# ExtendJ

The Extensible Java Compiler

Jesper Öqvist

# About Me

I work with Robots (Cognibotics AB) and Teaching (Lund University)

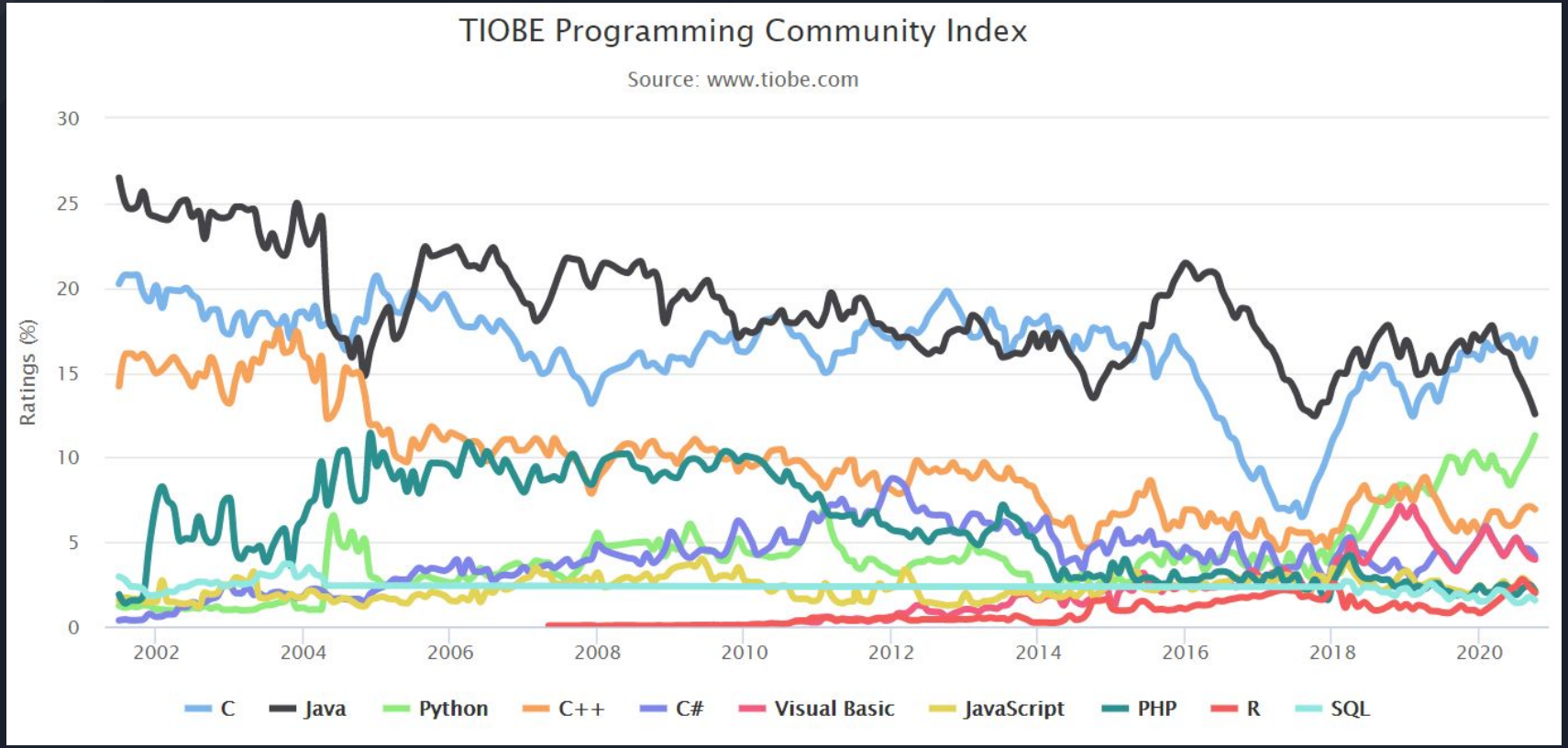
# What I Like About Compilers

- Parsing
  - Context-Free Grammars
  - LR parsing
- Static Analysis
  - Name analysis
  - Type Checking
  - Type Inference
- Code Generation
  - Bytecode/Machine code generation
- Optimizations
  - Common subexpression elimination
  - Inlining
  - Register allocation

