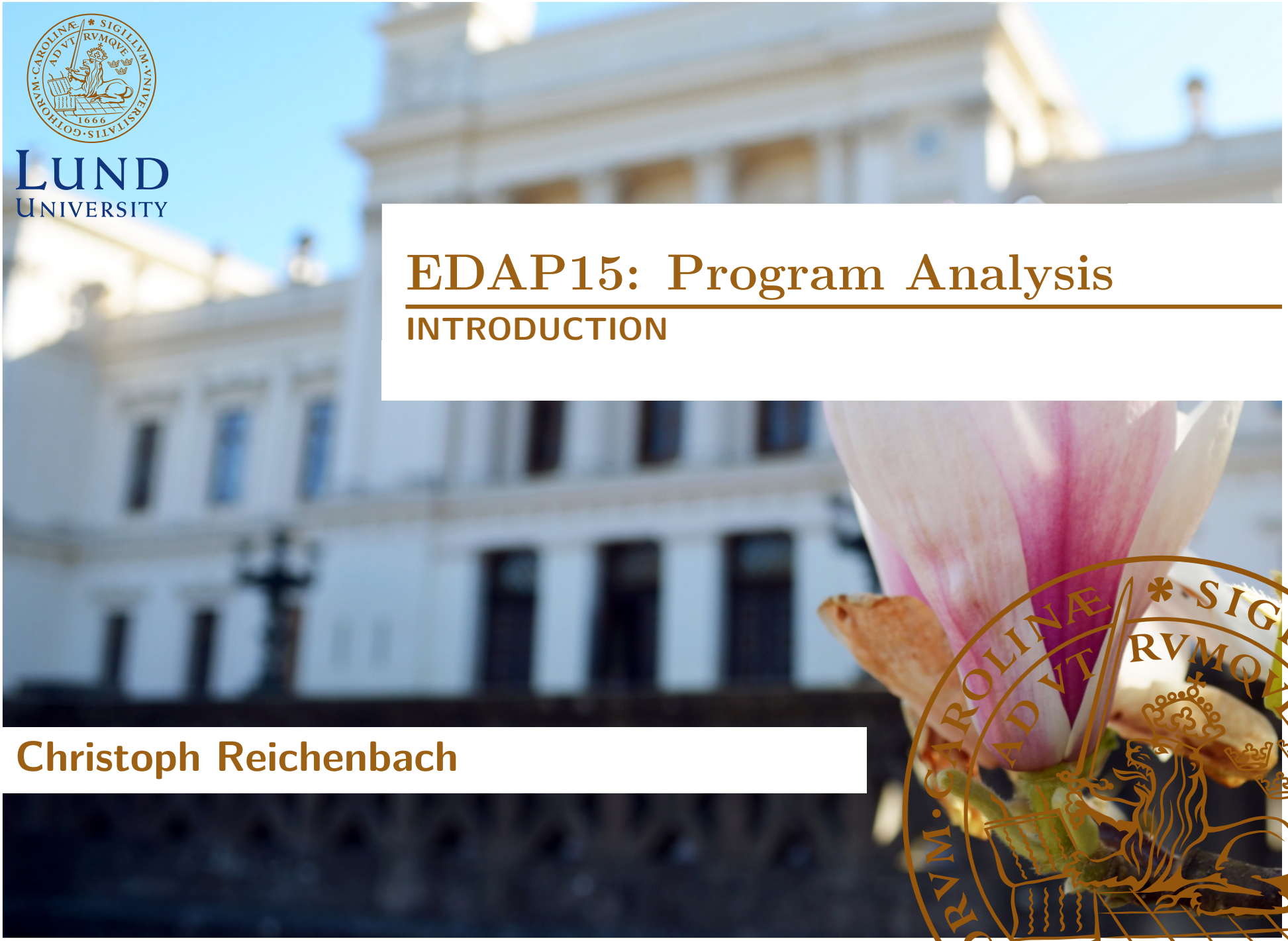


LUND UNIVERSITY

EDAP15: Program Analysis

INTRODUCTION

Christoph Reichenbach

Welcome!

- **EDAP15: Program Analysis**
- **Instructor**: Christoph Reichenbach
  `christoph.reichenbach@cs.lth.se`
- **Teaching Assistant**: Noric Couderc
  `noric.couderc@cs.lth.se`
- **Course Homepage**:
  `http://cs.lth.se/EDAP15`

Course Format

- *Completely Online*
- Lectures
  - Lectures are recorded, available later
  - Later lectures may be *flipped*, will announce
  - Questions:
    - Ask in Zoom text chat
    - Online forum
- Online Quizzes
- Group Exercises
- Oral Exam

Topics

- Concepts and techniques for understanding programs
  - Analysing program structure
  - Analysing program behaviour
- Language focus: Teal, a teaching language
  - Concepts generalise to other mainstream languages:
    - Imperative
    - Object-Oriented

Goals

- **Understand**:
  - What is program analysis (not) good for?
  - What are strenghts and limitations of given analyses?
  - How do analyses influence on each other?
  - How do programming language features influence analyses?
  - What are some of the most important analyses?
- **Be able to**:
  - Implement typical program analyses
  - Critically assess typical program analyses

Resources

- **Course website** (`http://cs.lth.se/EDAP15`)
  - Links to everything listed here
  - List of expected skills
  - Slides
  - Announcements
- Textbooks
- **Moodle**
  - Quizzes
  - Videos
  - Forum
- **Course git** (GitLab)
  - Homework assignments

Textbooks

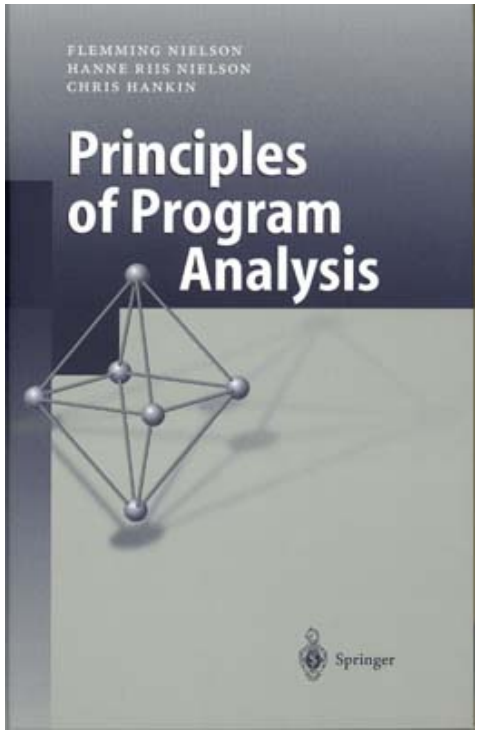
Static Program Analysis

Møller & Schwartzbach

- Optional
- PDF online from authors

Principles of Program Analysis

Nielson, Nielson & Hankin

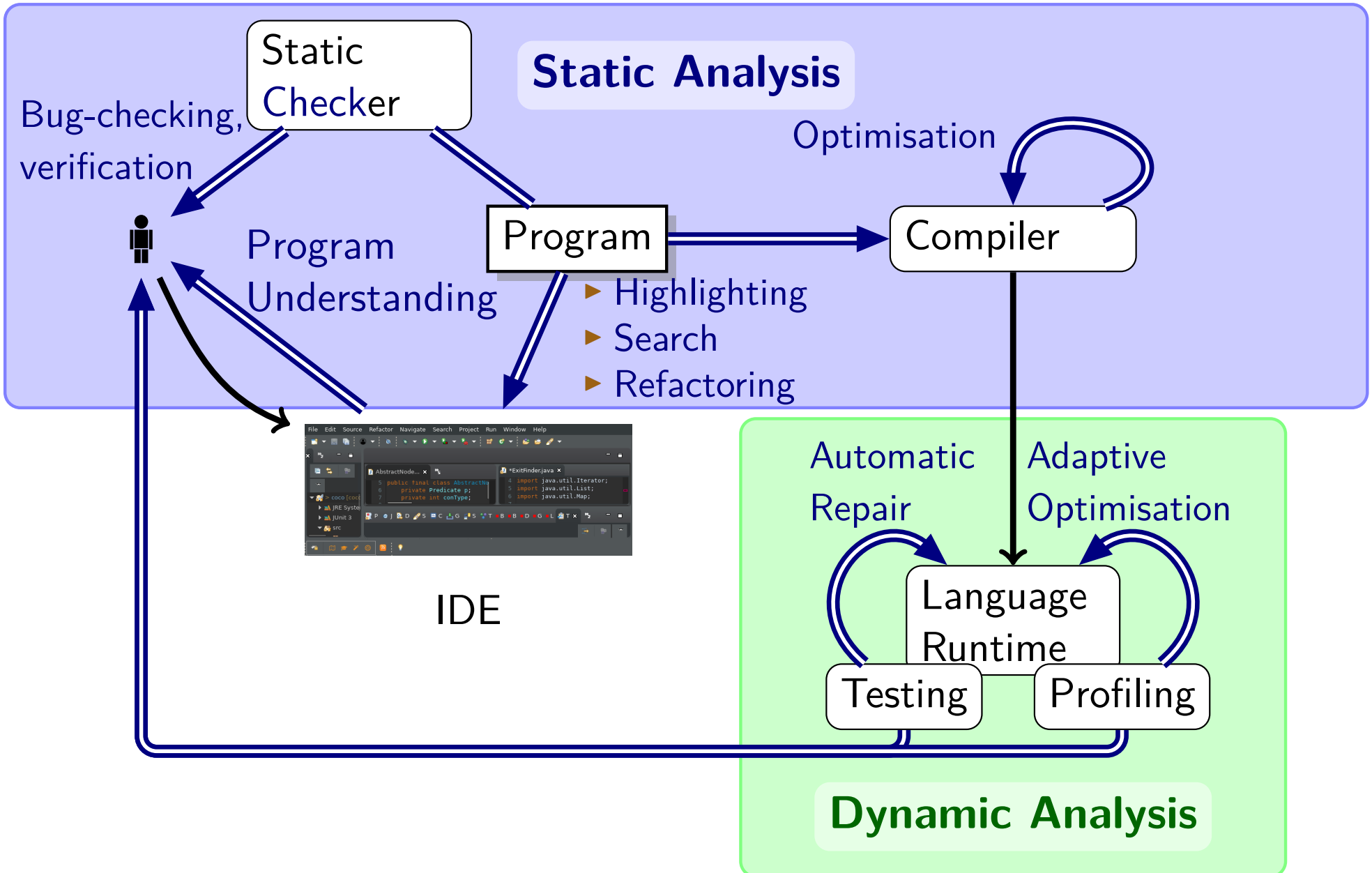


- Optional
- 3 copies in the library
- Theory-driven

How to Pass

- **2020-11-04 18:00**: Form Groups of 2
  - Must be completed this Wednesday, 18:00
  - Contact Noric if you can't find a partner
- **Homework projects**:
  - **Who**: Groups
  - **What**: Implement program analyses in Teal
    - **HW1**/**HW6** are standalone
    - **HW2–4** build on each other
  - **Start**: Homework up every Friday
  - **Grading**:
    - Submit solutions in course git
    - Explain solution to TA
  - **Deadline**:
    - **HW1–5**: Thursday 20:00, 13 days after homework is up
    - **HW6**: 2021-01-12, 20:00
- **Final Exam** starting 2021-01-15
  - **Admission**: Passed all homework projects
  - **Format**: *oral exam*

Uses of Program Analysis



Categories of Program Analyses

	Manual / Interactive	Automatic
Static Analysis	▸ Interactive Theorem Provers	▸ (Most) Type Checkers ▸ Static Checkers (FindBugs, SonarQube, . . .) ▸ Compiler Optimisers
Dynamic Analysis	▸ Debuggers	▸ Unit Tests ▸ Benchmarks ▸ Profilers

- Examines *structure*
- Sees entire program (mostly. . .)

