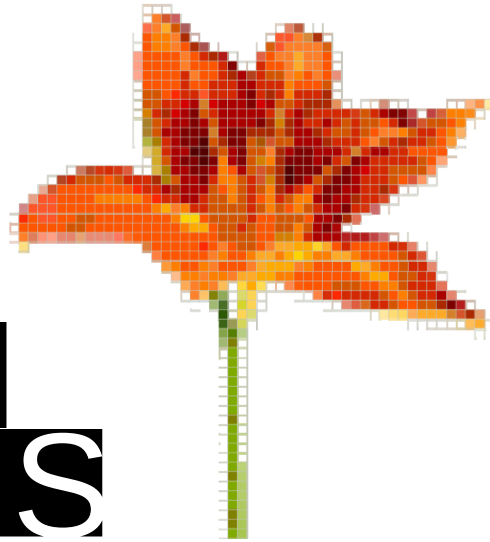
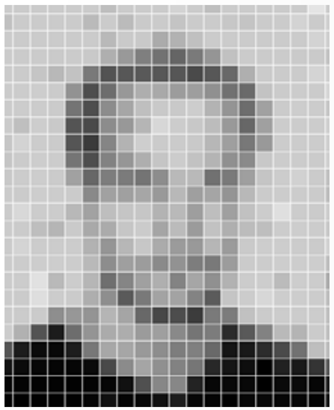
# DATA-FLOW ANALYSIS



[ HOW TO ANALYZE LANGUAGES AUTOMATICALLY ]

Claus Brabrand

((( **brabrand@itu.dk** )))

Associate Professor, Ph.D.

((( Programming, Logic, and Semantics )))

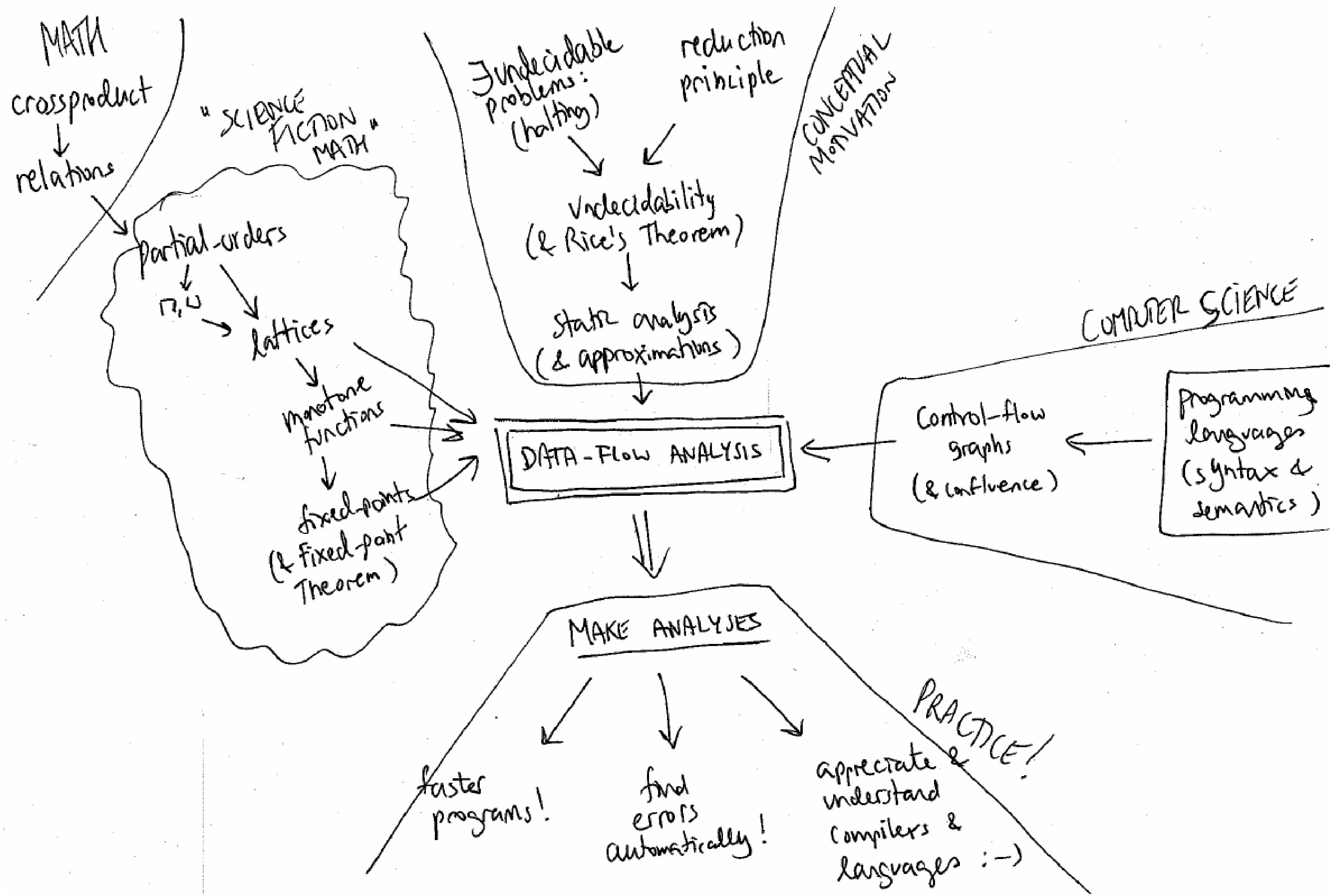
IT University of Copenhagen

# Abstract

- "Data-Flow Analysis":

In this 3*3 hour **mini course** we will look at **data-flow analysis**. Rather than just look at the classical "monotone framework" analyses (which are usually synonymous with teaching data-flow analysis: reaching definitions, live variables, available expressions, and very busy expressions), we will instead take one step backwards and look at the general theory and practice behind these analyses. The idea is that you will then learn how to ***design your own*** customized data-flow analyses for automatically analyzing whatever aspects of programming languages you want to. (From this perspective, the monotone framework analyses are just special cases.)

**Keywords:**
- undecidability, approximation, control-flow graphs, partial-orders, lattices, transfer functions, monotonicity, [how to solve] fixed-point equations – and how all of these things combine to enable you to design data-flow analyses.

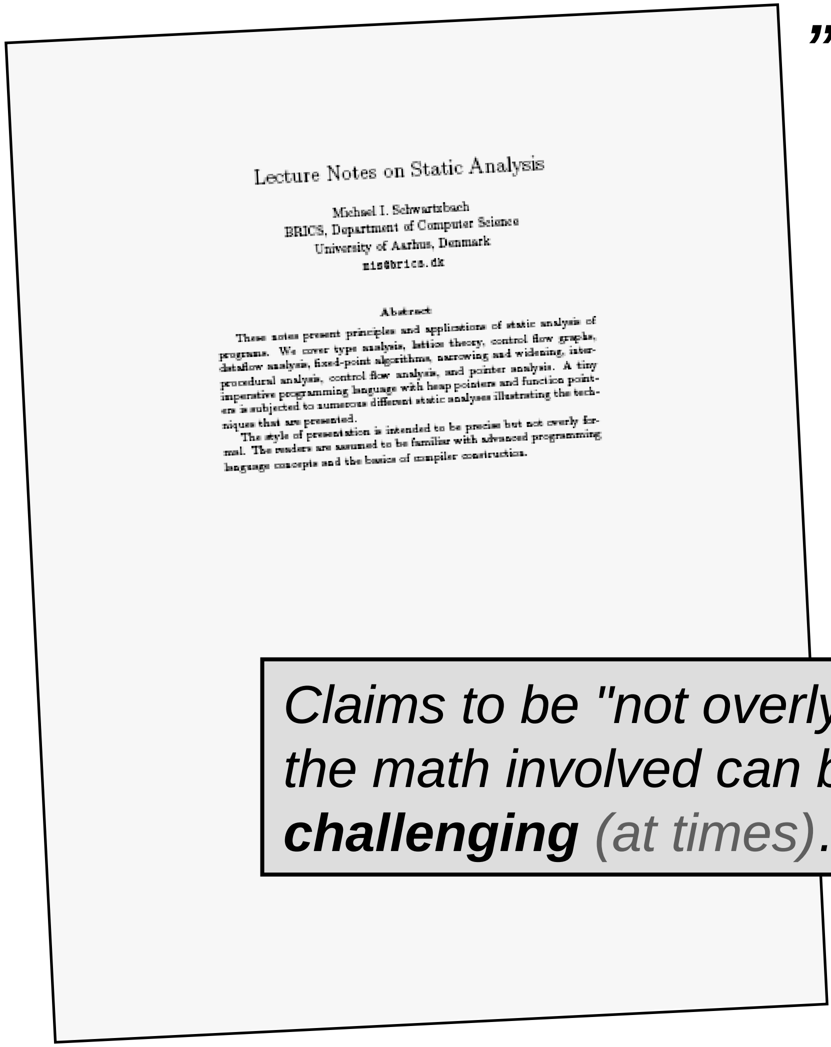


MATH
crossproduct
relations
"SCIENCE FICTION MATH"
partial-orders
⊓, ⊔
lattices
monotone functions
fixed-points
(& Fixed-point Theorem)
∃ Undecidable problems: (halting)
reduction principle
CONCEPTUAL MOTIVATION
Undecidability
(& Rice's Theorem)
Static analysis
(& approximations)
DATA-FLOW ANALYSIS
COMPUTER SCIENCE
Control-flow graphs
(& confluence)
Programming languages
(syntax & semantics)
MAKE ANALYSES
PRACTICE!
faster programs!
find errors automatically!
appreciate & understand compilers & languages :-)

