Automatic Software Analysis

Claus
BRABRAND

~~Jesper~~
~~**BENGTSON**~~

Automatic Software Analysis
`/* ANALYSIS I */`

Claus
BRABRAND

Jesper
BENGTSON

Abstract

- **"Data-Flow Analysis":**

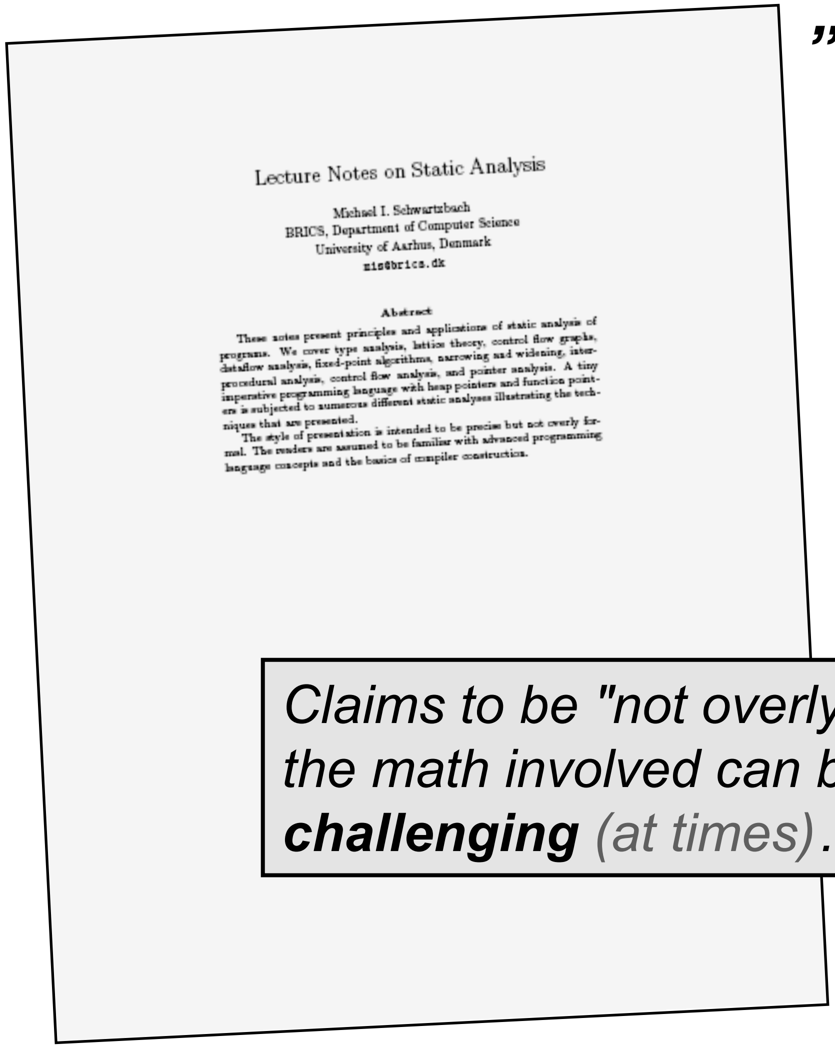
In this 3*3 hour **mini course** we will look at **data-flow analysis**. Rather than just look at the classical "monotone framework" analyses (which are usually synonymous with teaching data-flow analysis: reaching definitions, live variables, available expressions, and very busy expressions), we will 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.

Schedule

##	TEACHER	WEDNESDAY	TOPIC	ASSIGNMENT
01	JB+CB	Aug 30	INTRODUCTION	--
02	JB	Sep 06	1) **Semantics** (Operational Semantics)	(A1)
03	JB	Sep 13		(A2)
04	JB	Sep 20		(A3)
05	JB	Sep 27		(A4)
07	CB	Oct 04	2) **Analysis** (Dataflow Analysis)	(A5)
08	CB	Oct 11		(A6)
09	CB	Oct 25		(A7)
10	CB	Nov 01		(A8)
11	JB+CB	Nov 08	TOOLS 3) **Tools**	(A9)
12	--	Nov 15	4) **Project** (in (1), (2), and/or (3))	--
13	--	Nov 22		--
14	--	Nov 29		--
15	JB+CB	Dec 06	PROJECT DEMONSTRATIONS	(P)

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"

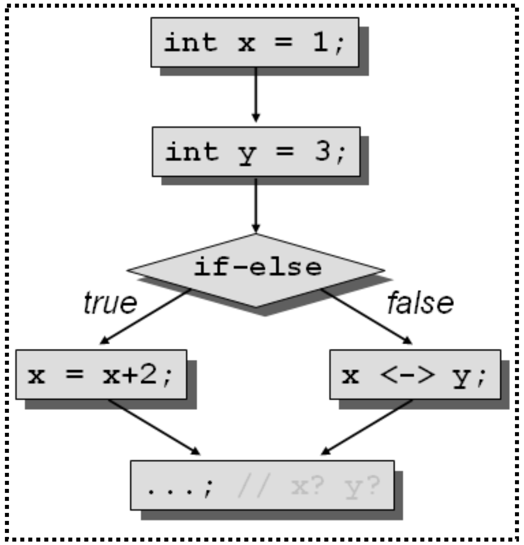
by Michael I. Schwartzbach

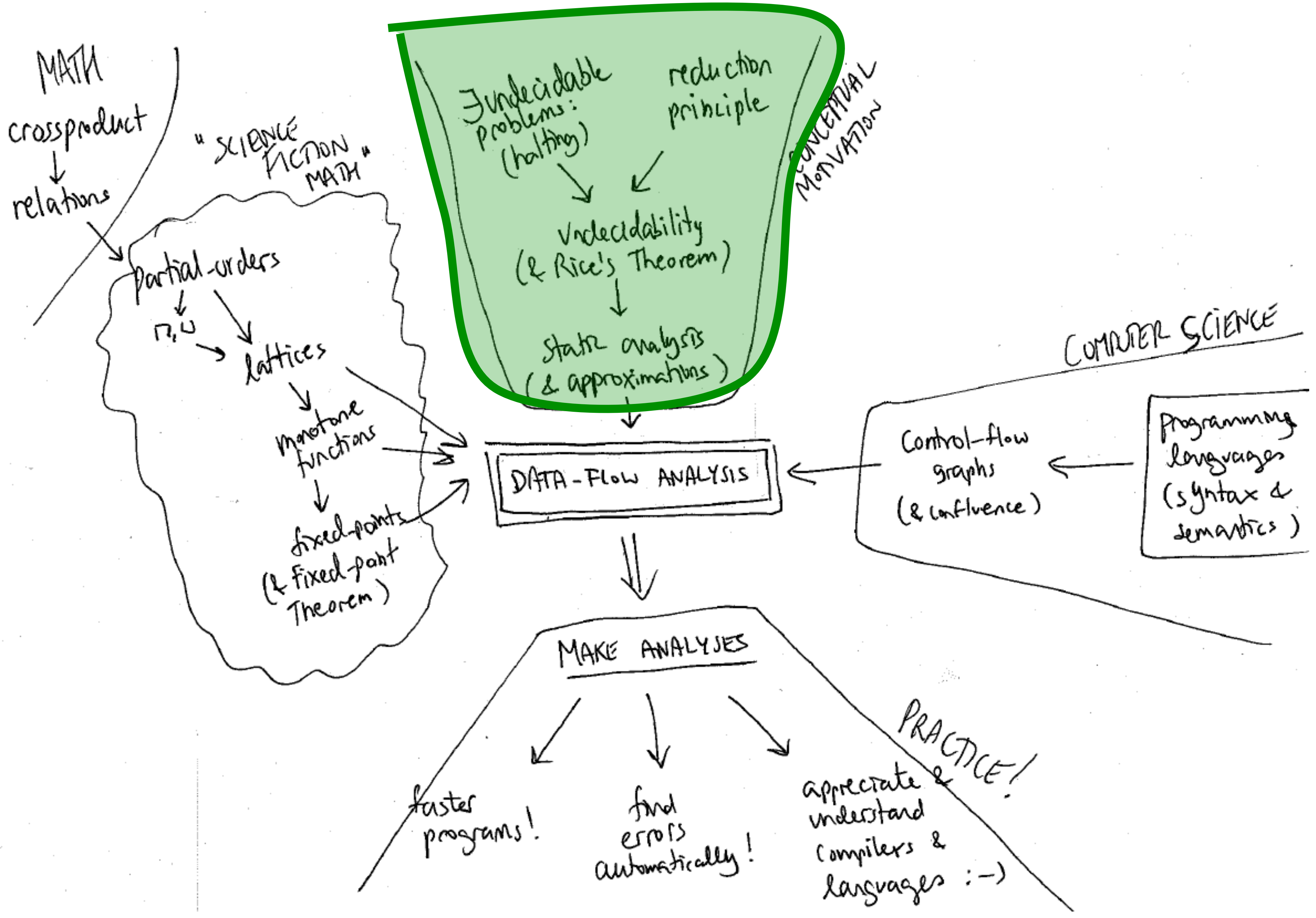
(Aarhus University)