- Examines *behaviour*
- Sees interactions program $\leftrightarrow$ world

Our Focus

Summary

- Program analyses used in *Software Tools*:
  - IDEs
  - Compilers
  - Bug Checkers
  - Run-time systems

  ...
- Main Categories:
  - **Static Analysis**:
    Examine program structure
  - **Dynamic Analysis**:
    Examine program run-time behaviour
  - **Automatic Analysis**:
    "Black Box": Minimal user interaction
  - **Manual / Interactive Analysis**:
    User in the loop
    - Advanced manual analyses exploit automatic analysis

Examples of Program Analysis

Questions:

- 'Is the program well-formed?'

  ```
  gcc -c program.c
  javac Program.java
  ```

  At least for C, C++, Java; not so easy for JavaScript!
- 'Does my factorial function produce the right input in the range 0–5?'

Java

```
@Test // Unit Test
public void testFactorial() {
  int[] expected = new int[] { 1, 1, 2, 6, 24, 120 };
  for (int i = 0; i < expected.length; i++) {
      assertEquals(expected[i], factorial(i));
} }
```

Let's Analyse a Program!

- MISRA-C standard specifies:
  "*The library functions . . . ,* `gets`*, . . . shall not be used.*"
- Given some `program.c`:

```
user@host$ grep 'gets' program.c    # string search
     gets(input_buffer);
     /* The code below gets the system configuration */
     int failed_gets_counter = 0;
 user@host$
```

At least 2 of 3 resuls were wrong: "*False Positives*"

A First Challenge, Continued

```
user@host$ grep 'gets(' program.c
    gets(input_buffer);
user@host$
```

- More precise: no false positives!
- Will this catch *all* calls to gets?

C: program2.c

```c
#include <stdio.h>
void f(char* target_buffer) {
    char *(*dummy)(char*) = gets;
    dummy(target_buffer);
}
```

String search not smart enough: "*False Negative*"

A First Challenge, Continued Again

C: program2.c

```c
#include <stdio.h>
void f(char* target_buffer) {
    char *(*dummy)(char*) = gets;
    dummy(target_buffer);
}
```

```
user@host$ cc -c program.c -o program.o
user@host$ nm program.o
                 # check symbol table in compiled program
0000000000000000 T f
                 U gets          ⟵ Aha!
                 U _GLOBAL_OFFSET_TABLE_
 user@host$
```

Using a more powerful analysis yielded better results

A First Challenge, Solved?

C: program3.c

```
#include<stdio.h>
#include<dlfcn.h>
int f(char* target_buffer) {
    void* handle = dlopen("/lib/x86_64-linux-gnu/libc.so.6",
                          RTLD_LAZY);
    void* sym = dlsym(handle, "gets");
    void(*p)(char*) = sym;
    p(target_buffer);
    return 0;
}
```

- Dynamic library loading: `gets` will not show up in symbol table

Fancier program $\Longrightarrow$ harder analysis

Analysis vs. Property-of-Interest

- **Distinguish**:
  - Property of interest: P
    - All lines that reference `gets`
    - Does the program type-check?
    - Where does the program spend most execution time?
  - Analysis $\mathcal{A}$ that should find P
    - Want for all programs φ:

$$P(\varphi) = \mathcal{A}(\varphi)$$

- Key questions:
  - What data does P depend on?
  - How can $\mathcal{A}$ compute P from that data?
  - How can $\mathcal{A}$ access that data?
    - *Can $\mathcal{A}$ access all relevant data?*

And How Good Is It?

- As we saw, program analyses may be incorrect
- We often describe them with *Information Retrieval* terminology:

	$\mathcal{A}(\varphi)$	*not* $\mathcal{A}(\varphi)$
$P(\varphi)$	**True Positive**	**False Negative**
not $P(\varphi)$	**False Positive**	**True Negative**

- $\#\text{Reports} = \#\text{True Positives} + \#\text{False Positives}$
- **Precision** $= \frac{\#\text{True Positives}}{\#\text{Reports}}$
  "What fraction of our reports was in P?"
- **Recall** $= \frac{\#\text{True Positives}}{\#\text{True Positives}+\#\text{False Negatives}}$
  "What fraction of P did we find?"
- #False Negatives is usually impossible to determine in program analysis

Summary

- Purpose of program analysis $\mathcal{A}$:
  - Compute **Property-of-interest** P
- Program Analysis is nontrivial
  - Programs can hide information that $\mathcal{A}$ wants
  - Analysis $\mathcal{A}$ can misunderstand parts of the program

Soundness and Completeness

Can we always build a $\mathcal{A}$ with $\mathcal{A}(\varphi) = P(\varphi)$?

- Connection to Mathematical Logic:
  - Assume P is boolean
  - $\mathcal{A}$ is **sound** (with respect to P) iff:

$$\mathcal{A}(\varphi) \implies P(\varphi) \qquad \text{(Perfect Precision)}$$

  - $\mathcal{A}$ is **complete** (with respect to P) iff:

$$P(\varphi) \implies \mathcal{A}(\varphi) \qquad \text{(Perfect Recall)}$$

  - $\mathcal{A}(\varphi) = P(\varphi)$ iff $\mathcal{A}$ is both sound & complete

What if $P(\varphi)$ checks whether φ terminates?

The Bottom Line

"Everything interesting about the behaviour of programs is undecidable."

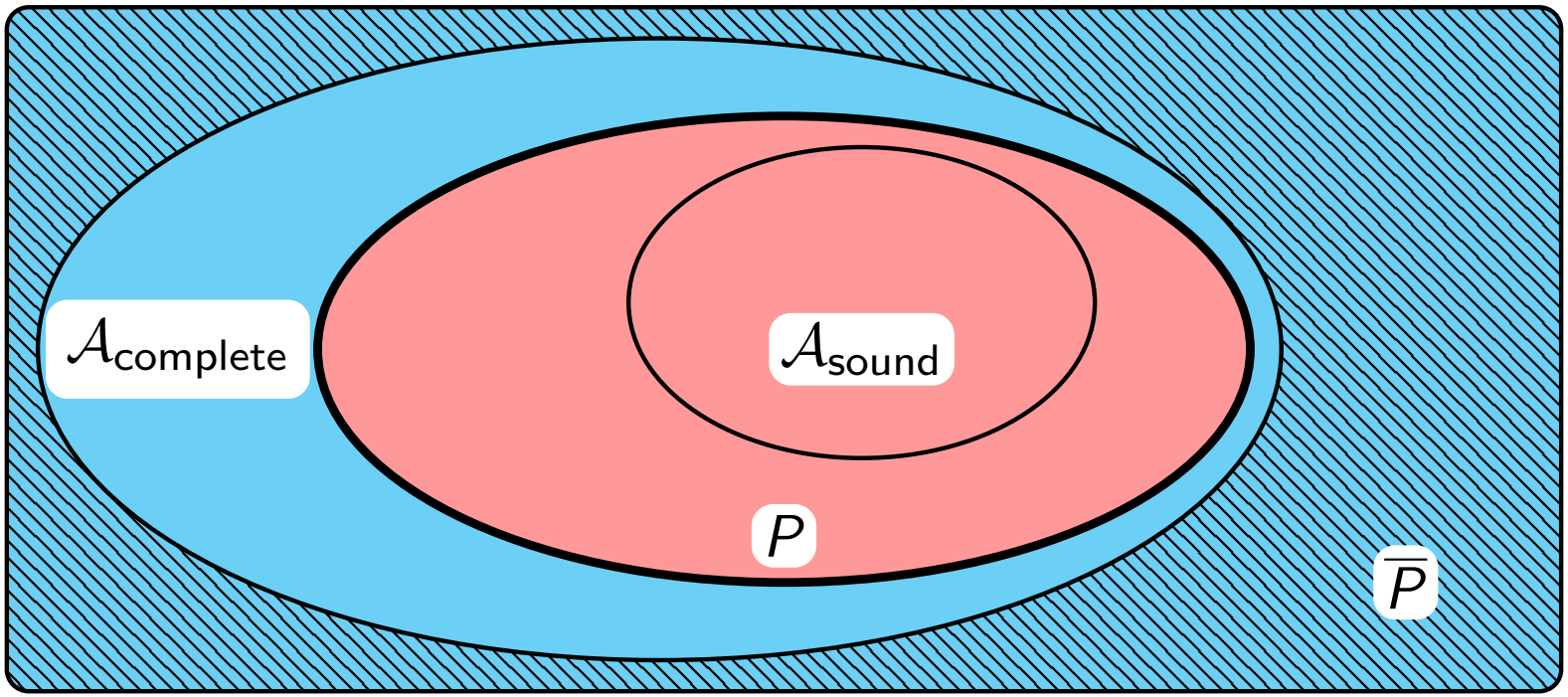
— H.G. Rice [1953], paraphrased by Anders Møller

We must choose:

