

# CS738: Advanced Compiler Optimizations

## Data Flow Analysis

Amey Karkare

karkare@cse.iitk.ac.in

<http://www.cse.iitk.ac.in/~karkare/cs738>

Department of CSE, IIT Kanpur



# Agenda

- ▶ Static analysis and compile-time optimizations

# Agenda

- ▶ Static analysis and compile-time optimizations
- ▶ For the next few lectures

# Agenda

- ▶ Static analysis and compile-time optimizations
- ▶ For the next few lectures
- ▶ *Intraprocedural* Data Flow Analysis

# Agenda

- ▶ Static analysis and compile-time optimizations
- ▶ For the next few lectures
- ▶ *Intraprocedural* Data Flow Analysis
  - ▶ Classical Examples

# Agenda

- ▶ Static analysis and compile-time optimizations
- ▶ For the next few lectures
- ▶ *Intraprocedural* Data Flow Analysis
  - ▶ Classical Examples
  - ▶ Components

# Assumptions

- ▶ Intraprocedural: Restricted to a single function

# Assumptions

- ▶ Intraprocedural: Restricted to a single function
- ▶ Input in 3-address format



# Assumptions

- ▶ Intraprocedural: Restricted to a single function
- ▶ Input in 3-address format
- ▶ Unless otherwise specified

# 3-address Code Format

## ► Assignments

# 3-address Code Format

- ▶ Assignments

$x = y \text{ op } z$

# 3-address Code Format

## ► Assignments

$x = y \text{ op } z$

$x = \text{op } y$

# 3-address Code Format

## ► Assignments

$x = y \text{ op } z$

$x = \text{op } y$

$x = y$

# 3-address Code Format

- ▶ Assignments

$x = y \text{ op } z$

$x = \text{op } y$

$x = y$

- ▶ Jump/control transfer

# 3-address Code Format

- ▶ Assignments

$x = y \text{ op } z$

$x = \text{op } y$

$x = y$

- ▶ Jump/control transfer

goto L

# 3-address Code Format

- ▶ Assignments

$x = y \text{ op } z$

$x = \text{op } y$

$x = y$

- ▶ Jump/control transfer

goto L

if x relop y goto L



# 3-address Code Format

- ▶ Assignments

$x = y \text{ op } z$

$x = \text{op } y$

$x = y$

- ▶ Jump/control transfer

goto L

if x relop y goto L

- ▶ Statements can have label(s)

# 3-address Code Format

- ▶ Assignments

$x = y \text{ op } z$

$x = \text{op } y$

$x = y$

- ▶ Jump/control transfer

goto L

if x relop y goto L

- ▶ Statements can have label(s)

L: ...

# 3-address Code Format

- ▶ Assignments

$x = y \text{ op } z$

$x = \text{op } y$

$x = y$

- ▶ Jump/control transfer

goto L

if x relop y goto L

- ▶ Statements can have label(s)

L: ...

- ▶ Arrays, Pointers and Functions to be added later when needed

# Data Flow Analysis

- ▶ Class of techniques to derive information about flow of data

# Data Flow Analysis

- ▶ Class of techniques to derive information about flow of data
  - ▶ along program execution paths

# Data Flow Analysis

- ▶ Class of techniques to derive information about flow of data
  - ▶ along program execution paths
- ▶ Used to answer questions such as:

# Data Flow Analysis

- ▶ Class of techniques to derive information about flow of data
  - ▶ along program execution paths
- ▶ Used to answer questions such as:
  - ▶ whether two identical expressions evaluate to same value

# Data Flow Analysis

- ▶ Class of techniques to derive information about flow of data
  - ▶ along program execution paths
- ▶ Used to answer questions such as:
  - ▶ whether two identical expressions evaluate to same value
    - ▶ used in common subexpression elimination



# Data Flow Analysis

- ▶ Class of techniques to derive information about flow of data
  - ▶ along program execution paths
- ▶ Used to answer questions such as:
  - ▶ whether two identical expressions evaluate to same value
    - ▶ used in common subexpression elimination
  - ▶ whether the result of an assignment is used later

# Data Flow Analysis

- ▶ Class of techniques to derive information about flow of data
  - ▶ along program execution paths
- ▶ Used to answer questions such as:
  - ▶ whether two identical expressions evaluate to same value
    - ▶ used in common subexpression elimination
  - ▶ whether the result of an assignment is used later
    - ▶ used by dead code elimination

# Data Flow Abstraction

- ▶ Basic Blocks (BB)

# Data Flow Abstraction

- ▶ Basic Blocks (BB)
  - ▶ sequence of 3-address code stmts

# Data Flow Abstraction

- ▶ Basic Blocks (BB)
  - ▶ sequence of 3-address code stmts
  - ▶ single entry at the first statement

# Data Flow Abstraction

- ▶ Basic Blocks (BB)
  - ▶ sequence of 3-address code stmts
  - ▶ single entry at the first statement
  - ▶ single exit at the last statement

# Data Flow Abstraction

- ▶ Basic Blocks (BB)
  - ▶ sequence of 3-address code stmts
  - ▶ single entry at the first statement
  - ▶ single exit at the last statement
  - ▶ Typically we use “maximal” basic block (maximal sequence of such instructions)

# Identifying Basic Blocks

- ▶ *Leader*. The first statement of a basic block



# Identifying Basic Blocks

- ▶ *Leader*. The first statement of a basic block
  - ▶ The first instruction of the program (procedure)

# Identifying Basic Blocks

- ▶ *Leader*. The first statement of a basic block
  - ▶ The first instruction of the program (procedure)
  - ▶ Target of a branch (conditional and unconditional goto)

# Identifying Basic Blocks

- ▶ *Leader*. The first statement of a basic block
  - ▶ The first instruction of the program (procedure)
  - ▶ Target of a branch (conditional and unconditional goto)
  - ▶ Instruction immediately following a branch

# Special Basic Blocks

- ▶ Two special BBs are added to simplify the analysis

# Special Basic Blocks

- ▶ Two special BBs are added to simplify the analysis
  - ▶ empty (?) blocks!

# Special Basic Blocks

- ▶ Two special BBs are added to simplify the analysis
  - ▶ empty (?) blocks!
- ▶ *Entry*: The first block to be executed for the procedure analyzed

# Special Basic Blocks

- ▶ Two special BBs are added to simplify the analysis
  - ▶ empty (?) blocks!
- ▶ *Entry*: The first block to be executed for the procedure analyzed
- ▶ *Exit*: The last block to be executed

# Data Flow Abstraction

- ▶ Control Flow Graph (CFG)



# Data Flow Abstraction

- ▶ Control Flow Graph (CFG)
- ▶ A rooted directed graph  $G = (N, E)$

# Data Flow Abstraction

- ▶ Control Flow Graph (CFG)
- ▶ A rooted directed graph  $G = (N, E)$
- ▶  $N$  = set of BBs

# Data Flow Abstraction

- ▶ Control Flow Graph (CFG)
- ▶ A rooted directed graph  $G = (N, E)$
- ▶  $N$  = set of BBs
  - ▶ including *Entry*, *Exit*

# Data Flow Abstraction

- ▶ Control Flow Graph (CFG)
- ▶ A rooted directed graph  $G = (N, E)$
- ▶  $N$  = set of BBs
  - ▶ including *Entry*, *Exit*
- ▶  $E$  = set of edges

# CFG Edges

- ▶ Edge  $B_1 \rightarrow B_2 \in E$  if control can transfer from  $B_1$  to  $B_2$

# CFG Edges

- ▶ Edge  $B_1 \rightarrow B_2 \in E$  if control can transfer from  $B_1$  to  $B_2$ 
  - ▶ Fall through

# CFG Edges

- ▶ Edge  $B_1 \rightarrow B_2 \in E$  if control can transfer from  $B_1$  to  $B_2$ 
  - ▶ Fall through
  - ▶ Through jump (goto)

# CFG Edges

- ▶ Edge  $B_1 \rightarrow B_2 \in E$  if control can transfer from  $B_1$  to  $B_2$ 
  - ▶ Fall through
  - ▶ Through jump (goto)
  - ▶ Edge from *Entry* to (all?) real first BB(s)



# CFG Edges

- ▶ Edge  $B_1 \rightarrow B_2 \in E$  if control can transfer from  $B_1$  to  $B_2$ 
  - ▶ Fall through
  - ▶ Through jump (goto)
  - ▶ Edge from *Entry* to (all?) real first BB(s)
  - ▶ Edge to *Exit* from all last BBs

# CFG Edges

- ▶ Edge  $B_1 \rightarrow B_2 \in E$  if control can transfer from  $B_1$  to  $B_2$ 
  - ▶ Fall through
  - ▶ Through jump (goto)
  - ▶ Edge from *Entry* to (all?) real first BB(s)
  - ▶ Edge to *Exit* from all last BBs
    - ▶ BBs containing return

# CFG Edges

- ▶ Edge  $B_1 \rightarrow B_2 \in E$  if control can transfer from  $B_1$  to  $B_2$ 
  - ▶ Fall through
  - ▶ Through jump (goto)
  - ▶ Edge from *Entry* to (all?) real first BB(s)
  - ▶ Edge to *Exit* from all last BBs
    - ▶ BBs containing return
    - ▶ Last real BB

# Data Flow Abstraction: Control Flow Graph

- ▶ Graph representation of paths that program may exercise during execution

# Data Flow Abstraction: Control Flow Graph

- ▶ Graph representation of paths that program may exercise during execution
- ▶ Typically one graph per procedure

# Data Flow Abstraction: Control Flow Graph

- ▶ Graph representation of paths that program may exercise during execution
- ▶ Typically one graph per procedure
- ▶ Graphs for separate procedures have to be combined/connected for interprocedural analysis

# Data Flow Abstraction: Control Flow Graph

- ▶ Graph representation of paths that program may exercise during execution
- ▶ Typically one graph per procedure
- ▶ Graphs for separate procedures have to be combined/connected for interprocedural analysis
  - ▶ Later!

