Methodologies for Software Processes

Lecture 2- Lattices and Introduction in Dataflow Analyis

(our slides are taken from other courses that use "Principles of Program Analysis" as textbook)

Partial orders, Lattices, etc.

- We aim at computing properties on programs
- How can we represent these properties?
- Which kind of algebraic features have to be satisfied on these representations?
- Which conditions guarantee that this computation terminates?

Motivating Example (1)

- Consider the renovation of the building of a firm. In this process several tasks are undertaken
  - Remove asbestos
  - Replace windows
  - Paint walls
  - Refinish floors
  - Assign offices
  - Move furniture in office
- …

Motivating Example (2)

- Clearly, some things had to be done before others could begin
    - Asbestos had to be removed before anything (except assigning offices)
    - Painting walls had to be done before refinishing floors to avoid ruining them, etc.
- On the other hand, several things could be done concurrently:
    - Painting could be done while replacing the windows
    - Assigning offices could be done at anytime before moving furniture in office
- This scenario can be nicely modeled using <u>partial orderings</u>

Partial Orderings: Definitions

- **Definitions**:
  - A relation R on a set S is called a partial order if it is
    - Reflexive
    - Antisymmetric
    - Transitive
  - A set S together with a partial ordering R is called a partially ordered set (poset, for short) and is denote (S,R)
- Partial orderings are used to give an order to sets that may not have a natural one
- In our renovation example, we could define an ordering such that $(a,b) \in R$ if ‘a must be done before b can be done’

Partial Orderings: Notation

- We use the notation:
  - $a < b$, when $(a,b) \in R$
  - $a \,!\!< b$, when $(a,b) \in R$ and $a \neq b$
- The notation $<$ is not to be mistaken for “less than”
- The notation $<$ is used to denote any partial ordering

Comparability: Definition

- **Definition**:
  - The elements a and b of a poset $(S, <)$ are called <u>comparable</u> if either $a<b$ or $b<a$.
  - When for $a,b \in S$, we have neither $a<b$ nor $b<a$, we say that a,b are <u>incomparable</u>
- Consider again our renovation example
  - Remove Asbestos $< a_i$ for all activities a_i except assign offices
  - Paint walls < Refinish floors
  - Some tasks are incomparable: Replacing windows can be done before, after, or during the assignment of offices

Total orders: Definition

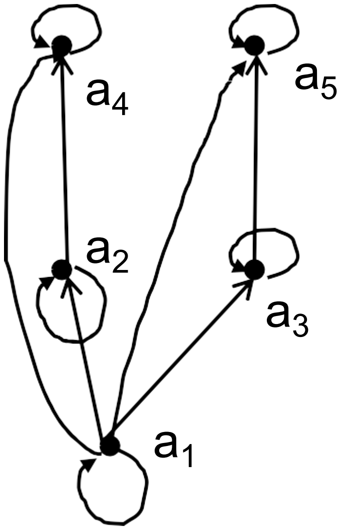
- **Definition**:
  - If $(S,<)$ is a poset and every two elements of S are comparable, S is called a <u>totally ordered set</u>.
  - The relation < is said to be a <u>total order</u>
- Example
  - The relation “less than or equal to” over the set of integers $(Z, \leq)$ since for every $a,b \in Z$, it must be the case that $a \leq b$ or $b \leq a$
  - What happens if we replace $\leq$ with $<$?

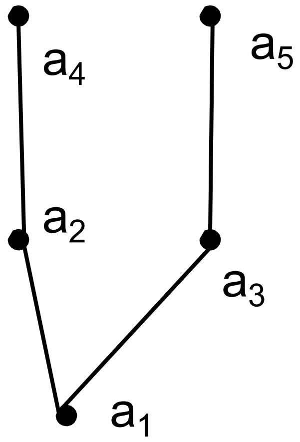
The relation < is not reflexive, and $(Z,<)$ is not a poset

Hasse Diagrams

- Like relations and functions, partial orders have a convenient graphical representation: Hasse Diagrams
  - Consider the <u>digraph</u> representation of a partial order
  - Because we are dealing with a partial order, we know that the relation must be reflexive and transitive
  - Thus, we can simplify the graph as follows
    - Remove all self loops
    - Remove all transitive edges
    - Remove directions on edges assuming that they are oriented upwards
  - The resulting diagram is far simpler

Hasse Diagram: Example

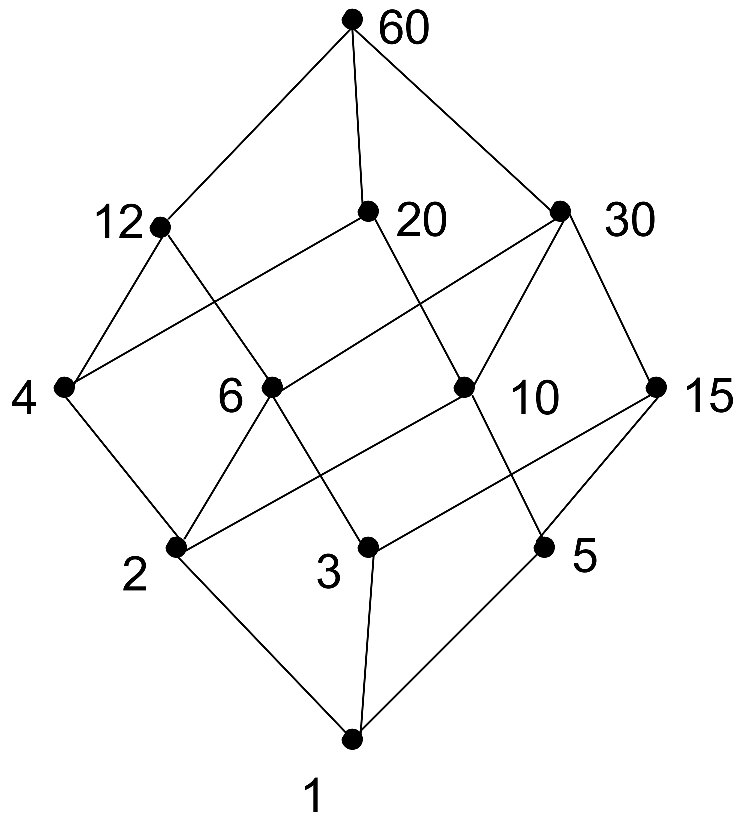




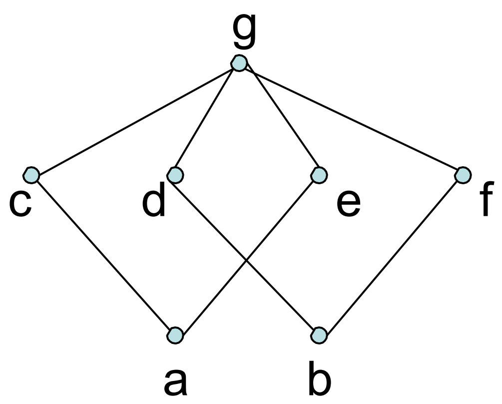
Hasse Diagrams: Example (1)

- Of course, you need not always start with the complete relation in the partial order and then trim everything.
- Rather, you can build a Hasse Diagram directly from the partial order
- Example: Draw the Hasse Diagram
    - for the following partial ordering: {(a,b) | a|b }
  - on the set {1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60}
    - (these are the divisors of 60 which form the basis of the ancient Babylonian base-60 numeral system)

Hasse Diagram: Example (2)