- **Soundness**
- **Completeness**
- **Decidability**

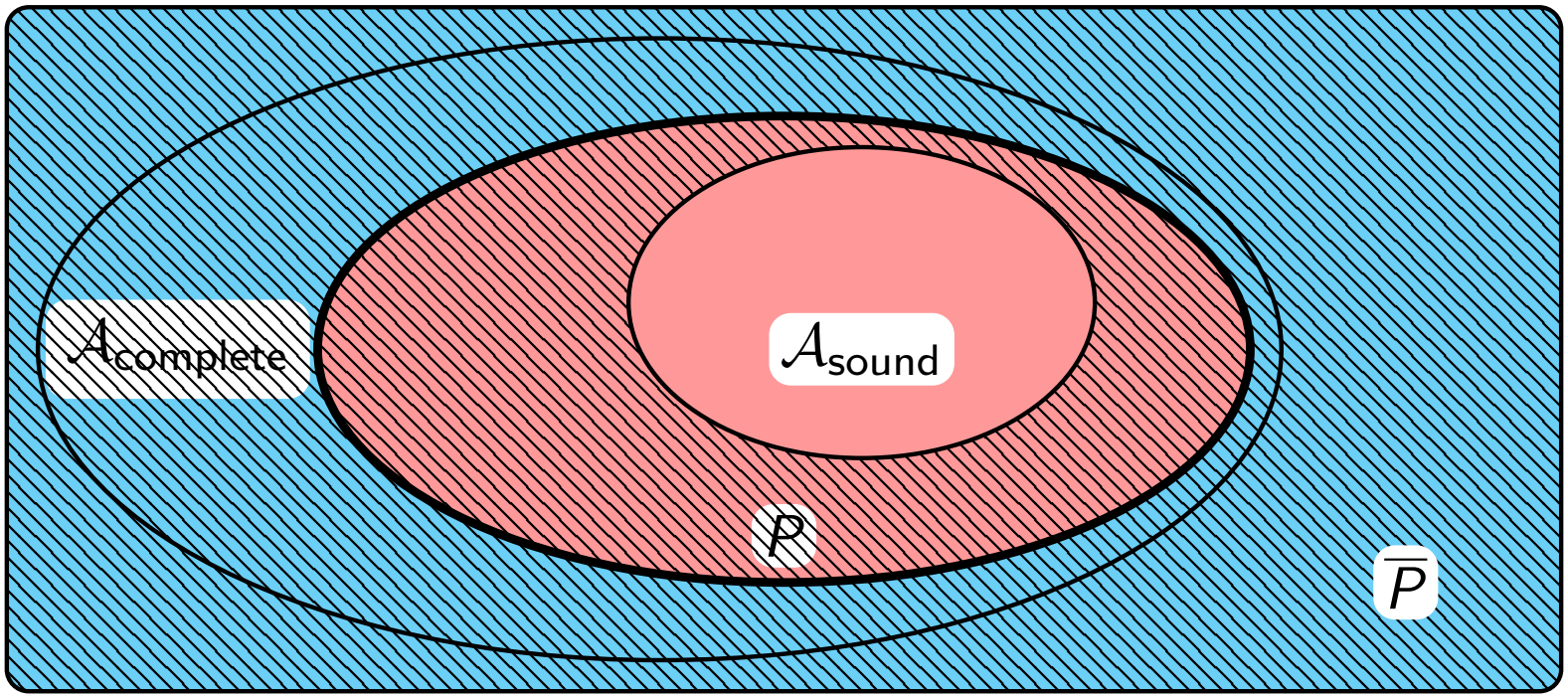
. . . pick any two.

Soundness and Completeness: Caveat



- *Beware*: "sound" and "complete" be confusing:
  - Example: $P(\varphi)$ is "φ has a bug"
  - If you now want to check $\overline{P}$, the *negation* of P:
    - $\overline{P}(\varphi)$ is "φ does not have a bug"
    - $\overline{\mathcal{A}_{\text{complete}}}$ (= run $\mathcal{A}_{\text{complete}}$ and invert output) is *sound wrt* $\overline{P}$

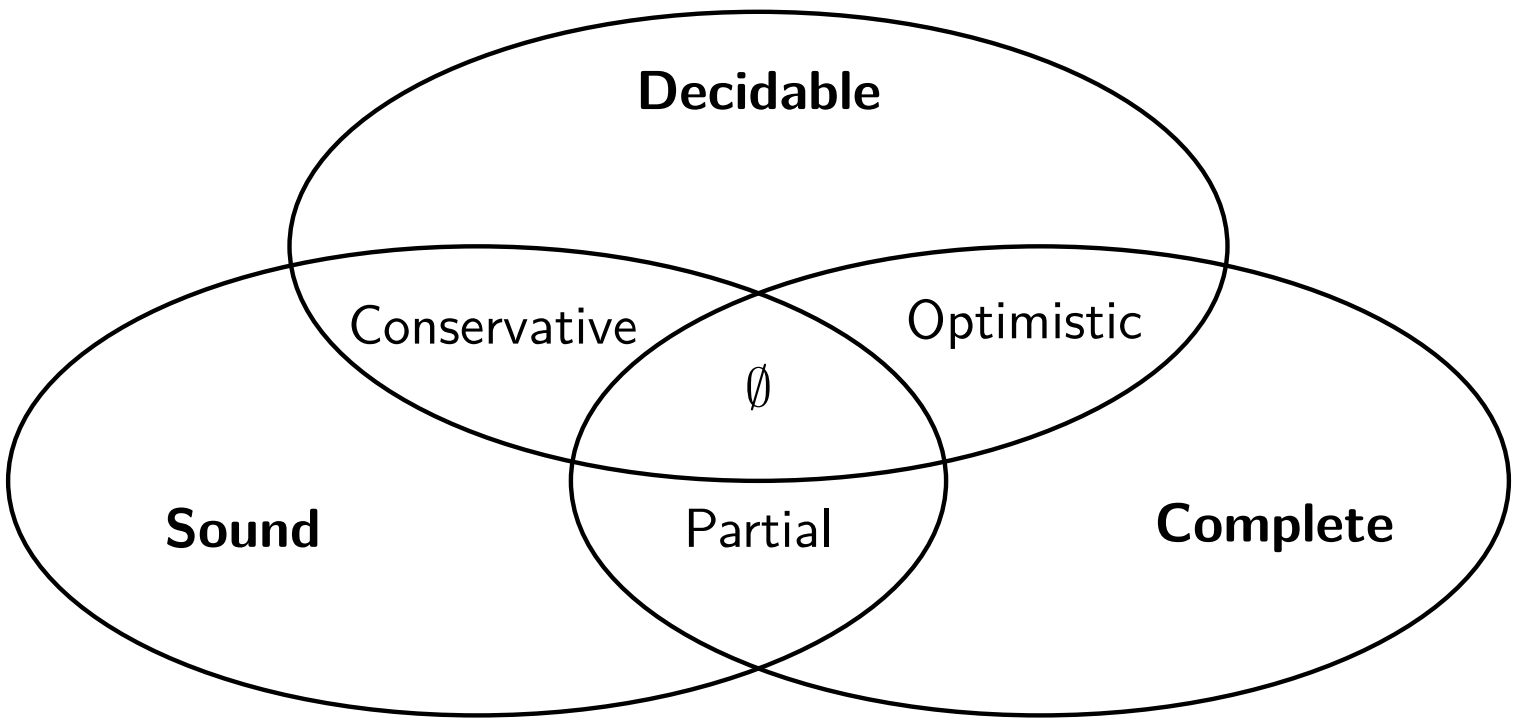
Soundness and Completeness: Caveat



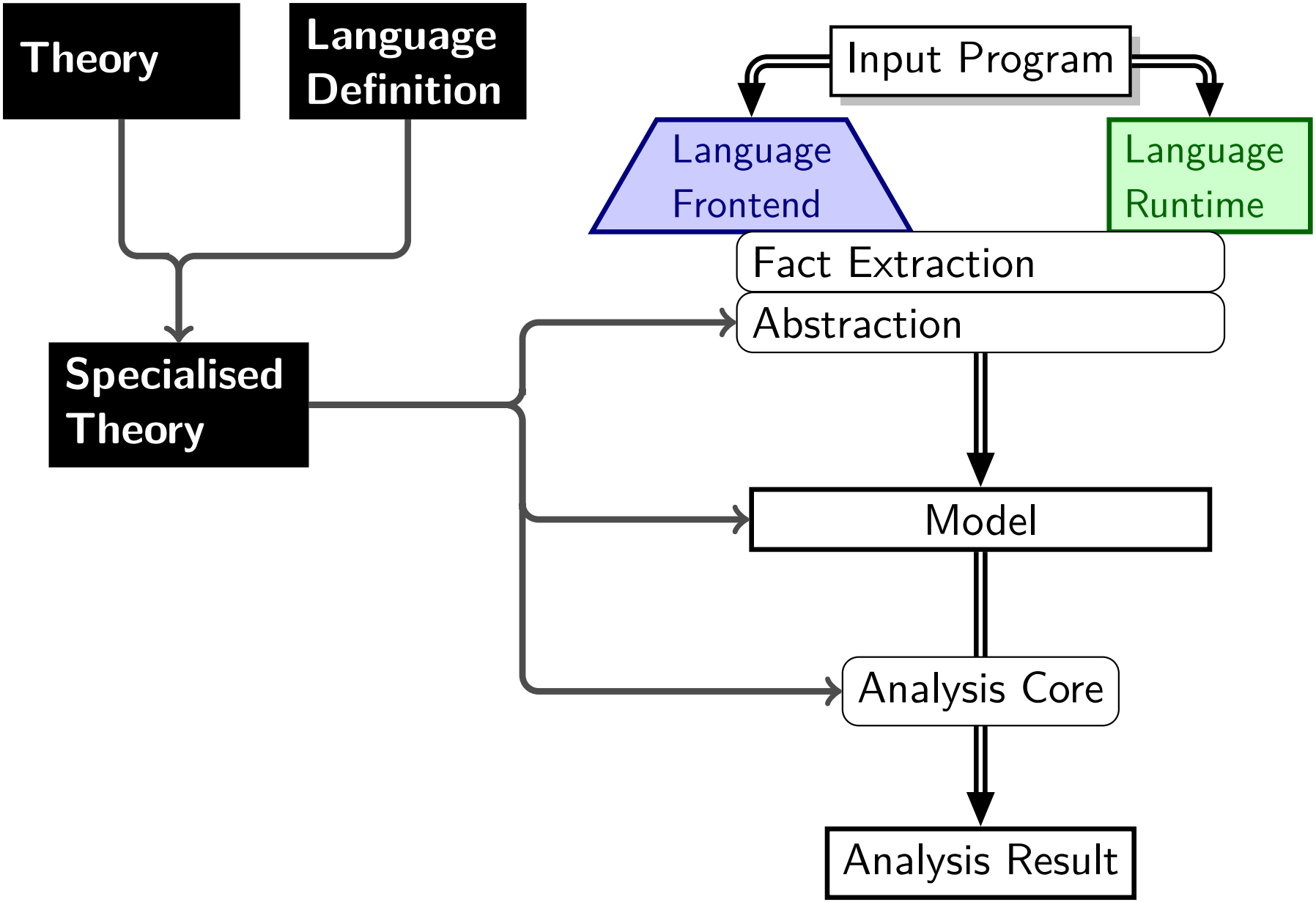
- *Beware*: "sound" and "complete" be confusing:
  - Example: $P(\varphi)$ is "φ has a bug"
  - If you now want to check $\overline{P}$, the *negation* of P:
    - $\overline{P}(\varphi)$ is "φ does not have a bug"
    - $\overline{\mathcal{A}_{\text{complete}}}$ (= run $\mathcal{A}_{\text{complete}}$ and invert output) is *sound wrt* $\overline{P}$
    - $\overline{\mathcal{A}_{\text{sound}}}$ is *complete wrt* $\overline{P}$