# Data Flow Abstraction: Control Flow Graph

- ▶ Graph representation of paths that program may exercise during execution
- ▶ Typically one graph per procedure
- ▶ Graphs for separate procedures have to be combined/connected for interprocedural analysis
  - ▶ Later!
  - ▶ Single procedure, single flow graph for now.



# Data Flow Abstraction: Program Points

- ▶ Input state/Output state for Stmt

# Data Flow Abstraction: Program Points

- ▶ Input state/Output state for Stmt
  - ▶ Program point before/after a stmt

# Data Flow Abstraction: Program Points

- ▶ Input state/Output state for Stmt
  - ▶ Program point before/after a stmt
  - ▶ Denoted  $IN[s]$  and  $OUT[s]$

# Data Flow Abstraction: Program Points

- ▶ Input state/Output state for Stmt
  - ▶ Program point before/after a stmt
  - ▶ Denoted  $IN[s]$  and  $OUT[s]$
  - ▶ Within a basic block:

# Data Flow Abstraction: Program Points

- ▶ Input state/Output state for Stmt
  - ▶ Program point before/after a stmt
  - ▶ Denoted  $IN[s]$  and  $OUT[s]$
  - ▶ Within a basic block:
    - ▶ Program point after a stmt is same as the program point before the next stmt

# Data Flow Abstraction: Program Points

- ▶ Input state/Output state for BBs

# Data Flow Abstraction: Program Points

- ▶ Input state/Output state for BBs
  - ▶ Program point before/after a bb

# Data Flow Abstraction: Program Points

- ▶ Input state/Output state for BBs
  - ▶ Program point before/after a bb
  - ▶ Denoted  $IN[B]$  and  $OUT[B]$



# Data Flow Abstraction: Program Points

- ▶ Input state/Output state for BBs
  - ▶ Program point before/after a bb
  - ▶ Denoted  $IN[B]$  and  $OUT[B]$
  - ▶ For  $B_1$  and  $B_2$ :

# Data Flow Abstraction: Program Points

- ▶ Input state/Output state for BBs
  - ▶ Program point before/after a bb
  - ▶ Denoted  $IN[B]$  and  $OUT[B]$
  - ▶ For  $B_1$  and  $B_2$ :
    - ▶ if there is an edge from  $B_1$  to  $B_2$  in CFG, then the program point *after* the last stmt of  $B_1$  *may be* followed immediately by the program point *before* the first stmt of  $B_2$ .

# Data Flow Abstraction: Execution Paths

- ▶ An execution path is of the form

$$p_1, p_2, p_3, \dots, p_n$$

# Data Flow Abstraction: Execution Paths

- ▶ An execution path is of the form

$$p_1, p_2, p_3, \dots, p_n$$

where  $p_i \rightarrow p_{i+1}$  are adjacent program points in the CFG.

# Data Flow Abstraction: Execution Paths

- ▶ An execution path is of the form

$$p_1, p_2, p_3, \dots, p_n$$

where  $p_i \rightarrow p_{i+1}$  are adjacent program points in the CFG.

- ▶ Infinite number of possible execution paths in practical programs.

# Data Flow Abstraction: Execution Paths

- ▶ An execution path is of the form

$$p_1, p_2, p_3, \dots, p_n$$

where  $p_i \rightarrow p_{i+1}$  are adjacent program points in the CFG.

- ▶ Infinite number of possible execution paths in practical programs.
- ▶ Paths having no finite upper bound on the length.

# Data Flow Abstraction: Execution Paths

- ▶ An execution path is of the form

$$p_1, p_2, p_3, \dots, p_n$$

where  $p_i \rightarrow p_{i+1}$  are adjacent program points in the CFG.

- ▶ Infinite number of possible execution paths in practical programs.
- ▶ Paths having no finite upper bound on the length.
- ▶ Need to *summarize* the information at a program point with a finite set of facts.

# Data Flow Schema

- ▶ Data flow values associated with each program point



# Data Flow Schema

- ▶ Data flow values associated with each program point
  - ▶ Summarize all possible states at that point

# Data Flow Schema

- ▶ Data flow values associated with each program point
  - ▶ Summarize all possible states at that point
- ▶ *Domain*: set of all possible data flow values

# Data Flow Schema

- ▶ Data flow values associated with each program point
  - ▶ Summarize all possible states at that point
- ▶ *Domain*: set of all possible data flow values
- ▶ Different domains for different analyses/optimizations

# Data Flow Problem

- ▶ Constraints on data flow values

# Data Flow Problem

- ▶ Constraints on data flow values
  - ▶ Transfer constraints

# Data Flow Problem

- ▶ Constraints on data flow values
  - ▶ Transfer constraints
  - ▶ Control flow constraints

# Data Flow Problem

- ▶ Constraints on data flow values
  - ▶ Transfer constraints
  - ▶ Control flow constraints
- ▶ **Aim:** To find a solution to the constraints

# Data Flow Problem

- ▶ Constraints on data flow values
  - ▶ Transfer constraints
  - ▶ Control flow constraints
- ▶ **Aim:** To find a solution to the constraints
  - ▶ Multiple solutions possible



# Data Flow Problem

- ▶ Constraints on data flow values
  - ▶ Transfer constraints
  - ▶ Control flow constraints
- ▶ **Aim:** To find a solution to the constraints
  - ▶ Multiple solutions possible
  - ▶ Trivial solutions, . . . , Exact solutions

# Data Flow Problem

- ▶ Constraints on data flow values
  - ▶ Transfer constraints
  - ▶ Control flow constraints
- ▶ **Aim:** To find a solution to the constraints
  - ▶ Multiple solutions possible
  - ▶ Trivial solutions, . . . , Exact solutions
- ▶ We typically compute approximate solution

# Data Flow Problem

- ▶ Constraints on data flow values
  - ▶ Transfer constraints
  - ▶ Control flow constraints
- ▶ **Aim:** To find a solution to the constraints
  - ▶ Multiple solutions possible
  - ▶ Trivial solutions, . . . , Exact solutions
- ▶ We typically compute approximate solution
  - ▶ Close to the exact solution (as close as possible!)

# Data Flow Problem

- ▶ Constraints on data flow values
  - ▶ Transfer constraints
  - ▶ Control flow constraints
- ▶ **Aim:** To find a solution to the constraints
  - ▶ Multiple solutions possible
  - ▶ Trivial solutions, . . . , Exact solutions
- ▶ We typically compute approximate solution
  - ▶ Close to the exact solution (as close as possible!)
  - ▶ Why not exact solution?

# Data Flow Constraints: Transfer Constraints

- ▶ Transfer functions

# Data Flow Constraints: Transfer Constraints

- ▶ Transfer functions
  - ▶ relationship between the data flow values before and after a stmt

# Data Flow Constraints: Transfer Constraints

- ▶ Transfer functions
  - ▶ relationship between the data flow values before and after a stmt
- ▶ forward functions: Compute facts *after* a statement *s* from the facts available *before s*.

# Data Flow Constraints: Transfer Constraints

- ▶ Transfer functions
  - ▶ relationship between the data flow values before and after a stmt
- ▶ forward functions: Compute facts *after* a statement  $s$  from the facts available *before*  $s$ .
  - ▶ General form:

$$\text{OUT}[s] = f_s(\text{IN}[s])$$



# Data Flow Constraints: Transfer Constraints

- ▶ Transfer functions
  - ▶ relationship between the data flow values before and after a stmt
- ▶ forward functions: Compute facts *after* a statement  $s$  from the facts available *before*  $s$ .
  - ▶ General form:

$$\text{OUT}[s] = f_s(\text{IN}[s])$$

- ▶ backward functions: Compute facts *before* a statement  $s$  from the facts available *after*  $s$ .

