CS558
Programming Languages

Fall 2023
Lecture 2b

Andrew Tolmach
Portland State University

Semantics

- Informal vs. Formal
- Informal semantics
  - Descriptions in English (or other natural language)
  - Usually structured around grammar
  - Imprecise, incomplete, inconsistent!

Example: FORTRAN-II DO loops

DO Statement

GENERAL FORM	EXAMPLES
"DO n i = m_1, m_2" or "DO n i = m_1, m_2, m_3", where n is a statement number, i is a nonsubscripted fixed point variable, and m_1, m_2, m_3 are each either an unsigned fixed point constant or a nonsubscripted fixed point variable. If m_3 is not stated, it is assumed to be 1.	DO 30 I = 1, 10 DO 30 I = 1, M, 3

The DO statement is a command to execute repeatedly the statements which follow, up to and including the statement with statement number n. The first time, the statements are executed with i = m_1. For each succeeding execution, i is increased by m_3. After they have been executed with i equal to the highest value in this sequence of values which does not exceed m_2, control passes to the statement following the last statement in the range of the DO.

http://www.eah-jena.de/~kleine/history/languages/C28-6054-4_7090_FORTRANII.pdf

Consider

```
      DO 100 I = 10,9,1
...
100   CONTINUE
```

How many times is body executed?

That depends…!

“Experimental” semantics

- What does language feature X mean?
- Write a program that uses X, run it, and see!
- Implementation becomes standard of correctness
  - Requires reasoning from particular cases to general specification
  - Wholly non-portable, and subject to accidental change

Semantics from Interpreters

- In homework, we're building definitional interpreters for toy languages illustrating different PL constructs
- Our main goal is to study the interpreter code to understand implementation issues associated with each construct
- Interpreter also serves as semantic definition for each language
  - Defines meaning of language in terms of meaning of Scala
  - (Of course, requires knowing Scala's semantics too)
- Since interpreters are executable, can also use for "experimental" semantics

Axiomatic Semantics

- Interpreters give a kind of operational semantics for imperative statements (= commands)
- In axiomatic semantics, we give a logical interpretation to statements
- The state of an imperative program is defined by the values of all its variables
- We characterize a state by giving a logical predicate (or assertion) that is satisfied by the state's values
- We define the semantics of statements by saying how they affect arbitrary predicates
- Structured programming leads to simple axiomatic semantics!

Triples involving Assertions

$$\{\ P\ \}\ \mathtt{S}\ \{\ Q\ \}$$

- This Hoare triple claims that
  - if precondition P is true before the execution of S
  - then postcondition Q is true after the execution of S, if S terminates
  - (triple doesn't say anything if S doesn't terminate)
- Example: $\{y \geq 3\}\ \mathtt{x\ :=\ y\ +\ 1}\ \{x \geq 4\ \}$

precondition

postcondition

This triple's claim happens to be true!

Examples of triples

- Not all of these triples claim true things!

$\{x + y = c\}$ `while x > 0 do`
`y := y + 1;`
`x := x - 1`
`end` $\{x + y = c\}$ ✔

$\{ y = 2 \}$ `x := y + 1` $\{ x = 4 \}$ X

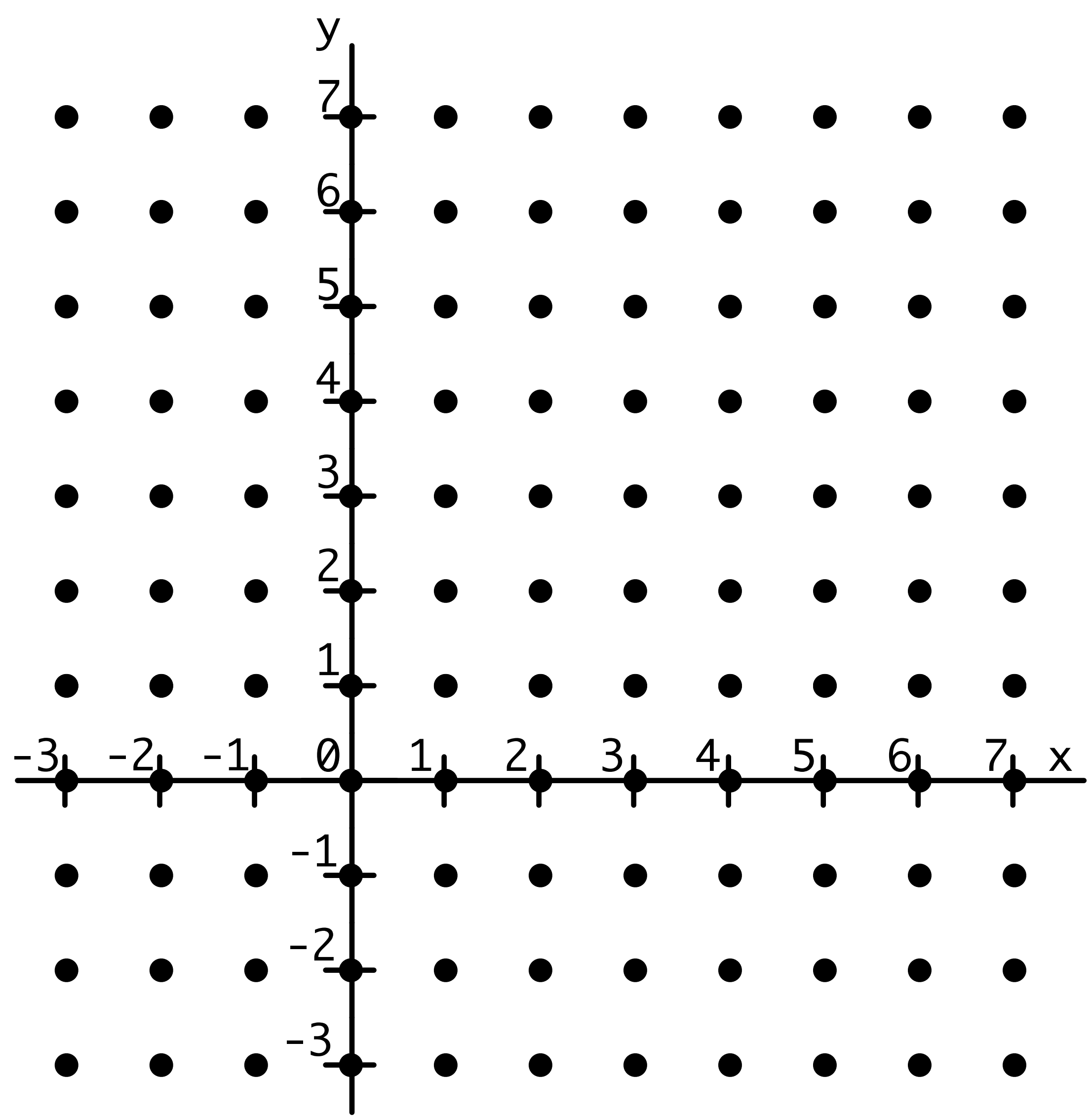
$\{ y = 2 \}$ `x := y + z` $\{ x = 4 \}$ X