Summary

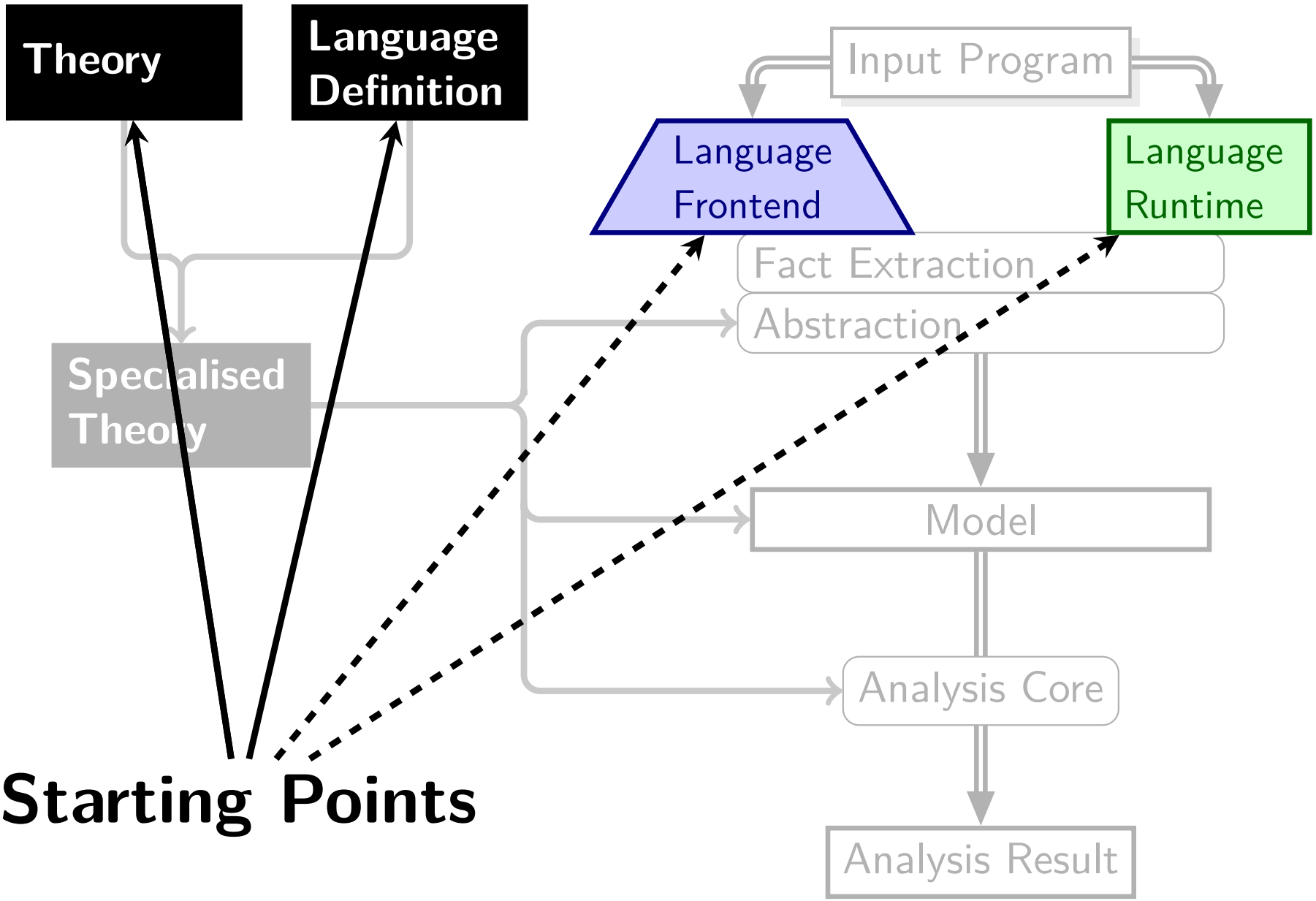
- Given property P and analysis $\mathcal{A}$:
  - $\mathcal{A}$ is **sound** if it triggers only on P
    $P =$ "program has bug": $\mathcal{A}$ reports *only* bugs
  - $\mathcal{A}$ is **complete** if it always triggeres on P
    $P =$ "program has bug": $\mathcal{A}$ reports *all* bugs
- If P is nontrivial (i.e., depend on behaviour):



Building a Program Analysis

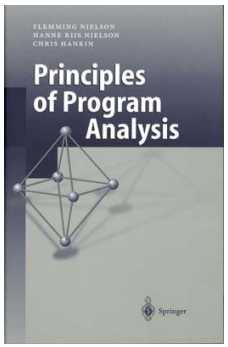


Building a Program Analysis

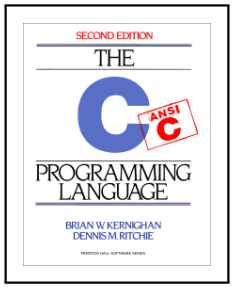


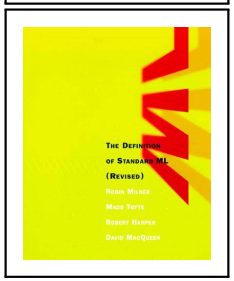
Gathering Our Tools

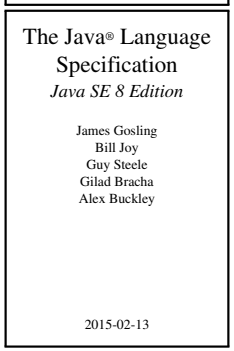
Theories



Language Definitions







Tools

Compilers

Analysis Frame-works

jastadd

PHASAR

Astrée

FindBugs™

Soot

Hardware

Language Definitions

- Pure theory
- Define structure (syntax) and meaning (semantics) of language
- Abstracts over many details

Syntax example:

$$
\begin{array}{rcl}
e & ::= & \text{zero} \\
 & | & \text{one} \\
 & | & \langle e \rangle + \langle e \rangle \\
 & | & \langle e \rangle \text{-} \langle e \rangle \\
 & | & \text{neg } \langle e \rangle \\
 & | & (\ \langle e \rangle\ ) \\
 & | & \text{log } \langle e \rangle
\end{array}
$$

- **Property of Interest**: Does a given program $\varphi \in e$ compute a positive number?

First, we must understand the language *semantics*

Language Definitions: Semantics

- Language Definitions also specify *Semantics*:
  - **Static Semantics**:
    - Connect parts of the program structure (variables, functions, classes, . . .)
    - Enforce restrictions (e.g., via *type checking*)
  - **Dynamic Semantics**:
    - Specify program run-time behaviour
- We will (mostly) treat semantics informally in this course
- Here: assume "obvious" semantics

Simplifying the Lanugage

- Let's make it easier to analyse the language
- We don't need parentheses for the analysis
- a-$b = a$+neg b
  $\Rightarrow$ Abstraction (we join similar problems into one)
- log is too difficult
  $\Rightarrow$ *Restrict to sub-language (give up on some problems)*

$$
\begin{array}{rcl}
e & ::= & \text{zero} \\
 & | & \text{one} \\
 & | & \langle e \rangle + \langle e \rangle \\
 & & \\
 & | & \text{neg } \langle e \rangle
\end{array}
$$

Simplification helps us get started, but restricting to a sublanguage can quickly render an analysis impractical

Finding a Good Theory

- Recall: $P(\varphi)$ should detect if φ computes positive result
- There are many theories for program analysis
- We pick *Abstract Interpretation* (Patrick & Radhia Cousot):
  - Map all values to a simpler *abstract domain*
  - Map all operations so they respect the abstraction
- For example: classify programs into *abstract domain* containing:
  - D^0: Computes 0
  - D^+: Computes a positive value
  - D^-: Computes a negative value
- Notation: $\boxed{\varphi \rightsquigarrow^D a}$, where a is one of D^0, D^+, D^-

Semantics

$$
\begin{aligned}
\ominus D^0 &= D^0 \\
\ominus D^+ &= D^- \\
\ominus D^- &= D^+ \\
\ominus D^? &= D^?
\end{aligned}
$$

$$
\begin{array}{lcl}
e & ::= & \mathsf{zero} \\
 & | & \mathsf{one} \\
 & | & \langle e \rangle + \langle e \rangle \\
 & | & \mathsf{neg}\ \langle e \rangle
\end{array}
$$

