Code analysis *and* transformation

CFA

Simone Campanoni
simone.campanoni@northwestern.edu

Outline

- CFA and a first example: dominators

- Another example of CFA: dominance frontier

- Example of CFA and CFT: basic block merging and splitting

Control Flow Analysis

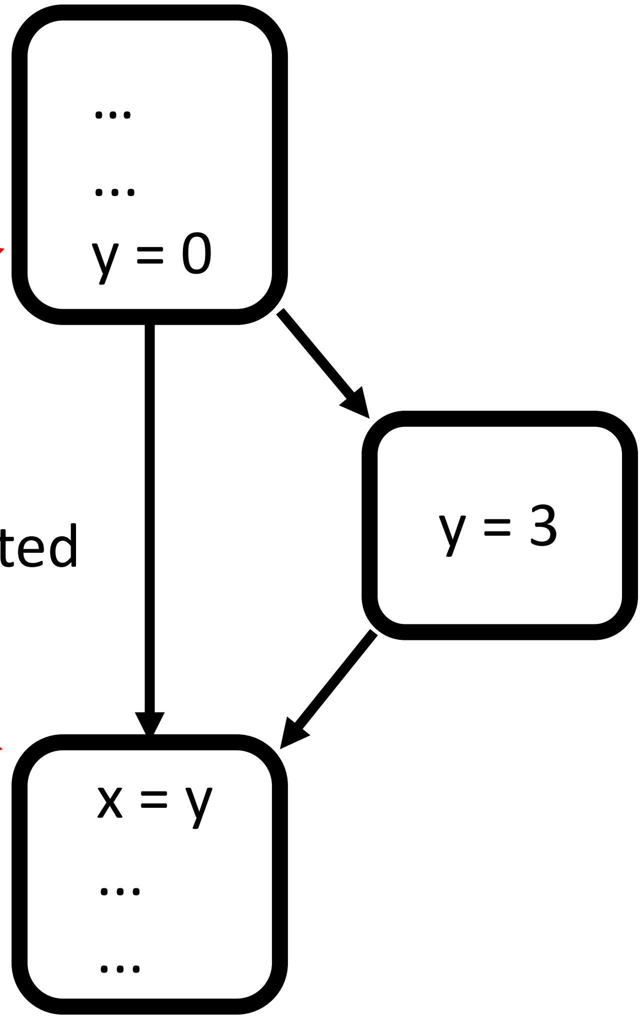
- Storing order $\neq$ executing order
- Control Flow Analyses are designed
  to understand the possible execution paths (control flows)
  while ignoring data values and operations/operators

- We need to identify all possible control flows
  between **instructions**
- We need to identify all possible control flows
  between **basic blocks**

- Let's look at an example of CFA

Control Flow Graph

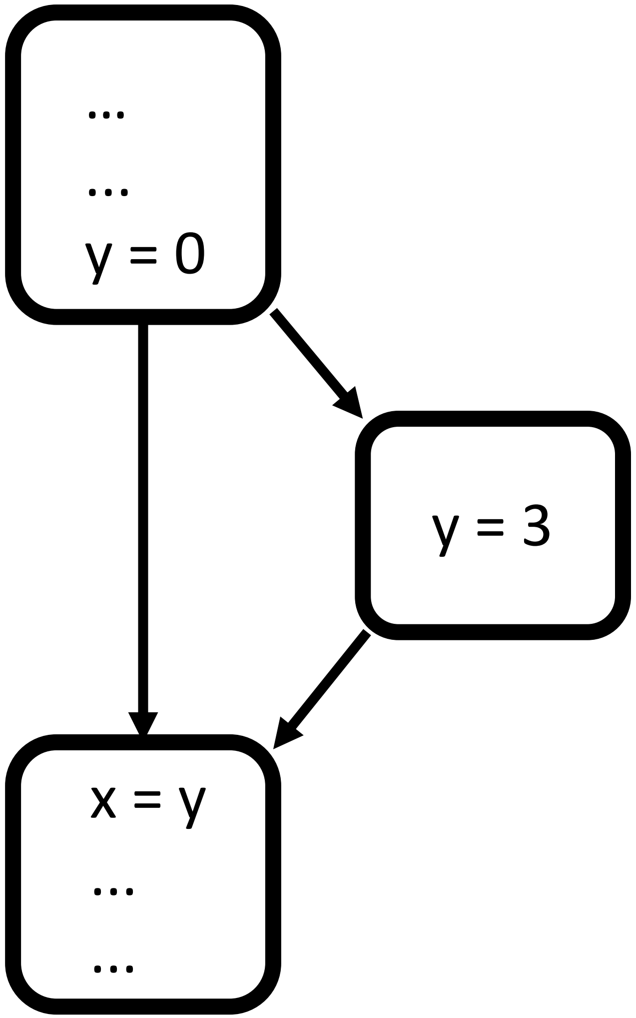
After executing this basic block
This other basic block might be executed



Sometimes “may” isn’t enough

How can I know that a given basic block will be executed no matter what?

This is what our first CFA computes.

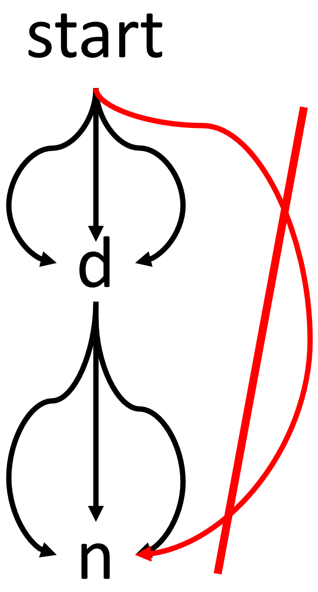


Dominators

Definition: Node *d* dominates node *n* in a CFG (*d dom n*)
iff every control flow from the start node to *n* goes through *d*.
Every node dominates itself.

What is the relation between
instructions within a basic block?

1



What is the relation between
instructions in different basic blocks?

It depends on the CFG

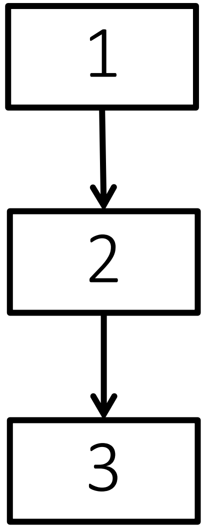
In other words, dominators depend on the control flows

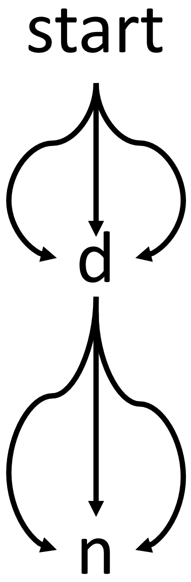
Dominators

Definition: Node *d* dominates node *n* in a CFG (*d dom n*)
iff every control flow from the start node to *n* goes through *d.*
Every node dominates itself.

What are the dominators of basic blocks 1 and 2?

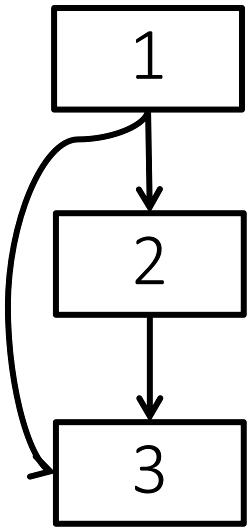
What are the dominators of basic blocks 1, 2, and 3?