# ⊢ Agenda

- Introduction:
  - Undecidability, Reduction, and Approximation
- Data-flow Analysis:
  - Quick tour of everything & running example
- Control-Flow Graphs:
  - Control-flow, data-flow, and confluence
- ”Science-Fiction Math”: *(next monday)*
  - Lattice theory, monotonicity, and fixed-points
- Putting it all together…: *(next wednesday)*
  - Example revisited

# Notes on Static Analysis



Lecture Notes on Static Analysis

Michael I. Schwartzbach
BRICS, Department of Computer Science
University of Aarhus, Denmark
mis@brics.dk

Abstract

These notes present principles and applications of static analysis of programs. We cover type analysis, lattice theory, control flow graphs, dataflow analysis, fixed-point algorithms, narrowing and widening, interprocedural analysis, control flow analysis, and pointer analysis. A tiny imperative programming language with heap pointers and function pointers is subjected to numerous different static analyses illustrating the techniques that are presented.

The style of presentation is intended to be precise but not overly formal. The readers are assumed to be familiar with advanced programming language concepts and the basics of compiler construction.

## *"Lecture Notes on Static Analysis"*

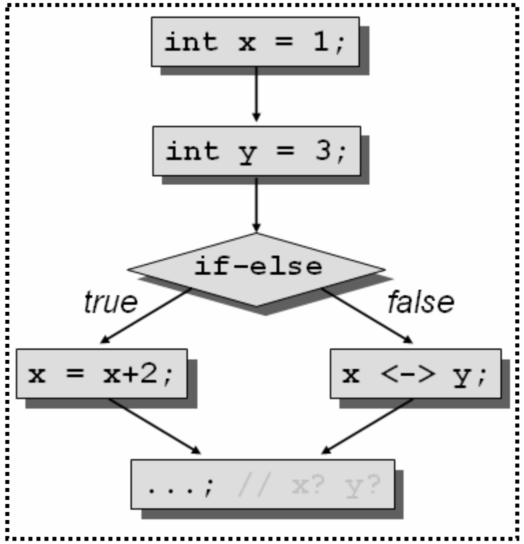
by Michael I. Schwartzbach

(Aarhus University)

Chapter 1, 2, 4, 5, 6 (until p. 19)

(Excl. "pointers")

*Claims to be "not overly formal", but the math involved can be quite* ***challenging*** *(at times)…*



# Quiz: Optimization?

- If you want a ***fast C***-program, should you use:
  - LOOP 1:

```
for (i = 0; i < N; i++) {
    a[i] = a[i] * 2000;
    a[i] = a[i] / 10000;
}
```

?

  - LOOP 2 (optimized by programmer):

***…or…***

```
b = a;
for (i = 0; i < N; i++) {
    *b = *b * 2000;
    *b = *b / 10000;
    b++;
}
```

?

  - i.e., ”array-version” or ”optimized pointer-version” ?

# Answer:

- Results (of running the programs):

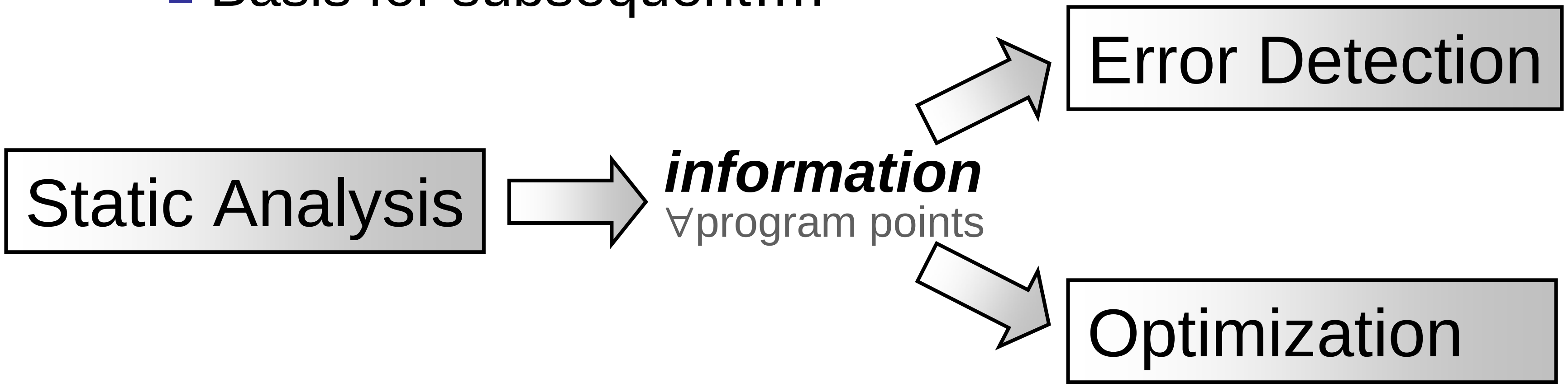
| LOOP | opt. level | SPARC | MIPS | Alpha |
|---|---|---|---|---|
| #1 (array) | no opt | 20.5 | 21.6 | 7.85 |
| #2 (ptr) | no opt | 19.5 | 17.6 | 7.55 |

- Compilers use highly sophisticated static analyses for optimization! (you'll learn how to do this!!!)
  - Recommendation: focus on writing clear code for people **(and compilers)** to understand!

# ⊢ Data-Flow Analysis

- ***Purpose*** (of Data-Flow Analysis):
  - ***Gather information*** (on running behavior of program)
    - ”∀program points”

- ***Usage*** (of static analysis):
  - Basis for subsequent…:



# ⊢ Analyses for Error Detection

- Example Analyses:
  - ***"Symbol Checking":***
    - Catch (dynamic) symbol errors
  - ***"Type Checking":***
    - Catch (dynamic) type errors
  - ***"Initialized Variable Analysis":***
    - Catch unintialized variables
  - …
  - …
  - …
  - …

# ⊢ Analyses for Optimization

- Example Analyses:
  - ***"Constant Propagation Analysis":***
    - Precompute constants (e.g., replace '**`5*x+z`**' by '**`42`**')
  - ***"Live Variables Analysis":***
    - Dead-code elimination (e.g., get rid of unused variable '**`z`**')
  - ***"Available Expressions Analysis":***
    - Avoid recomputing already computed exprs (cache results)
  - …
  - …
  - …
  - …

# Conceptual Motivation

- *Undecidability*
- *Reduction principle*
- *Approximation*

# ⊢ Rice's Theorem (1953)

> *"Any interesting problem about the **runtime behavior** of a program**
> *is undecidable"*
>
> -- Rice's Theorem **[paraphrased]** (1953)

**) written in a turing-complete language*

- Examples:
  - *does program 'P' always halt when run?*
  - *is the value of integer variable 'x' always positive?*
  - *does variable 'x' always have the same value?*
  - *which variables can pointer 'p' point to?*
  - *does expression 'E' always evaluate to true?*
  - *what are the possible outputs of program 'P'?*
  - …

# ⊢ Undecidability (self-referentiality)

- *Consider "The Book-of-all-Books":*
  - This book contains the **titles** of all books that do **not** have a ***self-reference***
    (i.e. don't contain their title inside)
  - Finitely many books; i.e.:
    - We can sit down & figure out whether to include or not...
  - **Q:** What about **`"The Book-of-all-Books"`**;
    - Should it be *included* or *not*?


