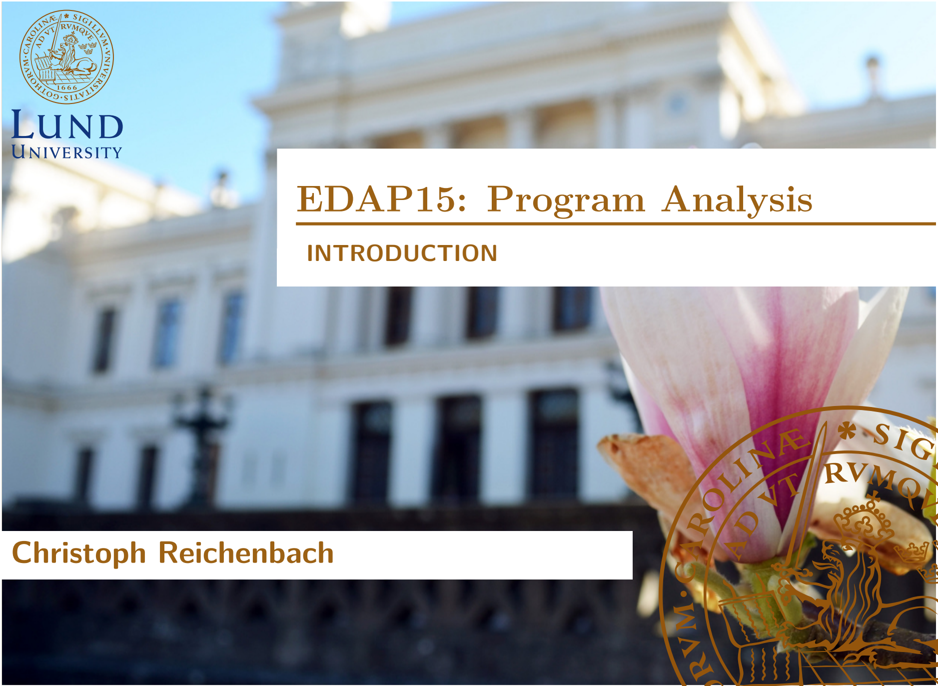


Lund University

EDAP15: Program Analysis

INTRODUCTION

Christoph Reichenbach

Welcome!

- **EDAP15: Program Analysis**
- **Instructor**: Christoph Reichenbach
  `christoph.reichenbach@cs.lth.se`
- **Teaching Assistants**:
  - Alexandru Dura
  - Anton Risberg Alakűla
- **Course Homepage**:
  `http://cs.lth.se/EDAP15`

Course Format

- **Moodle**: Sign up today!
- **Lectures**
  - In Person
  - Partially 'Flipped':
    - Check Moodle for videos to watch before lecture
- **Core material**
  - Lectures (*bring your questions!*)
  - Videos
- **Self-Study material**
  - Online Quizzes
  - Textbooks (optional)
- **Questions**
  - *Ask in class*
    - Ask-and-Upvote system (or just raise your hand!)
  - Online forum
  - Office hours
- **Mandatory Activities**: Homework & Quizzes

Topics

- Concepts and techniques for understanding programs
  - Analysing program structure
  - Analysing program behaviour
- Practical concerns in program analysis

Language focus: Teal, a teaching language

- Concepts generalise to other mainstream languages:
  - Imperative
  - Object-Oriented

Goals

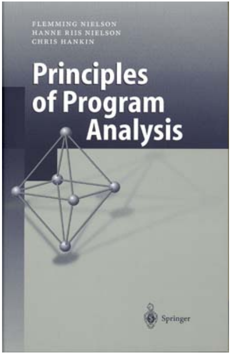
- **Understand**:
  - What is program analysis (not) good for?
  - What are strenghts and limitations of given analyses?
  - How do analyses influence each other?
  - How do programming language features influence analyses?
  - What are some of the most important analyses?
- **Be able to**:
  - Implement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

