Control Flow Analysis [ASU1e 10.4] [ALSU2e 9.6] [Muchnick 7]

- necessary to enable global optimizations beyond basic blocks
- basis for data-flow analysis
- reconstruction of if-then-else, **loops**
  from MIR, from unstructured source code or from target code
  → loops: candidates for loop transformations, software pipelining
  → if-then-else: candidates for predication
- identify basic blocks of a routine
- construct its flow graph / basic block graph

1. Dominator-based analysis (iterative)
2. Interval analysis (recursive)
3. Structural analysis (recursive)

Control Flow Analysis – Running Example

```
// Fibonacci - Iterative alg.

unsigned int fib (
  unsigned int m )
{
  unsigned int f0 = 0,
               f1 = 1,
               f2, i;
  if (m <= 1)
     return m;
  else {
     for (i=2; i<=m; i++) {
        f2 = f0 + f1;
        f0 = f1;
        f1 = f2;
     }
     return f2;
  }
}
```

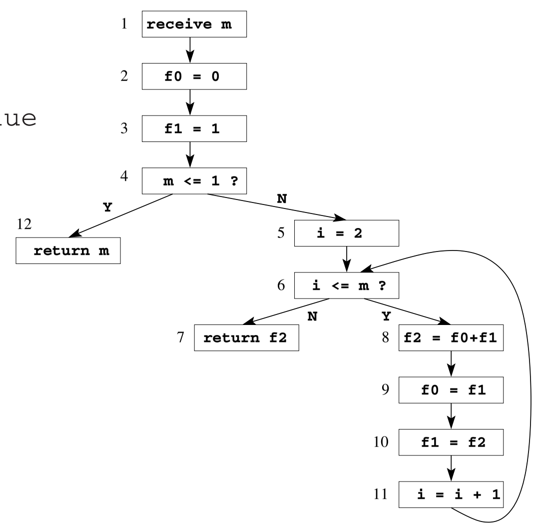
```
// MIR, flattened

 1      receive m
          // read arg value
 2      f0 = 0
 3      f1 = 1
 4      if m<=1 goto L3
 5      i = 2
 6  L1: if i<=m goto L2
 7      return f2
 8  L2: f2 = f0 + f1
 9      f0 = f1
10      f1 = f2
11      i = i + 1
12      goto L1
13  L3: return f2
```


Example (cont.): Control Flow Graph

```
// MIR

 1      receive m
          // read arg value
 2      f0 = 0
 3      f1 = 1
 4      if m<=1 goto L3
 5      i = 2
 6  L1: if i<=m goto L2
 7      return f2
 8  L2: f2 = f0 + f1
 9      f0 = f1
10      f1 = f2
11      i = i + 1
12      goto L1
13  L3: return f2
```



Detecting basic blocks

basic block (BB)
= max. sequence of consecutive statements (IR or target level)
that can be entered by program control only via the first one
and left only via the last one.

first instruction ("leader") of a BB: either

+ entry point of a procedure, or
+ branch target, or
+ instruction immediately following a branch or return

! call instructions need not delimit the basic block
(ok for most cases, but not for e.g. instruction scheduling)

! exception-based control transfer not considered here

Basic-block graph

Terminology: in [Muchnick'97] called control-flow graph CFG,
whereas "our" CFG (statement level) is there called a "flowchart"

rooted, directed graph $G = (N, E)$

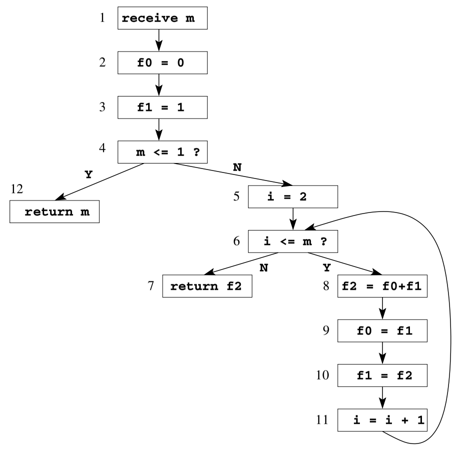
nodes = basic blocks + $\boxed{\text{entry}}$ + $\boxed{\text{exit}}$