# Data Flow Constraints: Transfer Constraints

- ▶ Transfer functions
  - ▶ relationship between the data flow values before and after a stmt
- ▶ forward functions: Compute facts *after* a statement  $s$  from the facts available *before*  $s$ .
  - ▶ General form:

$$\text{OUT}[s] = f_s(\text{IN}[s])$$

- ▶ backward functions: Compute facts *before* a statement  $s$  from the facts available *after*  $s$ .
  - ▶ General form:

$$\text{IN}[s] = f_s(\text{OUT}[s])$$

# Data Flow Constraints: Transfer Constraints

- ▶ Transfer functions
  - ▶ relationship between the data flow values before and after a stmt
- ▶ forward functions: Compute facts *after* a statement  $s$  from the facts available *before*  $s$ .
  - ▶ General form:

$$\text{OUT}[s] = f_s(\text{IN}[s])$$

- ▶ backward functions: Compute facts *before* a statement  $s$  from the facts available *after*  $s$ .
  - ▶ General form:

$$\text{IN}[s] = f_s(\text{OUT}[s])$$

- ▶  $f_s$  depends on the statement and the analysis

# Data Flow Constraints: Control Flow Constraints

- ▶ Relationship between the data flow values of two points that are related by program execution semantics

# Data Flow Constraints: Control Flow Constraints

- ▶ Relationship between the data flow values of two points that are related by program execution semantics
- ▶ For a basic block having  $n$  statements:

$$\text{IN}[s_{i+1}] = \text{OUT}[s_i], i = 1, 2, \dots, n - 1$$

# Data Flow Constraints: Control Flow Constraints

- ▶ Relationship between the data flow values of two points that are related by program execution semantics
- ▶ For a basic block having  $n$  statements:

$$\text{IN}[s_{i+1}] = \text{OUT}[s_i], i = 1, 2, \dots, n - 1$$

- ▶  $\text{IN}[s_1]$ ,  $\text{OUT}[s_n]$  to come later

# Data Flow Constraints: Notations

- ▶  $\text{PRED}(B)$ : Set of predecessor BBs of block  $B$  in CFG

# Data Flow Constraints: Notations

- ▶  $\text{PRED}(B)$ : Set of predecessor BBs of block  $B$  in CFG
- ▶  $\text{SUCC}(B)$ : Set of successor BBs of block  $B$  in CFG



# Data Flow Constraints: Notations

- ▶  $\text{PRED}(B)$ : Set of predecessor BBs of block  $B$  in CFG
- ▶  $\text{SUCC}(B)$ : Set of successor BBs of block  $B$  in CFG
- ▶  $f \circ g$ : Composition of functions  $f$  and  $g$

# Data Flow Constraints: Notations

- ▶  $\text{PRED}(B)$ : Set of predecessor BBs of block  $B$  in CFG
- ▶  $\text{SUCC}(B)$ : Set of successor BBs of block  $B$  in CFG
- ▶  $f \circ g$ : Composition of functions  $f$  and  $g$
- ▶  $\oplus$ : An abstract operator denoting some way of combining facts present in a set .

# Data Flow Constraints: Basic Blocks

## ► **Forward**

# Data Flow Constraints: Basic Blocks

## ► Forward

- For  $B$  consisting of  $s_1, s_2, \dots, s_n$

$$f_B = f_{s_n} \circ \dots \circ f_{s_2} \circ f_{s_1}$$

$$\text{OUT}[B] = f_B(\text{IN}[B])$$

# Data Flow Constraints: Basic Blocks

## ► Forward

- For  $B$  consisting of  $s_1, s_2, \dots, s_n$

$$f_B = f_{s_n} \circ \dots \circ f_{s_2} \circ f_{s_1}$$

$$\text{OUT}[B] = f_B(\text{IN}[B])$$

- Control flow constraints

$$\text{IN}[B] = \bigoplus_{P \in \text{PRED}(B)} \text{OUT}[P]$$

# Data Flow Constraints: Basic Blocks

## ► Forward

- For  $B$  consisting of  $s_1, s_2, \dots, s_n$

$$f_B = f_{s_n} \circ \dots \circ f_{s_2} \circ f_{s_1}$$

$$\text{OUT}[B] = f_B(\text{IN}[B])$$

- Control flow constraints

$$\text{IN}[B] = \bigoplus_{P \in \text{PRED}(B)} \text{OUT}[P]$$

## ► Backward

$$f_B = f_{s_1} \circ f_{s_2} \circ \dots \circ f_{s_n}$$

$$\text{IN}[B] = f_B(\text{OUT}[B])$$

$$\text{OUT}[B] = \bigoplus_{S \in \text{SUCC}(B)} \text{IN}[S]$$

# Data Flow Equations

- ▶ Typical Equation

$$\text{OUT}[s] = \text{IN}[s] - \textit{kill}[s] \cup \textit{gen}[s]$$

# Data Flow Equations

- ▶ Typical Equation

$$\text{OUT}[s] = \text{IN}[s] - \textit{kill}[s] \cup \textit{gen}[s]$$

$\textit{gen}(s)$ : information generated



# Data Flow Equations

- ▶ Typical Equation

$$\text{OUT}[s] = \text{IN}[s] - \text{kill}[s] \cup \text{gen}[s]$$

*gen*(*s*): information generated

*kill*(*s*): information killed

# Data Flow Equations

## ► Typical Equation

$$\text{OUT}[s] = \text{IN}[s] - \text{kill}[s] \cup \text{gen}[s]$$

*gen*(*s*): information generated

*kill*(*s*): information killed

## ► Example:

```
a = b*c // generates expression b * c
c = 5    // kills expression b*c
d = b*c  // is b*c redundant here?
```

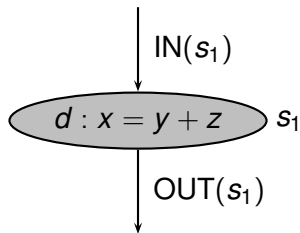
# Example Data Flow Analysis

- ▶ Reaching Definitions Analysis
- ▶ Definition of a variable  $x$ :  $x = \dots \text{something} \dots$
- ▶ Could be more complex (e.g. through pointers, references, implicit)

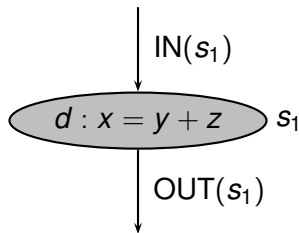
# Reaching Definitions Analysis

- ▶ A definition  $d$  reaches a point  $p$  if
  - ▶ there is a path from the point *immediately following*  $d$  to  $p$
  - ▶  $d$  is not “killed” along that path
  - ▶ “Kill” means redefinition of the left hand side ( $x$  in the earlier example)

# RD Analysis of a Structured Program

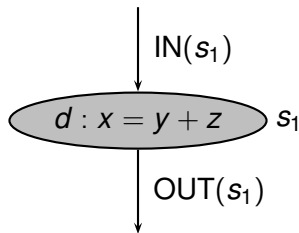


# RD Analysis of a Structured Program



$$OUT(s_1) = IN(s_1) - KILL(s_1) \cup GEN(s_1)$$

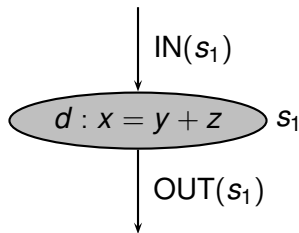
# RD Analysis of a Structured Program



$$OUT(s_1) = IN(s_1) - KILL(s_1) \cup GEN(s_1)$$

$$GEN(s_1) =$$

# RD Analysis of a Structured Program

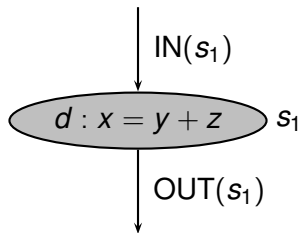


$$OUT(s_1) = IN(s_1) - KILL(s_1) \cup GEN(s_1)$$

$$GEN(s_1) = \{d\}$$



# RD Analysis of a Structured Program

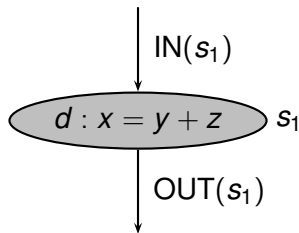


$$OUT(s_1) = IN(s_1) - KILL(s_1) \cup GEN(s_1)$$

$$GEN(s_1) = \{d\}$$

$$KILL(s_1) =$$

# RD Analysis of a Structured Program

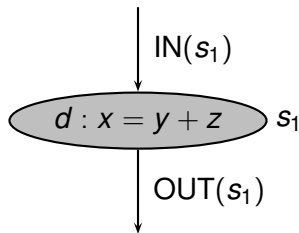


$$OUT(s_1) = IN(s_1) - KILL(s_1) \cup GEN(s_1)$$

$$GEN(s_1) = \{d\}$$

$$KILL(s_1) = D_x - \{d\}, \text{ where } D_x: \text{ set of all definitions of } x$$

