Control Flow Analysis

PLT (Fall 2019)

Baishakhi Ray

Representing Control Flow

High-level representation

–Control flow is implicit in an AST

Low-level representation:

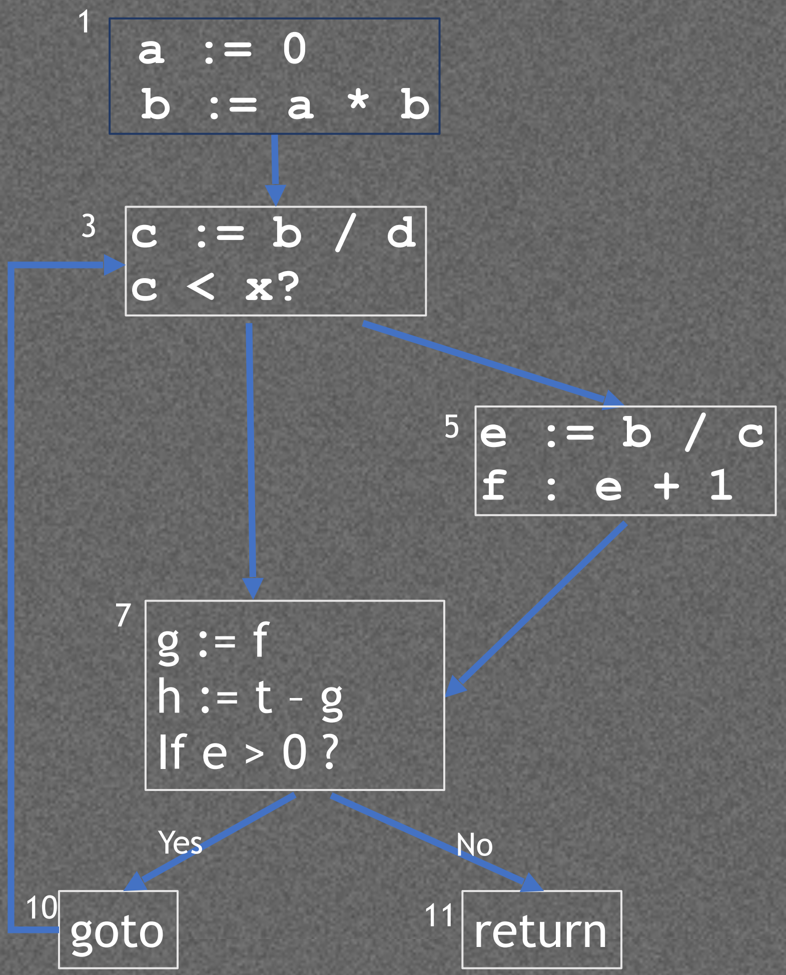
–Use a **Control-flow graph (CFG)**

–Nodes represent statements (low-level linear IR)

–Edges represent explicit flow of control

What Is Control-Flow Analysis?

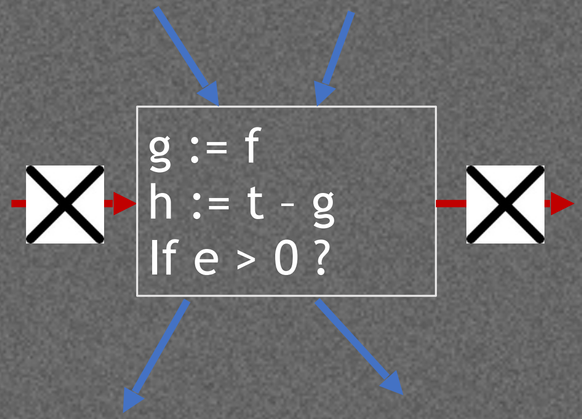
```
1        a := 0
2        b := a * b
3  L1:   c := b/d
4        if c < x goto L2
5        e := b / c
6        f := e + 1
7  L2:   g := f
8        h := t - g
9        if e > 0 goto L3
10       goto L1
11 L3:   return
```



Adopted From U Penn CIS 570: Modern Programming Language Implementation (Autumn 2006)

Basic Blocks

- A **basic block** is a sequence of straight line code that can be entered only at the beginning and exited only at the end



Building basic blocks

- Identify **leaders**
  - The first instruction in a procedure, or
  - The target of any branch, or
  - An instruction immediately following a branch (implicit target)
- Gobble all subsequent instructions until the next leader

Basic Block Example

```
1         a := 0
2         b := a * b
3  L1: c := b/d
4         if c < x goto L2
5         e := b / c
6         f := e + 1
7  L2: g := f
8         h := t - g
9         if e > 0 goto L3
10 goto L1
11 L3:  return
```

Leaders?