$\{ True \}$ `x := 10` $\{ x = 10 \}$ ✔

$\{ False \}$ `x := 10` $\{ x = 20 \}$ ✔

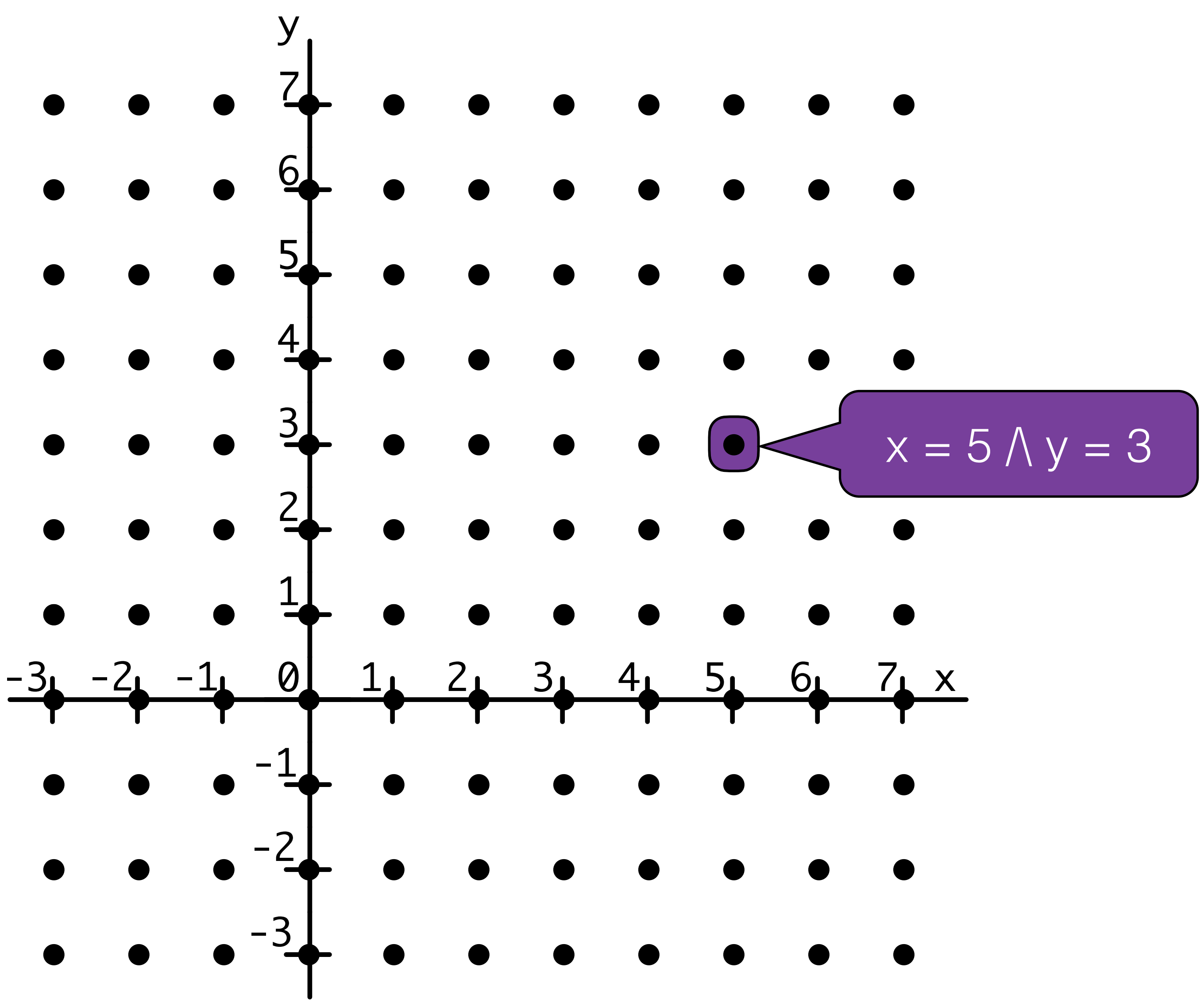
Visualizing Predicates

(assume just two integer variables, x and y)



Visualizing Predicates

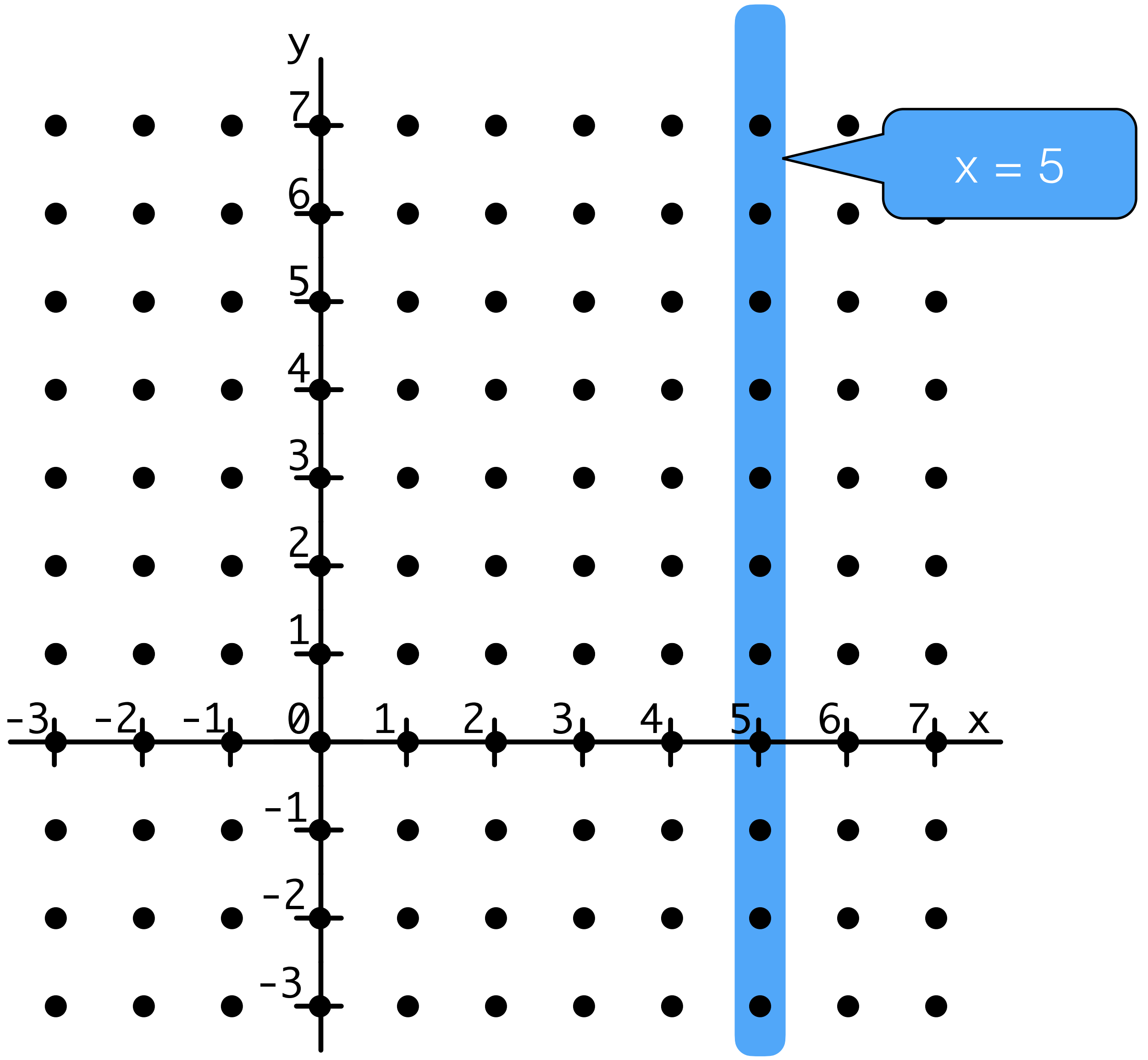
(assume just two integer variables, x and y)



