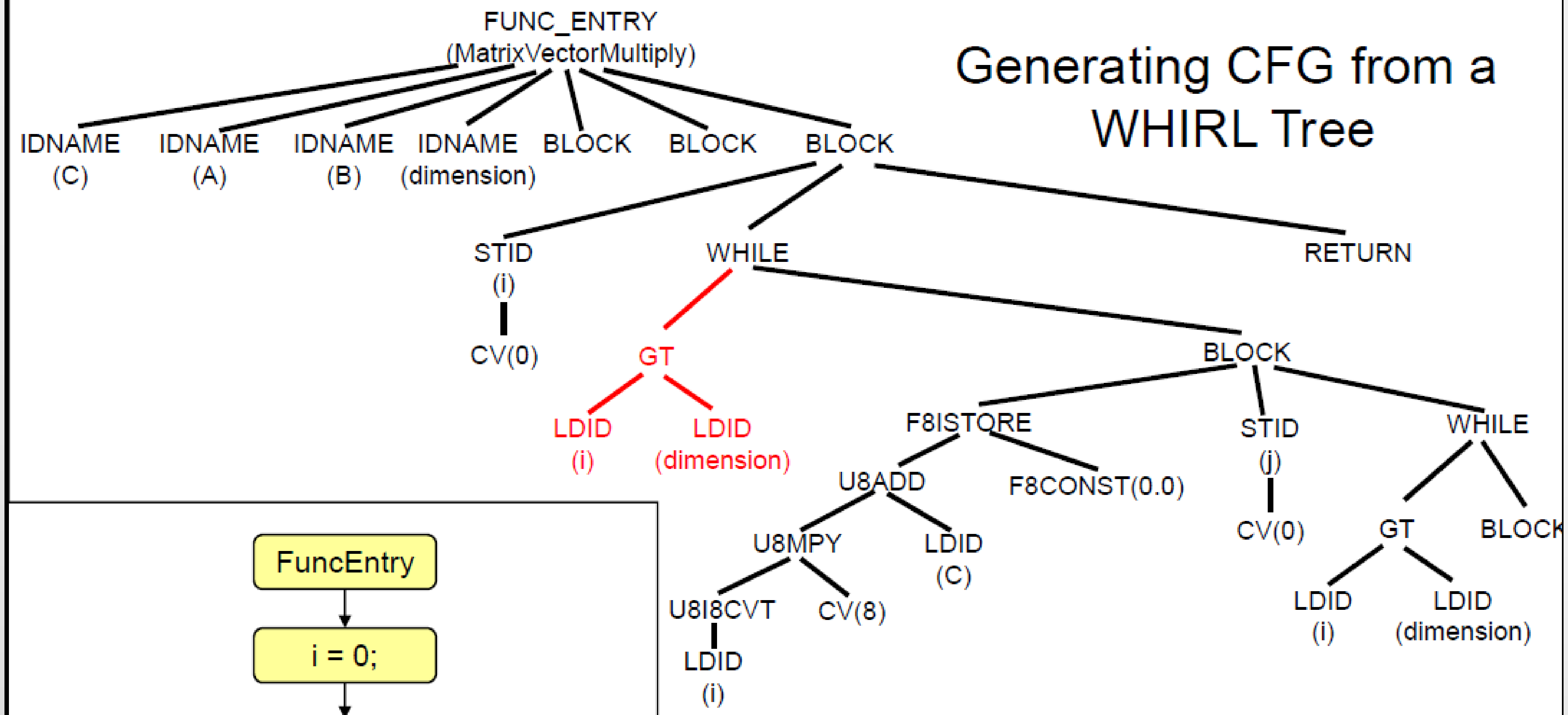
Generating CFG from a WHIRL Tree

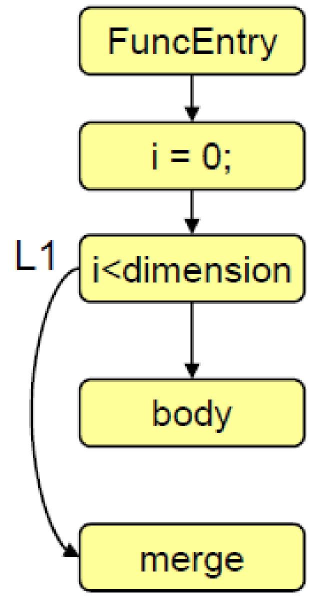




```
void MatrixVectorMultiply(double *C, double *A, double *B, int dimension)
{
  int i, j;
  for(i=0 ; i<dimension ; i++)
   {
     C[i] = 0.0;
     for(j=0 ; j<dimension ; j++)
       C[i] = C[i] + A[i*dimension+j]*B[j];
   }
}
```

