Code analysis *and* transformation

DFA

Simone Campanoni
simone.campanoni@northwestern.edu

Data Flow Analysis outline

- Concepts needed by most code analyses
- Why do we need DFA? (opportunities)
- Introduction to DFA (concepts)
- A DFA example: reaching definitions (concept application)
- Implementation of DFA (actual implementation)

Variables and constants

x = 0;

y = x + 1;

Constants

Variable definitions

Variable uses

Now that we know variables,
we can talk about how
data ~~stored in them could evolve~~ through the code
flows

Now that we know variables,
we can talk about how
data flows through the code

Data flows

```
int sumcalc (int a, int b, int N){
  int x,y;
  x = 0;
  y = 0;
  for (int i=0; i <= N; i++){
    x = x + (a * b);
    x = x + b*y;
  }
  return x;
}
```

Data flows from a definition to its uses

Data flow examples

```
int sumcalc (int a, int b, int N){
  int x,y;
  x = 0;
  y = 0;
  for (int i=0; i <= N; i++){
    x = x + (a * b);
    x = x + b*y;
  }
  return x;
}
```

Understanding data flows require understanding the ~~possible sequence of instructions that could be executed at run time~~ control flows

Control flows

Control flow: sequence of instructions in a program that may execute at run-time in that order

(common simplification: we ignore data values and arithmetic operations)

```
x = a;
y = x + 1;
x++;
return x + y;
```

```
x = a;
y = x + 1;
if (y > 5){
  x--;
} else {
  x++;
}
```

How can we automatically identify and represent the control flows?

Let us start by looking at

how to iterate over instructions of a function

in LLVM

Functions and instructions

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

for (auto& inst : instructions(F)){
  errs() << inst << "\n";
}
```

Iteration order:
Follows the order used to store instructions in a function F

What is the instruction that will be executed after inst?

The iteration order of instructions isn't the execution one
We cannot use iteration order to analyze data flows

Storing order ≠ executing order

A storing order

```
int myF (int a){
  int x = a + 1;
  if (a > 5){
    x++;
  } else {
    x--;
  }
  return x; }
```

```
    int x = a + 1
    tmp = a > 5
    branch_ifnot tmp L1
    x++
    branch L2
L1: x--
L2: return x
```

What is the next instruction executed?

```
    int x = a + 1
    tmp = a > 5
    branch_if tmp L1
    x--
    branch L2
L1: x++
L2: return x
```

Another (correct) storing order

When the storing order is chosen (compile time), the execution order isn't known

Storing order ≠ executing order

~~**Common pitfall 1:**~~
~~**if instruction i1 has been stored before i2,**~~
~~**then i2 is always executed after i1**~~

i1
i2

~~**Common pitfall 2:**~~
~~**if instruction i1 has been stored before i2,**~~
~~**then i2 can execute after i1**~~

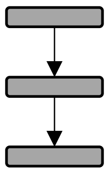
How can we automatically identify and represent the control flows?

We could represent the control flows using a directed graph:

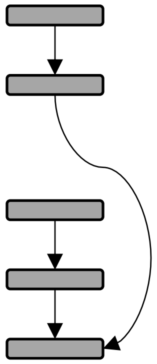
- Node: instruction
- Direct edge: points to the possible next instruction that could be executed at run-time

Representing the control flow of the program

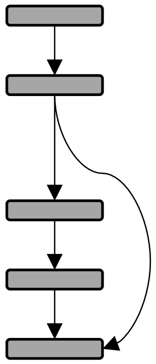
- Most instructions



- Jump instructions



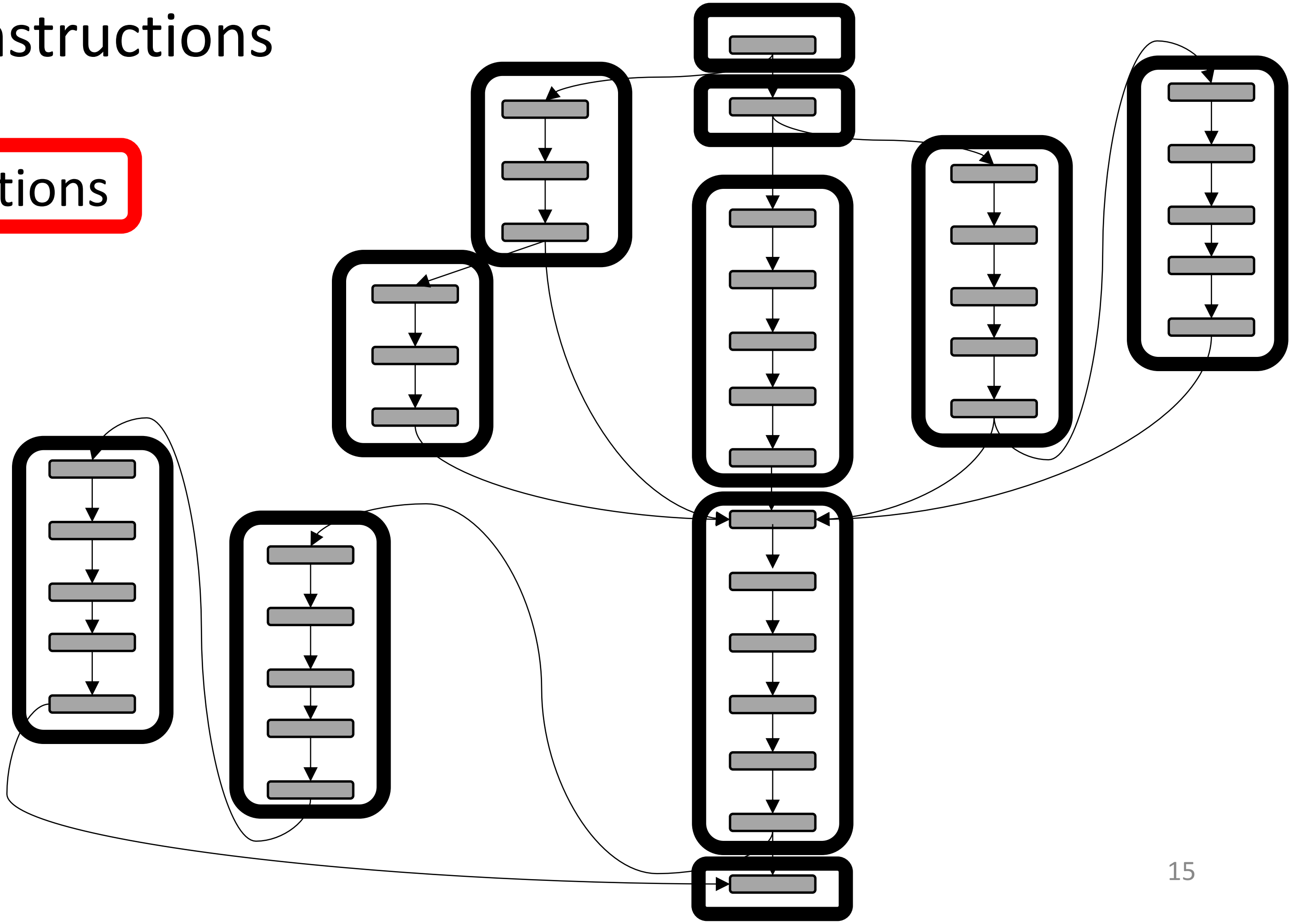
- Branch instructions



Representing the control flow of the program

A graph where nodes are instructions

- Very large
- Lot of straight-line connections
- Can we simplify it?



Basic block

Sequence of instructions that is always
entered at the beginning
and exited at the end

Basic blocks

A basic block is a maximal sequence of instructions such that

- Only the first one can be reached
  from outside this basic block

- All instructions within are executed consecutively
  if the first one get executed
  - Only the last instruction can be a branch/jump
  - Only the first instruction can be a label

- The storing sequence is the execution order in a basic block

Basic blocks in compilers

- Automatically identified
  - Algorithm:

```
Inst = F.entryPoint()
B = new BasicBlock()
While (Inst){
  if Inst is Label {
    B = new BasicBlock()
  }
  B.add(Inst)
  if Inst is Branch/Jump{
    B = new BasicBlock()
  }
  Inst = F.nextInst(Inst)
}
Add missing labels
Add explicit jumps
Delete empty basic blocks
```

Basic blocks in compilers

- Automatically identified

```
Inst = F.entryPoint()
B = new BasicBlock()
While (Inst){
  if Inst is Label {
    B = new BasicBlock()
  }
  B.add(Inst)
  if Inst is Branch/Jump{
    B = new BasicBlock()
  }
  Inst = F.nextInst(Inst)
}
Add missing labels
Add explicit jumps
Delete empty basic blocks
```

  - Algorithm:
  - Code changes trigger the re-identification
  - Increase the compilation time
- Enforced by design
  - Instruction exists only within the context of its basic block
  - To define a function:
    - you define its basic blocks first
    - Then you define the instructions of each basic block

Basic blocks in compilers

- Automatically identified