Each predicate P corresponds to
a set S_P of points in the (discrete) plane

Visualizing Predicates

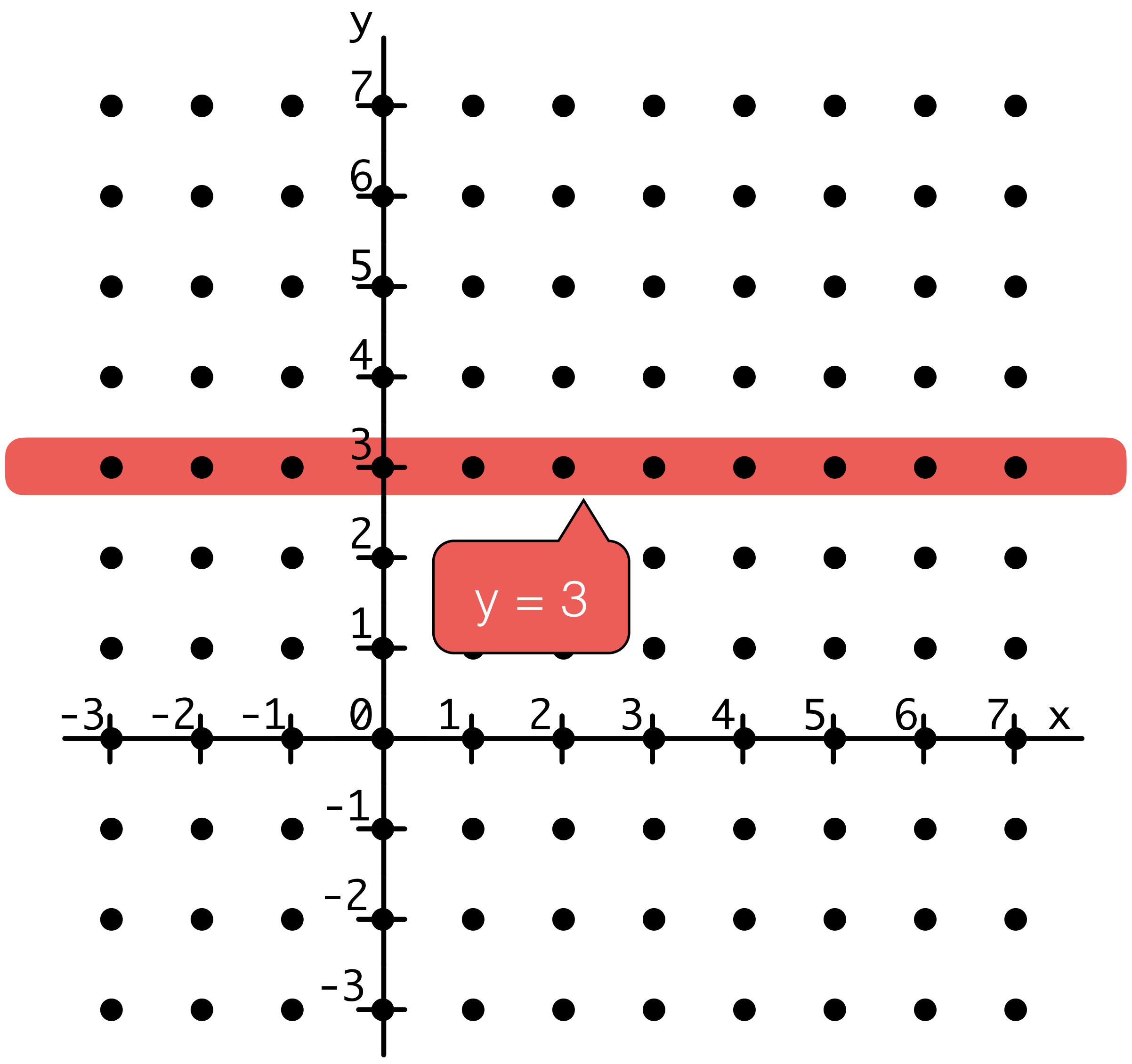
(assume just two integer variables, x and y)



Each predicate P corresponds to
a set S_P of points in the (discrete) plane

Visualizing Predicates

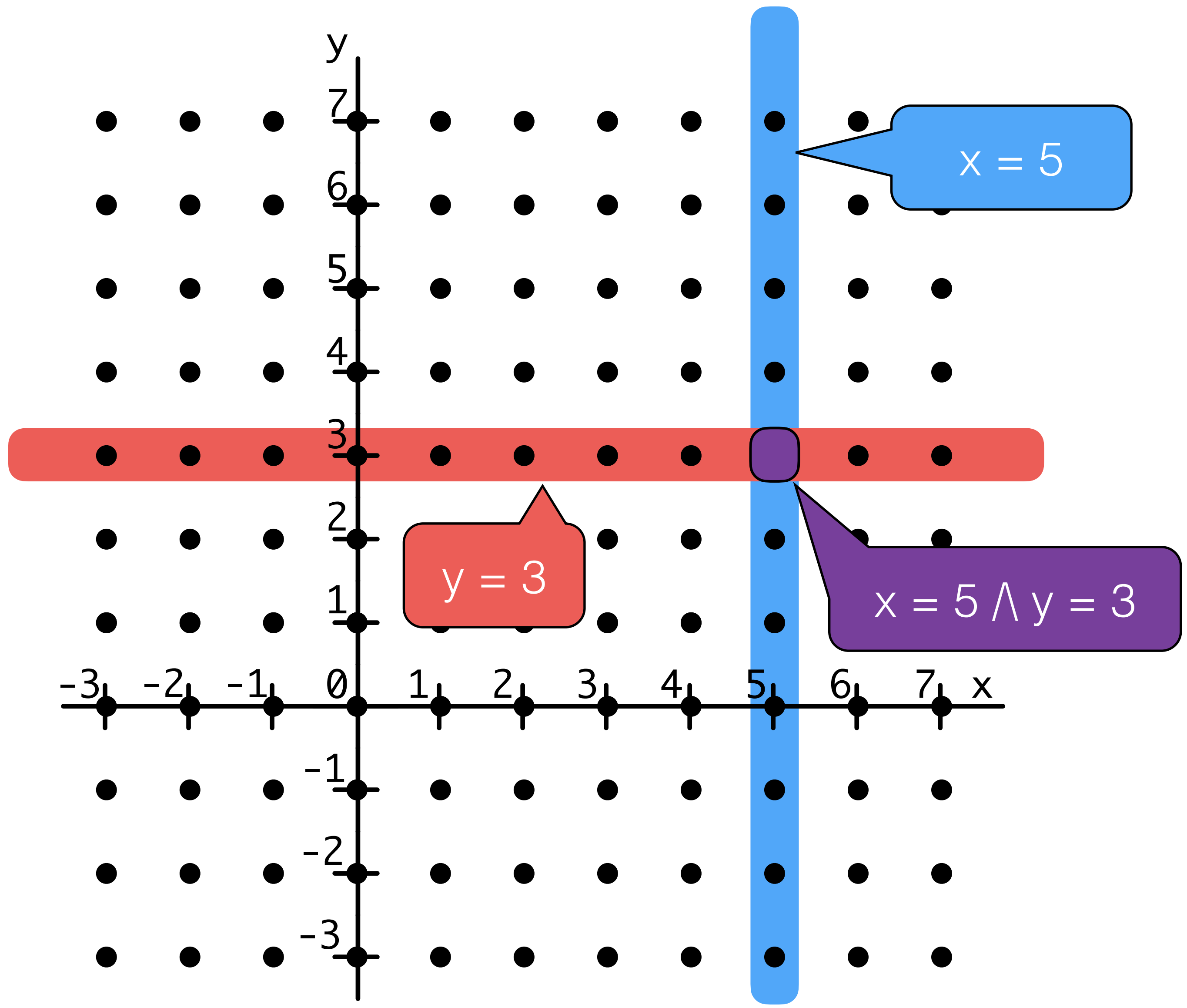
(assume just two integer variables, x and y)



