L17: Data-Flow Analysis

Software Analysis

IIT Guwahati · August 8, 2026

Data Flow Analysis

- A **static analysis** reasoning about the flow of data in programs.

Data Flow Analysis

- A **static analysis** reasoning about the flow of data in programs.
- Different kinds of data: **constants**, **variables**, **expressions**.

Data Flow Analysis

- A **static analysis** reasoning about the flow of data in programs.
- Different kinds of data: **constants**, **variables**, **expressions**.
- Used by **bug-finding tools** and by **compilers**.

Data Flow Analysis

- A **static analysis** reasoning about the flow of data in programs.
- Different kinds of data: **constants**, **variables**, **expressions**.
- Used by **bug-finding tools** and by **compilers**.
- *How:* propagate facts along the edges of a control-flow graph; for loops, **iterate to a fixed point**.

The WHILE Language

```
x = 5;
y = 1;
while (x != 1) {
    y = x * y;
    x = x - 1
}
```

Computes the factorial of 5 in `x`.

The WHILE Language

```
x = 5;
y = 1;
while (x != 1) {
    y = x * y;
    x = x - 1
}
```

(statement) `S ::= x = a | S1 ; S2 | if (b) { S1 } else { S2 } | while (b) { S1 }`

Computes the factorial of 5 in `x`.

The WHILE Language

```
x = 5;
y = 1;
while (x != 1) {
    y = x * y;
    x = x - 1
}
```

Computes the factorial of 5 in x.

(statement) S ::= x = a | S1 ; S2 |
if (b) { S1 } else { S2 } |
while (b) { S1 }

(arithmetic) a ::= x | n | a1 + a2 | a1 - a2 | a1 * a2

The WHILE Language

```
x = 5;
y = 1;
while (x != 1) {
    y = x * y;
    x = x - 1
}
```

Computes the factorial of 5 in `x`.

(statement) `S ::= x = a | S1 ; S2 | if (b) { S1 } else { S2 } | while (b) { S1 }`

(arithmetic) `a ::= x | n | a1 + a2 | a1 - a2 | a1 * a2`

(boolean) `b ::= true | !b | b1 && b2 | a1 != a2`

The WHILE Language

```
x = 5;
y = 1;
while (x != 1) {
    y = x * y;
    x = x - 1
}
```

Computes the factorial of 5 in x.

(statement) S ::= x = a | S1 ; S2 |
if (b) { S1 } else { S2 } |
while (b) { S1 }

(arithmetic) a ::= x | n | a1 + a2 | a1 - a2 | a1 * a2

(boolean) b ::= true | !b | b1 && b2 | a1 != a2

(variable) x

(constant) n

The WHILE Language

```
x = 5;
y = 1;
while (x != 1) {
    y = x * y;
    x = x - 1
}
```

Computes the factorial of 5 in `x`.

(statement) `S ::= x = a | S1 ; S2 | if (b) { S1 } else { S2 } | while (b) { S1 }`

(arithmetic) `a ::= x | n | a1 + a2 | a1 - a2 | a1 * a2`

(boolean) `b ::= true | !b | b1 && b2 | a1 != a2`

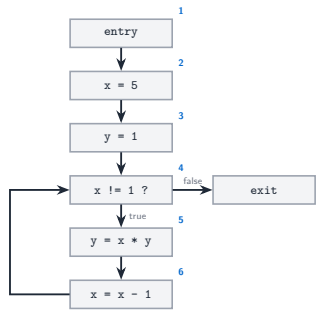
(variable) `x`

(constant) `n`

- No functions, pointers or threads.
- Loops alone already make interesting properties **undecidable**.

The Control-Flow Graph

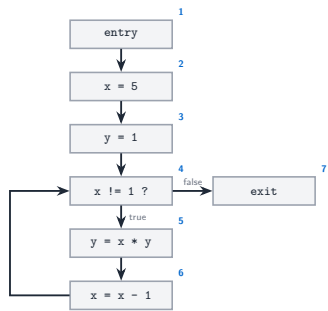
A directed graph that summarising the flow of control in *all* possible runs of the program.



The Control-Flow Graph

A directed graph that summarising the flow of control in *all* possible runs of the program.

Each **node** is one primitive statement — an assignment or a condition test.

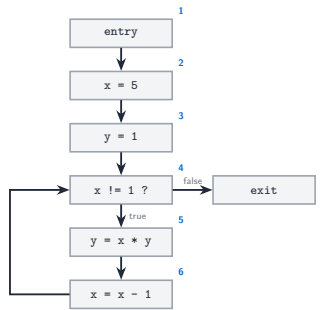


The Control-Flow Graph

A directed graph that summarising the flow of control in *all* possible runs of the program.

Each **node** is one primitive statement — an assignment or a condition test.

Each **edge** is a possible immediate successor in some run.



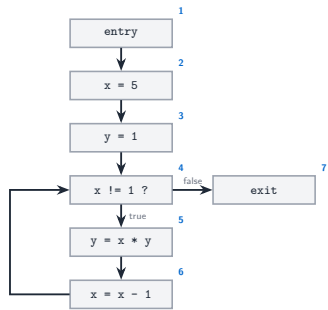
The Control-Flow Graph

A directed graph that summarising the flow of control in *all* possible runs of the program.

Each **node** is one primitive statement — an assignment or a condition test.

Each **edge** is a possible immediate successor in some run.

Nodes **1** and **7** carry no statement; they are placeholders.



The Control-Flow Graph

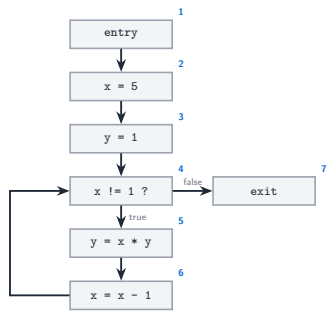
A directed graph that summarising the flow of control in *all* possible runs of the program.

Each **node** is one primitive statement — an assignment or a condition test.

Each **edge** is a possible immediate successor in some run.

Nodes **1** and **7** carry no statement; they are placeholders.

Edge 6 $\rightarrow$ 4 is the **back edge**.



The Control-Flow Graph

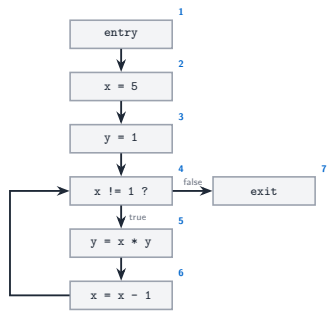
A directed graph that summarising the flow of control in *all* possible runs of the program.

Each **node** is one primitive statement — an assignment or a condition test.

Each **edge** is a possible immediate successor in some run.

Nodes **1** and **7** carry no statement; they are placeholders.

Edge 6 $\rightarrow$ 4 is the **back edge**.



Soundness, Completeness, Termination

- No analysis can be sound, complete *and* terminating.

Soundness, Completeness, Termination

- No analysis can be sound, complete *and* terminating.
- Data-flow analysis gives up **completeness**.

Soundness, Completeness, Termination

- No analysis can be sound, complete *and* terminating.
- Data-flow analysis gives up **completeness**.
- **Sound** $\Rightarrow$ it reports *all* facts that can occur in a real run.

Soundness, Completeness, Termination

- No analysis can be sound, complete *and* terminating.
- Data-flow analysis gives up **completeness**.
- **Sound** $\Rightarrow$ it reports *all* facts that can occur in a real run.
- **Incomplete** $\Rightarrow$ it may report facts that can *never* occur.

Abstracting Control-Flow Conditions

```
while (x != 1) { ...  }
```

Abstracting Control-Flow Conditions

```
while (x != 1) { ... }   ⟶   while (*) { ... }
```

Abstracting Control-Flow Conditions

```
while (x != 1) { ... }   ⟶   while (*) { ... }
```

- Conditions are abstracted by **non-deterministic choice** $*$.

Abstracting Control-Flow Conditions

```
while (x != 1) { ... }   ⟶   while (*) { ... }
```

- Conditions are abstracted by **non-deterministic choice** $*$.
- Each branch is assumed to go *either* way.

Abstracting Control-Flow Conditions

```
while (x != 1) { ... }   ⟶   while (*) { ... }
```

- Conditions are abstracted by **non-deterministic choice** $*$.
- Each branch is assumed to go *either* way.
- **All real paths are covered** — **and some impossible ones too**.