```
Inst = F.entryPoint()
B = new BasicBlock()
While (Inst){
  if Inst is Label {
    B = new BasicBlock()
  }
  B.add(Inst)
  if Inst is Branch/Jump{
    B = new BasicBlock()
  }
  Inst = F.nextInst(Inst)
}
Add missing labels
Add explicit jumps
Delete empty basic blocks
```

  - Algorithm:
  - Code changes trigger the re-identification
  - Increase the compilation time
- Enforced by design
  - Instruction exists only within the context of its basic block
  - To define a function:
    - you define its basic blocks first
    - Then you define the instructions of each basic block

What about calls?
- *Program exits*
- *Infinite loops in callees*

Basic blocks in LLVM

- Every basic block in LLVM must
  - Have a label associated to it
  - Have a "terminator" at the end of it

```
6:
  store i32 0, i32* %2, align 4
  br label %14
```

- The first basic block of LLVM (entry point) cannot have predecessors

Basic blocks in LLVM

- LLVM organizes “compiler concepts” in containers
  - A basic block is a container of ordered LLVM instructions (BasicBlock)
  - A function is a container of basic blocks (Function)
  - A module is a container of functions (Module)

```
for (auto& B : F) {
  for (auto& I : B) {
    I.print(errs());
    errs() << "\n";
  }
}
```

Basic blocks in LLVM (2)

- **LLVM C++ Class** “BasicBlock”
- **Uses:**
  - BasicBlock *b = … ;
  - Function *f = b.getParent();
  - Module *m = b.getModule();
  - Instruction *i = b.getTerminator();
  - Instruction *i = b.front();
  - size_t b.size();

Basic blocks in LLVM in action

```
function()
{
    int a = 1;               //Sequential instructions
    int b = 2;

    if (b == 2)              //Jump instruction
    {
        ++b;                 //Jump target
    }

    int c = 3;               //Sequential instructions
    int d = 4;               //----------------------

    while (a < 5)            //Jump instruction and jump target
    {
        ++a;                 //Jump target
    }

    int e = 5;               //Sequential instructions
    int f = 6;               //----------------------
}
```

Bitcode generation

```
void @_Z8functionv() #0 {
entry:
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %d = alloca i32, align 4
  %e = alloca i32, align 4
  %f = alloca i32, align 4
  store i32 1, i32* %a, align 4
  store i32 2, i32* %b, align 4
  %0 = load i32* %b, align 4
  %cmp = icmp eq i32 %0, 2
  br i1 %cmp, label %if.then, label %if.end
```

- All function variables are declared at the beginning of the function
- A variable access becomes a memory access

Basic blocks in LLVM in action

```
function()
{
    int a = 1;        //Sequential instructions
    int b = 2;        //-----------------------

    if (b == 2)       //Jump instruction
    {
        ++b;
    }

    int c = 3;        //Sequential
    int d = 4;

    while (a < 5)     //Jump instruct
    {
        ++a;          target
    }

    int e = 5;
    int f = 6;        //-----------------------
}
```

Bitcode generation

Bitcode generation

Bitcode generation

Bitcode generation

```
while.cond:
  %2 = load i32* %a, align 4
  %cmp1 = icmp slt i32 %2, 5
  br i1 %cmp1, label %while.body, label %while.end

while.body:
  %3 = load i32* %a, align 4
  %inc2 = add nsw i32 %3, 1
  store i32 %inc2, i32* %a, align 4
  br label %while.cond

