Lund University

EDA045F: Program Analysis

LECTURE 1: INTRODUCTION

Christoph Reichenbach

Welcome!

- **Program Analysis**
- **Instructor**: Christoph Reichenbach
  `christoph.reichenbach@cs.lth.se`
- **Course Homepage**:

`http://fileadmin.cs.lth.se/cs/Education/EDA045F/2018/web/index.html`

- **Moodle**: EDA045F

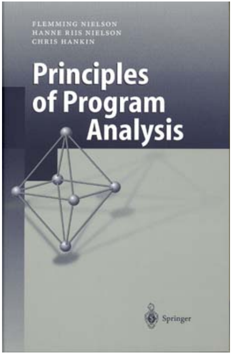
Topics

- Concepts and techniques for understanding programs
  - Analysing program structure
  - Analysing program behaviour
- Lanugage focus: **Java**, **C**, **C++**
  - Concepts transferrable to most other languages

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
  - Understand literature on typical program analyses

Book



Principles of Program Analysis

Nielson, Nielson & Hankin

- 3 copies in the library
- Optional (goes deeper into the theory)

Structure

2018-09-14	today	
2018-09-21		Homework #1 start
2018-09-28		
2018-10-05		Homework #2 start
2018-10-12		
2018-10-19	End of lp1	Homework #3 start

— break —

2018-11-07	lp2: *Wednesdays* 15:15	
2018-11-14		Homework #4 start
2018-11-21		
2018-11-28		Homework #5 start
2018-12-05		
2018-12-12		Homework #6 start
2018-12-19	End of lp2	

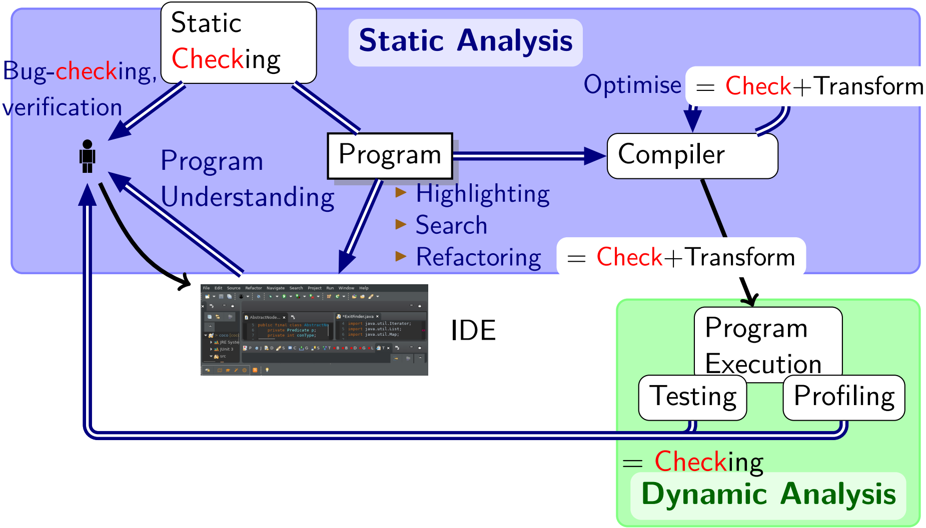
How to Pass

- **Homework projects**:
  - **Who**: Groups of 2 (or individuals)
  - **What**: Work with program analysis frameworks
  - **Start**: Homework will be up immediately after class
  - **Deadline**: 13 days for each project (Thu/Tue evening)
    Exception: Homework 3 deadline is 2018-11-13
    (giving you almost a month)

$\Rightarrow$ 0–5 points per project

  - *I will not answer homework-related questions on the day of the deadline*
- **Final Exam** after lp2
  - **Admission**: 2 or more points in 5 or more homework projects

Uses of Program Analysis



Many uses are for checking boolean properties

Checks in Program Analysis

Given a program, check that some property P holds

Typical properties:

- No type errors
- Some particular refactoring / optimisation won't change program behaviour
- **Program Verification**: The program meets all requirements

Verification vs. Validation

According to Barry Boehm:

- **Verification**: 'Am I building the product right?'
- **Validation**: 'Am I building the right product?'