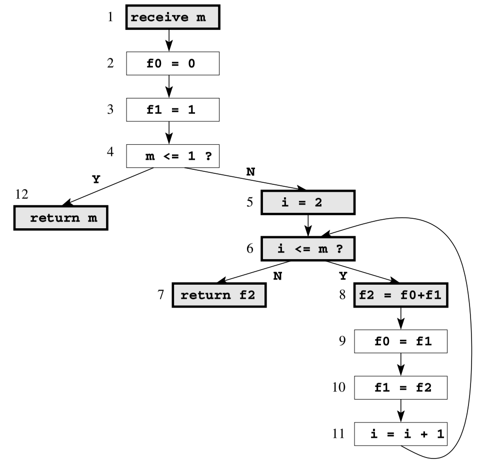
edges = control flow edges from CFG/flowchart
(there connecting BB exits to the leaders of their successor BBs)
+ $\boxed{\text{enter}}$ $\rightarrow$ initial basic block
+ final basic blocks (no succ.) $\rightarrow$ $\boxed{\text{exit}}$

successor BB's of a BB b: $Succ(b) = \{n \in N : (b,n) \in E\}$

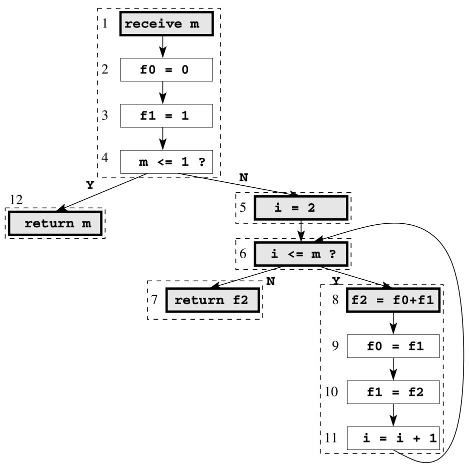
predecessor BB's of a BB b: $Pred(b) = \{n \in N : (n,b) \in E\}$

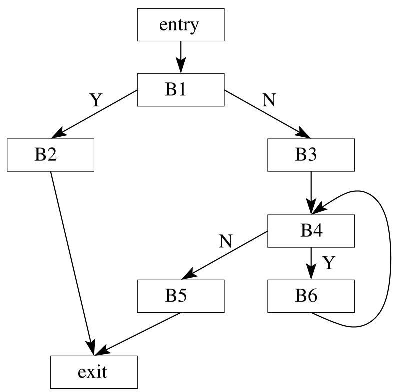
Example (cont.): CFG $\rightarrow$ Basic Block Leaders





Example (cont.): CFG + Basic Blocks $\rightarrow$ Basic Block Graph





Extended basic blocks, regions

Extended basic block (EBB)
= max. sequence of instructions beginning with a leader
that contains no join nodes other than (maybe) its first node

$\rightarrow$ single entry, multiple exits, tree-like internal control flow

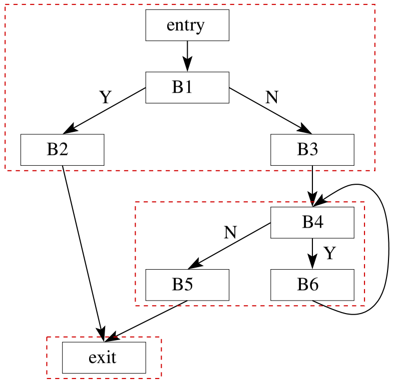
$\rightarrow$ EBB also known as treegion

EBB's are useful for some optimizations e.g. instruction scheduling

Algorithm for computing the EBB's of a CFG: see e.g. [Muchnick 7.1]

Region
= strongly connected subgraph (SCC) of the CFG with a single entry

Example (cont.): Extended Basic Blocks



Finding Loops

Programs spend most of the execution time in loops.

→ Optimizations that exploit the loop structure are important

- loop unrolling, loop parallelization, software pipelining, ...

Loops may be expressed in programs by different constructs (while, for, goto, ..., compiler-converted tail-recursion)

→ Find uniform treatment for program loops

Use a general approach based on graph-theoretic properties of CFG.