# RD Analysis of a Structured Program



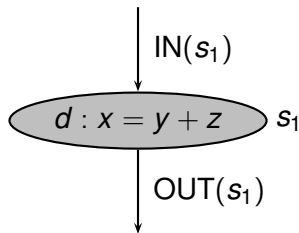
$$OUT(s_1) = IN(s_1) - KILL(s_1) \cup GEN(s_1)$$

$$GEN(s_1) = \{d\}$$

$$KILL(s_1) = D_x - \{d\}, \text{ where } D_x: \text{ set of all definitions of } x$$

$$KILL(s_1) =$$

# RD Analysis of a Structured Program



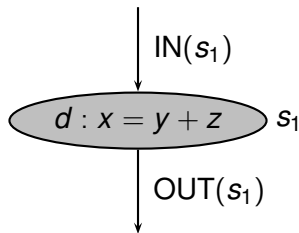
$$OUT(s_1) = IN(s_1) - KILL(s_1) \cup GEN(s_1)$$

$$GEN(s_1) = \{d\}$$

$$KILL(s_1) = D_x - \{d\}, \text{ where } D_x: \text{ set of all definitions of } x$$

$$KILL(s_1) = D_x?$$

# RD Analysis of a Structured Program



$$OUT(s_1) = IN(s_1) - KILL(s_1) \cup GEN(s_1)$$

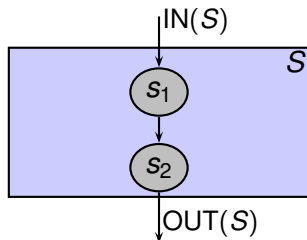
$$GEN(s_1) = \{d\}$$

$$KILL(s_1) = D_x - \{d\}, \text{ where } D_x: \text{ set of all definitions of } x$$

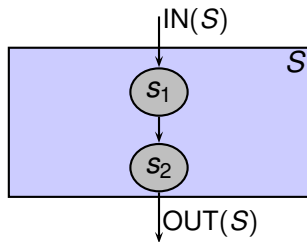
$$KILL(s_1) = D_x? \text{ will also work here}$$

but may not work in general

# RD Analysis of a Structured Program

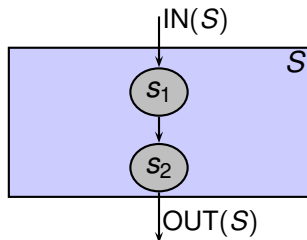


# RD Analysis of a Structured Program



$GEN(S) =$

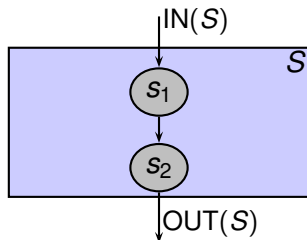
# RD Analysis of a Structured Program



$$GEN(S) = GEN(s_1) - KILL(s_2) \cup GEN(s_2)$$



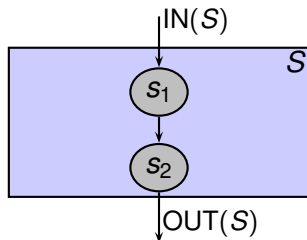
# RD Analysis of a Structured Program



$$GEN(S) = GEN(s_1) - KILL(s_2) \cup GEN(s_2)$$

$$KILL(S) =$$

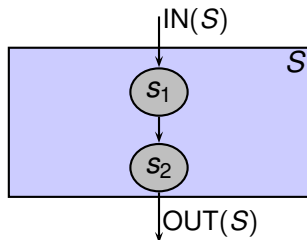
# RD Analysis of a Structured Program



$$GEN(S) = GEN(s_1) - KILL(s_2) \cup GEN(s_2)$$

$$KILL(S) = KILL(s_1) - GEN(s_2) \cup KILL(s_2)$$

# RD Analysis of a Structured Program

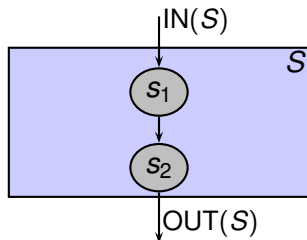


$$GEN(S) = GEN(s_1) - KILL(s_2) \cup GEN(s_2)$$

$$KILL(S) = KILL(s_1) - GEN(s_2) \cup KILL(s_2)$$

$$IN(s_1) =$$

# RD Analysis of a Structured Program

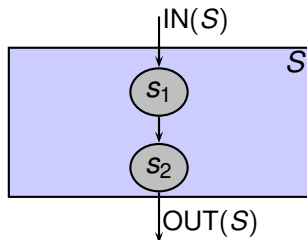


$$GEN(S) = GEN(s_1) - KILL(s_2) \cup GEN(s_2)$$

$$KILL(S) = KILL(s_1) - GEN(s_2) \cup KILL(s_2)$$

$$IN(s_1) = IN(S)$$

# RD Analysis of a Structured Program



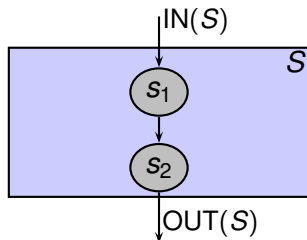
$$GEN(S) = GEN(s_1) - KILL(s_2) \cup GEN(s_2)$$

$$KILL(S) = KILL(s_1) - GEN(s_2) \cup KILL(s_2)$$

$$IN(s_1) = IN(S)$$

$$IN(s_2) =$$

# RD Analysis of a Structured Program



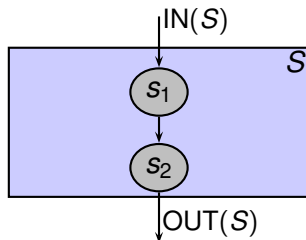
$$GEN(S) = GEN(s_1) - KILL(s_2) \cup GEN(s_2)$$

$$KILL(S) = KILL(s_1) - GEN(s_2) \cup KILL(s_2)$$

$$IN(s_1) = IN(S)$$

$$IN(s_2) = OUT(s_1)$$

# RD Analysis of a Structured Program



$$GEN(S) = GEN(s_1) - KILL(s_2) \cup GEN(s_2)$$

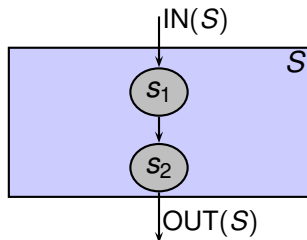
$$KILL(S) = KILL(s_1) - GEN(s_2) \cup KILL(s_2)$$