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 This Course

- **This Week**
  1. **today**: register in Moodle
  2. **2024-01-17, 18:00**: Find lab partner, register for lab slot
  3. **2024-01-19, 18:00**: Mandatory quizzes in Moodle (see below)
- **Every Week**
  1. Work on homework exercises
  2. Present homework solutions to TAs (labs or Zoom)
  3. **Fri**: Lab slots (for help & presenting solutions)
  4. **Fri, 18:00**: Mandatory quizzes in Moodle
     - Score 70% to pass
     - Your best attempt counts
     - No limit on number of retries

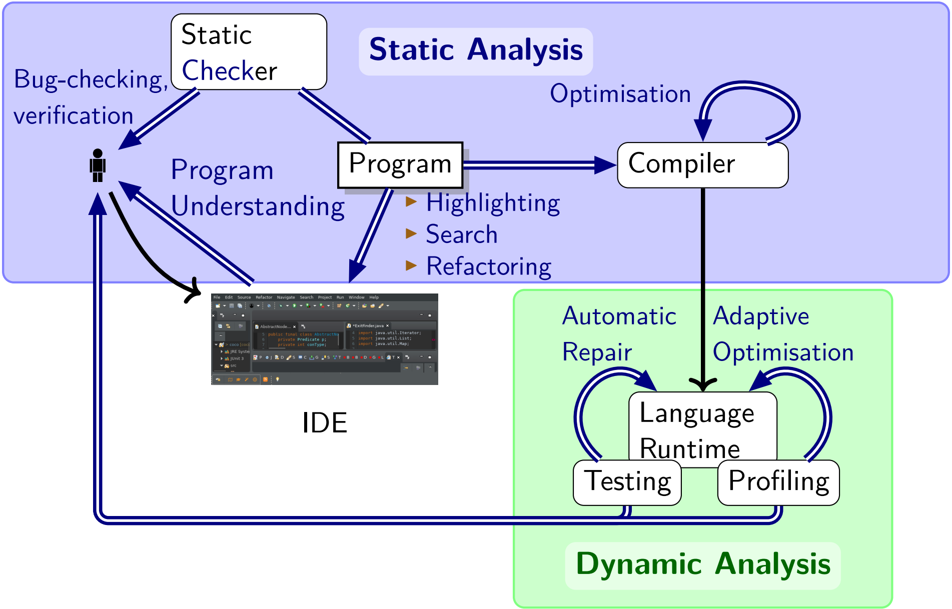
Passing vs. Grades

- Passing these requirements gives you a grade of **3**
  - TAs must have approved all homework exercise solutions
- For higher grades (**4**, **5**):
  - Additional oral exam
  - Registration opens after course completion

Homework Exercises

- **Exercises**:
  - Exercise 0: Group exercise, W3 (**this** week!)
  - Exercise 1: Group exercise, W4
  - Exercise 2–4: Solo exercises, W6/7/8
- **To pass**:
  - Pass our internal tests
  - **Explain**(!) implementation and rationale to TA
- **Presenting to TAs**
  - You can present *once a week*
    Additional slots depend on TA capacity
  - Zoom or during lab hours
- **Labs** for help with / presenting homework exercises
  - *Get started on on exercises before coming to lab*
  - Every Friday (7 weeks)
  - Extra lab slot on 2024-03-05 (Tue), 13:00-15:00
  - Presenting older labs has lower priority

Uses of Program Analysis



Categories of Program Analyses

	Manual / Interactive	Automatic
Static Analysis ▸ Examines *structure* ▸ Sees entire program (mostly. . .)	▸ Interactive Theorem Provers	▸ (Most) Type Checkers ▸ Static Checkers (FindBugs, SonarQube, . . .) ▸ Compiler Optimisers
Dynamic Analysis ▸ Examines *behaviour* ▸ Sees interactions program $\leftrightarrow$ world	▸ Debuggers	▸ Unit Tests ▸ Benchmarks ▸ Profilers

