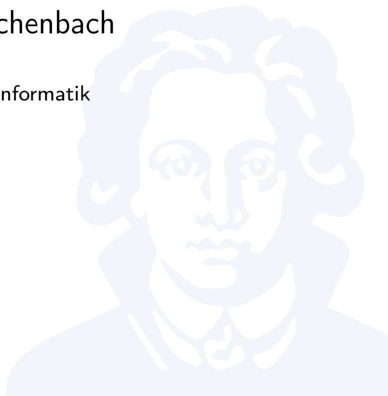
Software Engineering Tools 00

Prof. Dr. Christoph Reichenbach

Fachbereich 12 / Institut für Informatik

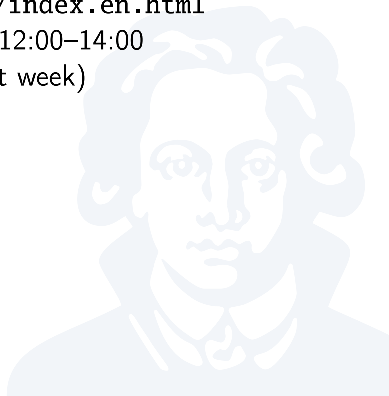
11. April 2014



Welcome to Software Engineering Tools!

Contact	`reichenb@cs.uni-frankfurt.de`
Class web page	`http://sepl.cs.uni-frankfurt.de/2014-ss/m-swe-pr/index.en.html`
Office hours	210a, RMS 10, Thu 12:00–14:00 (NOT this week/next week)

- Structure:
  - 3 lectures + today
  - project with individual meetings
  - final project presentation

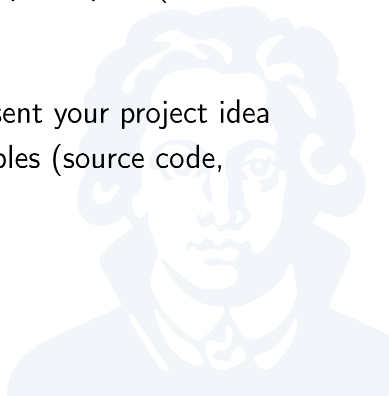


Expectations

- *Praktikum*: You have to implement/extend:
  - Software tool, or
  - Language extension
- Specifically:
  - Pick a project
  - Define project goals
  - Meet with me at least every 2-3 weeks to discuss project status
  - Give a 1h talk that demonstrates
    - Why the project matters
    - What it does
    - How you implemented it
    - What observations you made
  - Submit deliverables (source code, measurements, ...)

Important Dates

- TODAY: e-mail me if you want to participate (include Matrikelnummer)
- May 7th: Agree on a project idea
- July 23th/24th, 14:00–18:00: Present your project idea
- July 31st: Submit project deliverables (source code, write-up, as per project goals)



Overview

- Today:
  - Overview: software tools
  - Example: Java compiler & run-time
  - Preview
  - Project ideas
- Lecture 01: **Syntax & Semantics & Types**
- Lecture 02: **Static Program Analysis**
- Lecture 03: **Dynamic Program Analysis**

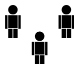
Literature

- "Compilers: Principles, Techniques, and Tools", by Aho, Lam, Sethi, & Ullman ("The Dragon Book")
  How to build compilers
- "Principles of Program Analysis", by Nielson, Nielson, & Hankin
  Static analysis from a formal perspective