# Compiler Engineers

- New general purpose languages (e.g. Julia, Swift, Rust)
- New Domain-Specific Languages (DSL)
- New compilers for old languages (COBOL, old DSLs, etc.)
- Maintenance of existing compilers
- Development of existing languages
- New compiler optimizations
- Smart programming tools

# Programming Language Trends



# ExtendJ - Extensible Java Compiler

A declaratively specified Java compiler.

Can be extended with JastAdd attributes (Java language extensions).

Demonstration of a large-scale compiler written with JastAdd.

extendj.org

# ExtendJ - Extensible Java Compiler

| | |
|---|---|
| Java 4-6 | **Torbjörn Ekman** |
| Java 7 | **Jesper Öqvist** (Master's Thesis) |
| Java 8 | **Erik Hogeman** (Master's Thesis) |

Maintainer: **Jesper Öqvist**

License: Modified BSD

extendj.org

# Compiling Java vs SimpliC

| | SimpliC | ExtendJ |
|---|---|---|
| Parser (LOC) | 312 | 1 736 |
| Classes | 48 | 263 |
| Attributes (LOC) | 6 000 | 23 000 |

extendj.org

# ExtendJ Goals

ExtendJ should be easy to extend with

- Static Analyses
- Language Features
- Metrics, Refactoring, Test Selection,
- etc.

Used for

- Research,
- PL experiments based on Java,
- Compiler course projects.

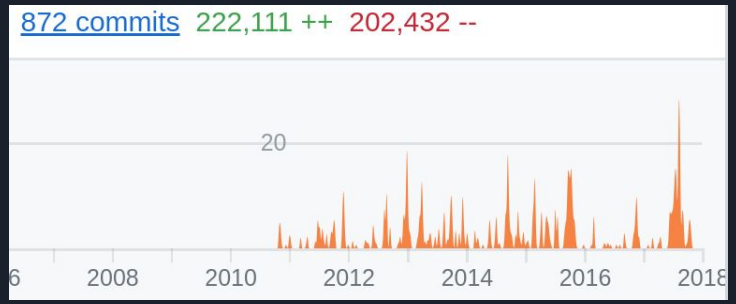
extendj.org

# My Work on ExtendJ

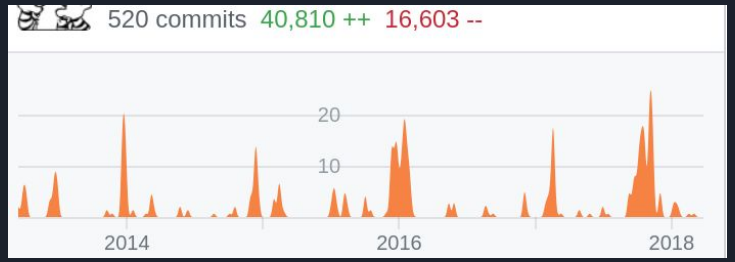
Added 6,7,8

General Improvements
Bug Fixes (300+)
Regression Tests (1000+)

Side Effect Removal




ExtendJ Commits




Regression Test Commits

# Student Project: Bug Pattern Checker

Ella Eriksson & Zimon Kuhs

```
String vegetable;
if (vegetable == “pizza”) …
```

# Student Project: Bug Pattern Checker

Ella Eriksson & Zimon Kuhs

```
String vegetable;
if (vegetable == “pizza”) …
```

```
=> Suggestion: replace == with .equals()
```

# Student Project: Spread Operator

Filip Stenström & Wawrzyn Chonewicz

```
Person[]  P;
...
String[]  N = P*.getName();
```