☺

- *"Self-referential paradox"* (many guises):
  - *e.g. "This sentence is false"*

☺

# ⊢ Termination Undecidable!

- ***Assume*** termination is *decidable* (in Java);
  - i.e. ∃ some program, ***halts*****: Program → bool**

```
bool halts(Program p) { ... }
```

```
-- P0.java --
Program p0 = read_program("P0.java");
if (halts(p0)) loop();
else halt();
```

  - **Q:** Does $P_0$ *loop* or *terminate*...? **:)**
- **Hence: *halts*** cannot exist!
  - *i.e., "Termination is undecidable"* **) for turing-complete languages*

# ⊢ Rice’s Theorem (1953)

> *“Any interesting problem about the runtime behavior program* is undecidable”*
>
> -- Rice’s Theorem **[paraphrased]** (1953)

**) written in a turing-complete language*

- Examples:


  - *does program ’P’ always halt?*
  - *is the value of integer variable ’x’ always positive?*
  - *does variable ’x’ always have the same value?*
  - *which variables can pointer ’p’ point to?*
  - *does expression ’E’ always evaluate to true?*
  - *what are the possible outputs of program ’P’?*
  - *…*

# ⊢ Reduction: solve *always-pos* ⇒ solve *halts*

1) Assume *'x-is-always-pos(P)'* is decidable
2) Given *P* (here's how we could solve 'halts(*P*)'):
3) Construct (veeeeery clever) reduction program *R*:

```
-- R.java --
int x = 1;
P /* insert program P here :-) */
x = -1;
```

4) Run "supposedly decidable" analysis:

   *res* = ***x-is-always-positive*(*R*)**

5) Deduce from result:

   if (*res*) then ***P loops!***; else ***P halts*** :-)

6) THUS: *'x-is-always-pos(P)'* must be ***undecidable!***

# ⊢ Reduction Principle

- Reduction principle (in short):

$$\frac{\phi(P)\ \textbf{undecidable}\ \wedge\ [\textbf{solve}\ \psi(P) \Rightarrow \textbf{solve}\ \phi(P)]}{\psi(P)\ \textbf{undecidable}}$$

- Example:

*reduction*

$$\frac{\text{'halts(P)'}\ \textbf{undecidable}\ \wedge\ [\textbf{solve}\ \text{'x-is-always-pos(P)'} \Rightarrow \textbf{solve}\ \text{'halts(P)'}]}{\text{'x-is-always-pos(P)'}\ \textbf{undecidable}}$$

- Exercise:
  - Carry out reduction + whole explanation for:
    - *"which variables can pointer 'q' point to?"*

# ⊢ Answer

1) Assume *'which-var-q-points-to(P)'* is decidable:
2) Given P (here's how to (cleverly) decide halts(*P*)):
3) Construct (veeeeery clever) reduction program *R*:

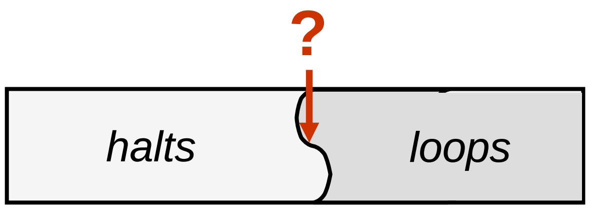
```
-- R.java --
ptr q = 0xffff;
P /* insert program P (assume w/o 'q') */
q = null;
```

4) Run *'which-var-q-points-to(R)'* = res
5) If (null ∈ res) *P* halts! else; *P* loops! :-)
6) <u>THUS</u>:

   *'which-var-q-points-to(P)'* must be ***undecidable!***

# Undecidability

- Undecidability means that…:



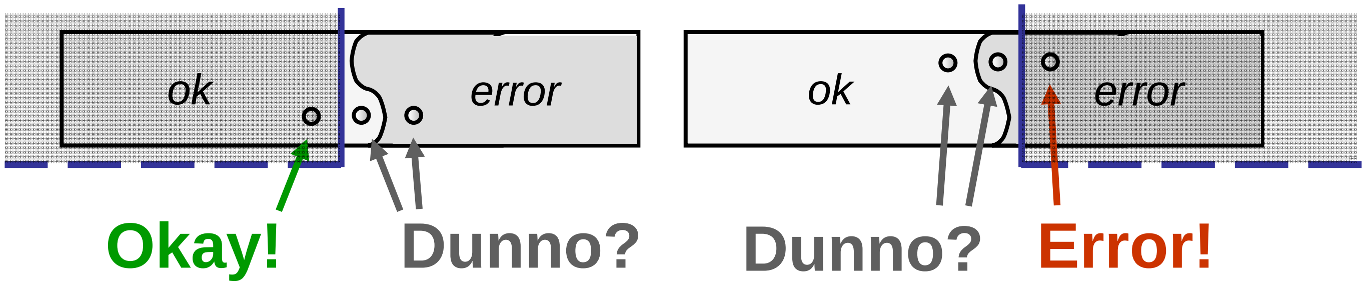

  - *…no-one can decide this **line** (for all programs)!*

- ***However(!)…***

# ⊢ "Side-Stepping Undecidability"

However, just because it's undecidable, doesn't mean there aren't (good) ***approximations***! Indeed, the whole area of static analysis works on "***side-stepping undecidability***":

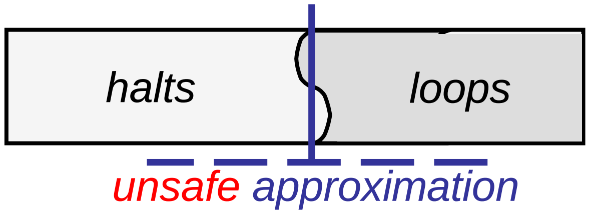
- Compilers use ***safe approximations*** (computed via "static analyses") such that:



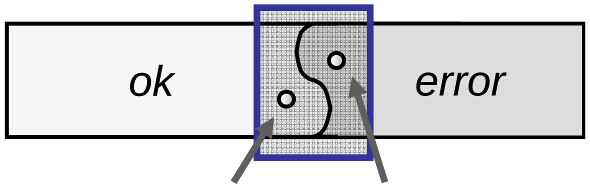
# ⊢ "Side-Stepping Undecidability"

However, just because it's undecidable, doesn't mean there aren't (good) ***approximations***! Indeed, the whole area of static analysis works on "***side-stepping undecidability***":

- ***Unsafe approximation:***



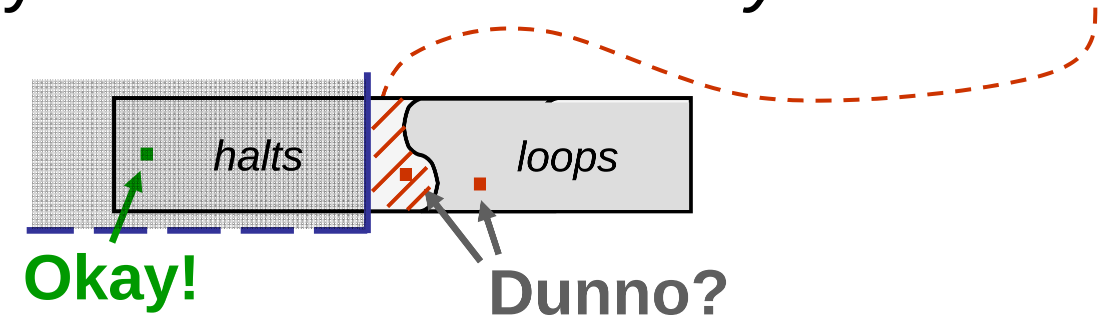

- For ***testing*** it may be okay to "abandon" safety and use ***unsafe approximations:***




**Here are some programs for you to (manually) consider !**

# ⊢ "Slack"

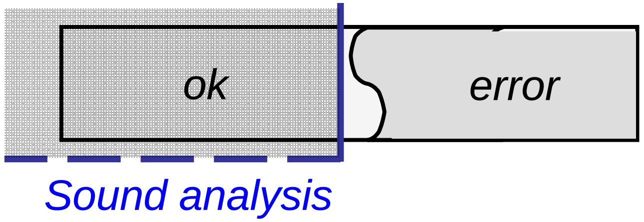
- Undecidability means: *"there'll always be a slack":*




- However, still useful:
  (possible *interpretations* of "**Dunno?**"):
  - Treat as **error** (i.e., *reject program*):
    - *"Sorry, program not accepted!"*
  - Treat as **warning** (i.e., *warn programmer*):
    - *"Here are some **potential** problems: …"*