How we develop software

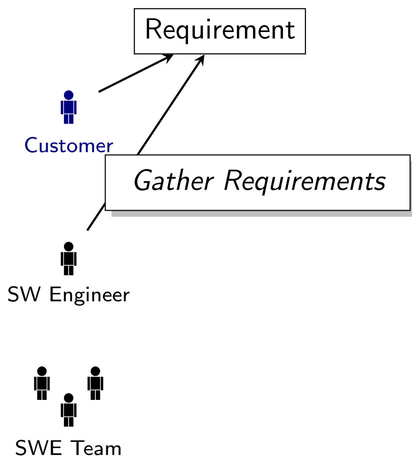
Customer

SW Engineer

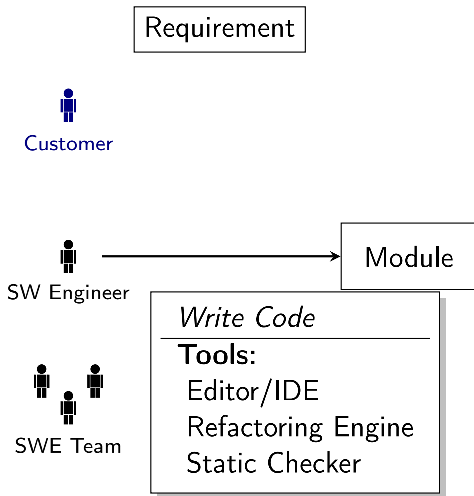


SWE Team

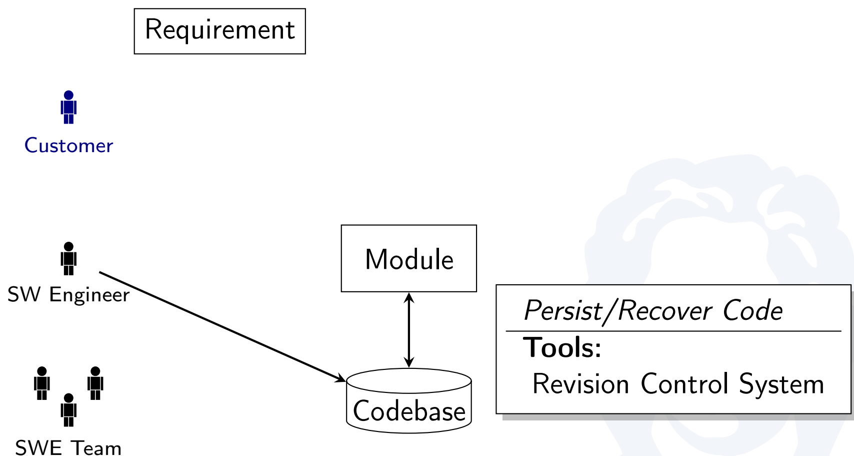
How we develop software



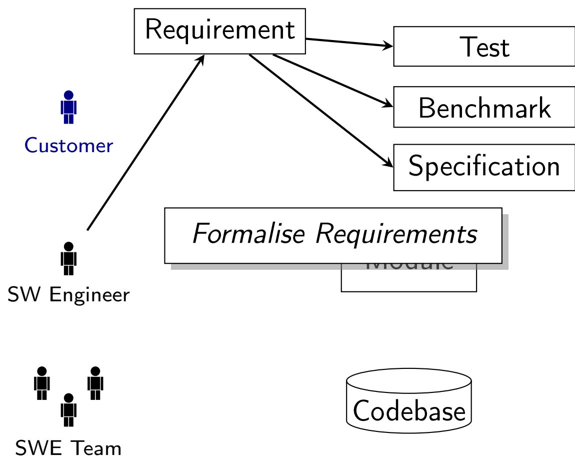
How we develop software



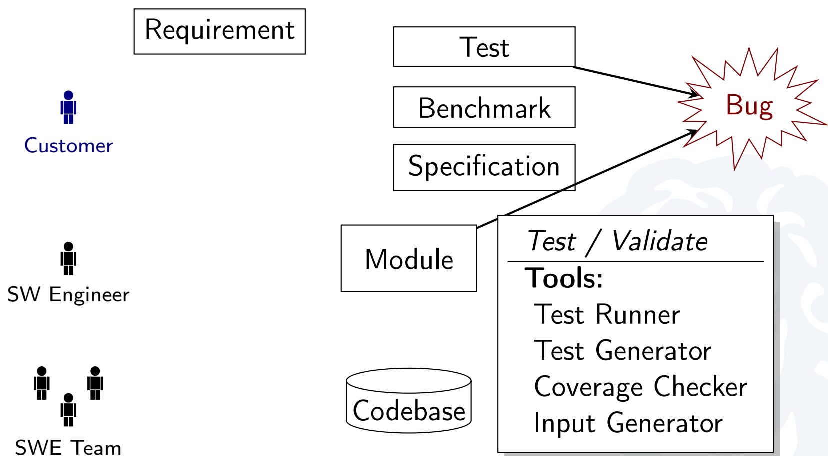
How we develop software



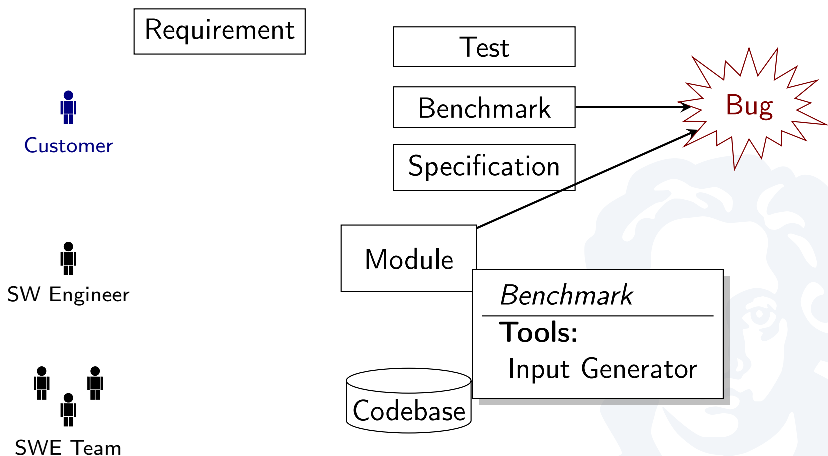
How we develop software



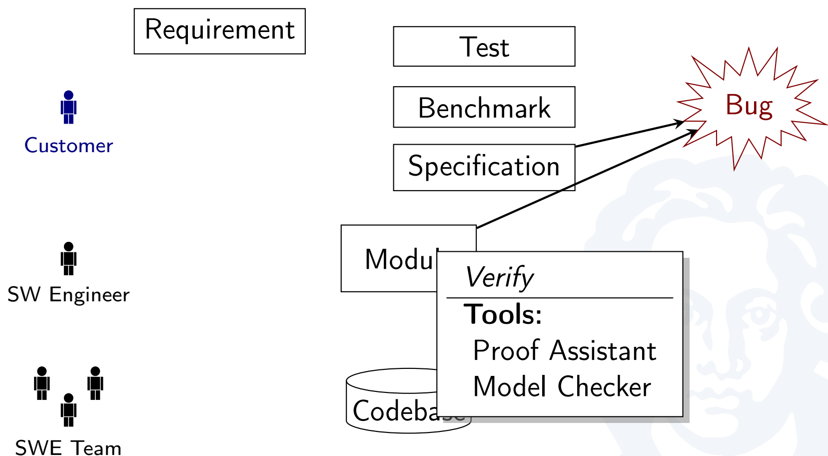
How we develop software



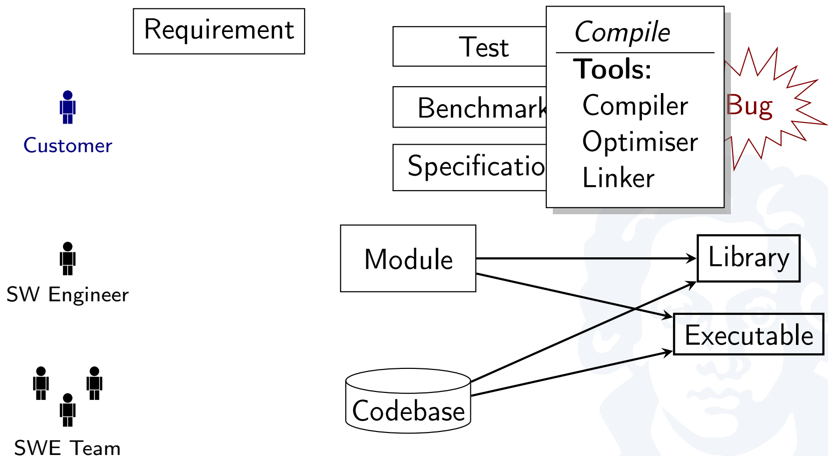
How we develop software



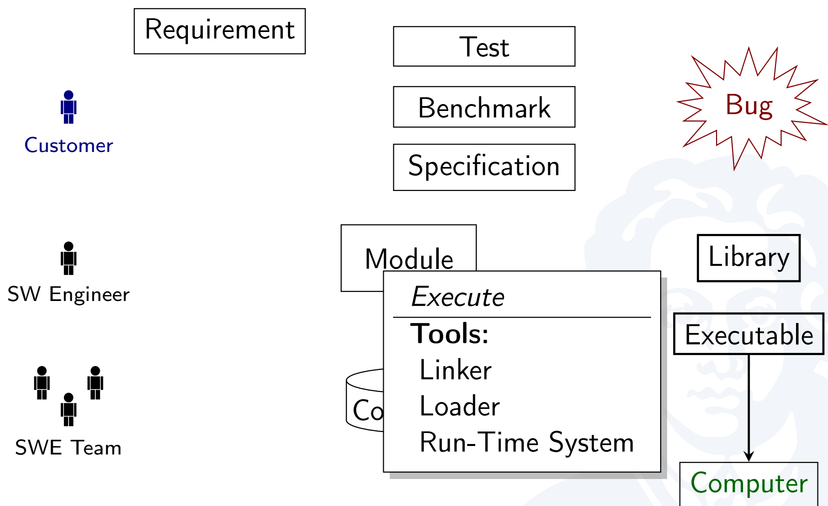
How we develop software



How we develop software



How we develop software



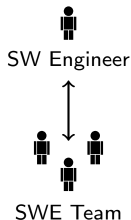
Software Tools: Examples

- Editor / IDE
- Refactoring Engine
- Static Code Checker
- Revision Control System
- Test Runner
- Test Generator
- Test Coverage Checker
- Input Generator
- Proof Assistant
- Model Checker
- Compiler
- Optimizer
- Linker
- Loader
- Run-Time System
- ...

Software Engineering Tools

- Include all Software Tools
- Also, tools for handling processes:
  - Issue Tracking
  - Scheduling / Planning
  - Document Management

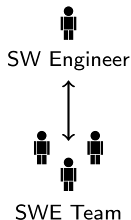
  ...



Software Engineering Tools

- Include all Software Tools
- Also, tools for handling processes:
  - Issue Tracking
  - Scheduling / Planning
  - Document Management

  ...



We focus on *Software Tools*

The purposes of Software Tools

- Human vs. artefact
  - Creation
  - Manipulation
  - Measurement
  - Visualisation
- Artefact vs. artefact
  - Translation
  - Generation
  - Validation
- Artefact vs. hardware
  - Execution

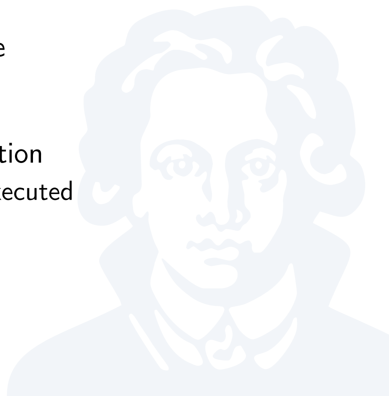
Incomplete list!

Usage Modes

- Activation
  - Automatic
  - Interactive

Usage Modes

- Activation
  - Automatic
  - Interactive
- Activity cycle
  - Static: at or before compile time
    - Can see whole program
    - Can see all possibilities
  - Dynamic: during program execution
    - Can see what actually gets executed
    - Can observe execution time

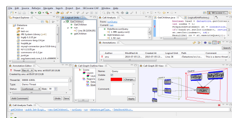


Usage Modes

- Activation
  - Automatic
  - Interactive
- Activity cycle
  - Static: at or before compile time
    - Can see whole program
    - Can see all possibilities
  - Dynamic: during program execution
    - Can see what actually gets executed
    - Can observe execution time

Modern approaches increasingly hybrid

Components of Software Tools



```
for (int i = 0; i < 10; i++) {
  z += i;
  if (z > 42) {
    System.out.println(z);
    z /= 10;
} }
```

```
E8 03 CD 10 c5 01 45 39 e5
38 83 eb 04 4c 39 ff 0f 84
01 45 84 f6 48 89 44 24 38
af 84 d2 79 ab 89 d0 83 e0
74 24 38 31 d2 4c 89 44 24
48 89 7c 24 38 e8 66 92 01
```