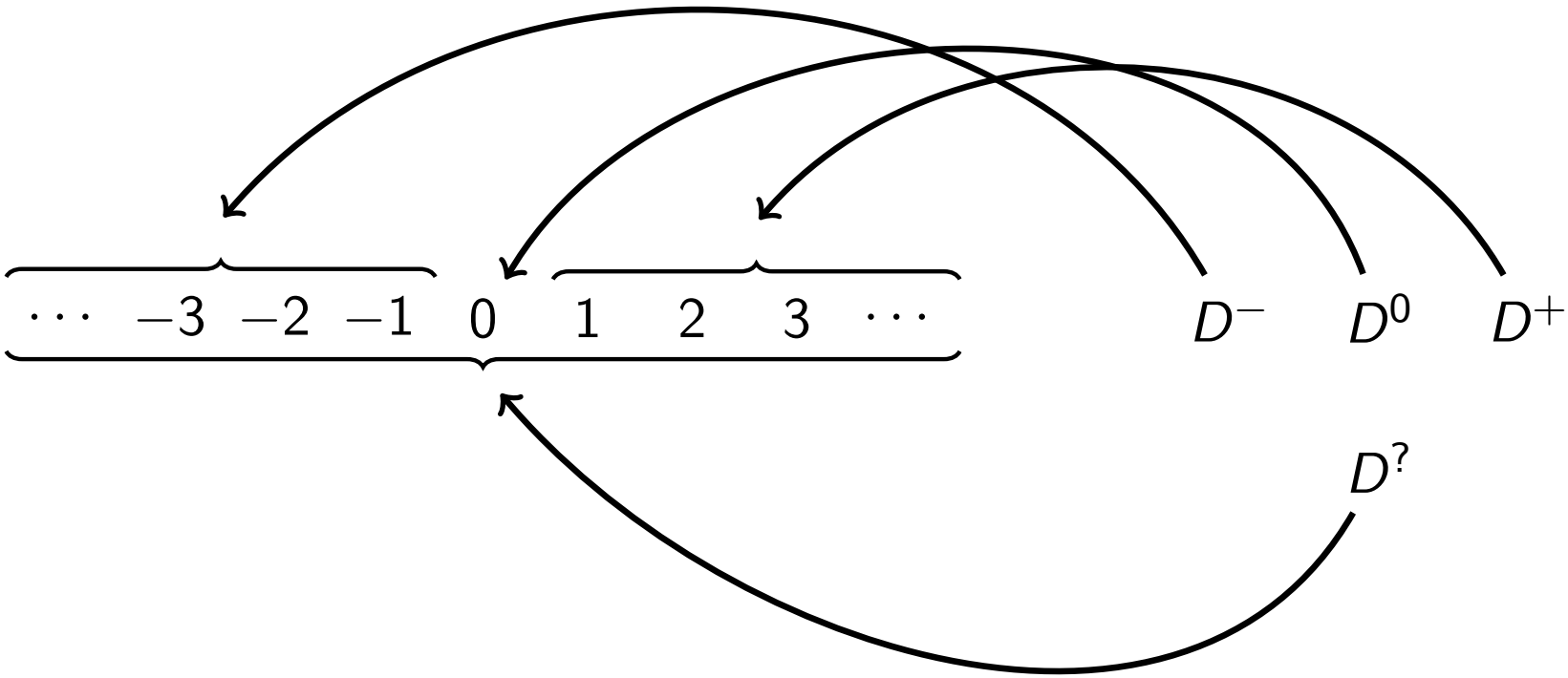
$$
a_1 \oplus a_2 = \left\{
\begin{array}{c||c|c|c}
 & D^+ & D^0 & D^- \\ \hline\hline
D^+ & D^+ & D^+ & D^? \\ \hline
D^0 & D^+ & D^0 & D^- \\ \hline
D^- & D^? & D^- & D^-
\end{array}
\right.
$$

$$D^? \oplus a = D^? = a \oplus D^?$$

$$\mathsf{zero} \rightsquigarrow^D D^0 \qquad \mathsf{one} \rightsquigarrow^D D^+$$

$$\frac{\text{if } x \rightsquigarrow^D a \text{ then}}{\mathsf{neg}\ x \rightsquigarrow^D \ominus a} \qquad \frac{\text{if } x \rightsquigarrow^D a_1 \text{ and } y \rightsquigarrow^D a_2 \text{ then}}{x + y \rightsquigarrow^D a_1 \oplus a_2}$$

Correspondence: Concrete and Abstract



Also:

- $\ominus$ *"is compatible with"* neg
- $\oplus$ *"is compatible with"* $+$

Abstract Interpretation **explores these ideas in great detail**

Building a Better Analysis

Property of Interest: Does a given program $\varphi \in e$ compute a positive number?

$$
\begin{array}{rcl}
e & ::= & \mathsf{zero} \\
& | & \mathsf{one} \\
& | & \langle e \rangle + \langle e \rangle \\
& | & \langle e \rangle \text{-} \langle e \rangle \\
& | & \mathsf{neg}\ \langle e \rangle \\
& | & (\ \langle e \rangle\ ) \\
& | & \mathsf{log}\ \langle e \rangle
\end{array}
$$

- How else could we analyse this for a given program φ?
  - Just *run φ and check the result*
  - In general: only feasible for very restricted languages
  - Infeasible as soon as we add parameters or recursion

Summary

- We can mathematically formalise *syntax* and *semantics*
- Semantics derive from syntax
- **Abstract Interpretation**: Theory for program analysis
  - Map program semantics into *abstract domain*
  - Map operations to *compatible* operations on abstract domain
  - Challenge: remain precise yet decidable
  - Foundation to other static analysis theories

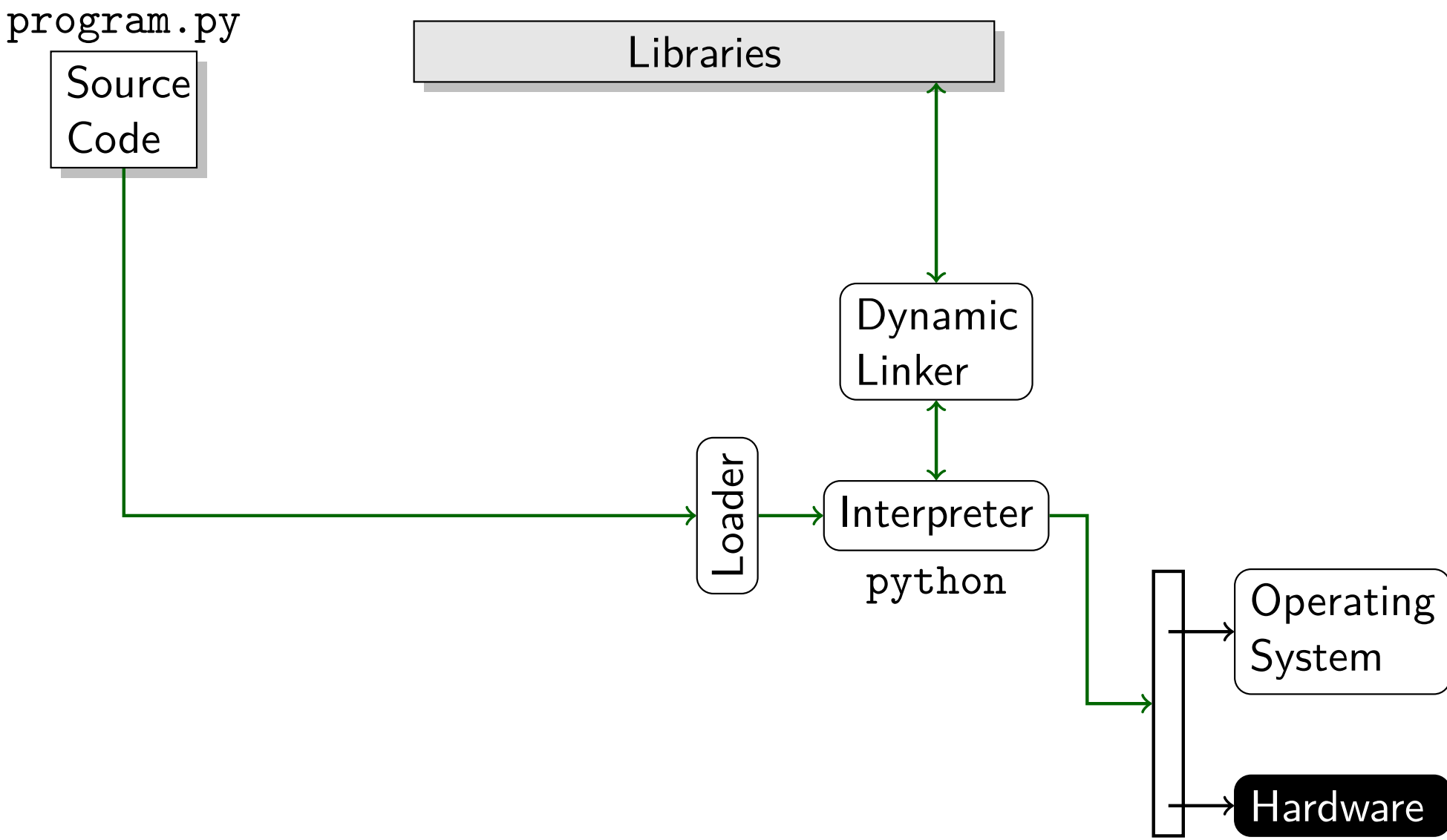
Some Theories: Dynamic

- **Testing**
- **Input Generation and Fuzzing**
  Call program with "random" inputs to see if it crashes
  - **Dynamic Invariant Detection**
    Record program behaviour, describe concisely (e.g., "always returns zero")
  - **Test Generation**
    Automatically build test suite to capture current behaviour
- **Monitoring**
  Capture which parts of the code execute
- **Benchmarking**
  Measure resource usage (time, memory, bandwidth)