- {1, 3, 5, 7, 10, 11}

Blocks?

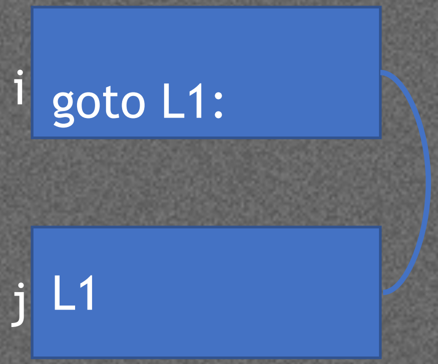
- {1, 2}
- {3, 4}
- {5, 6}
- {7, 8, 9}
- {10}
- {11}

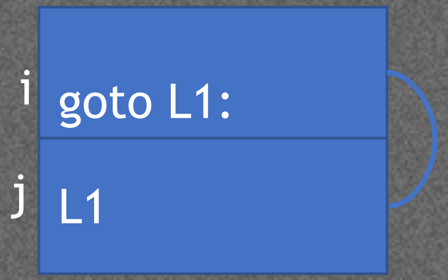
Adopted From U Penn CIS 570: Modern Programming Language Implementation (Autumn 2006)

Building a CFG From Basic Block

Construction

- Each CFG node represents a basic block
- There is an edge from node i to j if
  - Last statement of block i branches to the first statement of j, or
  - Block i does **not** end with an unconditional branch and is immediately followed in program order by block j (fall through)