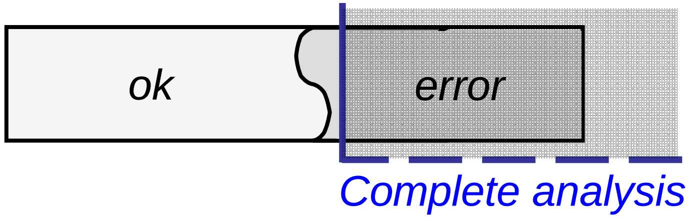
# ⊢ Soundness & Completeness

- ***Soundness:***




- Analysis reports no errors
  ⇒ Really are no errors

- ***Completeness:***




- Analysis reports an error
  ⇒ Really is an error

…or alternative (equivalent) formulation, via "contra-position":

$$P \Rightarrow Q \equiv \neg Q \Rightarrow \neg P$$

- Really are error(s)
  ⇒ Analysis reports error(s)

- Really no error(s)
  ⇒ Analysis reports no error(s)

# ⊢ Example: Type Checking

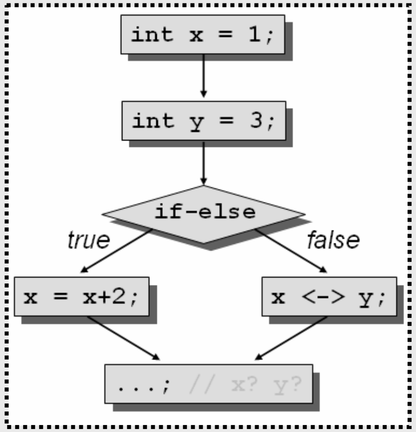
- Will this program have type error (when run)?
  - ```
    void f() {
        var b;
        if (<EXP>) {
            b = 42;
        } else {
            b = true;
        }
        ...
        if (b) ...; // error is b is '42'
    }
    ```
  - **Undecidable** (because of reduction)**:**
    - Type error ⇔ ***<EXP>*** evaluates to **`true`**
    ```

# ⊢ Example: Type Checking

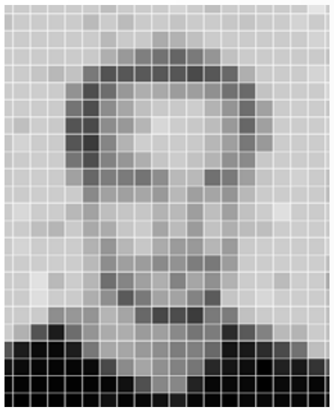
- Hence, languages use ***static requirements:***
  - 
```
void f() {
    bool b; // instead of "var b;"
    if (<EXP>) {
        b = 42;
    } else {
        b = true;
    }
}
```
    **Static compiler error:** Regardless of what <EXP> evaluates to when run
  - All variables must be ***declared***
  - And have ***only one type*** (throughout the program)
  - This is (very) easy to check (i.e., "type-checking")

# ⊢ Agenda

- Introduction:
  - Undecidability, Reduction, and Approximation
- Data-flow Analysis:
  - Quick tour & running example
- Control-Flow Graphs:
  - Control-flow, data-flow, and confluence
- ”Science-Fiction Math”:
  - Lattice theory, monotonicity, and fixed-points
- Putting it all together…:
  - Example revisited




# 5' Crash Course on Data-Flow Analysis




Claus Brabrand
((( brabrand@itu.dk )))

Associate Professor, Ph.D.
((( Programming, Logic, and Semantics )))
IT University of Copenhagen

# ⊢ Data-Flow Analysis

- **IDEA:** *"Simulate runtime execution at compile-time using abstract values"*

- We (only) need 3 things:
  - A ***control-flow graph***
  - A ***lattice***
  - ***Transfer functions***

- Example: *"(integer) constant propagation"*

# ⊢ Control-flow graph

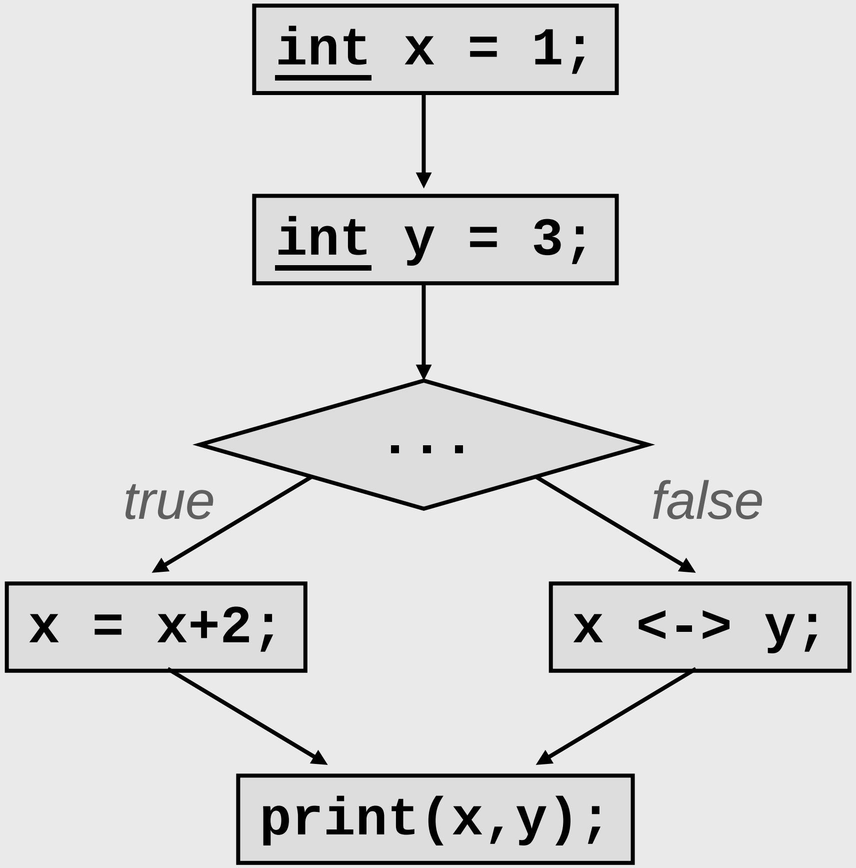
We (only) need 3 things:
- A ***control-flow graph***
- A ***lattice***
- ***Transfer functions***

**Given program:**

```
int x = 1;
int y = 3;