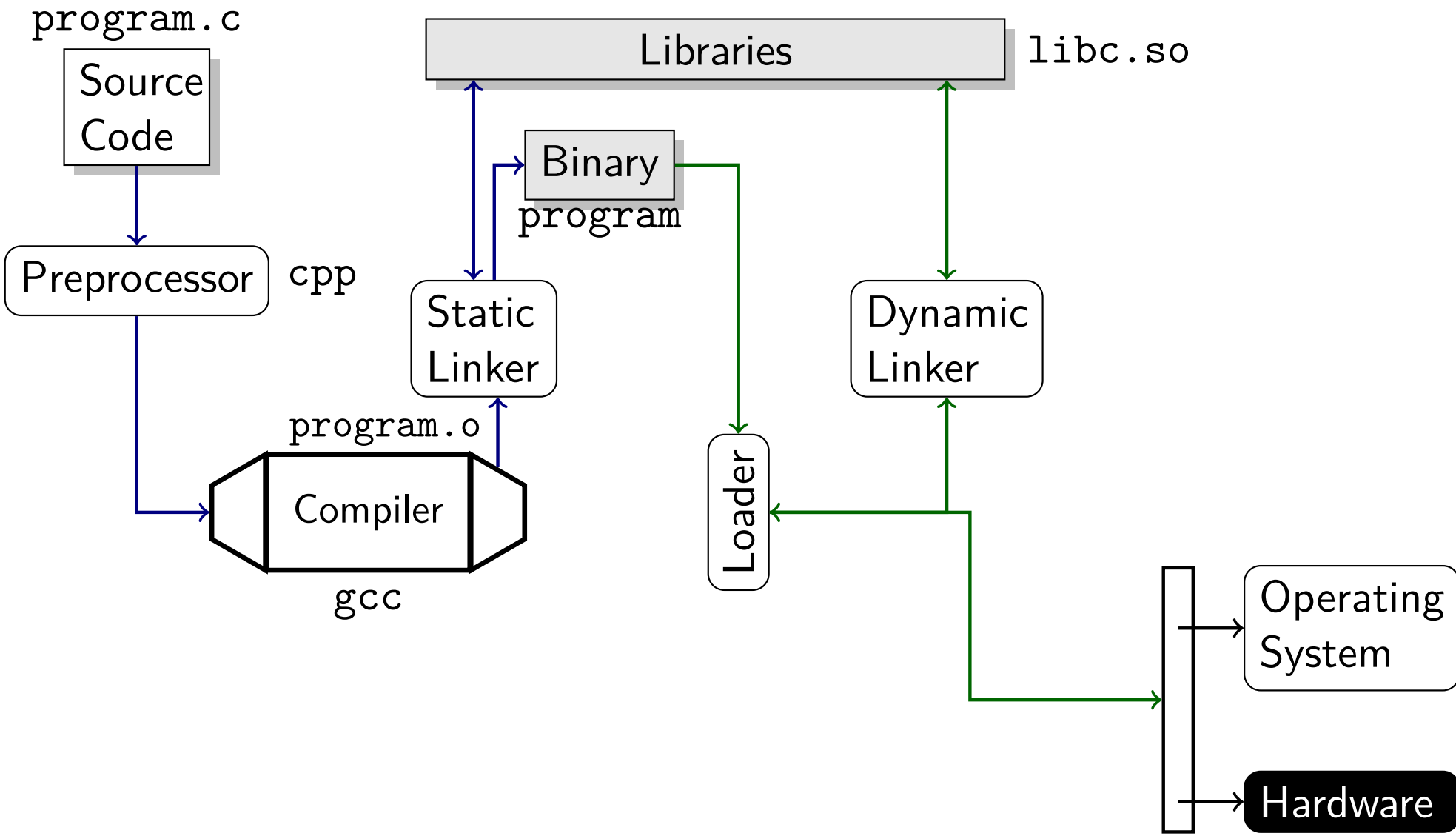
Some Theories: Static

- **Abstract Interpretation**
  - **Type Theory**
    3 : Int
  - **Dataflow Analysis Theory**
    What values can my variables store?
  - **Hoare Logic**
    If $x > y$ holds before I run x := x-1, is $x \geq y$ true?
  - **Symbolic Execution**
    Use SMT solving to find properties of variables and possible execution paths
  - **Model Checking**
    Given some formal model (e.g., a FSM), does it describe my code?
  - **Statistical Analysis**

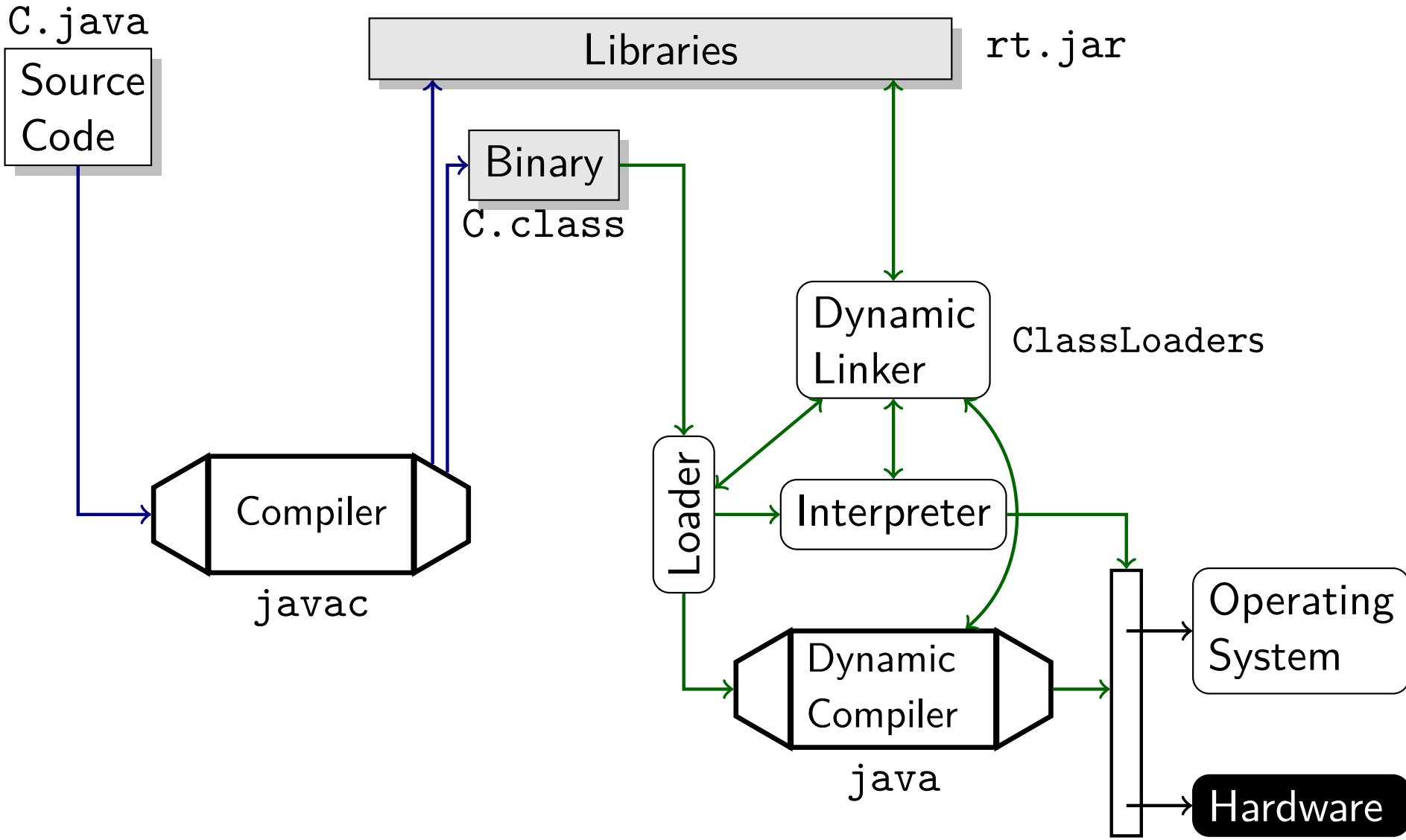
Program Execution Pipeline



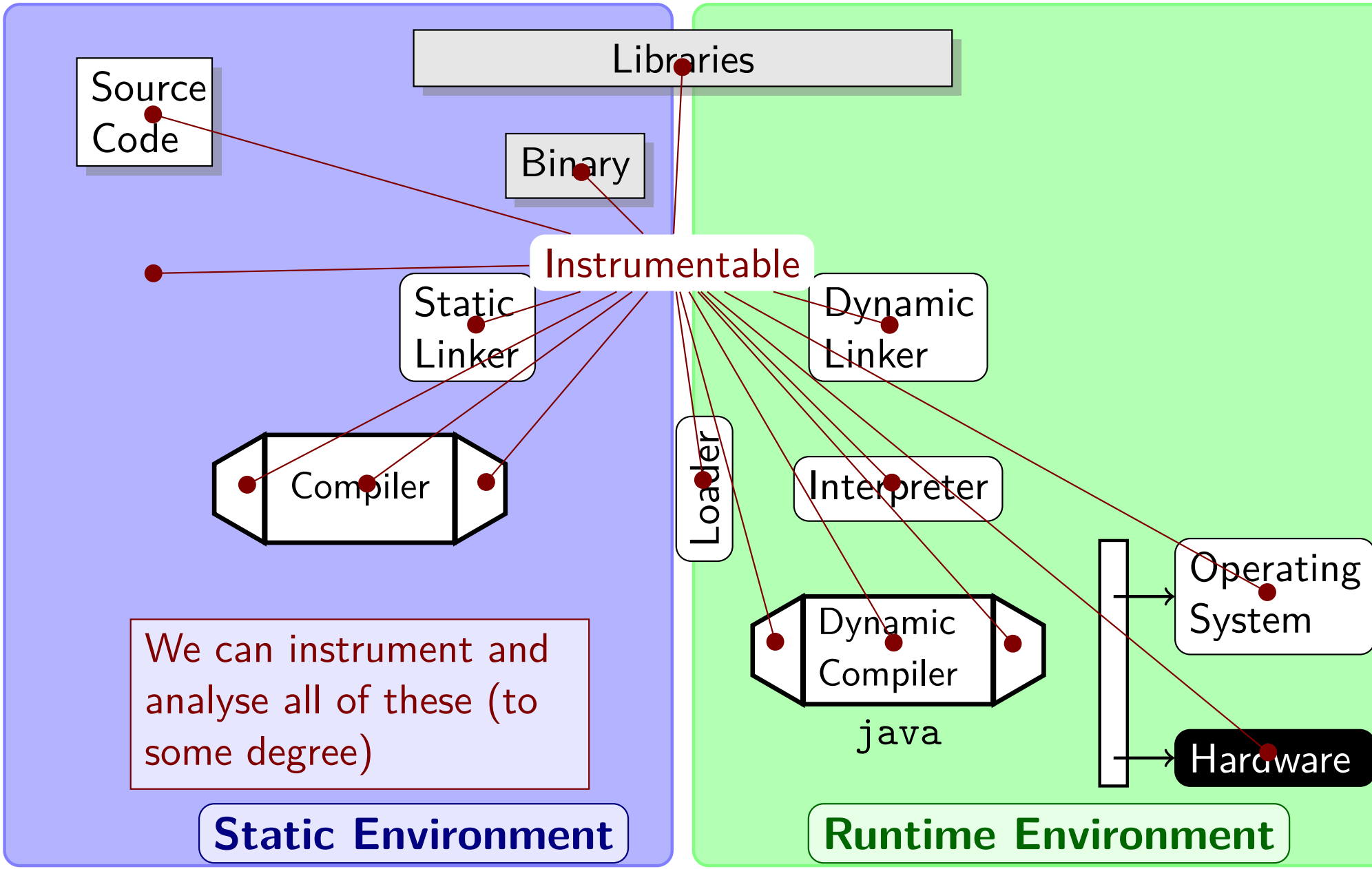
Program Execution Pipeline



Program Execution Pipeline



Program Execution Pipeline



Static vs. Dynamic Program Analyses

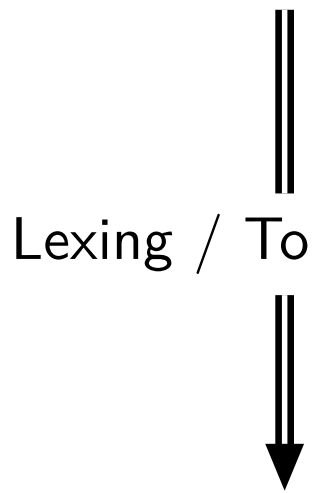
	Static Analysis	Dynamic Analysis
Principle	Analyse program structure	Analyse program execution
Input	Independent	Depends on input
Hardware/OS	Independent	Depends on hardware and OS
Perspective	Sees everything	Sees that which actually happens
Completeness (bug-finding)	Possible	Must try all possible inputs. . .
Soundness (bug-finding)	Possible	Always, for free