Example: Software Security:

- Given a *Threat Model* (set of possible attacks):

Program is *secured* vs. known attacks	**Verification**
Threat model is *complete*	**Validation**

We focus on Verification

Verification: Safety vs. Security

- **Security**:
  External attackers cannot compromise the system
- **Safety**:
  The system does not 'go wrong'
  - Example: *Strong Typing* in Java / Haskell / ...
  - Languages without strong typing (C, C++) often restricted
    Example: MISRA-C, restricted C89 for automotive systems:

2.2: Source code shall only use `/* ... */` style comments.

5.1: Identifiers [...] shall not [use] more than 31 characters.

9.1: All automatic variables shall have been assigned a value before being used.

12.4: The right-hand operator of a logical `&&` or `||` operator shall not contain side effects.

14.1: There shall be no unreachable code.

16.2: Functions shall not call themselves, either directly or indirectly.

20.10: The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>` shall not be used.

Verification vs. Program Analysis

Manual Verification

Proof Assistants

- CoQ
- Isabelle
- Agda

...

Automatic Verification

Program Analysis-based Verifiers

- Lint
- C/Haskell/... type checking
- Astrée

...

Our focus is on (mostly) automated approaches

Summary

- Program analyses:
  - Link information
  - Collect information for visualisation
  - Decide **yes**/**no** checks
- Same frameworks / approaches, different outputs
- Checks test some program property P
- Typical checks:
  - Program doesn't 'go wrong' (safe)
  - Program cannot be compromised (secure)
- Program analyses are *fully automatic*

Everyday 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));
} }
```

A First Challenge

- Given program.c:
- Property: 'The library functions . . . , gets, . . . shall not be used.'

```
user@host$ grep gets program.c
     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: Analysis is *imprecise*

A First Challenge, Continued

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

- More precise!
- 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);
}
```

```
user@host$ cc -c program.c; nm program.o
0000000000000000 T f
                 U gets          ⟵ Aha!
                 U _GLOBAL_OFFSET_TABLE_
user@host$
```

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, "puts");
    void(*p)(char*) = sym;
    p(target_buffer);
    return 0;
}
```

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

Analysis doesn't catch everything (unsound)

Soundness and Completeness

Analysis (Check) A tests a property P:

- A is **sound** (with respect to P) iff:
  - When A triggers, P is true
- A is **complete** means:
  - When P is true, A triggers
- Interpretation differs between *verification* and *bug-finding*:

	P	**Soundness**	**Completeness**
Verification	there is no bug	A finds all bugs	A finds only bugs
Bug-finding	there is a bug	A finds only bugs	A finds all bugs

- Other common terms:
  - **Precision**: Fraction of reported bugs that are real bugs
  - **Recall**: Fraction of real bugs that were reported

We want: Analyses that are sound and complete

The Unfortunate (?) Bottom Line

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

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

We must choose:

- **Sound**
- **Complete**
- **Terminating**

. . . pick any two.

Gaming the System?

- Idea:
  - select analysis A_s: **Sound + Terminating**
  - select analysis A_c: **Complete + Terminating**
  - Report intersection of A_s and A_c $\Rightarrow$ perfect solution?

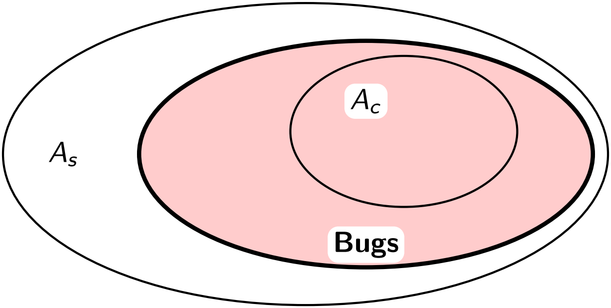
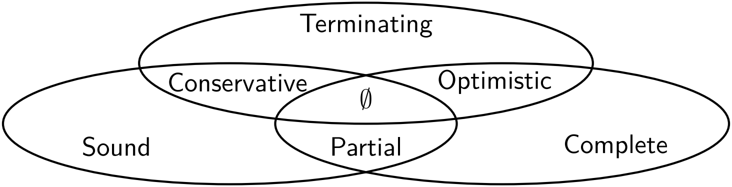


Figure for **Verification** (swap A_s and A_c for bug finding)