if (...) {
  x = x+2;
} else {
  x <-> y;
}
print(x,y);
```

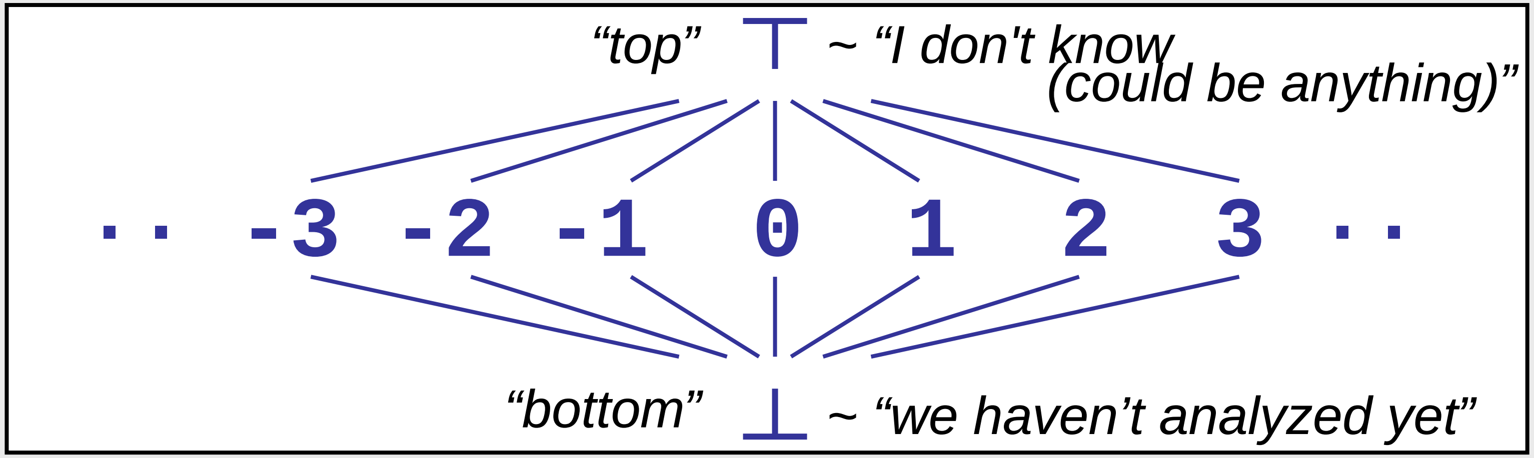
→



# ⊢ A Lattice

We (only) need 3 things:
- A ***control-flow graph***
- A ***lattice***
- ***Transfer functions***

- Lattice ***L*** of ***abstract values*** of interest and their relationships (i.e. ordering "≤"):

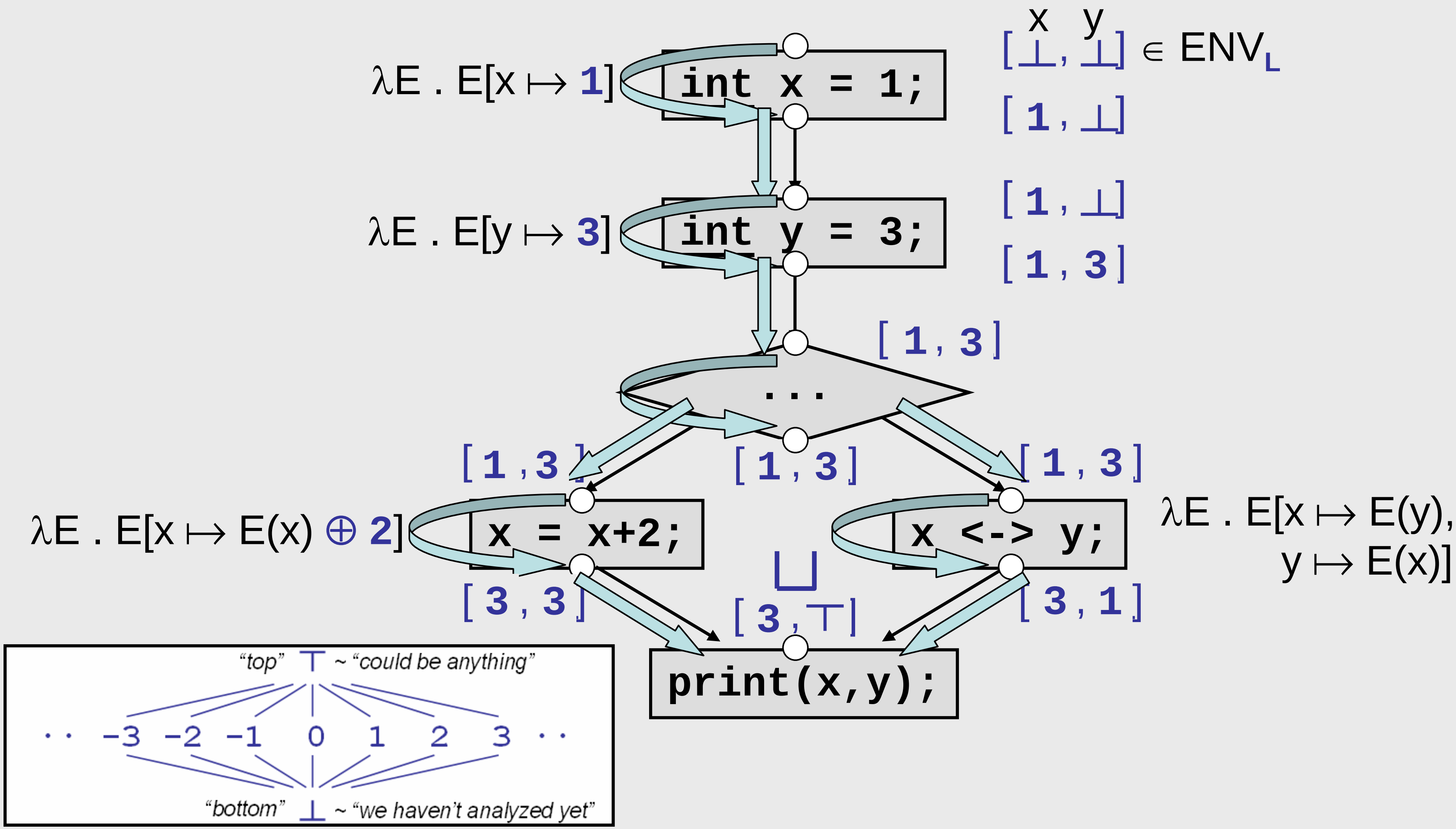



- Induces *least-upper-bound* operator: ⊔
  - for ***combining information***

# ⊢ Data-Flow Analysis

We (only) need 3 things:

- A ***control-flow graph***
- A ***lattice***
- ***Transfer functions***



# ⊢ Agenda

- Introduction:
  - Undecidability, Reduction, and Approximation
- Data-flow Analysis:
  - Quick tour & running example
- Control-Flow Graphs:
  - Control-flow, data-flow, and confluence
- ”Science-Fiction Math”:
  - Lattice theory, monotonicity, and fixed-points
- Putting it all together…:
  - Example revisited

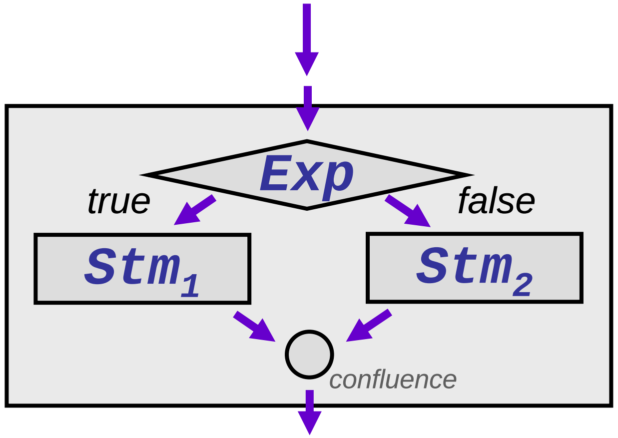
# ⊢ Control Structures

- ***Control Structures:***
  - ***Statements** (or Expr's)* that ***affect** "flow of control":*

- **`if-else`**:

[syntax] ■ `if ( Exp ) Stm₁ else Stm₂`

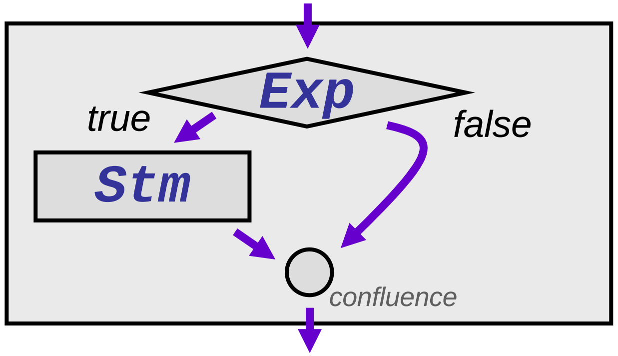
[semantics] ■ *The expression must be of type **boolean**; if it evaluates to **true**, Statement-1 is executed, otherwise Statement-2 is executed.*




- **`if`**:

[syntax] ■ `if ( Exp ) Stm`

[semantics] ■ *The expression must be of type **boolean**; if it evaluates to **true**, the given statement is executed, otherwise not.*



# ⊢ Control Structures (cont'd)

- **`while`:**

[syntax] `while ( Exp ) Stm`

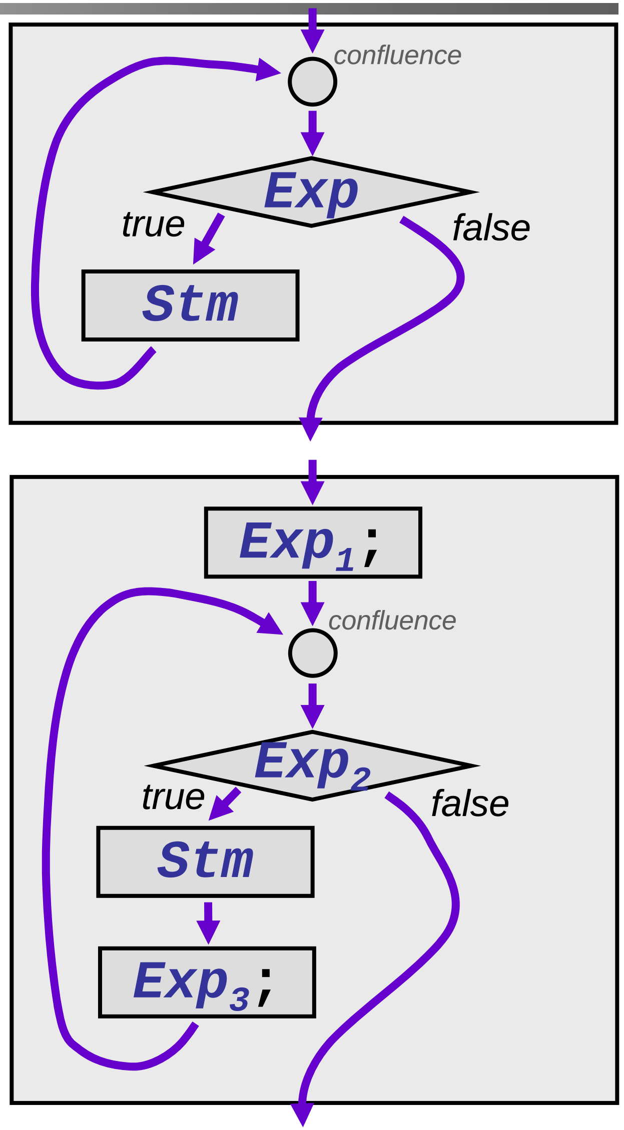
[semantics] *The expression must be of type **boolean**; if it evaluates to **false**, the given statement is skipped, otherwise it is executed and afterwards the expression is evaluated again. If it is still true, the statement is executed again. This is continued until the expression evaluates to **false**.*

- **`for`:**

[syntax] `for (`$Exp_1$ `;` $Exp_2$ `;` $Exp_3$`) Stm`

[semantics] *Equivalent to:*