User Interface

Source Code

Binary Code

tokenise

parse

unparse

load

emit

run

Program Representation

Transformation

Analysis

Traces

Java lexing

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

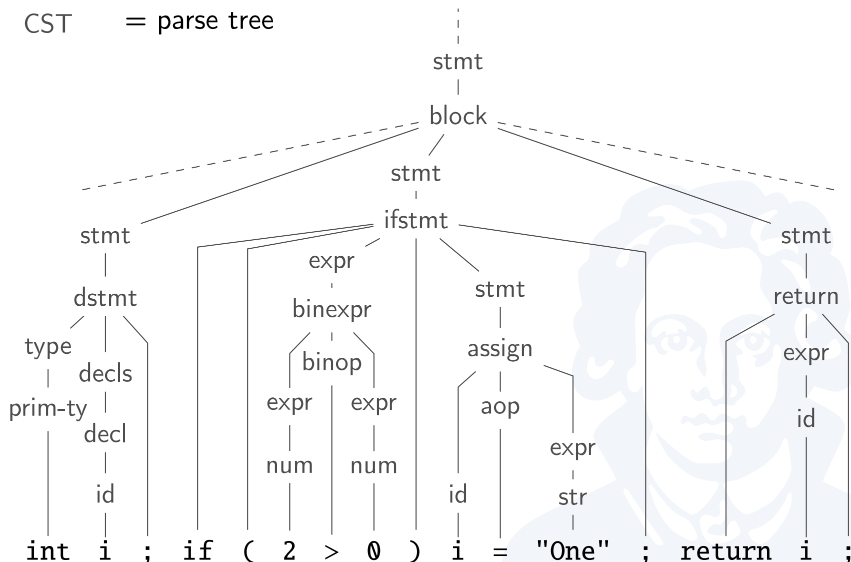
Java lexing

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

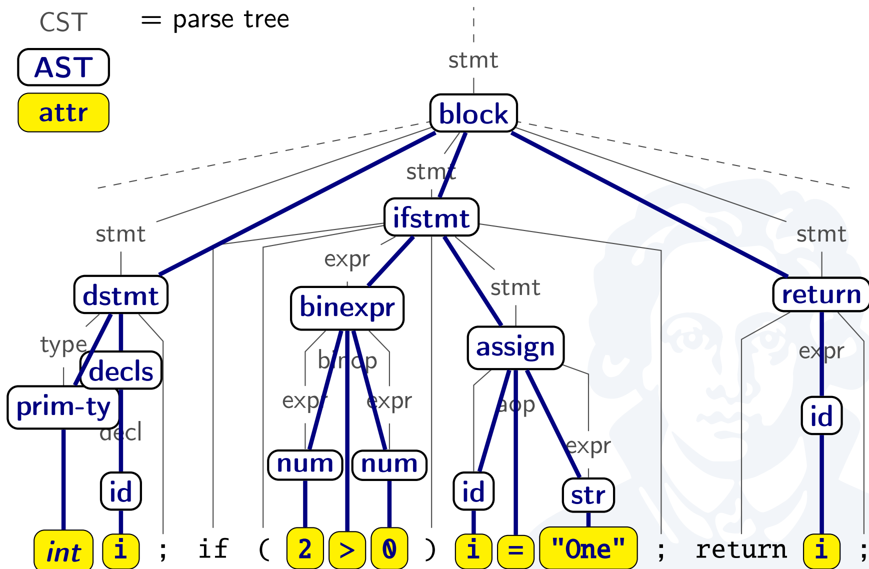
Lexing / Tokenisation

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

Java parsing



Java parsing



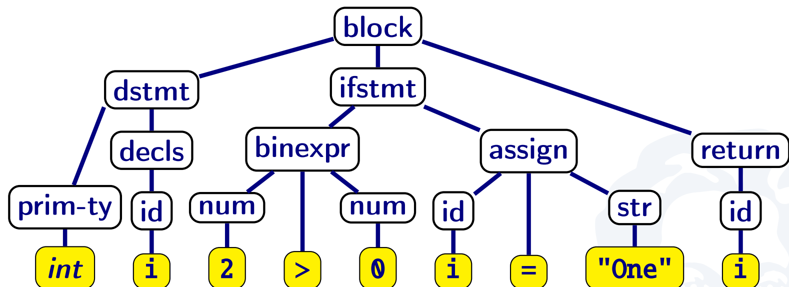
Parsing in general

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

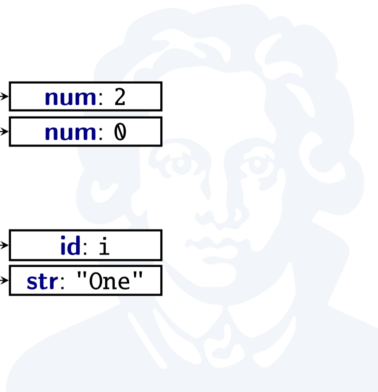
dstmt
type: → **prim-ty**: *int*
decls-list: → **id** : i

ifstmt
cond: → **binexpr**: >
- *lhs:* → **num**: 2
- *rhs:* → **num**: 0

true: → **assign**: =
- *lhs:* → **id**: i
- *rhs:* → **str**: "One"

false: → ◇

return → **id** : i

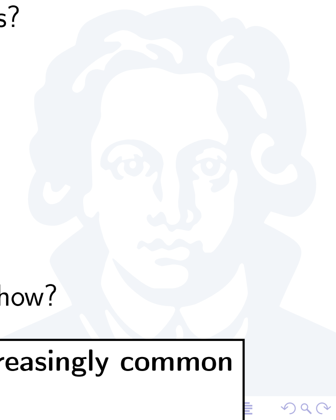


Program Analysis

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

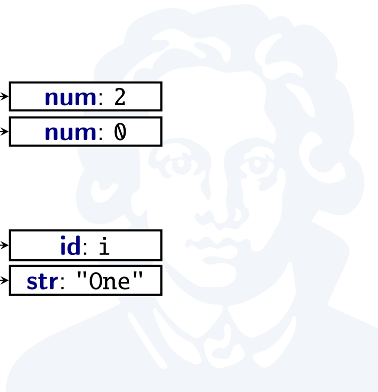
dstmt
type: → **prim-ty**: *int*
decls-list: → **id** : i

ifstmt
cond: → **binexpr**: >
- *lhs:* → **num**: 2
- *rhs:* → **num**: 0

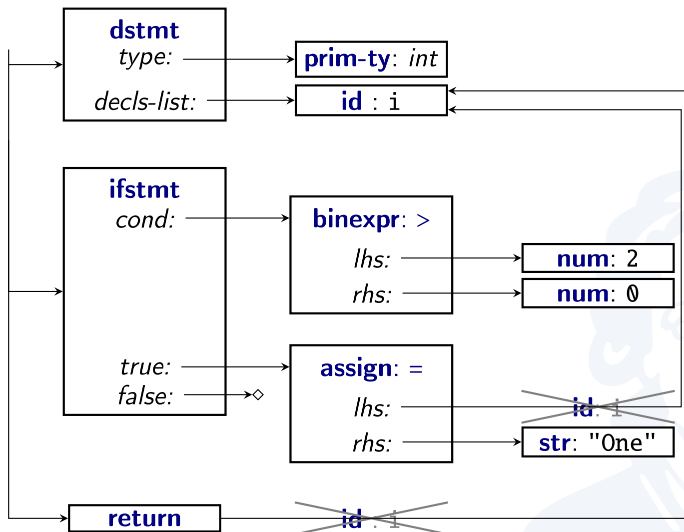
true: → **assign**: =
- *lhs:* → **id**: i
- *rhs:* → **str**: "One"