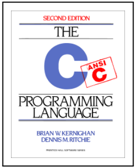
Summary

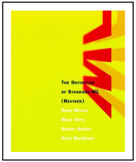
- **Verification** checks absence of bugs
- **Bug-finding** checks presence of bugs
- Given property P and analysis A:
  - A is **sound** if it triggers only on P (maybe misses some)
    Verifier: A finds all bugs
  - A is **precise** if it always triggeres on P (and possibly on non-P)
    Verifier: A finds only bugs
- **Bug-finders** swap the meanings of **sound** and **complete**
- If P is nontrivial (depend on behaviour, not program structure), the following holds for **Verifiers**:

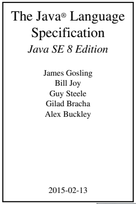


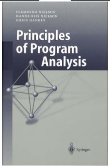
Gathering Our Tools

Theory









Tools

Compilers

Analysis Frameworks

jastadd

PhASAR

Astrée

FindBugs™

Soot

Hardware

Language Definitions

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

Syntax example:

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

Let's develop a check: will the given program compute a positive number?

Simplifying the Lanugage

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

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

Restricted scope, but simplified our job

Semantics

- What does a program p compute?
- Notation: $\boxed{p \Downarrow i}$, where i is some number in $\mathbb{Z}$

$$\text{zero} \Downarrow 0 \qquad\qquad \text{one} \Downarrow 1$$

$$\frac{x \Downarrow i}{\text{neg } x \Downarrow -i} \qquad\qquad \frac{x \Downarrow i \quad y \Downarrow j}{x \mathbin{\mathbf{+}} y \Downarrow i + j}$$

Computing Is-The-Result-Positive

- Our semantics are unambiguous

✓ Can compute value of any program

- In other languages, computing the output:
  - depends on input
  - may not terminate
- As example, let's classify programs in our toy language into:
  - A^0: Computes 0
  - A^+: Computes a positive value
  - A^-: Computes a negative value
- Notation: $\boxed{p \Downarrow^A a}$, where a is one of A^0, A^+, A^-

Semantics

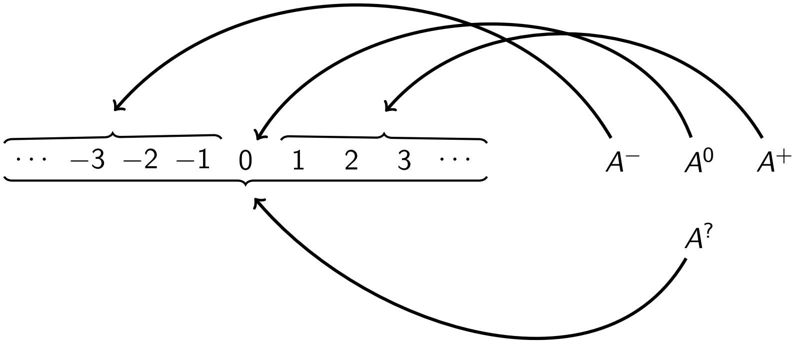
$$\begin{aligned} \ominus A^0 &= A^0 \\ \ominus A^+ &= A^- \\ \ominus A^- &= A^+ \\ \ominus A^? &= A^? \end{aligned}$$

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

$$\text{zero} \Downarrow^A A^0 \qquad \text{one} \Downarrow^A A^+$$

$$\frac{x \Downarrow^A a}{\text{neg } x \Downarrow^A \ominus a} \qquad \frac{x \Downarrow^A a_1 \quad y \Downarrow^A a_2}{x + y \Downarrow^A a_1 \oplus a_2}$$

Correspondence: Abstract and Concrete



Also:

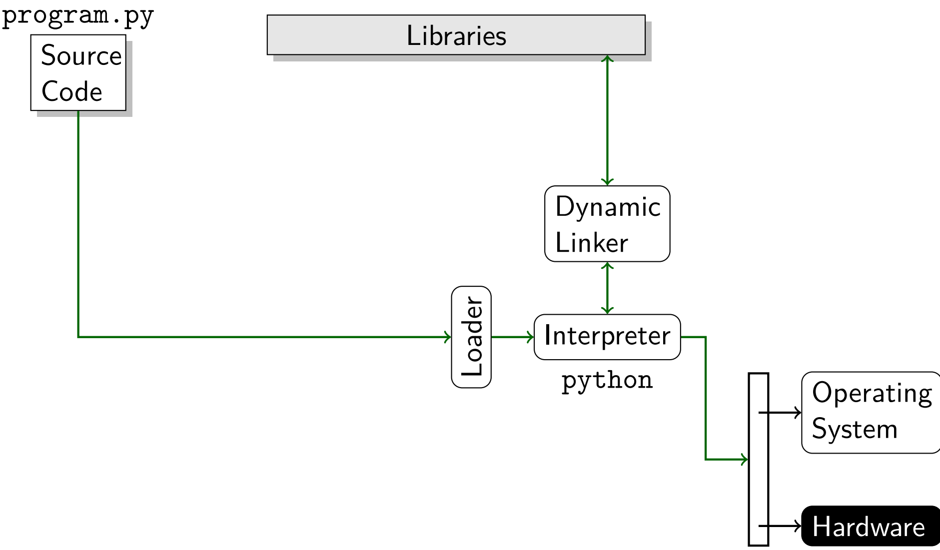
- $\ominus$ *is compatible with* neg
- $\oplus$ *is compatible with* $+$

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

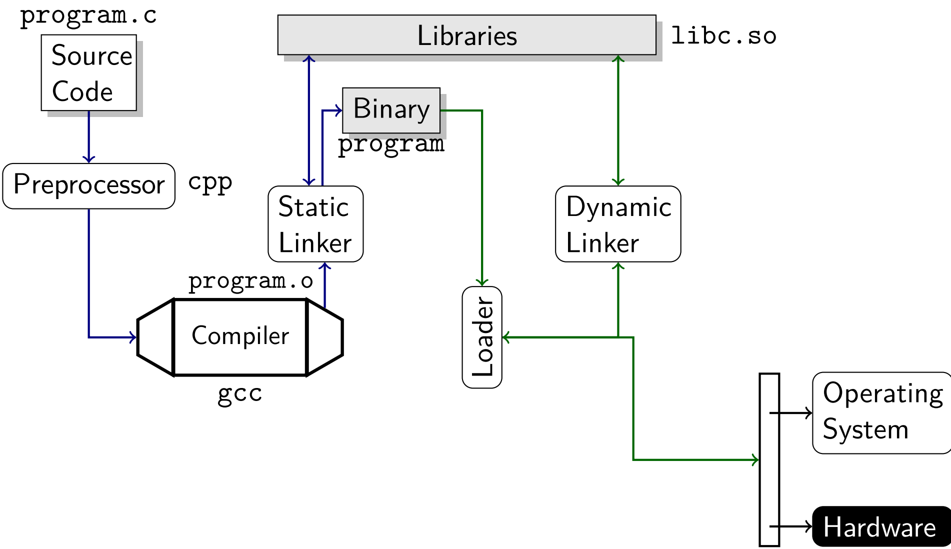
Summary

- We can mathematically formalise *syntax* and *semantics*
- Semantics derive from syntax, with suitable notation
- **Abstract Interpretation**:
  - Abstract over the program's semantics
  - Goal: check if some property P holds
  - Challenge: remain precise yet decidable
  - May have abstractions A_1, A_2 where A_1 is strictly more precise than A_2