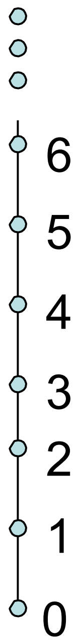


Example



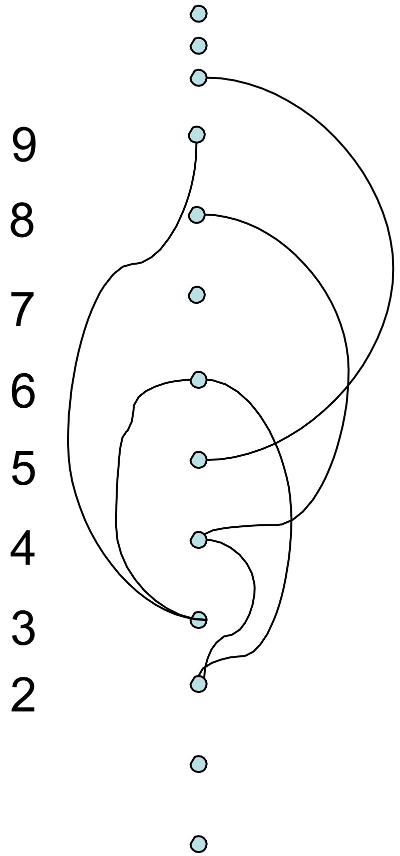
- L= {a,b,c,d,e,f,g}
- $< = \{(a,c), (a,e), (b,d), (b,f), (c,g), (d,g), (e,g), (f,g)\}^{RT}$
- (L, <) is a partial order

Example



- L= N (natural numbers)
- $< \; = \{(0,1), (1,2), (2,3), (3,4), (4,5), \ldots\}^{RT}$
- (L, <) is a totally ordered set (infinite)

Example

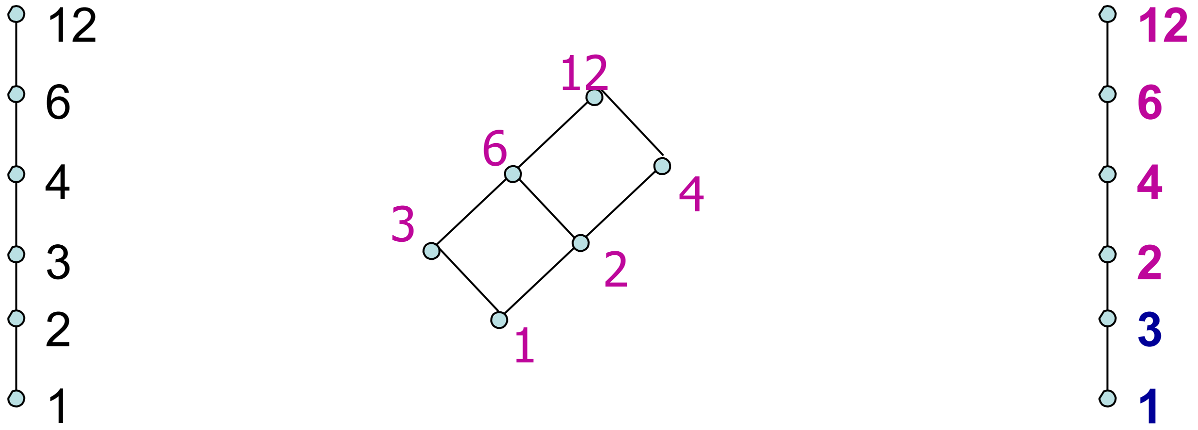


- L= N (natural numbers)
- < ={(n,m): ∃ k such that m=n*k}

- (L, <) is a partially ordered set (infinite)

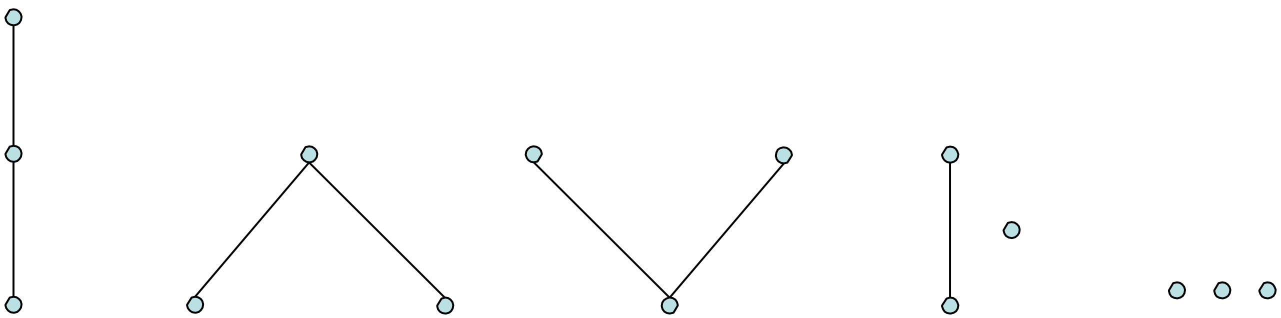
Example

- On the same set E={1,2,3,4,6,12} we can define different partial orders:



Example

- All possible partial orders on a set of three elements (modulo renaming)



Extremal Elements: Summary

We will define the following terms:

- A maximal/minimal element in a poset $(S, <)$
- The maximum (greatest)/minimum (least) element of a poset $(S, <)$
- An upper/lower bound element of a subset A of a poset $(S, <)$
- The greatest lower/least upper bound element of a subset A of a poset $(S, <)$

Extremal Elements: Maximal

- **Definition**: An element a in a poset (S, $<$) is called <u>maximal</u> if it is not less than any other element in S. That is: $\neg(\exists\ b \in S\ (a<b))$
- If there is one <u>unique</u> maximal element a, we call it the <u>maximum</u> element (or the <u>greatest</u> element)

Extremal Elements: Minimal

- **Definition**: An element a in a poset (S, <) is called minimal if it is not greater than any other element in S. That is: $\neg(\exists\ b \in S\ (b<a))$
- If there is one unique minimal element a, we call it the minimum element (or the least element)

Extremal Elements: Upper Bound

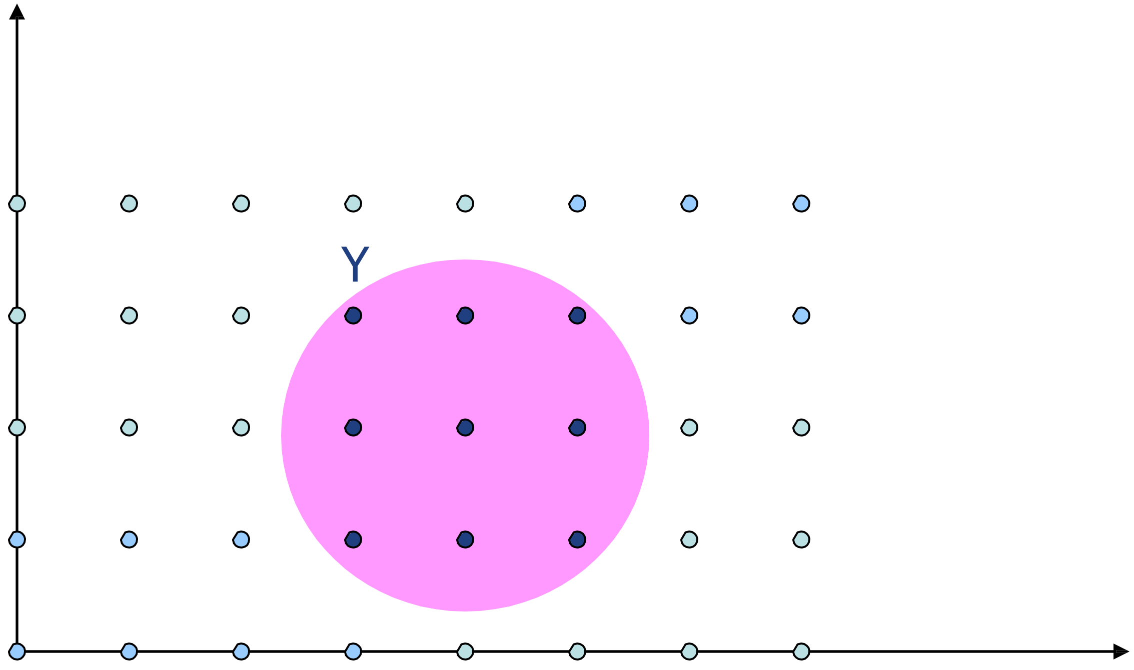
- **Definition**: Let $(S,<)$ be a poset and let $A \subseteq S$. If u is an element of S such that $a < u$ for all $a \in A$ then u is an upper bound of A
- An element x that is an upper bound on a subset A and is less than all other upper bounds on A is called the least upper bound on A. We abbreviate it as lub.