Classic Data-Flow Analyses

Classic Data-Flow Analyses

Reaching Definitions

find uninitialised variable uses

Classic Data-Flow Analyses

Reaching Definitions

find uninitialised variable uses

Very Busy Expressions

reduce code size

Classic Data-Flow Analyses

Reaching Definitions

find uninitialised variable uses

Very Busy Expressions

reduce code size

Available Expressions

avoid recomputing expressions

Classic Data-Flow Analyses

Reaching Definitions

find uninitialised variable uses

Very Busy Expressions

reduce code size

Available Expressions

avoid recomputing expressions

Live Variables

allocate registers efficiently

Classic Data-Flow Analyses

Reaching Definitions

find uninitialised variable uses

Very Busy Expressions

reduce code size

Available Expressions

avoid recomputing expressions

Live Variables

allocate registers efficiently

Modern Data-Flow Analyses

Modern Data-Flow Analyses

Interval Analysis

check memory safety

integer overflows, buffer overruns

Modern Data-Flow Analyses

Interval Analysis

check memory safety

integer overflows, buffer overruns

Type-State Analysis

temporal safety properties

correct use of protocols and library APIs

Modern Data-Flow Analyses

Interval Analysis

check memory safety

integer overflows, buffer overruns

Type-State Analysis

temporal safety properties

correct use of protocols and library APIs

Taint Analysis

check information flow

data leaks, code injection

Modern Data-Flow Analyses

Interval Analysis

check memory safety

integer overflows, buffer overruns

Type-State Analysis

temporal safety properties

correct use of protocols and library APIs

Taint Analysis

check information flow

data leaks, code injection

Concurrency Analysis

concurrency safety properties

data races, deadlocks

Modern Data-Flow Analyses

Interval Analysis

check memory safety

integer overflows, buffer overruns

Type-State Analysis

temporal safety properties

correct use of protocols and library APIs

Taint Analysis

check information flow

data leaks, code injection

Concurrency Analysis

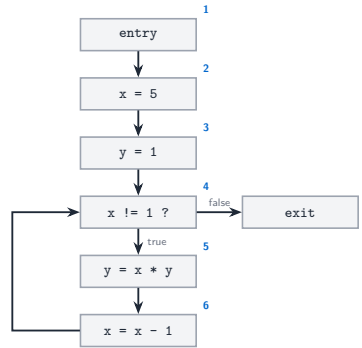
concurrency safety properties

data races, deadlocks

Reaching Definitions

Goal: Which Assignments Might Still Hold?

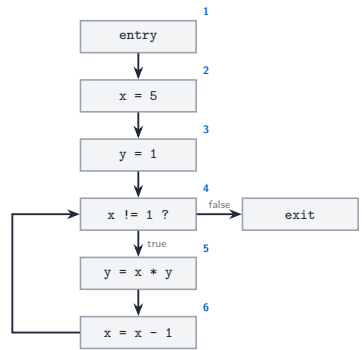
A definition **reaches** a program point p if, along *some* path to p, the assignment has been made and not overwritten.



Goal: Which Assignments Might Still Hold?

A definition **reaches** a program point p if, along *some* path to p, the assignment has been made and not overwritten.

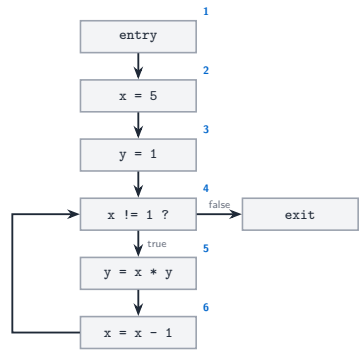
- **P1** — entry of node 4.



Goal: Which Assignments Might Still Hold?

A definition **reaches** a program point p if, along *some* path to p, the assignment has been made and not overwritten.

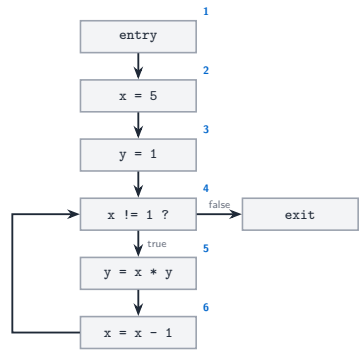
- **P1** — entry of node 4.
- **P2** — exit of node 6.



Goal: Which Assignments Might Still Hold?

A definition **reaches** a program point p if, along *some* path to p, the assignment has been made and not overwritten.

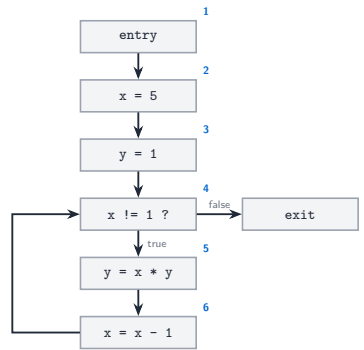
- **P1** — entry of node 4.
- **P2** — exit of node 6.
- `x = 5` **reaches P1**: no assignment to `x` on the way.



Goal: Which Assignments Might Still Hold?

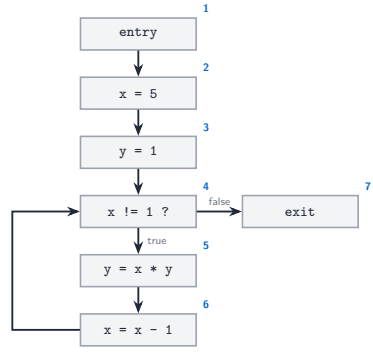
A definition **reaches** a program point p if, along *some* path to p, the assignment has been made and not overwritten.

- **P1** — entry of node 4.
- **P2** — exit of node 6.
- `x = 5` **reaches P1**: no assignment to `x` on the way.
- `x = 5` **does not reach P2**: node 6 overwrites `x` every time.



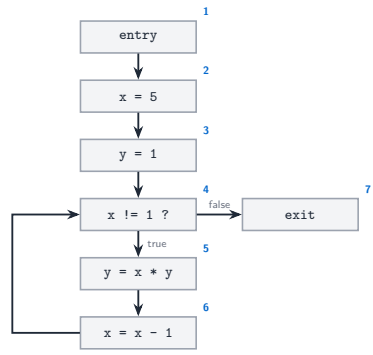
Quiz: Which Statements Are True?

1. `y = 1` reaches P1.
2. `y = 1` reaches P2.
3. `y = x * y` reaches P1.



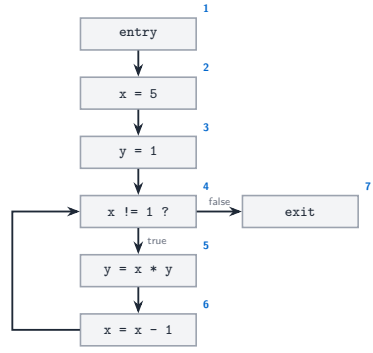
Quiz: Which Statements Are True?

1. `y = 1` reaches P1. **TRUE**
2. `y = 1` reaches P2.
3. `y = x * y` reaches P1.



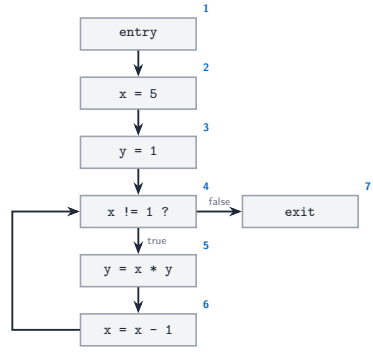
Quiz: Which Statements Are True?

1. `y = 1` reaches P1. **TRUE**
2. `y = 1` reaches P2. **FALSE**
3. `y = x * y` reaches P1.



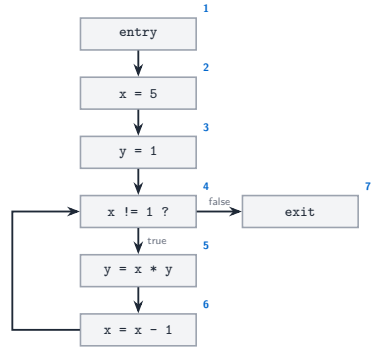
Quiz: Which Statements Are True?

1. `y = 1` reaches P1. **TRUE**
2. `y = 1` reaches P2. **FALSE**
3. `y = x * y` reaches P1. **TRUE**



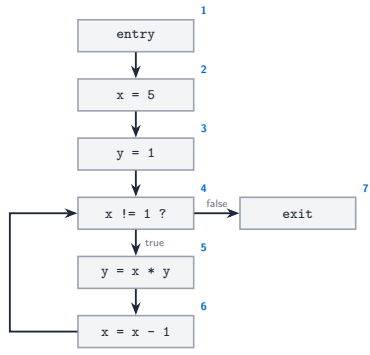
Quiz: Which Statements Are True?