Our Focus

Summary

- Program analyses are key components in *Software Tools*:
  - IDEs
  - Compilers
  - Bug and Vulnerability 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

```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: $P(\varphi)$
    Examples:
    - All lines in φ that reference the '`gets`' function
    - Does φ type-check?
    - Where does φ spend most execution time?
  - **Analysis** $\mathcal{A}(\varphi)$ that approximates $P(\varphi)$

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

And How Good Is It?

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

r is...	$r \in \mathcal{A}(\varphi)$	$r \notin \mathcal{A}(\varphi)$
$r \in P(\varphi)$	**True Positive**	**False Negative**
$r \notin P(\varphi)$	**False Positive**	**True Negative**

- How well does $\mathcal{A}$ approximate $\mathcal{P}$?
  - Assume $\mathcal{A}(\varphi)$ returns $n = \#\mathcal{A}(\varphi)$ reports
    $n = \#\text{True Positives} + \#\text{False Positives reports}$
  - Are the reports good?
    Precision $= \frac{\#\text{True Positives}}{n}$
  - Are the reports comprehensive?
    Recall $= \frac{\#\text{True Positives}}{\#\text{True Positives} + \#\text{False Negatives}}$
- #False Negatives (and thus **Recall**) is usually impossible to determine in program analysis

Summary

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

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

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

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

  - $\mathcal{A}(\varphi) = P(\varphi)$ 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."

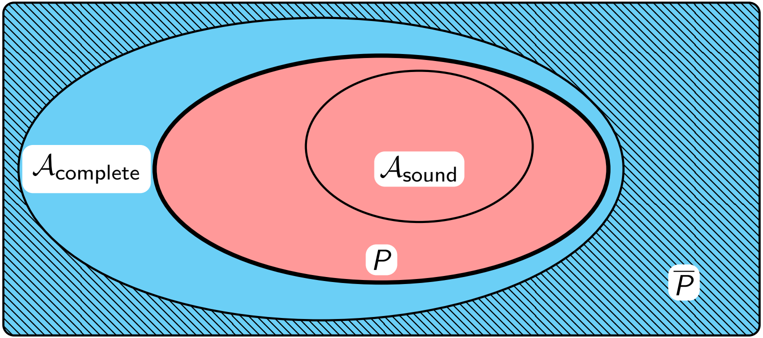
— Anders Møller, paraphrasing H.G. Rice [1953]

We must choose:

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

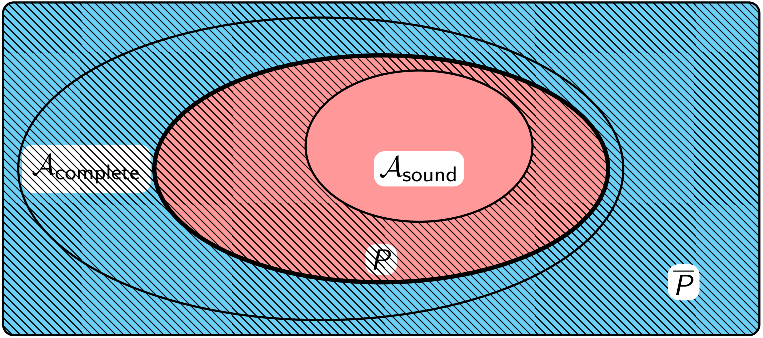
...pick any two.

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* $\overline{P}$

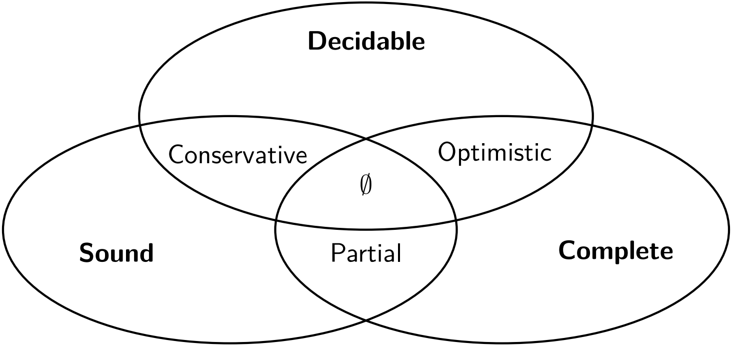
Soundness and Completeness: Caveat



- *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., depends on behaviour):



Lecture Overview

Foundations

- 01 Foundations
- 02 Constructing Program Analyses in JastAdd

Static Analysis

Properties

- 03 04 Types
- 05 06 07 Data Flow