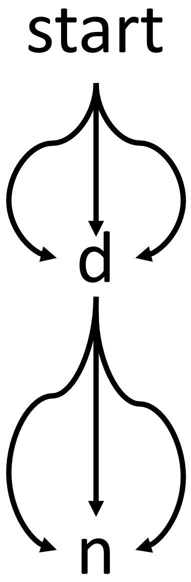




Dominators

Definition: Node *d* dominates node *n* in a CFG (*d dom n*)
iff every control flow from the start node to *n* goes through *d.*
Every node dominates itself.



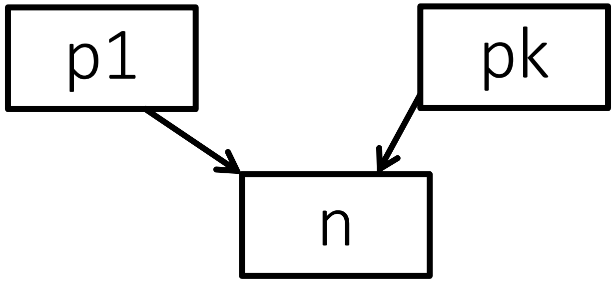


What are now the dominators of basic blocks 1, 2, and 3?

Now that we know what we want to obtain
(the dominance binary relation between basic blocks),

let us define an algorithm (a CFA) that computes it

A CFA to find dominators



Consider a block n with k predecessors p_1, …, p_k

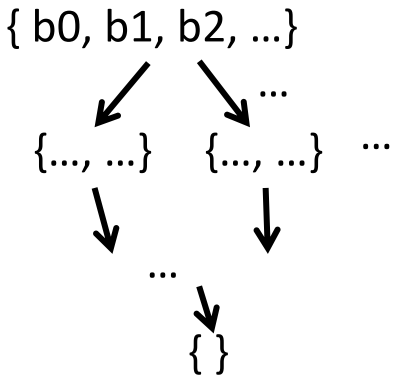
Observation 1: if d dominates each p_i ($1 \le i \le k$), then d dominates n

Observation 2: if d dominates n, then it must dominate all p_i

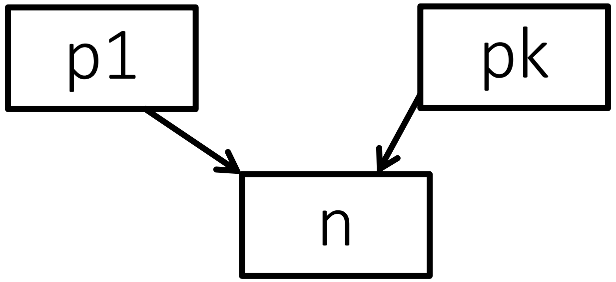
$$D[n] = \{n\} \cup \left(\bigcap_{p \in \text{predecessors}(n)} D[p]\right)$$

To compute it:

- By iteration
- Initialize each D[n] to ?



A CFA to find dominators



Consider a block n with k predecessors $p_1, \ldots, p_k$

Observation 1: if d dominates each p_i ($1 <= i <= k$), then d dominates n

Observation 2: if d dominates n, then it must dominate all p_i

$$D[n] = \{n\} \cup \left(\bigcap_{p \in predecessors(n)} D[p]\right)$$

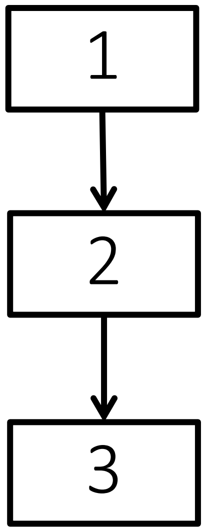
To compute it:

- By iteration
- Initialize each D[n] to include every one

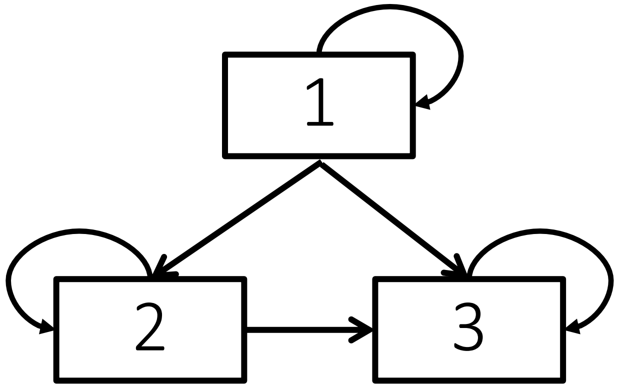
This is our first CFA

Notice: this CFA does not depend on values and/or operations/operators

Dominance



CFG



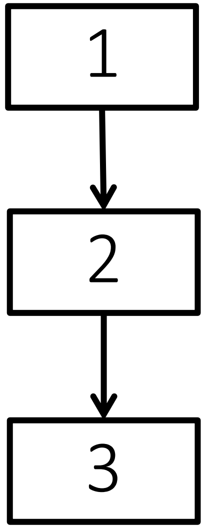
Dominators

We can now introduce new concepts based on the dominator relation

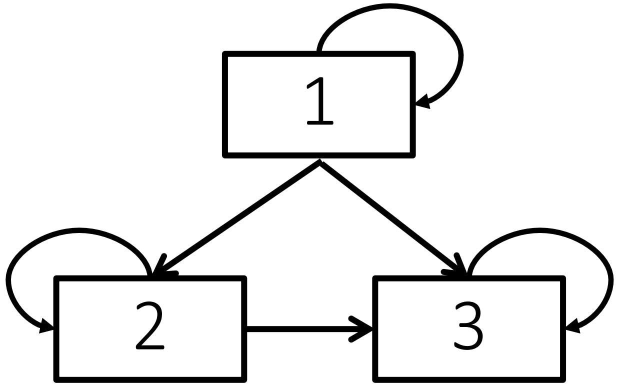
Strict dominance

Definition:
a node *d* strictly dominates *n* iff

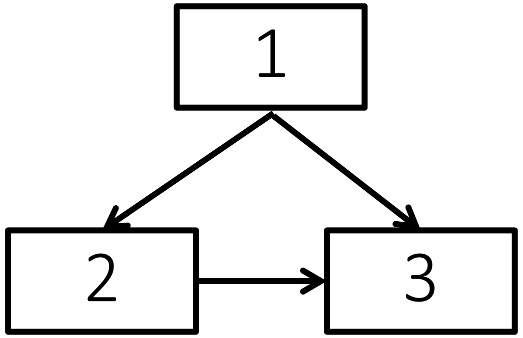
- *d* dominates *n* and
- *d* is not *n*



CFG



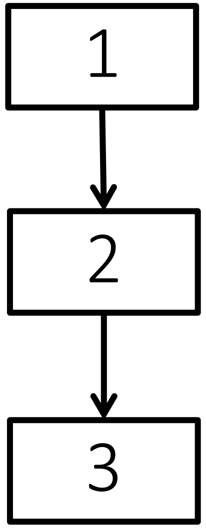
Dominators



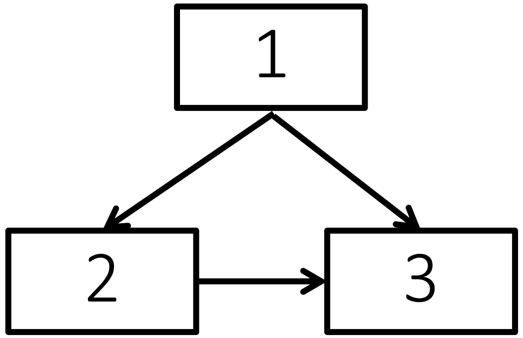
Strict dominators

Immediate dominators

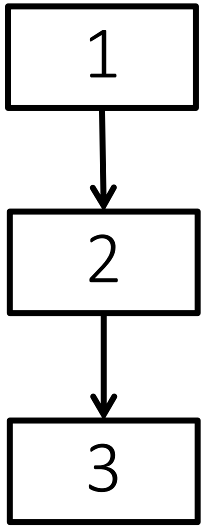
Definition: the immediate dominator of a node *n*
is the unique node that strictly dominates *n*
but does not strictly dominate another node that strictly dominates *n*