$$IN(s_1) = IN(S)$$

$$IN(s_2) = OUT(s_1)$$

$$OUT(S) =$$

# RD Analysis of a Structured Program



$$GEN(S) = GEN(s_1) - KILL(s_2) \cup GEN(s_2)$$

$$KILL(S) = KILL(s_1) - GEN(s_2) \cup KILL(s_2)$$

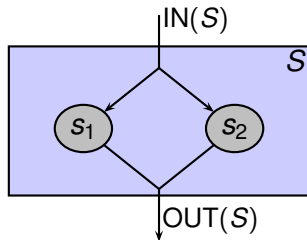
$$IN(s_1) = IN(S)$$

$$IN(s_2) = OUT(s_1)$$

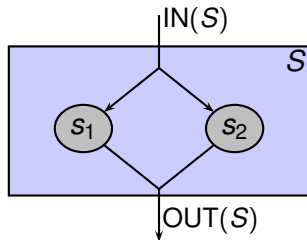
$$OUT(S) = OUT(s_2)$$



# RD Analysis of a Structured Program

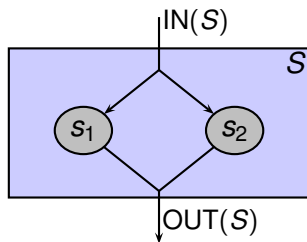


# RD Analysis of a Structured Program



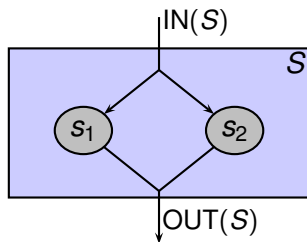
$GEN(S) =$

# RD Analysis of a Structured Program



$$GEN(S) = GEN(s_1) \cup GEN(s_2)$$

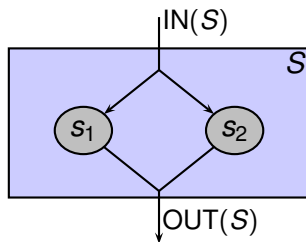
# RD Analysis of a Structured Program



$$GEN(S) = GEN(s_1) \cup GEN(s_2)$$

$$KILL(S) =$$

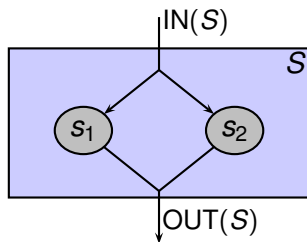
# RD Analysis of a Structured Program



$$GEN(S) = GEN(s_1) \cup GEN(s_2)$$

$$KILL(S) = KILL(s_1) \cap KILL(s_2)$$

# RD Analysis of a Structured Program

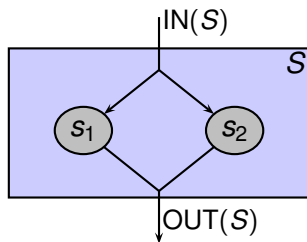


$$GEN(S) = GEN(s_1) \cup GEN(s_2)$$

$$KILL(S) = KILL(s_1) \cap KILL(s_2)$$

$$IN(s_1) =$$

# RD Analysis of a Structured Program

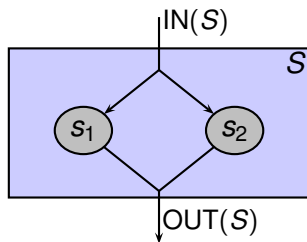


$$GEN(S) = GEN(s_1) \cup GEN(s_2)$$

$$KILL(S) = KILL(s_1) \cap KILL(s_2)$$

$$IN(s_1) = IN(s_2) = IN(S)$$

# RD Analysis of a Structured Program



$$GEN(S) = GEN(s_1) \cup GEN(s_2)$$

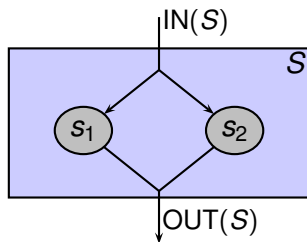
$$KILL(S) = KILL(s_1) \cap KILL(s_2)$$

$$IN(s_1) = IN(s_2) = IN(S)$$

$$OUT(S) =$$



# RD Analysis of a Structured Program



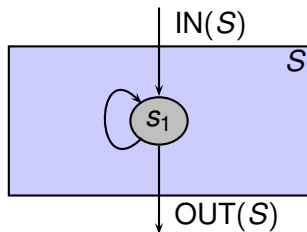
$$GEN(S) = GEN(s_1) \cup GEN(s_2)$$

$$KILL(S) = KILL(s_1) \cap KILL(s_2)$$

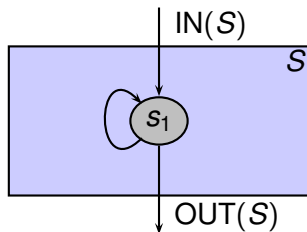
$$IN(s_1) = IN(s_2) = IN(S)$$

$$OUT(S) = OUT(s_1) \cup OUT(s_2)$$

# RD Analysis of a Structured Program

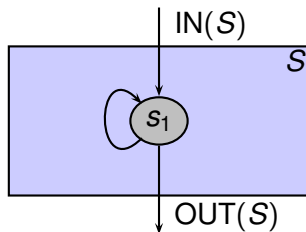


# RD Analysis of a Structured Program



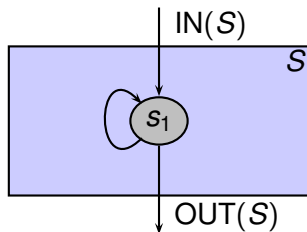
$GEN(S) =$

# RD Analysis of a Structured Program



$$GEN(S) = GEN(s_1)$$

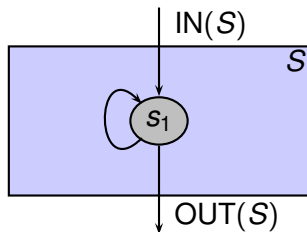
# RD Analysis of a Structured Program



$$GEN(S) = GEN(s_1)$$

$$KILL(S) =$$

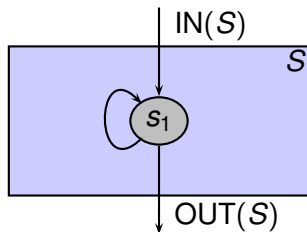
# RD Analysis of a Structured Program



$$GEN(S) = GEN(s_1)$$

$$KILL(S) = KILL(s_1)$$

# RD Analysis of a Structured Program

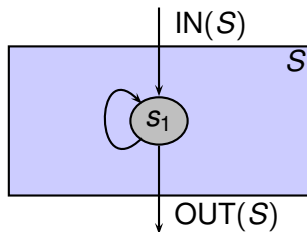


$$GEN(S) = GEN(s_1)$$

$$KILL(S) = KILL(s_1)$$

$$OUT(S) =$$

# RD Analysis of a Structured Program



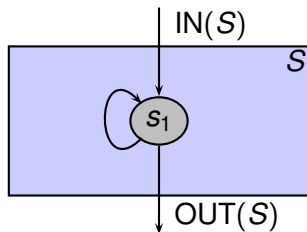
$$GEN(S) = GEN(s_1)$$

$$KILL(S) = KILL(s_1)$$

$$OUT(S) = OUT(s_1)$$



# RD Analysis of a Structured Program



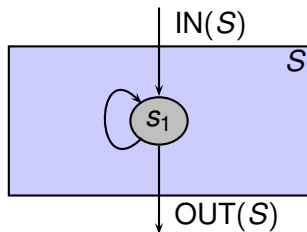
$$GEN(S) = GEN(s_1)$$

$$KILL(S) = KILL(s_1)$$

$$OUT(S) = OUT(s_1)$$

$$IN(s_1) =$$

# RD Analysis of a Structured Program



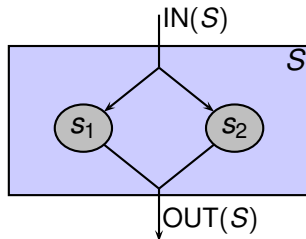
$$GEN(S) = GEN(s_1)$$

$$KILL(S) = KILL(s_1)$$

$$OUT(S) = OUT(s_1)$$

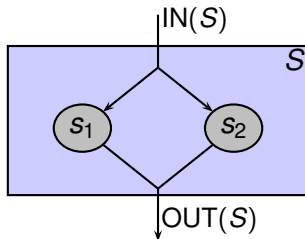
$$IN(s_1) = IN(S) \cup GEN(s_1)$$

# RD Analysis is Approximate



- Assumption: All paths are feasible.

# RD Analysis is Approximate

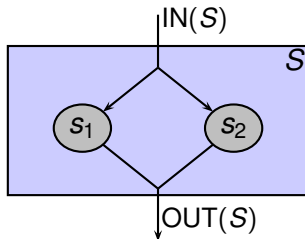


► Assumption: All paths are feasible.

► Example:

```
if (true) s1;  
else      s2;
```

# RD Analysis is Approximate

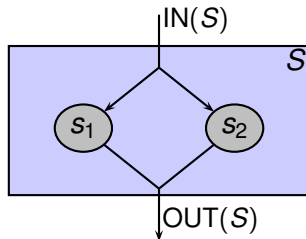


► Assumption: All paths are feasible.

► Example:

```
if (true) s1;  
else      s2;
```

# RD Analysis is Approximate



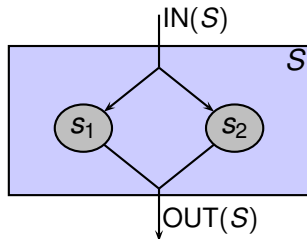
► Assumption: All paths are feasible.

► Example:

```
if (true) s1;  
else      s2;
```

$$\begin{array}{ccccc} \textbf{Fact} & & \textbf{Computed} & & \textbf{Actual} \\ \text{GEN}(S) & = & \text{GEN}(s_1) \cup \text{GEN}(s_2) & \subseteq & \text{GEN}(s_1) \end{array}$$

# RD Analysis is Approximate



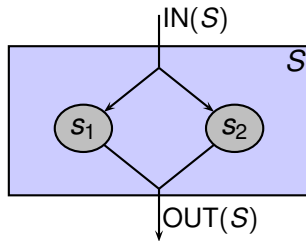
► Assumption: All paths are feasible.

► Example:

```
if (true) s1;  
else      s2;
```

Fact	Computed	Actual
$GEN(S)$	$GEN(s_1) \cup GEN(s_2)$	$GEN(s_1)$
$KILL(S)$	$KILL(s_1) \cap KILL(s_2)$	$KILL(s_1)$

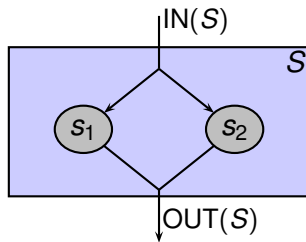
# RD Analysis is Approximate



► Thus,



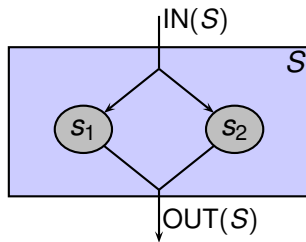
# RD Analysis is Approximate



► Thus,

$$\text{true GEN}(S) \subseteq \text{analysis GEN}(S)$$

# RD Analysis is Approximate

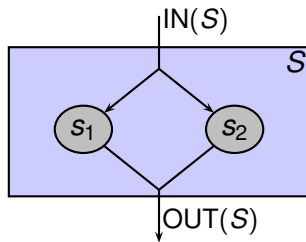


► Thus,

$\text{true GEN}(S) \subseteq \text{analysis GEN}(S)$

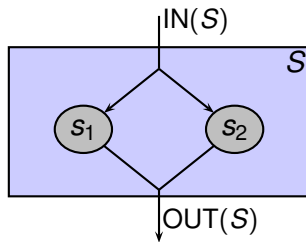
$\text{true KILL}(S) \supseteq \text{analysis KILL}(S)$

# RD Analysis is Approximate



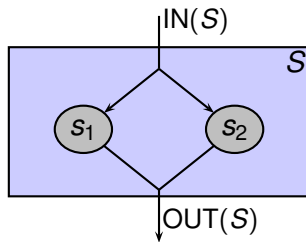
- ▶ Thus,  
     $\text{true GEN}(S) \subseteq \text{analysis GEN}(S)$   
     $\text{true KILL}(S) \supseteq \text{analysis KILL}(S)$
- ▶ More definitions computed to be reaching than actually do!