```
{ Exp1;
  while ( Exp2 ) { Stm  Exp3; }
}
```



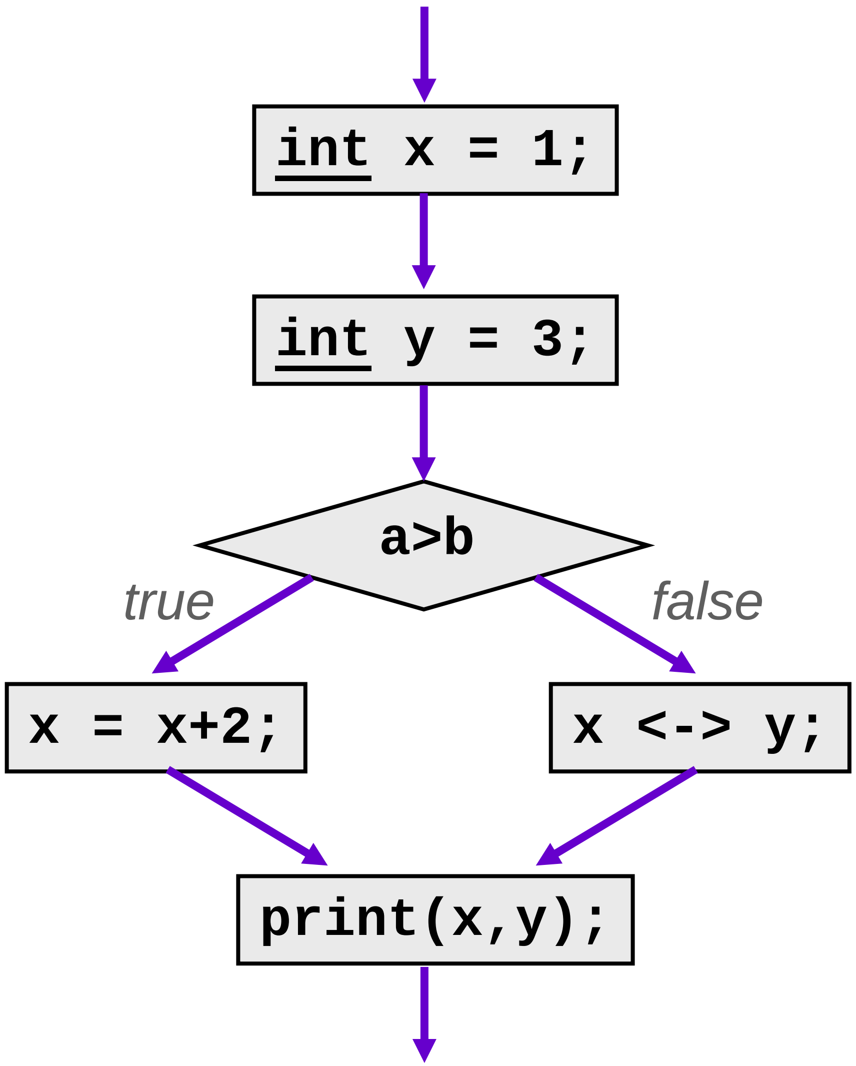
# ⊢ Control-flow graph

**Given program:**

```
int x = 1;
int y = 3;

if (a>b) {
  x = x+2;
} else {
  x <-> y;
}
print(x,y);
```

→



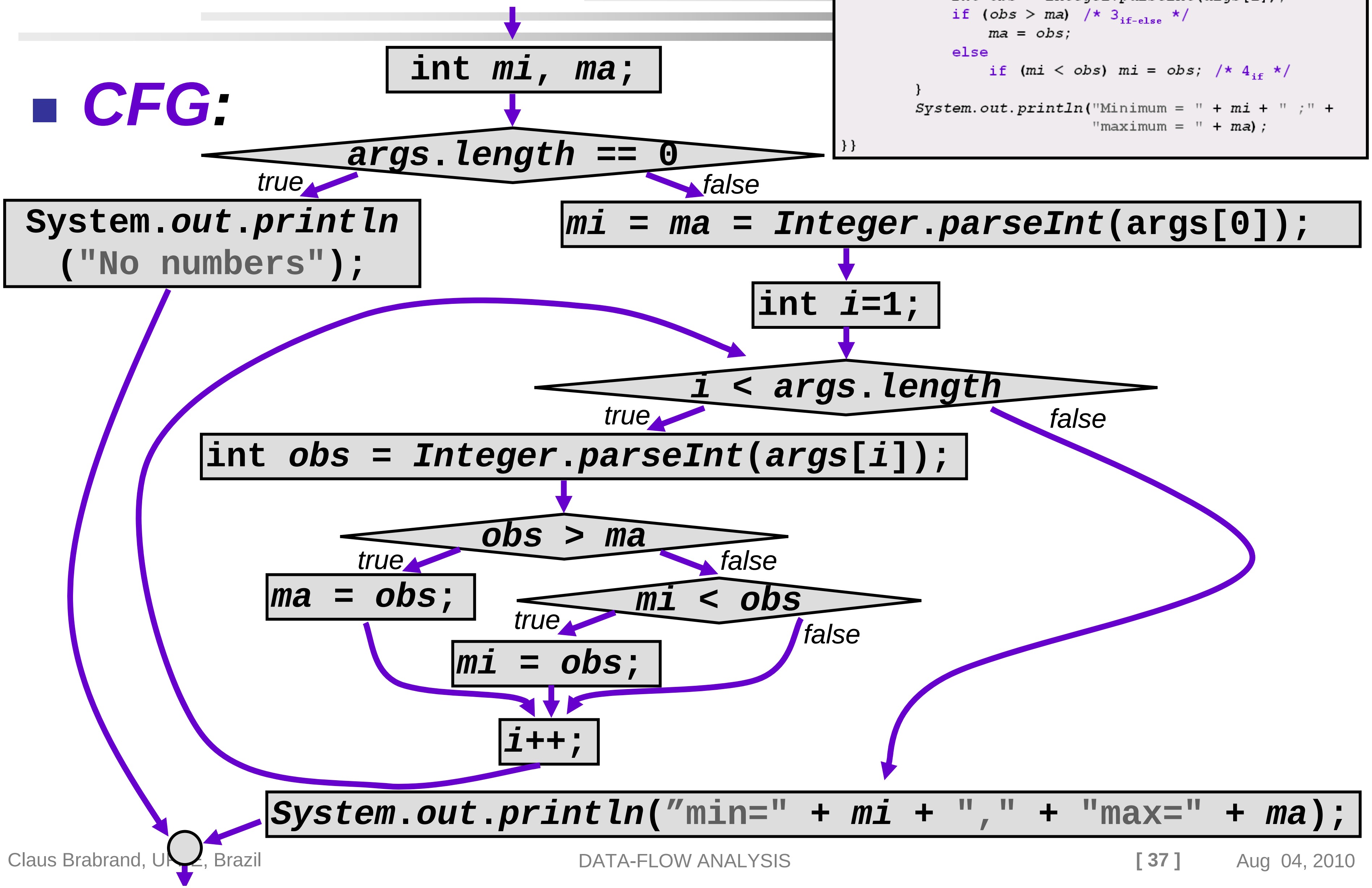
# Exercise: Draw a Control-Flow Graph for:

```
public static void main ( String[] args ) {
    int mi, ma;
    if (args.length == 0)
        System.out.println("No numbers");
    else {
        mi = ma = Integer.parseInt(args[0]);
        for (int i=1; i < args.length; i++) {
            int obs = Integer.parseInt(args[i]);
            if (obs > ma)
                ma = obs;
            else
                if (mi < obs) mi = obs;
        }
        System.out.println("min=" + mi + "," +
                           "max=" + ma);
}}
```

# ⊢ Control-Flow Graph