CFG



Strict dominators

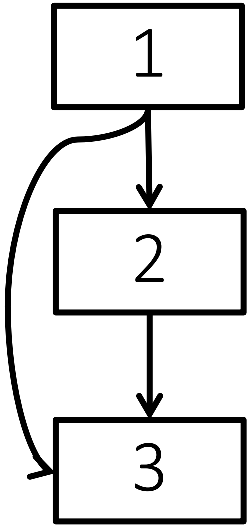


Dominator tree

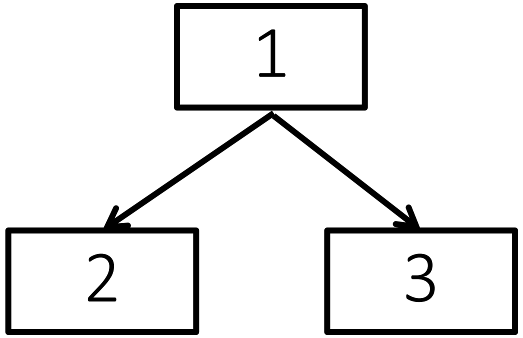
Immediate dominators

Immediate dominators

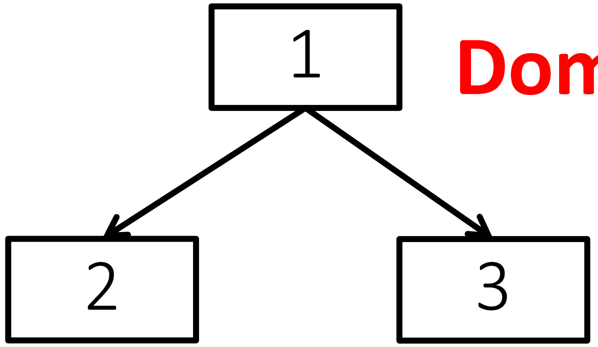
Definition: the immediate dominator of a node *n*
is the unique node that strictly dominates *n*
but does not strictly dominate another node that strictly dominates *n*



CFG



Strict dominators



Dominator tree

Immediate dominators

Dominators in LLVM

```
#include "llvm/IR/Dominators.h"
```

```
bool runOnFunction (Function &F) override {
  errs() << "=== Dominators\n";

  DominatorTree &DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();

  for (auto& bb : F){
    auto inst = bb.begin();
    errs() << *inst << "\n";

    auto instNode = DT.getNode(&bb);

    for (auto child : instNode->getChildren()){
      auto dominatedBB = child->getBlock();
      auto dominatedInst = dominatedBB->begin();
      errs() << " -> " << *dominatedInst << "\n";
    }

  }
  return false;
}
```

```
int main(int argc, char *argv[]) {
  printf("START\n");
  if (argc > 0){
    printf("THEN\n");
  } else {
    printf("ELSE\n");
  }
  printf("END\n");

  if (argc == 0){
    return 0;
  }

  return 1;
}
```

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<DominatorTreeWrapperPass>();
  AU.setPreservesAll();
}
```

Dominators in LLVM

Notice the order

You cannot assume any order

```
=== Dominators
 %3 = alloca i32, align 4
 ->   %10 = call i32 (i8*, ...) @printf
 ->   %14 = call i32 (i8*, ...) @printf
 ->   %12 = call i32 (i8*, ...) @printf
 %10 = call i32 (i8*, ...) @printf(i8*
 %12 = call i32 (i8*, ...) @printf(i8*
 %14 = call i32 (i8*, ...) @printf(i8*
 ->   store i32 0, i32* %3, align 4
 ->   %20 = load i32, i32* %3, align 4
 ->   store i32 1, i32* %3, align 4
 store i32 0, i32* %3, align 4
 store i32 1, i32* %3, align 4
 %20 = load i32, i32* %3, align 4
```

```
#include "llvm/IR/Dominators.h"
```

```
bool runOnFunction (Function &F) override {
  errs() << "=== Dominators\n";

  DominatorTree &DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();

  for (auto& bb : F){
    auto inst = bb.begin();
    errs() << *inst << "\n";

    auto instNode = DT.getNode(&bb);

    for (auto child : instNode->getChildren()){
      auto dominatedBB = child->getBlock();
      auto dominatedInst = dominatedBB->begin();
      errs() << " -> " << *dominatedInst << "\n";
    }

  }
  return false;
}
```

Notice the order

What is going to be the output?

```
define dso_local i32 @main(i32, i8**) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  %6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([7 x i
  %7 = load i32, i32* %4, align 4
  %8 = icmp sgt i32 %7, 0
  br i1 %8, label %9, label %11

9:                                                ; preds = %2
  %10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([6 x
  br label %13

11:                                               ; preds = %2
  %12 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([6 x
  br label %13

13:                                               ; preds = %11, %9
  %14 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x
  %15 = load i32, i32* %4, align 4
  %16 = icmp eq i32 %15, 0
  br i1 %16, label %17, label %18

17:                                               ; preds = %13
  store i32 0, i32* %3, align 4
  br label %19

18:                                               ; preds = %13
  store i32 1, i32* %3, align 4
  br label %19

19:                                               ; preds = %18, %17
  %20 = load i32, i32* %3, align 4
  ret i32 %20
}
```

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<DominatorTreeWrapperPass>();
  AU.setPreservesAll();
}
```

Dominators in LLVM: example 2

```
#include "llvm/IR/Dominators.h"
```

```
bool runOnFunction (Function &F) override {
  errs() << "=== Dominators\n";

  DominatorTree &DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();

  for (auto& bb : F){
    auto inst = bb.begin();
    errs() << *inst << "\n";

    auto instNode = DT.getNode(&bb);

    for (auto child : instNode->getChildren()){
      auto dominatedBB = child->getBlock();
      auto dominatedInst = dominatedBB->begin();
      errs() << " -> " << *dominatedInst << "\n";
    }

  }
  return false;
}
```

```
int main(int argc, char *argv[]) {
  printf("START\n");

  if (argc > 0){
    printf("THEN\n");

    if (argc > 20){
      printf("Inside THEN");

      if (argc > 40){
        printf("Inside the inside of THEN");
      }
    }
  }

  printf("END\n");

  return 1;
}
```

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<DominatorTreeWrapperPass>();
  AU.setPreservesAll();
}
```

Dominators in LLVM: example 2

```
#include "llvm/IR/Dominators.h"
```

```
bool runOnFunction (Function &F) override {
  errs() << "=== Dominators\n";

  DominatorTree &DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();

  for (auto& bb : F){
    auto inst = bb.begin();
    errs() << *inst << "\n";

    auto instNode = DT.getNode(&bb);

    for (auto child : instNode->getChildren()){
      auto dominatedBB = child->getBlock();
      auto dominatedInst = dominatedBB->begin();
      errs() << " -> " << *dominatedInst << "\n";
    }