Each predicate P corresponds to a set S_P of points in the (discrete) plane

Visualizing Predicates

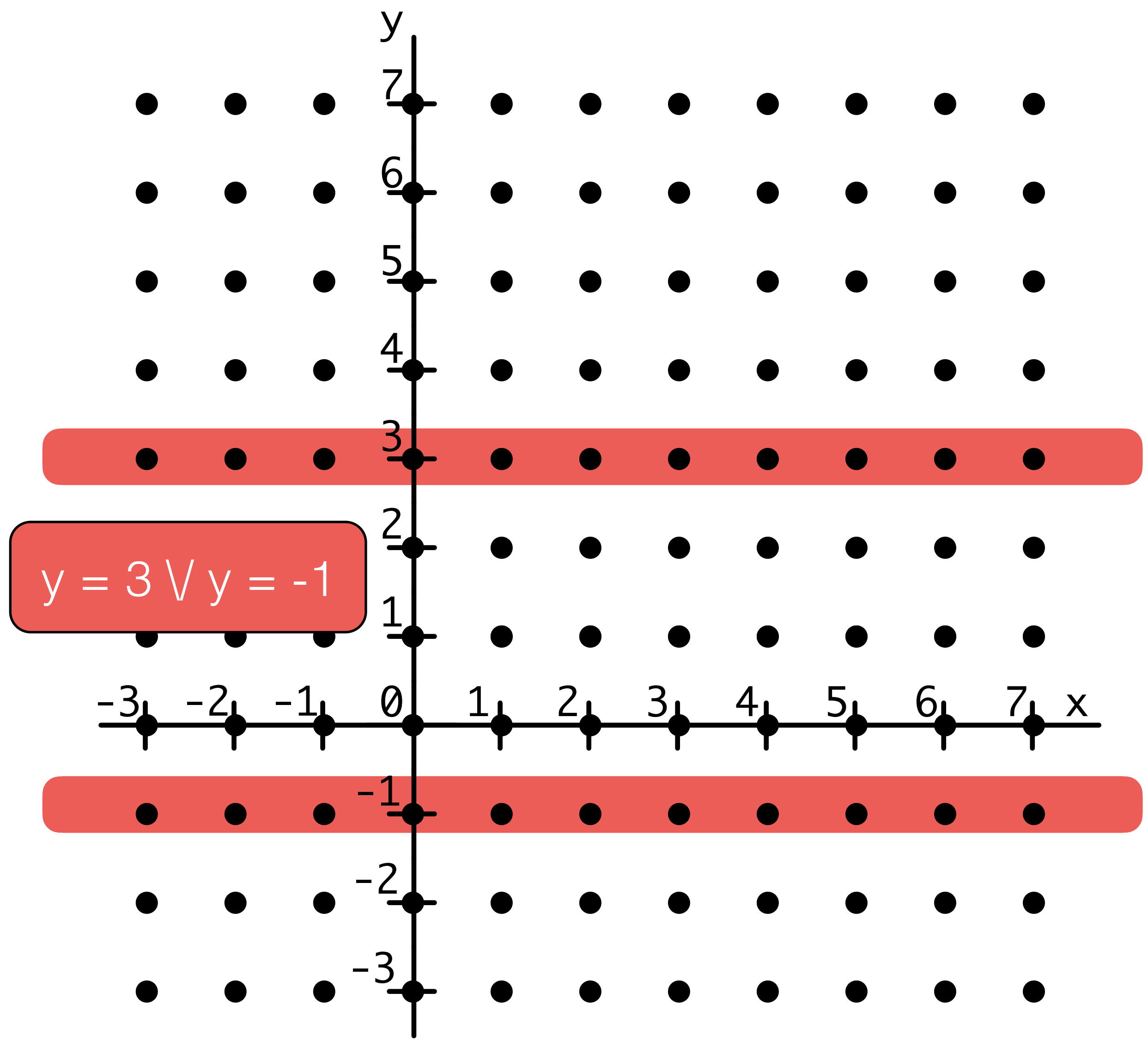
(assume just two integer variables, x and y)



P ∧ Q corresponds to $S_P \cap S_Q$

Visualizing Predicates

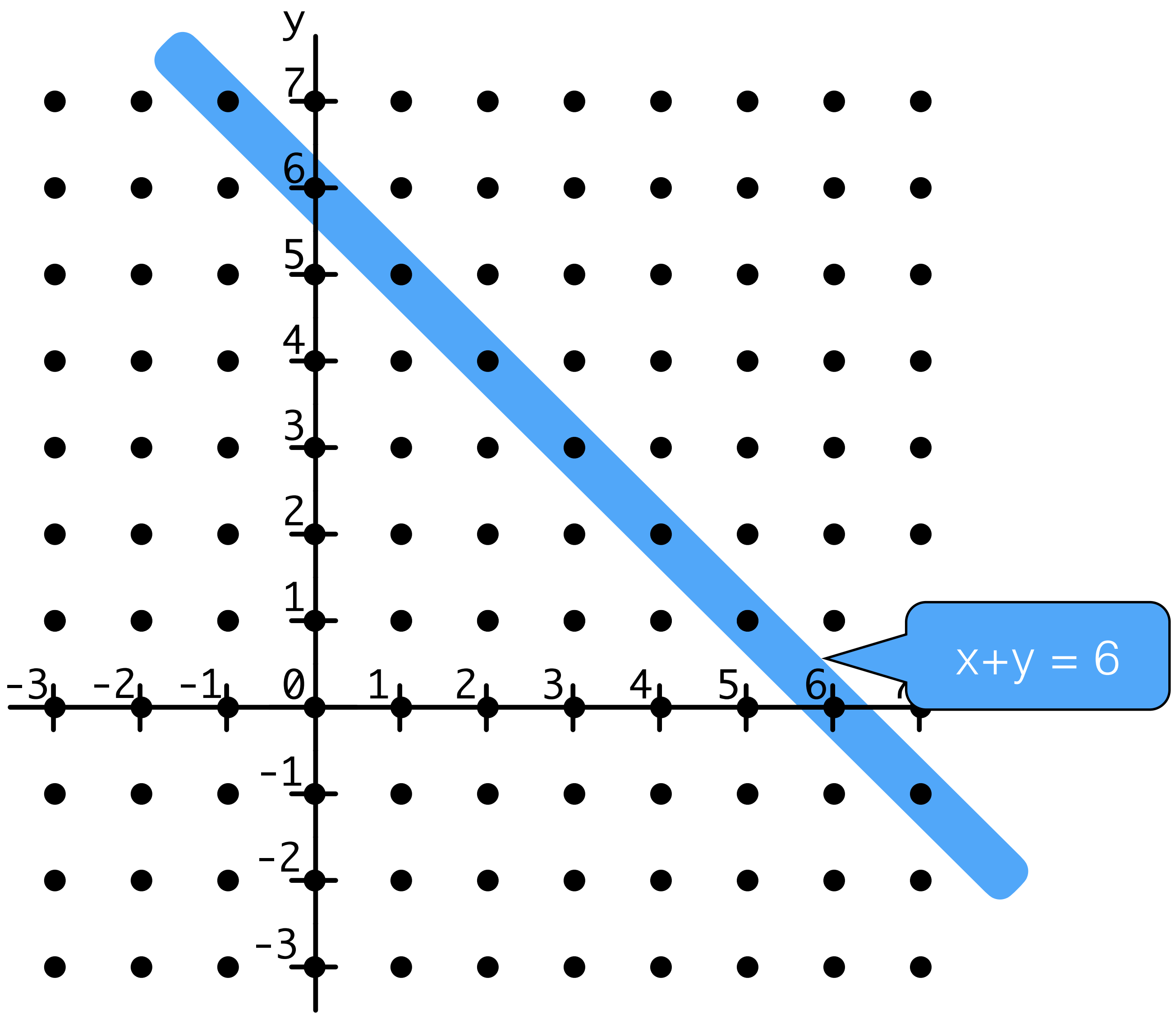
(assume just two integer variables, x and y)