Summary

- **Preprocessor**: Transforms source code before compilation
- **Static compiler**: Tranlates source code into executable (machine or intermediate) code
- **Interpreter**: Step-by-step execution of source or intermediate code
- **Dynamic (JIT) compiler**: Translates code into machine-executable code
- **Loader**: System tool that ensures that OS starts executing another program
- **Linker**: System tool that connects references between programs and libraries
  - **Static linker**: Before running
  - **Dynamic linker**: While running
- **Machine code**: Code that is executable by a machine
- **Static Analysis**: Analyse program without executing it
- **Dynamic Analysis**: Analyse program execution

Java lexing

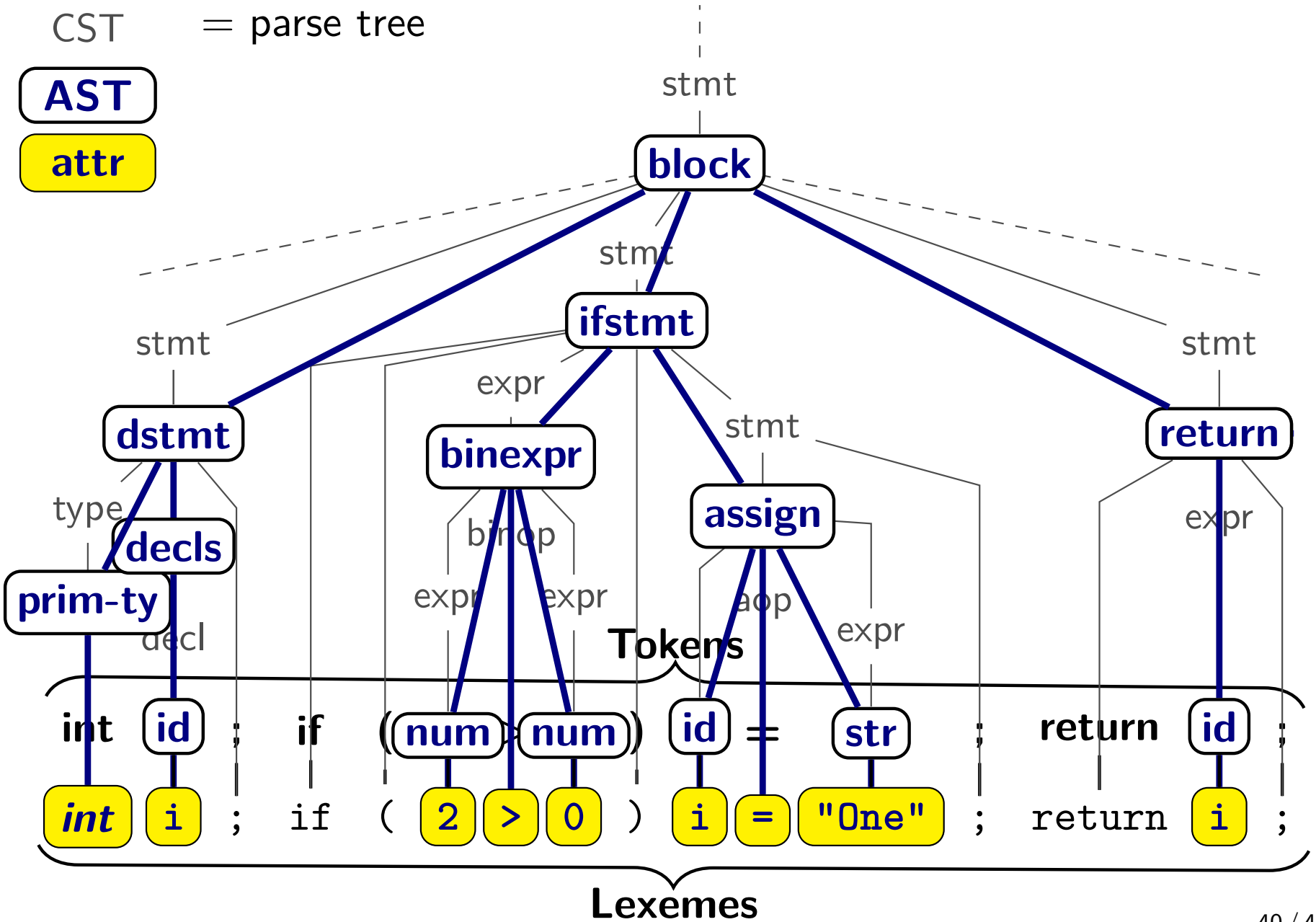
```
int i;
if (2 > 0) {
  i = "One";
}
return i;
```



Lexing / Tokenisation

```
int i ; if ( 2 > 0 ) i = "One" ; return i ;
```

Java lexing & parsing



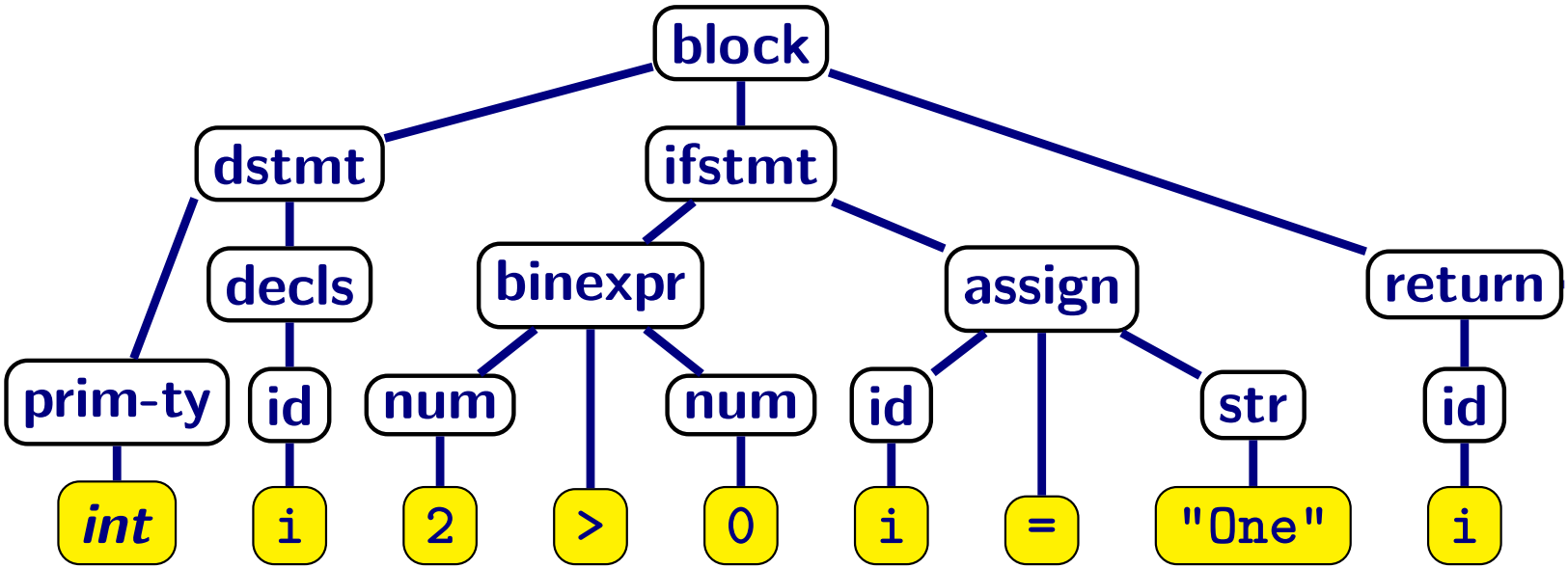
Parsing in general

*Translate text files into **meaningful** in-memory structures*

- CST = Concrete Syntax Tree
  - Full "parse", cf. language BNF grammar
  - Not usually materialised in memory
- AST = Abstract Syntax Tree
  - Standard in-memory representation
  - Avoids syntactic sugar from CST, preserves important nonterminals as **AST nodes**
  - Converts useful tokens into attributes
- The AST is the most common **Intermediate Representation** (IR) of program code
  - Effective for frontend analyses
  - Other IRs focus e.g. on optimisations in the backend