# Multiplicities Extension

I developed a Java language extension implementing Multiplicities by Friedrich Steimann:

```
@any Person people;
people += alice;        // += → .add()
people += bob;
people.work();          // Call .work() on alice and bob.
```

*Friedrich Steimann, Jesper Öqvist, Görel Hedin:*
*Multiplicities: First implementation and case study (JOT, 2014)*

# Multiplicities

In collaboration with professor Friedrich Steimann (Fernuniversität Hagen),
I wrote a language extension to support programming with Multiplicities:

```
Collection<Person> pl =
     new ArrayList<Person>();
pl.add(bob);
pl.add(alice);
for (Person p: pl) {
     p.print();
}
```

```
@any Person pl;
pl += bob;
pl += alice;
pl.print();
```

*Multiplicities: First implementation and case study (JOT, 2014)*

# Demo

<Demo> Multiplicities

extendj.org

*Friedrich Steimann, Jesper Öqvist, Görel Hedin:*
*Multiplicities: First implementation and case study (JOT, 2014)*

# Incremental Regression Testing

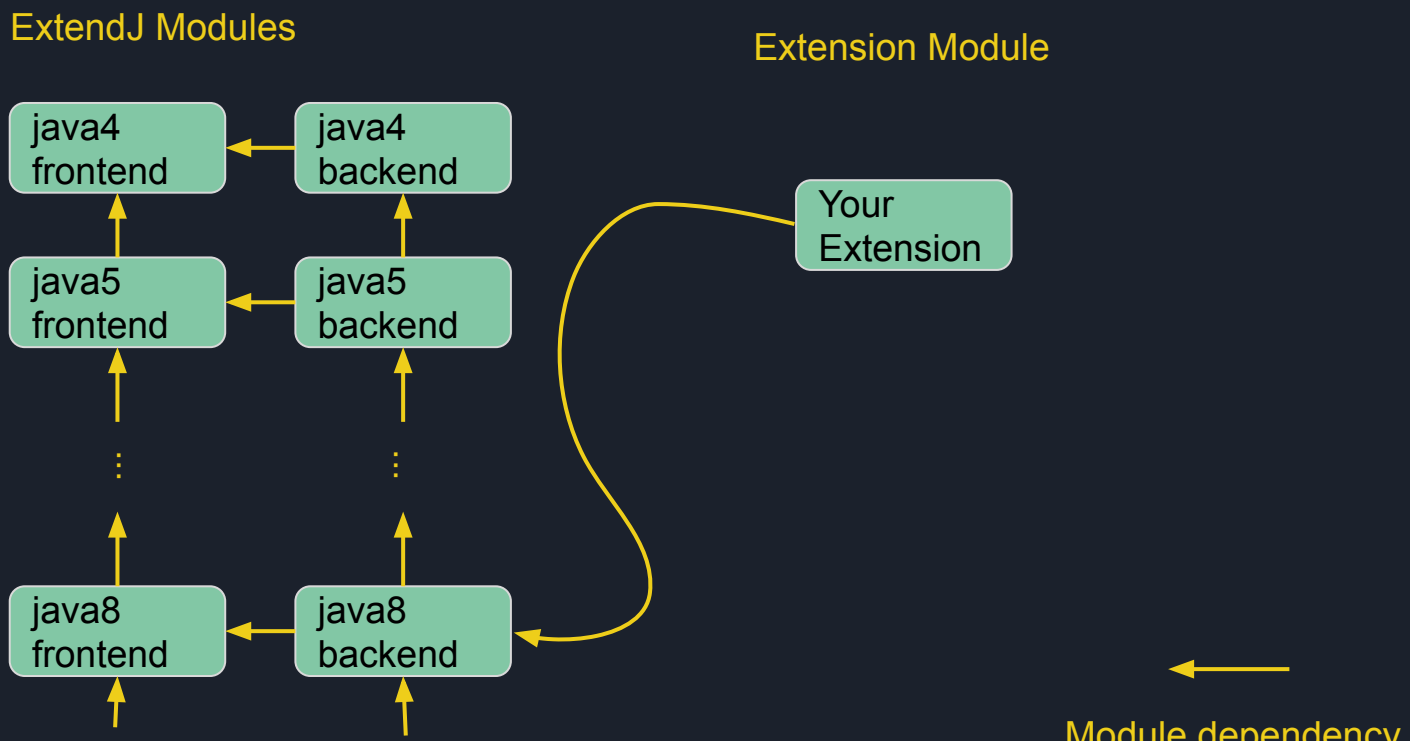
Our idea: run only the tests that are affected by the current change.

