Program Representations

Last Time

- Why compilers are fun

Announcements

- Lab 1 due Friday 9/11 5 pm
- Lab 1 Q&A 1-2 Wednesday 9/1, Painter 5.38N

Today

- Control flow graphs
- Spanning trees
- Strongly connected components
- Identifying Loops
- Reducible Control Flow Graphs

Sequence of Instructions

```
1          A = 4
2          t1 = A * B

3    L1:   t2 = t1 / C
4          if t2 < W goto L2

5          M = t1 * k
6          t3 = M + I

7    L2:   H = I
8          M = t3 - H
9          if t3 ≥ 0 goto L3

10         goto L1

11   L3:   halt
```

Control Flow Graph

- Divides instructions into **basic blocks**
- Two instructions are in the same basic block *iff* the execution of an instruction in the block guarantees execution can only proceed to the next instruction.
- Edges between basic blocks represent potential flow of control.

More formally, *CFG* $= \langle V, E, Entry \rangle$, where

V = vertices or nodes, representing an instruction or basic block (group of instructions).
E = edges, potential flow of control $E \subseteq V \times V$
$Entry \in V$, unique program entry

For convenience, assume all V are reachable from $Entry$,

$$(\forall v \in V)[Entry \xrightarrow{*} v]$$

Control Flow Graph Construction

Constructing *CFG*s with basic blocks (sets of instructions)

- Identify *Leaders* - first instruction of a basic block
- In lexicographic order, construct a block by appending subsequent instructions up to, but not including, the next leader.

Leader identification

1. First instruction in the program, or
2. target instruction of any conditional or unconditional branch, or
3. the instruction immediately following a conditional or unconditional branch (this instruction is an implicit target).

Basic Block Partition Algorithm

Input: set of instructions,
instr(i) = i^{th} instruction in sequence
Output: set of *leaders*, set of basic blocks where *block(x)* is the set of instructions in the block with *leader x*.
Algorithm:

```
leaders = 1                  // Leaders, first instruction
for i = 1 to |n|             // n = number of instructions
    if instr(i) is a branch then
        leaders = leaders ∪ all potential targets of instr(i)
endfor
worklist = leaders           // Basic blocks
while worklist not empty do
    x = smallest numbered instr in worklist
    worklist = worklist - {x}
    block(x) = {x}
    if instr(x) is a branch then
        last = x
    else {
        for (i = x + 1; i ≤ |n| and i ∉ leaders; i++ )
            block(x) = block(x) ∪ {i}
        endfor
        last = i - 1
    }
endwhile
```


Basic Block Example

```
1         A = 4
2         t1 = A * B

3   L1:   t2 = t1 / C
4         if t2 < W goto L2

5         M = t1 * k
6         t3 = M + I

7   L2:   H = I
8         M = t3 - H
9         if t3 ≥ 0 goto L3

10        goto L1

11  L3:   halt
```

Leaders =

Blocks =

Determining the Edges in a Control Flow Graph

$\exists$ directed edge from B_1 to B_2 if:

1. $\exists$ a branch from the last instruction of B_1 to the first instruction B_2 (B_2 is a leader).
2. B_2 immediately follows B_1 in program order and B_1 does not end with an unconditional branch.

Input: *block()*, a sequence of basic blocks
Output: *CFG* where nodes are basic blocks
Algorithm:

> **for** i = 1 to the number of blocks **do**
>   x = last instruction of *block(i)*
>   **if** *instr(x)* is a branch **then**
>   **for** each target *y* of *instr(x)*
>   create edge from block *i* to block *y*
>   **endfor**
>   **if** *instr(x)* is not an unconditional branch **then**
>   create edge from block *i* to block *i* + 1
> **endfor**

Basic Block Example

```
 1          A = 4
 2          t1 = A * B

 3   L1:    t2 = t1 / C
 4          if t2 < W goto L2

 5          M = t1 * k
 6          t3 = M + I

 7   L2:    H = I
 8          M = t3 - H
 9          if t3 ≥ 0 goto L3

10          goto L1

11   L3:    halt
```

Edges =

Path =

Simple Path =

Cycle =

Data Structures?