1. `y = 1` reaches P1. **TRUE**
2. `y = 1` reaches P2. **FALSE**
3. `y = x * y` reaches P1. **TRUE**



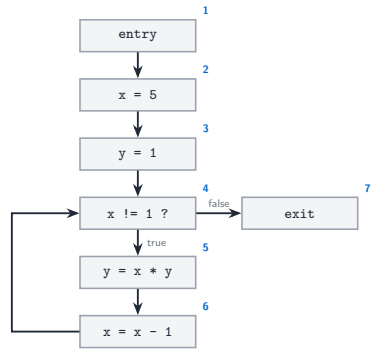
Result of the Analysis (Informally)

- A **set of facts** at each program point.



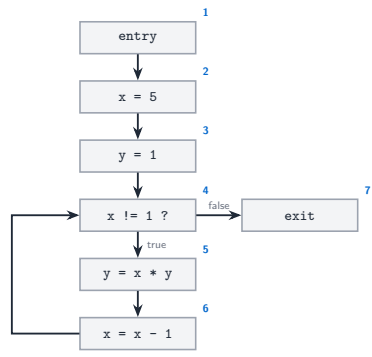
Result of the Analysis (Informally)

- A **set of facts** at each program point.
- For reaching definitions, a fact is a pair:
  ⟨ variable name, node label ⟩



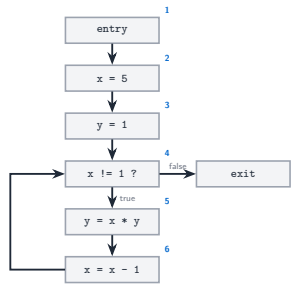
Result of the Analysis (Informally)

- A **set of facts** at each program point.
- For reaching definitions, a fact is a pair:
  ⟨ variable name, node label ⟩
- Examples: ⟨x, 2⟩, ⟨y, 5⟩.



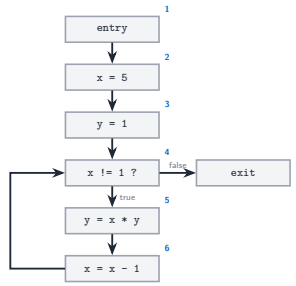
Result of the Analysis (Formally)

- $\mathrm{IN}[n]$ — facts at the **entry** of node n.



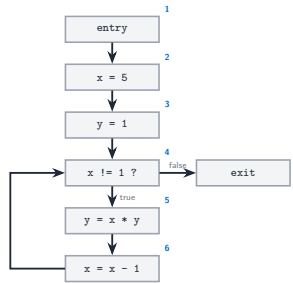
Result of the Analysis (Formally)

- $\mathrm{IN}[n]$ — facts at the **entry** of node n.
- $\mathrm{OUT}[n]$ — facts at the **exit** of node n.



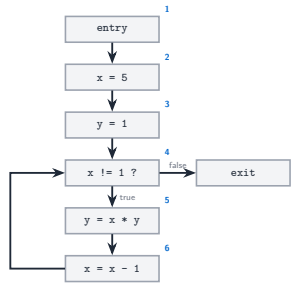
Result of the Analysis (Formally)

- $\mathrm{IN}[n]$ — facts at the **entry** of node n.
- $\mathrm{OUT}[n]$ — facts at the **exit** of node n.
- The analysis computes both, for every node.



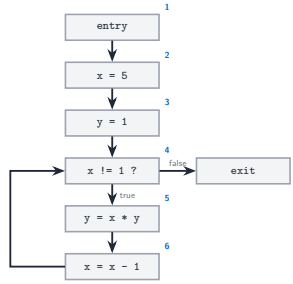
Result of the Analysis (Formally)

- $\mathrm{IN}[n]$ — facts at the **entry** of node n.
- $\mathrm{OUT}[n]$ — facts at the **exit** of node n.
- The analysis computes both, for every node.
- It repeats **two operations** until nothing changes.



Result of the Analysis (Formally)

- $\mathrm{IN}[n]$ — facts at the **entry** of node n.
- $\mathrm{OUT}[n]$ — facts at the **exit** of node n.
- The analysis computes both, for every node.
- It repeats **two operations** until nothing changes.
- That point is called **saturated**, or a **fixed point**.

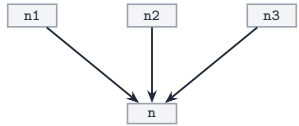


Operation #1: Merge at the Entry

$$\text{IN}[n] = \bigcup_{n' \in pred(n)} \text{OUT}[n']$$

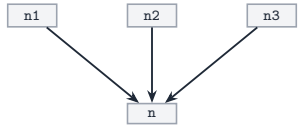
Operation #1: Merge at the Entry

$$\text{IN}[n] = \bigcup_{n' \in pred(n)} \text{OUT}[n']$$



Operation #1: Merge at the Entry

$$\text{IN}[n] = \bigcup_{n' \in pred(n)} \text{OUT}[n']$$



$$\text{IN}[n] = \text{OUT}[\texttt{n1}] \cup \text{OUT}[\texttt{n2}] \cup \text{OUT}[\texttt{n3}]$$

Operation #2: Transfer Through the Node

$$\mathrm{OUT}[n] \;=\; \big(\mathrm{IN}[n] - \textbf{KILL}[n]\big) \;\cup\; \textbf{GEN}[n]$$

Operation #2: Transfer Through the Node

$$\text{OUT}[n] \;=\; \big(\text{IN}[n] - \textbf{KILL}[n]\big) \;\cup\; \textbf{GEN}[n]$$

- **KILL**[n] — definitions *overwritten* by node n.

Operation #2: Transfer Through the Node

$$\text{OUT}[n] \;=\; \big(\text{IN}[n] - \textbf{KILL}[n]\big) \;\cup\; \textbf{GEN}[n]$$

- **KILL**[n] — definitions *overwritten* by node n.
- **GEN**[n] — definitions *generated* by node n.

Operation #2: Transfer Through the Node

$$\text{OUT}[n] \;=\; \big(\text{IN}[n] - \textbf{KILL}[n]\big) \;\cup\; \textbf{GEN}[n]$$

- **KILL**[n] — definitions *overwritten* by node n.
- **GEN**[n] — definitions *generated* by node n.
- Unlike Operation #1, this **depends on the statement** at n.

GEN and KILL: the Two Cases

Statement at n	**GEN**$[n]$	**KILL**$[n]$
a condition, e.g. `x != 1`		
an assignment `x = a`		

GEN and KILL: the Two Cases

Statement at n	**GEN**$[n]$	**KILL**$[n]$
a condition, e.g. `x != 1`	$\emptyset$	$\emptyset$
an assignment `x = a`		

GEN and KILL: the Two Cases

Statement at n	**GEN**$[n]$	**KILL**$[n]$
a condition, e.g. `x != 1`	$\emptyset$	$\emptyset$
an assignment `x = a`	$\{\langle \mathtt{x}, \mathtt{n} \rangle\}$	$\{\langle \mathtt{x}, \mathtt{m} \rangle \ : \ m \neq n\}$

GEN and KILL: the Two Cases

Statement at n	**GEN**$[n]$	**KILL**$[n]$
a condition, e.g. `x != 1`	$\emptyset$	$\emptyset$
an assignment `x = a`	$\{\langle \mathtt{x}, \mathtt{n} \rangle\}$	$\{\langle \mathtt{x}, \mathtt{m} \rangle \ : \ m \neq n\}$

A condition changes nothing: $\mathrm{OUT}[n] = \mathrm{IN}[n]$.

GEN and KILL: the Two Cases

Statement at n	**GEN**$[n]$	**KILL**$[n]$
a condition, e.g. `x != 1`	$\emptyset$	$\emptyset$
an assignment `x = a`	$\{\langle \mathtt{x}, \mathtt{n} \rangle\}$	$\{\langle \mathtt{x}, \mathtt{m} \rangle \ : \ m \neq n\}$

A condition changes nothing: $\mathrm{OUT}[n] = \mathrm{IN}[n]$.

An assignment to x kills **every other** definition of x — including $\langle \mathtt{x}, ? \rangle$.