while.end:
  store i32 5, i32* %e, align 4
  store i32 6, i32* %f, align 4
  ret void
}
```

How can we automatically identify and represent
the control flows?

We could represent the control flows using a directed graph:

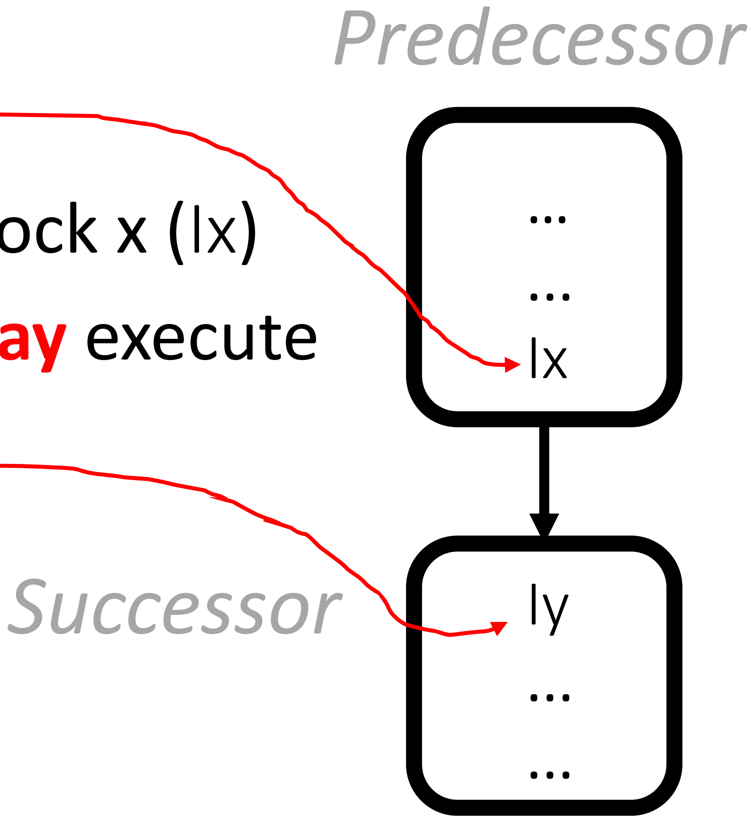
- Node: ~~instruction~~ Basic block
- Direct edge: points to the possible next instruction
  that could be executed at run-time

Control Flow Graph (CFG)

- A CFG is a graph G = <Nodes, Edges>
- Nodes: Basic blocks
- Edges: (x,y) ϵ Edges if and only if

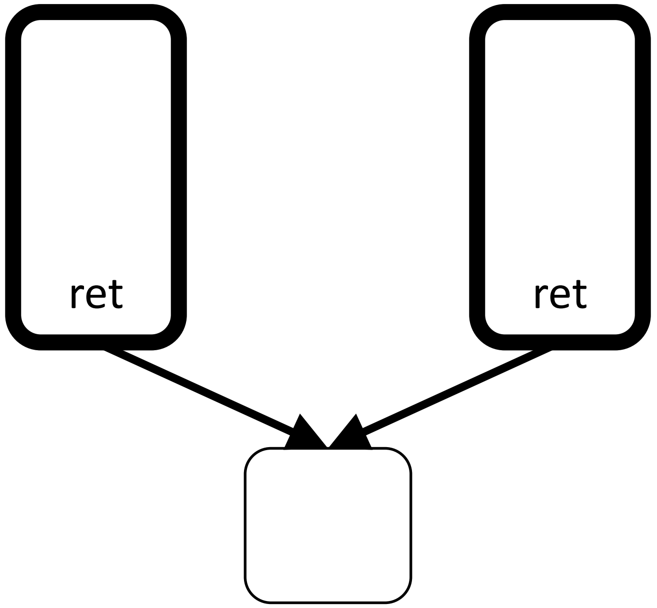
  after executing the last instruction of basic block x (lx)

  the first instruction of the basic block x (lx) **may** execute



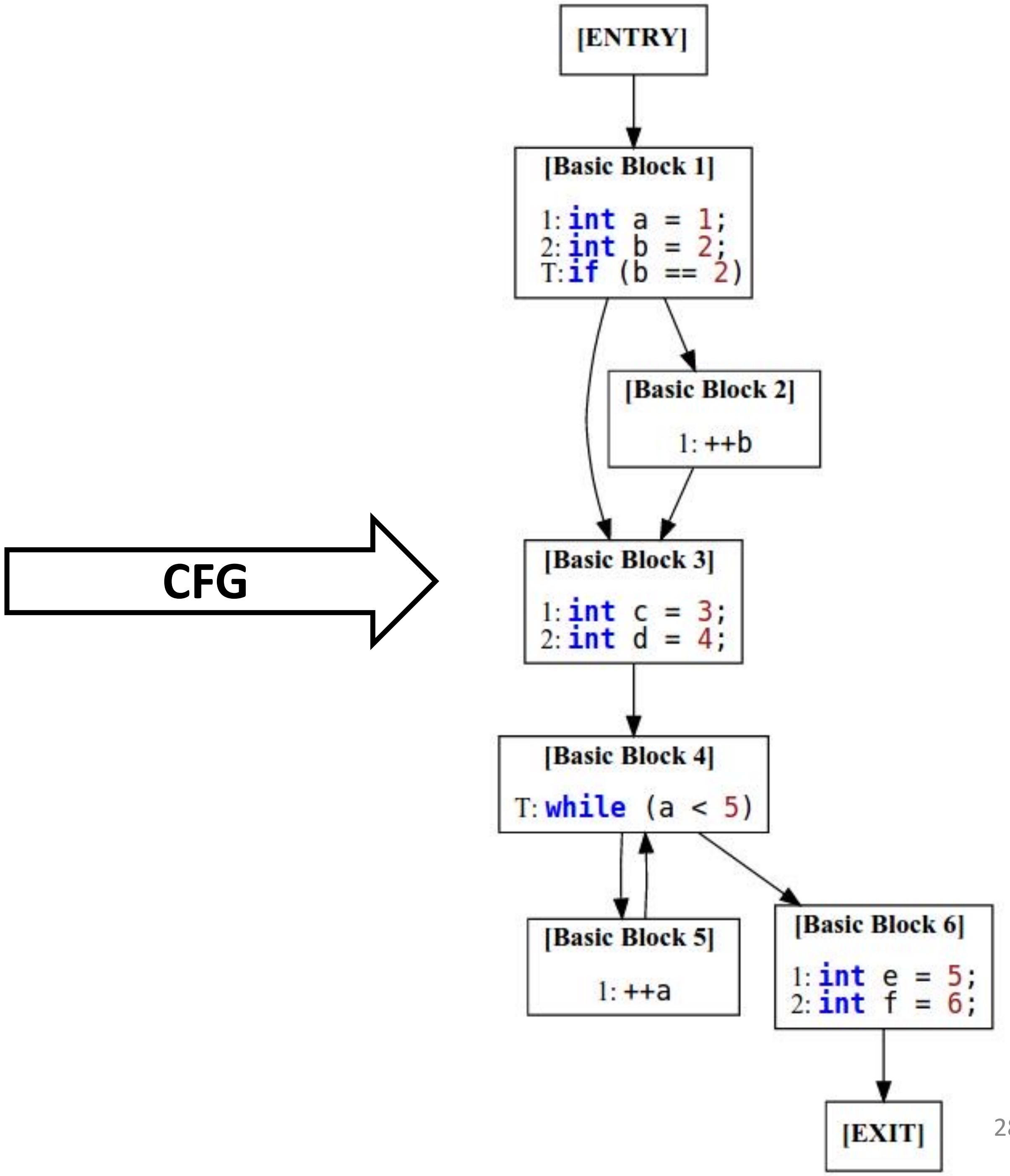
Control Flow Graph (CFG)

- Entry node: block with the first instruction of the function
- Exit nodes: blocks with the return instruction
  - Some compilers make a single exit node by adding a special node

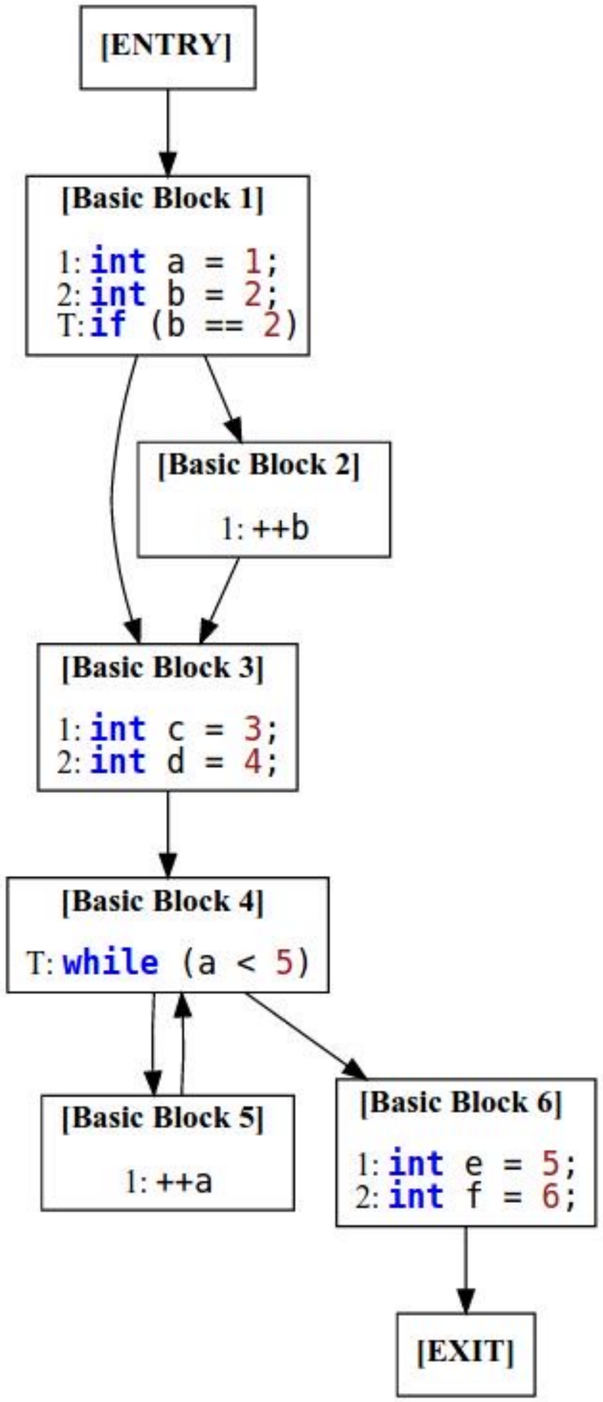


CFG example

```
function()
{
    int a = 1;        //Sequential instructions
    int b = 2;        //----------------------

    if (b == 2)       //Jump instruction
    {
        ++b;          //Jump target
    }

    int c = 3;        //Sequential instructions
    int d = 4;        //----------------------

    while (a < 5)     //Jump instruction and jump target
    {
        ++a;          //Jump target
    }

    int e = 5;        //Sequential instructions
    int f = 6;        //----------------------
}
```

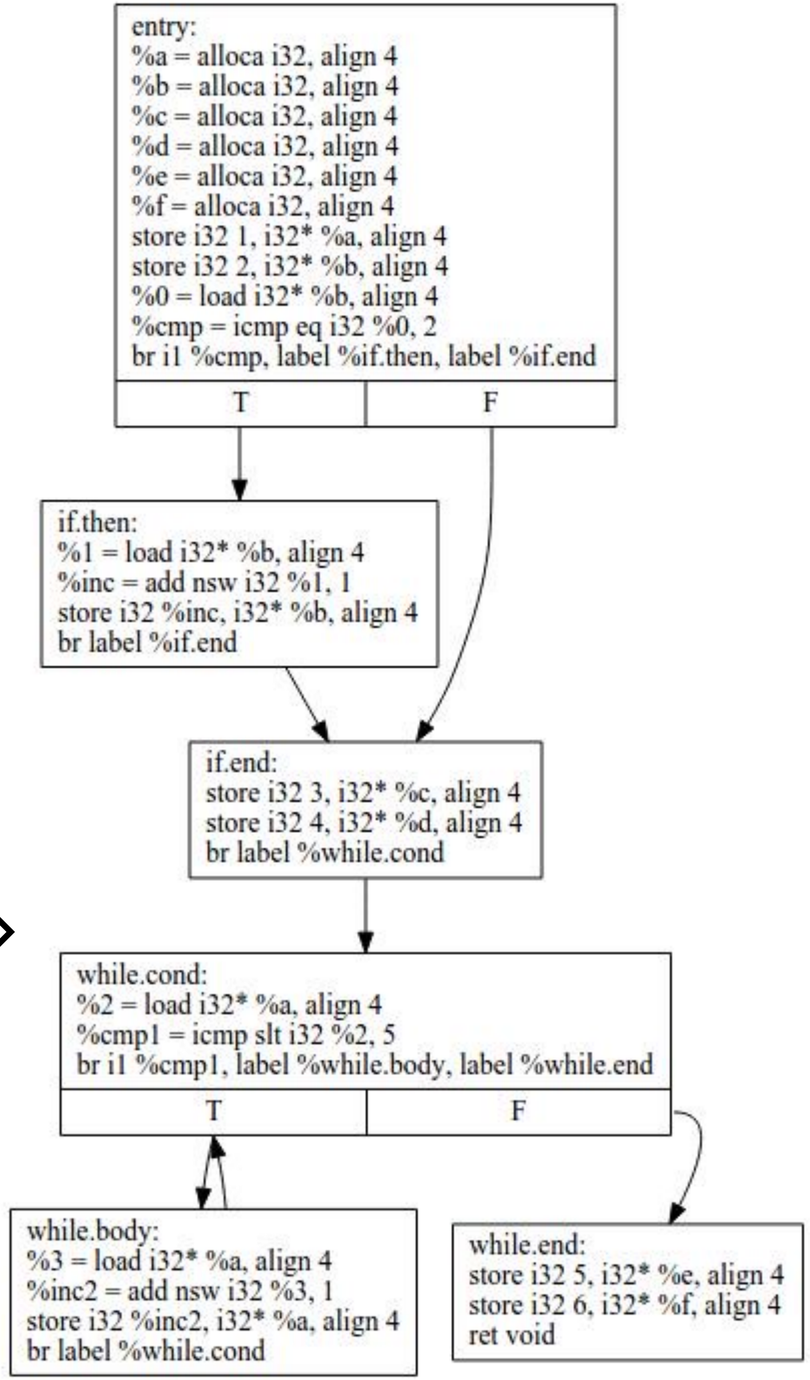


CFG in LLV



Differences?

Bitcode generation



opt -view-cfg

F.viewCFG();

Navigating the instructions within a basic block