Regression tests need to be run after each change to the code to guard against defects.

Regression testing can be time consuming.

*Extraction-Based Regression Test Selection (PPPJ, ACM, 2016)*

# Building Extensions



# Compiler Passes

Typically, compilation is divided into passes.

**One Pass**: all translation done while parsing. Few languages are single-pass.

(C is one-pass. Declaration order matters!)

# Single Pass

*Storage limitations on the B compiler demanded a one-pass technique in which output was generated as soon as possible, and the syntactic redesign that made this possible was carried forward into C.*

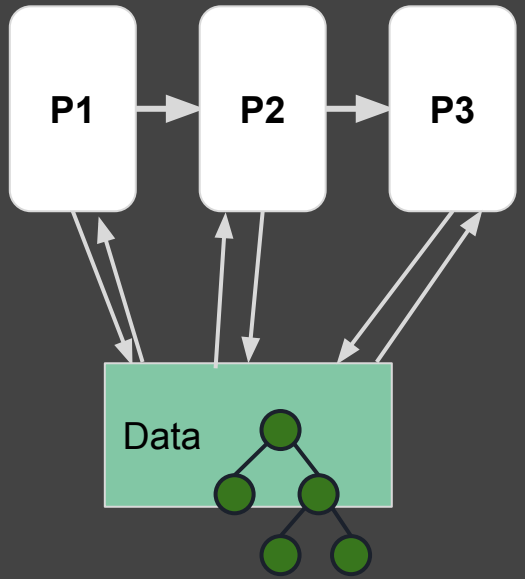
- Dennis M. Ritchie, The Development of the C Language

# DEC PDP-7

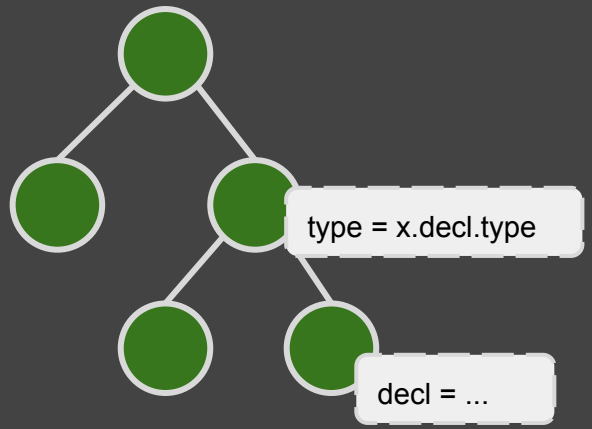
The B compiler ran on a PDP-7 which had about 18KB of memory.