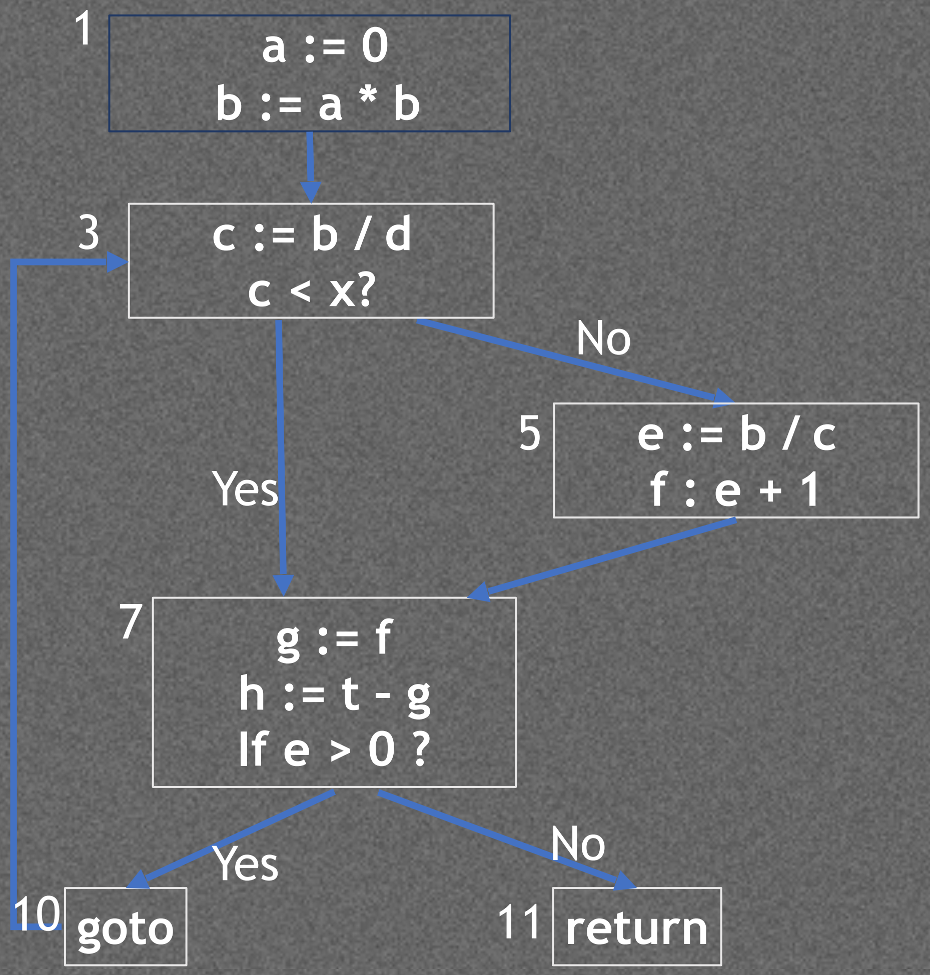




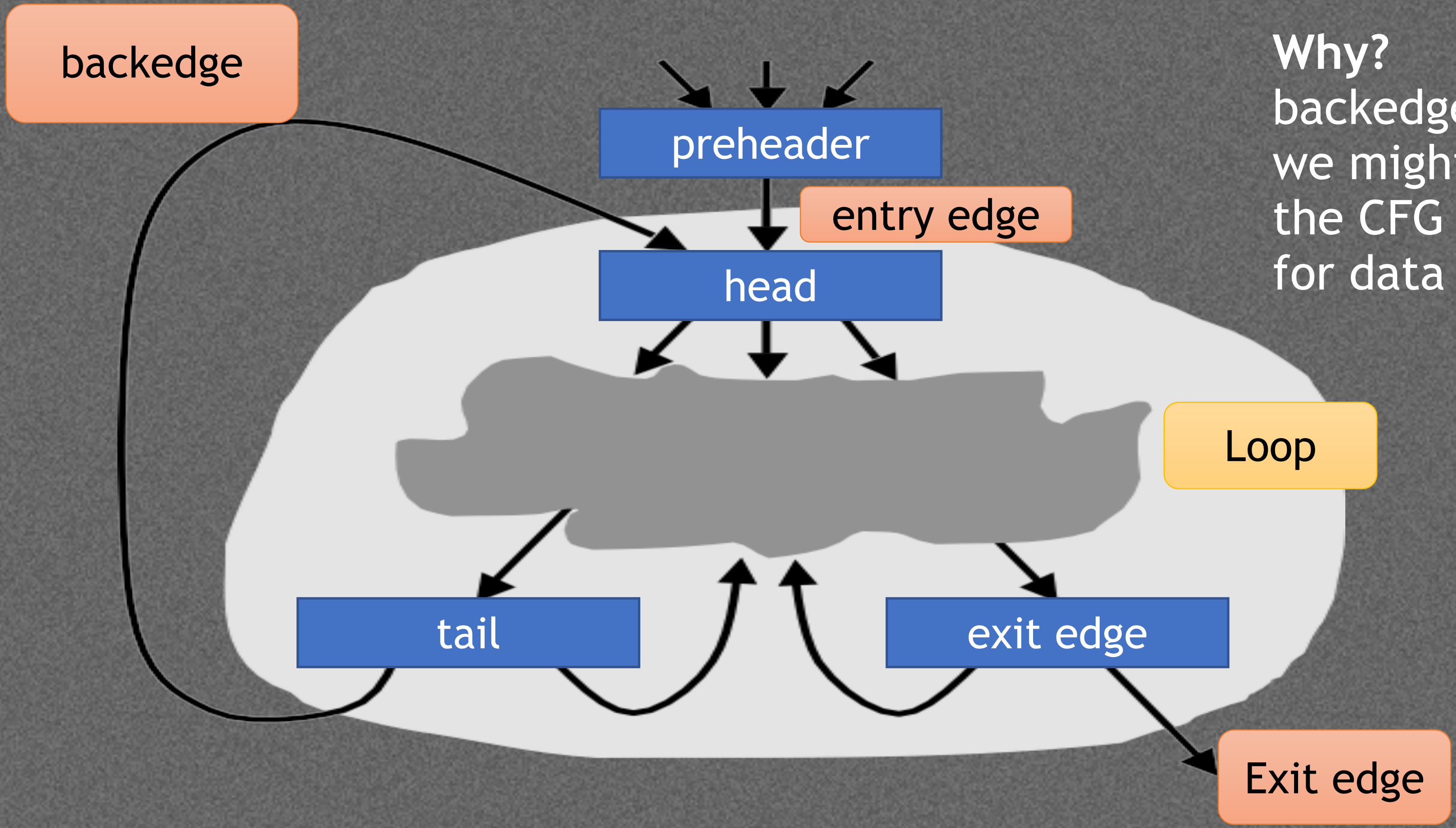
Building a CFG From Basic Block

Construction

- Each CFG node represents a basic block
- There is an edge from node i to j if
  - Last statement of block i branches to the first statement of j, or
  - Block i does **not** end with an unconditional branch and is immediately followed in program order by block j (fall through)