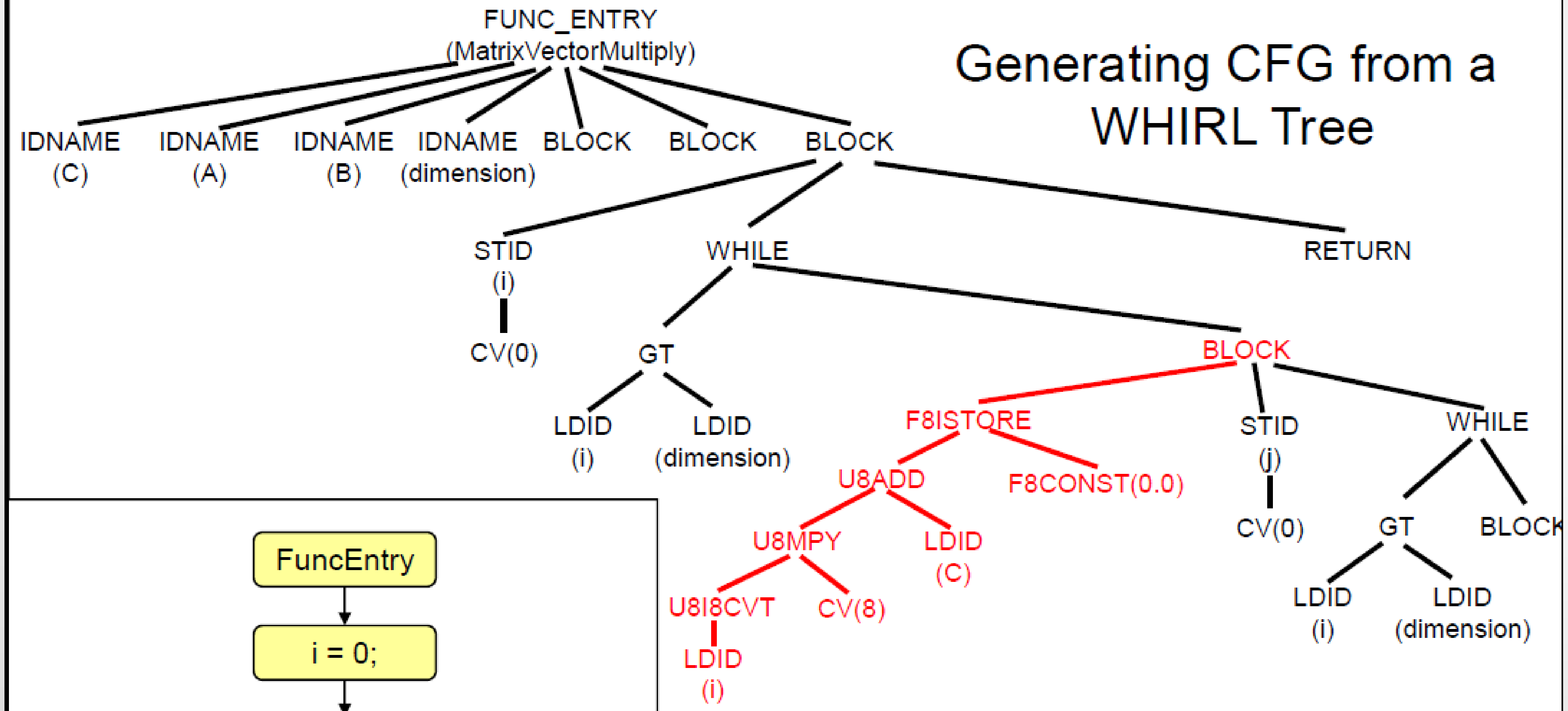
Generating CFG from a WHIRL Tree

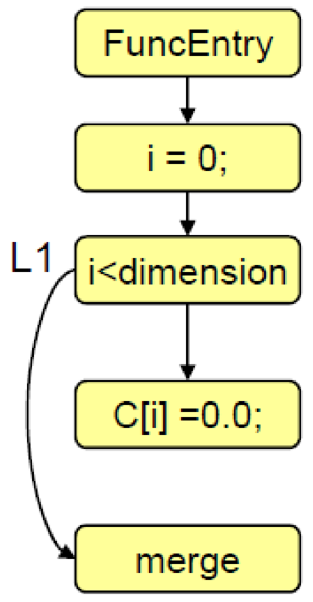




```
void MatrixVectorMultiply(double *C, double *A, double *B, int dimension)
{
  int i, j;
  for(i=0 ; i<dimension ; i++)
   {
     C[i] = 0.0;
     for(j=0 ; j<dimension ; j++)
       C[i] = C[i] + A[i*dimension+j]*B[j];
   }
}
```