GEN and KILL: the Two Cases

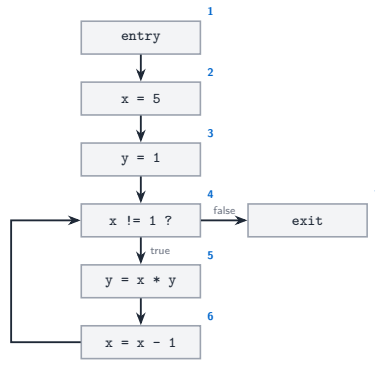
Statement at n	**GEN**$[n]$	**KILL**$[n]$
a condition, e.g. `x != 1`	$\emptyset$	$\emptyset$
an assignment `x = a`	$\{\langle \mathtt{x}, \mathtt{n} \rangle\}$	$\{\langle \mathtt{x}, \mathtt{m} \rangle \ : \ m \neq n\}$

A condition changes nothing: $\mathrm{OUT}[n] = \mathrm{IN}[n]$.

An assignment to x kills **every other** definition of x — including $\langle \mathtt{x}, ? \rangle$.

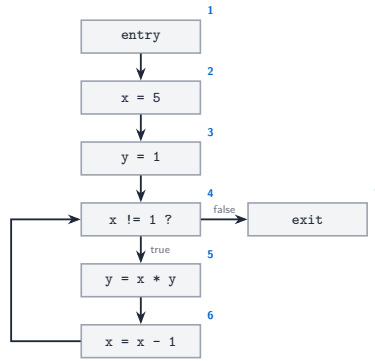
GEN and KILL for Our Program

n	Statement	GEN	KILL
2	`x = 5`		
3	`y = 1`		
4	`x != 1`		
5	`y = x * y`		
6	`x = x - 1`		



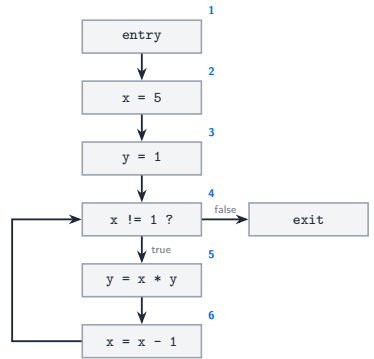
GEN and KILL for Our Program

n	Statement	GEN	KILL
2	`x = 5`	$\{\langle x, 2\rangle\}$	$\{\langle x, ?\rangle, \langle x, 6\rangle\}$
3	`y = 1`		
4	`x != 1`		
5	`y = x * y`		
6	`x = x - 1`		



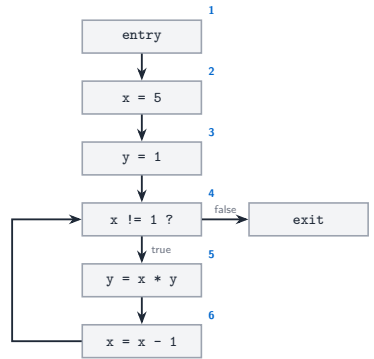
GEN and KILL for Our Program

n	Statement	GEN	KILL
2	`x = 5`	$\{\langle x, 2\rangle\}$	$\{\langle x, ?\rangle, \langle x, 6\rangle\}$
3	`y = 1`	$\{\langle y, 3\rangle\}$	$\{\langle y, ?\rangle, \langle y, 5\rangle\}$
4	`x != 1`		
5	`y = x * y`		
6	`x = x - 1`		



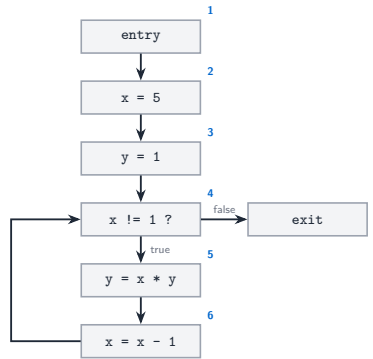
GEN and KILL for Our Program

n	Statement	GEN	KILL
2	`x = 5`	$\{\langle x, 2\rangle\}$	$\{\langle x, ?\rangle, \langle x, 6\rangle\}$
3	`y = 1`	$\{\langle y, 3\rangle\}$	$\{\langle y, ?\rangle, \langle y, 5\rangle\}$
4	`x != 1`	$\emptyset$	$\emptyset$
5	`y = x * y`		
6	`x = x - 1`		



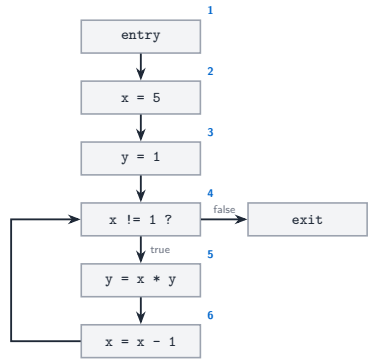
GEN and KILL for Our Program

n	Statement	GEN	KILL
2	`x = 5`	$\{\langle x, 2\rangle\}$	$\{\langle x, ?\rangle, \langle x, 6\rangle\}$
3	`y = 1`	$\{\langle y, 3\rangle\}$	$\{\langle y, ?\rangle, \langle y, 5\rangle\}$
4	`x != 1`	$\emptyset$	$\emptyset$
5	`y = x * y`	$\{\langle y, 5\rangle\}$	$\{\langle y, ?\rangle, \langle y, 3\rangle\}$
6	`x = x - 1`		



GEN and KILL for Our Program

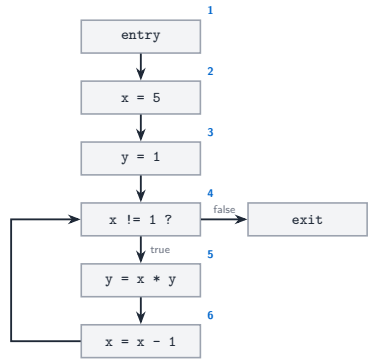
n	Statement	GEN	KILL
2	`x = 5`	$\{\langle x, 2\rangle\}$	$\{\langle x, ?\rangle, \langle x, 6\rangle\}$
3	`y = 1`	$\{\langle y, 3\rangle\}$	$\{\langle y, ?\rangle, \langle y, 5\rangle\}$
4	`x != 1`	$\emptyset$	$\emptyset$
5	`y = x * y`	$\{\langle y, 5\rangle\}$	$\{\langle y, ?\rangle, \langle y, 3\rangle\}$
6	`x = x - 1`	$\{\langle x, 6\rangle\}$	$\{\langle x, ?\rangle, \langle x, 2\rangle\}$



GEN and KILL for Our Program

n	Statement	GEN	KILL
2	`x = 5`	$\{\langle x, 2\rangle\}$	$\{\langle x, ?\rangle, \langle x, 6\rangle\}$
3	`y = 1`	$\{\langle y, 3\rangle\}$	$\{\langle y, ?\rangle, \langle y, 5\rangle\}$
4	`x != 1`	$\emptyset$	$\emptyset$
5	`y = x * y`	$\{\langle y, 5\rangle\}$	$\{\langle y, ?\rangle, \langle y, 3\rangle\}$
6	`x = x - 1`	$\{\langle x, 6\rangle\}$	$\{\langle x, ?\rangle, \langle x, 2\rangle\}$

$\langle x, ?\rangle$ and $\langle y, ?\rangle$ are **hypothetical** definitions at the program entry: "uninitialised at the start".



Chaotic Iteration Algorithm

1. **Initialise:** Set $\mathrm{IN}[n]$ and $\mathrm{OUT}[n]$ to $\emptyset$ for every node n.
2. **Seed the entry node:** Set
$$\mathrm{OUT}[1] = \{\langle x, ?\rangle, \langle y, ?\rangle\}.$$
3. **Repeat the following until convergence:**
   3.1 Compute $\mathrm{IN}[n]$ as the union of the OUT sets of all predecessors of n:
$$\mathrm{IN}[n] = \bigcup_{n' \in \mathrm{pred}(n)} \mathrm{OUT}[n'].$$
   3.2 Compute $\mathrm{OUT}[n]$ by applying the transfer operation:
$$\mathrm{OUT}[n] = \Big(\mathrm{IN}[n] - \mathrm{KILL}[n]\Big) \cup \mathrm{GEN}[n].$$
4. **Stop** when no IN or OUT set changes.