false: → ◇

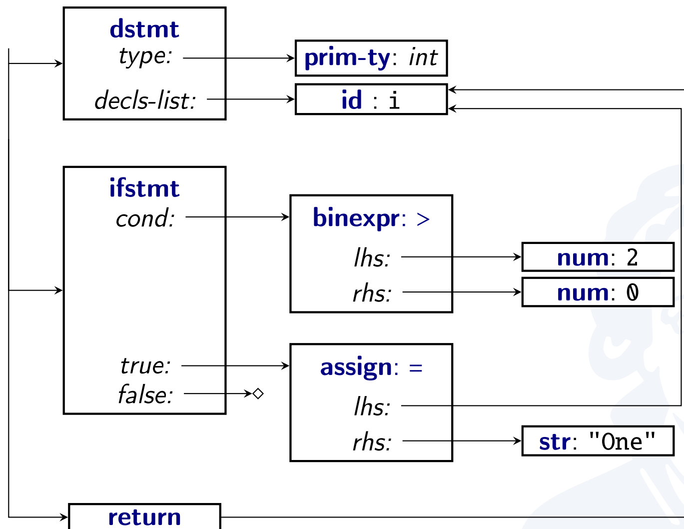
return → **id** : i



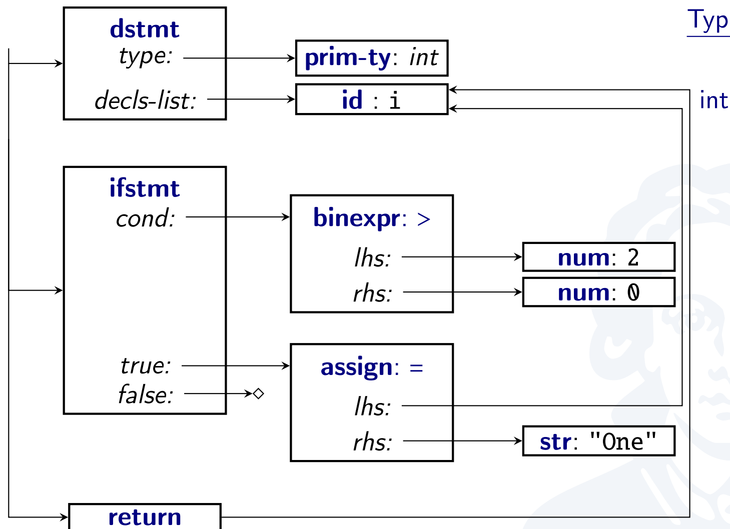
Name Analysis



Type Analysis



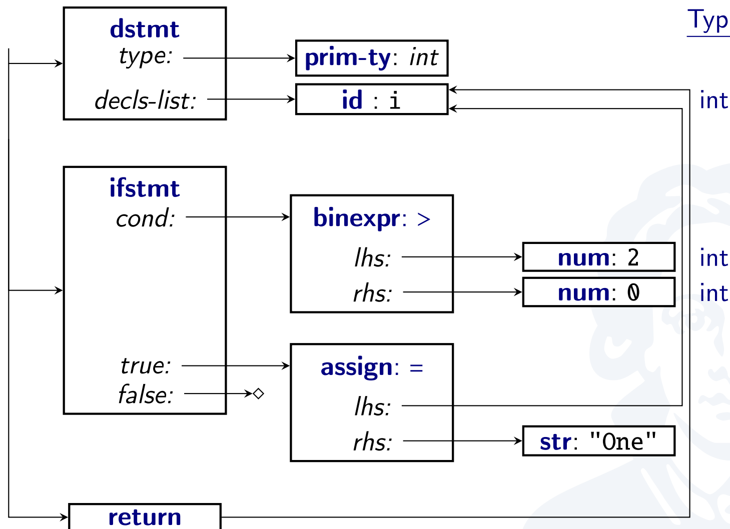
Type Analysis



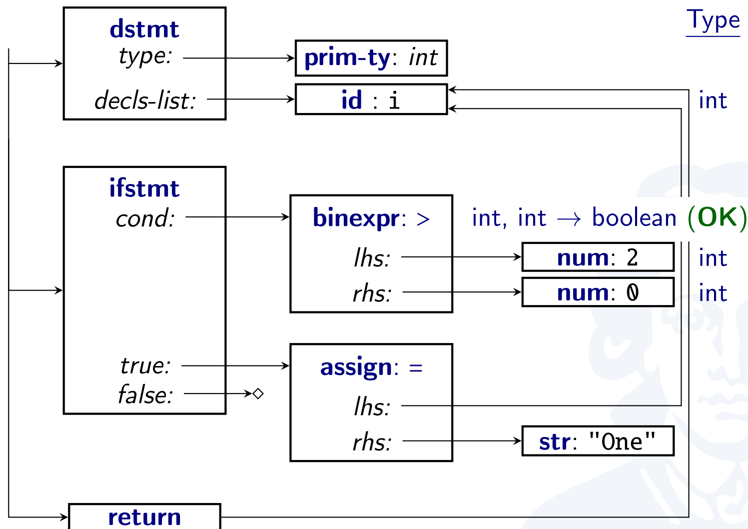
Type

int

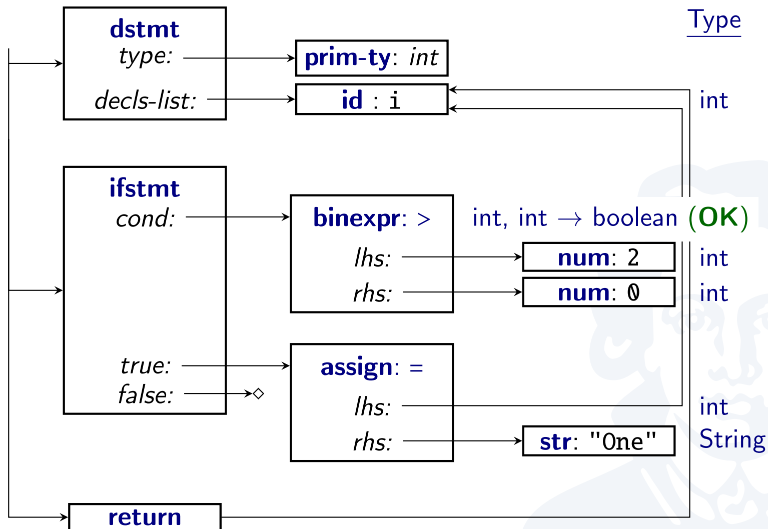
Type Analysis



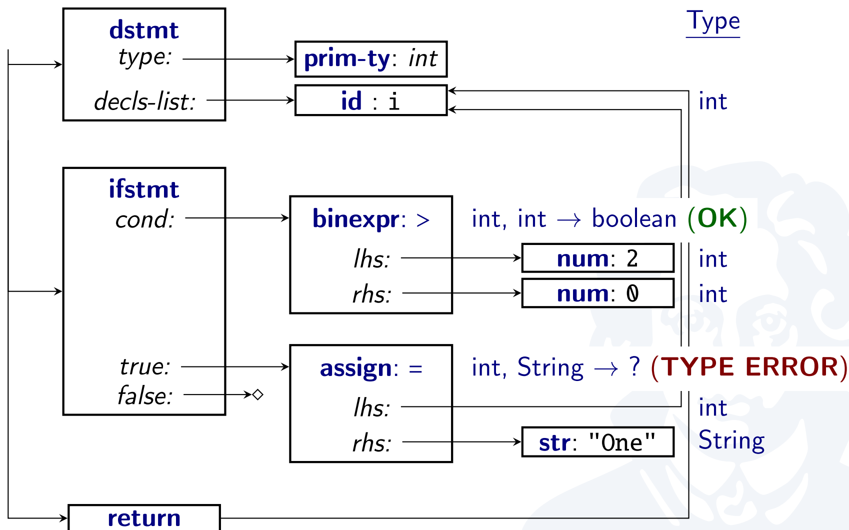
Type Analysis



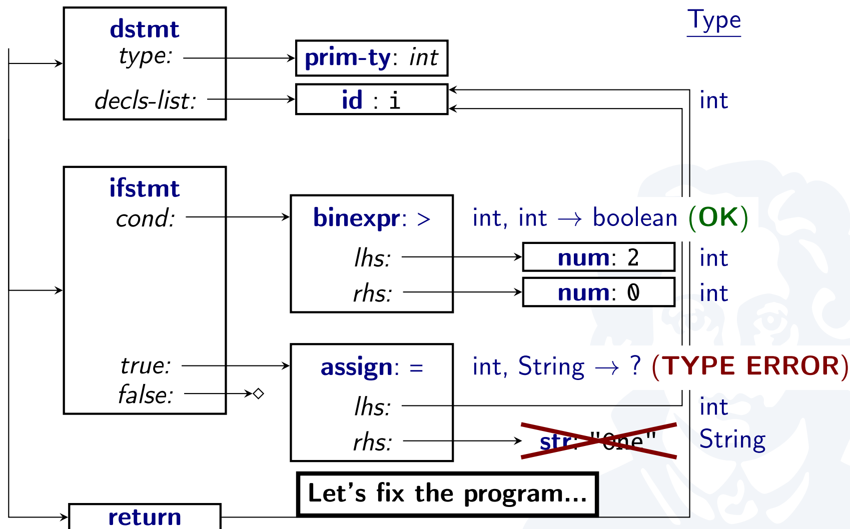
Type Analysis



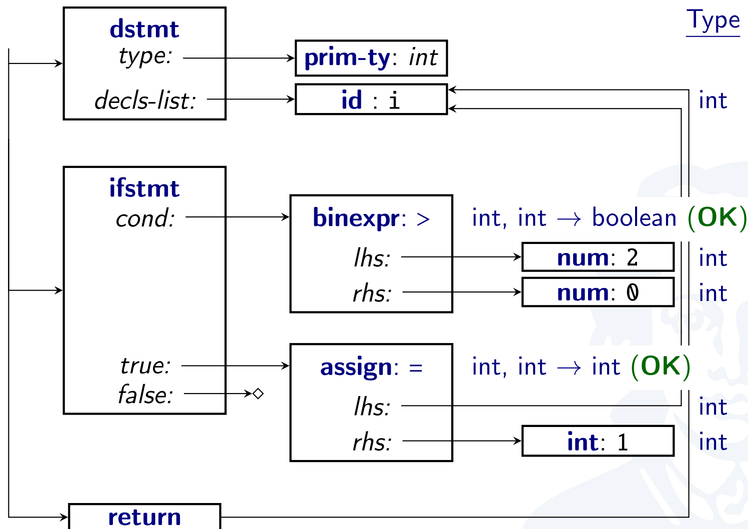
Type Analysis



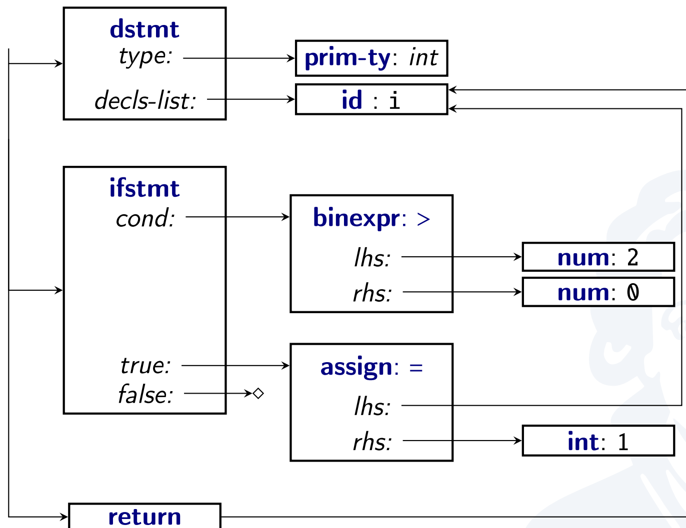
Type Analysis



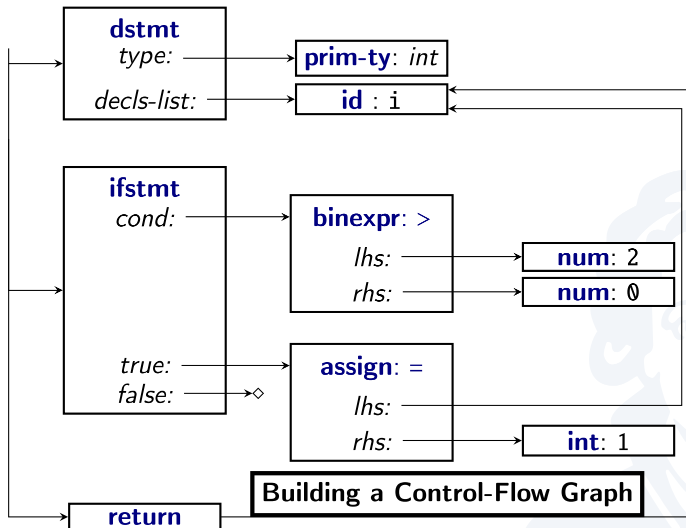
Type Analysis



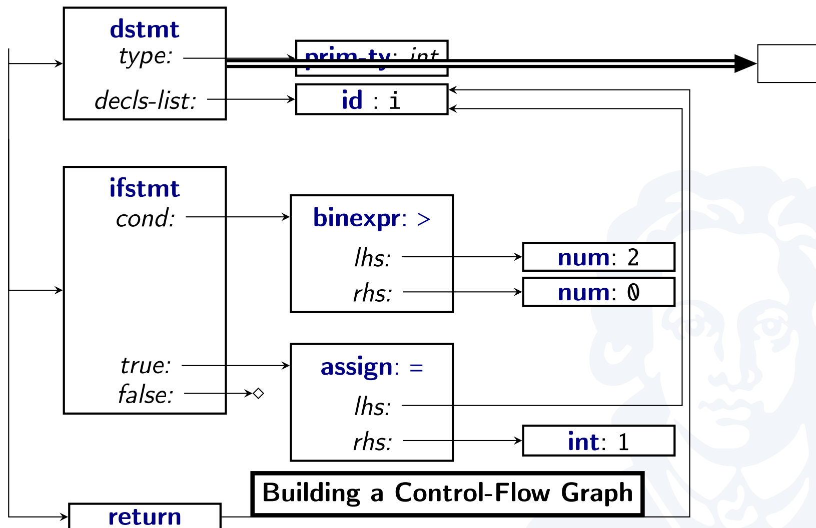
Definite Assignment Analysis



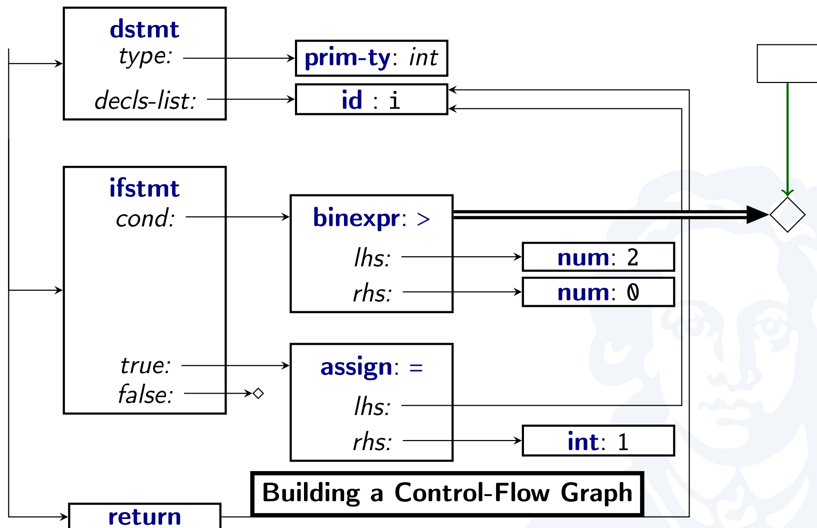
Definite Assignment Analysis



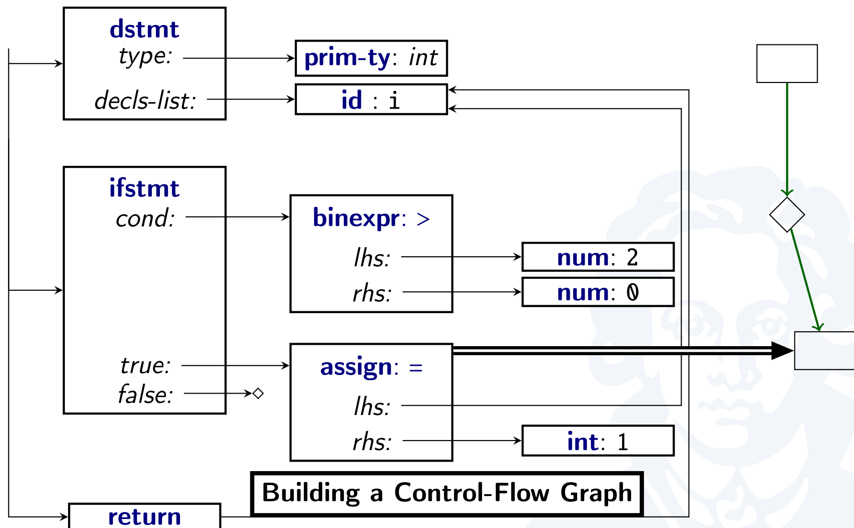
Definite Assignment Analysis



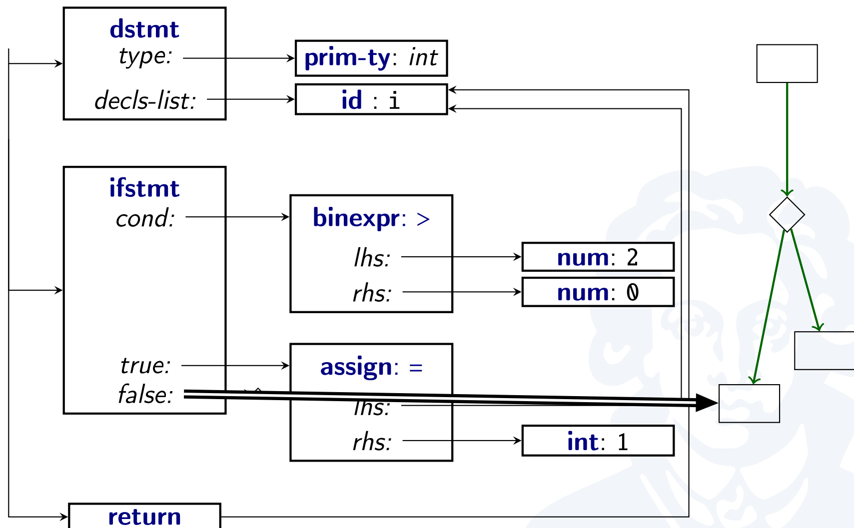
Definite Assignment Analysis



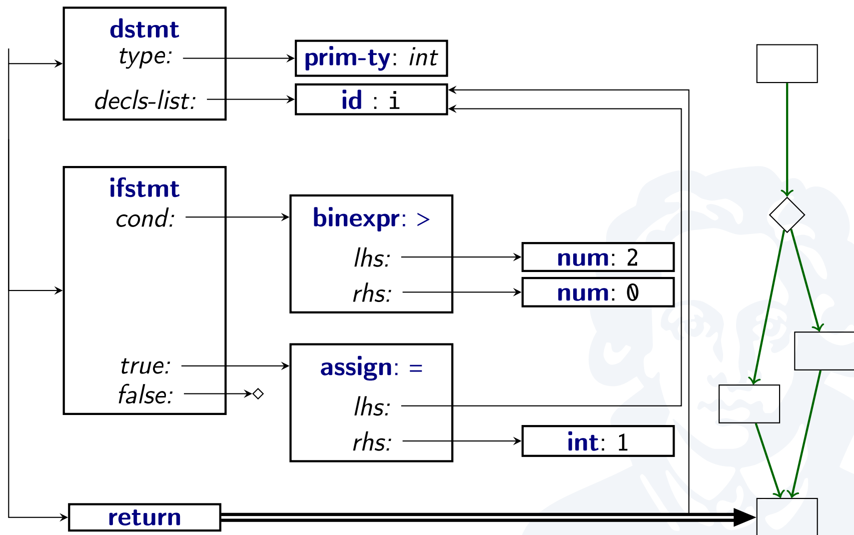
Definite Assignment Analysis



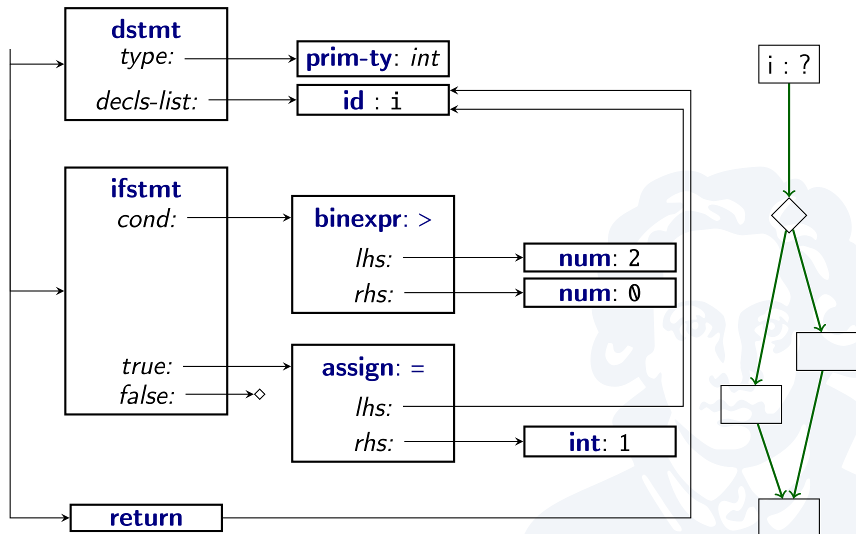
Definite Assignment Analysis



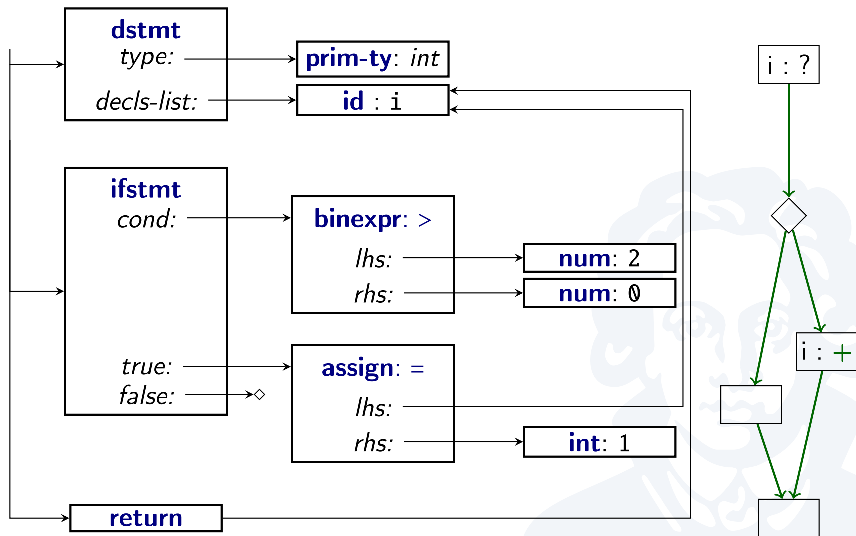
Definite Assignment Analysis



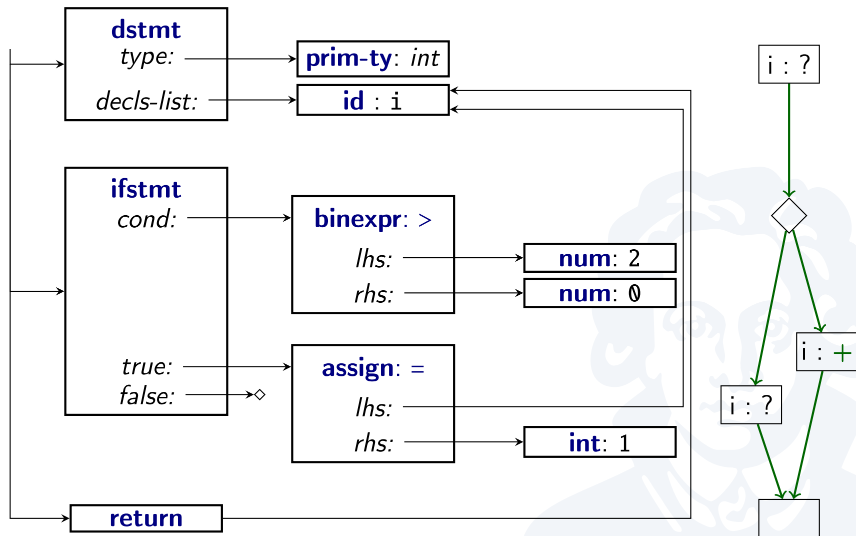
Definite Assignment Analysis



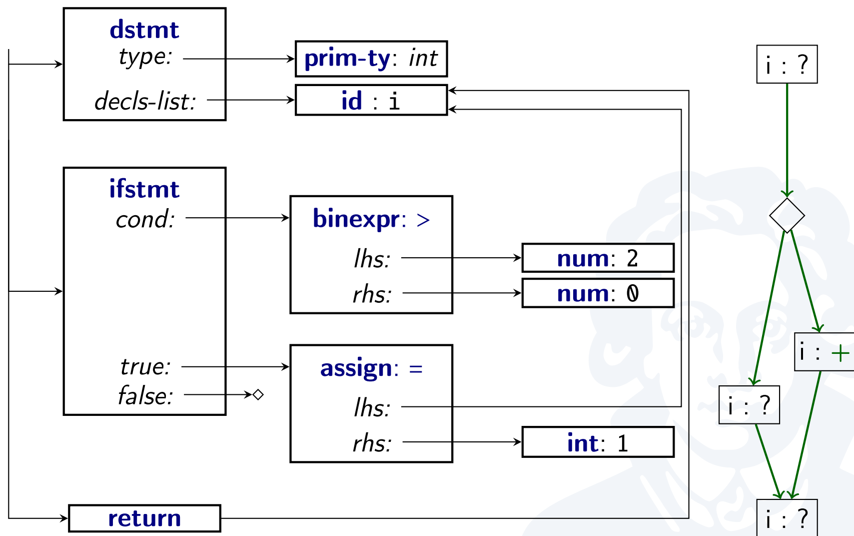