## Pass-Oriented vs. Attribute-Oriented

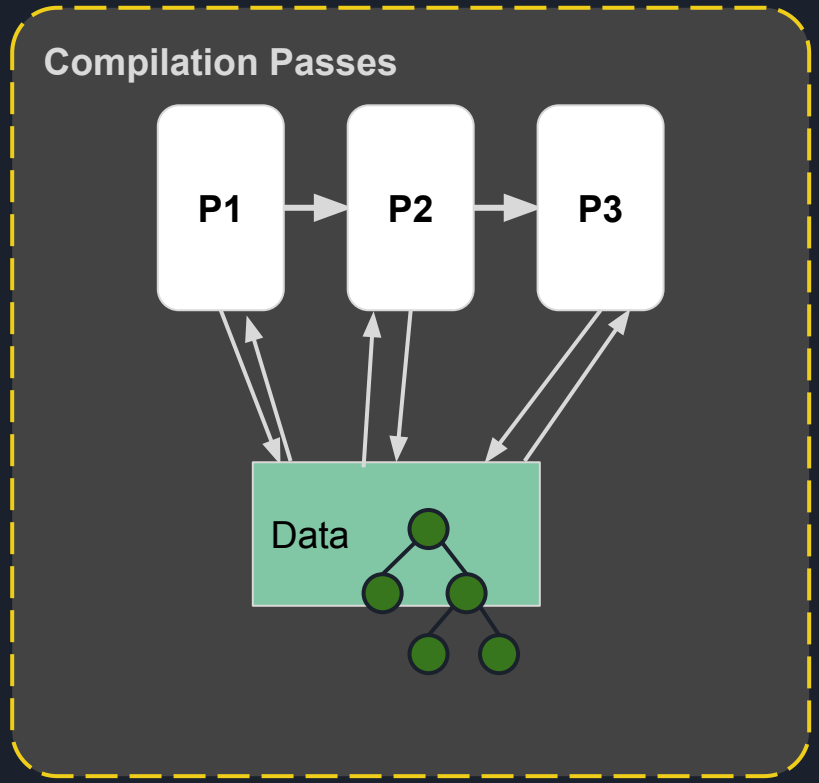
### Compilation Passes

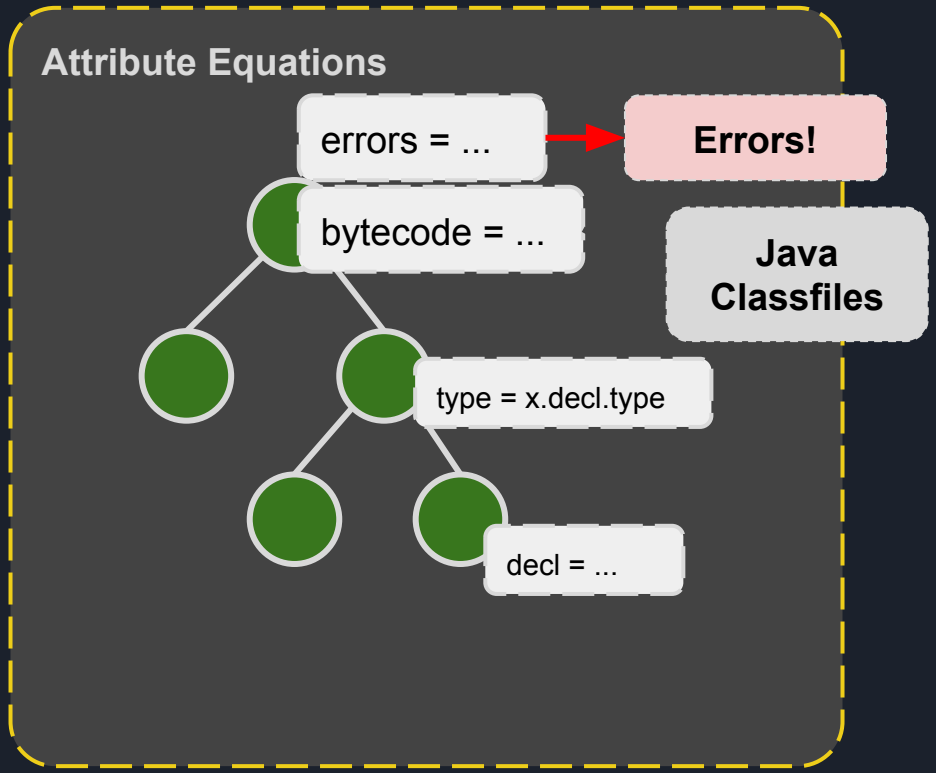



### Attribute Equations



## Pass-Oriented vs. Attribute-Oriented



# Attributes for Compilation

- Compilation is divided into small computations (attributes).
- Attributes are declarative

  Say what should be computed, not how to do it.
- Attributes have no side-effects.  Order independent!
- Attribute evaluator schedules attributes:
  - Memoization.
  - Demand evaluation.
  - Parallel computation.
  - Incremental evaluation.