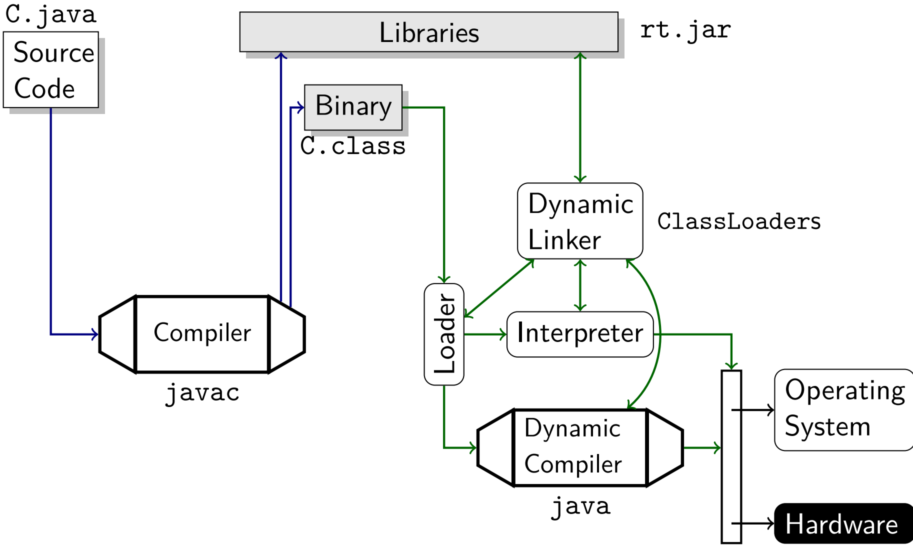
Program Execution Pipeline



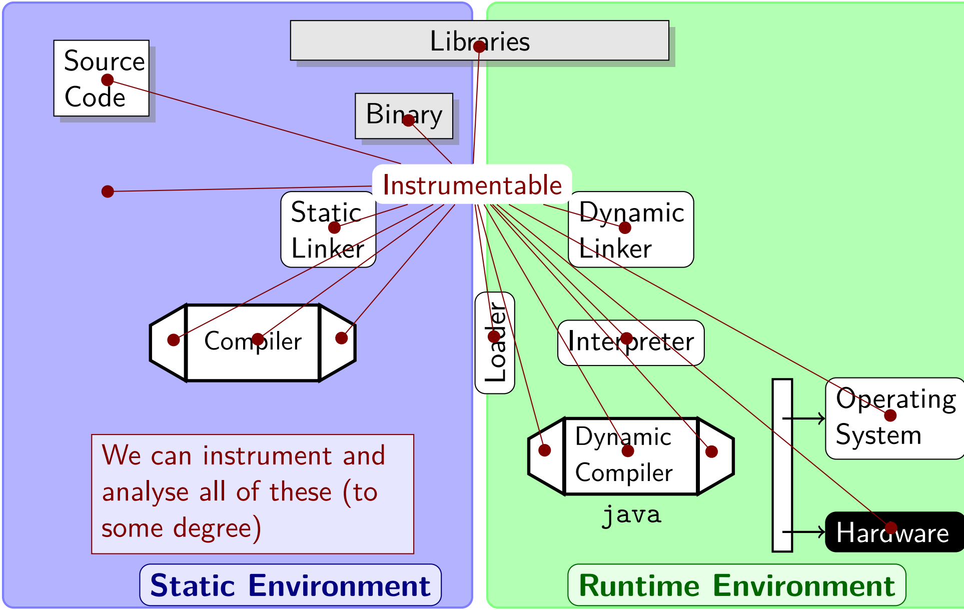
Program Execution Pipeline



Program Execution Pipeline



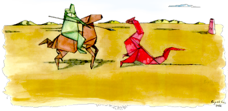
Program Execution Pipeline



We can instrument and analyse all of these (to some degree)

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
Soundness	Possible	Must try all possible inputs
Precision	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