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)…*




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

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”:
  - Lattice theory, monotonicity, and fixed-points
- Putting it all together…:
  - Example revisited

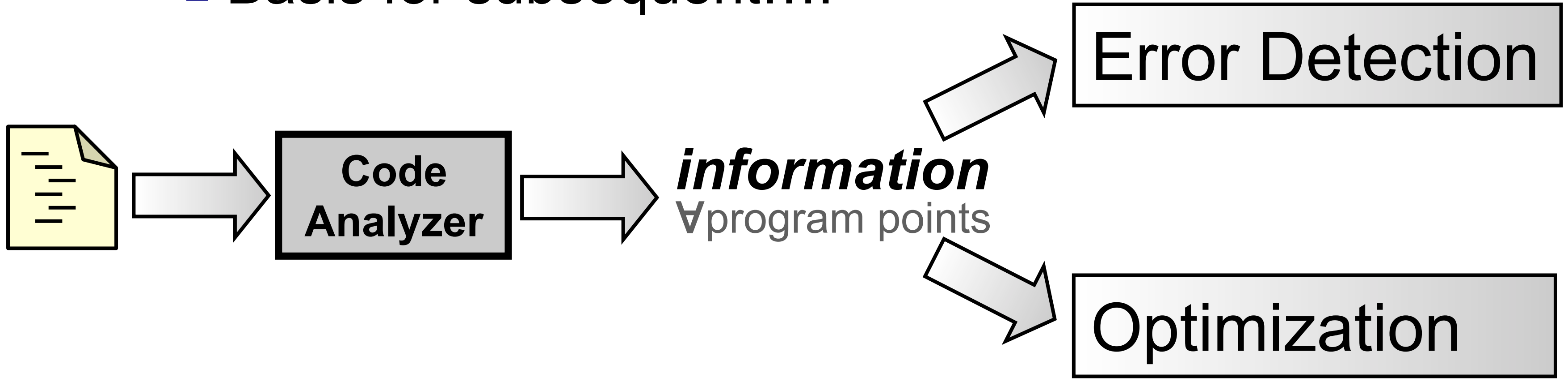
Conceptual Motivation

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

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

- ***Uninitialized Variable Analysis:***
  - Catch unintialized variables
- ***Null-Pointer Analysis:***
  - Catch null-pointer errors
- ***Information Leak Analysis:***
  - Which parts of the program leaks "secret information"
- …

Analyses for Optimization

- ***Constant Propagation Analysis:***
  - Precompute constants (e.g., replace '**`5*x+z`**' by '**`42`**')
- ***Live Variables Analysis:***
  - Elimiate dead code (e.g., get rid of unused variable '**`z`**')
- ***Available Expressions Analysis:***
  - Avoid recomputing expressions (cache results)
- …

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*?

The Book-of-all-Books

☺

- *"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 = $\boxed{\textit{x-is-always-positive}(R)}$

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):

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

- Example:

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

- **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*:

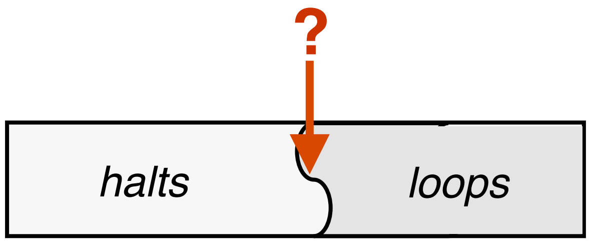
```
-- R.java --
    ptr q = 0xFFFF;
    P /* insert program P (assume w/o 'q') */
    q = null;
```

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

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

Undecidability

- Undecidability means that…:



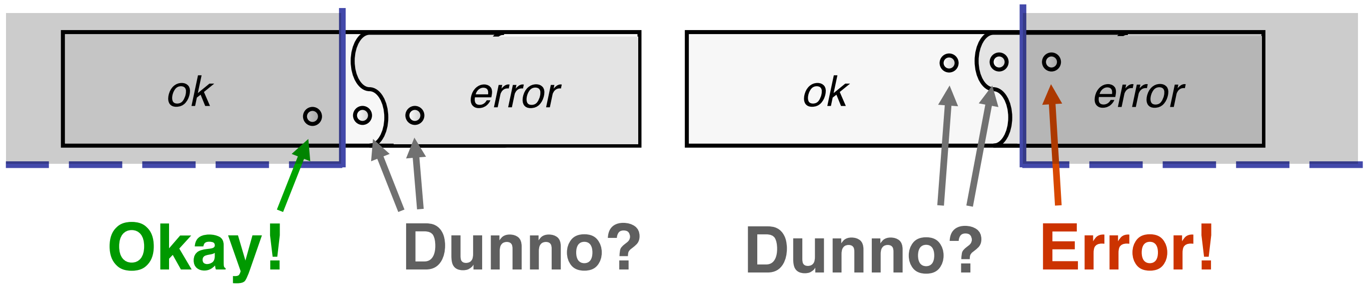
  - *…no-one can decide this **line** (for all programs)!*

- ***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***":

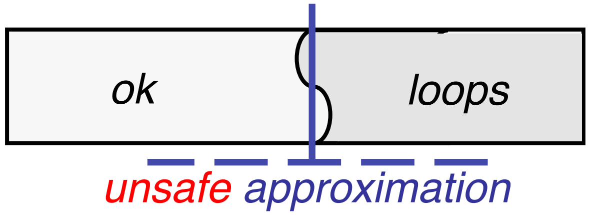
- Compilers use ***safe approximations*** (computed via "static analyses") such that:



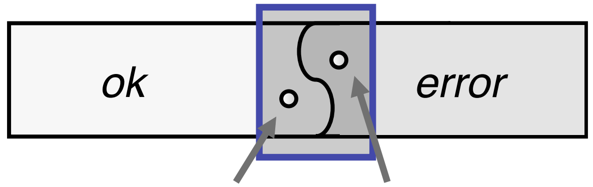
Side-Stepping Undecidability

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:***



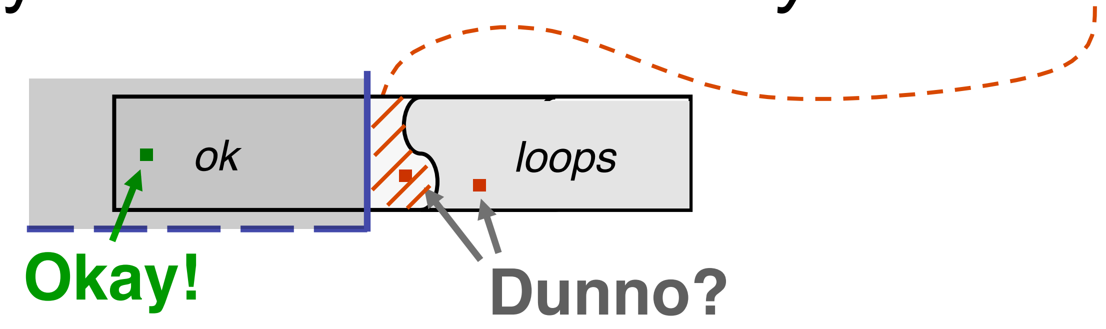
- For ***testing*** it may be okay to "abandon" safety and use ***unsafe approximations:***



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: …"*

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 if b is '42'
    }
    ```
  - **Undecidable** (because of reduction):
    - Type error ⇔ *<EXP>* evaluates to `true`