- 08 09 Memory

Control Flow

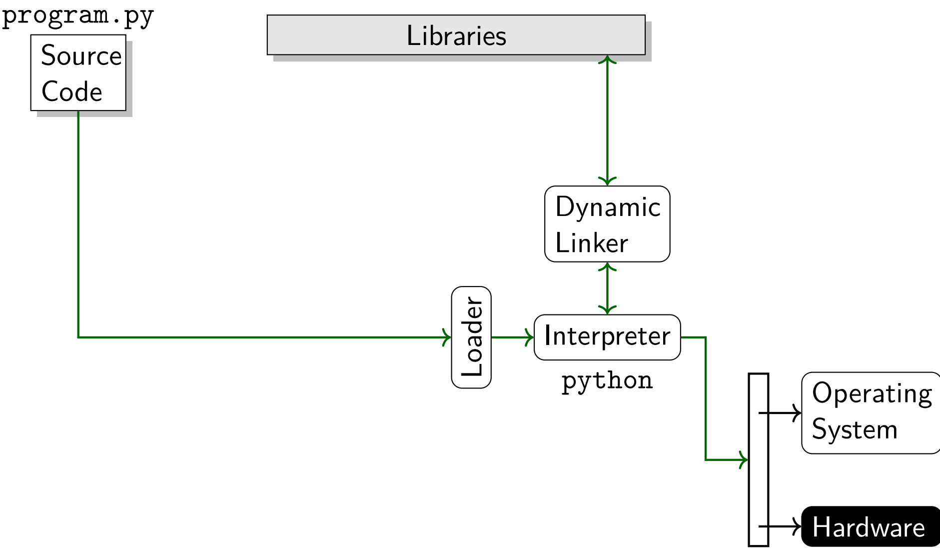
- 05 Intraprocedural
- 10 Interprocedural
- 11 Indirect

Dynamic Analysis

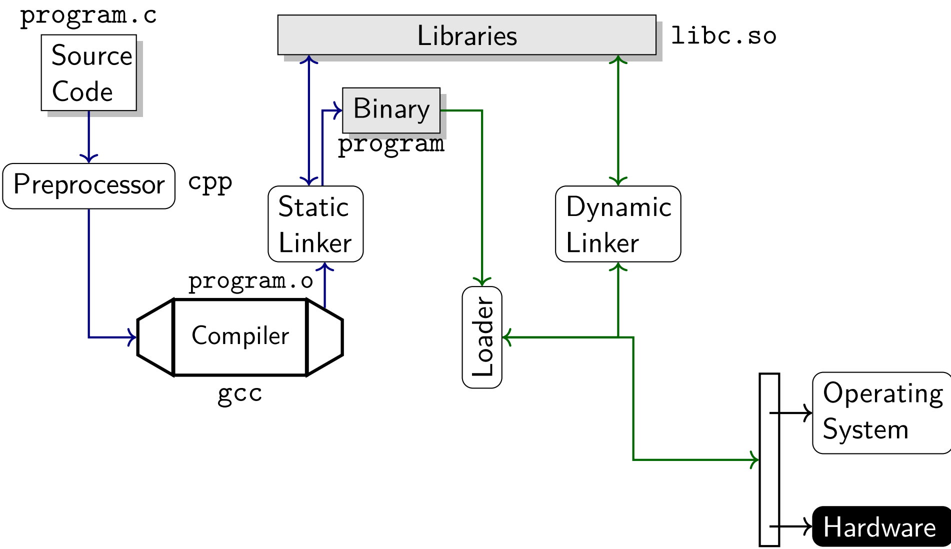
- 12 Instrumentation
- 13 Analysis

- 14 Review

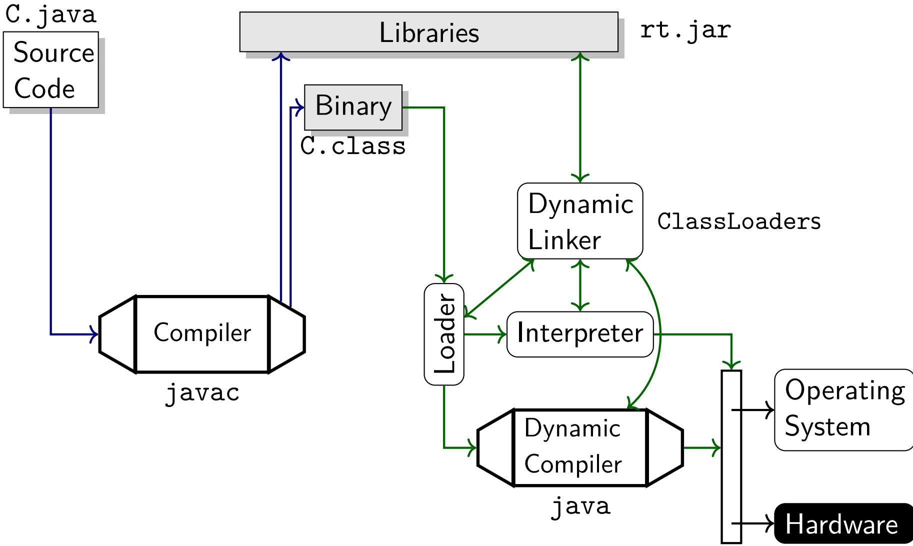
Program Execution Pipeline



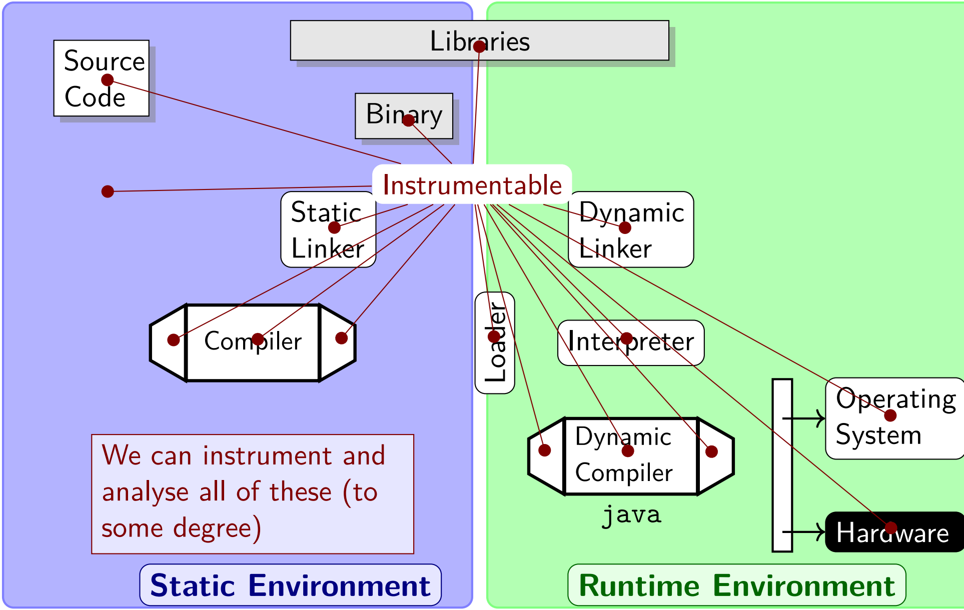
Program Execution Pipeline



Program Execution Pipeline



Program Execution Pipeline



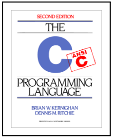
Static vs. Dynamic Program Analyses

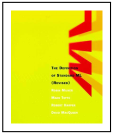
	Static Analysis	**Dynamic Analysis**
Principle	Analyse program structure	Analyse program execution
Input	Independent	Depends on input
Hardware/OS	Independent	Depends on hardware and OS
Perspective	Sees everything	Sees that which actually happens
Completeness (bug-finding)	Possible	Must try all possible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

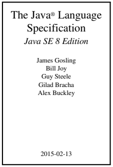
Summary

- **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

Defining Language Behaviour







- Many languages have multiple *language implementations*