Spanning Trees

CFG $= (V_G, E_G, \text{Entry}_G, \text{Exit}_G)$, then we can construct a spanning tree, ST $= \langle V_T, E_T, Root_T, Exit_T \rangle$ with

$V_T = V_G$
$E_T \subseteq E_G$
$Root_T = Entry_G$
$Exit_T = Exit_G$

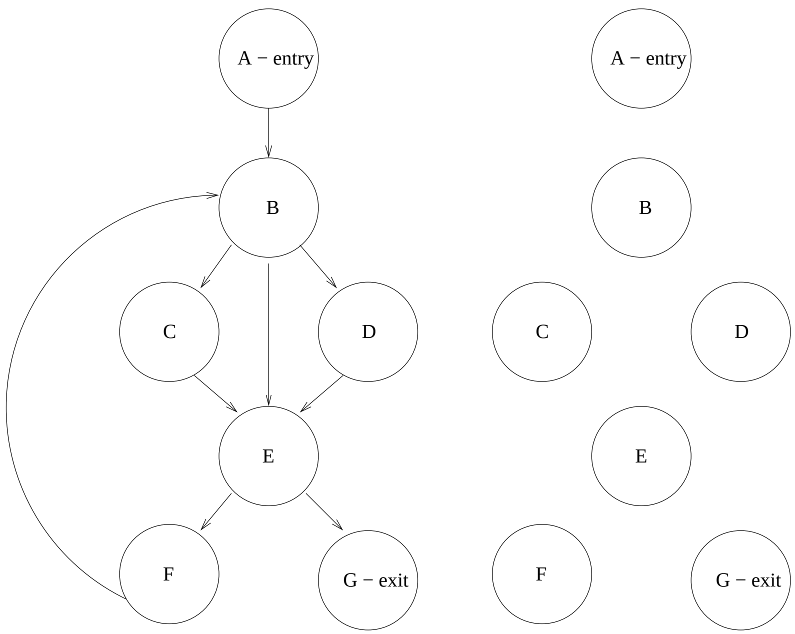
Given a spanning tree, the edges in the *CFG* may be partitioned as follows:

1. **Spanning tree edges** are in the *CFG* and the ST
2. **Advancing edges** (v,w) in *CFGare not spanning tree edges, but* w *is a descendant of* v *in ST.*
3. **Back edges** (v,w) *in CFGsuch that* $v = w$ *or* w *is an ancestor of* v *in ST.*
4. **Cross edges** (v,w) *in CFGsuch that* w *in neither an ancestor nor a descendant of* v *in the spanning tree.*

Spanning Tree Algorithm

```
procedure Span(v)
    for  w in Succ(v) do
        if not InTree(w) then
            add w,v →* w to ST
            InTree(w) = true
            Span(w)
    endfor
end Span

main ()
    for  v ∈ V do InTree = false
    InTree(Root) = true
    Span(Root)
```

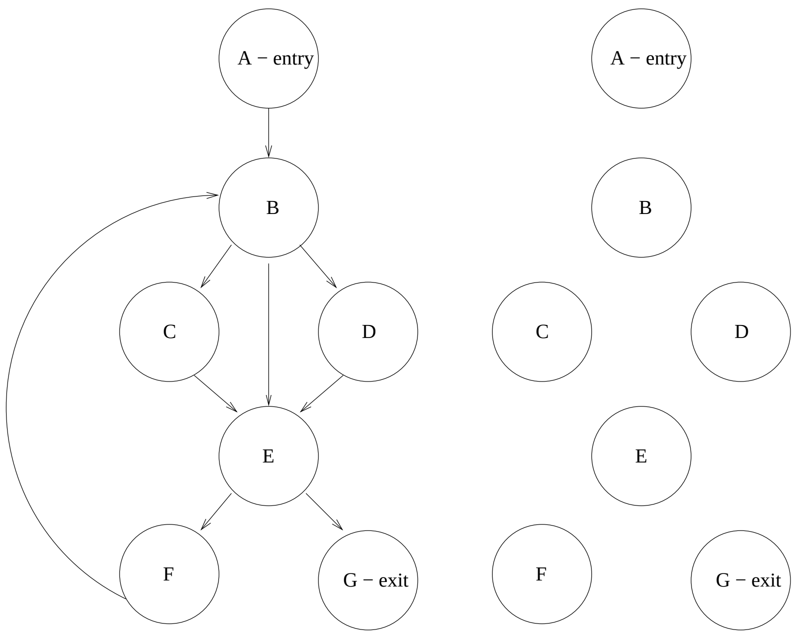
Spanning Tree Example



Spanning Edge Identification

```
procedure DFST(v)
    num(v) = vnum++
    InStack(v) = true
    for  w ∈ Succ(v) do
        if not InTree(w) then

            add w, v →* w to ST
            InTree(w) = true
            DFST(w)
        else if


        else if


        else

    endfor
    InStack(v) = false
end DFST

vnum = 0
DFST(root)
```

Spanning Edge Identification - Example



Cycles - Strongly Connected Regions (SCR)

$\forall s_1, s_2 \in S$, if S is a cycle, then $s_1 \xrightarrow{*} s_2$ and $s_2 \xrightarrow{*} s_1$

Compute maximal SCR on a direct graph.