  }
  return false;
}
```

What is going to be the output?

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<DominatorTreeWrapperPass>();
  AU.setPreservesAll();
}
```

```
=== Dominators
 %3 = alloca i32, align 4
 ->  %10 = call i32 (i8*, ...) @printf(
 ->  %22 = call i32 (i8*, ...) @printf(
 %10 = call i32 (i8*, ...) @printf(i8*
 ->  %14 = call i32 (i8*, ...) @printf(
 ->  br label %21
 %14 = call i32 (i8*, ...) @printf(i8*
 ->  %18 = call i32 (i8*, ...) @printf(
 ->  br label %20
 %18 = call i32 (i8*, ...) @printf(i8*
 br label %20
 br label %21
 %22 = call i32 (i8*, ...) @printf(i8*
```

```
define dso_local i32 @main(i32, i8**) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  %6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([7 x i
  %7 = load i32, i32* %4, align 4
  %8 = icmp sgt i32 %7, 0
  br i1 %8, label %9, label %21

9:                                                ; preds = %2
  %10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([6 x
  %11 = load i32, i32* %4, align 4
  %12 = icmp sgt i32 %11, 20
  br i1 %12, label %13, label %20

13:                                               ; preds = %9
  %14 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([12 x
  %15 = load i32, i32* %4, align 4
  %16 = icmp sgt i32 %15, 40
  br i1 %16, label %17, label %19

17:                                               ; preds = %13
  %18 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([26 x
  br label %19

19:                                               ; preds = %17, %13
  br label %20

20:                                               ; preds = %19, %9
  br label %21

21:                                               ; preds = %20, %2
  %22 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x
  ret i32 1
}
```

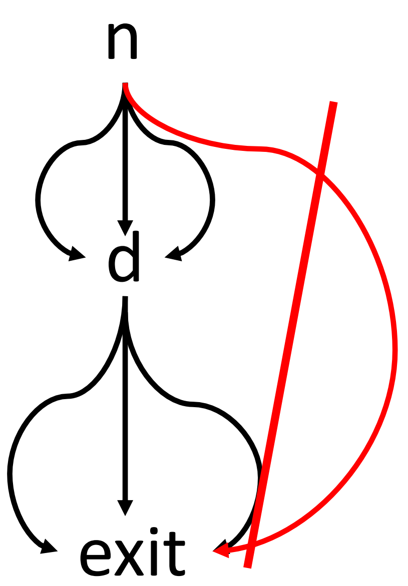
LLVM-specific notes for dominators

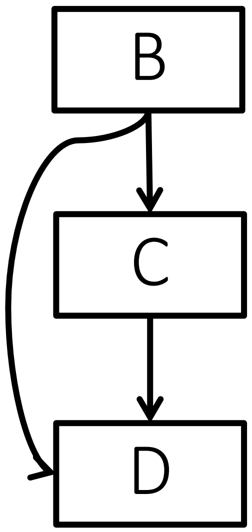
- bool DominatorTree::dominates (...)
  - bool dominates (Instruction *i, Instruction *j)
    Return true if the basic block that includes i is an immediate dominator of the basic block that includes j
  - bool dominates (Instruction *i, BasicBlock *b)
    Return true if the basic block that includes i is an immediate dominator of b
- If the first argument
  is not reachable from the entry point of the function, return false
- If the second argument (either instruction or basic block)
  is not reachable from the entry point of the function, return true

Post-dominators

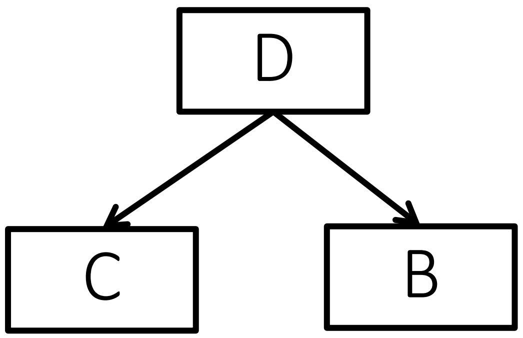
Assumption: Single exit node in CFG

Definition: Node *d* post-dominates node *n* in a graph iff every path from *n* to the exit node goes through *d*





CFG



Immediate post-dominator tree

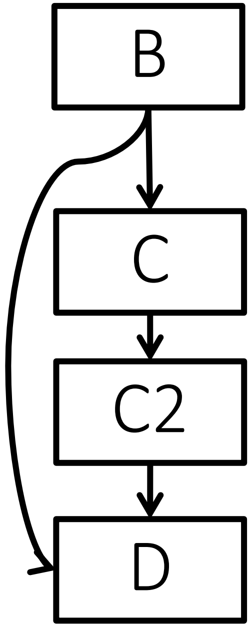
```
B: if (par1 > 5)
C:     varX = par1 + 1
D: print(varX)
```

How to compute post-dominators?

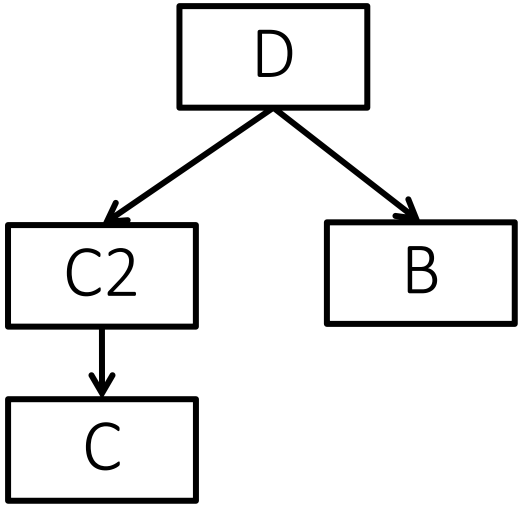
Post-dominators

Assumption: Single exit node in CFG

Definition: Node *d* post-dominates node *n* in a graph iff every path from *n* to the exit node goes through *d*



CFG



Immediate post-dominator tree

```
B: if (par1 > 5)
C:     varX = par1 + 1
C2:   …
D: print(varX)
```

Post dominators in LLVM

```
#include "llvm/Analysis/PostDominators.h"
```

```
bool runOnFunction (Function &F) override {
  errs() << "=== Post dominators\n";

  PostDominatorTree& PDT = getAnalysis<PostDominatorTreeWrapperPass>().getPostDomTree();

  for (auto& bb : F){
    auto inst = bb.begin();
    errs() << *inst << "\n";

    auto instNode = PDT.getNode(&bb);

    for (auto child : instNode->getChildren()){
      auto dominatedBB = child->getBlock();
      auto dominatedInst = dominatedBB->begin();
      errs() << " -> " << *dominatedInst << "\n";
    }

  }
  return false;
}
```

```
int main(int argc, char *argv[]) {
  printf("START\n");
  if (argc > 0){
    printf("THEN\n");
  } else {
    printf("ELSE\n");
  }
  printf("END\n");

  if (argc == 0){
    return 0;
  }

  return 1;
}
```

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<PostDominatorTreeWrapperPass>();
  AU.setPreservesAll();
}
```

Post dominators in LLVM

```
#include "llvm/Analysis/PostDominators.h"
```

```
bool runOnFunction (Function &F) override {
  errs() << "=== Post dominators\n";

  PostDominatorTree& PDT = getAnalysis<PostDominatorTreeWrapperPass>().getPostDomTree();

  for (auto& bb : F){
    auto inst = bb.begin();
    errs() << *inst << "\n";

    auto instNode = PDT.getNode(&bb);

    for (auto child : instNode->getChildren()){
      auto dominatedBB = child->getBlock();
      auto dominatedInst = dominatedBB->begin();
      errs() << " -> " << *dominatedInst << "\n";
    }

  }
  return false;
}
```

What is going to be the output?

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<PostDominatorTreeWrapperPass>();
  AU.setPreservesAll();
}
```