# RD Analysis is Approximate



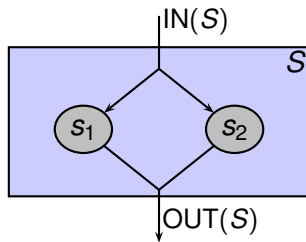
- ▶ Thus,  
 $\text{true GEN}(S) \subseteq \text{analysis GEN}(S)$   
 $\text{true KILL}(S) \supseteq \text{analysis KILL}(S)$
- ▶ More definitions computed to be reaching than actually do!
- ▶ Later we shall see that this is **SAFE** approximation

# RD Analysis is Approximate



- ▶ Thus,
  - $\text{true GEN}(S) \subseteq \text{analysis GEN}(S)$
  - $\text{true KILL}(S) \supseteq \text{analysis KILL}(S)$
- ▶ More definitions computed to be reaching than actually do!
- ▶ Later we shall see that this is **SAFE** approximation
  - ▶ prevents optimizations

# RD Analysis is Approximate



- ▶ Thus,
  - $\text{true GEN}(S) \subseteq \text{analysis GEN}(S)$
  - $\text{true KILL}(S) \supseteq \text{analysis KILL}(S)$
- ▶ More definitions computed to be reaching than actually do!
- ▶ Later we shall see that this is **SAFE** approximation
  - ▶ prevents optimizations
  - ▶ but NO wrong optimization

## RD at BB level

- ▶ A definition  $d$  can reach the start of a block from any of its predecessor

$$\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$$

## RD at BB level

- ▶ A definition  $d$  can reach the start of a block from any of its predecessor
  - ▶ if it reaches the end of some predecessor

$$\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$$



## RD at BB level

- ▶ A definition  $d$  can reach the start of a block from any of its predecessor
  - ▶ if it reaches the end of some predecessor

$$\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$$

- ▶ A definition  $d$  reaches the end of a block if

$$\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$$

## RD at BB level

- ▶ A definition  $d$  can reach the start of a block from any of its predecessor

- ▶ if it reaches the end of some predecessor

$$\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$$

- ▶ A definition  $d$  reaches the end of a block if

- ▶ either it is generated in the block

$$\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$$

## RD at BB level

- ▶ A definition  $d$  can reach the start of a block from any of its predecessor
  - ▶ if it reaches the end of some predecessor

$$\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$$

- ▶ A definition  $d$  reaches the end of a block if
  - ▶ either it is generated in the block
  - ▶ or it reaches block and not killed

$$\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$$

# Solving RD Constraints

- ▶ KILL & GEN known for each BB.

# Solving RD Constraints

- ▶ KILL & GEN known for each BB.
- ▶ A program with  $N$  BBs has  $2N$  equations with  $2N$  unknowns.

# Solving RD Constraints

- ▶ KILL & GEN known for each BB.
- ▶ A program with  $N$  BBs has  $2N$  equations with  $2N$  unknowns.
  - ▶ Solution is possible.

# Solving RD Constraints

- ▶ KILL & GEN known for each BB.
- ▶ A program with  $N$  BBs has  $2N$  equations with  $2N$  unknowns.
  - ▶ Solution is possible.
  - ▶ Iterative approach (on the next slide).

for each block  $B$  {



```
for each block  $B$  {  
   $\text{OUT}(B) = \emptyset$ ;
```

```
for each block  $B$  {  
     $\text{OUT}(B) = \emptyset$ ;  
}  
 $\text{OUT}(\text{Entry}) = \emptyset$ ; // note this for later discussion
```

```
for each block  $B$  {  
     $\text{OUT}(B) = \emptyset$ ;  
}  
 $\text{OUT}(\text{Entry}) = \emptyset$ ; // note this for later discussion  
change = true;  
while (change) {  
    change = false;
```

```
for each block  $B$  {  
     $\text{OUT}(B) = \emptyset$ ;  
}  
 $\text{OUT}(\text{Entry}) = \emptyset$ ; // note this for later discussion  
change = true;  
while (change) {  
    change = false;  
    for each block  $B$  other than  $\text{Entry}$  {
```

```
for each block  $B$  {  
     $\text{OUT}(B) = \emptyset$ ;  
}  
 $\text{OUT}(\text{Entry}) = \emptyset$ ; // note this for later discussion  
change = true;  
while (change) {  
    change = false;  
    for each block  $B$  other than  $\text{Entry}$  {  
         $\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$ ;
```

```

for each block  $B$  {
     $\text{OUT}(B) = \emptyset$ ;
}
 $\text{OUT}(\text{Entry}) = \emptyset$ ; // note this for later discussion
change = true;
while (change) {
    change = false;
    for each block  $B$  other than  $\text{Entry}$  {
         $\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$ ;
        oldOut =  $\text{OUT}(B)$ ;
         $\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$ ;
    }
}

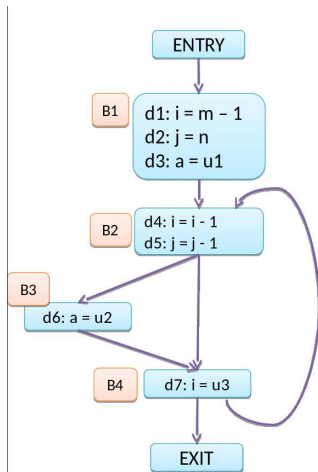
```

```

for each block  $B$  {
     $\text{OUT}(B) = \emptyset$ ;
}
 $\text{OUT}(\text{Entry}) = \emptyset$ ; // note this for later discussion
change = true;
while (change) {
    change = false;
    for each block  $B$  other than  $\text{Entry}$  {
         $\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$ ;
        oldOut =  $\text{OUT}(B)$ ;
         $\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$ ;
        if ( $\text{OUT}(B) \neq \text{oldOut}$ ) then {
            change = true;
        }
    }
}

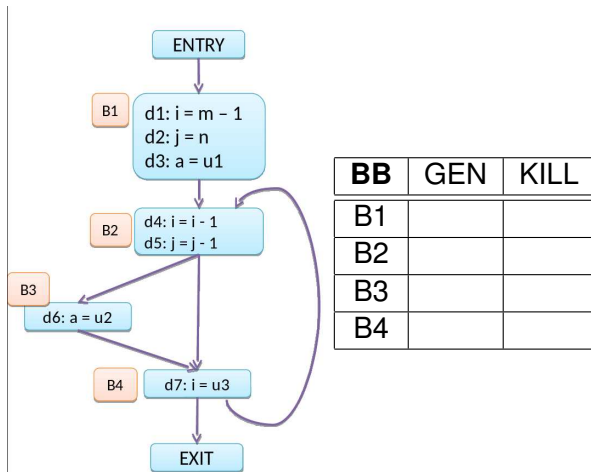
```

# Reaching Definitions: Example

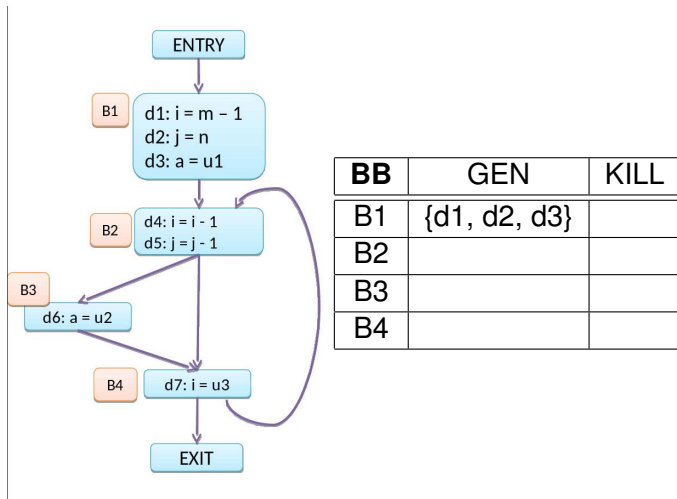




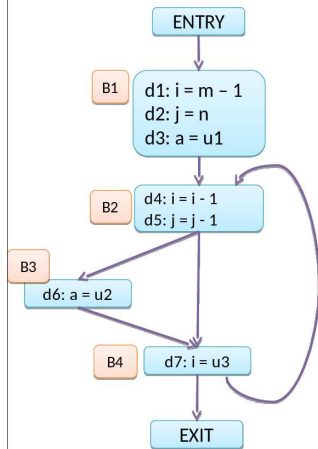
# Reaching Definitions: Example



# Reaching Definitions: Example

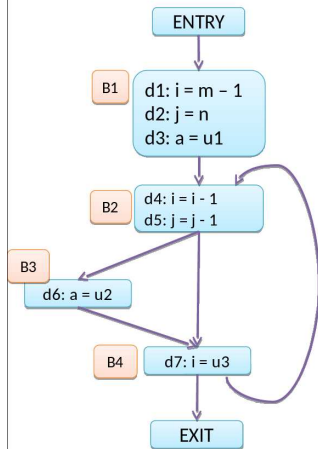


# Reaching Definitions: Example



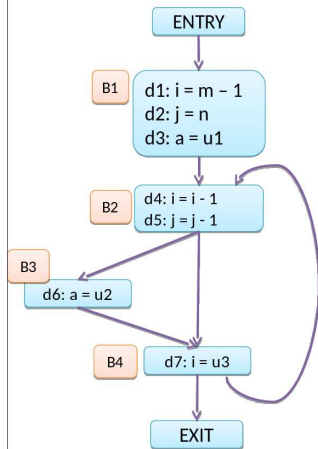
BB	GEN	KILL
B1	{d1, d2, d3}	{d4, d5, d6, d7}
B2		
B3		
B4		