Program analysis starts on the AST

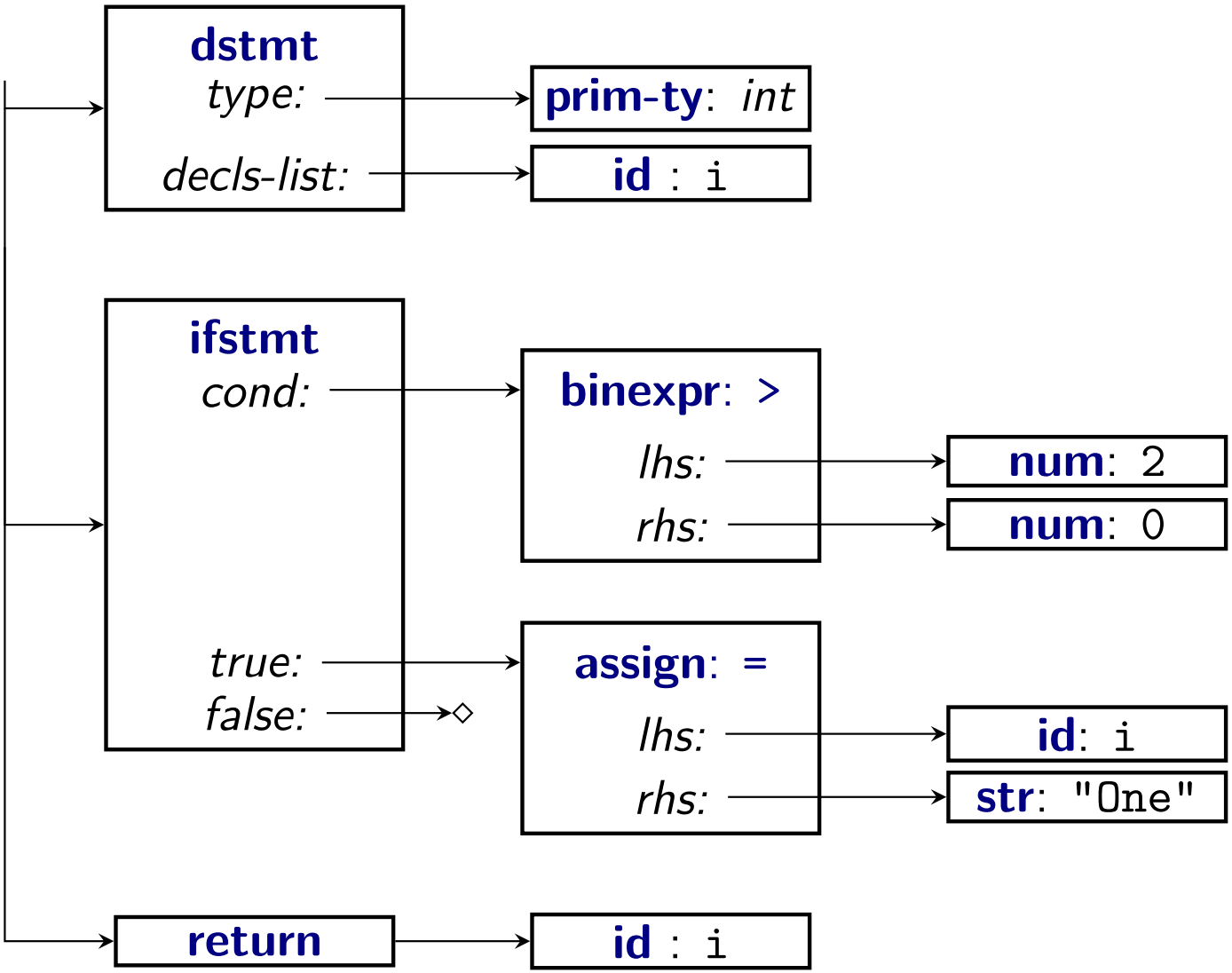
In-Memory Representation



Typical in-memory representations for this AST:

- Algebraic values (functional)
- Records (imperative)

In-Memory Representation



Program Analysis

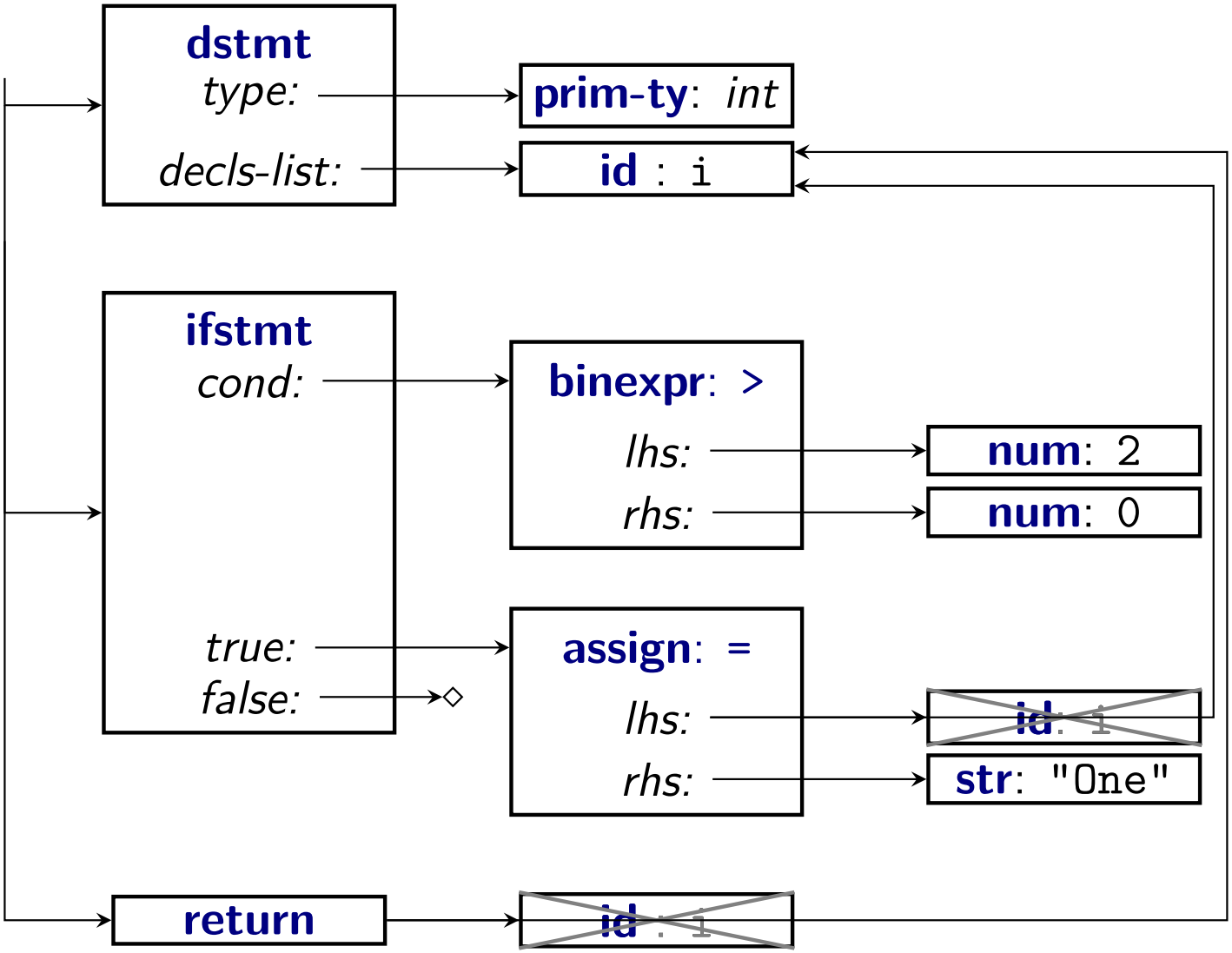
We run numerous code analyses on the AST:

- *Name Analysis*:
  - Which name *use* binds to which *declaration*?
- *Type Analysis*:
  - What are the types of all expressions?
- *Static Correctness Checks*:
  - Are there type errors?
  - Is a variable unused?
  - Are we initialising all variables?

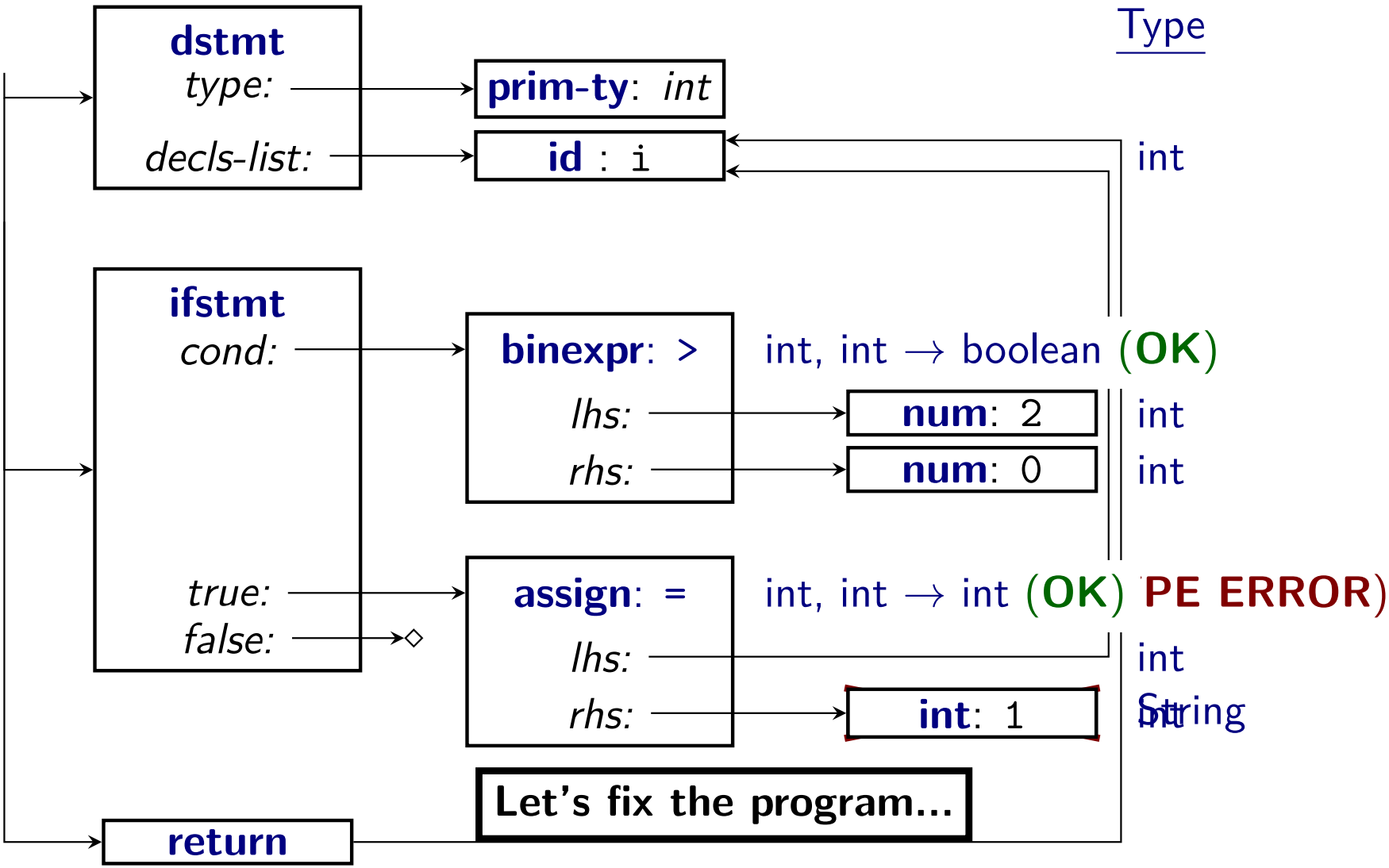
  ...
- *Optimisations*:
  - Can we speed up the program somehow?

Advanced static correctness checks increasingly common in compilers

Name Analysis



Type Analysis



Summary

- Compiler represents programs in **intermediate representations** (IRs)
- Compiler can be separated into:
  - *Frontend*: process incoming source code, generate IR
  - *Middle-end*: optimise IR
  - *Back-end*: translate IR into executable code
- Parser matches *concrete syntax tree* (CST), generates *abstract syntax tree* (AST)
- Typical analyses on AST:
  - *Name analysis*: which variable use belongs to which definition?
  - *Type analysis*: do variable/operator/function types agree? Any implicit conversions needed?

  ...

Outlook

- **Remember**:
  - Join Moodle if you have not done so yet
  - Form groups by Wednesday, 18:00!
- Next Lecture (Wednesday, same channel):
  - Type-Based Analysis

`http://cs.lth.se/EDAP15`