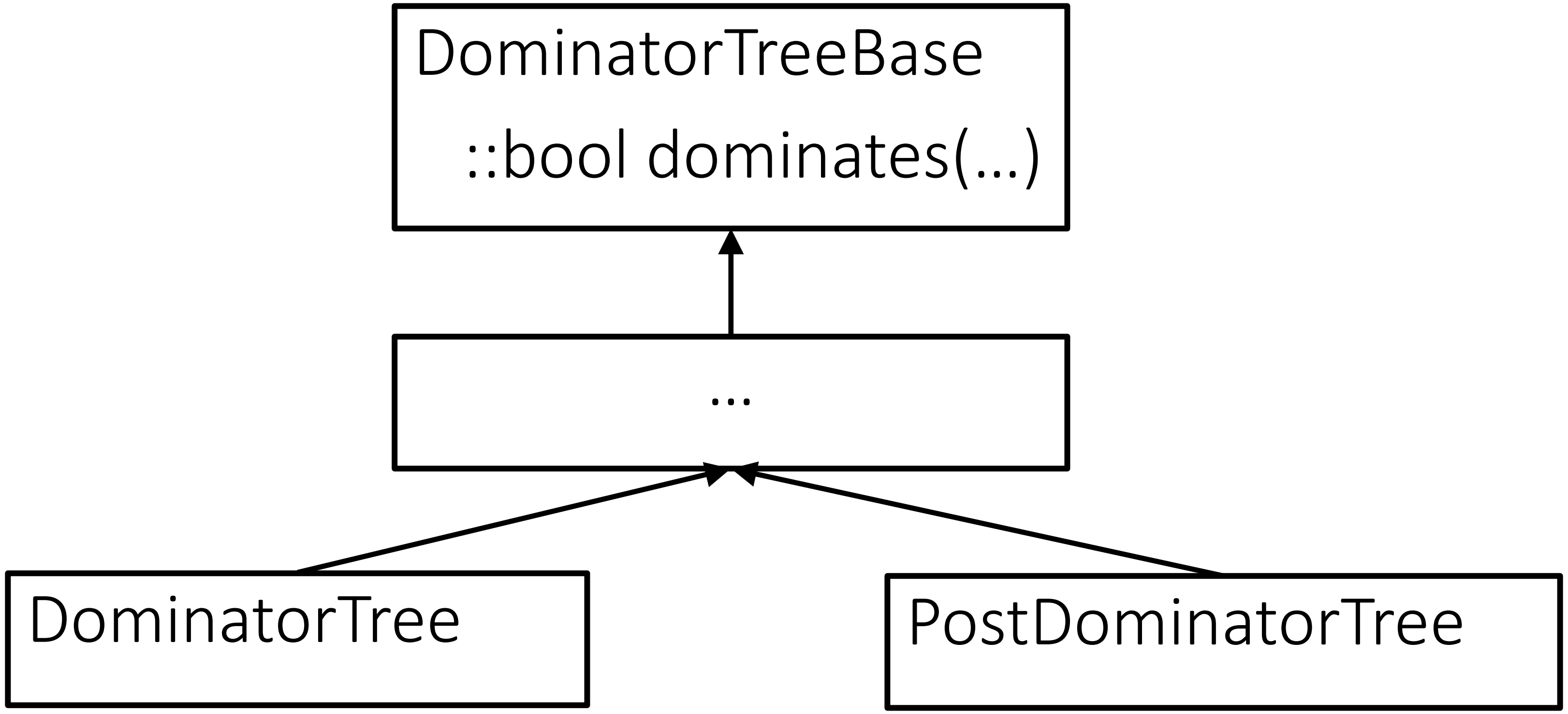
```
=== Post dominators
 %3 = alloca i32, align 4
 %10 = call i32 (i8*, ...) @printf(i8*
 %12 = call i32 (i8*, ...) @printf(i8*
 %14 = call i32 (i8*, ...) @printf(i8*
 ->   %10 = call i32 (i8*, ...) @printf(
 ->   %3 = alloca i32, align 4
 ->   %12 = call i32 (i8*, ...) @printf(
 store i32 0, i32* %3, align 4
 store i32 1, i32* %3, align 4
 %20 = load i32, i32* %3, align 4
 ->   store i32 0, i32* %3, align 4
 ->   %14 = call i32 (i8*, ...) @printf(
 ->   store i32 1, i32* %3, align 4
```

```
define dso_local i32 @main(i32, i8**) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  %6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([7 x i
  %7 = load i32, i32* %4, align 4
  %8 = icmp sgt i32 %7, 0
  br i1 %8, label %9, label %11

9:                                                ; preds = %2
  %10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([6 x
  br label %13

11:                                               ; preds = %2
  %12 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([6 x
  br label %13

13:                                               ; preds = %11, %9
  %14 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x
  %15 = load i32, i32* %4, align 4
  %16 = icmp eq i32 %15, 0
  br i1 %16, label %17, label %18

17:                                               ; preds = %13
  store i32 0, i32* %3, align 4
  br label %19

18:                                               ; preds = %13
  store i32 1, i32* %3, align 4
  br label %19

19:                                               ; preds = %18, %17
  %20 = load i32, i32* %3, align 4
  ret i32 %20
}
```

LLVM-specific notes for post dominators

- bool PostDominatorTree::dominates (…)
  - bool dominates (Instruction *i, Instruction *j)
    Return true if the basic block that includes i is an immediate post-dominator of the basic block that includes j
  - bool dominates (Instruction *i, BasicBlock *b)
    Return true if the basic block that includes i
    is an immediate post-dominator of b
- If the first argument
  is not reachable from the entry point of the function, return false
- If the second argument (either instruction or basic block)
  is not reachable from the entry point of the function, return true

LLVM-specific notes for *dominators

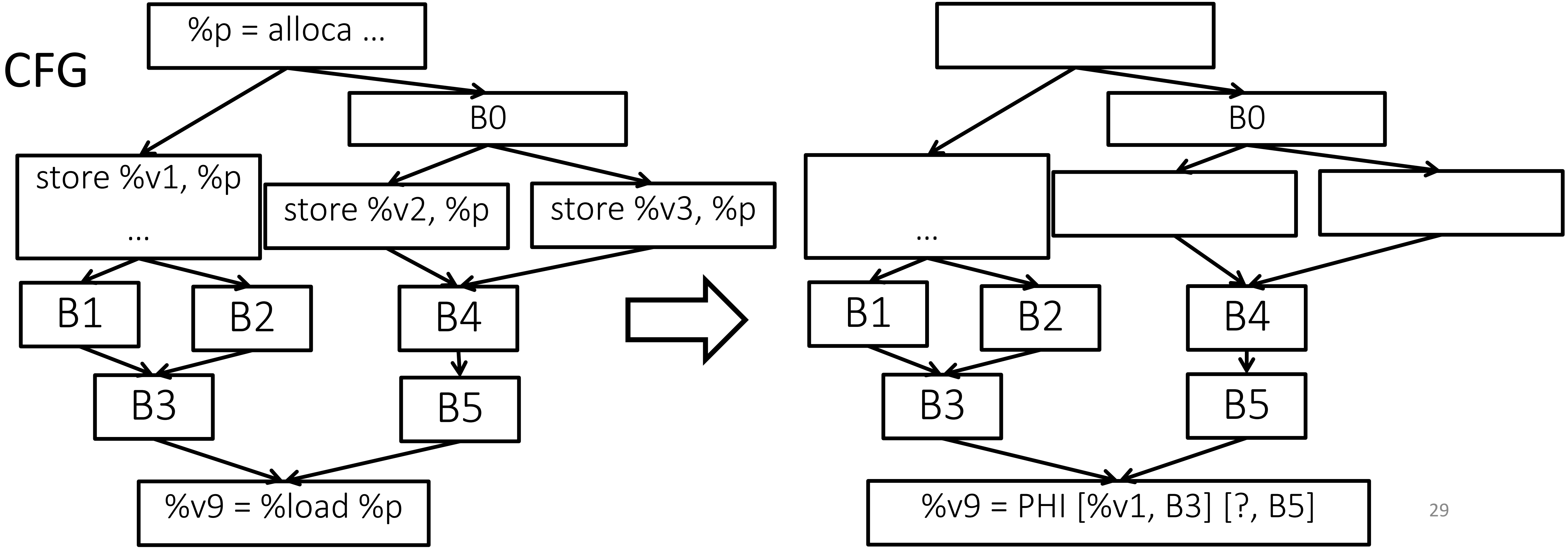


Outline

- CFA and a first example: dominators
- Another example of CFA: dominance frontier
- Example of CFA and CFT: basic block merging and splitting