# Reaching Definitions: Example



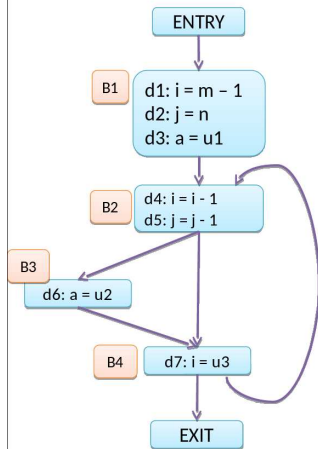
BB	GEN	KILL
B1	{d1, d2, d3}	{d4, d5, d6, d7}
B2	{d4, d5}	
B3		
B4		

# Reaching Definitions: Example



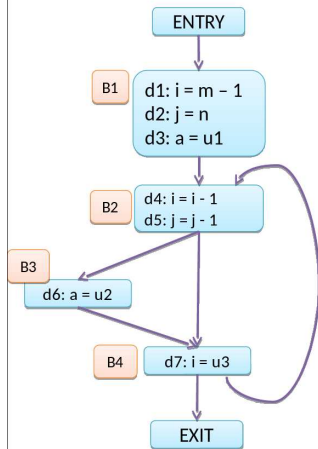
BB	GEN	KILL
B1	{d1, d2, d3}	{d4, d5, d6, d7}
B2	{d4, d5}	{d1, d2, d7}
B3		
B4		

# Reaching Definitions: Example



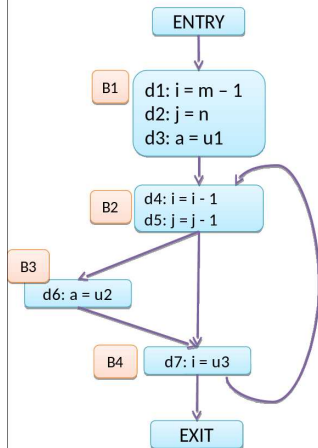
BB	GEN	KILL
B1	{d1, d2, d3}	{d4, d5, d6, d7}
B2	{d4, d5}	{d1, d2, d7}
B3	{d6}	
B4		

# Reaching Definitions: Example



BB	GEN	KILL
B1	{d1, d2, d3}	{d4, d5, d6, d7}
B2	{d4, d5}	{d1, d2, d7}
B3	{d6}	{d3}
B4		

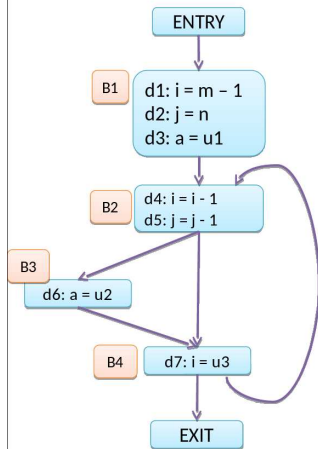
# Reaching Definitions: Example



BB	GEN	KILL
B1	{d1, d2, d3}	{d4, d5, d6, d7}
B2	{d4, d5}	{d1, d2, d7}
B3	{d6}	{d3}
B4	{d7}	

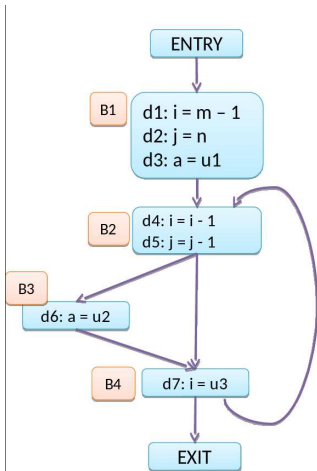


# Reaching Definitions: Example



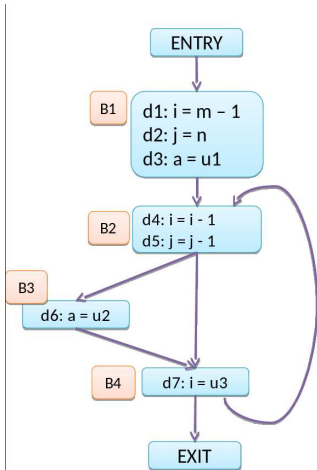
BB	GEN	KILL
B1	{d1, d2, d3}	{d4, d5, d6, d7}
B2	{d4, d5}	{d1, d2, d7}
B3	{d6}	{d3}
B4	{d7}	{d1, d4}

# Reaching Definitions: Example



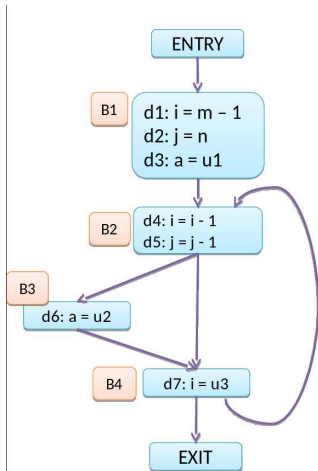
Pass#	Pt	B1	B2	B3	B4
Init	IN	-	-	-	-
	OUT	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

# Reaching Definitions: Example



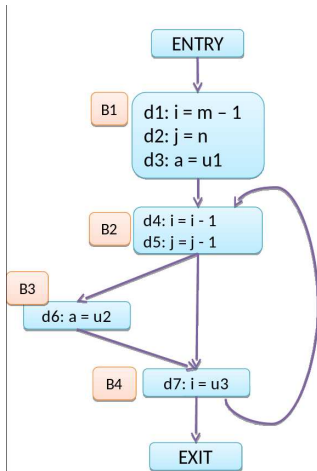
Pass#	Pt	B1	B2	B3	B4
Init	IN	-	-	-	-
	OUT	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	IN	$\emptyset$	d1, d2, d3	d3, d4, d5	d3, d4, d5, d6
	OUT	d1, d2, d3	d3, d4, d5	d4, d5, d6	d3, d5, d6, d7

# Reaching Definitions: Example



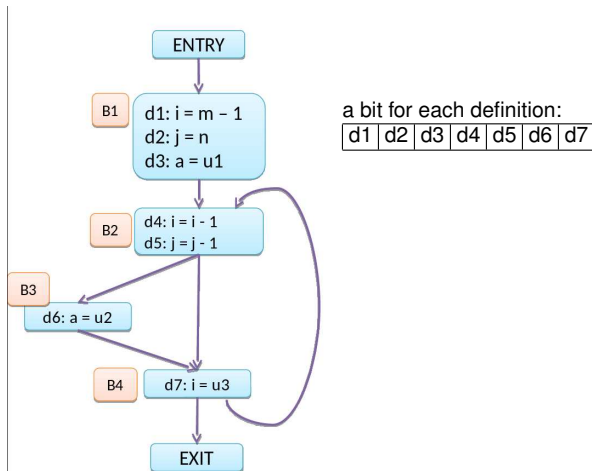
Pass#	Pt	B1	B2	B3	B4
Init	IN	-	-	-	-
	OUT	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	IN	$\emptyset$	d1, d2, d3	d3, d4, d5	d3, d4, d5, d6
	OUT	d1, d2, d3	d3, d4, d5	d4, d5, d6	d3, d5, d6, d7
2	IN	$\emptyset$	d1, d2, d3, d5, d6, d7	d3, d4, d5, d6	d3, d4, d5, d6
	OUT	d1, d2, d3	d3, d4, d5, d6	d4, d5, d6	d3, d5, d6, d7

# Reaching Definitions: Example

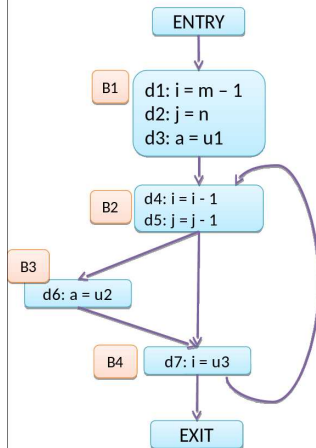


Pass#	Pt	B1	B2	B3	B4
Init	IN	-	-	-	-
	OUT	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	IN	$\emptyset$	d1, d2, d3	d3, d4, d5	d3, d4, d5, d6
	OUT	d1, d2, d3	d3, d4, d5	d4, d5, d6	d3, d5, d6, d7
2	IN	$\emptyset$	d1, d2, d3, d5, d6, d7	d3, d4, d5, d6	d3, d4, d5, d6
	OUT	d1, d2, d3	d3, d4, d5, d6	d4, d5, d6	d3, d5, d6, d7
3	IN	$\emptyset$	d1, d2, d3, d5, d6, d7	d3, d4, d5, d6	d3, d4, d5, d6
	OUT	d1, d2, d3	d3, d4, d5, d6	d4, d5, d6	d3, d5, d6, d7