Example: Type Checking

- Hence, languages use ***static requirements:***

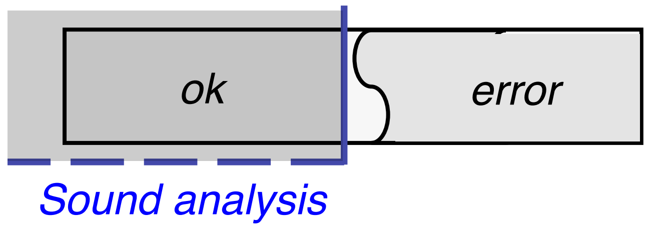
  -
    ```
    void f() {
        bool b; // instead of "var b;"

        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")

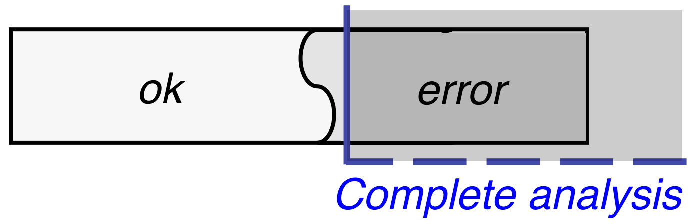
Soundness & Completeness

- ***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”:

$\boxed{P \Rightarrow Q} \equiv \boxed{\neg Q \Rightarrow \neg P}$

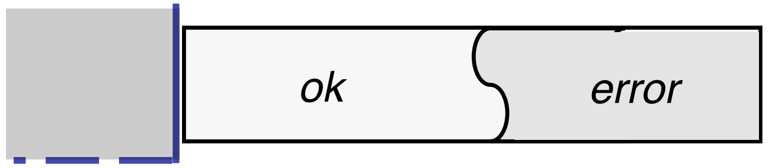
- Really are error(s)
  ⇒ Analysis reports error(s)

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

$\boxed{\vdash P \Rightarrow \models P} \equiv \boxed{\not\models P \Rightarrow \not\vdash P}$ $\boxed{\not\vdash P \Rightarrow \not\models P} \equiv \boxed{\models P \Rightarrow \vdash P}$

Trivial Soundness/Completeness!

- Trivial Soundness:



Sound analysis

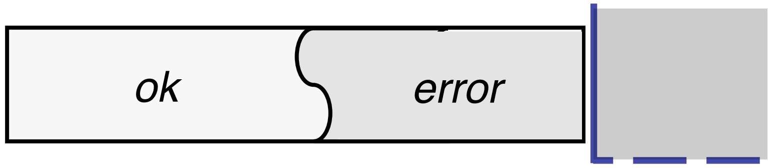
- Analysis =

```
read_program(...);
print("may be error!");
```

- Soundness:
  ~~Analysis reports no errors~~
  => really are no errors

- Never says "**`okay!`**";
  hence trivially sound!

- Trivial Completeness:



Complete analysis

- Analysis =

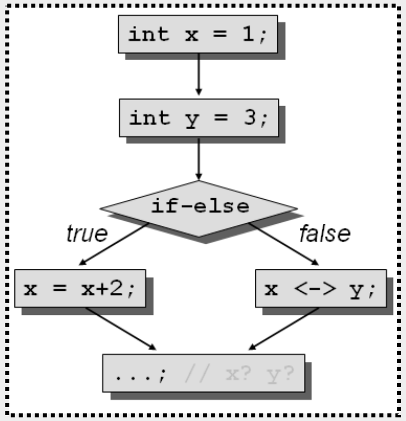
```
read_program(...);
print("may be okay!");
```

- Completeness:
  ~~Analysis reports an error~~
  => really is an error

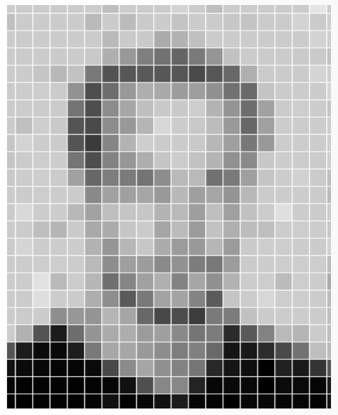
- Never says "**`error!`**";
  hence trivially complete!

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.
(((Software and Systems Section)))
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"*

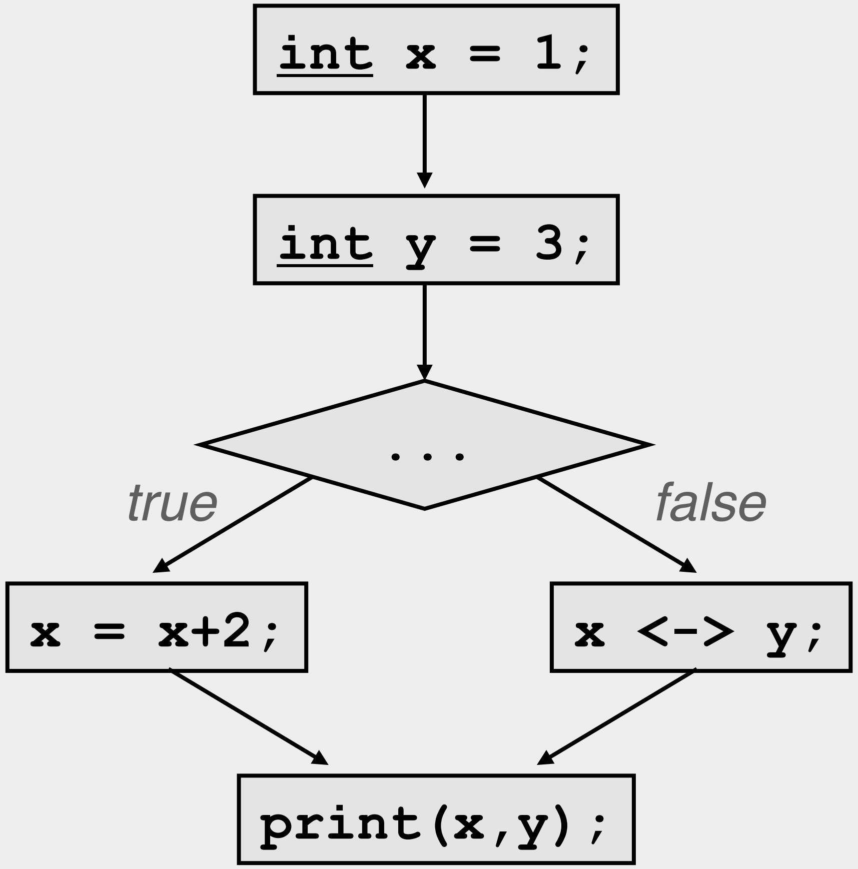
Control-flow graph

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);
```

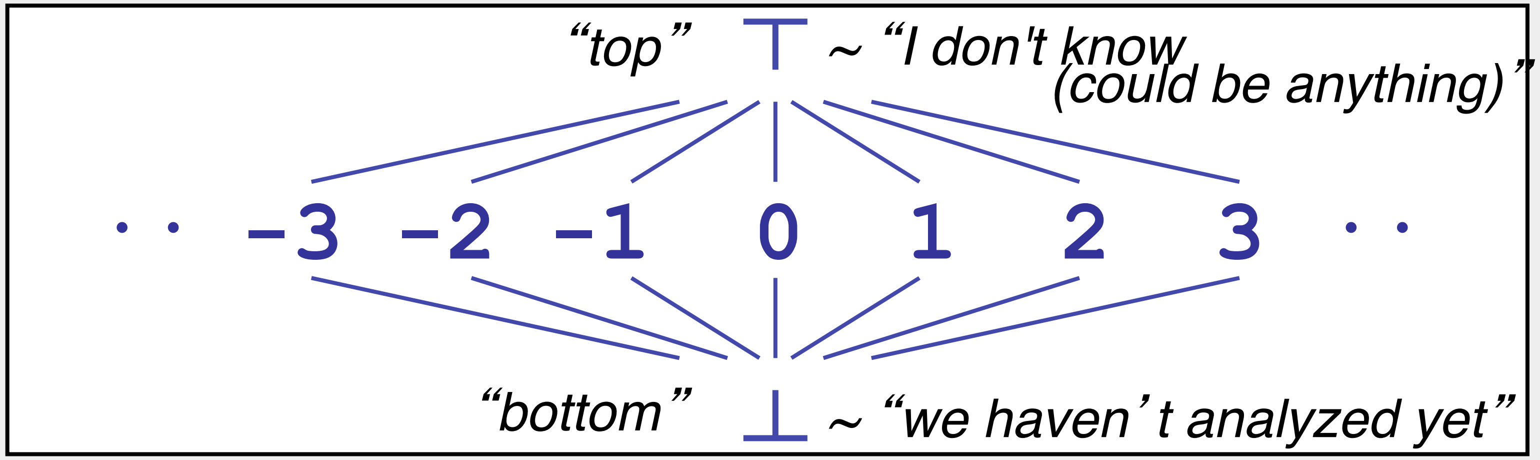
$\longrightarrow$



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 "≤"):