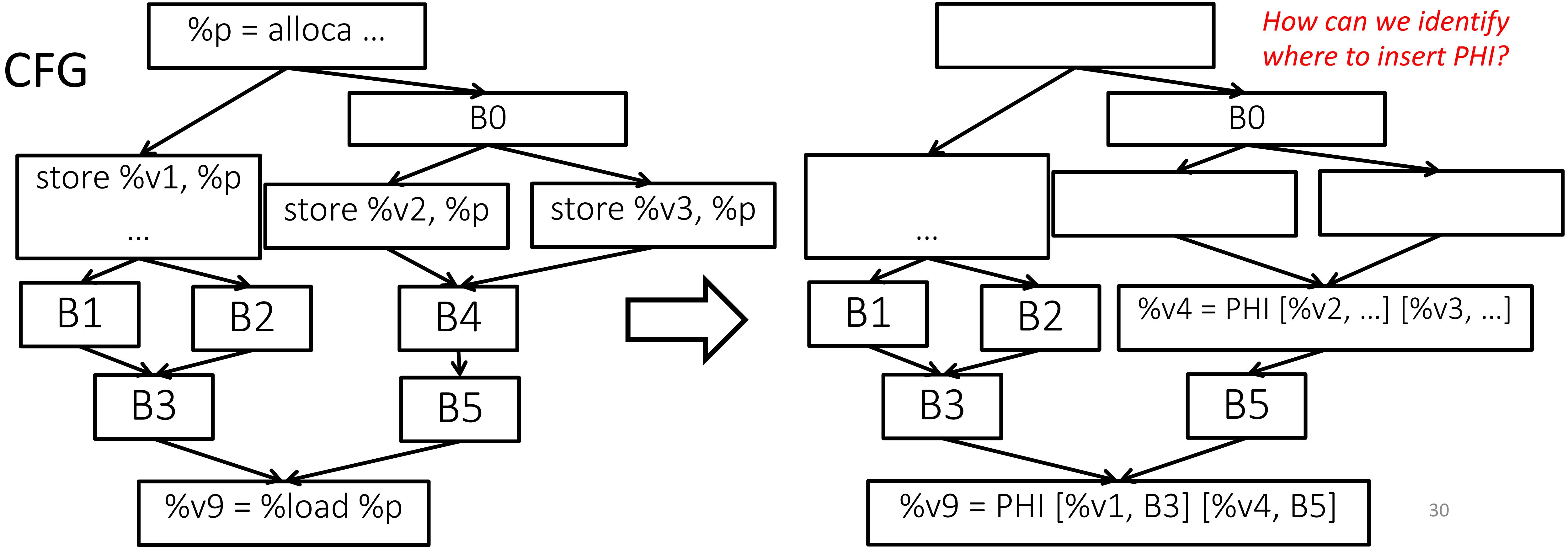
A problem: deciding where the place PHIs

- Problem:
  we would like to map a stack location into a set of IR variables



A problem: deciding where the place PHIs

- Problem:
  we would like to map a stack location into a set of IR variables



How can we identify where to insert PHI?

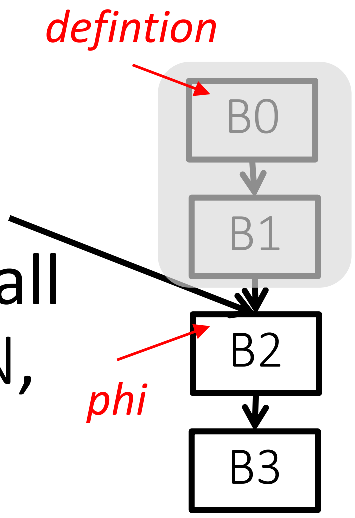
A problem: deciding where the place PHIs

- Problem:
  we would like to map a stack location into a set of IR variables

- Solutions:
  - Simple:
    insert PHI in all basic blocks for all variables (expensive)
  - Smarter:
    for each variable, identify the subset of basic blocks that need PHI

Dominance frontier

- Dominators of block N tell us which basic blocks **must** be executed prior to N
- We need to identify blocks “just after” those blocks that are dominated by N
- **Definition:**
  The Dominance Frontier of a basic block N, DF(N), is the set of all blocks that are immediate successors to blocks dominated by N, but which aren’t strictly dominated by N



DF(B0) = {B2}

Definition

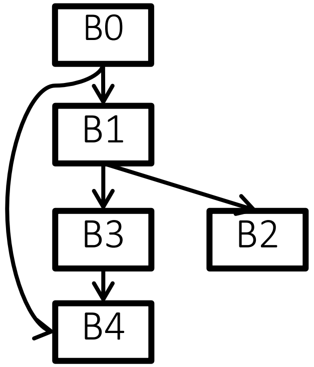
Where to insert PHI

Dominance frontier

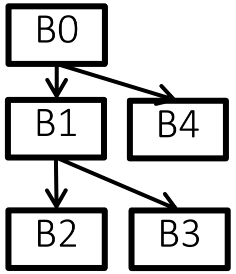
- **Definition:**
  The Dominance Frontier of a basic block N, DF(N), is the set of all blocks that are immediate successors to blocks dominated by N,
  but which aren't strictly dominated by N
- DF(N) includes a basic block X if and only if
  1. N dominates a predecessor of X
  2. N does not strictly dominate X
- How can we compute DF(N)?

Dominance frontier computation

1. From the CFG



2. Compute the dominators tree



3. Compute the local dominance frontier for all nodes

$DF_{local}[N]$ = successors of N in the CFG
that are not strictly dominated by N

$DF_{local}[B0] = \{\}$
$DF_{local}[B1] = \{\}$
$DF_{local}[B2] = \{\}$
$DF_{local}[B3] = \{B4\}$
$DF_{local}[B4] = \{\}$

3. Compute the dominance frontier

$$DF[N] = DF_{local}[N] \cup_{c \in children(N)} DF_{up}[c]$$

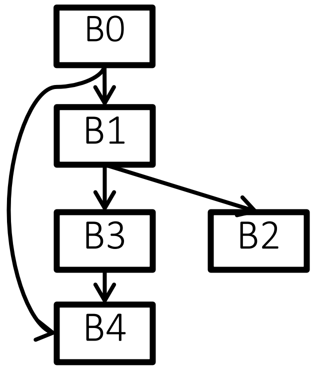
Dominance frontier computation

Compute the dominance frontier

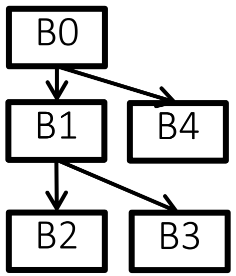
$DF[N] = DF_{local}[N] \cup_{c \in children(N)} DF_{up}[c]$

$Df_{local}[N]$ = successors of N in the CFG
that are not strictly dominated by N

$DF_{up}[c]$ = nodes in DF[c] that are not strictly dominated by parent(c) of the dominator tree