P ⋁ Q corresponds to $S_P \cup S_Q$

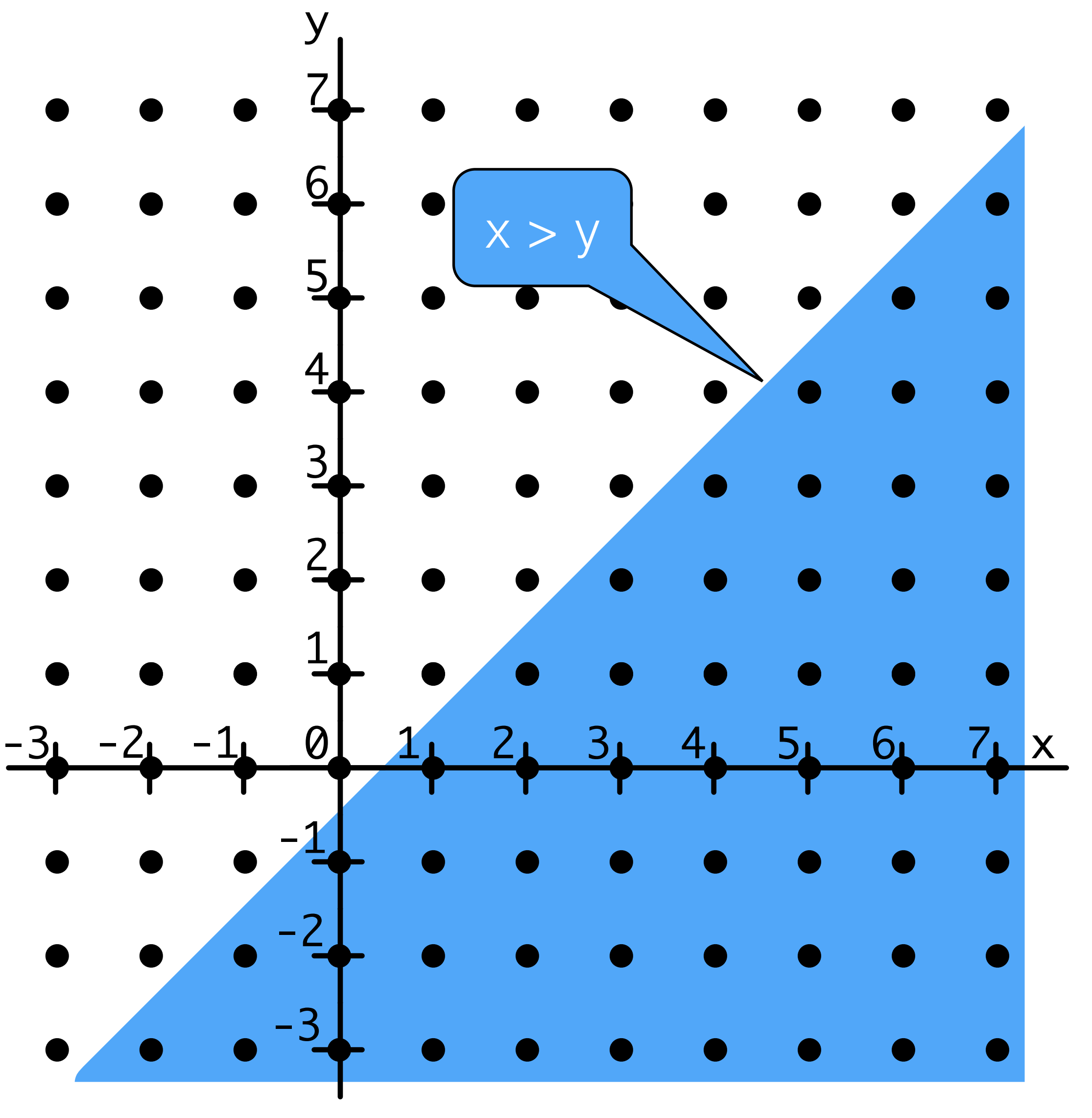
Visualizing Predicates

(assume just two integer variables, x and y)