Extremal Elements: Lower Bound

- **Definition**: Let $(S,<)$ be a poset and let $A \subseteq S$. If l is an element of S such that $l < a$ for all $a \in A$ then l is an <u>lower bound of A</u>
- An element x that is a lower bound on a subset A and is greater than all other lower bounds on A is called the <u>greatest lower bound on A</u>. We abbreviate it glb.

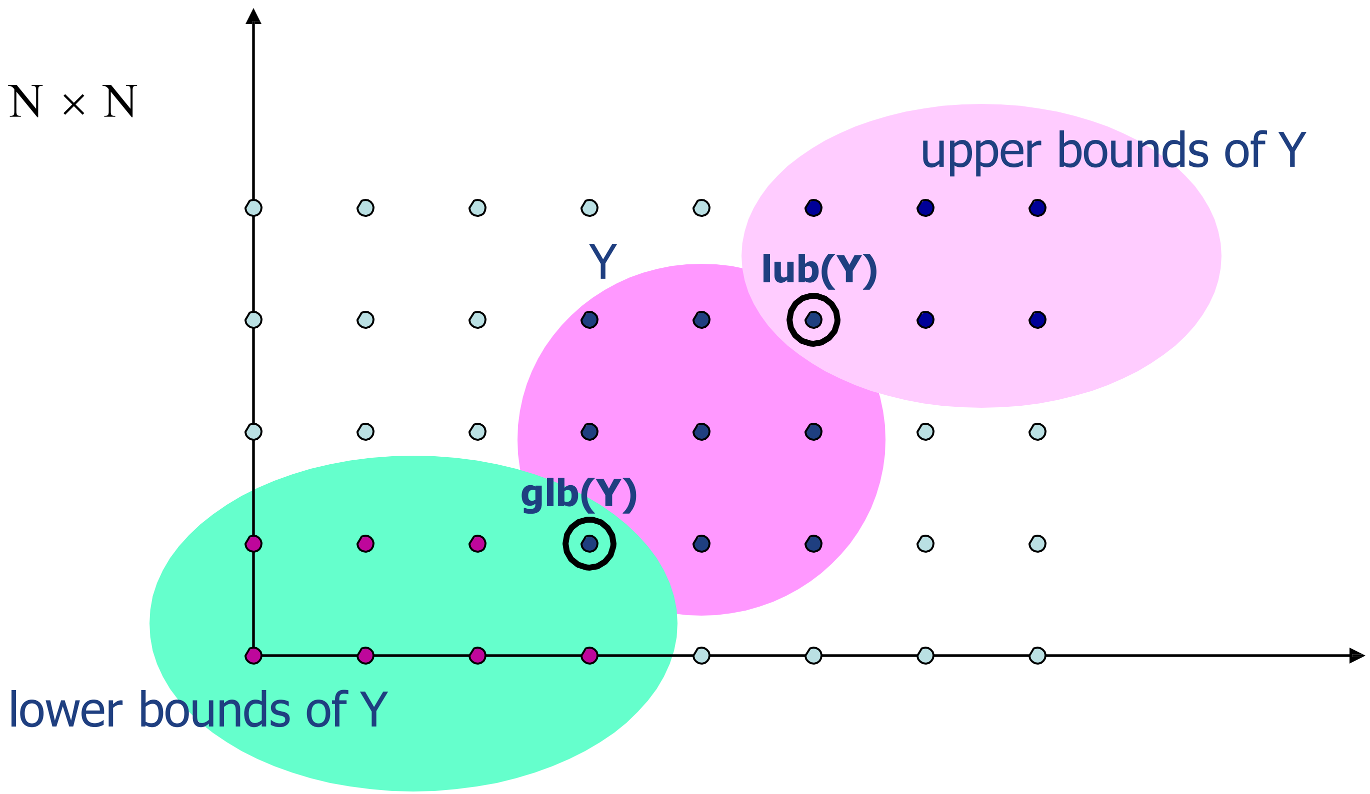
Example

N x N



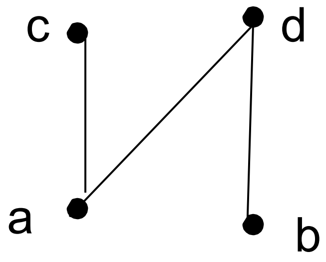
$(x_1,y_1) \leq_{N \times N} (x_2,y_2) \Leftrightarrow x_1 \leq_N x_2 \wedge y_1 \leq_N y_2$

Example



$$(x_1,y_1) \leq_{N \times N} (x_2,y_2) \iff x_1 \leq_N x_2 \wedge y_1 \leq_N y_2$$

Extremal Elements: Example 1



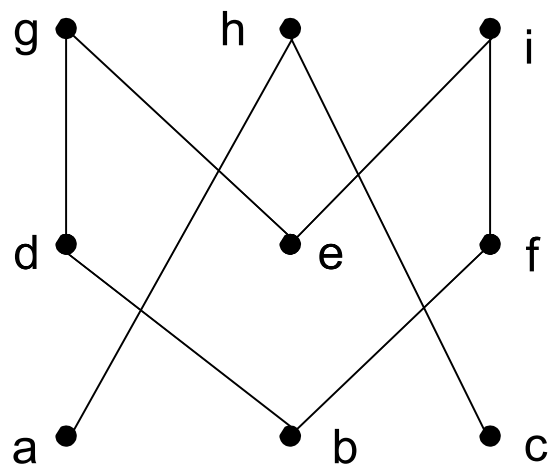
What are the minimal, maximal, minimum, maximum elements?

- Minimal: {a,b}
- Maximal: {c,d}
- There are no unique minimal or maximal elements, thus no minimum or maximum

Extremal Elements: Example 2

Give lower/upper bounds & glb/lub of the sets:

{d,e,f}, {a,c} and {b,d}



{d,e,f}

- Upper bounds: ∅, thus no lub

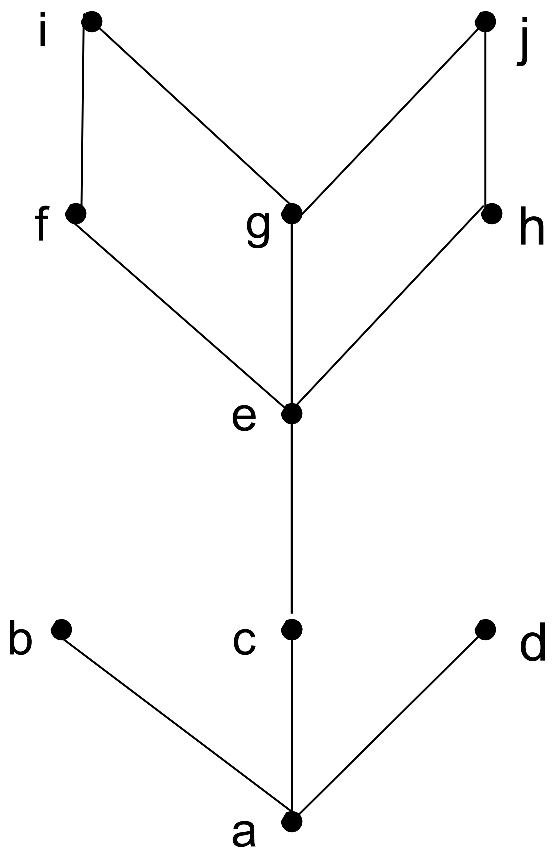
{a,c}

- Lower bounds: ∅, thus no glb
- Upper bounds: {h}, lub: h

{b,d}

- Lower bounds: {b}, glb: b
- Upper bounds: {d,g}, lub: d because $d \preceq g$

Extremal Elements: Example 3



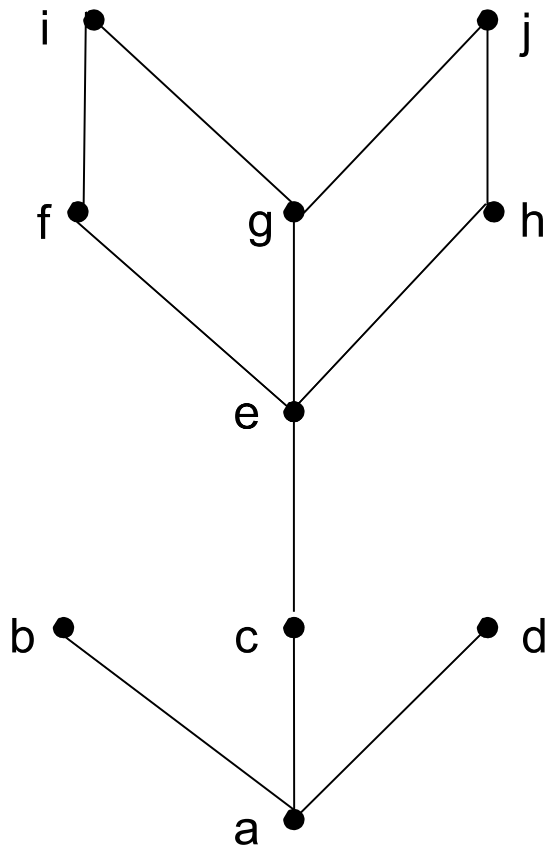
- Minimal/Maximal elements?
  - Minimal & Minimum element: a
  - Maximal elements: b,d,i,j
- Bounds, glb, lub of {c,e}?
  - Lower bounds: {a,c}, thus glb is c
  - Upper bounds: {e,f,g,h,i,j}, thus lub is e
- Bounds, glb, lub of {b,i}?
  - Lower bounds: {a}, thus glb is a
  - Upper bounds: ∅, thus lub Does not exist

Lattices

- A special structure arises when <u>every</u> pair of elements in a poset has an lub and a glb
- **Definition**: A <u>lattice</u> is a partially ordered set in which <u>every</u> pair of elements has both
  - a least upper bound and
  - a greatest lower bound

Lattices: Example 1

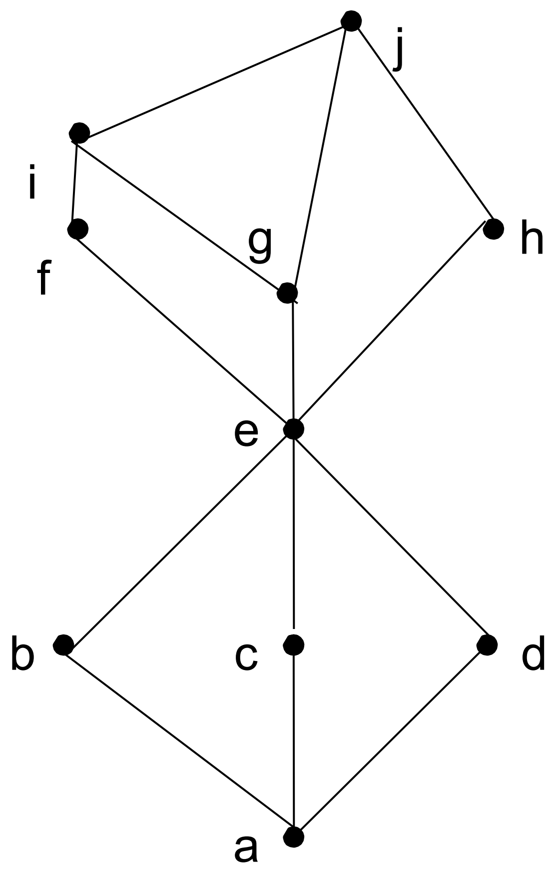
- Is this example from a lattice?



- **No, because the pair {b,c} does not have a least upper bound**

Lattices: Example 2

- What if we modified it as shown here?



- **Yes, because for any pair, there is an lub & a glb**

A Lattice Or Not a Lattice?

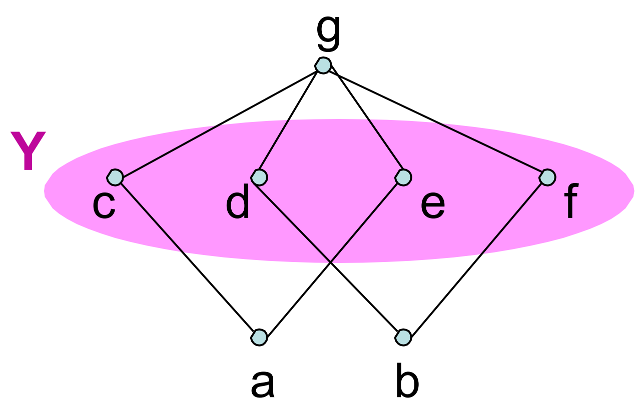
- To show that a partial order is not a lattice, it suffices to find a pair that does not have an lub or a glb (i.e., a counter-example)
- For a pair not to have an lub/glb, the elements of the pair must first be <u>incomparable</u>
- You can then view the upper/lower bounds on a pair as a sub-Hasse diagram: If there is no maximum/minimum element in this sub-diagram, then it is not a lattice

Complete lattices

- <u>Definition</u>:
  A lattice A is called a complete lattice if every subset S of A admits a glb and a lub in A.

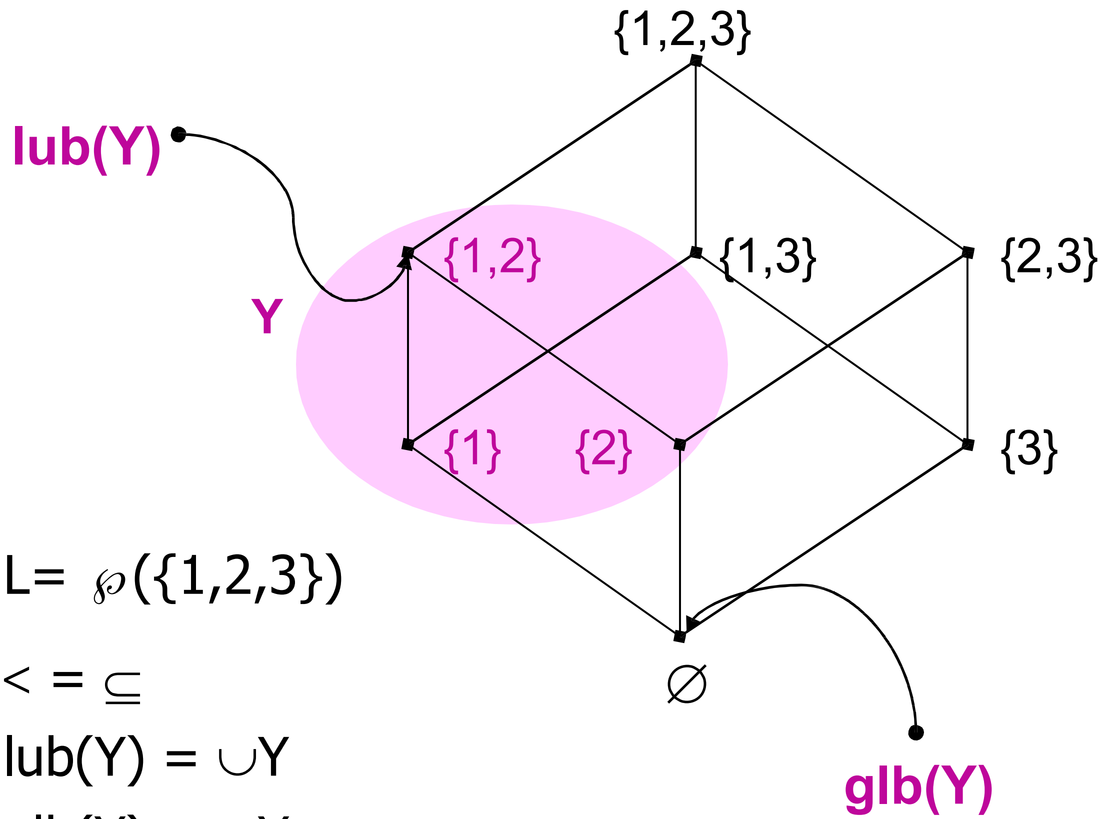
- <u>Exercise</u>:
  Show that for any (possibly infinite) set E, $(P(E),\subseteq)$ is a complete lattice
  (P(E) denotes the powerset of E, i.e. the set of all subsets of E).