# Memoization and Demand Evaluation

Memoization:

When attribute is computed, result stored for later reuse.

Demand Evaluation:

Attribute computed only if needed.

No redundant computation for unused features.

# Automatic Parallelization

Attributes are observationally pure:

- No side effects.
- Not order-dependent.

-> attributes can be parallelized.

Speedup depends on attribute structure.

For ExtendJ, speedup of 2x is possible.

# Attribute-Oriented Compiler

How to make a full compiler with attributes?

1. Split computations into meaningful attributes.
2. What should be synthesized/inherited?
3. What should be implicitly generated with higher-order attributes?

# ExtendJ Design

Specification divided into modules based on Java version.

All types are represented in AST (user-defined and primitives)

Generic types are represented with higher-order attributes.

Type and name lookup is demand-driven (no precomputation of symbol tables).

Minimal use of AST transforms (rewrites). Instead, try to use higher-order attrs.

# ExtendJ Challenges

Java is a very complicated language.

The official compliance test suite is proprietary, so we use our own regression tests and regular testing on Open Source projects to find errors.

ExtendJ is not perfectly compliant, but close enough for our needs.

Attributes have a performance cost. ExtendJ is a few times slower than javac.

# ExtendJ Overview: AST

Everything is a **declaration** or an **access**:

- TypeDecl
- MethodDecl
- VarAccess
- MethodAccess
- TypeAccess

# ExtendJ Overview: AST

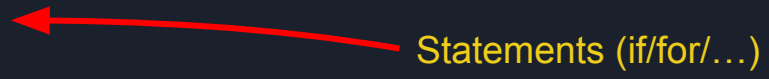
```
Program ::= CompilationUnit*;
CompilationUnit ::= TypeDecl*;
abstract TypeDecl ::= BodyDecl*;
ClassDecl : TypeDecl;
InterfaceDecl : TypeDecl;

abstract Stmt;
abstract Expr;

Access : Expr;
```