Robert Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM J. Computing*, 1:2, pp. 146-160, June 1972.

- uses a depth-first spanning tree
  left-to-right pre-order number in *Number*
- tracks the lowest numbered v to which each vertex has a path in *Lowlink*
- determines a number for SCR to which v belongs.

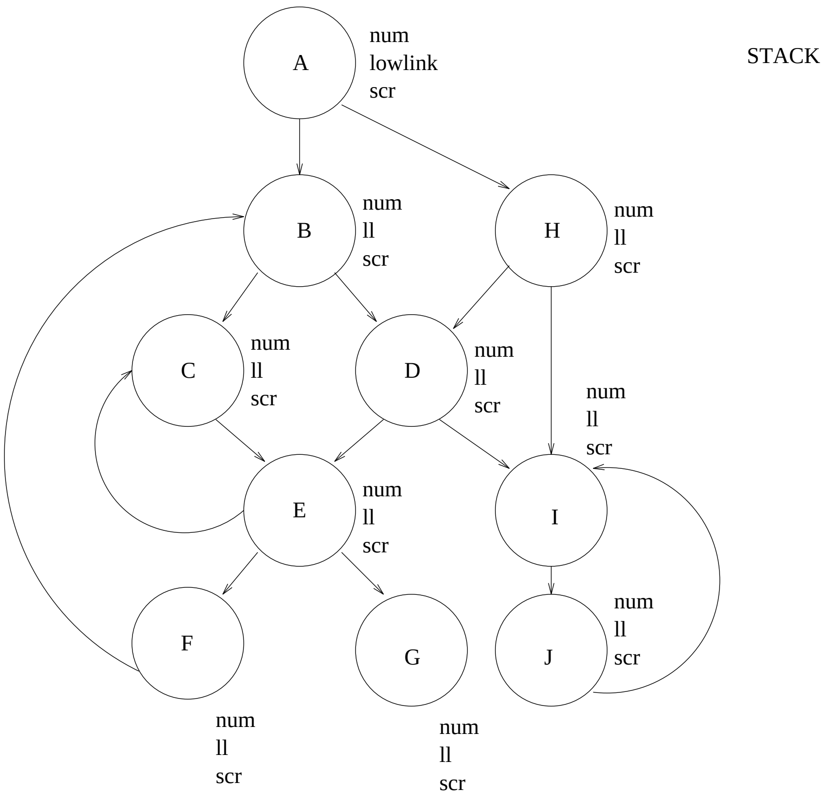
Tarjan's maximal SCR algorithm

i = 0
Lowlink($*$) = 0
Number($*$) = 0
SCRnum = 0
InStack($*$) = **false**
Stack = empty
for $v \in V$ **do**
    if *Number*(v) == 0 **then**
        Tarjan(v)
endfor

Tarjan's maximal SCR algorithm (continued)

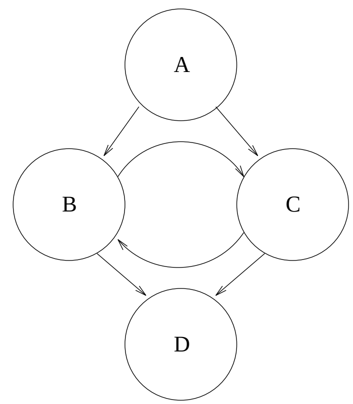
procedure Tarjan(v)
    Number(v) = *Lowlink*(v) = ++i
    InStack(v) = **true**
    push v on *Stack*
    for w in SUCC(v) **do**
        if *Number*(w) = 0 **then**
            Tarjan(w)
            Lowlink(v) = min (*Lowlink*(v), *Lowlink*(w))
        else if *InStack*(w) **then**
            Lowlink(v) = min (*Lowlink*(v), *Lowlink*(w))
    endfor
    if *Lowlink*(v) = *Number*(v) **then**
        SCRnum++
        repeat
            w = pop(*Stack*)
            InStack(w) = **false**
            SCR(w) = SCRnum;
        until w == v
end Tarjan

Tarjan's maximal SCR algorithm - Example

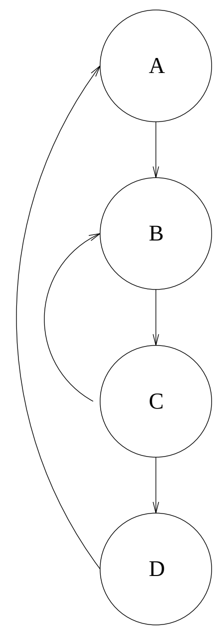


Identifying Loops and Loop Headers

- DFST does not find a unique header in irreducible graphs
- SCR do not differentiate inner loops