Chaotic Iteration Algorithm

1. **Initialise:** Set $\mathrm{IN}[n]$ and $\mathrm{OUT}[n]$ to $\emptyset$ for every node n.
2. **Seed the entry node:** Set
$$\mathrm{OUT}[1] = \{\langle x, ?\rangle, \langle y, ?\rangle\}.$$
3. **Repeat the following until convergence:**
   3.1 Compute $\mathrm{IN}[n]$ as the union of the OUT sets of all predecessors of n:
$$\mathrm{IN}[n] = \bigcup_{n' \in \mathrm{pred}(n)} \mathrm{OUT}[n'].$$
   3.2 Compute $\mathrm{OUT}[n]$ by applying the transfer operation:
$$\mathrm{OUT}[n] = \Big(\mathrm{IN}[n] - \mathrm{KILL}[n]\Big) \cup \mathrm{GEN}[n].$$
4. **Stop** when no IN or OUT set changes.

- **Iterative** — repeat until nothing changes.

Chaotic Iteration Algorithm

1. **Initialise:** Set $\mathrm{IN}[n]$ and $\mathrm{OUT}[n]$ to $\emptyset$ for every node n.
2. **Seed the entry node:** Set
$$\mathrm{OUT}[1] = \{\langle x, ? \rangle, \langle y, ? \rangle\}.$$
3. **Repeat the following until convergence:**
   3.1 Compute $\mathrm{IN}[n]$ as the union of the OUT sets of all predecessors of n:
$$\mathrm{IN}[n] = \bigcup_{n' \in \mathrm{pred}(n)} \mathrm{OUT}[n'].$$
   3.2 Compute $\mathrm{OUT}[n]$ by applying the transfer operation:
$$\mathrm{OUT}[n] = \Big(\mathrm{IN}[n] - \mathrm{KILL}[n]\Big) \cup \mathrm{GEN}[n].$$
4. **Stop** when no IN or OUT set changes.

- **Iterative** — repeat until nothing changes.
- **Chaotic** — nodes may be visited in any order.

Does It Always Terminate?

- The two operations are **monotonic**:

  the IN and OUT sets never shrink, they only grow.

Does It Always Terminate?

- The two operations are **monotonic**:

  the IN and OUT sets never shrink, they only grow.

- The largest any set can be is the set of **all definitions** in the program, which is **finite**.

Does It Always Terminate?

- The two operations are **monotonic**:

  the IN and OUT sets never shrink, they only grow.

- The largest any set can be is the set of **all definitions** in the program, which is **finite**.

- $\Rightarrow$ IN and OUT sets **cannot grow forever**.

Does It Always Terminate?

- The two operations are **monotonic**:

  the IN and OUT sets never shrink, they only grow.

- The largest any set can be is the set of **all definitions** in the program, which is **finite**.
- $\Rightarrow$ IN and OUT sets **cannot grow forever**.
- $\Rightarrow$ they stop changing in some iteration. **Termination guaranteed.**

Does It Always Terminate?

- The two operations are **monotonic**:

  the IN and OUT sets never shrink, they only grow.

- The largest any set can be is the set of **all definitions** in the program, which is **finite**.
- $\Rightarrow$ IN and OUT sets **cannot grow forever**.
- $\Rightarrow$ they stop changing in some iteration. **Termination guaranteed.**

Partial Order

A relation $\sqsubseteq$ on a set S is a **partial order** if, for all $A, B, C \in S$:

Partial Order

A relation $\sqsubseteq$ on a set S is a **partial order** if, for all $A, B, C \in S$:

reflexive $A \sqsubseteq A$

Partial Order

A relation $\sqsubseteq$ on a set S is a **partial order** if, for all $A, B, C \in S$:

reflexive	$A \sqsubseteq A$
antisymmetric	$A \sqsubseteq B$ and $B \sqsubseteq A \Rightarrow A = B$

Partial Order

A relation $\sqsubseteq$ on a set S is a **partial order** if, for all $A, B, C \in S$:

reflexive	$A \sqsubseteq A$
antisymmetric	$A \sqsubseteq B$ and $B \sqsubseteq A \Rightarrow A = B$
transitive	$A \sqsubseteq B$ and $B \sqsubseteq C \Rightarrow A \sqsubseteq C$

Partial Order

A relation $\sqsubseteq$ on a set S is a **partial order** if, for all $A, B, C \in S$:

reflexive	$A \sqsubseteq A$
antisymmetric	$A \sqsubseteq B$ and $B \sqsubseteq A \Rightarrow A = B$
transitive	$A \sqsubseteq B$ and $B \sqsubseteq C \Rightarrow A \sqsubseteq C$

The pair $(S, \sqsubseteq)$ is then called a **poset**.

Partial Order

A relation $\sqsubseteq$ on a set S is a **partial order** if, for all $A, B, C \in S$:

reflexive	$A \sqsubseteq A$
antisymmetric	$A \sqsubseteq B$ and $B \sqsubseteq A \Rightarrow A = B$
transitive	$A \sqsubseteq B$ and $B \sqsubseteq C \Rightarrow A \sqsubseteq C$

The pair $(S, \sqsubseteq)$ is then called a **poset**.

Our program. $S = 2^D$, the sets of definitions, with $A \sqsubseteq B$ meaning $A \subseteq B$.

Lattice

A poset $(S, \sqsubseteq)$ is a **lattice** if *every pair* $A, B \in S$ has:

Lattice

A poset $(S, \sqsubseteq)$ is a **lattice** if *every pair* $A, B \in S$ has:

a **least upper bound** $A \sqcup B$ the *join* — smallest value above both

Lattice

A poset $(S, \sqsubseteq)$ is a **lattice** if *every pair* $A, B \in S$ has:

a **least upper bound** $A \sqcup B$	the *join* — smallest value above both
a **greatest lower bound** $A \sqcap B$	the *meet* — largest value below both

Lattice

A poset $(S, \sqsubseteq)$ is a **lattice** if *every pair* $A, B \in S$ has:

a **least upper bound** $A \sqcup B$	the *join* — smallest value above both
a **greatest lower bound** $A \sqcap B$	the *meet* — largest value below both

Our program. $(2^D, \subseteq)$ is a lattice:

$$A \sqcup B = A \cup B \qquad A \sqcap B = A \cap B \qquad \bot = \emptyset \qquad \top = D$$

Lattice

A poset $(S, \sqsubseteq)$ is a **lattice** if *every pair* $A, B \in S$ has:

a **least upper bound** $A \sqcup B$	the *join* — smallest value above both
a **greatest lower bound** $A \sqcap B$	the *meet* — largest value below both

Our program. $(2^D, \subseteq)$ is a lattice:

$$A \sqcup B = A \cup B \qquad A \sqcap B = A \cap B \qquad \bot = \emptyset \qquad \top = D$$

Lattice

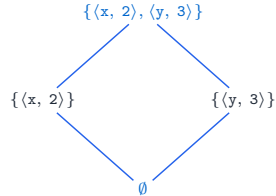
A poset $(S, \sqsubseteq)$ is a **lattice** if *every pair* $A, B \in S$ has:

a **least upper bound** $A \sqcup B$	the *join* — smallest value above both
a **greatest lower bound** $A \sqcap B$	the *meet* — largest value below both

Our program. $(2^D, \subseteq)$ is a lattice:

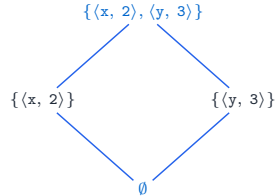
$$A \sqcup B = A \cup B \qquad A \sqcap B = A \cap B \qquad \bot = \emptyset \qquad \top = D$$

Example: Lattice



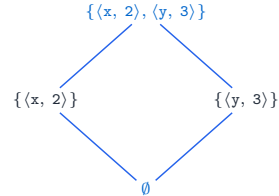
Example: Lattice

- A **lattice** is a poset where **every pair** of elements has:



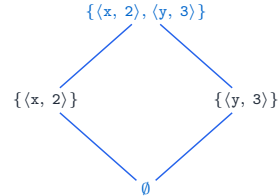
Example: Lattice

- A **lattice** is a poset where **every pair** of elements has:
  - a **least upper bound (LUB) = join** $\sqcup$



Example: Lattice

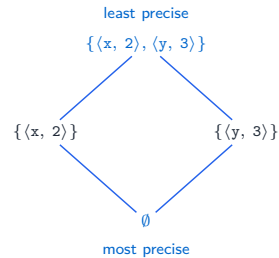
- A **lattice** is a poset where **every pair** of elements has:
  - a **least upper bound (LUB) = join** $\sqcup$
  - a **greatest lower bound (GLB) = meet** $\sqcap$