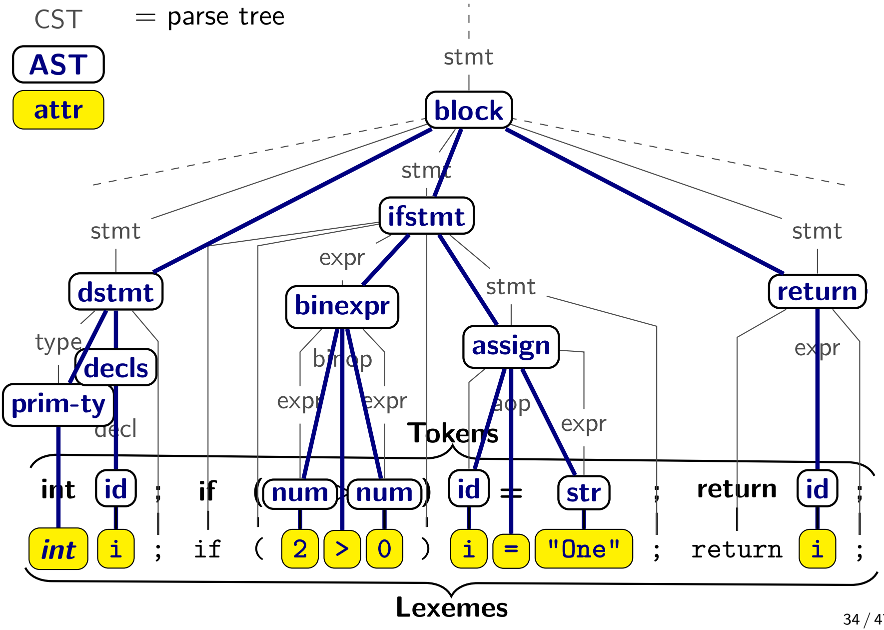
Java lexing

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

⇓ Lexing / Tokenisation ⇓

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

Java lexing & parsing



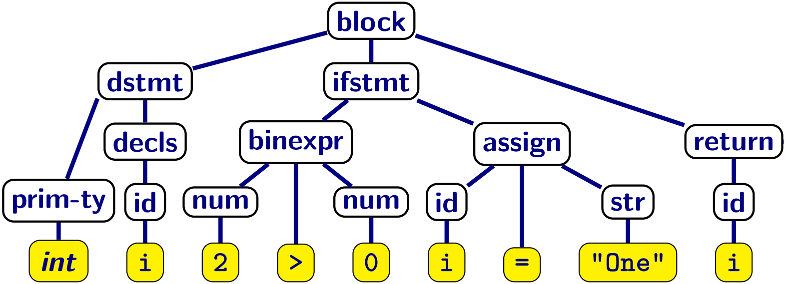
Parsing in general

*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

Program analysis starts on the AST

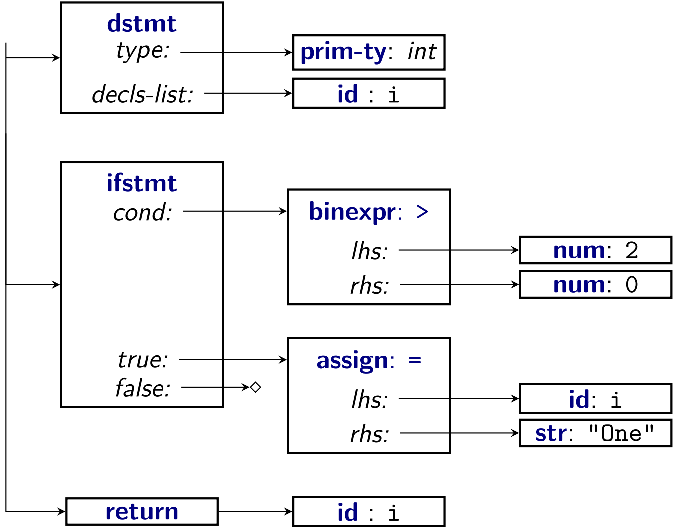
In-Memory Representation



Typical in-memory representations for this AST:

- Algebraic values (functional)
- Records (imperative)

In-Memory Representation



Program Analysis

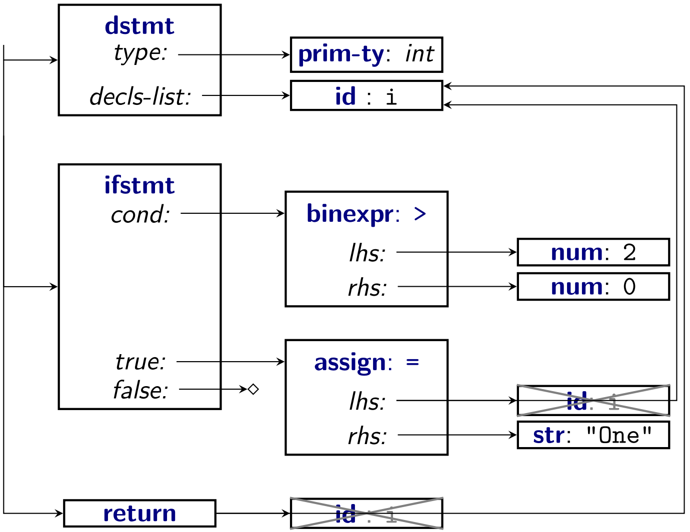
We run numerous code analyses on the AST:

- *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?