Graph-theoretic concepts of control-flow analysis (1)

depth-first search (dfs): recursively explore descendants of a node before any of its siblings (as far as not yet visited)

dfs-number: order in which dfs enters nodes

tree edges: edges followed by dfs via recursive calls

dfs-tree: (nodes, tree edges)

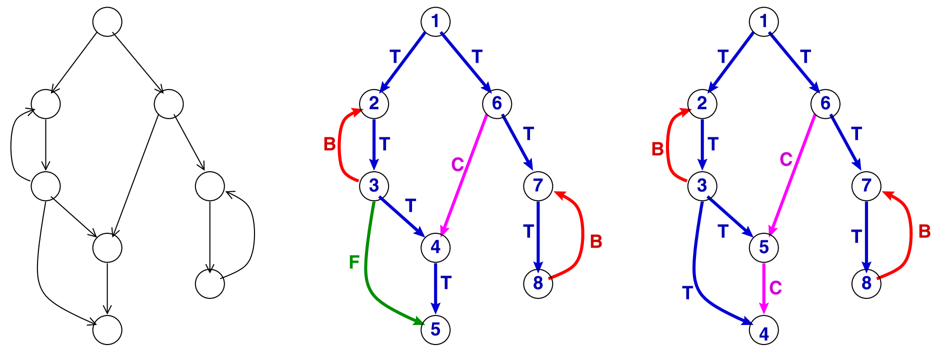
non-tree edges classified as

- forward edges “F”
- back edges “B”
- cross edges “C”

not unique! depends on ordering of descendants

see also DFS-slides on course homepage

Example: DFS-tree, edge classification



Graph-theoretic concepts of control-flow analysis (2)

preorder traversal of a digraph $G = (N, E)$:
at each node $b \in N$ process b before its descendants
(not unique; in dfsnum-order)

postorder traversal
at each node $b \in N$ process b after its descendants

Dominance, immediate dominance, strict dominance

Given: flow graph $G = (N, E)$, nodes $d, i, p, b \in N$

d dominates b (d dom b)
if every possible execution path $\boxed{\text{entry}} \rightarrow^* b$ includes d

dom is reflexive, transitive, antisymmetric $\rightarrow$ partial order on N

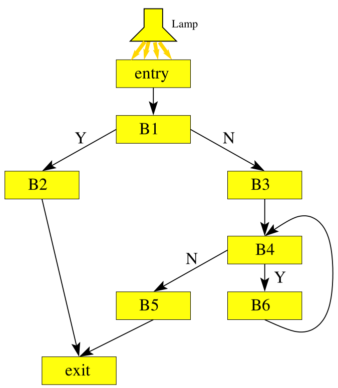
i immediately dominates b (i idom b)
if i dom b and there is no $c \in N$, $i \neq c \neq b$, with i dom c and c dom b

idom(b) is unique for each $b \in N$
$\rightarrow$ (i)dominator tree, rooted at $\boxed{\text{entry}}$

strict dominance d sdom b if d dom b and $d \neq b$

Dominance intuition (1)

Imagine a source of light going into the entry node,
nodes are transparent and edges are optical fibers

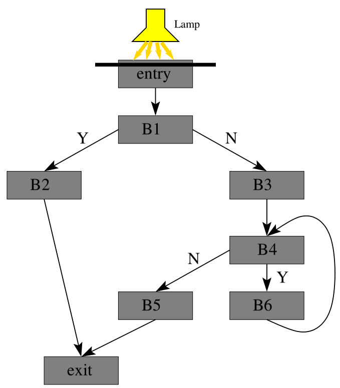


Place an opaque barrier at node $v \rightarrow$ nodes dominated by v get dark

(Adapted from a nice presentation by J. Amaral 2003)

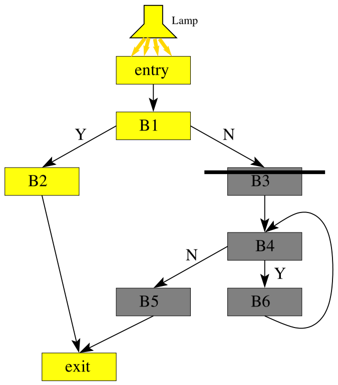
Dominance intuition (2)

The entry node dominates all nodes:



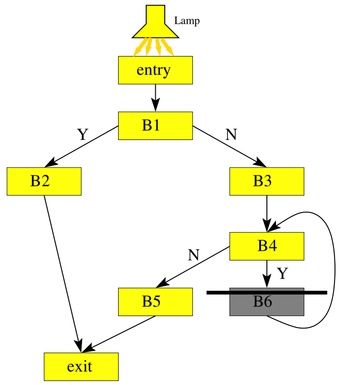
Dominance intuition (3)