Definite Assignment Analysis



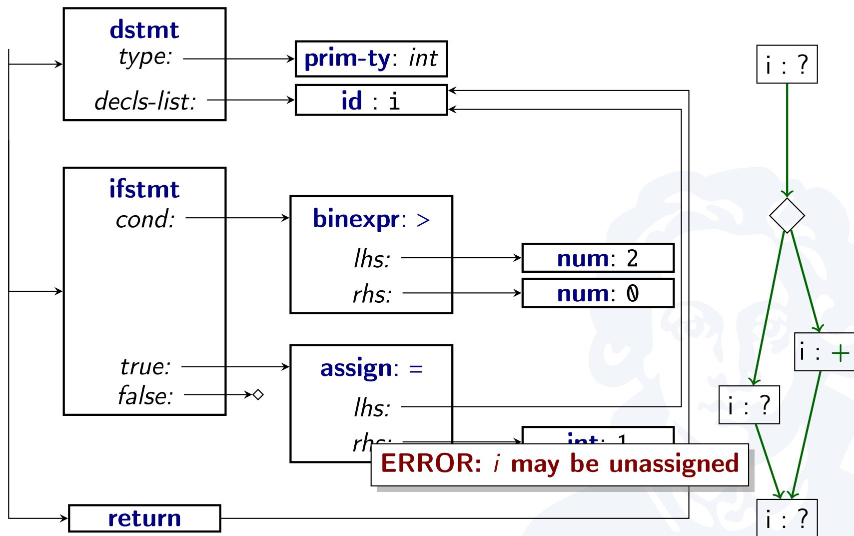
Definite Assignment Analysis



Definite Assignment Analysis



Definite Assignment Analysis



Definite Assignment Analysis

Definite Assignment Analysis:

- Ensures that any variable is assigned prior to use
- Is example of a *data-flow analysis*:

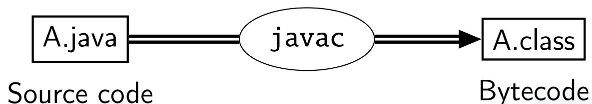
Let's fix the program *again*:

```
int i;
if (2 > 0) {
  i = 1;
} else {
  i = 0;
}
return i;
```

Emitting code

The compiler backend emits bytecode from the AST structure:

- Involves additional steps




```
...
int i;
if (2 > 0) {
  i = 1;
} else {
  i = 0;
}
return i;
...
```

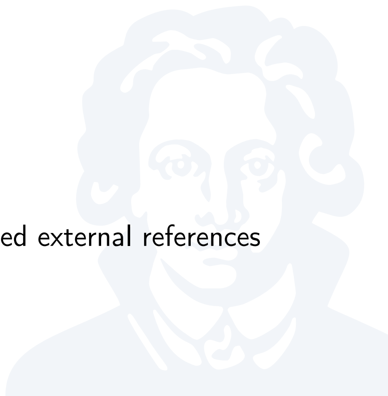
```
...
0:    iconst_2
1:    ifle 9
4:    iconst_1
5:    istore_1
6:    goto 11
9:    iconst_0
10:   istore_1
11:   iload_1
12:   ireturn
...
```

Would typically optimise this bytecode, too. . .

Java Runtime: Loading and Linking Bytecode

Scenario:

- During execution:
  Class `B` calls our code in class `A`
- The Java *Class loader*:
  - finds class `A` (e.g., on harddisk)
  - loads class into RAM
  - *validates* bytecode
  - registers class
  - asks *Linker* to resolve any required external references (none here)



Java Runtime: Interpretation