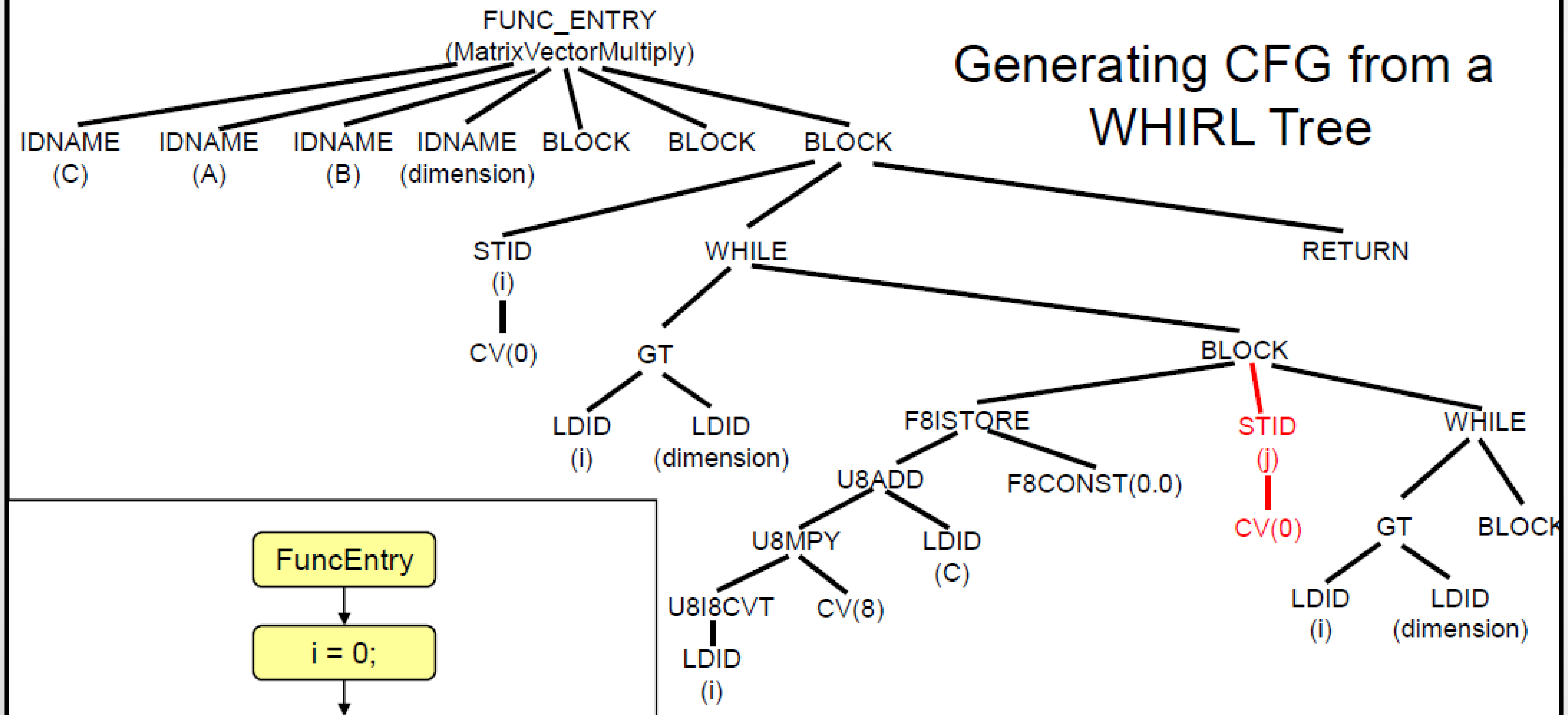
Generating CFG from a WHIRL Tree

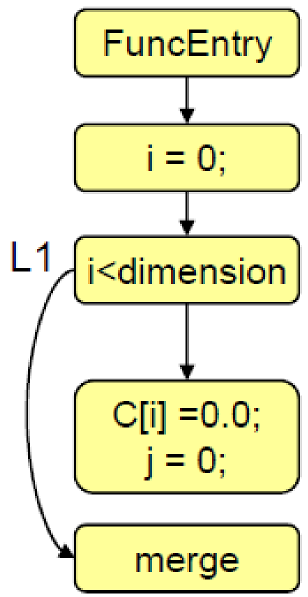




```
void MatrixVectorMultiply(double *C, double *A, double *B, int dimension)
{
  int i, j;
  for(i=0 ; i<dimension ; i++)
   {
     C[i] = 0.0;
     for(j=0 ; j<dimension ; j++)
       C[i] = C[i] + A[i*dimension+j]*B[j];
   }
}
```