Example



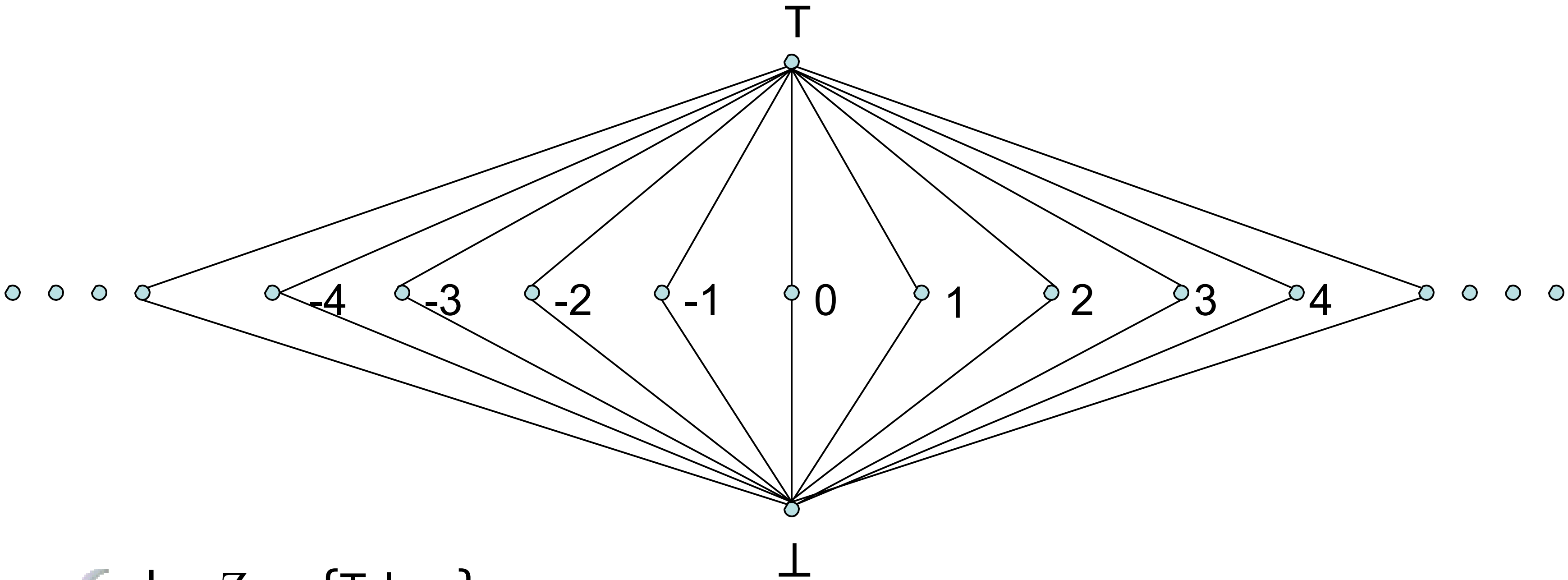
- L= {a,b,c,d,e,f,g}
- $\leq$ ={(a,c), (a,e), (b,d), (b,f), (c,g), (d,g), (e,g), (f,g)}^{T}
- (L,$\leq$) is not a lattice:
  a and b are lower bounds of Y, but a and b are not comparable

Example



- L= $\wp(\{1,2,3\})$
- $< \; = \; \subseteq$
- lub(Y) = $\cup$Y
- glb(Y) = $\cap$Y

Example

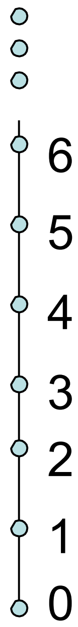


- L= $Z \cup \{T, \perp\}$
- For all $n \in Z : \perp < n < T$

Example

- L= Z_+
- $<$ total order on Z_+
- lub = max
- glb = min

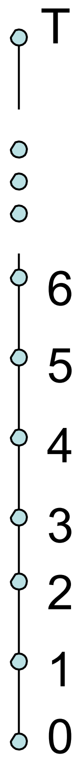
It is a lattice, but not complete:
For instance, the set of even numbers has no lub



Example

- L= $Z_+ \cup \{T\}$
- $<$ total order on $Z_+ \cup \{T\}$
- lub = max
- glb = min

This is a complete lattice



- Theorem:
  Let (L, <) be a partial order. The following conditions are equivalent:
  1. L is a complete lattice
  2. Each subset of L has a least upper bound
  3. Each subset of L has a greatest lower bound

- Proof:
  - $1 \rightarrow 2$ and $1 \rightarrow 3$ by definition
  - In order to prove that $2 \rightarrow 1$, let us define for each $Y \subseteq L$
    $glb(Y) = lub(\{l \in L \mid \text{for all } l' \in Y : l \leq l'\})$

Functions on partial orders

- Let $(P, \leq_P)$ and $(Q, \leq_Q)$ two partial orders. A function φ from P to Q is said:

  monotone (order preserving)
  if $p_1 \leq_P p_2 \Rightarrow \varphi(p_1) \leq_Q \varphi(p_2)$

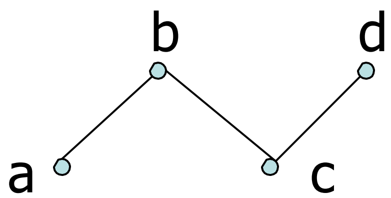
-

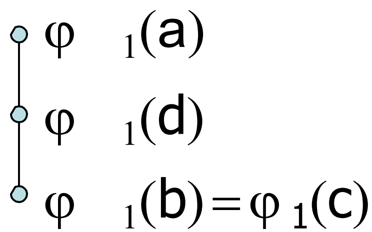
- embedding if

  $p_1 \leq_P p_2 \Leftrightarrow \varphi(p_1) \leq_Q \varphi(p_2)$

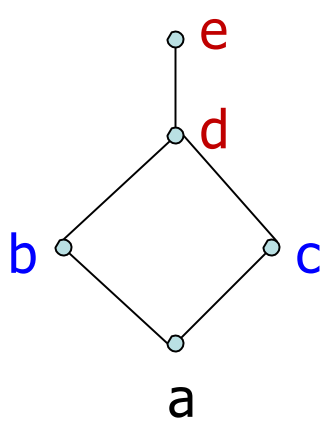
- Isomorphism if it is a surjective embedding