Bytecode is now in memory and gets *interpreted*:

```
...
     0:   iconst_2
     1:   ifle 9
     4:   iconst_1
     5:   istore_0
     6:   goto 11
     9:   iconst_0
     10:  istore_0
     11:  iload_0
     12:  ireturn
...
```

Java Runtime: Interpretation

Bytecode is now in memory and gets *interpreted*:

```
...
⇒   0:   iconst_2
    1:   ifle 9
    4:   iconst_1
    5:   istore_0
    6:   goto 11
    9:   iconst_0
    10:  istore_0
    11:  iload_0
    12:  ireturn
...
```

Java Runtime: Interpretation

Bytecode is now in memory and gets *interpreted*:

```
...
       0:   iconst_2    Load constant 2
⇒      1:   ifle 9
       4:   iconst_1
       5:   istore_0
       6:   goto 11
       9:   iconst_0
       10:  istore_0
       11:  iload_0
       12:  ireturn
...
```

Java Runtime: Interpretation

Bytecode is now in memory and gets *interpreted*:

```
...
        0:   iconst_2     Load constant 2
        1:   ifle 9       2 <= 0? No, so continue
  =>    4:   iconst_1
        5:   istore_0
        6:   goto 11
        9:   iconst_0
        10:  istore_0
        11:  iload_0
        12:  ireturn
...
```

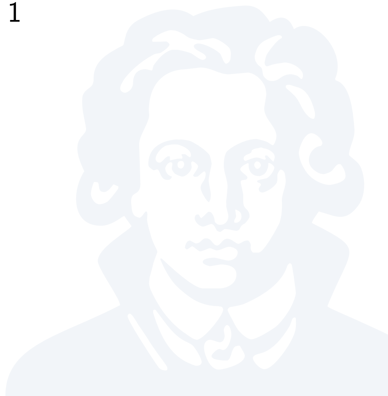
Java Runtime: Interpretation

Bytecode is now in memory and gets *interpreted*:

...