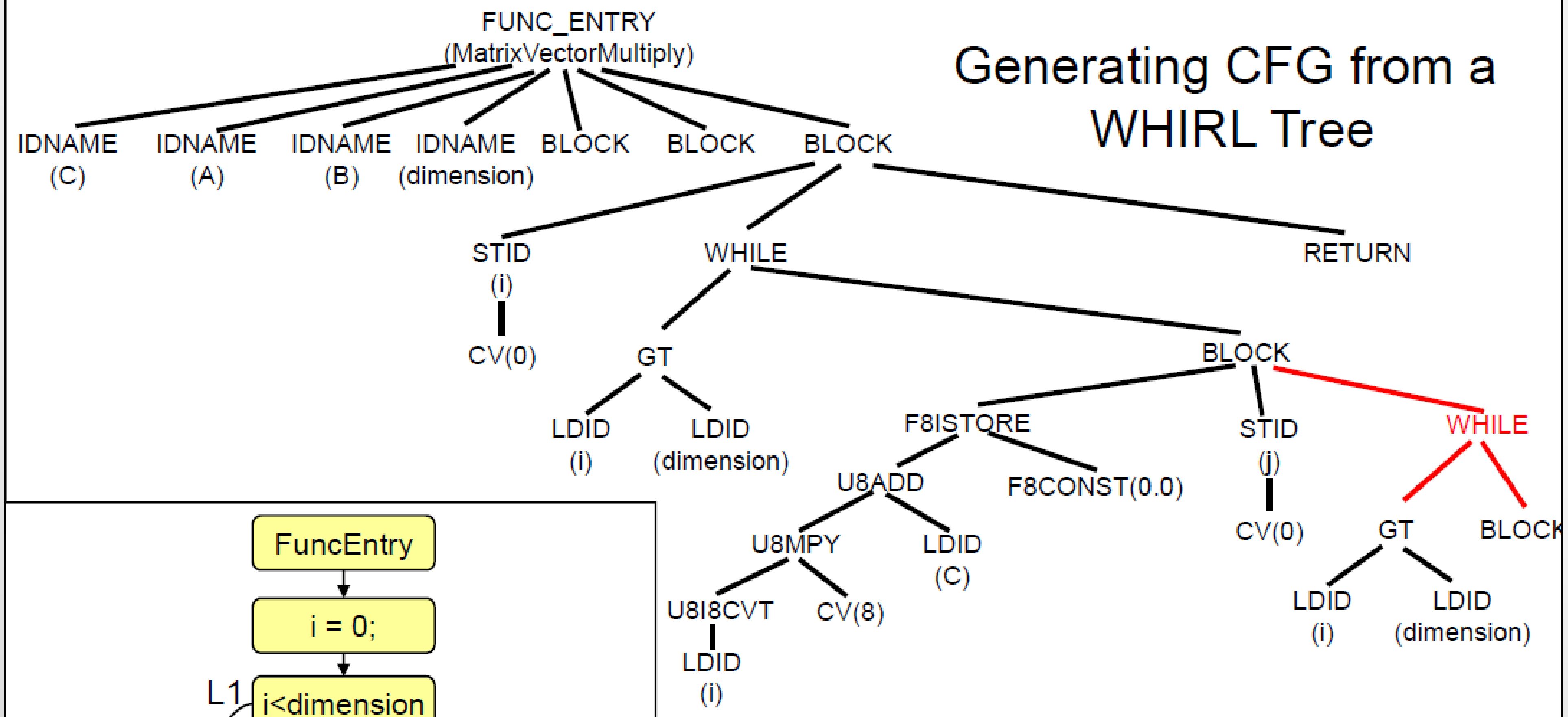
Generating CFG from a WHIRL Tree

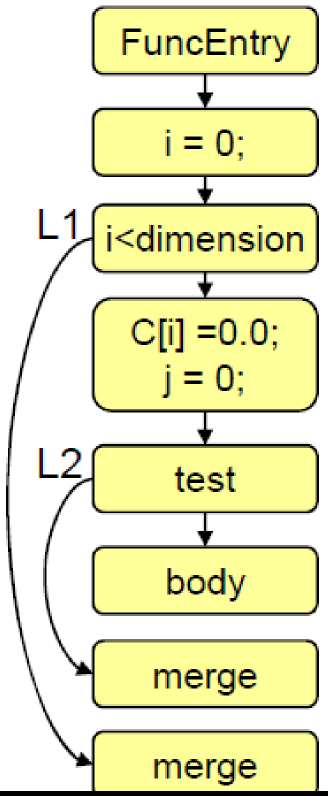




```
void MatrixVectorMultiply(double *C, double *A, double *B, int dimension)
{
 int i, j;
 for(i=0 ; i<dimension ; i++)
  {
   C[i] = 0.0;
   for(j=0 ; j<dimension ; j++)
     C[i] = C[i] + A[i*dimension+j]*B[j];
  }
}
```

Generating CFG from a WHIRL Tree





```
void MatrixVectorMultiply(double *C, double *A, double *B, int dimension)
{
  int i, j;
  for(i=0 ; i<dimension ; i++)
   {
     C[i] = 0.0;
     for(j=0 ; j<dimension ; j++)
       C[i] = C[i] + A[i*dimension+j]*B[j];
   }
}
```

References

- **[LT79]** Thomas Lengauer and Robert E. Tarjan, *A Fast Algorithm for Finding Dominators in a Flowgraph*, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 1, No. 1, pp. 121-141, July, 1979.
- **[AL96]** Stephen Alstrup and Peter W. Lauridsen, *A Simple Dynamic Algorithm for Maintaining a Dominator Tree*, Technical Report, 96/3, Department of Computer Science, University of Copenhagen. Universitetsparken 1, DK-2100 Denmark. 1996.
- **[Tar72]** Robert Tarjan, *Depth-First Search and Linear Graph Algorithms*, SIAM Journal on Computing, Vol. 1, Issue 2, pp. 146- 160, 1972.