- Language behaviour defined in *language specification*:
  - **Static Semantics**:
    Behaviour in static environment
  - **Dynamic Semantics**:
    Behaviour in runtime environment

Static vs. Dynamic Semantics

- Static semantics:
  - Identifier binding (C, Java)
  - Type checking (C, Java)
  - Other well-formedness constraints (C, Java)

Static Environment

- Dynamic semantics:
  - Execution, evaluation, control flow
  - Identifier binding (Python, JavaScript)
  - Type checking (Python, JavaScript, *Java*)
  - Dynamic dispatch (Java, Python, JavaScript)

Runtime Environment

Analysis vs. Semantics

- **Static Program Analysis**:
  - Analysing Static Semantics: *sound & complete* (most languages)
  - Analysing Dynamic Semantics: *sound **or** complete*
- **Dynamic Program Analysis**:
  - Analysing Static Semantics: ?
    - Depends on language; static information may or may not be available dynamically
  - Dynamic Semantics: *Sound*

Static Analysis

Analysing Program Structure

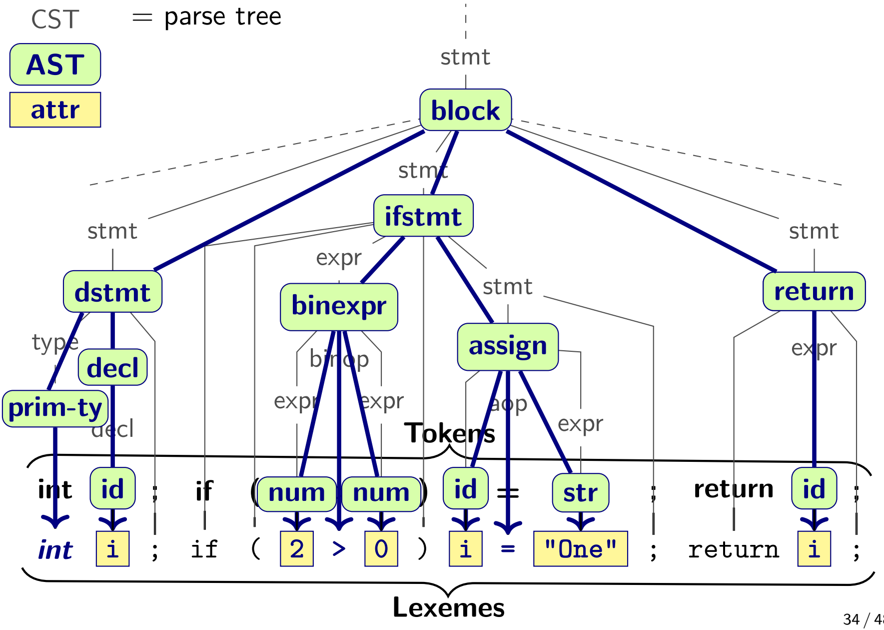
Java lexing

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