Node $B3$ dominates $B3$, $B4$, $B5$, $B6$:



Dominance intuition (4)

Node $B6$ only dominates itself:



Postdominance

p postdominates b (p pdom b)

if every possible execution path $b \rightarrow^* \boxed{\text{exit}}$ includes p

p pdom b $\iff$ b dom p in the reversed flow graph

Computing the dominators of a node

a dom b

iff

(1) $a = b$,

or

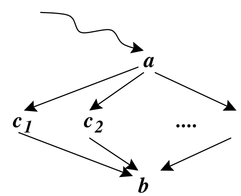
(2) $\exists$ unique immediate predecessor of b, namely a, i.e. $Pred(b) = \{a\}$,

or

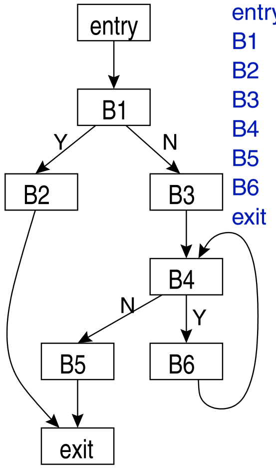
(3) for all $c \in Pred(b)$ $c \neq a$ and a dom c.

Algorithm 1:

change $\leftarrow$ true;
Domin$(r) \leftarrow \{r\}$; Domin$(n) \leftarrow N$ $\forall n \in N - \{r\}$
while (change)
  change $\leftarrow$ false
  for all $n \in N - \{r\}$ // in dfsnum order
    $D \leftarrow \{n\} \cup \bigcap_{p \in Pred(n)}$ Domin(p)
    if $D \neq$ Domin(n)
      Domin$(n) \leftarrow D$; change $\leftarrow$ true
return Domin



Example: Computing dominators



node i	Domin(i), init.	Domin(i), iter. 1	iter. 2 ...
entry	{entry} change=true	{entry}	...
B1	{entry, B1, B2, B3, B4, B5, B6, exit}	{entry, B1} change=true	...
B2	{entry, B1, B2, B3, B4, B5, B6, exit}	{entry, B1, B2}	...
B3	{entry, B1, B2, B3, B4, B5, B6, exit}	{entry, B1, B3}	...
B4	{entry, B1, B2, B3, B4, B5, B6, exit}	{entry, B1, B3, B4}	...
B5	{entry, B1, B2, B3, B4, B5, B6, exit}	{entry, B1, B3, B4,B5}	...
B6	{entry, B1, B2, B3, B4, B5, B6, exit}	{entry, B1, B3, B4,B6}	...
exit	{entry, B1, B2, B3, B4, B5, B6, exit}	{entry, B1, exit}	...

Extension for computing immediate dominators

... compute Domin ...

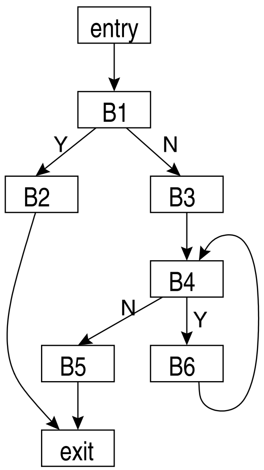
for each $n \in N$
  Tmp(n) $\leftarrow$ Domin(n) $-$ $\{n\}$

for each $n \in N - \{r\}$ // in dfsnum order
  // if a s in Tmp(n) has a dominator $t \neq s$, remove t from Tmp(n)
  for each $s \in$ Tmp(n)
    for each $t \in$ Tmp(n) $- \{s\}$
      if $t \in$ Tmp(s) **then**
        Tmp(n) $\leftarrow$ Tmp(n) $- \{t\}$

for each $n \in N - \{r\}$ // in dfsnum order
  idom(n) $\leftarrow$ the $b \in$Tmp(n)

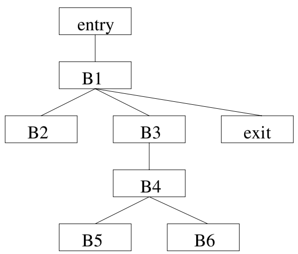
Total time: $O(n^2 e)$ if sets are represented by bitvectors