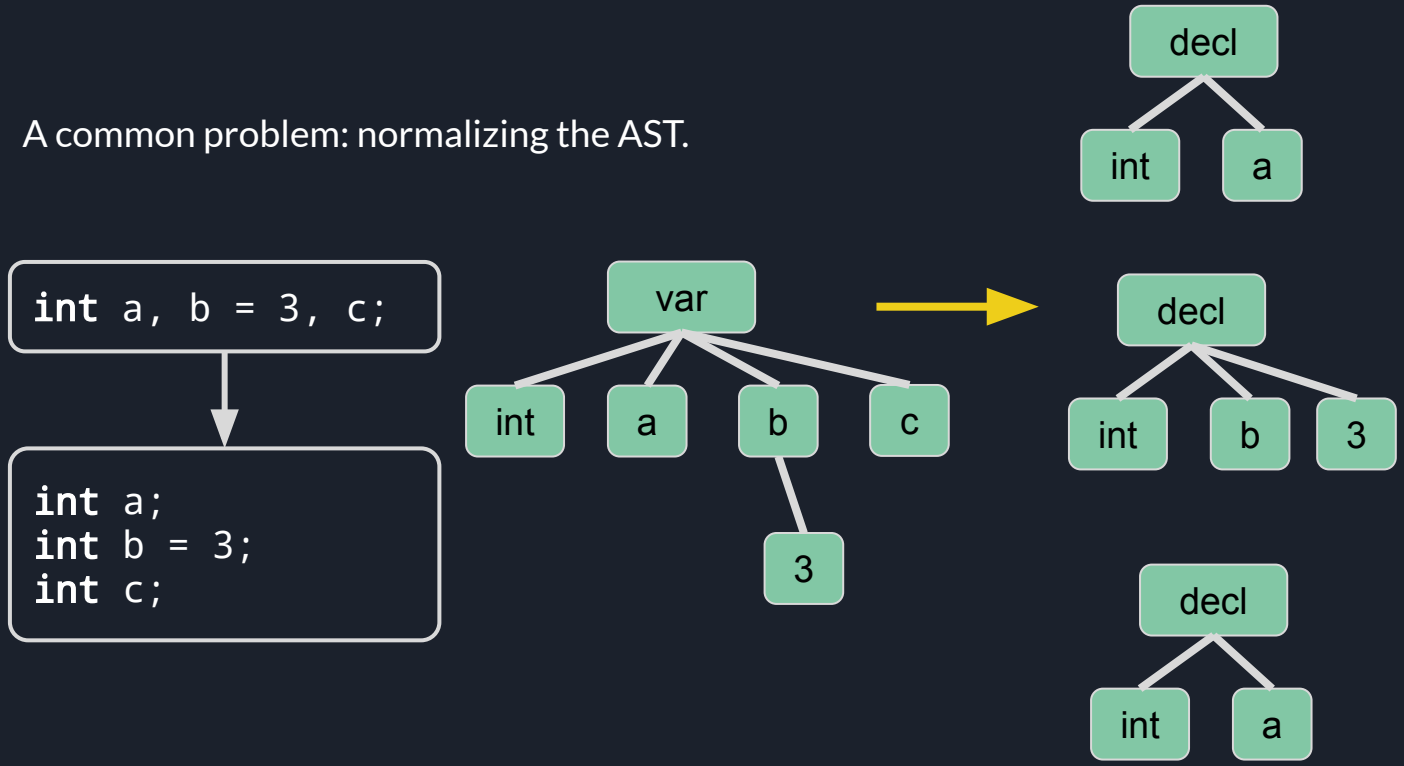



# ExtendJ Overview

- Name analysis
  - Classification, lookup
- Type analysis
  - Lookup, subtyping, generics, inference
- Control flow - exception handling, return checking
- Dataflow - definite assignment
- Normalization
  - Multiple declaration, enhanced for, try-with-resources, lambda
- Implicit code gen
  - Accessors, bridge methods
- Bytecode output