Example: Lattice

- A **lattice** is a poset where **every pair** of elements has:
  - a **least upper bound (LUB) = join** $\sqcup$
  - a **greatest lower bound (GLB) = meet** $\sqcap$
- In our example:

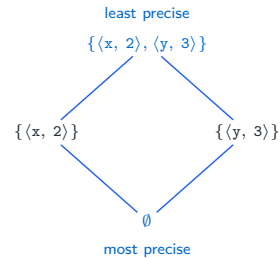
$$S = \{\emptyset, \{\langle \mathtt{x},\ 2\rangle\}, \{\langle \mathtt{y},\ 3\rangle\}, \{\langle \mathtt{x},\ 2\rangle, \langle \mathtt{y},\ 3\rangle\}\}$$



Example: Lattice

- A **lattice** is a poset where **every pair** of elements has:
  - a **least upper bound (LUB) = join** $\sqcup$
  - a **greatest lower bound (GLB) = meet** $\sqcap$
- In our example:

$$S = \{\emptyset, \{\langle \mathrm{x},\ 2\rangle\}, \{\langle \mathrm{y},\ 3\rangle\}, \{\langle \mathrm{x},\ 2\rangle, \langle \mathrm{y},\ 3\rangle\}\}$$

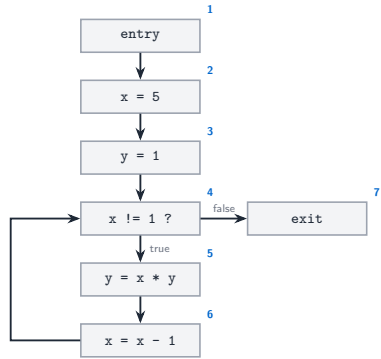


$$\{\langle \mathrm{x},\ 2\rangle\} \sqcup \{\langle \mathrm{y},\ 3\rangle\} = \{\langle \mathrm{x},\ 2\rangle, \langle \mathrm{y},\ 3\rangle\} \quad \text{(LUB / Join)}$$

$$\{\langle \mathrm{x},\ 2\rangle\} \sqcap \{\langle \mathrm{y},\ 3\rangle\} = \emptyset \quad \text{(GLB / Meet)}$$

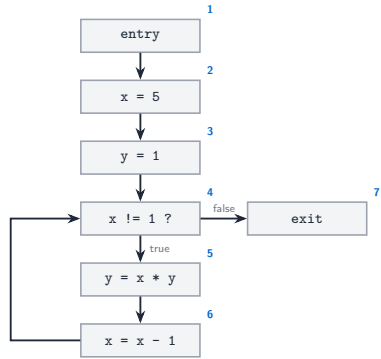
The Abstract Domain for Reaching Definitions

- Definitions: $\langle x, 2\rangle$, $\langle y, 3\rangle$, $\langle y, 5\rangle$, $\langle x, 6\rangle$.



The Abstract Domain for Reaching Definitions

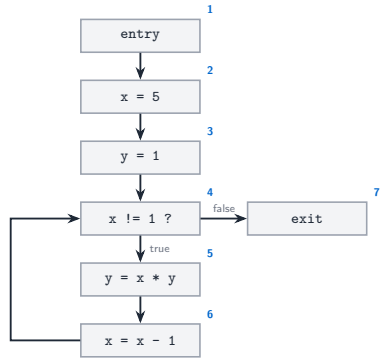
- Definitions: $\langle x, 2\rangle$, $\langle y, 3\rangle$, $\langle y, 5\rangle$, $\langle x, 6\rangle$.
- *Any* combination may reach a point.



The Abstract Domain for Reaching Definitions

- Definitions: $\langle x, 2\rangle$, $\langle y, 3\rangle$, $\langle y, 5\rangle$, $\langle x, 6\rangle$.
- *Any* combination may reach a point.
- So an abstract value is a **subset**, and the domain is the **powerset**:

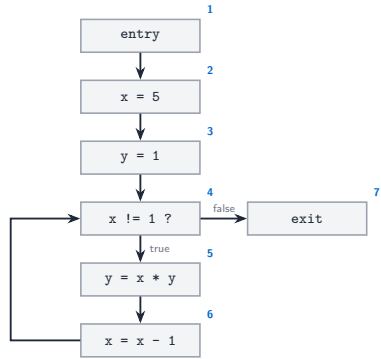
$$\left(2^D, \subseteq\right) \quad \leftarrow \text{set inclusion}$$



The Abstract Domain for Reaching Definitions

- Definitions: $\langle x, 2\rangle$, $\langle y, 3\rangle$, $\langle y, 5\rangle$, $\langle x, 6\rangle$.
- *Any* combination may reach a point.
- So an abstract value is a **subset**, and the domain is the **powerset**:

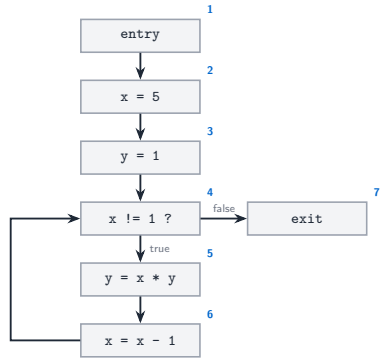
$$\left(2^D, \subseteq\right) \quad \leftarrow \text{set inclusion}$$



The Abstract Domain for Reaching Definitions

- Definitions: $\langle x, 2\rangle$, $\langle y, 3\rangle$, $\langle y, 5\rangle$, $\langle x, 6\rangle$.
- *Any* combination may reach a point.
- So an abstract value is a **subset**, and the domain is the **powerset**:

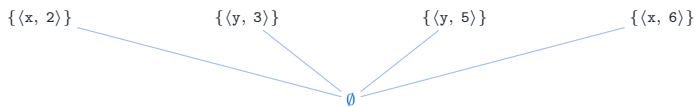
$$\left(2^D, \subseteq\right) \quad \leftarrow \text{set inclusion}$$