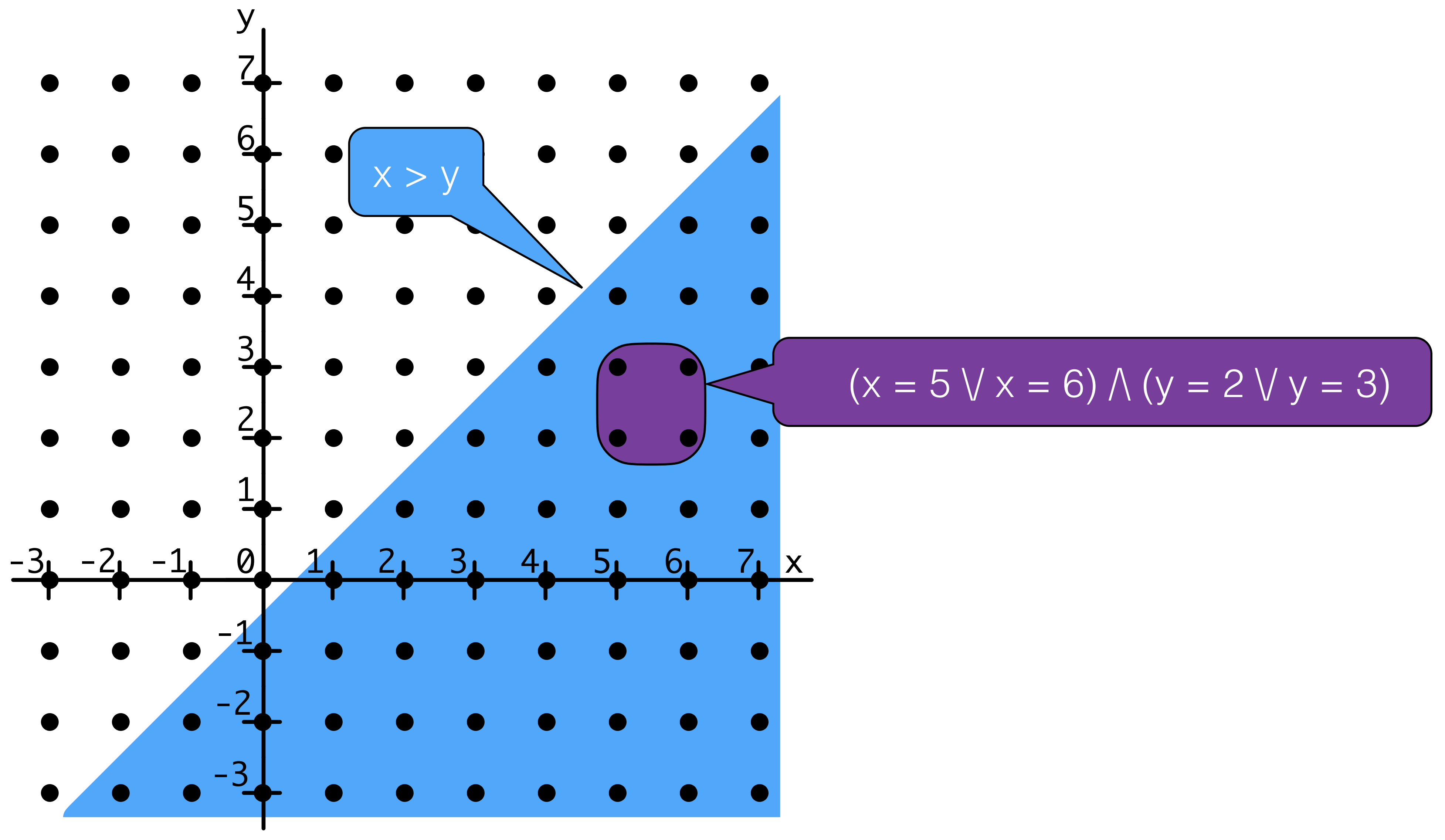
Visualizing Predicates

(assume just two integer variables, x and y)



Visualizing Predicates

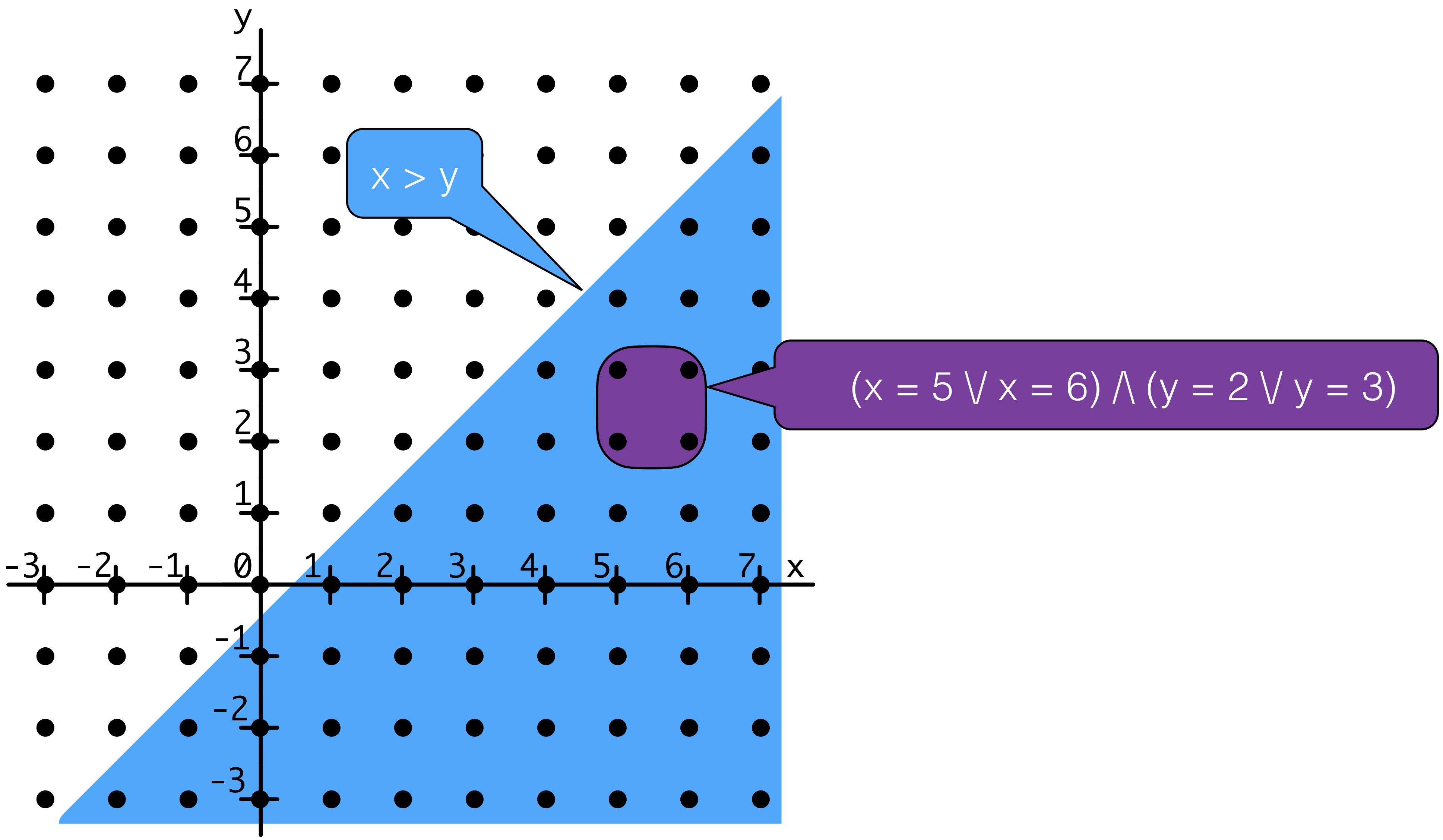
(assume just two integer variables, x and y)



P ⇒ Q corresponds to $S_P \subseteq S_Q$

Visualizing Predicates

(assume just two integer variables, x and y)