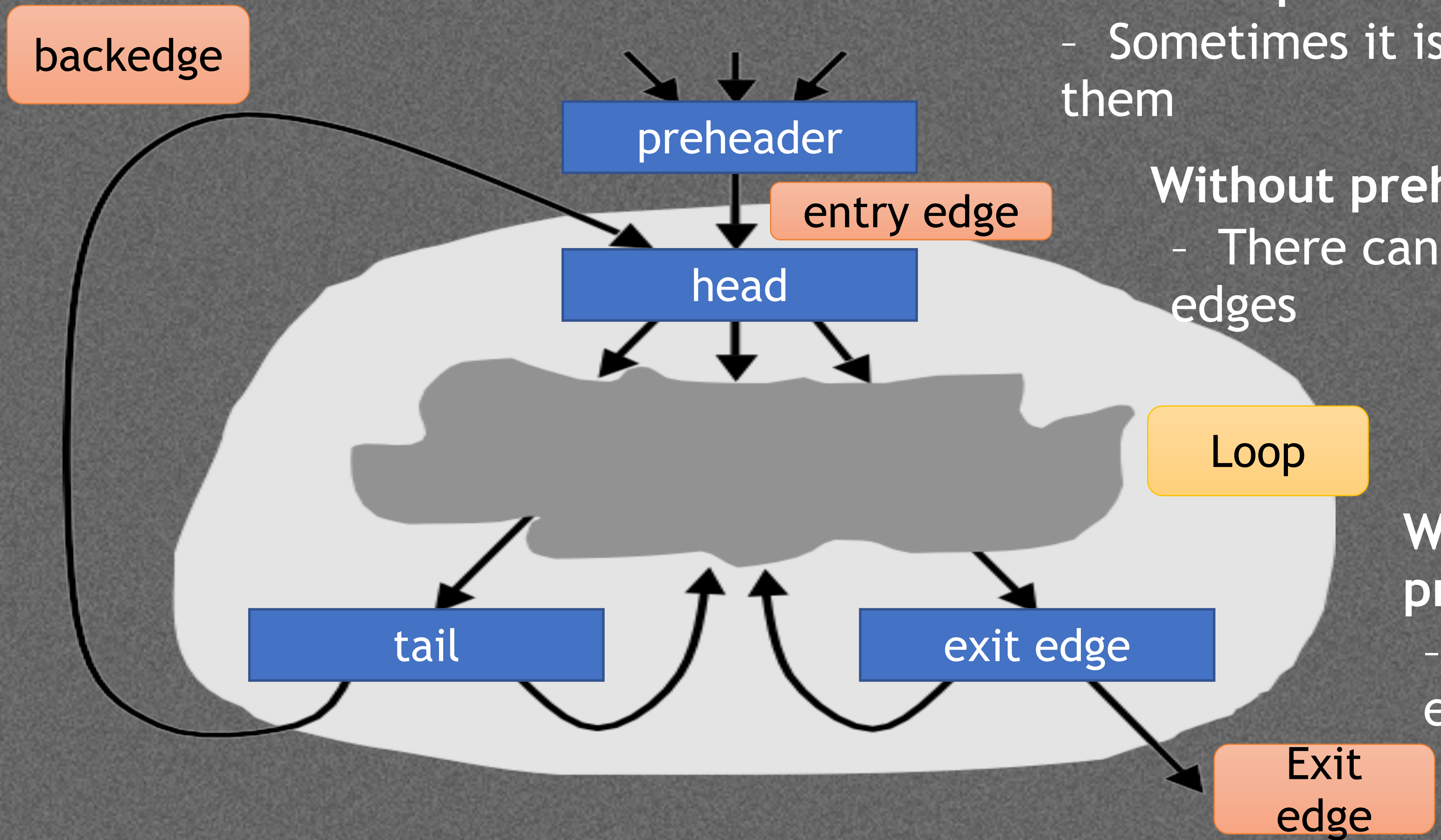


Looping



Why?
backedges indicate that we might need to traverse the CFG more than once for data flow analysis