```
auto nextInstruction = i->getNextNode();
auto prevInstruction = i->getPrevNode();
```

Navigating the CFG in LLVM: from a basic block to another

```
for (auto& B : F) {
  for (auto& I : B) {
    I.print(errs());
    errs() << "\n";
  }
}
```

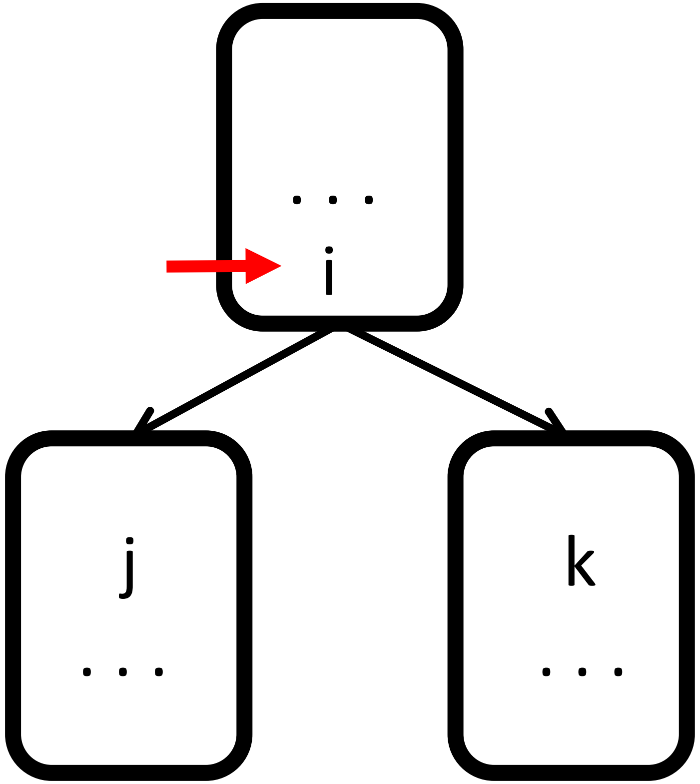
Successors of a basic block

```
for (auto succBB : successors(bb)){
```

Predecessors of a basic block

```
for (auto predBB : predecessors(bb)){
```

Navigating the CFG in LLVM: From an instruction to its successors



Let's say we want to iterate over the successors of i
so from i to j and k

How can we do it?

```
for (auto succBB : successors(bb)){
```

Navigating the CFG in LLVM:
From an instruction to its successors

```
for (auto &b : F){
  auto i = b.getTerminator();
  errs() << *i << "\n";
  for (auto succBB : successors(b)){
    auto firstInstOfSuccBB = succBB->front();
  }
}
```

```
define i32 @CAT_execution() #0 {
entry:
  %res = alloca i32, align 4
  %i = alloca i32, align 4
  %call = call i32 (i8*, ...) @printf(i8* getel
  store i32 0, i32* %res, align 4
  store i32 0, i32* %i, align 4
  br label %for.cond

for.cond:
  %0 = load i32, i32* %i, align 4
  %cmp = icmp slt i32 %0, 10000
  br i1 %cmp, label %for.body, label %for.end

for.body:
  %1 = load i32, i32* %res, align 4
  %inc = add nsw i32 %1, 1
  store i32 %inc, i32* %res, align 4
  br label %for.inc

for.inc:
  %2 = load i32, i32* %i, align 4
  %inc1 = add nsw i32 %2, 1
  store i32 %inc1, i32* %i, align 4
  br label %for.cond

for.end:
  %3 = load i32, i32* %res, align 4
  ret i32 %3
}
```

H0/tests

Output of the LLVM pass of the previous slide:

```
br label %for.cond
 %0 = load i32, i32* %i, align 4
br i1 %cmp, label %for.body, label %for.end
 %1 = load i32, i32* %res, align 4
 %3 = load i32, i32* %res, align 4
br label %for.inc
 %2 = load i32, i32* %i, align 4
br label %for.cond
 %0 = load i32, i32* %i, align 4
ret i32 %3
```

Now that we know how to traverse over the CFG,
we can introduce the first code transformation

Code transformation example: constant propagation

```
int sumcalc (int a, int b, int N){
  int x,y;
  x = 0;
  y = 0;
  for (int i=0; i <= N; i++){
    x = x + (a * b);
    x = x + b*y;
  }
  return x;
}
```

Replace a variable use
with a constant
while preserving
the original code semantics

Code transformation example: constant propagation

```
int sumcalc (int a, int b, int N){
  int x,y;
  x = 0;
  y = 0;
  for (int i=0; i <= N; i++){
    x = x + (a * b);
    x = x + b*y;
  }
  return x;
}
```

Replace a variable use
with a constant
while preserving
the original code semantics

Code transformation example: constant propagation

```
int sumcalc (int a, int b, int N){
  int x,y;
  x = 0;
  y = 0;
  for (int i=0; i <= N; i++){
    x = x + (a * b);
    x = x + b*0;
  }
  return x;
}
```

Replace a variable use
with a constant
while preserving
the original code semantics

Understanding requires understanding ~~how the value of a variable could evolve over time~~ data flows

and this is the job of data flow analyses

Data Flow Analysis outline

- Concepts needed by most code analyses
- Why do we need DFA? (opportunities)
- Introduction to DFA (concepts)
- A DFA example: reaching definitions (concept application)
- Implementation of DFA (actual implementation)

The need for DFAs

- We constantly need to improve programs
  (e.g., speed, energy efficiency, memory requirements)
- We constantly need to identify opportunities
- After having found an opportunity (e.g., propagating constants),
  you need to ask yourself:
  - What do I need to know to take advantage of this opportunity?
    (e.g., I need to know the possible values a given variable might have
    at a given point in the program)
  - How can I automatically compute this information?
    Often the solution relies on understanding
    how data flows through the code.
    This is often done by designing custom DFAs

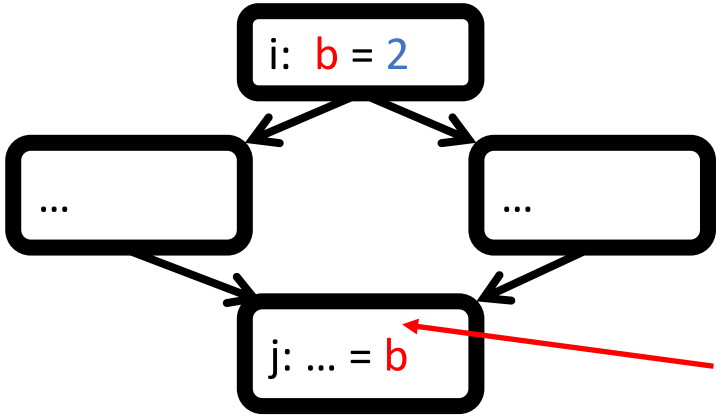
Let us go deeper in the need for data flow analysis
for code transformation

Let us introduce an actual code transformation implemented
by all compilers: constant propagation

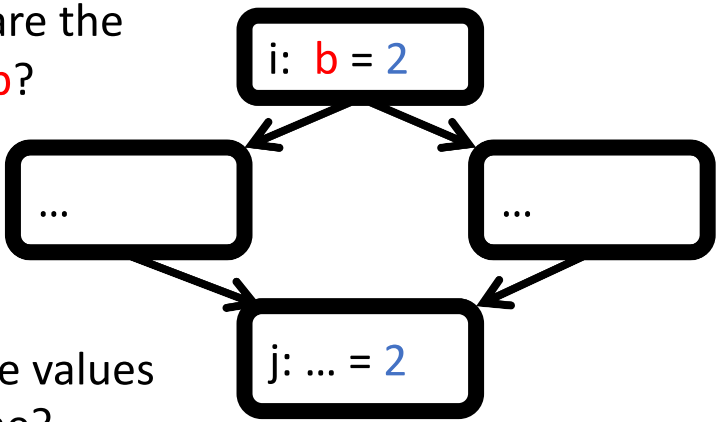
Transformation: constant propagation
Analysis: reaching definition DFA

- Opportunity: this code is “*better*” compared to this

Which information do I need to know if it is safe to replace b with 2



Among all possible run time control flows, what are the latest definitions of b?



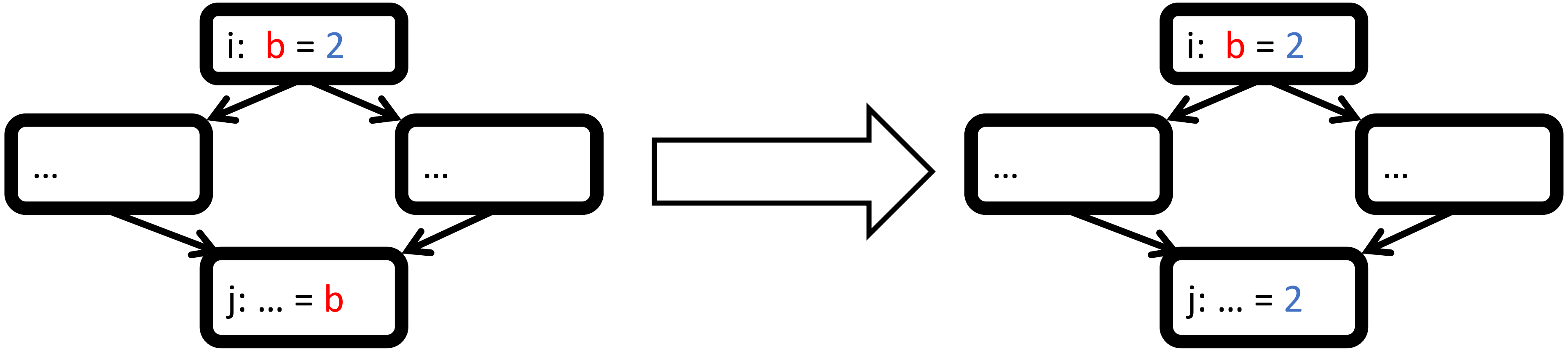
What are the possible values b can have at run time?

Constant propagation

- Find an instruction *i* that defines a variable with a constant expression

  Instruction i: b = CONSTANT_EXPRESSION

- Replace an use of b in an instruction *j* with that CONSTANT_EXPRESSION if
  - All control flows to *j* includes *i*
  - There are no intervening definition of that variable



Constant propagation: code example