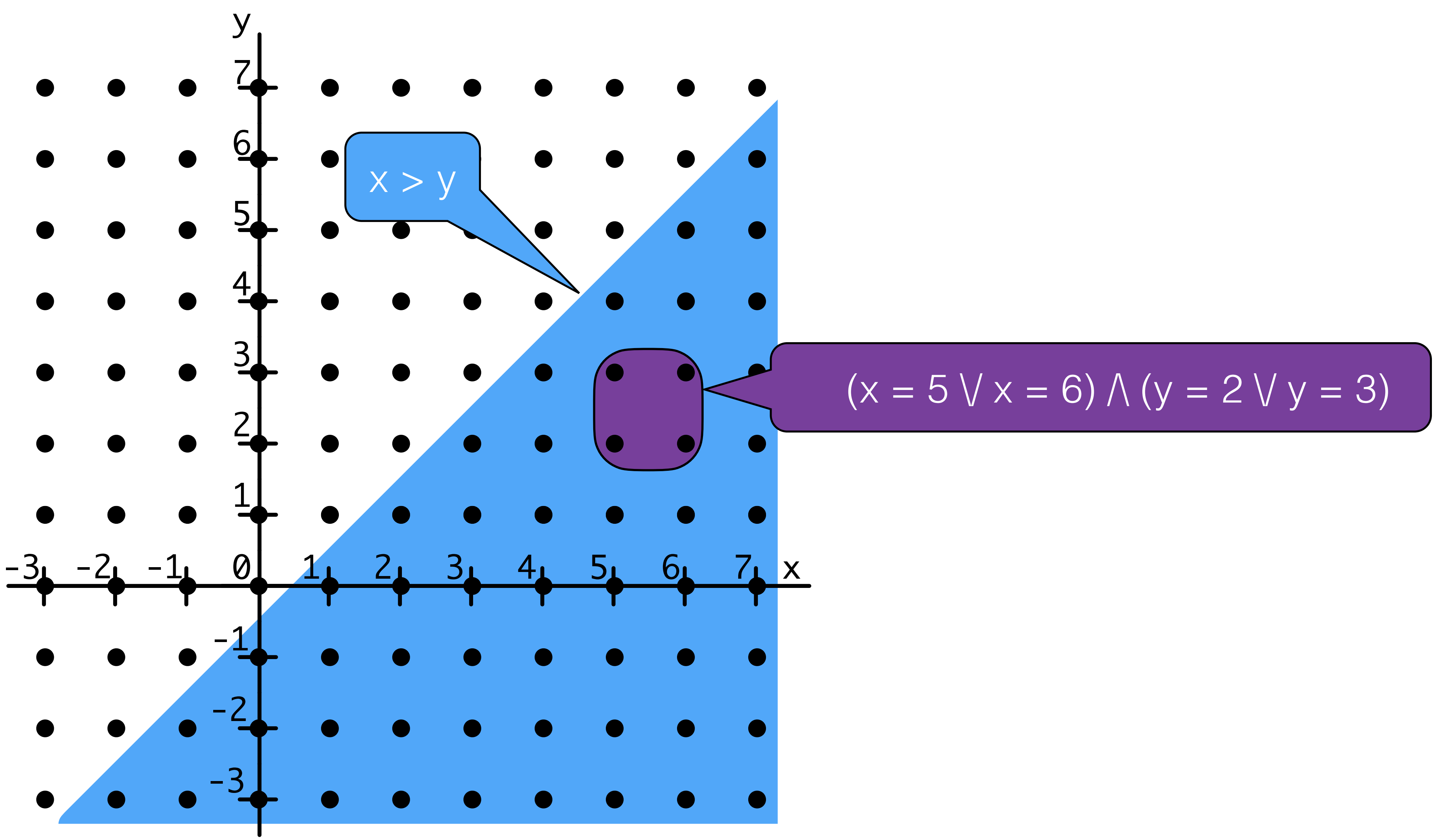


P ⇒ Q corresponds to $S_P \subseteq S_Q$

False corresponds to empty set

Visualizing Predicates

(assume just two integer variables, x and y)