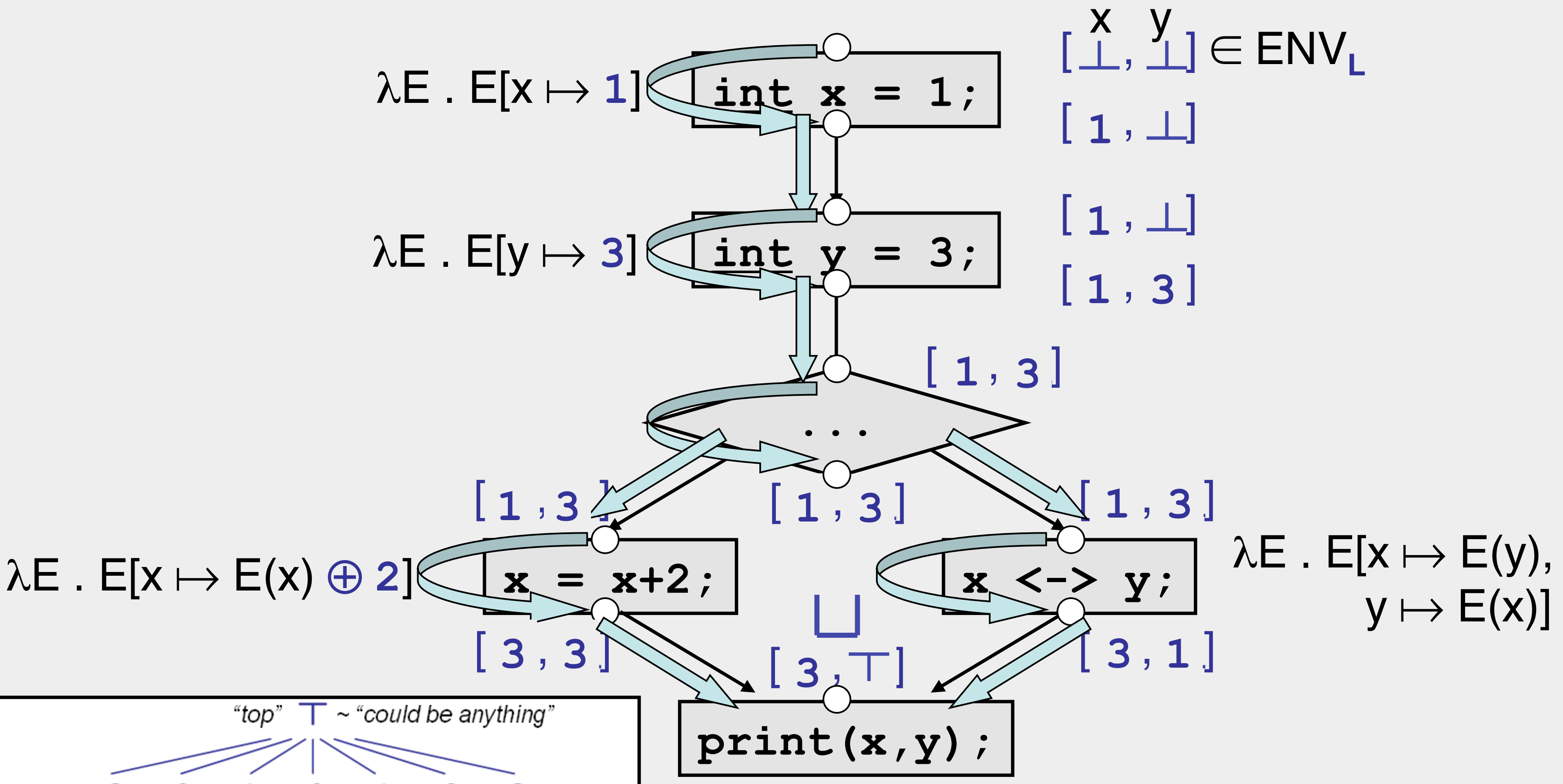
"top" ⊤ ~ "I don't know (could be anything)"
· · -3 -2 -1 0 1 2 3 · ·
"bottom" ⊥ ~ "we haven't analyzed yet"

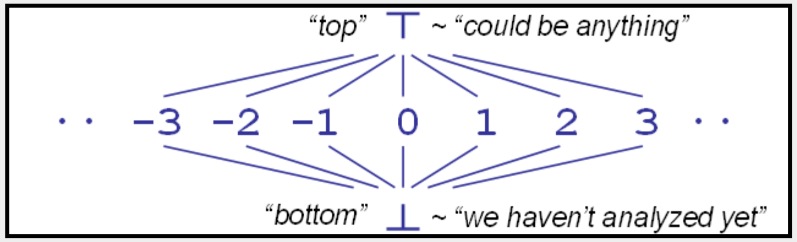
- Induces *least-upper-bound* operator: ⊔
  - for ***combining information***

Data-Flow Analysis

We (only) need 3 things:

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





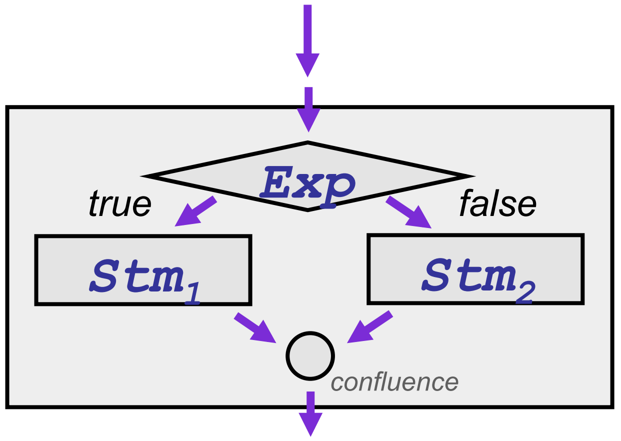
"top" ⊤ ~ "could be anything"
·· -3 -2 -1 0 1 2 3 ··
"bottom" ⊥ ~ "we haven't analyzed yet"

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

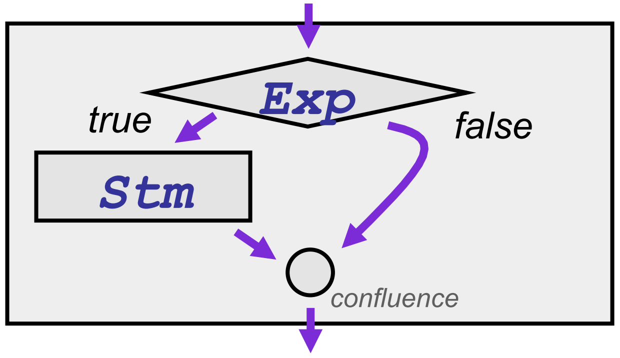
- ***Control Structures*:**
  - ***Statements*** ***(or Expr's)*** that ***affect*** ***"flow of control"*:**



- `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`

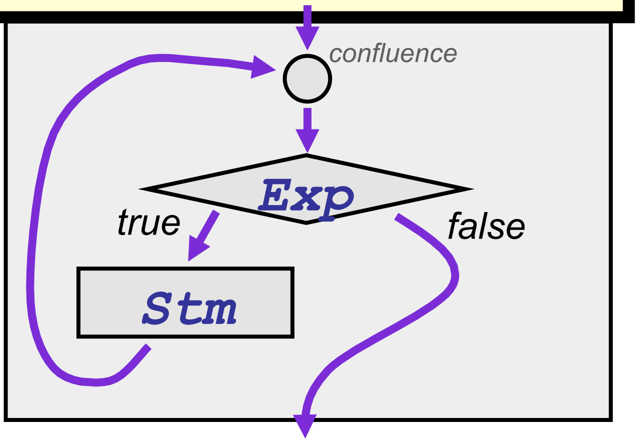
[semantics] ■ *The expression must be of type **boolean**; if it evaluates to **false**, the given statement is skipped, otherwise it is executed and afterwards the expression is evaluated again. If it is still true, the statement is executed again. This is continued until the expression evaluates to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

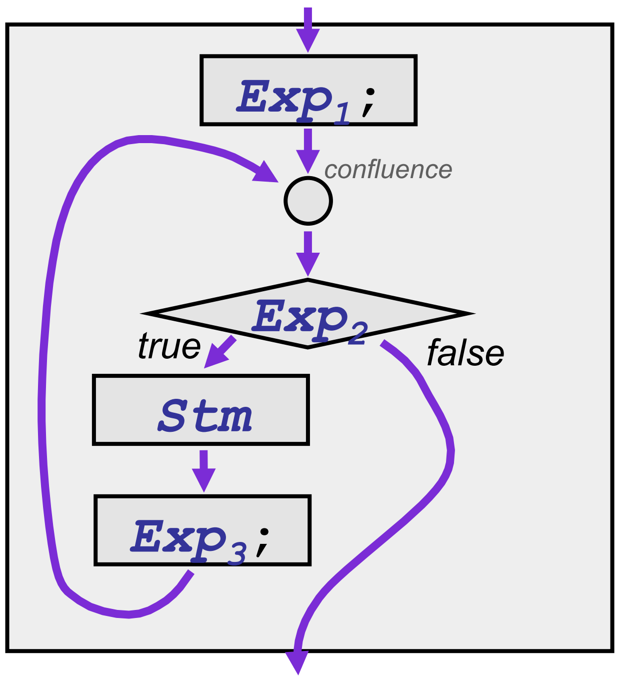
- **for**:

[syntax] ■ `for (Exp₁ ; Exp₂ ; Exp₃) Stm`

[semantics] ■ *Equivalent to:*

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



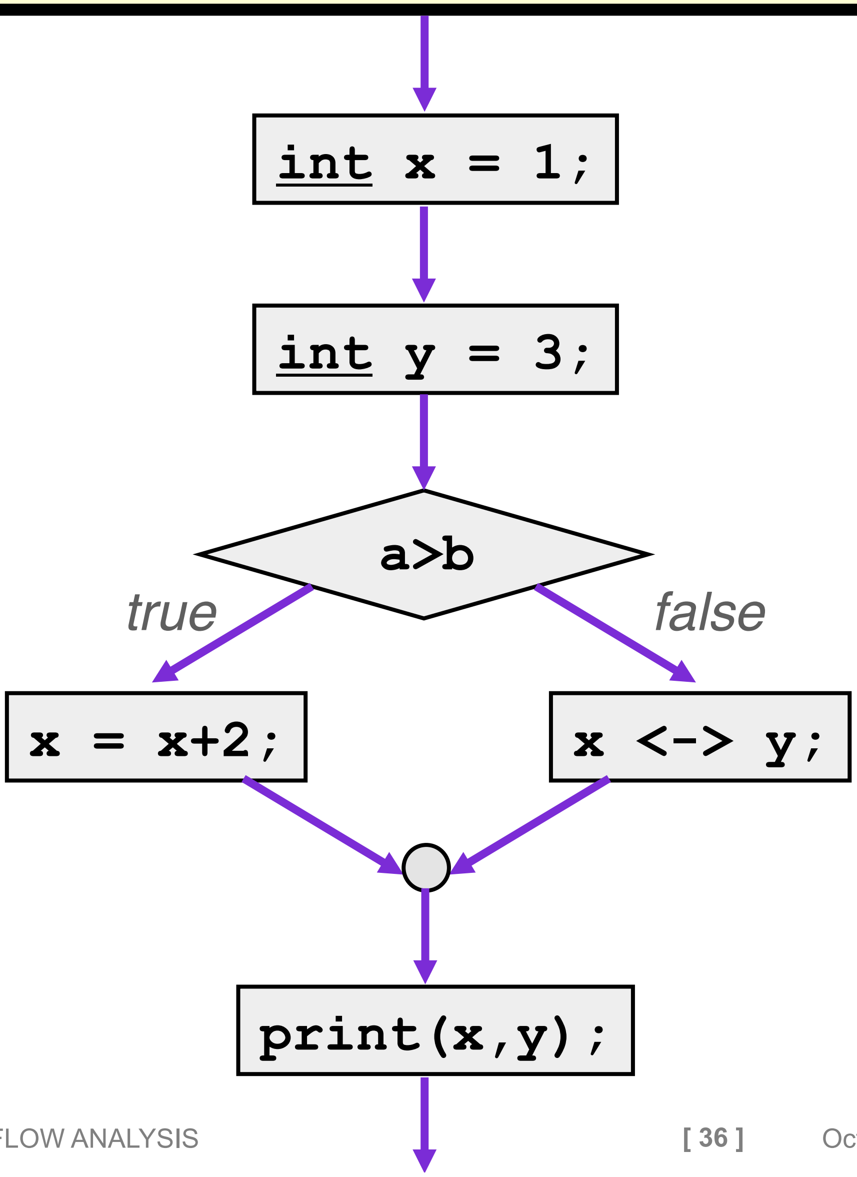


Control-flow graph

Given program:

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

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

$\rightarrow$



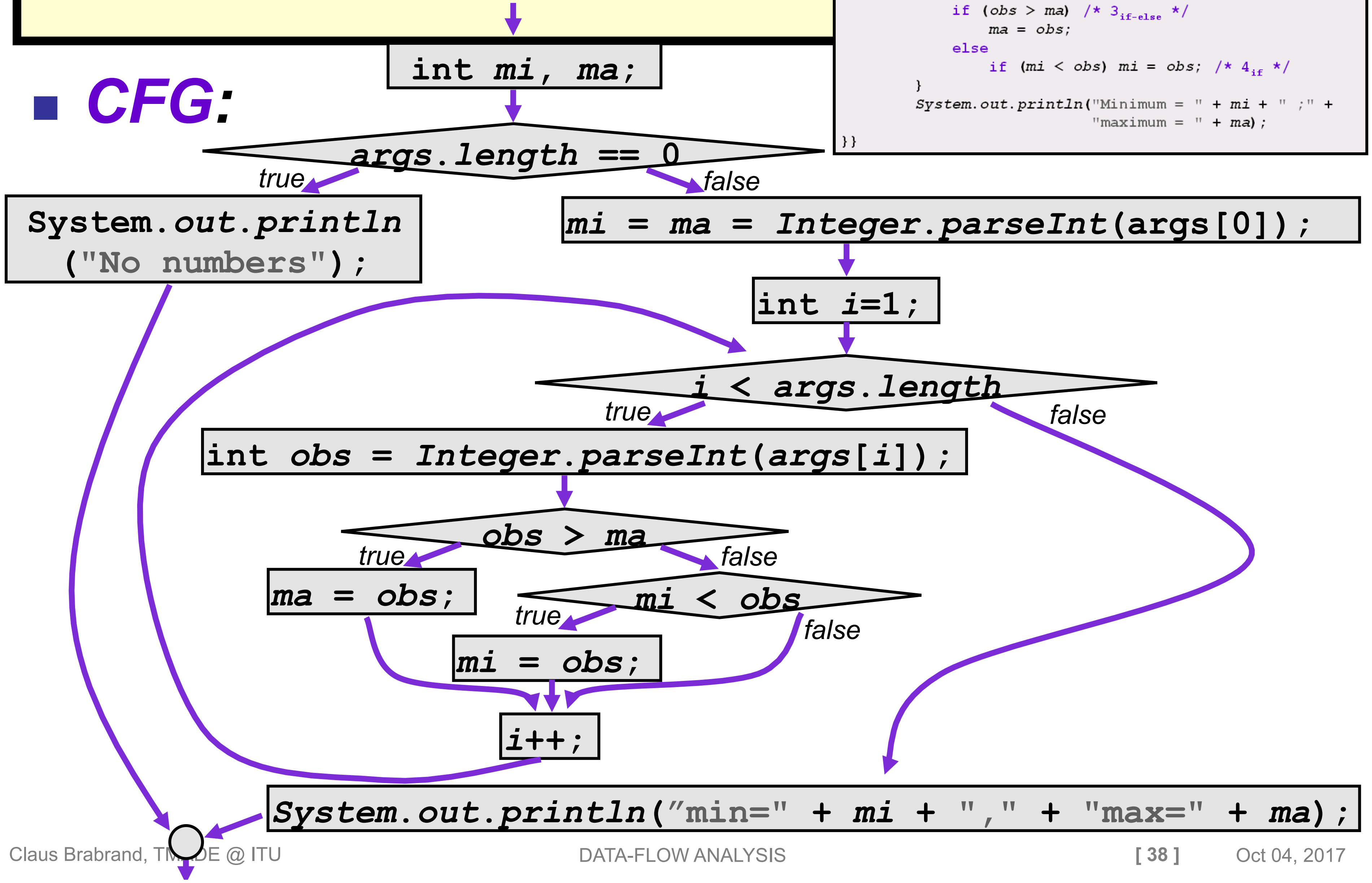
Exercise: Draw a Control-Flow Graph for:

```
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:**



int mi, ma;
args.length == 0
true
false
System.out.println ("No numbers");
mi = ma = Integer.parseInt(args[0]);
int i=1;
i < args.length
true
false
int obs = Integer.parseInt(args[i]);
obs > ma
true
false
ma = obs;
mi < obs
true
false
mi = obs;
i++;
System.out.println("min=" + mi + "," + "max=" + ma);

Control Structures (cont'd^2)

- `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` Stm_1 `catch ( ` Exp ` )` Stm_2 `finally` Stm_3
- `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*** = { true, false }
  - ***A*** × ***B*** = { (0,true), (0,false), (1,true), (1,false) }
    - i.e., *creates* ***sets of pairs***

- **Exercise:**
  - ***A*** × ***A*** = { (0,0), (0,1), (1,0), (1,1) }
  - ***Z*** × ***Z*** = { (0,0), (0,1), (0,1), …, (1,0), (1,1), …, (42,87), … }
  - ***(A × A)*** × ***B*** = { ((0,0),true), ((0,1),true), …, ((1,1),false) }

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

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

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

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

Relations

A relation is a set!

Relations

- Example[1]: "*even*" relation: $\boxed{\vdash_{even} \subseteq \mathbf{Z}}$
  - Written as: $\boxed{\vdash_{even} 4}$ as a short-hand for: $\boxed{4 \in \vdash_{even}}$
  - … and as: $\boxed{\nvdash_{even} 5}$ as a short-hand for: $\boxed{5 \notin \vdash_{even}}$

- Example[2]: "*equals*" relation: $\boxed{\text{'='} \subseteq \mathbf{Z} \times \mathbf{Z}}$
  - Written as: $\boxed{2 = 2}$ as a short-hand for: $\boxed{(2,2) \in \text{'='}}$
  - … and as: $\boxed{2 \neq 3}$ as a short-hand for: $\boxed{(2,3) \notin \text{'='}}$

- Example[3]: *DFA transition* relation: $\boxed{\text{'}\rightarrow\text{'} \subseteq \mathbf{Q} \times \Sigma \times \mathbf{Q}}$
  - Written as: $\boxed{q \xrightarrow{\sigma} q'}$ as a short-hand for: $\boxed{(q, \sigma, q') \in \text{'}\rightarrow\text{'}}$
  - … and as: $\boxed{p \overset{\sigma}{\not\rightarrow} p'}$ as a short-hand for: $\boxed{(p, \sigma, p') \notin \text{'}\rightarrow\text{'}}$

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: $(2,2) \in '='$
  - … and as: $2 \neq 3$ as a short-hand for: $(2,3) \notin '='$

- 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: $(7,8) \in '<'$
  - … and as: $8 \nless 7$ as a short-hand for: $(8,7) \notin '<'$

Inference System

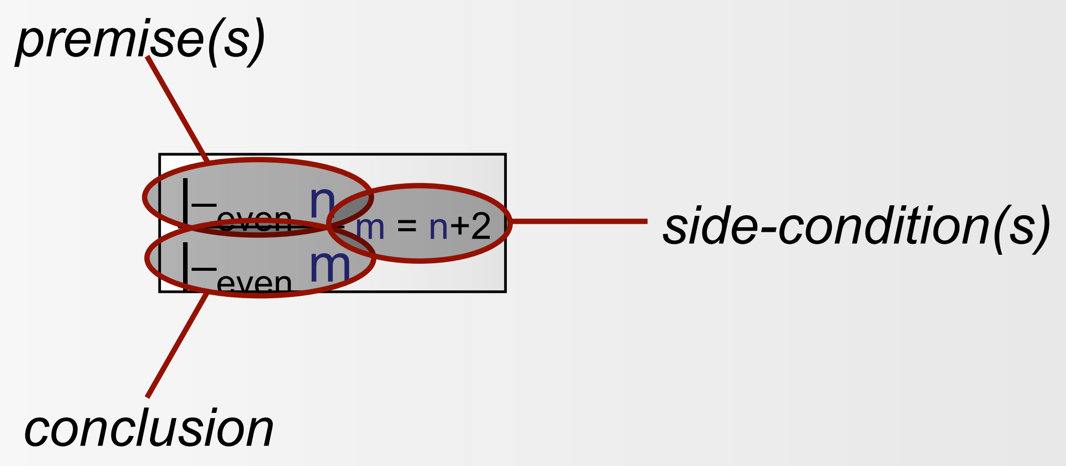
- Inference System:
  - is used for *specifying* relations
  - consists of *axioms* and *rules*

- Example: $\boxed{\vdash_{\text{even}} \subseteq \mathbf{Z}}$

- Axiom: $\boxed{\dfrac{}{\vdash_{\text{even}} 0}}$
  - *"0 (zero) is even"!*

- Rule: $\boxed{\dfrac{\vdash_{\text{even}} n}{\vdash_{\text{even}} m}\ m = n+2}$
  - *"If n is even, then m is even (where m = n+2)"*

Terminology



- Interpretation:
  - <u>Deductive:</u>
    "m is even, if n is even (where m = n+2)"
  - <u>Inductive:</u>
    "If n is even, then m is even (where m = n+2)"; or

Abbreviation

- Often, rules are *abbreviated*:

- Rule: $\dfrac{\vdash_{\text{even}} n}{\vdash_{\text{even}} m}\; m = n+2$
  - *"m is even, if n is even (where m = n+2)"*

Even so; this is what we mean

- *Abbreviated rule*: $\dfrac{\vdash_{\text{even}} n}{\vdash_{\text{even}} n+2}$
  - *"n+2 is even, if n is even"*

Relation Membership? $x \overset{?}{\in} \mathcal{R}$

- Axiom: $\boxed{\dfrac{}{\vdash_{even}\ 0}}$
  - *"0 (zero) is even"!*
- Rule: $\boxed{\dfrac{\vdash_{even}\ n}{\vdash_{even}\ n+2}}$
  - *"n+2 is even, if n is even"*
- Is 6 even?!?