```
int sumcalc (int a, int b, int N){
  int x,y;
  x = 0;
  y = 0;
  if (a > b){
    x = x + N;
  }
  if (b > N){  return y;}
  return x;
}
```

```
if (b > N){  return 0;}
```

Data-flow analysis is
a collection of techniques
for compile-time reasoning about
the run-time values

We need to analyze the "data-flows" of a program and represent them explicitly

But constant propagation (CP) has been done already …

- CP has been already designed and implemented

- Why should we study it?
  Why don’t we design and implement all possible transformations and analyses in a compiler and move on?

- It is always possible to invent new/better transformations

Full employment theorem for compiler writers

Since it is always possible to improve transformations,
let us learn the typical approach to create new data-flow analyses
that will drive the innovation

New transformations and analyses

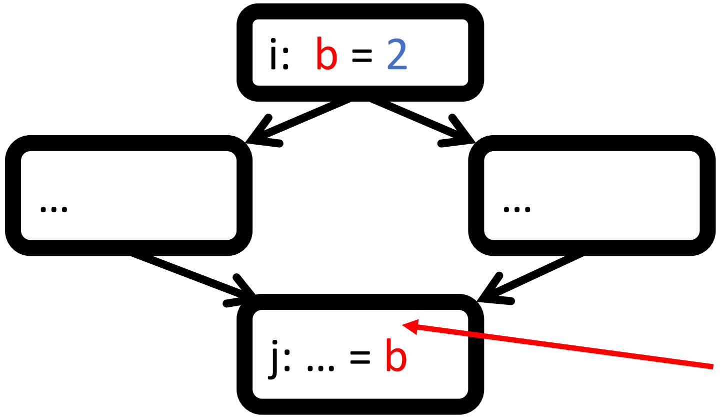
- New transformations (often) need to understand specific and new code properties related to how data might change through the code
  - So we need to know how to design a new data flow analysis that identifies these new code properties
- Generic recipe

  Data flow analysis (DFA):
  traverse the CFGs collecting information about what may happen at run time (Conservative approximation)

  Data flow value

  Transformation:
  Modify the code based on the result of data flow analysis
  (Correctness guaranteed by the conservative approximation of DFA)

New transformations and analyses



Among all possible run time control flows,
what are the latest definitions of b?

Data flow value

- Generic recipe

  Data flow analysis (DFA):
  traverse the CFGs collecting information about
  what may happen at run time (Conservative approximation)

  Transformation:
  Modify the code based on the result of data flow analysis
  (Correctness guaranteed by the conservative approximation of DFA)

Data Flow Analysis outline

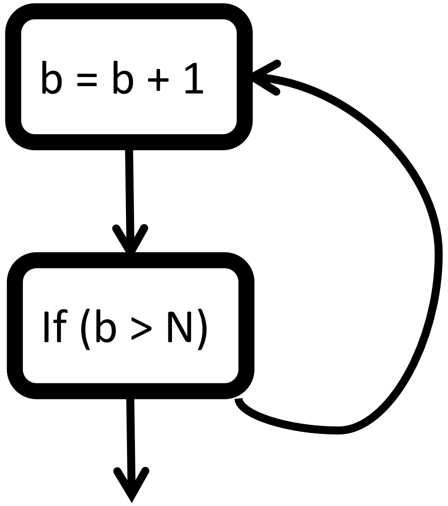
- Concepts needed by most code analyses
- Why do we need DFA? (opportunities)
- Introduction to DFA (concepts)
- A DFA example: reaching definitions (concept application)
- Implementation of DFA (actual implementation)

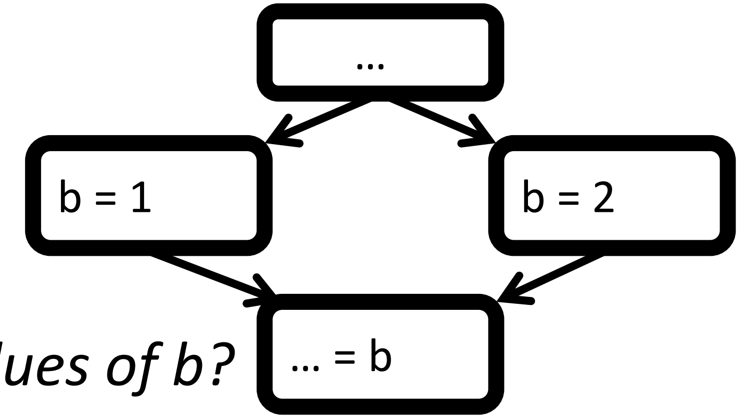
Concepts

- Static and dynamic control flows
- Data flow abstraction
- Data flow values
- Transfer functions
- GEN, KILL, IN, OUT sets

Static program vs. dynamic execution

- **Static**:
  Finite program
- **Dynamic**:
  Can have infinitely many possible control flows
- **Data flow analysis abstraction:**
  For each point in a program:
  combine information about all possible run-time instances
  of the same program point.



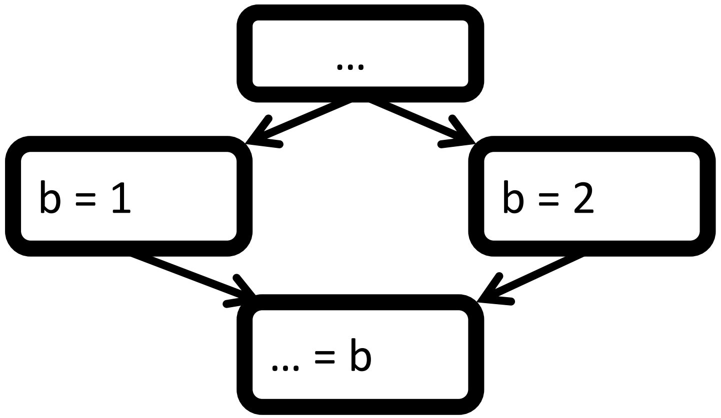


Data flow analysis (DFA):
traverse the CFGs collecting information about
what may happen at run time
(Conservative approximation)