	0:	iconst_2	Load constant 2
	1:	ifle 9	2 <= 0? No, so continue
	4:	iconst_1	load the value 1
⇒	5:	istore_0	
	6:	goto 11	
	9:	iconst_0	
	10:	istore_0	
	11:	iload_0	
	12:	ireturn	

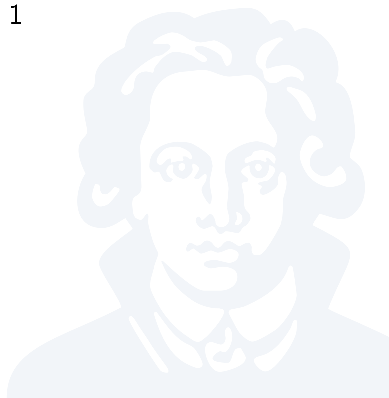
...



Java Runtime: Interpretation

Bytecode is now in memory and gets *interpreted*:

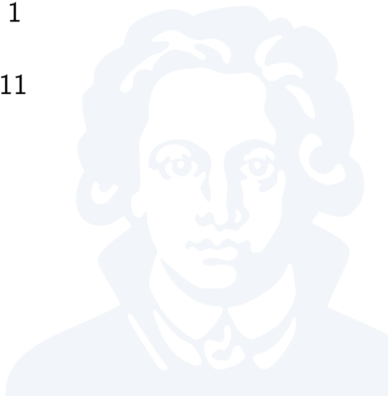
```
...
        0:   iconst_2     Load constant 2
        1:   ifle 9       2 <= 0? No, so continue
        4:   iconst_1     load the value 1
        5:   istore_0     i := 1
  ⇒     6:   goto 11
        9:   iconst_0
        10:  istore_0
        11:  iload_0
        12:  ireturn
...
```



Java Runtime: Interpretation

Bytecode is now in memory and gets *interpreted*:

```
...
        0:   iconst_2     Load constant 2
        1:   ifle 9       2 <= 0? No, so continue
        4:   iconst_1     load the value 1
        5:   istore_0     i := 1
        6:   goto 11      jump to label 11
        9:   iconst_0
        10:  istore_0
  ⇒     11:  iload_0
        12:  ireturn
...
```



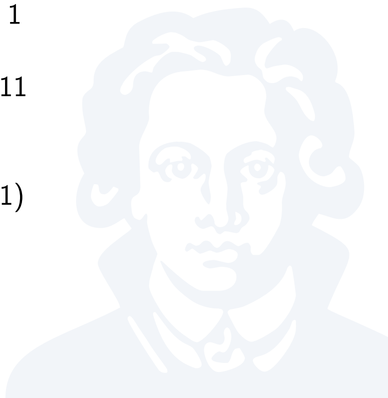
Java Runtime: Interpretation

Bytecode is now in memory and gets *interpreted*:

...

	`0:`	`iconst_2`	Load constant 2
	`1:`	`ifle 9`	2 <= 0? No, so continue
	`4:`	`iconst_1`	load the value 1
	`5:`	`istore_0`	i := 1
	`6:`	`goto 11`	jump to label 11
	`9:`	`iconst_0`	
	`10:`	`istore_0`	
	`11:`	`iload_0`	Load *i* (value 1)
⇒	`12:`	`ireturn`	

...



Java Runtime: Interpretation

Bytecode is now in memory and gets *interpreted*:

...

`0:`	`iconst_2`	Load constant 2
`1:`	`ifle 9`	2 <= 0? No, so continue
`4:`	`iconst_1`	load the value 1
`5:`	`istore_0`	i := 1
`6:`	`goto 11`	jump to label 11
`9:`	`iconst_0`	
`10:`	`istore_0`	
`11:`	`iload_0`	Load *i* (value 1)
`12:`	`ireturn`	Return 1.

...

And the method returns.

Java Runtime: Just-in-time compilation

- Interpretation is fine if code only used once...
- ...very inefficient for important code.
- Option: Compile code in JVM
  - Great for important code
  - Wasteful for unimportant code: *compilation takes time, too!*

Java Runtime: Just-in-time compilation

- Interpretation is fine if code only used once...
- ...very inefficient for important code.
- Option: Compile code in JVM
  - Great for important code
  - Wasteful for unimportant code:
    compilation takes time, too!

Let's measure how important the code is!

Java Runtime: Hotness Profiling

```
hotness[METHOD_ID]++
if (hotness[METHOD_ID] > 3) {
    m = compile(METHOD_ID);
    replace(METHOD_ID, m);
    return m();
} else {
            0:    iconst_2
            1:    ifle 9
            4:    iconst_1
            5:    istore_1
            6:    goto 11
            9:    iconst_0
            10:   istore_1
            11:   iload_1
            12:   ireturn
}
```

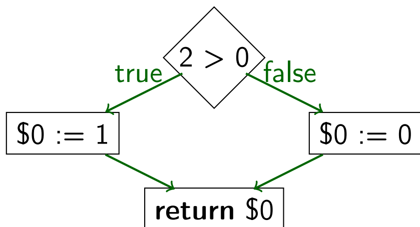
Java Runtime: Hotness Profiling

```
hotness[METHOD_ID]++
if (hotness[METHOD_ID] > 3) {
    m = compile(METHOD_ID);
    replace(METHOD_ID, m);
    return m();
} else {
            0:    iconst_2
            1:    ifle 9
            4:    iconst_1
            5:    istore_1
            6:    goto 11
            9:    iconst_0
            10:   istore_1
            11:   iload_1
            12:   ireturn
}
```

The JVM only compiles ‘hot’ methods.

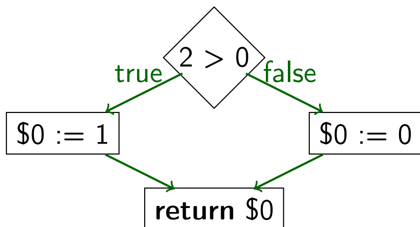
Java Runtime: Dynamic Compilation

- Just-in-time compiler translates bytecode into intermediate representation:

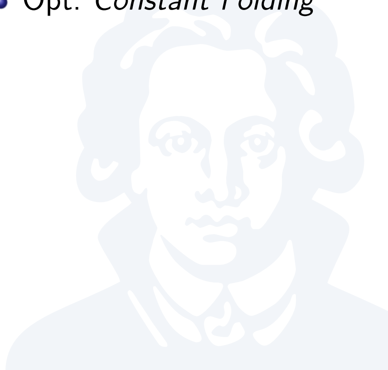


Java Runtime: Dynamic Compilation

- Just-in-time compiler translates bytecode into intermediate representation:

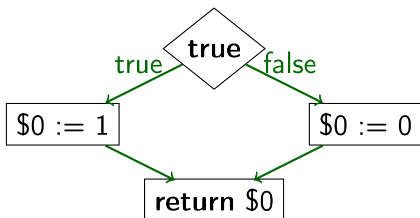


- Opt: *Constant Folding*

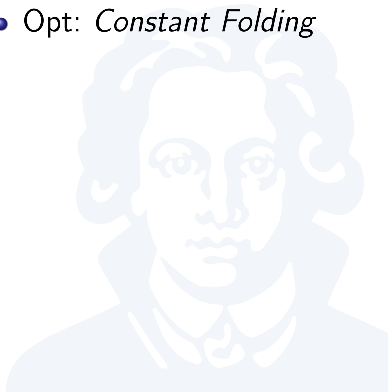


Java Runtime: Dynamic Compilation

- Just-in-time compiler translates bytecode into intermediate representation:

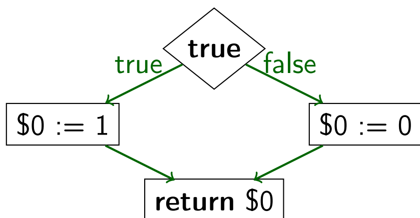


- Opt: *Constant Folding*



Java Runtime: Dynamic Compilation

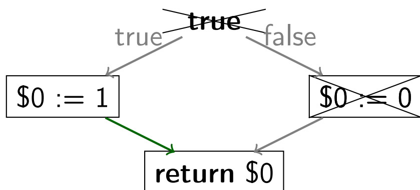
- Just-in-time compiler translates bytecode into intermediate representation:



- Opt: *Constant Folding*
- Opt: *Dead code elimination*

Java Runtime: Dynamic Compilation

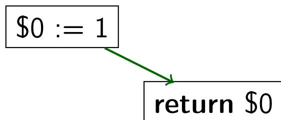
- Just-in-time compiler translates bytecode into intermediate representation:



- Opt: *Constant Folding*
- Opt: *Dead code elimination*

Java Runtime: Dynamic Compilation

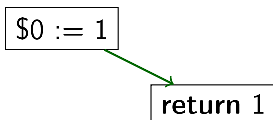
- Just-in-time compiler translates bytecode into intermediate representation:



- Opt: *Constant Folding*
- Opt: *Dead code elimination*
- Opt: *Constant Propagation*

Java Runtime: Dynamic Compilation

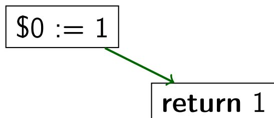
- Just-in-time compiler translates bytecode into intermediate representation:



- Opt: *Constant Folding*
- Opt: *Dead code elimination*
- Opt: *Constant Propagation*