# Reaching Definitions: Bitvectors



# Reaching Definitions: Bitvectors



a bit for each definition:

d1	d2	d3	d4	d5	d6	d7
----	----	----	----	----	----	----

Pass#	Pt	B1	B2	B3	B4
Init	IN	-	-	-	-
	OUT	0000000	0000000	0000000	0000000
1	IN	0000000	1110000	0011100	0011110
	OUT	1110000	0011100	0001110	0010111
2	IN	0000000	1110111	0011110	0011110
	OUT	1110000	0011110	0001110	0010111
3	IN	0000000	1110111	0011110	0011110
	OUT	1110000	0011110	0001110	0010111

# Reaching Definitions: Bitvectors

- Set-theoretic definitions:

$$\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$$

$$\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$$



# Reaching Definitions: Bitvectors

- ▶ Set-theoretic definitions:

$$\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$$

$$\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$$

- ▶ Bitvector definitions:

$$\text{IN}(B) = \bigvee_{P \in \text{PRED}(B)} \text{OUT}(P)$$

$$\text{OUT}(B) = \text{IN}(B) \wedge \neg \text{KILL}(B) \vee \text{GEN}(B)$$

# Reaching Definitions: Bitvectors

- ▶ Set-theoretic definitions:

$$\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$$

$$\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$$

- ▶ Bitvector definitions:

$$\text{IN}(B) = \bigvee_{P \in \text{PRED}(B)} \text{OUT}(P)$$

$$\text{OUT}(B) = \text{IN}(B) \wedge \neg \text{KILL}(B) \vee \text{GEN}(B)$$

- ▶ Bitwise  $\vee, \wedge, \neg$  operators

# Reaching Definitions: Application

## Constant Folding

```
while changes occur {
```

# Reaching Definitions: Application

## Constant Folding

```
while changes occur {  
    forall the stmts S of the program {
```

# Reaching Definitions: Application

## Constant Folding

```
while changes occur {  
    forall the stmts S of the program {  
        foreach operand B of S {
```

# Reaching Definitions: Application

## Constant Folding

```
while changes occur {  
  forall the stmts S of the program {  
    foreach operand B of S {  
      if there is a unique definition of B  
      that reaches S and is a constant C {
```

# Reaching Definitions: Application

## Constant Folding

```
while changes occur {  
  forall the stmts S of the program {  
    foreach operand B of S {  
      if there is a unique definition of B  
      that reaches S and is a constant C {  
        replace B by C in S;
```

# Reaching Definitions: Application

## Constant Folding

```
while changes occur {  
  forall the stmts S of the program {  
    foreach operand B of S {  
      if there is a unique definition of B  
      that reaches S and is a constant C {  
        replace B by C in S;  
      if all operands of S are constant {
```



# Reaching Definitions: Application

## Constant Folding

```
while changes occur {  
  forall the stmts S of the program {  
    foreach operand B of S {  
      if there is a unique definition of B  
      that reaches S and is a constant C {  
        replace B by C in S;  
      if all operands of S are constant {  
        replace rhs by eval(rhs);  
      }  
    }  
  }  
}
```

# Reaching Definitions: Application

## Constant Folding

```
while changes occur {  
  forall the stmts S of the program {  
    foreach operand B of S {  
      if there is a unique definition of B  
      that reaches S and is a constant C {  
        replace B by C in S;  
        if all operands of S are constant {  
          replace rhs by eval(rhs);  
          mark definition as constant;  
        }  
      }  
    }  
  }  
}
```

# Reaching Definitions: Application

- ▶ Recall the approximation in reaching definition analysis

# Reaching Definitions: Application

- ▶ Recall the approximation in reaching definition analysis  
 $\text{true GEN}(S) \subseteq \text{analysis GEN}(S)$

# Reaching Definitions: Application

- ▶ Recall the approximation in reaching definition analysis
$$\text{true GEN}(S) \subseteq \text{analysis GEN}(S)$$
$$\text{true KILL}(S) \supseteq \text{analysis KILL}(S)$$

# Reaching Definitions: Application

- ▶ Recall the approximation in reaching definition analysis
$$\text{true GEN}(S) \subseteq \text{analysis GEN}(S)$$
$$\text{true KILL}(S) \supseteq \text{analysis KILL}(S)$$
- ▶ Can it cause the application to infer

# Reaching Definitions: Application

- ▶ Recall the approximation in reaching definition analysis
$$\text{true GEN}(S) \subseteq \text{analysis GEN}(S)$$
$$\text{true KILL}(S) \supseteq \text{analysis KILL}(S)$$
- ▶ Can it cause the application to infer
  - ▶ an expression as a constant when it has different values for different executions?

# Reaching Definitions: Application

- ▶ Recall the approximation in reaching definition analysis
$$\text{true GEN}(S) \subseteq \text{analysis GEN}(S)$$
$$\text{true KILL}(S) \supseteq \text{analysis KILL}(S)$$
- ▶ Can it cause the application to infer
  - ▶ an expression as a constant when it has different values for different executions?
  - ▶ an expression as not a constant when it is a constant for all executions?



# Reaching Definitions: Application

- ▶ Recall the approximation in reaching definition analysis
$$\text{true GEN}(S) \subseteq \text{analysis GEN}(S)$$
$$\text{true KILL}(S) \supseteq \text{analysis KILL}(S)$$
- ▶ Can it cause the application to infer
  - ▶ an expression as a constant when it has different values for different executions?
  - ▶ an expression as not a constant when it is a constant for all executions?
- ▶ Safety? Profitability?

# Reaching Definitions: Summary

$$\blacktriangleright \text{GEN}(B) = \left\{ d_x \mid \begin{array}{l} d_x \text{ in } B \text{ defines variable } x \text{ and is not} \\ \text{followed by another definition of } x \text{ in } B \end{array} \right\}$$

# Reaching Definitions: Summary

- ▶  $\text{GEN}(B) = \left\{ d_x \mid \begin{array}{l} d_x \text{ in } B \text{ defines variable } x \text{ and is not} \\ \text{followed by another definition of } x \text{ in } B \end{array} \right\}$
- ▶  $\text{KILL}(B) = \{ d_x \mid B \text{ contains some definition of } x \}$

# Reaching Definitions: Summary

- ▶  $\text{GEN}(B) = \left\{ d_x \mid \begin{array}{l} d_x \text{ in } B \text{ defines variable } x \text{ and is not} \\ \text{followed by another definition of } x \text{ in } B \end{array} \right\}$
- ▶  $\text{KILL}(B) = \{ d_x \mid B \text{ contains some definition of } x \}$
- ▶  $\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$

# Reaching Definitions: Summary

- ▶  $\text{GEN}(B) = \left\{ d_x \mid \begin{array}{l} d_x \text{ in } B \text{ defines variable } x \text{ and is not} \\ \text{followed by another definition of } x \text{ in } B \end{array} \right\}$
- ▶  $\text{KILL}(B) = \{ d_x \mid B \text{ contains some definition of } x \}$
- ▶  $\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$
- ▶  $\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$

# Reaching Definitions: Summary

- ▶  $\text{GEN}(B) = \left\{ d_x \mid \begin{array}{l} d_x \text{ in } B \text{ defines variable } x \text{ and is not} \\ \text{followed by another definition of } x \text{ in } B \end{array} \right\}$
- ▶  $\text{KILL}(B) = \{ d_x \mid B \text{ contains some definition of } x \}$
- ▶  $\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$
- ▶  $\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$
- ▶ meet (  $\wedge$  ) operator: The operator to combine information coming along different predecessors is  $\cup$

# Reaching Definitions: Summary

- ▶  $\text{GEN}(B) = \left\{ d_x \mid \begin{array}{l} d_x \text{ in } B \text{ defines variable } x \text{ and is not} \\ \text{followed by another definition of } x \text{ in } B \end{array} \right\}$
- ▶  $\text{KILL}(B) = \{ d_x \mid B \text{ contains some definition of } x \}$
- ▶  $\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$
- ▶  $\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$
- ▶ meet (  $\wedge$  ) operator: The operator to combine information coming along different predecessors is  $\cup$
- ▶ What about the *Entry* block?

# Reaching Definitions: Summary

- ▶ Entry block has to be initialized specially:

$$\begin{aligned}\text{OUT}(\textit{Entry}) &= \text{EntryInfo} \\ \text{EntryInfo} &= \emptyset\end{aligned}$$



# Reaching Definitions: Summary

- ▶ Entry block has to be initialized specially:

$$\begin{aligned}\text{OUT}(\textit{Entry}) &= \text{EntryInfo} \\ \text{EntryInfo} &= \emptyset\end{aligned}$$

- ▶ A better entry info could be:

$$\text{EntryInfo} = \{x = \textit{undefined} \mid x \textit{ is a variable}\}$$

# Reaching Definitions: Summary

- ▶ Entry block has to be initialized specially:

$$\begin{aligned}\text{OUT}(\textit{Entry}) &= \text{EntryInfo} \\ \text{EntryInfo} &= \emptyset\end{aligned}$$

- ▶ A better entry info could be:

$$\text{EntryInfo} = \{x = \textit{undefined} \mid x \text{ is a variable}\}$$

- ▶ Why?