What are the possible values of b?

Example of data-flow questions

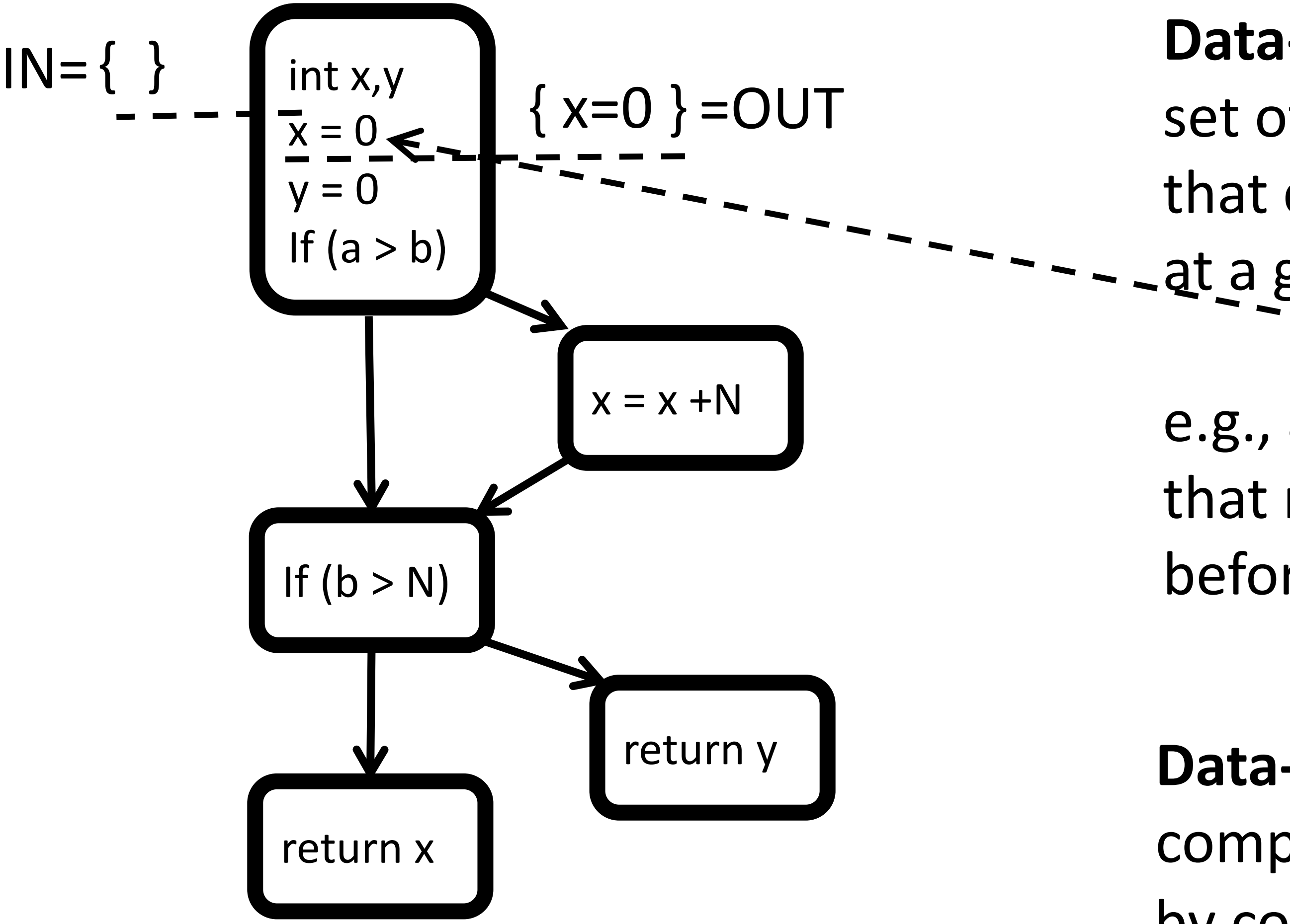
- What are the possible values of b just before an instruction “… = b”?
- Which instruction defines the value used in “… = b”?



Example of data-flow questions

- What are the possible values of b just before an instruction “… = b”?
- Which instruction defines the value used in “… = b”?
- Has the expression “a * b” been computed
  before another instruction? (“… = a * b”)
- What are the instructions that might read the value
  produced by an instruction “b = …”?
- What are the instructions that will (must) read the value
  produced by an instruction “b = …”?
- …

Data-flow expressed in CFG



Data-flow value:
set of all possible program states
that can be observed
at a given program point

e.g., all definitions in the program
that might have been executed
before that point

Data-flow analysis
computes IN and OUT sets
by computing
the DFA-specific transfer functions

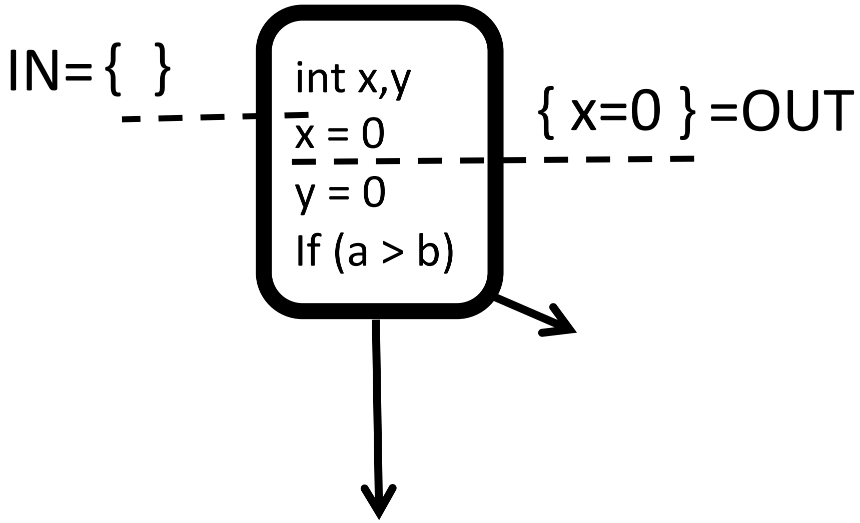
Transfer functions

- Let *i* be an instruction: IN[*i*] and OUT[*i*] are the set of data-flow values before and after the instruction *i* of a program
- A transfer function *fs* relates the data-flow values before and after an instruction *i*
- In a forward data-flow problem

$$\text{OUT}[\, i \,] = \boxed{fs}(\, \text{IN}[\, i \,] \,)$$

- In a backward data-flow problem

$$\text{IN}[\, i \,] = \boxed{fs}(\, \text{OUT}[\, i \,] \,)$$



***fs* is DFA-specific**

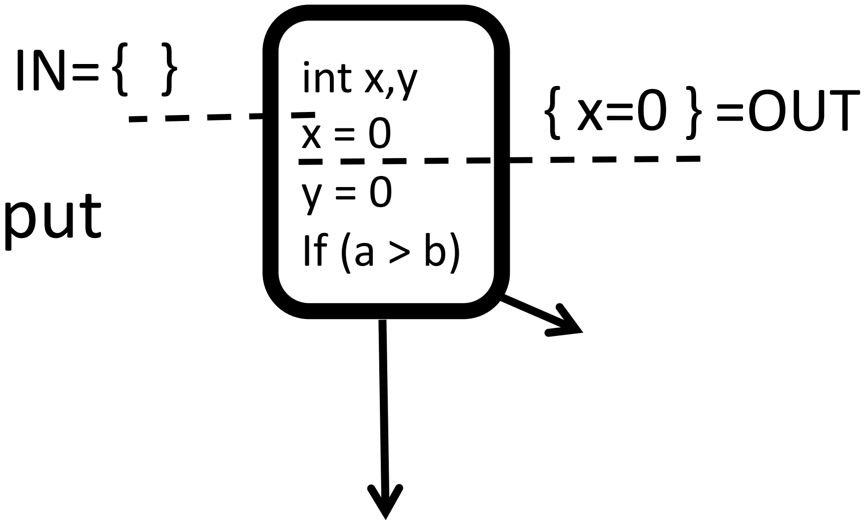
Transfer function internals: Y[*i*] = *fs* (X[*i*])

- It relies on information that reaches i
- It transforms such information to propagate the result to the rest of the CFG

GEN[i] = data flow value added by i

KILL[i] = data flow value removed because of i

- To do so, it relies on information specific to i
  - Encoded in GEN[i], KILL[i]
  - *fs* uses GEN[i] and KILL[i] to compute its output



- GEN[i] and KILL[i] are DFA-specific and (typically) data/control flow independent!

DFA steps

1) Define the DFA-specific sets GEN[i] and KILL[i], for all I
   and without looking at the control flows

2) Implement the DFA-specific transfer function *fs*

3) Compute all IN[i] and OUT[i] <u>following a DFA-generic algorithm</u>
   OUT[i] = *fs* (IN[i])
   IN[i] = *fs* (OUT[i])

   Compilers typically have a data flow framework/engine
   to help developing new DFAs
   (we will not rely on such framework/engine for this class)

Data Flow Analysis outline

- Concepts needed by most code analyses
- Why do we need DFA? (opportunities)
- Introduction to DFA (concepts)
- A DFA example: reaching definitions (concept application)
- Implementation of DFA (actual implementation)

Before introducing the reaching definition DFA,
let us go back to the previous example to formalize new terminology

Optimization example: constant propagation