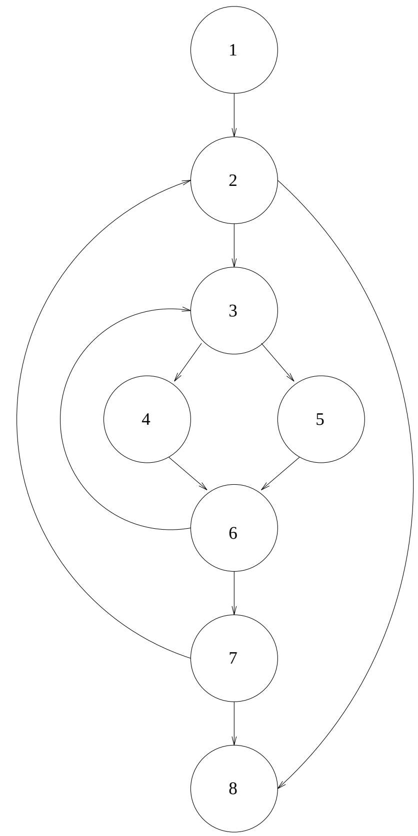
IRREDUCIBLE



Natural Loop

- Single entry, *header* dominates all vertices in loop. **dominates:** v dom w *iff* $v \xrightarrow{*} w$, and $\nexists$ P such that $P = entry \rightarrow x \xrightarrow{*} w$ where v not on P.
- There is at least one path from the header to itself.
- All vertices and edges on a path from the header to any back edges to the header are in the loop.
- Two natural loops are either entirely disjoint, or one is a proper subset of the other.

Natural Loop Example

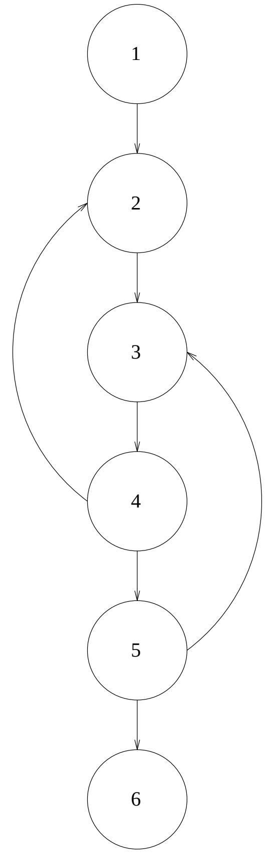


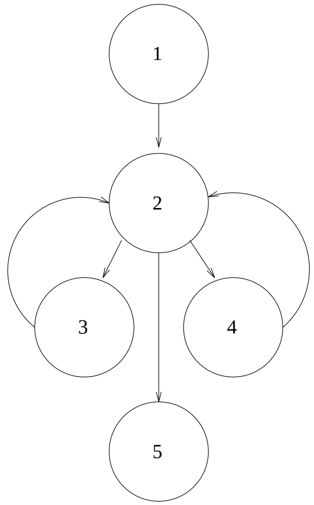
Natural Loop Algorithm

Given a back edge, $(t \rightarrow h)$
    addvertex (h)
    addedge $(t \rightarrow h)$
    insert (t)

procedure insert (v)
    if v not in loop then
        addvertex(v)
        for $p \in$ PRED(v) **do**
            addedge $(p \rightarrow v)$
            insert (p)
        endfor
end insert

Improperly Nested Loops





One loop or two?

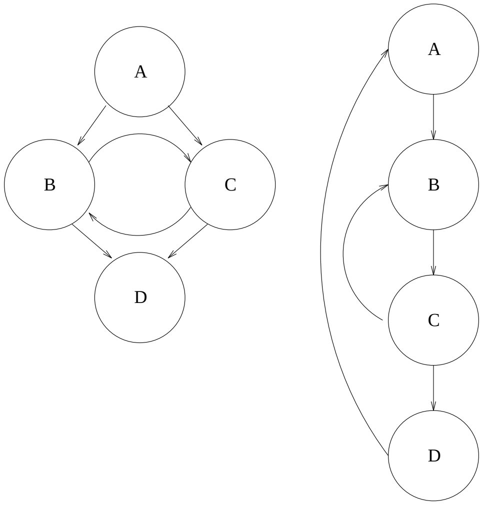
Inner loop?

Outer loop?

Reducible Control Flow Graphs

Intuitively, if all loops are single entry, the *CFG* is reducible. More formally,

- Given a spanning tree, for every back edge in the *CFG*, the head *dominates* the tail (i.e., you cannot execute the tail without executing the head first).



Next Time

Dataflow Analysis: how do values flow around the control graph to variables?

Read: T.J. Marlowe and B.G. Ryder, Properties of Data Flow Frameworks, pp. 121-163, ACTA Informatica, 28, 1990.