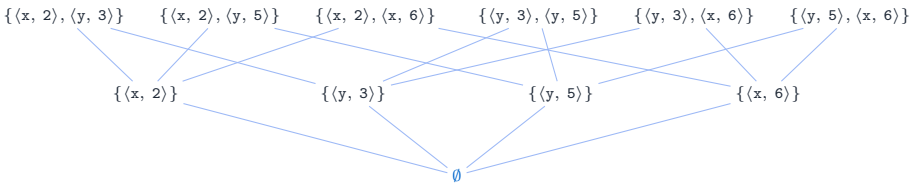
The Abstract Domain

$\emptyset$

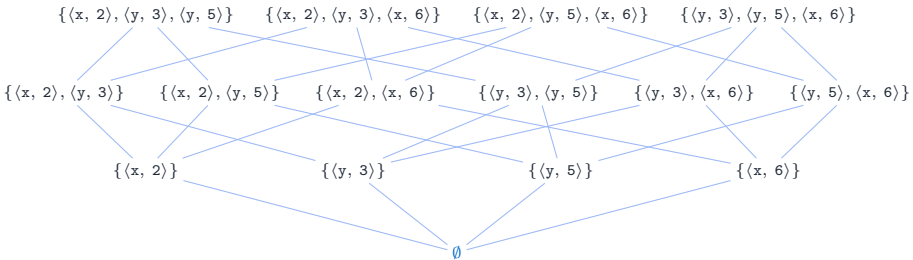
The Abstract Domain



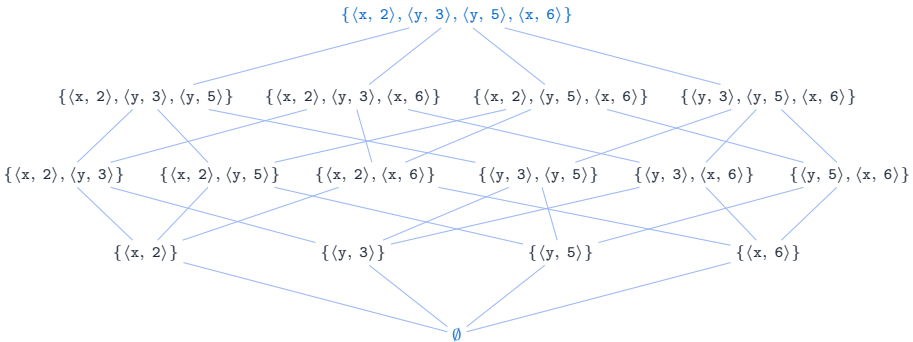
The Abstract Domain



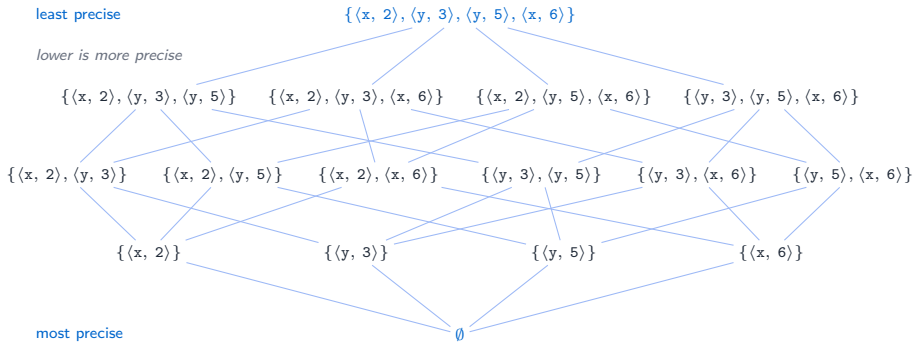
The Abstract Domain



The Abstract Domain



The Abstract Domain

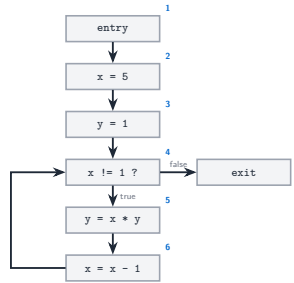


Example

Reaching definitions

Example: Initialisation

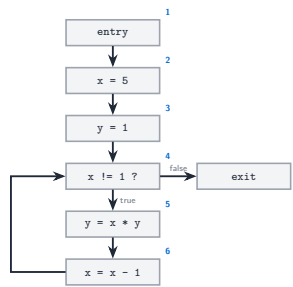
n	**IN**[n]	**OUT**[n]
1	—	
2	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$
5	$\emptyset$	$\emptyset$
6	$\emptyset$	$\emptyset$
7	$\emptyset$	—



Example: Initialisation

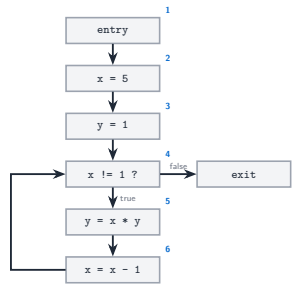
n	**IN**[n]	**OUT**[n]
1	—	$\{\langle \texttt{x}, ? \rangle, \langle \texttt{y}, ? \rangle\}$
2	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$
5	$\emptyset$	$\emptyset$
6	$\emptyset$	$\emptyset$
7	$\emptyset$	—

Only $\mathrm{OUT}[1]$ differs: both variables start uninitialised.



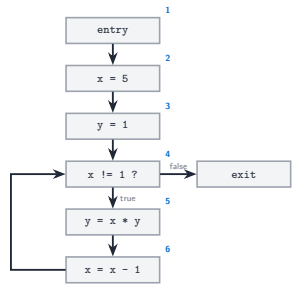
Pass 1

n	IN[n]	OUT[n]
1	__	$\{\langle x, ?\rangle, \langle y, ?\rangle\}$
2		
3		
4		
5		
6		
7		__



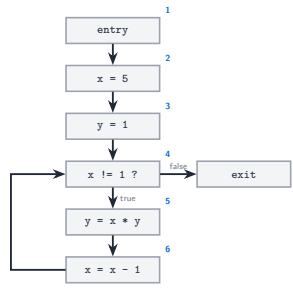
Pass 1

n	IN[n]	OUT[n]
1	__	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3		
4		
5		
6		
7		__



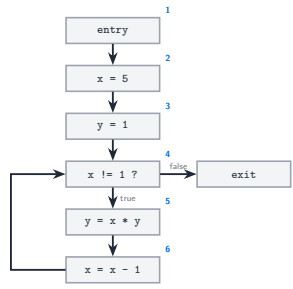
Pass 1

n	IN[n]	OUT[n]
1	__	$\{\langle x, ?\rangle, \langle y, ?\rangle\}$
2	$\{\langle x, ?\rangle, \langle y, ?\rangle\}$	$\{\langle x, 2\rangle, \langle y, ?\rangle\}$
3	$\{\langle x, 2\rangle, \langle y, ?\rangle\}$	$\{\langle x, 2\rangle, \langle y, 3\rangle\}$
4		
5		
6		
7		__



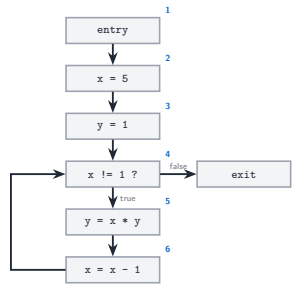
Pass 1

n	IN[n]	OUT[n]
1	__	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3	{⟨x, 2⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
4	{⟨x, 2⟩, ⟨y, 3⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
5		
6		
7		__



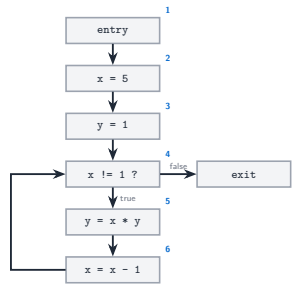
Pass 1

n	IN[n]	OUT[n]
1	__	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3	{⟨x, 2⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
4	{⟨x, 2⟩, ⟨y, 3⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
5	{⟨x, 2⟩, ⟨y, 3⟩}	{⟨x, 2⟩, ⟨y, 5⟩}
6		
7		__



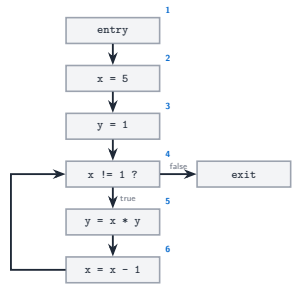
Pass 1

n	IN[n]	OUT[n]
1	__	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3	{⟨x, 2⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
4	{⟨x, 2⟩, ⟨y, 3⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
5	{⟨x, 2⟩, ⟨y, 3⟩}	{⟨x, 2⟩, ⟨y, 5⟩}
6	{⟨x, 2⟩, ⟨y, 5⟩}	{⟨x, 6⟩, ⟨y, 5⟩}
7		__



Pass 1

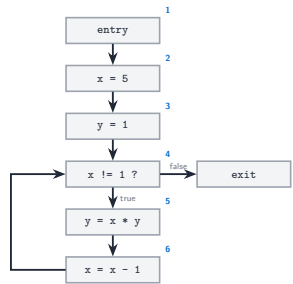
n	IN[n]	OUT[n]
1	—	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3	{⟨x, 2⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
4	{⟨x, 2⟩, ⟨y, 3⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
5	{⟨x, 2⟩, ⟨y, 3⟩}	{⟨x, 2⟩, ⟨y, 5⟩}
6	{⟨x, 2⟩, ⟨y, 5⟩}	{⟨x, 6⟩, ⟨y, 5⟩}
7	{⟨x, 2⟩, ⟨y, 3⟩}	—



Pass 1

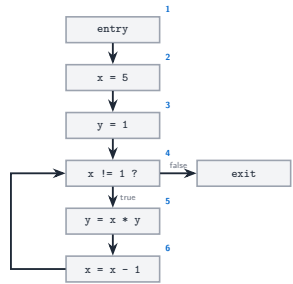
n	IN[n]	OUT[n]
1	—	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3	{⟨x, 2⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
4	{⟨x, 2⟩, ⟨y, 3⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
5	{⟨x, 2⟩, ⟨y, 3⟩}	{⟨x, 2⟩, ⟨y, 5⟩}
6	{⟨x, 2⟩, ⟨y, 5⟩}	{⟨x, 6⟩, ⟨y, 5⟩}
7	{⟨x, 2⟩, ⟨y, 3⟩}	—

Node 6 has not been visited when node 4 is computed, so the loop has not yet contributed anything.