Example: Computing immediate dominators



node i	init. Tmp(i)=Domin(i)–{i}	Tmp(i), iter. 1
entry	{entry}	{entry}
B1	{entry}	{entry}
B2	{entry, B1}	{B1}
B3	{entry, B1}	{B1}
B4	{entry, B1, B3}	{B3}
B5	{entry, B1, B3, B4}	{B4}
B6	{entry, B1, B3, B4}	{B4}
exit	{entry, B1}	{B1}

Dominator tree:



Computing dominators

Algorithm 2 [Lengauer/Tarjan'79]

based on depth first search and path compression

time $O(e \log n)$ or $O(e \cdot \alpha(e,n))$

(see e.g. [Muchnick pp. 185–190])

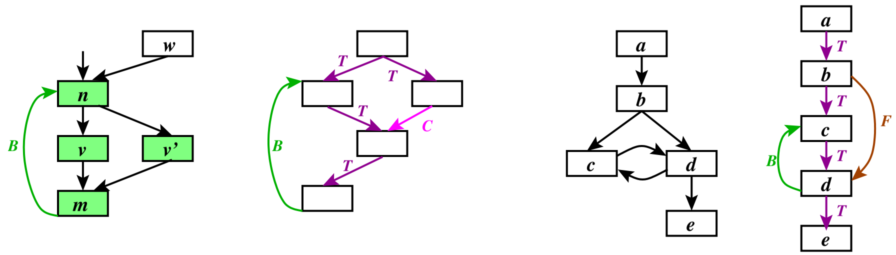
Loops and Strongly Connected Components

We call a (backward, B) edge (m,n) a loop back edge if n dom m.

Remark: Not every B edge is a loop back edge!

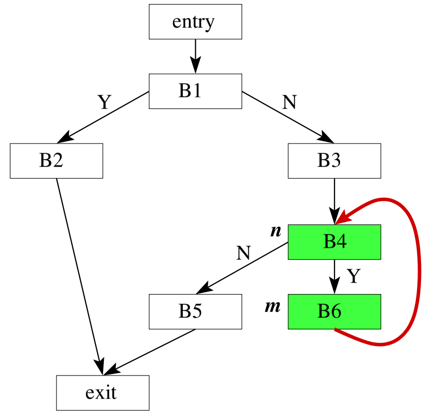
Natural loop of a loop back edge (m,n)
= subgraph of n and all nodes v
from which m can be reached without passing through n

n is the loop header.



c does not dominate $d \Rightarrow$ not a natural loop (2 entry points)

Example (cont.): Natural Loop

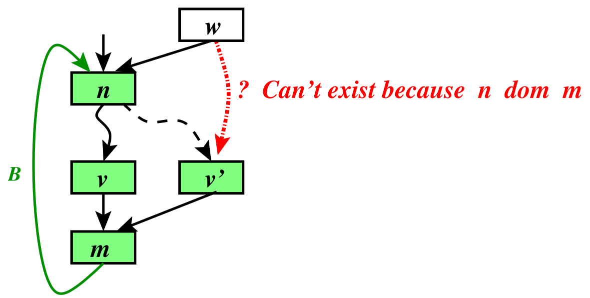


Identifying the natural loop of a loop back edge

Algorithm: Compute the loop node set for a given loop back edge (m,n)

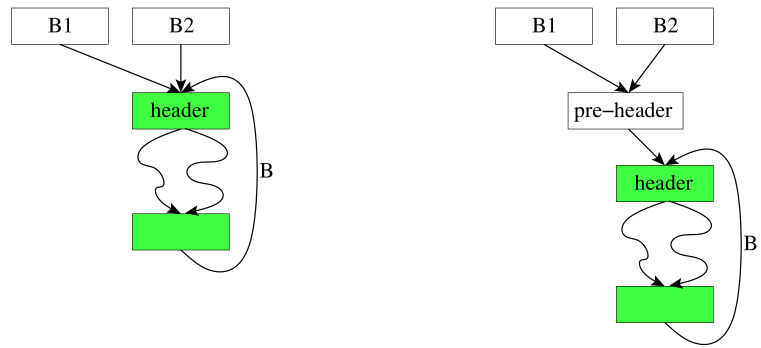
- Start by marking m and n as loop nodes.
- Backwards from m, (df)search predecessors v,

  stopping recursive backward search at already found loop nodes.