Lexing / Tokenisation

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

Java lexing & parsing



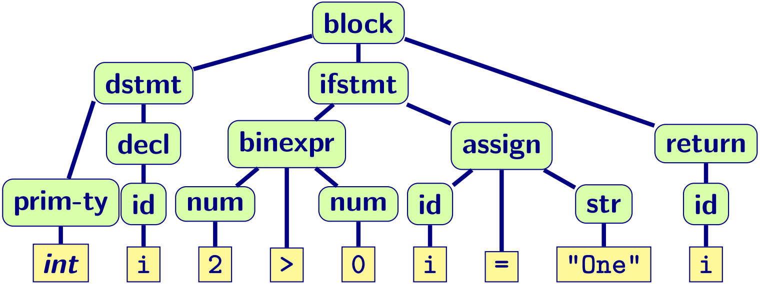
Parsing in general

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 intrinsic 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



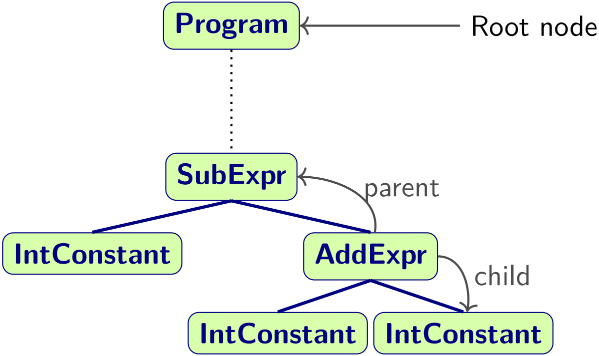
Typical in-memory representations for this AST:

- Algebraic values (functional)
- Records (imperative)
- Objects (object-oriented)

Summary

- Static program analysis operates on an **Intermediate Program Representation** (IR)
  - Our main IR: **Abstract Syntax Trees** (ASTs)
  - Other IRs can speed up / simplify certain tasks (more later)
- ASTs constructed by *Compiler Frontend*:
  - Scanning/lexing/tokenising
  - Parsing
  - Translation from parse tree into AST
  - Not covered in this course; see **EDAN65: Compiler Construction** for details

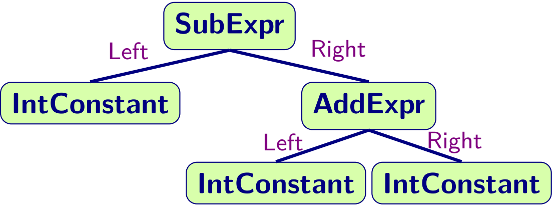
The AST as Data Structure

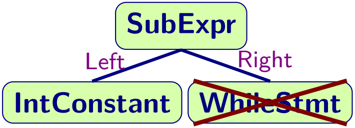


Structure of the AST

Abstract Grammar

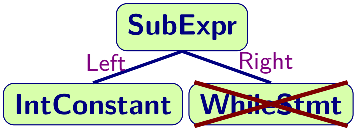
```
Program ::= ...; // start symbol