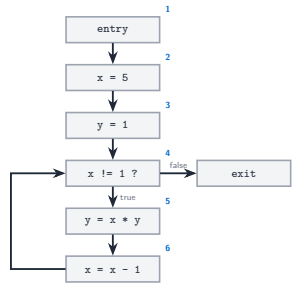
Pass 2

n	**IN**[n]	**OUT**[n]
1	—	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3	{⟨x, 2⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
4		
5		
6		
7		—



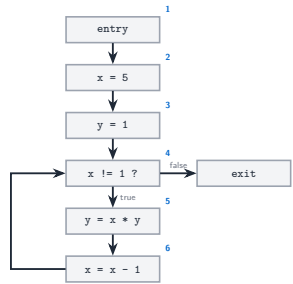
Pass 2

n	**IN**[n]	**OUT**[n]
1	—	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3	{⟨x, 2⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
4	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	
5		
6		
7		—



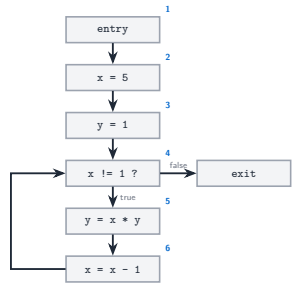
Pass 2

n	IN[n]	OUT[n]
1	—	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3	{⟨x, 2⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
4	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}
5		
6		
7		—



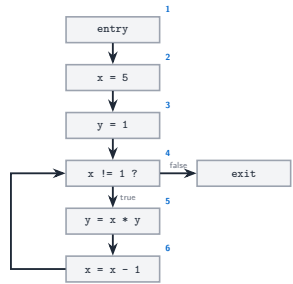
Pass 2

n	**IN**[n]	**OUT**[n]
1	—	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3	{⟨x, 2⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
4	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}
5	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 5⟩}
6		
7		—



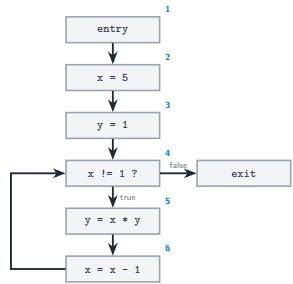
Pass 2

n	**IN**[n]	**OUT**[n]
1	—	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3	{⟨x, 2⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
4	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}
5	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 5⟩}
6	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 5⟩}	{⟨x, 6⟩, ⟨y, 5⟩}
7		—



Pass 2

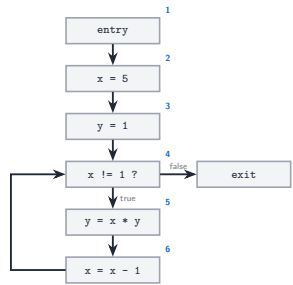
n	**IN**[n]	**OUT**[n]
1	—	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3	{⟨x, 2⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
4	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}
5	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 5⟩}
6	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 5⟩}	{⟨x, 6⟩, ⟨y, 5⟩}
7	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	—



Pass 2

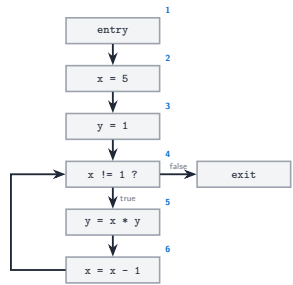
n	IN[n]	OUT[n]
1	—	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3	{⟨x, 2⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
4	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}
5	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 5⟩}
6	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 5⟩}	{⟨x, 6⟩, ⟨y, 5⟩}
7	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	—

Red entries changed. OUT[6] did *not*: ⟨x, 2⟩ is killed there either way.



Pass 3: Nothing Changes

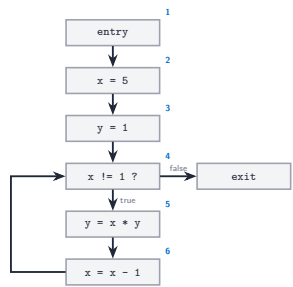
n	**IN**[n]	**OUT**[n]
1	—	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3	{⟨x, 2⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
4	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}
5	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 5⟩}
6	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 5⟩}	{⟨x, 6⟩, ⟨y, 5⟩}
7	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	—



Pass 3: Nothing Changes

n	IN[n]	OUT[n]
1	—	{⟨x, ?⟩, ⟨y, ?⟩}
2	{⟨x, ?⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, ?⟩}
3	{⟨x, 2⟩, ⟨y, ?⟩}	{⟨x, 2⟩, ⟨y, 3⟩}
4	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}
5	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 5⟩}
6	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 5⟩}	{⟨x, 6⟩, ⟨y, 5⟩}
7	{⟨x, 2⟩, ⟨x, 6⟩, ⟨y, 3⟩, ⟨y, 5⟩}	—

Saturated. This is the fixed point.



Reading the Result

- `x = 5` reaches P1: $\langle \mathtt{x}, 2 \rangle \in \mathrm{IN}[4]$ ✓

Reading the Result

- `x = 5` reaches P1: $\langle \mathtt{x}, 2 \rangle \in \mathrm{IN}[4]$ ✓
- `x = 5` misses P2: $\langle \mathtt{x}, 2 \rangle \notin \mathrm{OUT}[6]$ ✓

Reading the Result

- `x = 5` reaches P1: $\langle \mathtt{x}, 2 \rangle \in \mathrm{IN}[4]$ ✓
- `x = 5` misses P2: $\langle \mathtt{x}, 2 \rangle \notin \mathrm{OUT}[6]$ ✓
- `y = 1` misses P2: $\langle \mathtt{y}, 3 \rangle \notin \mathrm{OUT}[6]$ ✓

Reading the Result

- `x = 5` reaches P1: $\langle \mathtt{x}, 2 \rangle \in \mathrm{IN}[4]$ ✓
- `x = 5` misses P2: $\langle \mathtt{x}, 2 \rangle \notin \mathrm{OUT}[6]$ ✓
- `y = 1` misses P2: $\langle \mathtt{y}, 3 \rangle \notin \mathrm{OUT}[6]$ ✓
- `y = x * y` reaches both: $\langle \mathtt{y}, 5 \rangle \in \mathrm{IN}[4]$ and $\mathrm{OUT}[6]$ ✓

Reading the Result

- `x = 5` reaches P1: $\langle \mathtt{x}, 2 \rangle \in \mathrm{IN}[4]$ ✓
- `x = 5` misses P2: $\langle \mathtt{x}, 2 \rangle \notin \mathrm{OUT}[6]$ ✓
- `y = 1` misses P2: $\langle \mathtt{y}, 3 \rangle \notin \mathrm{OUT}[6]$ ✓
- `y = x * y` reaches both: $\langle \mathtt{y}, 5 \rangle \in \mathrm{IN}[4]$ and $\mathrm{OUT}[6]$ ✓

Application: Uninitialised Variables

- Variable v read at node n *may* be uninitialised $\iff \langle \mathrm{v}, ? \rangle \in \mathrm{IN}[n]$.

Application: Uninitialised Variables

- Variable v read at node n *may* be uninitialised $\iff \langle \mathrm{v}, ? \rangle \in \mathrm{IN}[n]$.
- Node 4 reads x, and $\mathrm{IN}[4] = \{\langle \mathrm{x}, 2\rangle, \langle \mathrm{x}, 6\rangle, \langle \mathrm{y}, 3\rangle, \langle \mathrm{y}, 5\rangle\}$.

Application: Uninitialised Variables

- Variable *v* read at node *n* *may* be uninitialised $\iff \langle \mathrm{v}, ? \rangle \in \mathrm{IN}[n]$.
- Node 4 reads x, and $\mathrm{IN}[4] = \{\langle \mathrm{x}, 2\rangle, \langle \mathrm{x}, 6\rangle, \langle \mathrm{y}, 3\rangle, \langle \mathrm{y}, 5\rangle\}$.
- $\langle \mathrm{x}, ? \rangle \notin \mathrm{IN}[4] \Rightarrow$ **x is always initialised there.** *Proved.*

Application: Uninitialised Variables

- Variable *v* read at node *n* *may* be uninitialised $\iff \langle \mathrm{v}, ? \rangle \in \mathrm{IN}[n]$.
- Node 4 reads x, and $\mathrm{IN}[4] = \{\langle \mathrm{x}, 2\rangle, \langle \mathrm{x}, 6\rangle, \langle \mathrm{y}, 3\rangle, \langle \mathrm{y}, 5\rangle\}$.
- $\langle \mathrm{x}, ? \rangle \notin \mathrm{IN}[4] \Rightarrow$ **x is always initialised there.** *Proved.*

Absence of the fact is a **proof** — soundness makes it meaningful.

Presence of the fact is only a **warning** — the path may be infeasible.