```
int sumcalc (int a, int b, int N){
  int x,y;
  x = 0;
  y = 0;
  if (a > b){
    x = x + N;
  }
  if (b > N){  return y;}
  return x;
}
```

Information needed just before an instruction *i*:
what are the definitions that ~~might execute before~~ *i*?

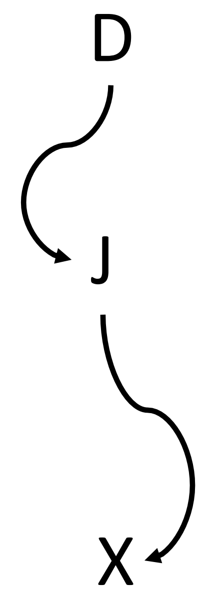
reach

```
if (b > N){  return 0;}
```

IN[return y] = {y=0}
IN[return x] = {x=0, x = x + N}

Let us define the concept of “*reaching*” more formally

Data-flow example: reaching definitions



- A definition *D reaches* a program point *X*
  if there is a control flow from *D* to *X*
  such that ~~the variable defined by~~ *D* is not ~~redefined~~ killed along that path

```
...
D: v = 0
J:  call printf(...)
X: ... = v ...
```

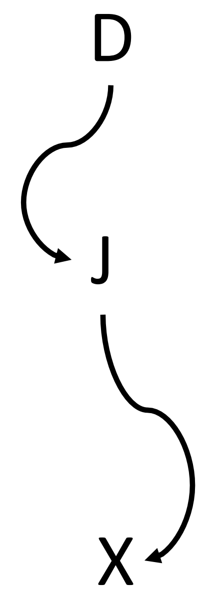
GEN[i] = data flow value added by i

GEN[D] = {D}

D reaches X

IN[X] = {D}

Data-flow example: reaching definitions



- A definition *D reaches* a program point *X* if there is a control flow from *D* to *X* such that *D* is not killed along that path

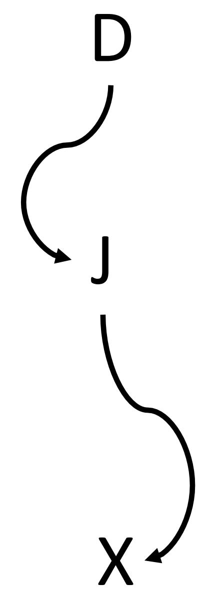
```
...
D: v = 0
J:  call printf(...)
X: ... = v ...
```

GEN[D] = {D}

D reaches X

IN[X] = {D}

Data-flow example: reaching definitions



- A definition *D reaches* a program point *X* if there is a control flow from *D* to *X* such that *D* is not killed along that path

```
...
D: v = 0
...
J : v = v + n
...
X: ... = v ...
```

GEN[i] = data flow value added by i

GEN[D] = {D}   KILL[D] = {J}

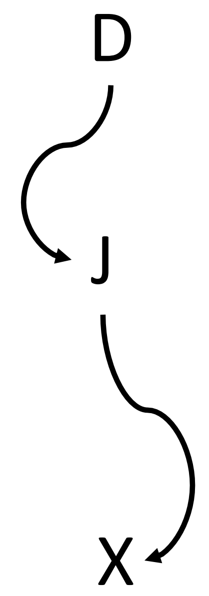
KILL[J] = {D}

KILL[i] = data flow value removed because of i

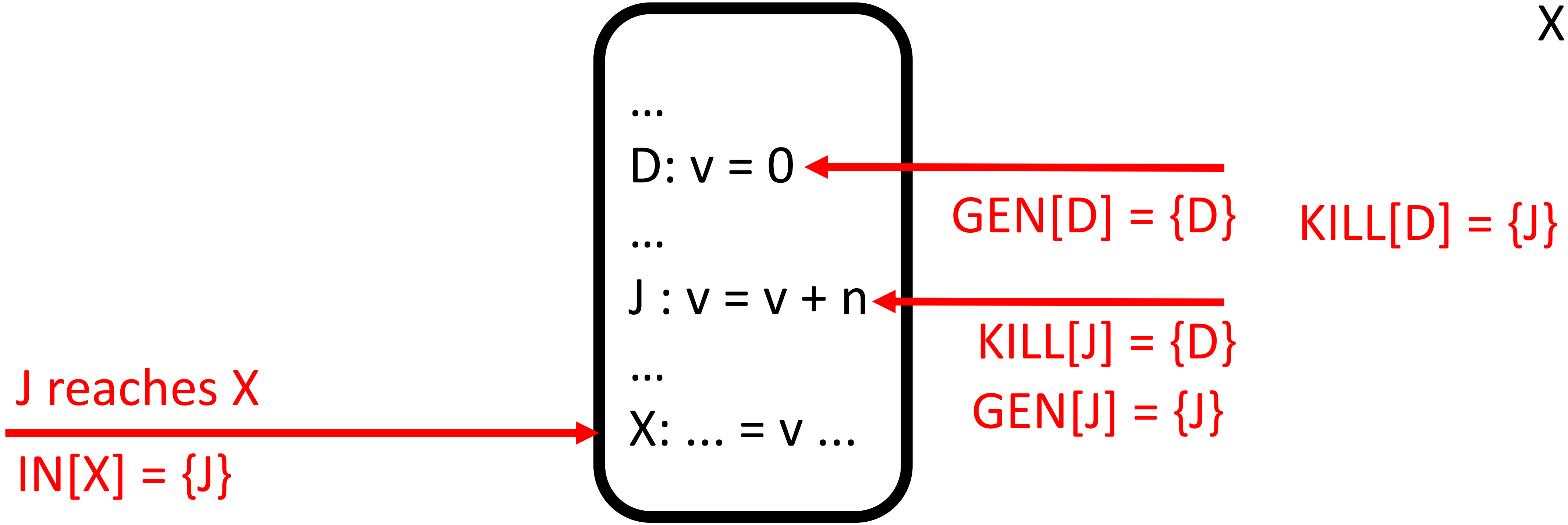
D does not reach X

D is not in IN[X]

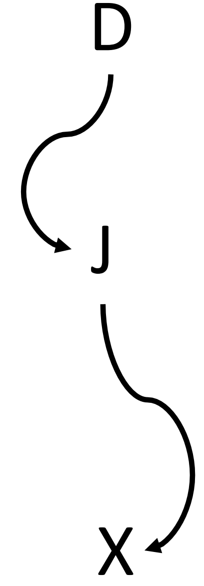
Data-flow example: reaching definitions



- A definition *D reaches* a program point *X* if there is a control flow from *D* to *X* such that *D* is not killed along that path



Data-flow example: reaching definitions



- A definition *D reaches* a program point *X*

  if there is a control flow from *D* to *X*

  such that *D* is not killed along that path

- The reaching definition **data-flow problem** for a flow graph is to compute **all** definitions that reach an instruction i

  (i.e., IN[i], OUT[i])

  for **all** *i* in that graph

Computing INs and OUTs

GEN[0] = { }
GEN[1] = {1}
GEN[2] = {2}
GEN[3] = { }

IN={ }