CFG



Dominator tree

$DF_{local}[B0] = \{\}$	$DF[B0] = \{\}$
$DF_{local}[B1] = \{\}$	$DF[B1] = \{\}$
$DF_{local}[B2] = \{\}$	$DF[B2] = \{\}$
$DF_{local}[B3] = \{B4\}$	$DF[B3] = \{B4\}$
$DF_{local}[B4] = \{\}$	$DF[B4] = \{\}$

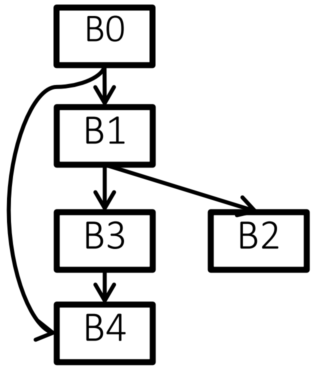
Dominance frontier computation

Compute the dominance frontier

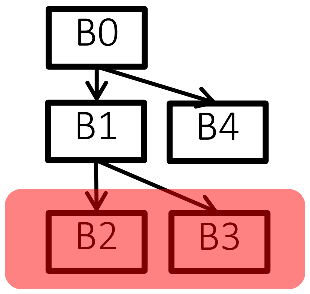
$DF[N] = DF_{local}[N]\ \underline{\bigcup_{c \in children(N)} DF_{up}[c]}$

$Df_{local}[N]$ = successors of N in the CFG
that are not strictly dominated by N

$DF_{up}[c]$ = nodes in DF[c] that are not strictly dominated by parent(c) of the dominator tree



CFG



Dominator tree

$DF_{local}[B0] = \{\}$	$DF[B0] = \{\}$
$DF_{local}[B1] = \{\}$	$DF[B1] = \{\}$
$DF_{local}[B2] = \{\}$	$DF[B2] = \{\}$
$DF_{local}[B3] = \{B4\}$	$DF[B3] = \{B4\}$
$DF_{local}[B4] = \{\}$	$DF[B4] = \{\}$

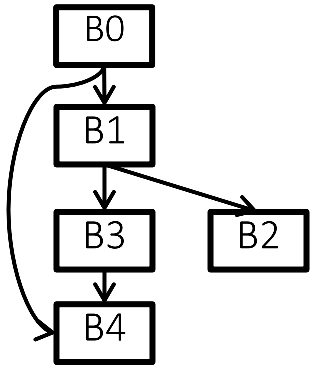
Dominance frontier computation

Compute the dominance frontier

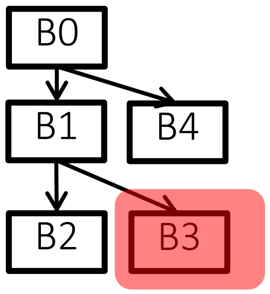
$DF[N] = DF_{local}[N] \cup_{c \in children(N)} DF_{up}[c]$

$Df_{local}[N]$ = successors of N in the CFG
that are not strictly dominated by N

$DF_{up}[c]$ = nodes in DF[c] that are not strictly dominated by parent(c) of the dominator tree



CFG



Dominator tree

$DF_{local}[B0] = \{\}$ $\quad$ $DF[B0] = \{\}$

$DF_{local}[B1] = \{\}$ $\quad$ $DF[B1] = \{\}$ ←

$DF_{local}[B2] = \{\}$ $\quad$ $DF[B2] = \{\}$

$DF_{local}[B3] = \{B4\}$ $\quad$ $DF[B3] = \{B4\}$

$DF_{local}[B4] = \{\}$ $\quad$ $DF[B4] = \{\}$

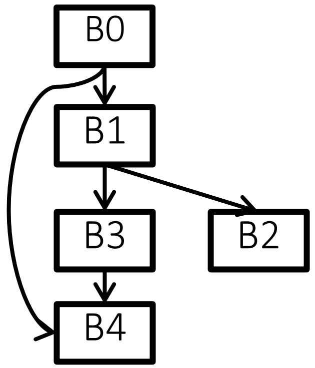
Dominance frontier computation

Compute the dominance frontier

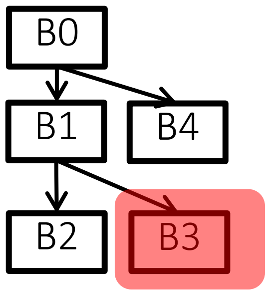
$DF[N] = DF_{local}[N]\ \underline{\bigcup_{c \in children(N)} DF_{up}[c]}$

$Df_{local}[N]$ = **successors of** N **in the CFG**
that are not strictly dominated by N

$DF_{up}[c]$ = **nodes in** DF[c] **that are not strictly dominated by** parent(c)
of the dominator tree



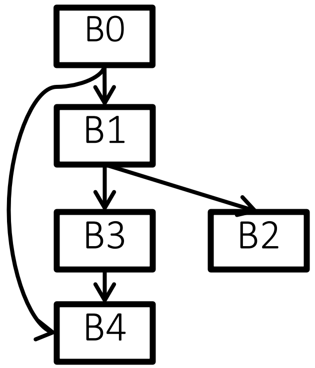
CFG



Dominator tree

$DF_{local}[B0] = \{\}$	$DF[B0] = \{\}$
$DF_{local}[B1] = \{\}$	$DF[B1] = \{B4\}$
$DF_{local}[B2] = \{\}$	$DF[B2] = \{\}$
$DF_{local}[B3] = \{B4\}$	$DF[B3] = \{B4\}$
$DF_{local}[B4] = \{\}$	$DF[B4] = \{\}$

Dominance frontier



CFG

DF[B0] = {}
DF[B1] = {B4}
DF[B2] = {}
DF[B3] = {B4}
DF[B4] = {}

If I have a re-definition in B0:

- No need for a phi

If I have a re-definition in B1:

- I need to add a phi in B4

This is how mem2reg *decides where to inject phis for* alloca *that can be safely removed*

Outline

- CFA and a first example: dominators
- Another example of CFA: dominance frontier
- Example of CFA and CFT: basic block merging and splitting

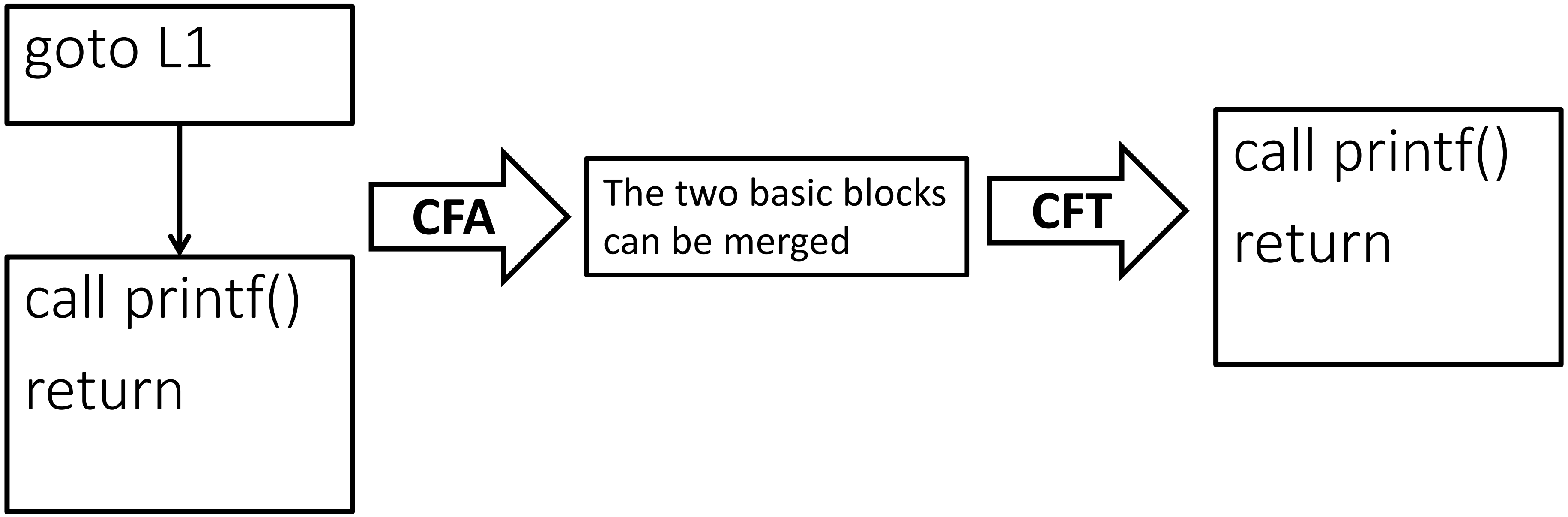
Another example of CFA (and CFT)