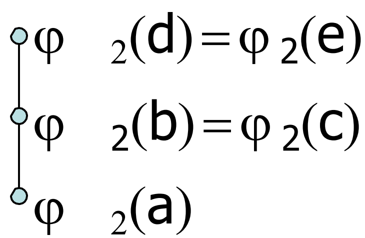
Examples





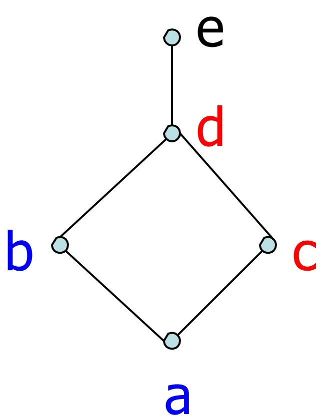
φ_1 is not monotone

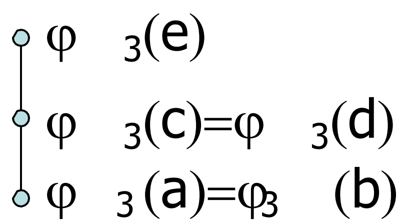




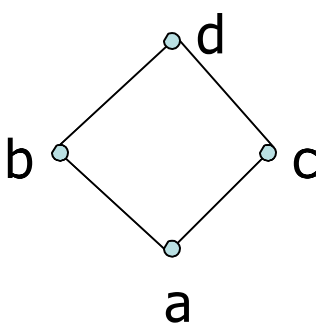
φ_2 is monotone, but it is not an embedding: $\varphi_2(b) \leq_Q \varphi_2(c)$ but it is not true that $b \leq_P c$

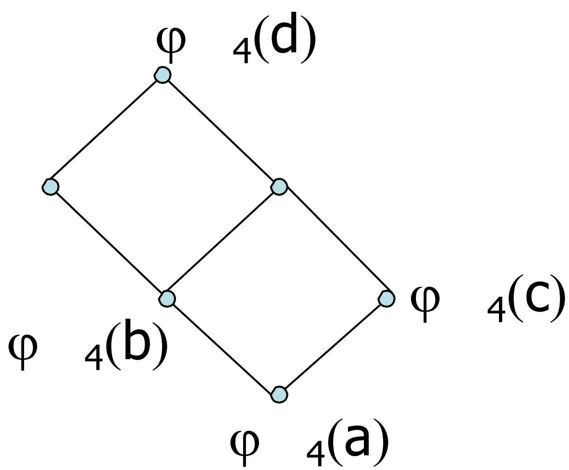
Examples





φ_3 is monotone but it is not an embedding: $\varphi_3(b) \leq_Q \varphi_3(c)$ but it is not true that $b \leq_P c$





φ_4 is an embedding, but not an isomorphism.

Ascending chains

- A sequence $(l_n)_{n\in\mathbb{N}}$ of elements in a partial order L is an ascending chain if

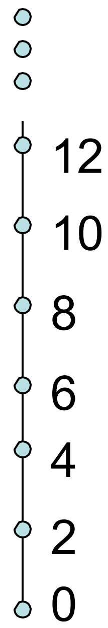
$$n \leq m \Rightarrow l_n \leq l_m$$

- A sequence $(l_n)_{n\in\mathbb{N}}$ converges if and only if

$$\exists \; n_0 \in \mathbb{N} : \text{for all } n \in \mathbb{N} : n_0 \leq n \Rightarrow l_n = l_{n_0}$$

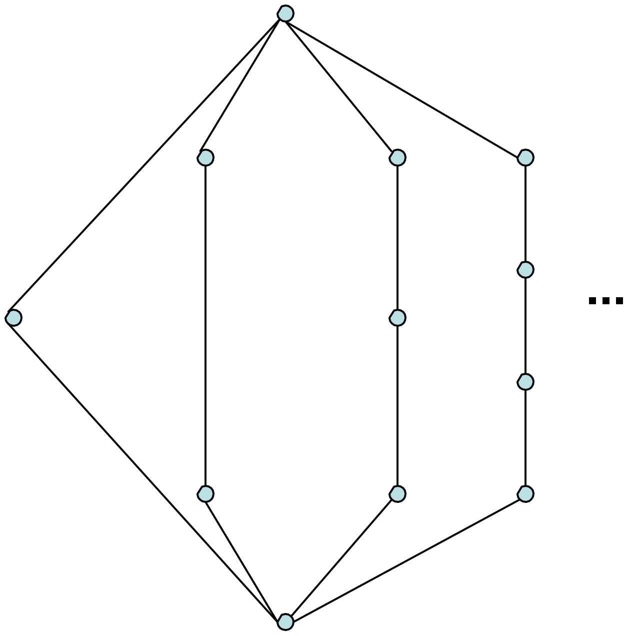
- A partial order $(L,\leq)$ satisfes the ascending chain condition (ACC) iff each ascending chain converges.

Example



- The set of even natural numbers satisfies the descending chain condition, but not the ascending chain condition

Example



- Infinite set
- Satisfies both ACC and DCC

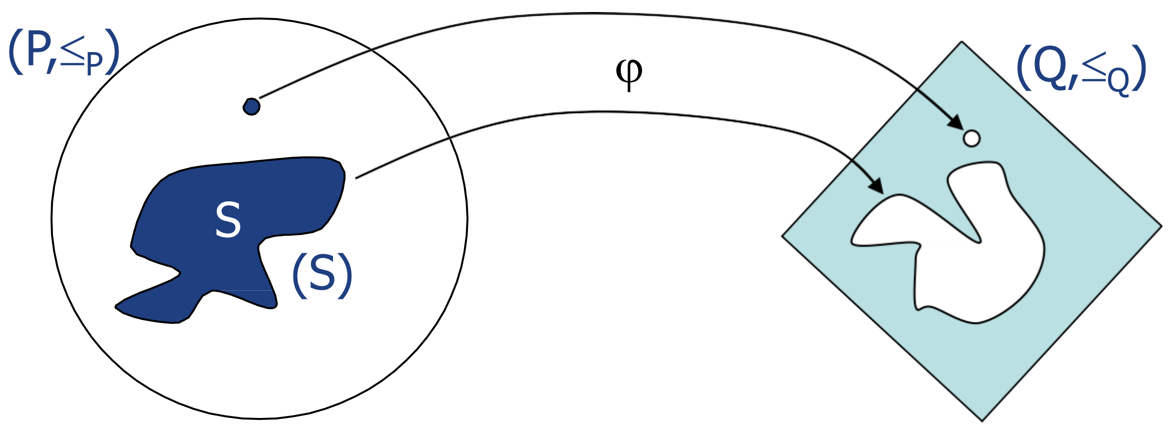
Lattices and ACC

- If P is a lattice, it has a bottom element and satisfies ACC, then it is a complete lattice
-
-
- If P is a lattice without infinite chains, then it is complete

Continuity

- In Calculus, a function is continuous if it preserves the limits.
- Given two partial orders $(P,\leq_P)$ and $(Q,\leq_Q)$, a function φ from P to Q is continuous if for every chain S in P

$$\varphi(lub(S)) = lub\{\varphi(x) \mid x \in S\}$$



Fixpoints

- Consider a monotone function f: $(P,\leq_P) \rightarrow (P,\leq_P)$ on a partial order P.
-
- An element x of P is a fixpoint of f if f(x)=x.
- The set of fixpoints of f is a subset of P called Fix(f):

$$Fix(f) = \{ I \in P \mid f(I)=I \}$$

Fixpoint on Complete Lattices

- Consider a monotone function $f: L \rightarrow L$ on a complete lattice L.
-
- Fix(f) is also a complete lattice:

$$lfp(f) = glb(Fix(f)) \quad \in Fix(f)$$
$$gfp(f) = lub(Fix(f)) \quad \in Fix(f)$$