```
0: int x,y
1: x = 0
2: y = 0
3: return x
```

{ x=0 } =OUT

KILL[0] = { }
KILL[1] = { }
KILL[2] = { }
KILL[3] = { }

Local reaching definitions

- Forward or backward?
  OUT[*i*] = *fs* (IN[i])

- GEN[*i*] = what *i* generates
- KILL[*i*] = what *i* kills (invalidates)

- *fs* within a basic block?
  Let *i* be an instruction and
  p be its only predecessor

IN[*i*] = OUT[*p*]
OUT[*i*] = GEN[*i*] U (IN[*i*] – KILL[*i*])

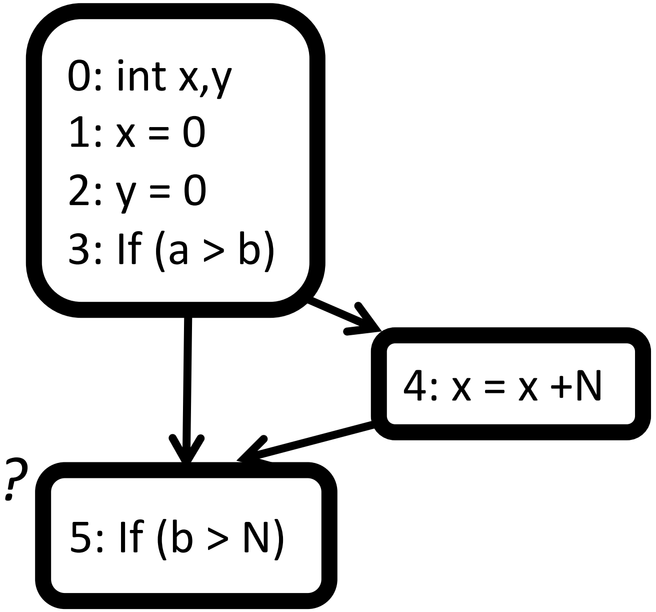
Data-flow example: reaching definitions

- A definition *d reaches* a program point *X* if there is a path from *d* to *X* such that *d* is not killed along that path
- The **data-flow problem** for a flow graph is to compute IN[*i*] and OUT[*i*] for all *i* in that graph

$$\mathrm{IN}[i] = \bigcup_{p \text{ a predecessor of } i} \mathrm{OUT}[p]$$

$$\mathrm{OUT}[i] = \mathrm{GEN}[i] \cup (\mathrm{IN}[i] - \mathrm{KILL}[i])$$

Should 1 be in IN[5]?

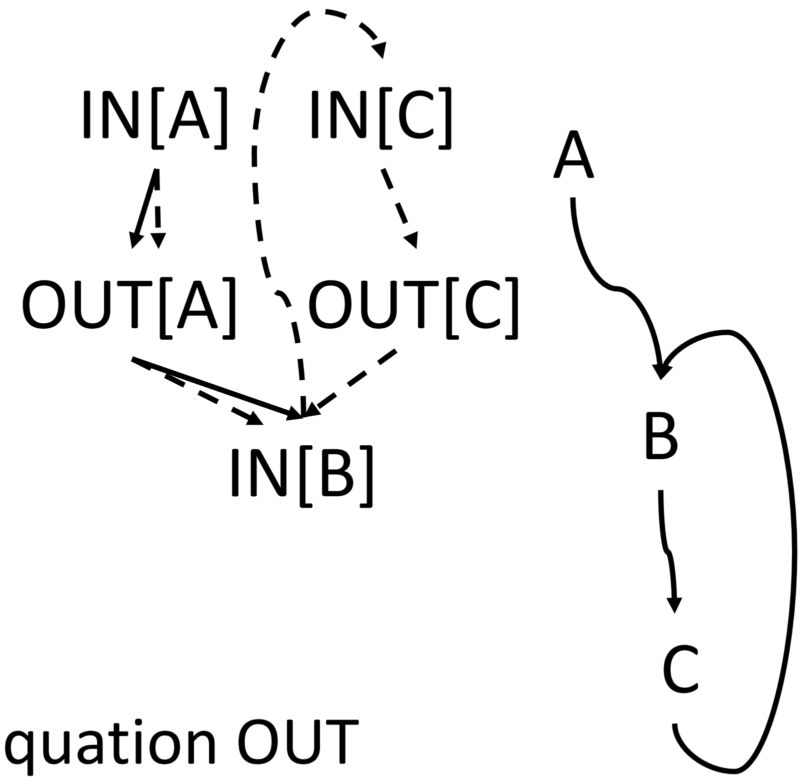


Global reaching definitions

Data Flow Analysis outline

- Concepts needed by most code analyses
- Why do we need DFA? (opportunities)
- Introduction to DFA (concepts)
- A DFA example: reaching definitions (concept application)
- Implementation of DFA (actual implementation)

- So far, we have defined **data-flow equations** (i.e., IN and OUT equations)



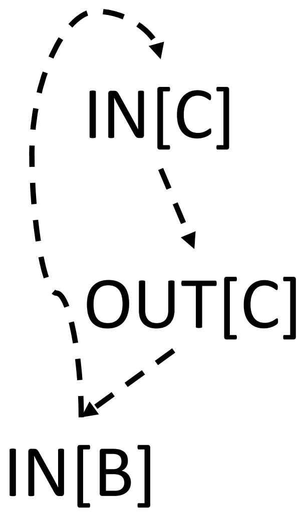
- How can we actually compute them?

- Main problem:
  - ➢input of equation IN depends on output of equation OUT

    $\mathrm{IN}[i] \quad = \bigcup_{p \text{ a predecessor of } i} \mathrm{OUT}[p]$
  - ➢Output of equation OUT depends on input of equation IN

    $\mathrm{OUT}[i] = \mathrm{GEN}[i] \cup (\mathrm{IN}[i] - \mathrm{KILL}[i])$

We break all possible dependence cycles
by iteratively computing
all IN and OUT sets
until a fixed point is reached



Steps for iterative algorithm

- Compute GEN and KILL sets for all instructions without using the CFG
  - GEN and KILL sets will not change anymore

- Compute IN and OUT sets with an iterative algorithm

```
do{
    Compute IN and OUT sets for all instructions
} while (any IN or OUT set changes from the previous iteration)
```

Iterative algorithm for reaching definitions

- Given GEN[i], KILL[i] for all instructions i,
  we compute IN[i] and OUT[i] for all i

```
for (each instruction i)  IN[i] = OUT[i] = { };
```

```
do {
  for (each instruction i) {
    IN[i] = ∪p a predecessor of i OUT[p];
    OUT[i] = GEN[i] ∪ (IN[i] – KILL[i]);
  }
} while (changes to any OUT occur)
```

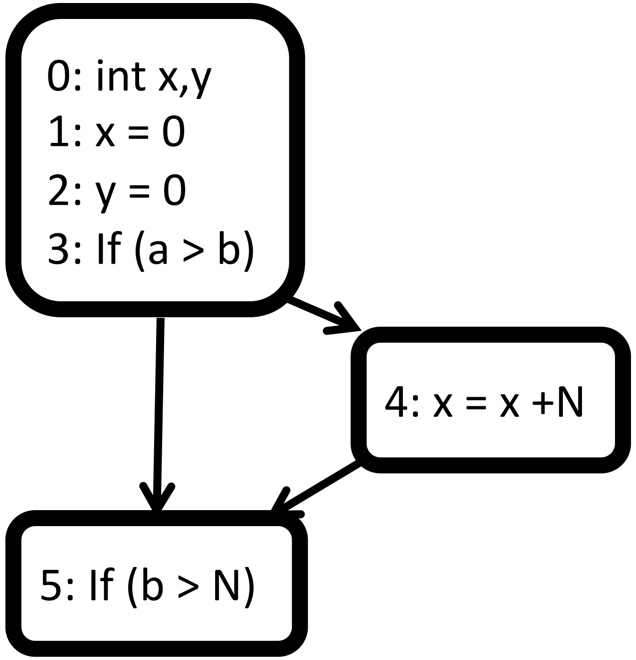
Reaching definition in action

GEN[0] = {}
GEN[1] = {1}
GEN[2] = {2}
GEN[3] = {}
GEN[4] = {4}
GEN[5] = {}

KILL[0] = {}
KILL[1] = {4}
KILL[2] = {}
KILL[3] = {}
KILL[4] = {1}
KILL[5] = {}

Done?

Why do we need to reach a fixed point?



IN[0] = { }
IN[1] = { }
IN[2] = {1 }
IN[3] = {1,2 }
IN[4] = {1,2 }
IN[5] = {1,2,4}

OUT[0] = { }
OUT[1] = {1 }
OUT[2] = {1,2 }
OUT[3] = {1,2 }
OUT[4] = {2,4 }
OUT[5] = {1,2,4}

$$IN[i] = \bigcup_{p \text{ a predecessor}} OUT[p]$$

$$OUT[i] = GEN[i] \cup (IN[i] - KILL[i])$$

Now that you know reaching definition

- It’s time for the homework H1

- What we learned was for forward data-flow analysis

$$OUT[\,s\,] = fs(\,IN[\,s\,]\,)$$

for (each instruction *i*) IN[i] = OUT[i] = { };

do {

  for (each instruction *i*) {

    $IN[i] = \cup_{p \text{ a predecessor of } i} OUT[p]$;

    $OUT[i] = GEN[i] \cup (IN[i] - KILL[i])$;

  }

} while (changes to any OUT occur)

- What about backward data-flow analysis?

$$IN[\,s\,] = fs(\,OUT[\,s\,]\,)$$

Forward DFA

for (each instruction i) IN[i] = OUT[i] = { };

do {

  for (each instruction i) {

    IN[i] = $fs_{p \text{ a predecessor of } i}$ (OUT[p])

    OUT[i] = fs(IN[i])

  }

} while (changes to any OUT occur)

Backward DFA

```
for (each instruction i)  IN[i] = OUT[i] = { };
do {
  for (each instruction i) {
    OUT[i] = fs_{s a successor of i} (IN[s])
    IN[i] = fs(OUT[i])
  }
} while (changes to any IN   occur)
```

Always have faith in your ability

Success will come your way eventually

Best of luck!