Looping



Not all loops have preheaders

- Sometimes it is useful to create them

Without preheader node

- There can be multiple entry edges

With single preheader node

- There is only one entry edge

Looping Terminology

Loop:	Strongly connected component of CFG
Loop entry edge:	Source not in loop & target in loop
Loop exit edge:	Source in loop & target not in loop
Loop header node:	Target of loop entry edge

Natural loop: Loop with only a single loop header

Back edge:	Target is loop header & source is in the loop
Loop tail node:	Source of back edge

Looping Terminology

Loop preheader node: Single node that's source of the loop entry edge
Nested loop: Loop whose header is inside another loop

Reducible flow graph: CFG whose loops are all natural loops

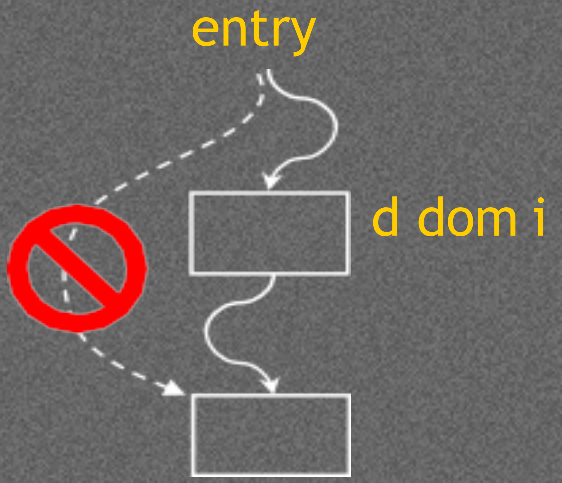
Identifying Loops

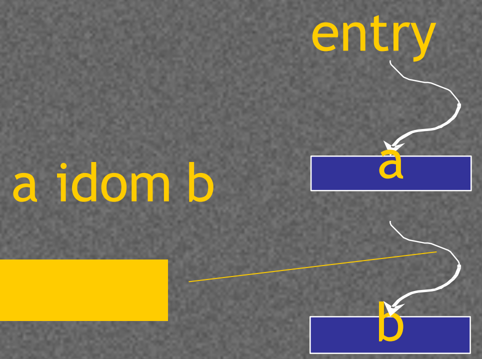
- Why is it important?
  - Most execution time spent in loops, so optimizing loops will often give most benefit

- Many approaches
  - Interval analysis
    - Exploit the natural hierarchical structure of programs
    - Decompose the program into nested regions called intervals
  - Structural analysis: a generalization of interval analysis
  - Identify dominators to discover loops

Dominators

- d **dom** i if all paths from entry to node i include d
- Strict Dominator (d sdom i)
  - If d dom i, but d != i
- Immediate dominator (a idom b)
  - a sdom b and there does not exist any node c such that a != c, c != b, a dom c, c dom b
- Post dominator (p pdom i)
  - If every possible path from i to exit includes p



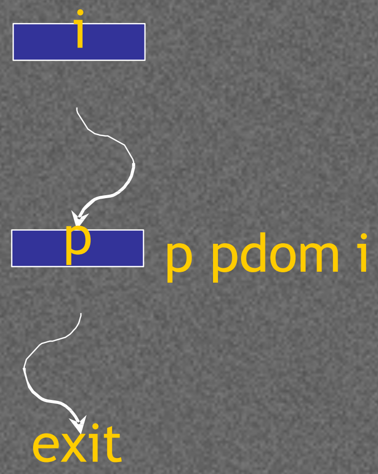


not $ c, a sdom c and c sdom b

Dominators

- **Post dominators (**p **pdom** i**)**

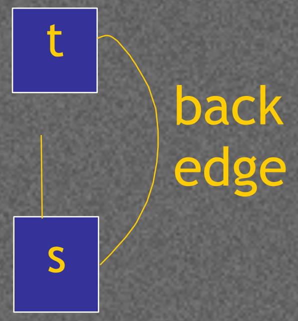
if every possible path from i to exit includes p (p dom i in the flow graph whose arcs are reversed and entry and exit are interchanged)