Java Runtime: Dynamic Compilation

- Just-in-time compiler translates bytecode into intermediate representation:



Might compile to e.g.:

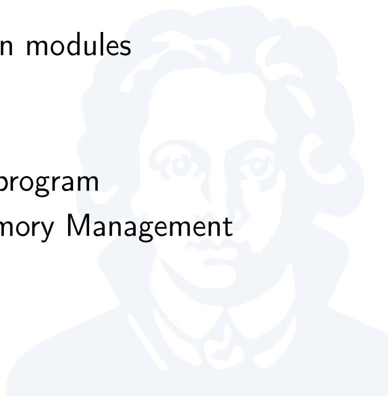
```
b8 01 00 00 00   mov eax,0x1
c3               ret
```

- Opt: *Constant Folding*
- Opt: *Dead code elimination*
- Opt: *Constant Propagation*

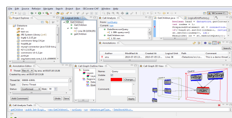
Java Runtime System

Many duties:

- *(Class) Loader*: Loads bytecode
- *Linker*: Resolves references between modules
- *Interpreter*: Slow execution
- *Profiler*: Find important code
- *(Just-in-time) Compiler*: Compile program
- *Garbage Collector*: Automatic Memory Management



Summary: Components of Software Tools



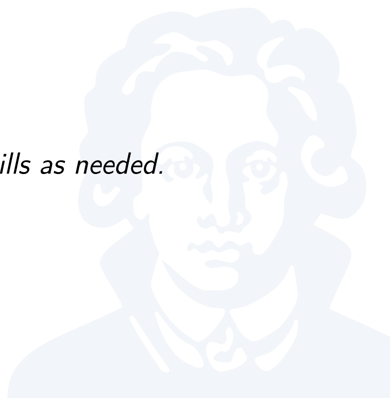
```
for (int i = 0; i < 10; i++) {
  z += i;
  if (z > 42) {
    System.out.println(z);
    z /= 10;
} }
```

```
E8 03 CD 10 c5 01 45 39 e5
38 83 eb 04 4c 39 ff 0f 84
01 45 84 f6 48 89 44 24 38
af 84 d2 79 ab 89 d0 83 e0
74 24 38 31 d2 4c 89 44 24
48 89 7c 24 38 e8 66 92 01
```

Preview: Lectures 01–03

- 01: Syntax, Semantics, Types
- 02: Static Program Analysis
- 03: Dynamic Program Analysis

You are expected to pick up additional skills as needed.

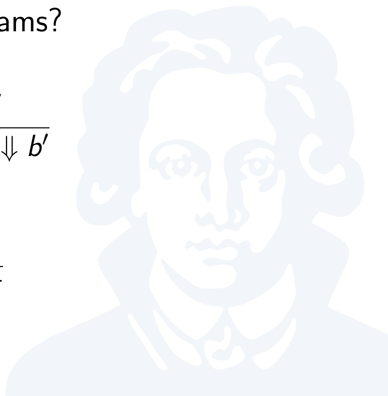


Preview: 01: Syntax, Semantics, Types

- How does meaning follow from structure?
- What is the purpose of types for meaning?
- What do types tell us about programs?

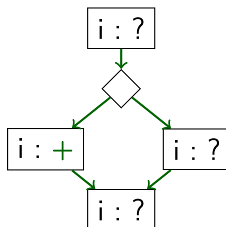
$$\frac{a \Downarrow \text{true} \quad b \Downarrow b'}{\textbf{if } a \textbf{ then } b \textbf{ else } c \Downarrow b'}$$

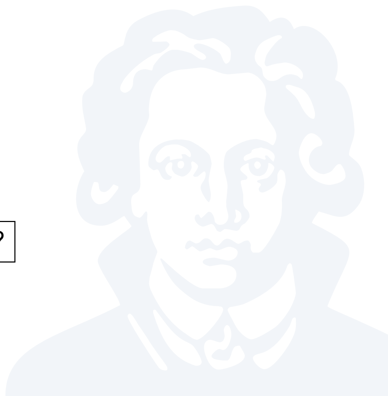
$$\frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma : \lambda x.e : \tau \to \sigma}$$



Preview: 02: Static Program Analysis

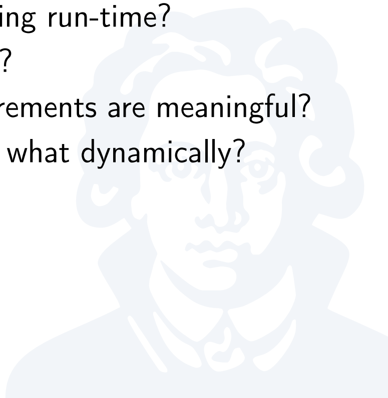
- How can we analyse programs?
  - Type / Effect analysis
  - Flow analysis
  - Constraint analysis
  - Abstract Interpretation





Preview: 03: Dynamic Program Analysis

- How can we analyse programs during run-time?
- What can we measure at run-time?
- How can we make sure our measurements are meaningful?
- What can/should we do statically, what dynamically?



Project proposals

- The following proposals are some possible ideas
- Details have to be worked out
- Proposals available first-come-first-serve
- *Other suggestions are very welcome!*

Contact me *before* May to start discussing options!

Project #1: Automatic (De)serialisation

Joint project with Prof. Dr. Koch, Bioinformatics

- *Serialisation*: information → byte stream
- Writing objects to disk, to the network, ...
- Java has built-in serialisation support
  - *However:* when classes change, loading fails

Goal:

- Design & build serialisation extension to Java
- Use Google Protocol Buffers
- Integrate into the Bioinformatics MonaLisa framework

Project #2: Fix-Up for Program Metamorphosis

Program Metamorphosis:

- Generalisation of *Refactoring*:
  - Source-to-source transformations
  - Individual transformations may alter behaviour
  - Sum of all transformations guaranteed to restore behaviour

Goal:

- Port existing Program Metamorphosis Eclipse plugin to latest Eclipse
- Extend plugin to allow adding elements
- Design & build automatic recovery mechanism

Project #3: Automatic Axiom Inference over Java

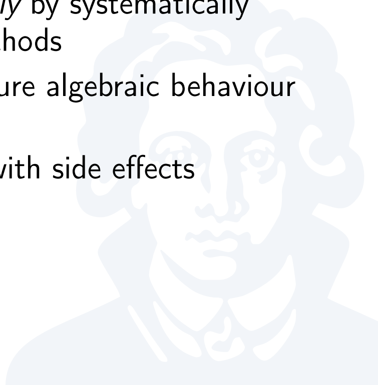
Axioms can be used to describe program behaviour:

- $\forall s, x.(s.\text{push}(x).state).\text{pop}().value = x$
- $\forall s, x.(s.\text{push}(x).state).\text{pop}().state = s$

Project #3: Automatic Axiom Inference over Java

Axioms can be used to describe program behaviour:

- $\forall s, x.(s.\text{push}(x).state).\text{pop}().value = x$
- $\forall s, x.(s.\text{push}(x).state).\text{pop}().state = s$
- These can be inferred *automatically* by systematically combining & calling Java API methods
- With enough axioms, we can capture algebraic behaviour of datatypes
- Doesn't scale well for operations with side effects



Project #3: Automatic Axiom Inference over Java

Axioms can be used to describe program behaviour:

- $\forall s, x.(s.\text{push}(x).\mathit{state}).\text{pop}().\mathit{value} = x$
- $\forall s, x.(s.\text{push}(x).\mathit{state}).\text{pop}().\mathit{state} = s$
- These can be inferred *automatically* by systematically combining & calling Java API methods
- With enough axioms, we can capture algebraic behaviour of datatypes
- Doesn't scale well for operations with side effects

Goal

- Extend existing tool for axiom inference:
  - Use bytecode analysis: what side effects are possible?
  - Expand axiom search to include side effects

Project #4: Probabilistic Modelling in Ren'Py

- Computer game design often involves randomness
- Large numbers of random factors can make it hard to balance the game

Goal:

- For a specially selected subset of the game framework Ren'Py:
- Use *Abstract Interpretation* to capture dependencies between
  - Randomly generated numbers
  - User selections
- Build statistical models
- Use formal reasoning or statistical sampling to answer questions about game balance

Project #5: Automatic Build File Generation

- Developers often experiment with various command-line tools
- Later go back, write their findings into scripts / build files

Goal:

- Automate this process:
- Use Linux `ptrace` mechanism to observe:
  - Which process creates which file?
  - With what command line?
  - From what input files?
- Collect observations
- Propose abstractions over collected data
- Write `Makefile` or similar specification that captures observations

Project #6: Infer Operational Semantics

- Assume as given:
  - A complete language syntax
  - An interpreter for the language
  - A means for obtaining the final last computed value, or 'timeout', or 'error'

Goal:

- Design systematic search that infers language semantics as rules
- Try to make 'reasonable' assumptions
- Can this be extended to handle side effects (e.g., via `ptrace`)?
- Try it out with an existing language (e.g., lua)

Project #7: PQL/C++

PQL/Java is a Java language extension that adds expressions such as:

```
Set<Integer> s = query(Set.contains(x)):
                    s1.contains(x)
                 && s2.contains(x)
                 && x > 10;
```

- Set intersection with value filter
- PQL can choose different implementation strategies for this query

Goal:

- Build a prototype for a similar system for C++

Next week:

Syntax, Semantics, Types