abstract Expr;
IntConstant :  Expr ::= <Value:int>;

AddExpr :  Expr ::= Left:Expr Right:Expr;
SubExpr :  Expr ::= Left:Expr Right:Expr;

abstract Stmt;
WhileStmt :  Stmt ::= Cond:Expr Body:Stmt;
```





Not allowed: WhileStmt is not an Expr!

Restricting AST Structure

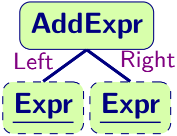


- Intuition:
  - SubExpr wants to subtract values from each other
  - WhileStmt does not compute a value
- Parser and type system guarantee that such nonsensical combinations don't occur
  - Otherwise program analyses would have to check for them

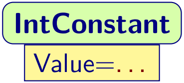
Abstract Grammars

- Grammar specifies all permissible tree constructions
- Consists of *production rules*:
  - *Production* (AddExpr): Name of the language construct
  - *Nonterminal* (Expr): Category ('supertype') for production
  - *Components* (Left:Expr): Child nodes
    - Nonterminal components: child nodes
    - Terminal components: intrinsic attributes

```
AddExpr : Expr ::= Left:Expr Right:Expr;
```




```
IntConstant : Expr ::= <Value:int>;
```



Summary

- Permissible structure of the AST is governed by the *Abstract Grammar*
- The grammar is specified in terms of *Production Rules*
  - Production rules describe the *components* of one Production
  - Each Production belongs to one Nonterminal
  - Standard notation: Backus-Naur Form (BNF)
  - Exact BNF syntax varies between tools; we will use JastAdd's variant
- Structure is enforced by parser and type system

⇒ Simplifies analysis construction

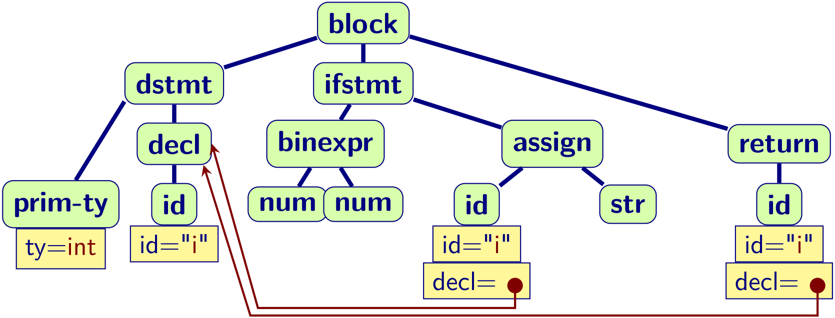
- Common nonterminals:
  - Expr: computes a value
  - Stmt: triggers a side effect or controls the order of side effects
  - Decl: declares or defines a variable/function/...

Some Basic Analyses

- *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?

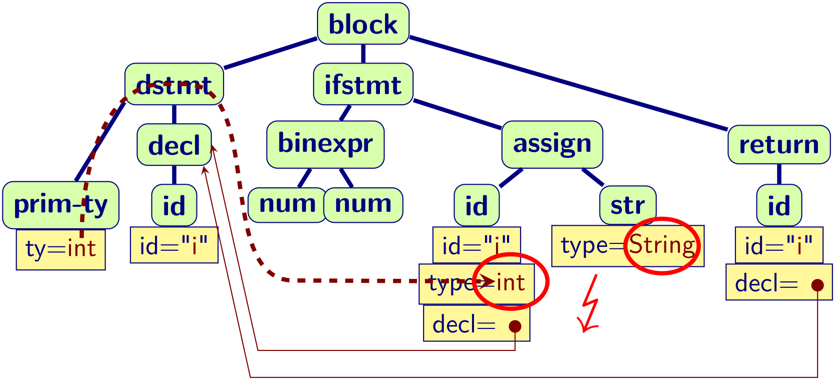
  ...

Example: Name Analysis



- For each **id**, compute the corresponding **decl**
- In AST-based IR: keep reference to
- Check that we found a **decl** node (otherwise **Error**)

Example: Type Analysis



- Check that all types are compatible with their operators
- Must first compute types
- **assign** node: type error!
  Trying to assign String to int variable

Summary

- Program analysis on AST:
  - Enrich AST nodes with additional information
  - Name Analysis: references to declarations
  - Type Analysis: types (computed, propagated)
  - Analyses often need to use results of earlier analyses
- Lecture 2 will introduce systematic strategies for computing such information

Moving Forward

- **How do we *build* static program analyses?**
  - Avoid building from scratch: many frameworks available
  - Re-use where you can
  - **Here**: JastAdd: Next lecture (Flipped!)
- **How do we *design* program analyses?**
  - Theoretical frameworks:
    - **Type Inference**
    - **Dataflow analysis**
    - Abstract interpretation

      ...
  - Language Definition:
    - **Static Semantics**:
      Compile-time/load-time behaviour
    - **Dynamic Semantics**:
      Run-time behaviour

Outlook

- **Remember**:
  - Join Moodle *today*
  - Form groups by Wednesday, 18:00
- Continuing on static program analysis:
  - Type Analysis
  - Data Flow Analysis
  - Heap Analysis
- **Next Lecture**: Wednesday, same time & place:
  - Topic: Building Program Analyses with *Reference Attribute Grammars* in *JastAdd*
  - **Flipped Classroom lecture**
    - Watch videos beforehand
    - Bring questions
    - We will discuss material from the videos *based on your questions*

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