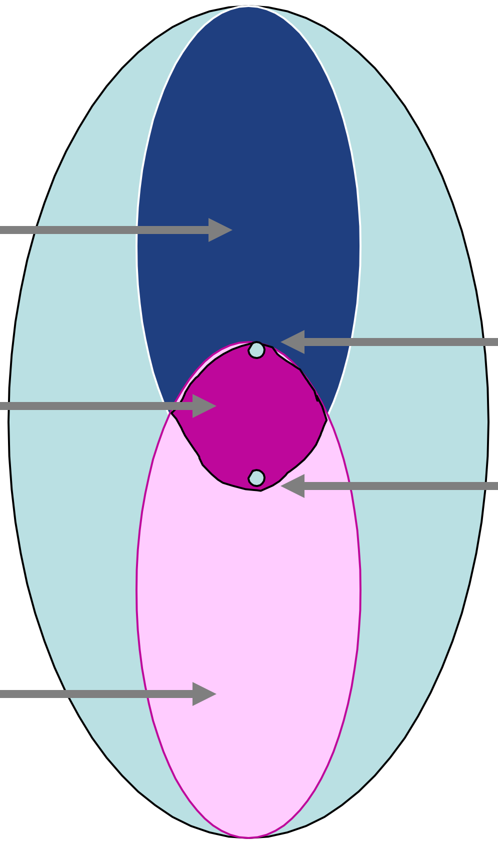
- Tarski Theorem:
  Let L be a complete lattice. If $f: L \rightarrow L$ is monotone then

$$lfp(f) = glb\{ l \in L \mid f(l) \leq l \}$$
$$gfp(f) = lub\{ l \in L \mid l \leq f(l) \}$$

Fixpoints on Complete Lattices

$\{\, l \in L \mid f(l) \leq_P l \,\}$

$\mathrm{Fix}(f) = \{\, l \in L \mid f(l) = l \,\}$



$gfp(f) = lub\{\, l \in L \mid l \leq f(l) \,\}$

$lfp(f) = glb\{\, l \in L \mid f(l) \leq l \,\}$

$\{\, l \in L \mid l \leq_P f(l) \,\}$

Kleene Theorem

- Let f be a monotone function: $(P,\leq_P) \rightarrow (P,\leq_P)$ on a complete lattice P.

Let $\alpha = \bigsqcup_{n\geq 0} f^{n}(\perp)$

  - If $\alpha \in Fix(f)$ then $\alpha = lfp(f)$
  -
  - Kleene Theorem

    If f is continuous then the least fixpoint of f exists , and it is equal to α

Dataflow analysis: what is it?

- A common framework for expressing algorithms that compute information about a program
- Why is such a framework useful?
- It provides a common language, which makes it easier to:
  - communicate your analysis to others
  - compare analyses
  - adapt techniques from one analysis to another
  - reuse implementations (eg: dataflow analysis frameworks)

Data flow analysis

- Goal :
  - collect information about how a procedure manipulates its data
- This information is used in various optimizations
  - For example, knowledge about what expressions are available at some point helps in common subexpression elimination.
- IMPORTANT!
  - Soundness is a must: Data flow analysis should never tell us that a transformation is safe when in fact it is not.
  - It is better to not perform a valid optimization if that one changes the function of the program.

Soundness is a must!

- Data flow analysis should never tell us that a transformation is safe when in fact it is not.

- When doing data flow analysis we must be
  - Conservative
    - Do not consider information that may not preserve the behavior of the program
  - Aggressive
    - Try to collect information that is as exact as possible, so we can get the greatest benefit from our optimizations.

Global Iterative Data Flow Analysis

- Global:
  - Performed on the control flow graph
  - Goal = to collect information at the beginning and end of each basic block
- Iterative:
  - Construct data flow equations that describe how information flows through each basic block and solve them by iteratively converging on a solution.
  - The "ingredients" of the equations:
    - Algebraic representation of the property of interest
    - Labels associated to the control flow diagrams

Global Iterative Data Flow Analysis

- Components of data flow equations
  - Sets containing collected information
    - **In** (or **entry**) set: information coming into the BB from outside (following flow of dats)
    - **gen** set: information generated/collected within the BB
    - **kill** set: information that, due to action within the BB, will affect what has been collected outside the BB
    - **out** (or **exit**) set: information leaving the BB
  - Functions (operations on these sets)
    - **Transfer functions** describe how information changes as it flows through a basic block
    - **Meet functions** describe how information from multiple paths is combined.

Global Iterative Data Flow Analysis

- Algorithm sketch
  - Typically, a bit vector is used to store the information.
    - For example, in reaching definitions, each bit position corresponds to one definition.
  - We use an iterative fixed-point algorithm.
  - Depending on the nature of the problem we are solving, we may need to traverse each basic block in a forward (top-down) or backward direction.
    - The order in which we "visit" each BB is not important in terms of algorithm correctness, but is important in terms of efficiency.
  - In & Out sets should be initialized in a conservative and aggressive way.

```
Initialize gen and kill sets
Initialize in or out sets (depending on "direction")
while there are no changes in in and out sets {
    for each BB {
        apply meet function
        apply transfer function
    }
}
```

Typical problems

- Reaching definitions
  - For each use of a variable, find all definitions that reach it.
- Upward exposed uses
  - For each definition of a variable, find all uses that it reaches.
- Live variables
  - For a point p and a variable v, determine whether v is live at p.
- Available expressions
  - Find all expressions whose value is available at some point p.
- Very Busy expressions
  - Find all expressions whose value will be used in all the next paths

Reaching definitions

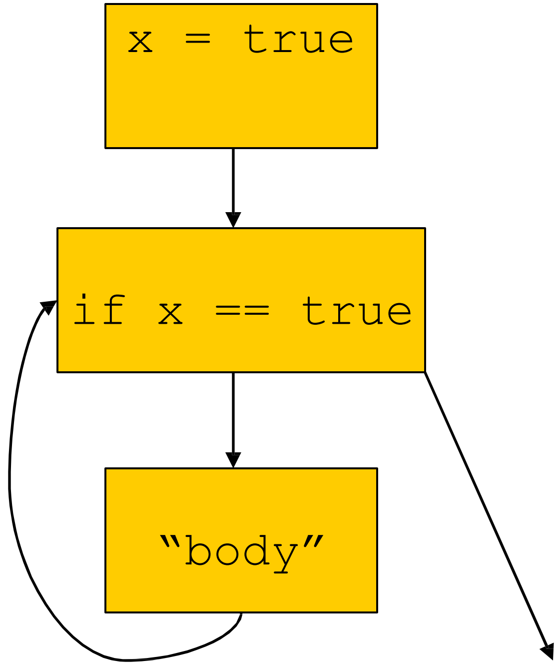
- Determine which <u>definitions</u> of a variable may reach a <u>use</u> of the variable.
  - For each use, list the definitions that reach it. This is also called a ud-chain.
  - In global data flow analysis, we collect such information at the endpoints of a basic block, but we can do additional local analysis within each block.
- Uses of reaching definitions :
  - constant propagation
    - we need to know that all the definitions that reach a variable assign it to the same constant
  - copy propagation
    - we need to know whether a particular copy statement is the only definition that reaches a use.
  - code motion
    - we need to know whether a computation is loop-invariant

Something obvious

```
boolean x = true;
while (x) {
   . . . // no change to x (and no exit/return stm)
}
```

- The program doesn't terminate.
  - Proof: the only assignment to x is at top, so x is always true.

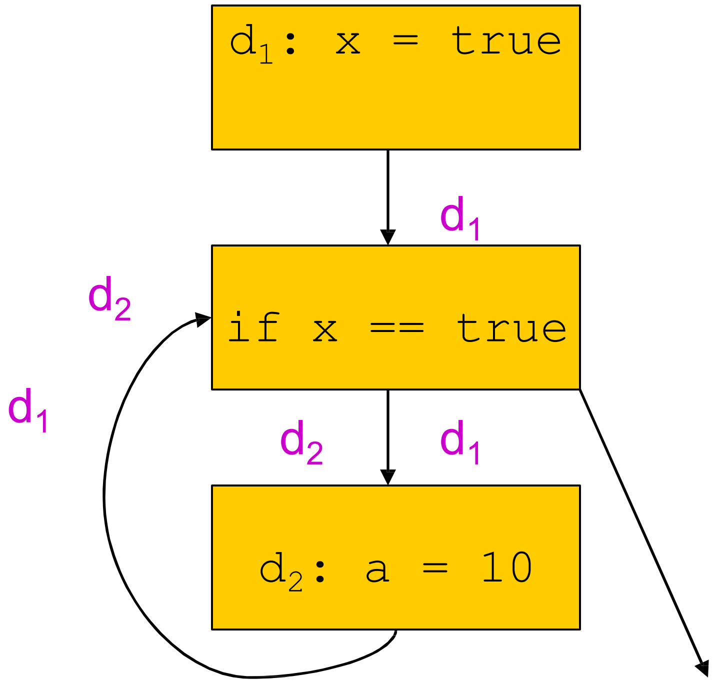
As a Control Flow Graph



Formulation: Reaching Definitions

- At each place, some variable x is assigned a *definition*.
- Ask:
  for this use of x, where could x last have been defined?
- In our example:
  only at
  `x=true`.