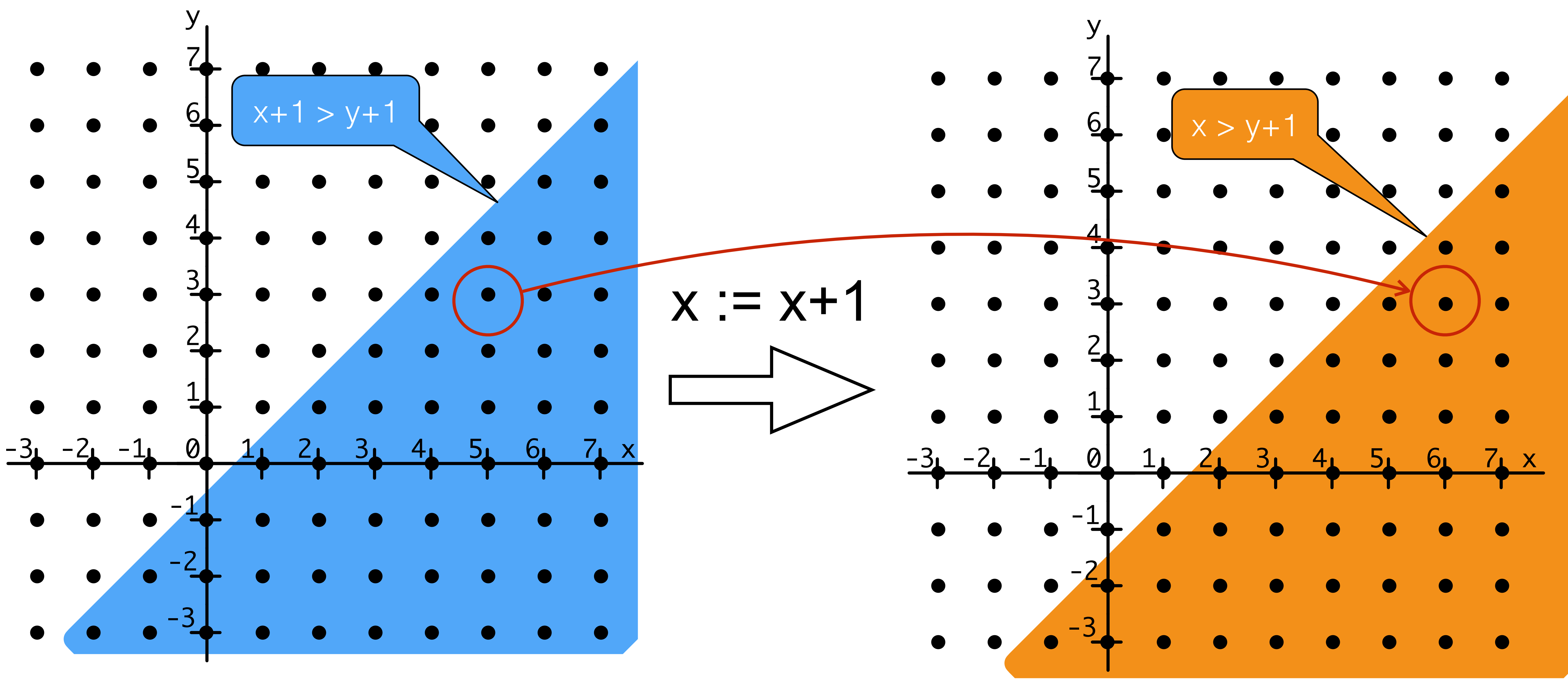


P ⇒ Q corresponds to $S_P \subseteq S_Q$

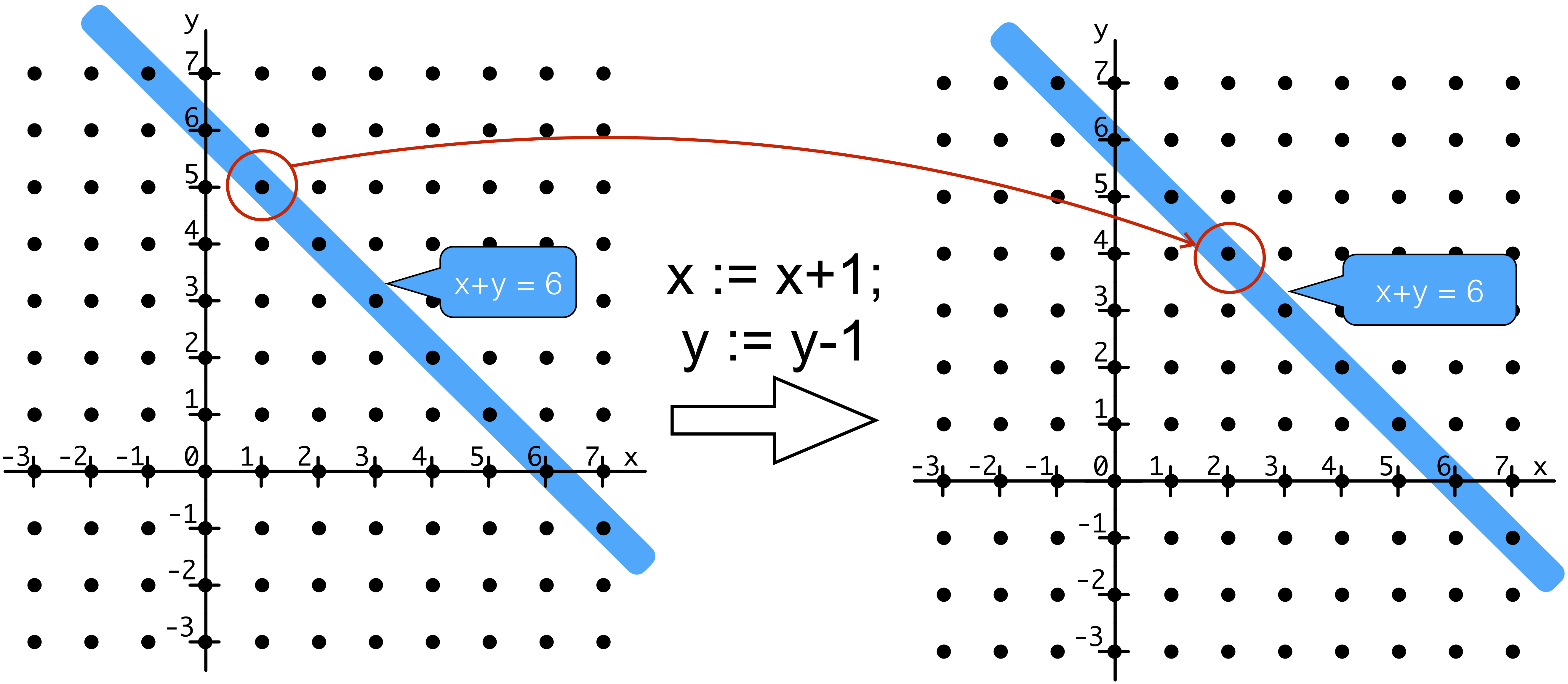
False corresponds to empty set

True corresponds to universal set

Visualizing Triples



Visualizing Invariants



Axioms and Rules of Inference

- How do we distinguish true triples from false ones?
- Who's to say which ones are true?
- It all depends on semantics of statements!
- For a suitably structured language, we can give a fixed set of axioms and rules of inference, one for each kind of statement
- True triples are those that can be logically deduced from these axioms and rules
- Of course, axioms and rules should capture what we want the statements to mean, and they need to be as strong as possible

Assignment Axiom

{ $P[E/x]$ } `x := E` { P }

where $P[E/x]$ means P with all instances of x replaced by E.

This axiom may seem backwards at first, but it makes sense if we start from the postcondition. For example, if we want to show $x \geq 4$ after the execution of

```
x := y + 1
```

then the necessary precondition is $y + 1 \geq 4$, i.e., $y \geq 3$.

More Axioms and Rules for Statements

Skip Axiom

$\{ P \}$ `skip` $\{ P \}$

Conditional Rule

$\{ P \land E \}$ S_1 $\{ Q \}$, $\{ P \land \neg E \}$ S_2 $\{ Q \}$

$\{ P \}$ `if E then` S_1 `else` S_2 `endif` $\{ Q \}$

Composition Rule

$\{ P \}$ S_1 $\{ Q \}$, $\{ Q \}$ S_2 $\{ R \}$

$\{ P \}$ S_1; S_2 $\{ R \}$

While Rule

$\{ P \land E \}$ `S` $\{ P \}$

$\{ P \}$ `while E do S` $\{ P \land \neg E \}$

Bookkeeping Rules

Consequence Rule

$$\frac{P \Rightarrow P',\ \{\ P'\ \}\ \texttt{S}\ \{\ Q'\ \},\ Q' \Rightarrow Q}{\{\ P\ \}\ \texttt{S}\ \{\ Q\ \}}$$

Here $P \Rightarrow Q$ means that "P implies Q," i.e., "Q is true whenever P is true," i.e. "P is false or Q is true." Hence we always have *False* $\Rightarrow Q$ for **any** Q !

Proof Tree Example

```
-----------------(ASSIGN)          -----------------(ASSIGN)
{x+y+1 = c+1}                      {x-1+y = c}
   y := y+1                           x := x-1
      {x+y = c+1}                       {x+y = c}
---------------------(CONSEQ)      ----------------(CONSEQ)
{x+y = c  ∧  x != 0}                  {x+y = c+1}
   y := y+1                            x := x-1
      {x+y = c+1}                        {x+y = c}
----------------------------------------------------------(COMP)
              {x+y = c  ∧  x != 0}
                  y := y+1; x := x-1
                             {x+y = c}
     ----------------------------------------------------------(WHILE)
     {x+y = c}
         while x != 0 do y := y+1; x := x-1 end
                                 {x+y = c  ∧  ¬ x != 0}
      ---------------------------------------------------------(CONSEQ)
      {x = c  ∧  y = 0 }
        while x != 0 do y := y+1; x := x-1 end
                                             {y = c}
```

Annotated Program Example

- Proof trees can get unwieldy fast.
- Common alternative is to annotate programs with assertions/assumptions.

$\{x = c \;\; \wedge \;\; y = 0\}$
$\{x + y = c\}$
`while x != 0 do`
  $\{x + y = c \;\; \wedge \;\; x \; != \;\; 0\}$
  $\{x + y + 1 = c + 1\}$
  `y := y + 1;`
  $\{x + y = c + 1\}$
  $\{x - 1 + y = c\}$
  `x := x - 1`
  $\{x + y = c\}$
`end`
$\{x + y = c \;\; \wedge \;\; \neg \;\; x \; != \;\; 0\}$
$\{y = c\}$

- Can obtain proof tree from annotated program
- Must check that annotations are consistent with each other and with rules/axioms.

Pros and cons of axiomatic semantics

- Gives a very clean semantics for structured statements
- But things get more complicated if we add features like
  - expressions with side-effects
  - statements that break out of loops
  - procedures
  - non-trivial data structures and aliases
- [See remainder of Gordon notes for more details]

Applying Axiomatic semantics

- Axiomatic viewpoint is very useful basis for formal proofs about program behavior
  - These are rarely done by hand
  - But there are beginning to be genuinely useful tools that support automated proof
  - e.g. Dafny (http://rise4fun.com/Dafny/tutorial)
- Thinking in terms of assertions is good for informal reasoning too
- Other styles of semantics use similar forms of rules