$$\boxed{\left.\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_{even}\ 0}\ [\text{axiom}_1]}{\vdash_{even}\ 2}\ [\text{rule}_1]}{\vdash_{even}\ 4}\ [\text{rule}_1]}{\vdash_{even}\ 6}\ [\text{rule}_1]\right\}\ \textit{inference tree}}$$

- The *inference tree* ***proves*** that: $\boxed{6 \in \vdash_{even}}$ ***written:*** $\boxed{\vdash_{even}\ 6}$

Example: less-than-or-equal-to

- Relation: $\boxed{`\leq' \subseteq \mathbf{N} \times \mathbf{N}}$

$$[\text{axiom}_1]\ \ \boxed{\frac{}{0 \leq 0}} \qquad [\text{rule}_1]\ \ \boxed{\frac{n \leq m}{n \leq m+1}} \qquad [\text{rule}_2]\ \ \boxed{\frac{n \leq m}{n+1 \leq m+1}}$$

- Is "$1 \leq 2$" **?** (why/why not)!?
  - Yes, because there exists an inference tree:
    - In fact, it has *two* inference trees:

$$\boxed{\frac{\dfrac{\dfrac{}{0 \leq 0}}{0 \leq 1}}{1 \leq 2}}\ \begin{array}{l}[\text{axiom}_1]\\ [\text{rule}_1]\\ [\text{rule}_2]\end{array} \qquad \boxed{\frac{\dfrac{\dfrac{}{0 \leq 0}}{1 \leq 1}}{1 \leq 2}}\ \begin{array}{l}[\text{axiom}_1]\\ [\text{rule}_2]\\ [\text{rule}_1]\end{array}$$

Exercise 1

- Activation Exercise:
  - **1.** Specify the signature of the relation: '<<'
    - `x << y`   "*y is-double-that-of x*"
  - **2.** Specify the relation via an inference system
    - i.e. axioms and rules
  - **3.** Prove that indeed:
    - `3 << 6`   "*6 is-double-that-of 3*"

Exercise 2

- Activation Exercise:
  - **1.** Specify the signature of the relation: '`//`'
    - `x // y` "**x** *is-half-that-of* ***y***"
  - **2.** Specify the relation via an inference system
    - i.e. axioms and rules
  - **3.** Prove that indeed:
    - `3 // 6` "*3 is-half-that-of 6*"

Syntactically different: '`3 << 6`' *vs.* '`3 // 6`'
Semantically the ***same*** relation: '`<<`' = '`//`' = { (0,0), (1,2), (2,4), (3,6), (4,8), ... }

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

- $\sim$ 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 $(\boldsymbol{S}, \sqsubseteq)$:
  - $\boldsymbol{S}$ is a *set*
  - '$\sqsubseteq$' is a *binary relation* on $\boldsymbol{S}$ (i.e., '$\sqsubseteq$' $\subseteq \boldsymbol{S} \times \boldsymbol{S}$) satisfying:

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

Visualization: *Hasse Diagram*

Partial-Order $(S, \sqsubseteq)$: $\Leftrightarrow$ ***Hasse Diagram:***

- ***Reflexive***:

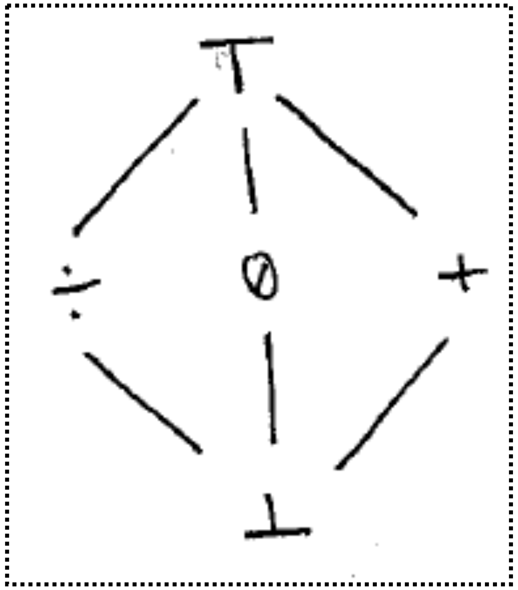
  $\forall x \in S: \; x \sqsubseteq x$

- ***Transitive***:

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

- ***Anti-Symmetric***:

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

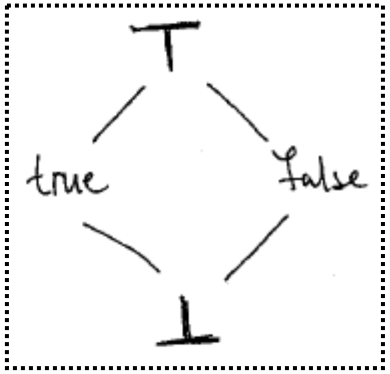


$S = \{\bot, \div, 0, +, \top\}$

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

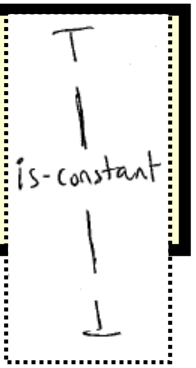
Exercise (Hasse Diagram)

- Given Hasse Diagram:

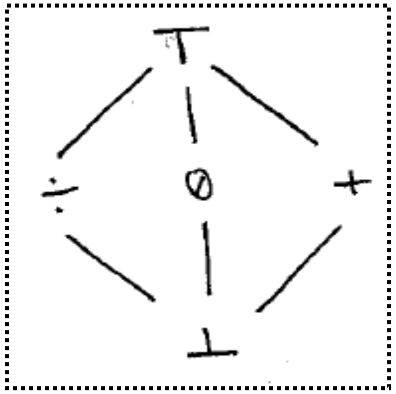


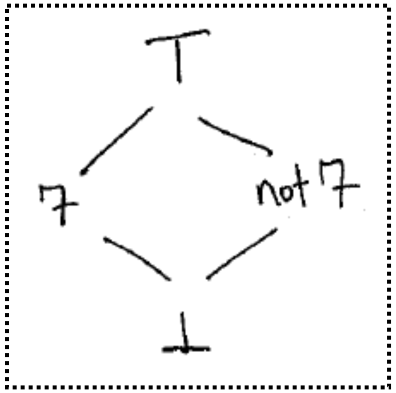
- Write down partial order (***B***,$\sqsubseteq$):
  - Set ***B*** = { … }
  - Relation '$\sqsubseteq$':
    - Signature
    - All elements of the relation (i.e., '$\sqsubseteq$' = { … })
    - Give example of element in '$\sqsubseteq$' (w/ + w/o shorthand)
    - Again, but for an element ***not***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

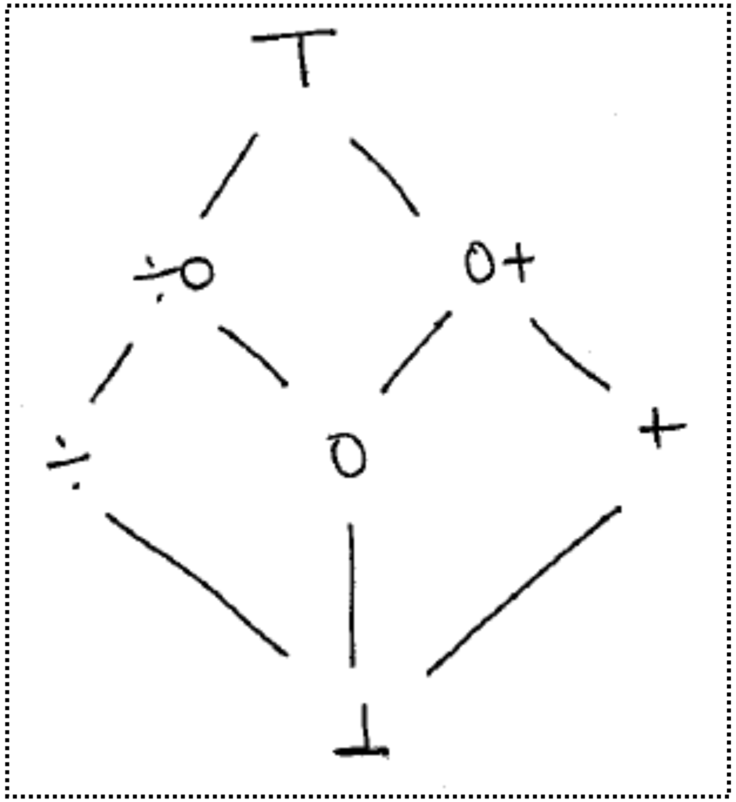
Example Partial-Orders

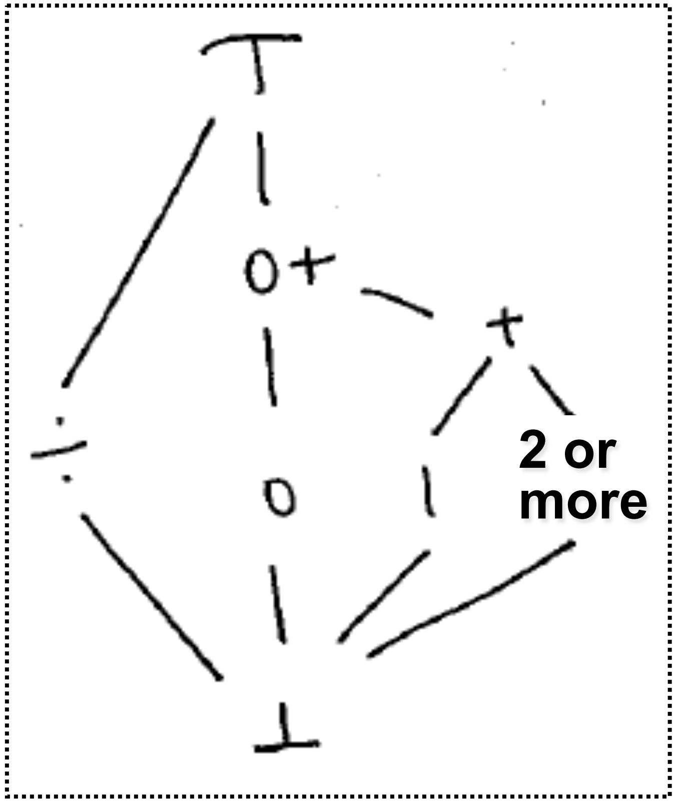


- Lattice Examples (as Hasse Diagrams):









- …depending on what is analysed for!

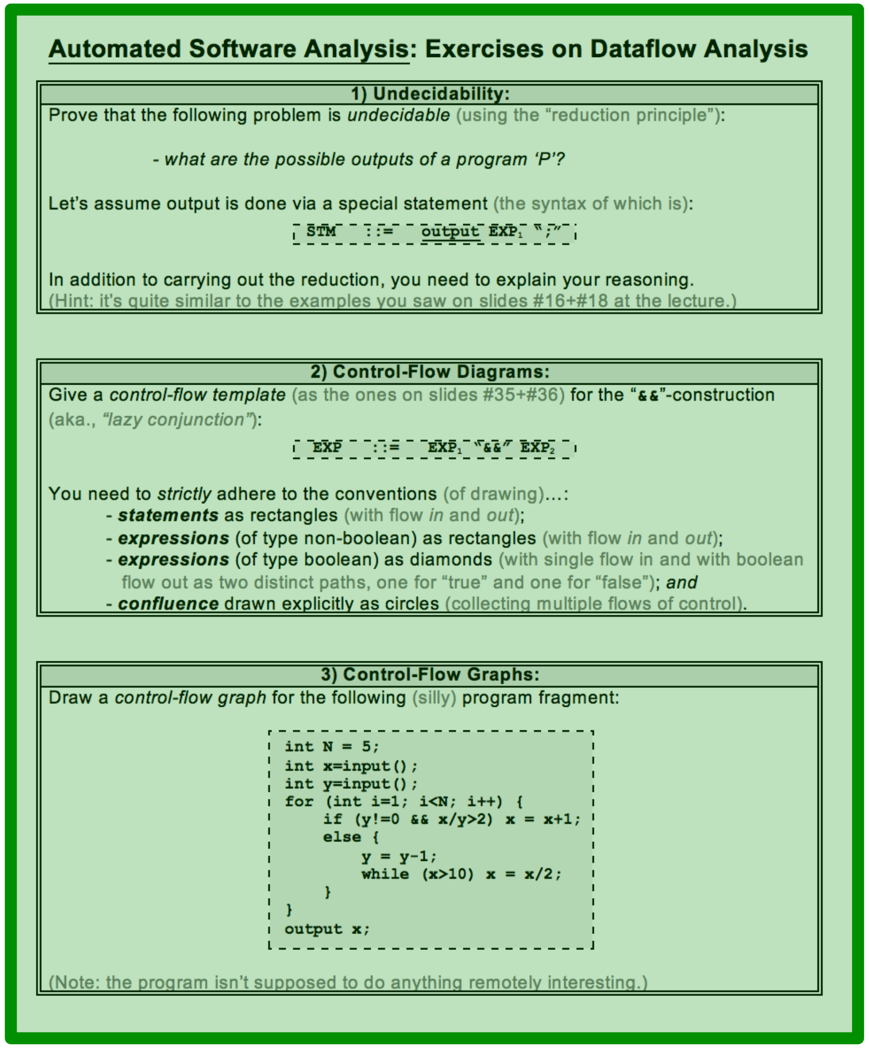
Least Upper Bound '$\sqcup$'

Least Upper Bound $\sqcup$

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

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

EXERCISES #1 – #3 (Analysis I)



Automated Software Analysis: Exercises on Dataflow Analysis

1) Undecidability:

Prove that the following problem is *undecidable* (using the "reduction principle"):

- *what are the possible outputs of a program 'P'?*

Let's assume output is done via a special statement (the syntax of which is):