Example: Reaching Definitions



Clincher

- Since at `x == true`, d_1 is the only definition of x that reaches, it must be that x is true at that point.
  - The conditional is not really a conditional and can be replaced by a branch.

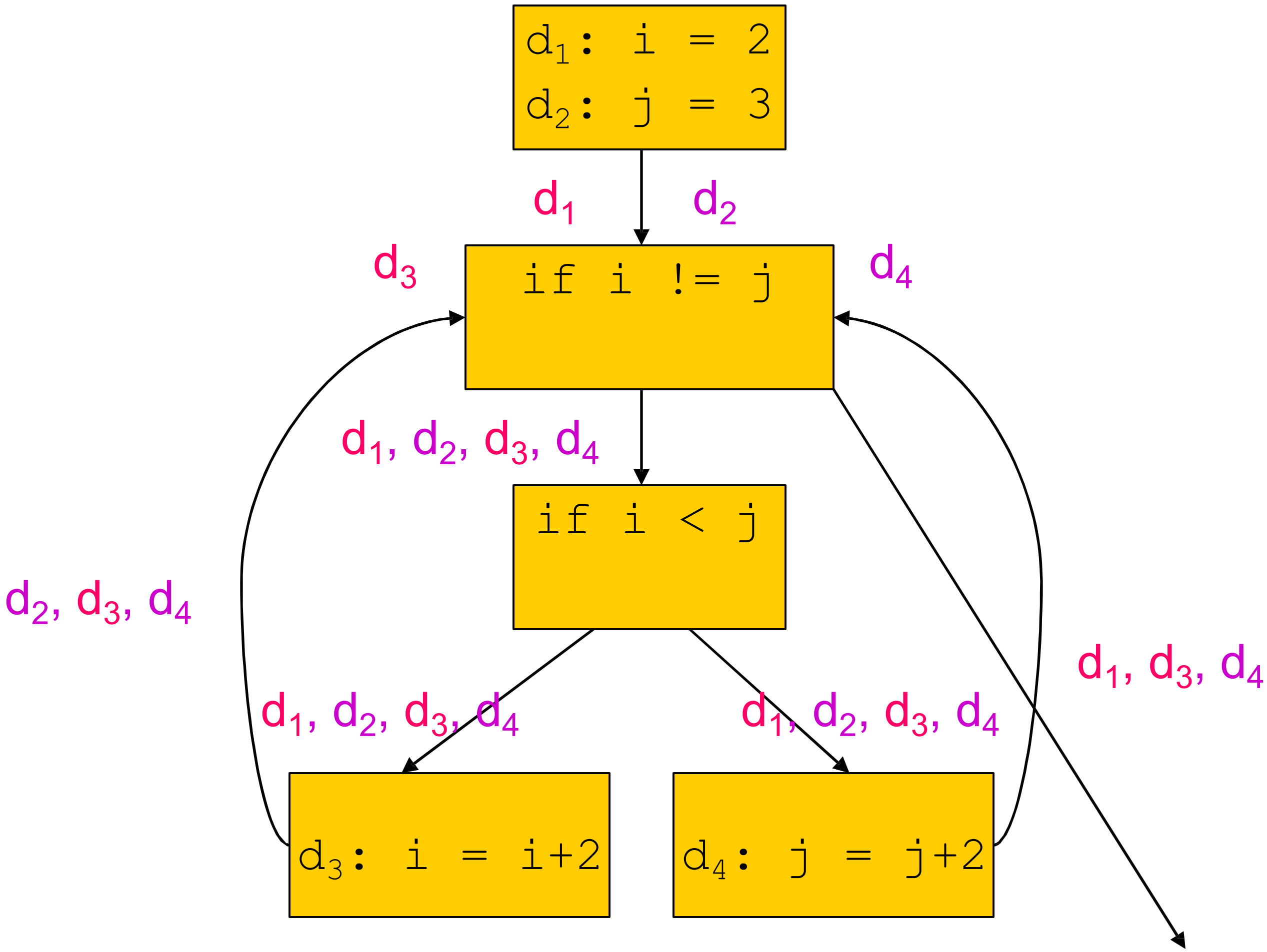
Not Always That Easy

```
int i = 2; int j = 3;
while (i != j) {
    if (i < j) i += 2;
    else j += 2;
}
```

- We'll develop techniques for this problem, but later

    …

The Control Flow Graph



DFA is Sufficient Only

- In this example, i can be defined in two places, and j in two places.
- No obvious way to discover that `i!=j` is always true.
- But OK, because reaching definitions is sufficient to catch most opportunities for *constant folding* (replacement of a variable by its only possible value).

Example: Be Conservative

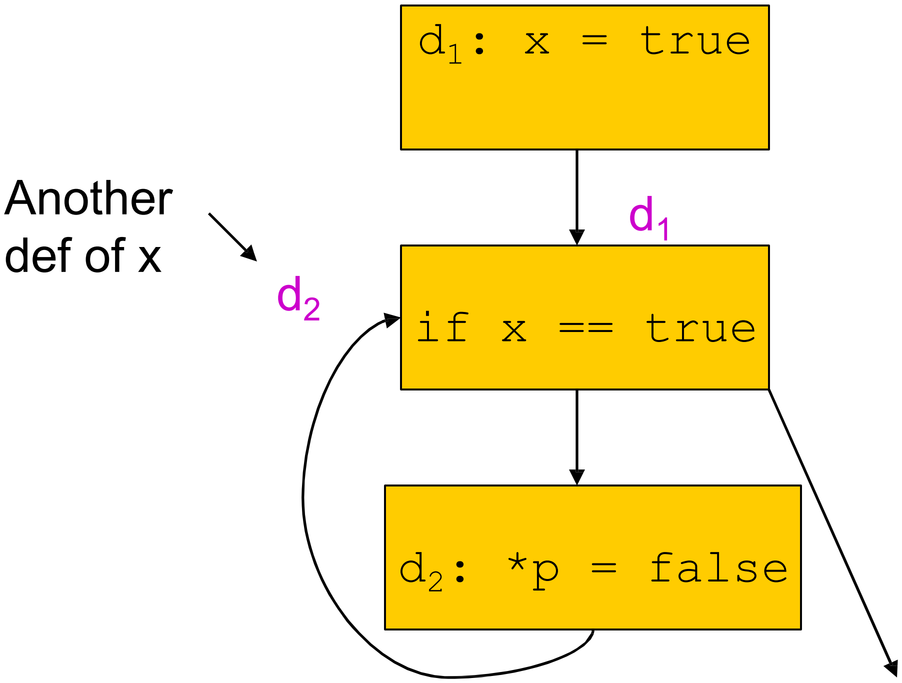
```
boolean x = true;

while (x) {
      . . *p = false; . . .
    .

}
```

- Is it possible that p points to x?

As a Control Flow Graph



Possible Resolution

- Just as data-flow analysis of “reaching definitions” can tell what definitions of x might reach a point, another DFA can eliminate cases where p definitely does not point to x.
- Example: the only definition of p is `p = &y` and there is no possibility that y is an alias of x.