A homework of this class could be the following one:
design and implement an algorithm to implement this CFA

- CFA: it says whether it is safe to merge two basic blocks
- CFT: it merges only the basic block pairs identified by the CFA

Existing LLVM pass: simplifycfg

```
    goto L1
L1: call printf()
    return
```

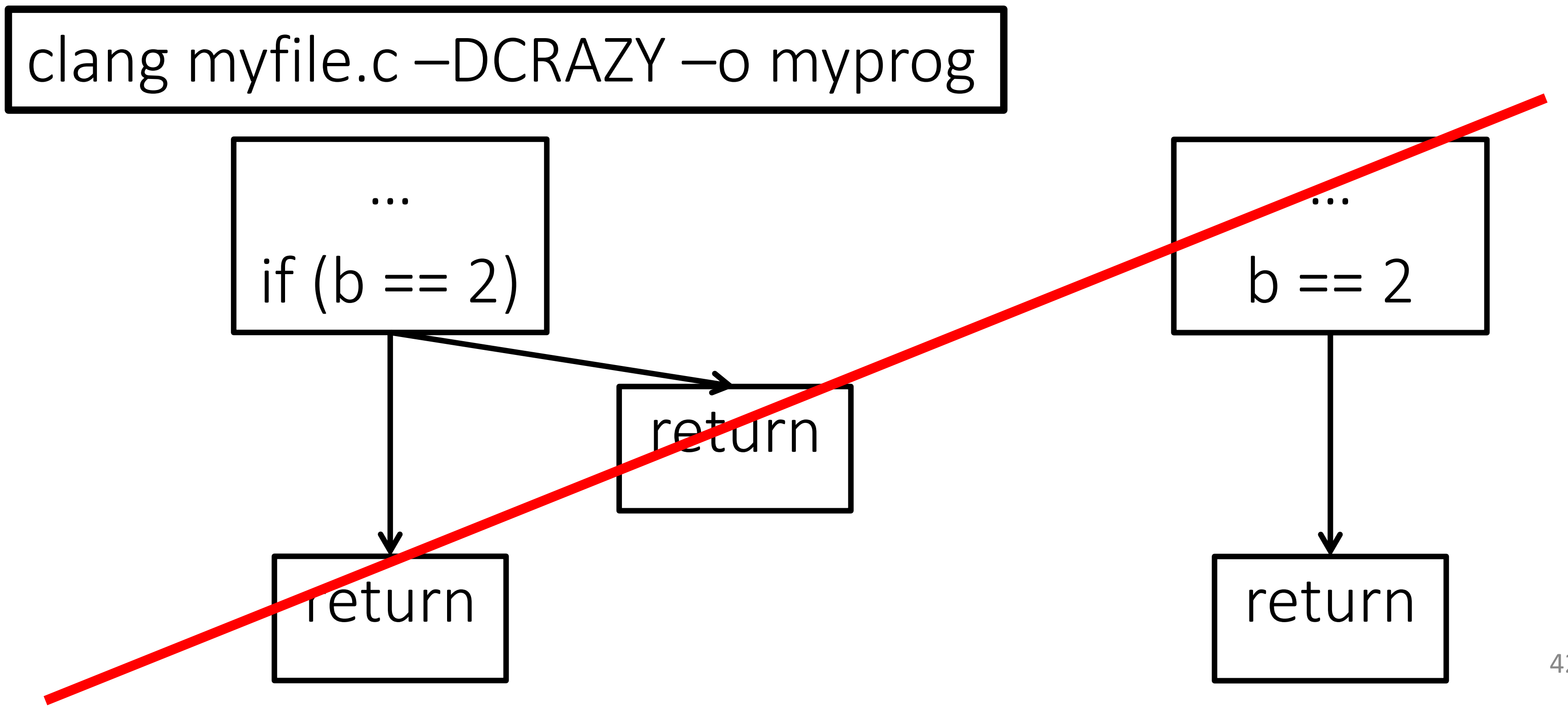


This is a simple CFA and CFG,
but useful after applying several other code transformations

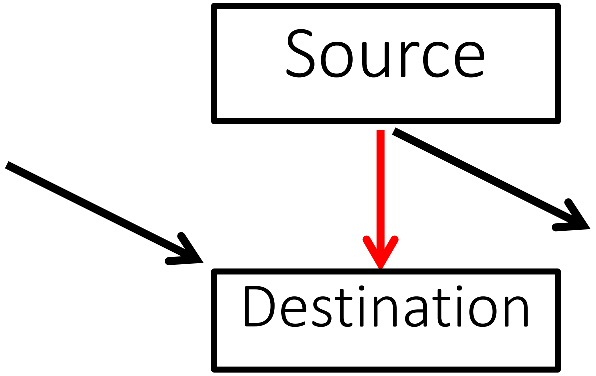
Another example of CFA

- What are the possible equivalent CFGs the compiler can choose from?
- The compiler needs to be able to transform CFGs
  - CFAs tell the compiler what are the equivalent CFGs

```
…
If (b == 2){
    return;
}
#ifdef CRAZY
printf(“Yep”);
#endif
return;
```



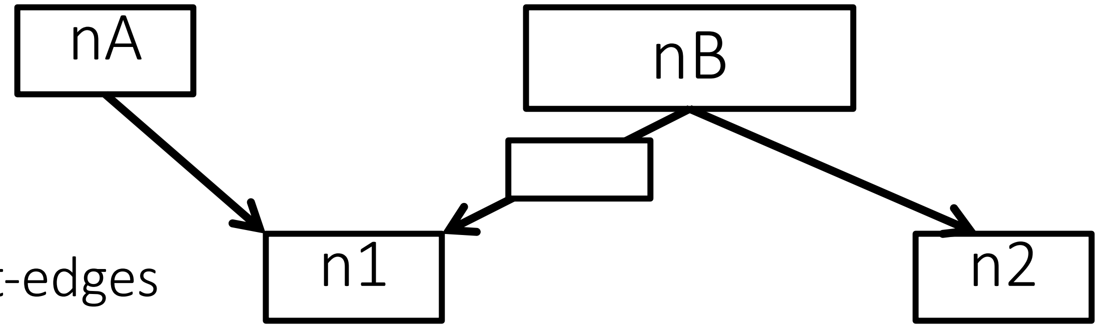
Critical edges



Definition:
A ***critical edge*** is an edge in the CFG
which is neither the only edge leaving its source block,
nor the only edge entering its destination block.

These edges must be *split*: a new block must be created
and inserted in the middle of the edge,
to insert computations on the edge without affecting any other edges.

opt --break-crit-edges




```
If (…){
   while (…){
      …
   }
}
A()
```

Always have faith in your ability

Success will come your way eventually

Best of luck!