Loop header, preheader

For technical reasons, add a pre-header (initially empty)
if the header has more than 2 predecessors:



→ Easier to place new instructions immediately *before* the loop

Properties of Natural Loops

- Two natural loops with different headers* are either disjoint or one is nested in the other.
- Each natural loop is a SCC.

Background:

Strongly connected component (SCC)
= subgraph $S = (N_S, E_S)$, $N_S \subseteq N$, $E_S \subseteq E$,
where every node in V_S is reachable in S
from every other node in V_S via edges in E_S.

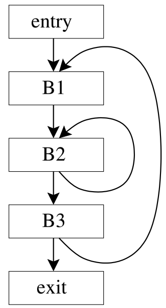
SCC's can be computed with Tarjan's algorithm (extension of dfs)
in time $O(|V| + |E|)$ [Tarjan'72]

* Several loops sharing a common header node is a pathological special case that must be treated ad hoc. See e.g. Muchnick 7.4 for more details.

Properties of Natural Loops (cont.)

A SCC $S \subseteq V$ is *maximal* if every SCC containing S is just S itself.

Example:

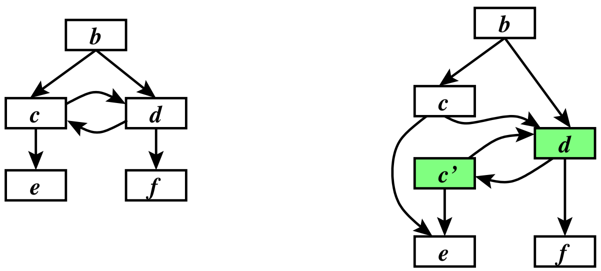


$S_1 = \{B1, B2, B3\}$ is a maximal SCC.
$S_2 = \{B2\}$ is a SCC but not a maximal SCC.

Reducibility of flow graphs

A flow-graph is reducible if all B edges in any DFS tree are loop back edges.

Intuitively: ... if there are no jumps into the middles of loops (e.g., goto's).



Reducible flow graphs are well-structured (loops properly nested).

Irreducible flow graphs are rare
and can be made reducible by replicating nodes.

Interval analysis

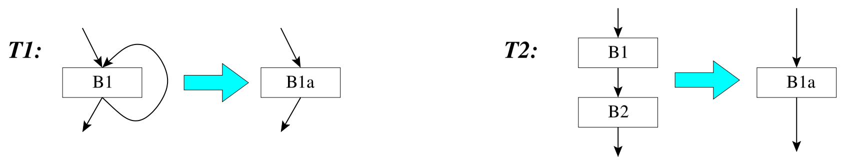
- Divide flowgraph into regions (e.g., loops in CFA)
- Repeatedly collapse a region to an abstract node

  → abstract flowgraph

  → nested regions (control tree)

  Hierarchical folding structure allows for faster / simpler data flow analysis

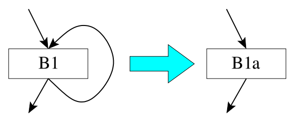
Simplest variant: $T1$-$T2$ Analysis [Ullman'73]



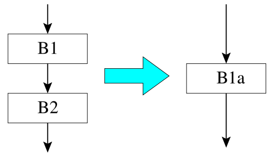
- Try to fold entire flow graph into a single node
- Works only for very restricted flow graphs

T1-T2 Analysis — Example

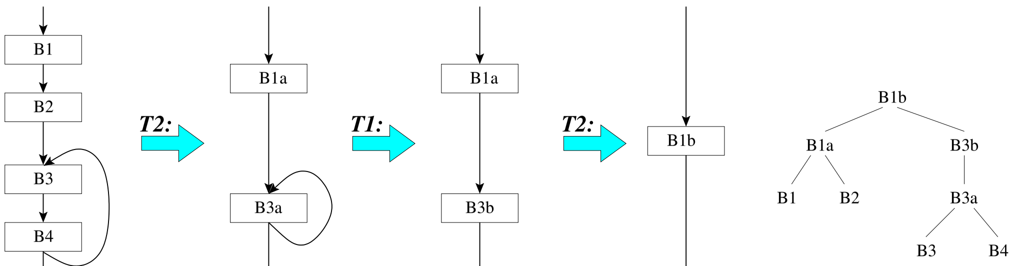
T1:



T2:



Example:



$T1$-$T2$ control tree

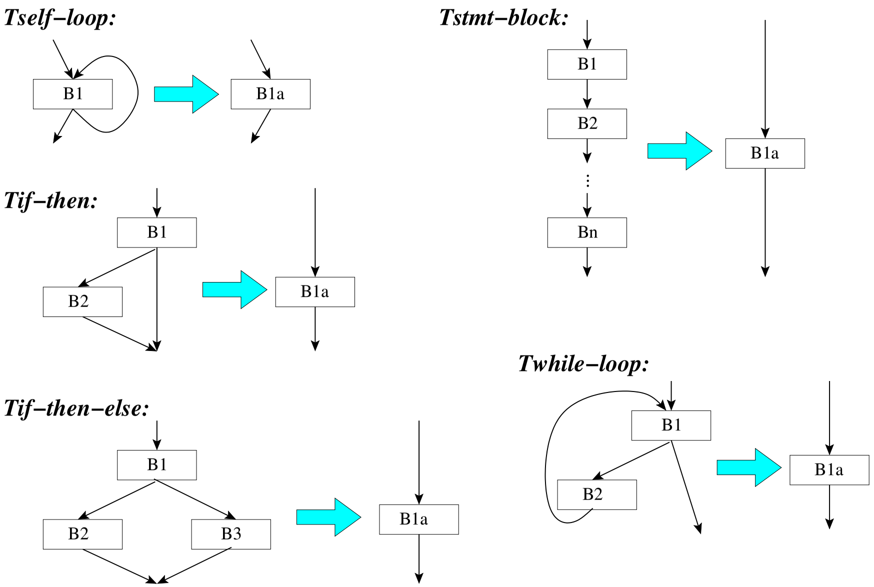
Structural Analysis

is a special case of interval analysis:

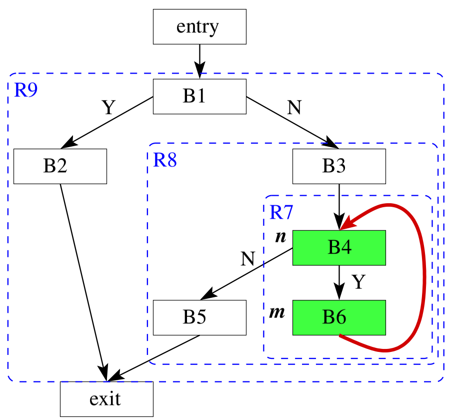
- CFG folding follows the hierarchical structure of the program
  $\rightarrow$ folding transformations for loops, if-then-else, switch, etc.
- Every region has 1 entry point
- Works only for well-structured programs
  - Extensions to handle arbitrary flowgraphs
    (define a new region / transf. for otherwise irreducible constructs)

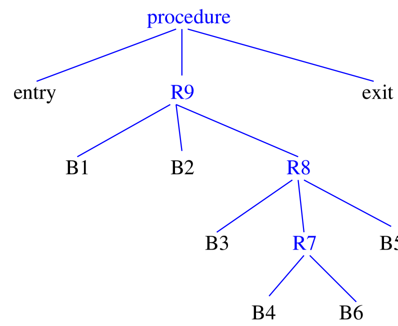
+ Equations etc. for dataflow analysis can be pre-formulated for each construct $\rightarrow$ faster

Structural analysis — Some regions and transformations



Example (cont.): Structural analysis





Region Hierarchy Tree

Remark: If only loop-based regions are of interest, the hierarchy flattens accordingly ($R8$ and $R9$ merged with top level).

Computing a bottom-up order of regions of a reducible flow graph

Input: A reducible flow graph G

Output: A bottom-up ordered list R of loop-based regions of G

1. $R \leftarrow \{B1, B2, ...\}$ = all leaf regions, i.e., all single blocks in G, in any order

2. **repeat**

    Choose a natural loop L such that,

        if there are any natural loops L' contained within L,

        then the (body and loop) regions for these L' were already added to R.

    R.add(the region consisting of the body of L)

        // body of L = L without the back edges to the header of L

    R.add (the loop region for L)

  until all natural loops have been considered

3. If the entire flow graph is not itself a natural loop,

   R.add(the region consisting of the entire flow graph).

Summary: Control-Flow Analysis

- Basic blocks, extended basic blocks
- Loop detection
- Dominator-based CFA
- Interval-based CFA
- Structural analysis