```
STM   ::=   output EXP1 ";"
```

In addition to carrying out the reduction, you need to explain your reasoning.
(Hint: it's quite similar to the examples you saw on slides #16+#18 at the lecture.)

2) Control-Flow Diagrams:

Give a *control-flow template* (as the ones on slides #35+#36) for the "&&"-construction (aka., *"lazy conjunction"*):

```
EXP   ::=   EXP1 "&&" EXP2
```

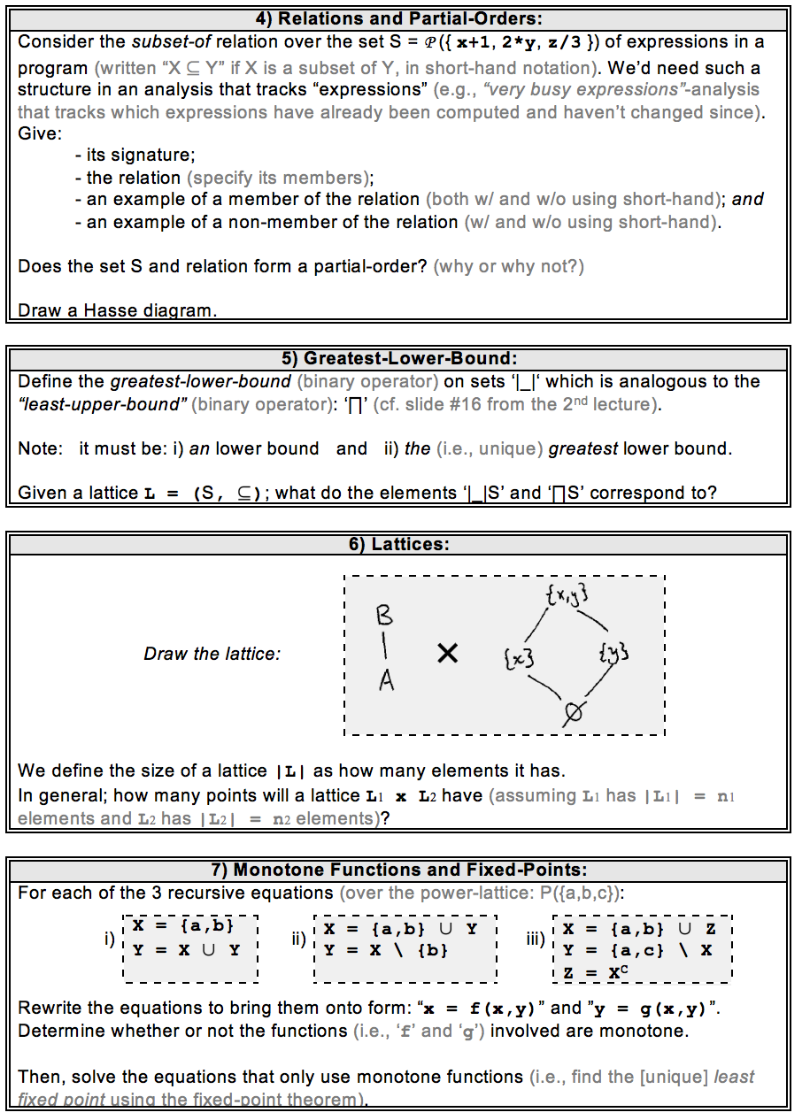
You need to *strictly* adhere to the conventions (of drawing)…:
- ***statements*** as rectangles (with flow *in* and *out*);
- ***expressions*** (of type non-boolean) as rectangles (with flow *in* and *out*);
- ***expressions*** (of type boolean) as diamonds (with single flow in and with boolean flow out as two distinct paths, one for "true" and one for "false"); *and*
- ***confluence*** drawn explicitly as circles (collecting multiple flows of control).

3) Control-Flow Graphs:

Draw a *control-flow graph* for the following (silly) program fragment:

```
int N = 5;
int x=input();
int y=input();
for (int i=1; i<N; i++) {
    if (y!=0 && x/y>2) x = x+1;
    else {
        y = y-1;
        while (x>10) x = x/2;
    }
}
output x;
```

(Note: the program isn't supposed to do anything remotely interesting.)



4) Relations and Partial-Orders:

Consider the *subset-of* relation over the set S = $\mathcal{P}$({ **x+1**, **2*y**, **z/3** }) of expressions in a program (written "X ⊆ Y" if X is a subset of Y, in short-hand notation). We'd need such a structure in an analysis that tracks "expressions" (e.g., *"very busy expressions"*-analysis that tracks which expressions have already been computed and haven't changed since).
Give:
- its signature;
- the relation (specify its members);
- an example of a member of the relation (both w/ and w/o using short-hand); *and*
- an example of a non-member of the relation (w/ and w/o using short-hand).

Does the set S and relation form a partial-order? (why or why not?)

Draw a Hasse diagram.

5) Greatest-Lower-Bound:

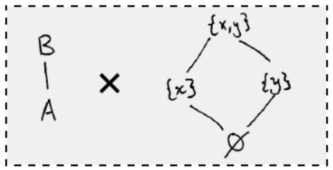
Define the *greatest-lower-bound* (binary operator) on sets '|_|' which is analogous to the *"least-upper-bound"* (binary operator): '∏' (cf. slide #16 from the 2nd lecture).

Note: it must be: i) *an* lower bound and ii) *the* (i.e., unique) *greatest* lower bound.

Given a lattice `L = (S, ⊆)`; what do the elements '|_|S' and '∏S' correspond to?

6) Lattices:

Draw the lattice:


$$\begin{tikzcd} B \arrow[d, no head] & & & \{x,y\} \arrow[dl, no head] \arrow[dr, no head] & \\ A & \times & \{x\} \arrow[dr, no head] & & \{y\} \arrow[dl, no head] \\ & & & \emptyset & \end{tikzcd}$$

We define the size of a lattice `|L|` as how many elements it has.
In general; how many points will a lattice $L_1 \times L_2$ have (assuming L_1 has $|L_1| = n_1$ elements and L_2 has $|L_2| = n_2$ elements)?

7) Monotone Functions and Fixed-Points:

For each of the 3 recursive equations (over the power-lattice: P({a,b,c}):

i)
```
X = {a,b}
Y = X ∪ Y
```
ii)
```
X = {a,b} ∪ Y
Y = X \ {b}
```
iii)
```
X = {a,b} ∪ Z
Y = {a,c} \ X
Z = X^C
```

Rewrite the equations to bring them onto form: "**x = f(x,y)**" and "**y = g(x,y)**".
Determine whether or not the functions (i.e., '**f**' and '**g**') involved are monotone.

Then, solve the equations that only use monotone functions (i.e., find the [unique] *least fixed point* using the fixed-point theorem).