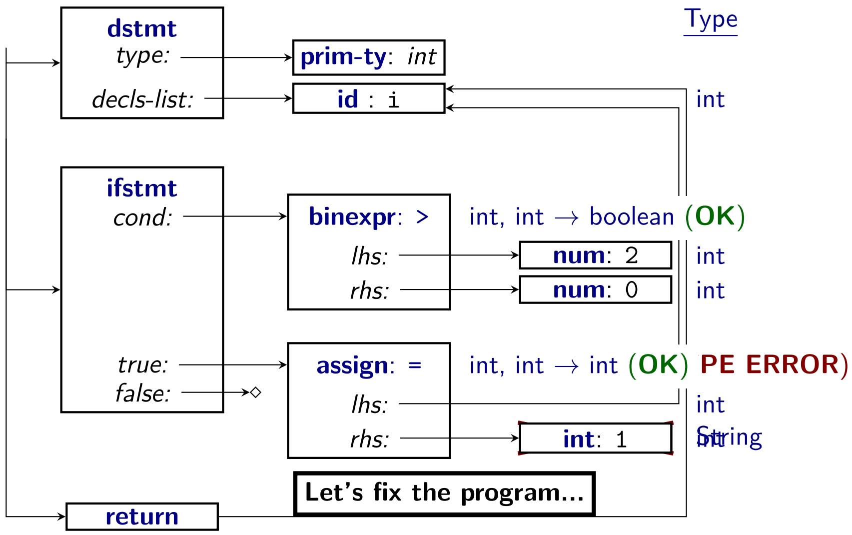
  ...
- *Optimisations*:
  - Can we speed up the program somehow?

Advanced static correctness checks increasingly common in compilers

Name Analysis



Type Analysis



Summary

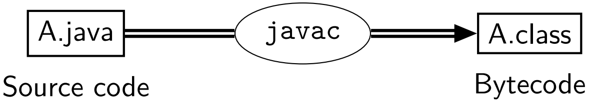
- Compiler represents programs in *intermediate representations* (IRs)
- Compiler can be separated into:
  - *Frontend*: process incoming source code, generate IR
  - *Middle-end*: optimise IR
  - *Back-end*: translate IR into executable code
- Parser matches *concrete syntax tree* (CST), generates *abstract 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?

  ...

Emitting code

The compiler backend emits bytecode from the AST structure:

- Involves additional steps




```
public static int myfun(int k) {
  int i;
  if (k > 0) {
    i = 1;
  } else {
    i = 0;
  }
  return i;
}
```

```
...
0:    iload_0
1:    ifle 9
4:    iconst_1
5:    istore_1
6:    goto 11
9:    iconst_0
10:   istore_1
11:   iload_1
12:   ireturn
...
```

Java Bytecode: Example

Let's call our function with `myfun(7)`:

...			
⇒	`0:`	`iload_0`	Load first function parameter as `int`
⇒	`1:`	`ifle 9`	7 <= 0? No, so continue
⇒	`4:`	`iconst_1`	load the value 1
⇒	`5:`	`istore_1`	i := 1
⇒	`6:`	`goto 11`	jump to label 11
	`9:`	`iconst_0`	
	`10:`	`istore_1`	
⇒	`11:`	`iload_1`	Load *i* (value 1)
⇒	`12:`	`ireturn`	Return 1.
...			

And the method returns.

Java Bytecode Overview

- 202 instructions
- Operate on *Value Stack* and *Local Variables*
- Complex heap and thread model
- Statically typed
- Variations due to compression:
  - `iload` *i*: Load local variable *i* as `int`
  - `iload_0`: Same as `iload 0`
  - `iload_1`: Same as `iload 1`
  - `iload_2`: Same as `iload 2`
  - `iload_3`: Same as `iload 3`
- Variations due to typing:
  `iload` (`int`), `lload` (`long`), `dload` (`double`), `aload (objects)`
- Many instructions have 'wide' variants

Optimised for space, checkability; not too convenient to work on

Which Abstraction Is Right for You?

- Different tools use different intermediate representations:
  - JastAdd
  - Soot: *Jimple*, *Shimple*, *Grimple*, *Baf*
  - WALA
  - Eclipse JDT and CDT
  - gcc: *Gimple*
  - LLVM IR / LLVM bitcode
- Some are *graph-based* ($\rightarrow$ next week!)
- Different strengths and weaknesses:
  - Eclipse JDT/CDT support source-to-source transformation
  - Soot's '*Grimple*': Easier to read but harder to analyse than '*Jimple*'

Summary

- Compilers and analysis tools use *Intermediate representations* (IRs)
- IRs simplify analysing code
- Different IRs have different advantages
- Program analysis tools introduce abstractions to simplify analysis

To be continued...

Next week:

- Graph-based representations
- Foundations of Dataflow Analysis
- Homework #1