```
public static void main ( String[] args ) {
    int mi, ma;
    if (args.length == 0)  /* 1if-else */
        System.out.println("No numbers");
    else {
        mi = ma = Integer.parseInt(args[0]);
        for (int i=1; i < args.length; i++) { /* 2for */
            int obs = Integer.parseInt(args[i]);
            if (obs > ma)  /* 3if-else */
                ma = obs;
            else
                if (mi < obs) mi = obs;  /* 4if */
        }
        System.out.println("Minimum = " + mi + " ;" +
                           "maximum = " + ma);
}}
```

- **CFG:**



# ⊢ Control Structures (cont'd²)

- **do-while:** **exercise**
  - `do Stm while ( Exp );`
- **"? :"; *"conditional expression"*:**
  - $Exp_1$ ? $Exp_2$ : $Exp_3$
- **"||"; *"lazy disjunction"*** (aka., *"short-cut ∨"*)**:**
  - $Exp_1$ || $Exp_2$
- **"&&"; *"lazy conjunction"*** (aka., *"short-cut ∧"*)**:**
  - $Exp_1$ && $Exp_2$
- **switch:**
  - `switch ( Exp ) { Swb* }`

*Swb*:

`case Exp : Stm* break;`

`default : Stm* break;`

# ⊢ Control Structures (cont'd[3])

- **`try-catch-finally`** (***exceptions***):
  - `try` ***Stm₁*** `catch ( ` ***Exp*** ` )` ***Stm₂*** `finally` ***Stm₃***
- **`return`** / **`break`** / **`continue`**:
  - `return ;` `return Exp ;` `break ;` `continue ;`
- ***"method invocation"*:**
  - e.g.; ***`f(x)`***
- ***"recursive method invocation"*:**
  - e.g.; ***`f(x)`***
- ***"virtual dispatching"*:**
  - e.g.; ***`f(x)`***

# ⊢ Control Structures (cont'd[4])

- *"**function pointers**"*:
  - e.g.; `(*f)(x)`
- *"**higher-order functions**"*:
  - e.g.; `λf.λx.(f x)`
- *"**dynamic evaluation**"*:
  - e.g.; `eval(some-string-which-has-been-dynamically-computed)`
- Some constructions (and thus languages) require a separate ***control-flow analysis*** for determining control-flow in order to do data-flow analysis

# ⊢ Agenda

- Introduction:
  - Undecidability, Reduction, and Approximation
- Data-flow Analysis:
  - Quick tour & running example
- Control-Flow Graphs:
  - Control-flow, data-flow, and confluence
- ”Science-Fiction Math”:
  - Lattice theory, monotonicity, and fixed-points
- Putting it all together…:
  - Example revisited

# MATH

# ⊢ Agenda

- Relations:
  - Crossproducts, powersets, and relations
- Lattices:
  - Partial-Orders, least-upper-bound, and lattices
- Monotone Functions:
  - Monotone Functions and Transfer Functions
- Fixed Points:
  - Fixed Points and Solving Recursive Equations
- Putting it all together…:
  - Example revisited

# ⊢ Crossproduct: '×'

- ***Crossproduct*** (binary operator on ***sets***):
  - Given sets:
    - $A = \{ 0, 1 \}$
    - $B = \{ \text{true}, \text{false} \}$
  - $A \times B = \{ (0,\text{true}), (0,\text{false}), (1,\text{true}), (1,\text{false}) \}$
    - i.e., *creates* ***sets of pairs***

- Exercise:
  - $A \times A =$ { (0,0), (0,1), (1,0), (1,1) }
  - $Z \times Z =$ { (0,0), (0,1), (0,1), …, (1,0), (1,1), …, (42,87), … }
  - $(A \times A) \times B =$ { ((0,0),true), ((0,1),true), …, ((1,1),false) }

# ⊢ Powersets : '$\mathcal{P}(\boldsymbol{S})$'

- ***Powerset*** (unary operator on ***sets***):
  - Given set "$\boldsymbol{S} = \{ A, B \}$";
  - $\mathcal{P}(\boldsymbol{S}) = \{ \emptyset, \{A\}, \{B\}, \{A,B\} = \boldsymbol{S} \}$
    - i.e., creates the ***set of all subsets*** (of the set)
  - Note: $X \subseteq \boldsymbol{S} \quad \Leftrightarrow \quad X \in \mathcal{P}(\boldsymbol{S})$

- Exercise:
  - $\mathcal{P}(\boldsymbol{Z}) =$ $\{ \emptyset, \{0\}, \{1\}, \{2\}, \ldots, \{0,1\}, \ldots \{13,42,87\}, \ldots \boldsymbol{Z} \}$
  - $\mathcal{P}(\boldsymbol{Z} \times \boldsymbol{Z}) =$ $\{ \emptyset, \{(0,0)\}, \{(1,1)\}, \ldots, \{(0,0),(3,2),(4,9)\}, \ldots \boldsymbol{Z}\times\boldsymbol{Z} \}$
  - If a set $\boldsymbol{S}$ has $|\boldsymbol{S}|$ elements;
    - How many elements does $\mathcal{P}(\boldsymbol{S})$ have? Answer: $2^{|S|}$

    '$\mathcal{P}(\boldsymbol{S})$' is (therefore) often written '$2^{\boldsymbol{S}}$'

# ⊢ Relations

- Example: "*equals*" relation:
  - Signature: $'=' \subseteq \mathbf{Z} \times \mathbf{Z}$ …same as saying: $'=' \in \mathcal{P}(\mathbf{Z} \times \mathbf{Z})$
  - Relation is: *equals* = { (0,0), (1,1), (2,2), (3,3), (4,4), … }
  - Written as: $2 = 2$ as a short-hand for: $(2,2) \in \text{'='}$
  - … and as: $2 \neq 3$ as a short-hand for: $(2,3) \notin \text{'='}$

- Example: "*less-than*" relation:
  - Signature: $'<' \subseteq \mathbf{Z} \times \mathbf{Z}$ …same as saying: $'<' \in \mathcal{P}(\mathbf{Z} \times \mathbf{Z})$
  - Relation is: *less-than* = { (0,1), (0,2), (0,3), …, (1,2), (1,3), … }
  - Written as: $7 < 8$ as a short-hand for: $(7,8) \in \text{'<'}$
  - … and as: $8 \not< 7$ as a short-hand for: $(8,7) \notin \text{'<'}$

# ⊢ Exercises

- Example: "*less-than-or-equal-to*" relation:
  - Signature: $\text{'}\leq\text{'} \subseteq \mathbf{Z} \times \mathbf{Z}$ …same as saying: $\text{'}\leq\text{'} \in \mathcal{P}(\mathbf{Z} \times \mathbf{Z})$
  - Relation is: $\text{'}\leq\text{'} = \{ (0,0), (0,1), (0,2), \ldots, (1,1), (1,1), \ldots, (2,3), \ldots \}$
  - Written as: $2 \leq 3$ as a short-hand for: $(2,3) \in \text{'}\leq\text{'}$
  - … and as: $3 \not\leq 2$ as a short-hand for: $(3,2) \notin \text{'}\leq\text{'}$

- Example: "*is-congruent-modulo-3*" relation:
  - Signature: $\text{'}\equiv_3\text{'} \subseteq \mathbf{Z} \times \mathbf{Z}$ …same as saying: $\text{'}\equiv_3\text{'} \in \mathcal{P}(\mathbf{Z} \times \mathbf{Z})$
  - Relation is: $\text{'}\equiv_3\text{'} = \{ (0,0), (0,3), (0,6), \ldots, (1,1), (1,4), \ldots, (6,9), \ldots \}$
  - Written as: $6 \equiv_3 9$ as a short-hand for: $(6,9) \in \text{'}\equiv_3\text{'}$
  - … and as: $7 \not\equiv_3 8$ as a short-hand for: $(7,8) \notin \text{'}\equiv_3\text{'}$

# ⊢ Equivalence Relation

- Let '~' be a ***binary relation*** *over set* $A$:
  - '~' $\subseteq A \times A$

- ~ is an ***equivalence relation*** iff:
  - *Reflexive*:
    - $\forall x \in A: \quad x \sim x$
  - *Symmetric*:
    - $\forall x,y \in A: \quad x \sim y \;\Leftrightarrow\; y \sim x$
  - *Transitive*:
    - $\forall x,y,z \in A: \quad x \sim y \;\wedge\; y \sim z \;\Rightarrow\; x \sim z$

# ⊢ Agenda

- Relations:
  - Crossproducts, powersets, and relations
- Lattices:
  - Partial-Orders, least-upper-bound, and lattices
- Monotone Functions:
  - Monotone Functions and Transfer Functions
- Fixed Points:
  - Fixed Points and Solving Recursive Equations
- Putting it all together…:
  - Example revisited

# ⊢ Partial-Order

- A ***Partial-Order*** is a structure $(S, \sqsubseteq)$:
  - ***S*** is a *set*
  - '$\sqsubseteq$' is a *binary relation* on ***S*** (i.e., '$\sqsubseteq$' $\subseteq S \times S$) satisfying:

- ***Reflexivity***:
  - $\forall x \in S: \quad x \sqsubseteq x$
- ***Transitivity***:
  - $\forall x,y,z \in S: \quad x \sqsubseteq y \;\wedge\; y \sqsubseteq z \;\Rightarrow\; x \sqsubseteq z$
- ***Anti-Symmetry***:
  - $\forall x,y \in S: \quad x \sqsubseteq y \;\wedge\; y \sqsubseteq x \;\Rightarrow\; x = y$

# ⊢ Visualization: *Hasse Diagram*

***Partial-Order* ($S$, ⊑):** ⇔ ***Hasse Diagram:***

- ***Reflexive***:

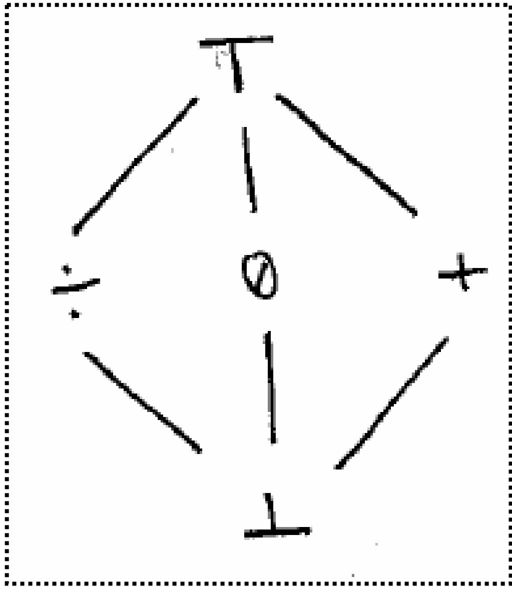
  $\forall x \in S: \; x \sqsubseteq x$

- ***Transitive***:

  $\forall x,y,z \in S: \; x \sqsubseteq y \;\wedge\; y \sqsubseteq z \;\Rightarrow\; x \sqsubseteq z$

- ***Anti-Symmetric***:

  $\forall x,y \in S: \; x \sqsubseteq y \;\wedge\; y \sqsubseteq x \;\Rightarrow\; x = y$

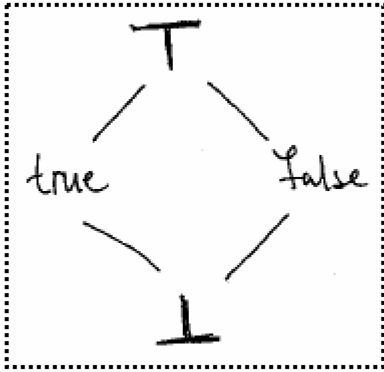



$S = \{\bot, \div, 0, +, \top\}$

$`\sqsubseteq' = \{(\bot,\bot), (\bot,\div), (\bot,0), (\bot,+), (\bot,\top), (\div,\div), (\div,\top), (0,0), (0,\top), (+,+), (+,\top), (\top,\top)\}$

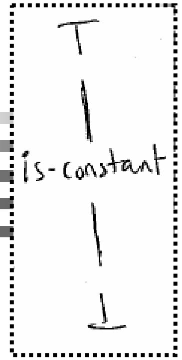
# Exercise (Hasse Diagram)

- Given Hasse Diagram:

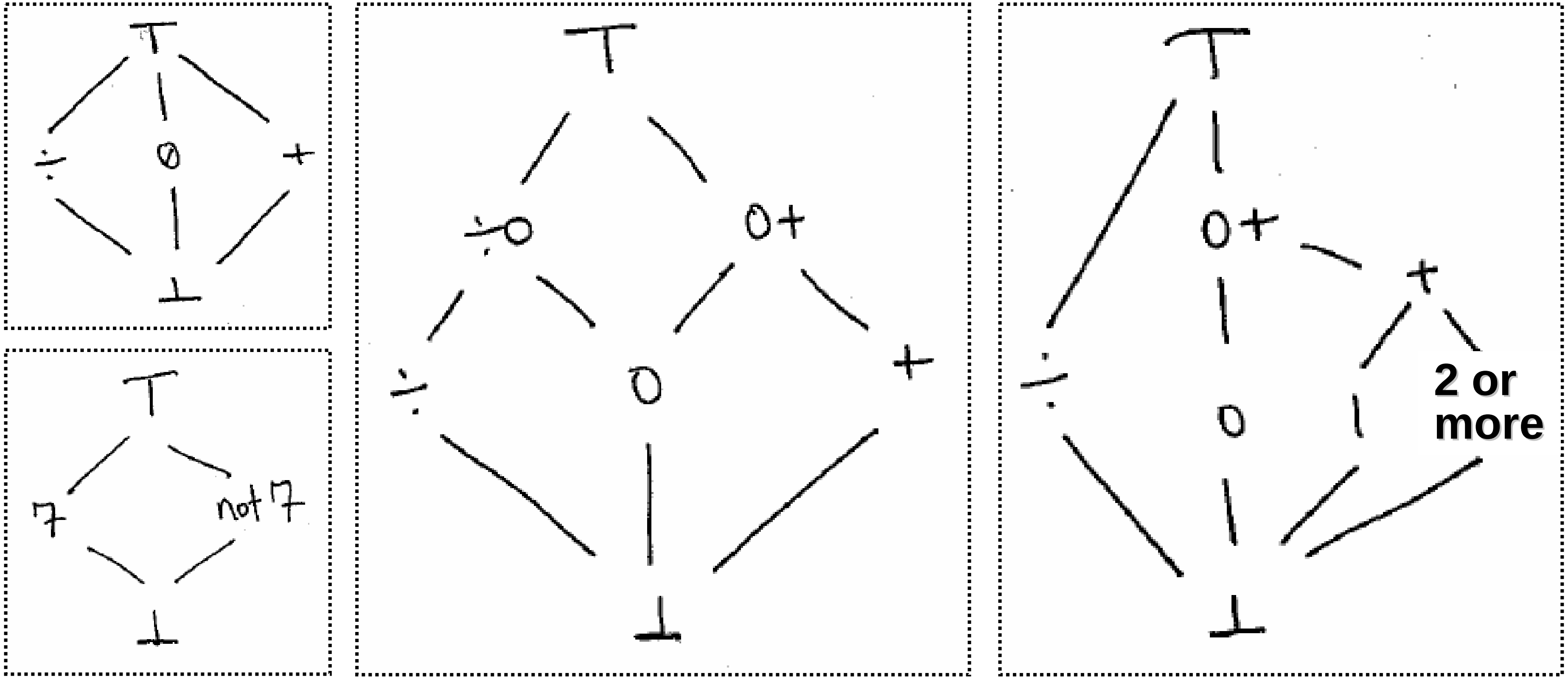



- Write down partial order (***B***, $\sqsubseteq$):
  - Set ***B*** = { … }
  - Relation '$\sqsubseteq$':
    - Signature
    - All elements of the relation (i.e., '$\sqsubseteq$' = { … } )
    - Give example of element in '$\sqsubseteq$' (w/ + w/o shorthand)
    - Again, but for an element ***not*** in the relation

# ⊢ Example Partial-Orders




- Lattice Examples (as Hasse Diagrams):




- …depending on what is analysed for!

# Least Upper Bound '⊔'

# ⊢ "Least Upper Bound" $\sqcup$

- ***Upper bound:***
  - We say that 'z' is an upper bound for set 'X'
    - …written $X \sqsubseteq z$ if $\forall x \in X: \; x \sqsubseteq z$

- ***Least upper bound:***
  - We say that 'z' is ***the*** *least upper bound* of set 'X'
    - …written $z = \sqcup X$ if $\underbrace{X \sqsubseteq z}_{\text{upper bound}} \wedge \underbrace{\forall z': X \sqsubseteq z' \Rightarrow z \sqsubseteq z'}_{\text{least}}$

# ⊢ Example: Least upper bound

- Analyses use '⊔' to ***combine information*** (at confluence points):