Identifying Natural Loops and Dominators

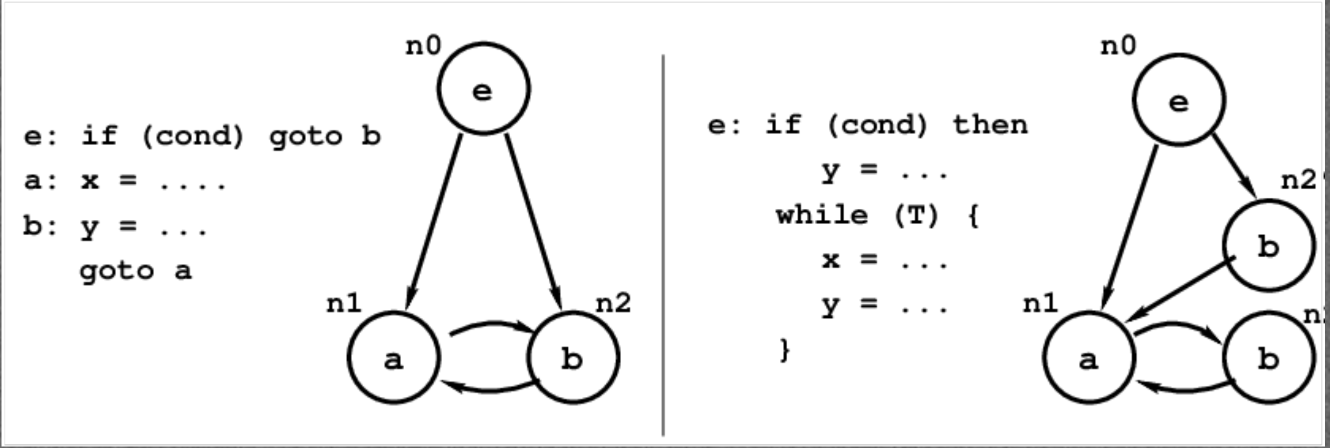
- Back Edge
  - A **back edge** of a natural loop is one whose target of the back edge dominates its source

- Natural Loop
  - The **natural loop** of a back edge (m→n), where n dominates m, is the set of nodes x such that n dominates x and there is a path from x to m not containing n



Reducibility

- A CFG is **reducible** (well-structured) if we can partition its edges into two disjoint sets, the **forward edges** and the **back** edges, such that
  - The forward edges form an acyclic graph in which every node can be reached from the entry node
  - The back edges consist only of edges whose targets dominate their sources
  - Non-natural loops ⇔ irreducibility

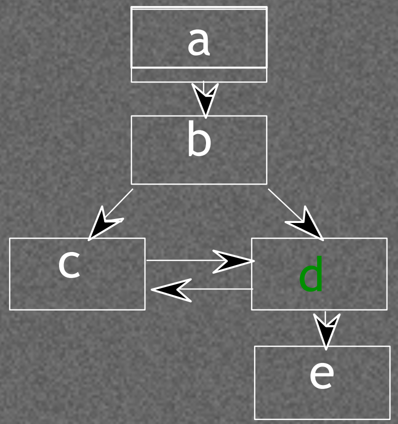


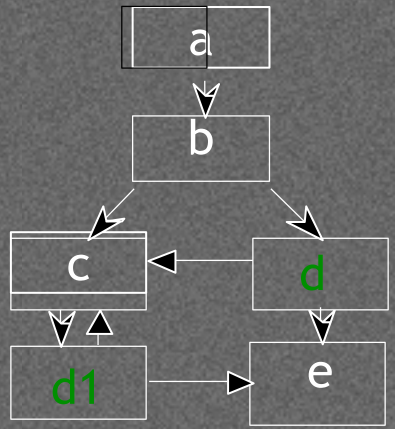
Reducibility

- Structured control-flow constructs give rise to reducible CFGs
- Value of reducibility:
  - Dominance useful in identifying loops
  - Simplifies code transformations (every loop has a single header)
  - Permits interval analysis

Handling Irreducible CFG's

- Node splitting
  - Can turn irreducible CFGs into reducible CFGs





General idea

- Reduce graph (iteratively remove self edges, merge nodes with single pred)
- More than one node => irreducible
  - Split any multi-parent node and start over

Why go through all this trouble?

– We can work on the binary code
– Most modern languages still provide a **goto** statement
– Languages typically provide multiple types of loops. This analysis lets us treat them all uniformly
– We may want a compiler with multiple front ends for multiple languages; rather than translating each language to a CFG, translate each language to a canonical IR and then to a CFG