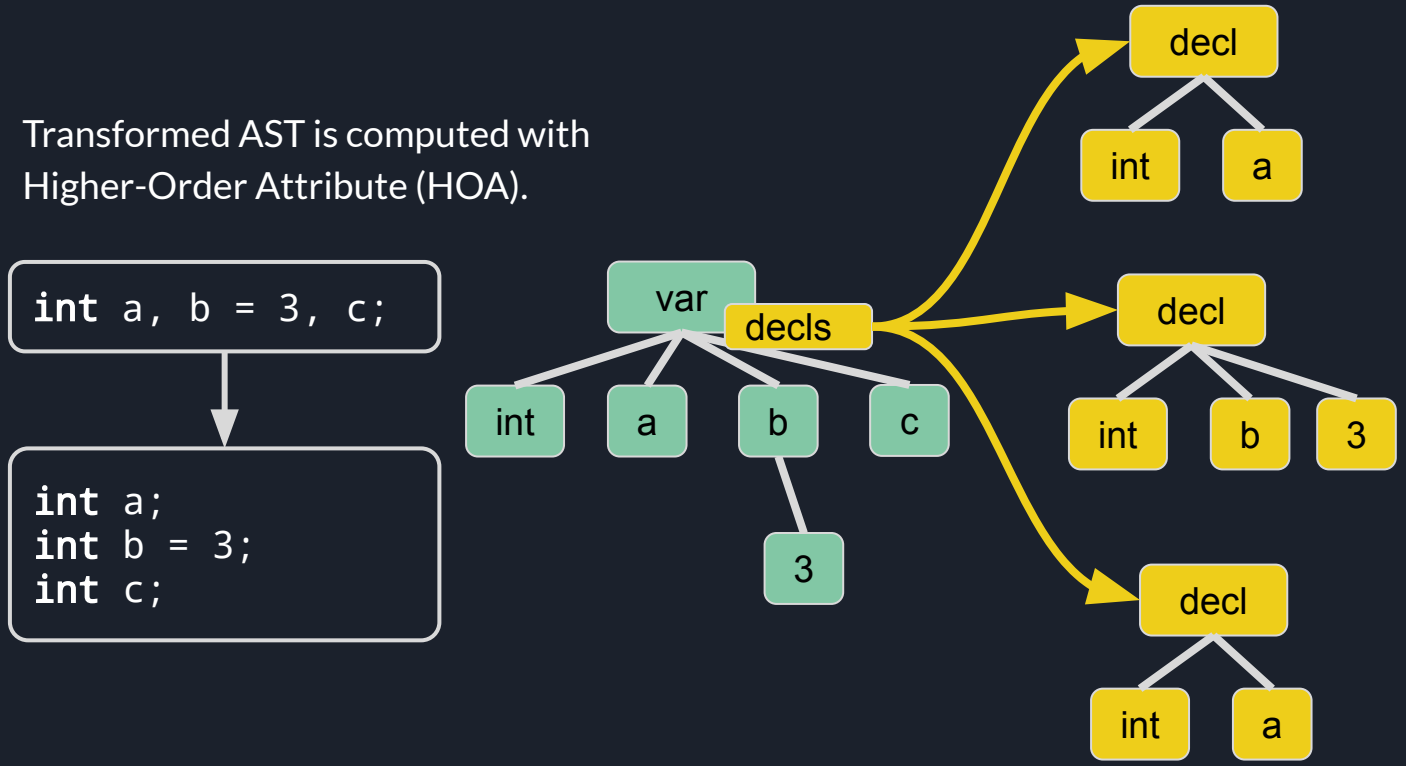
# AST Transformation

A common problem: normalizing the AST.



# AST Transformation

Transformed AST is computed with Higher-Order Attribute (HOA).

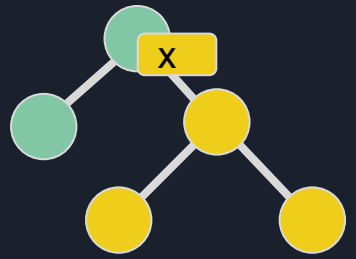


# Higher-Order Attributes

HOAs compute a new part of the AST (with attributes).

HOAs are used for:

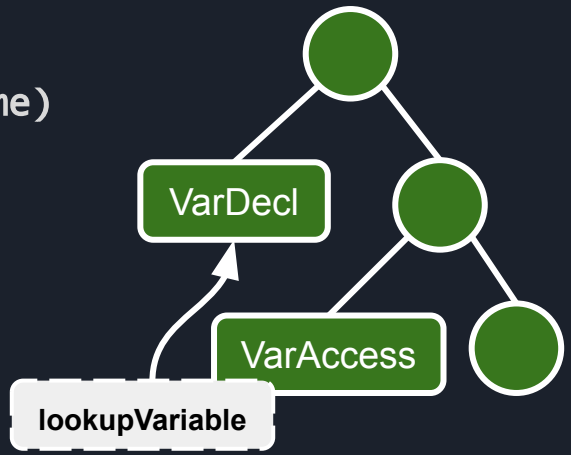
- AST transformation / normalization.
- Reifying implicit constructs.
- Code generation by desugaring.



# Name analysis: lookup

Inherited attributes for name lookups:

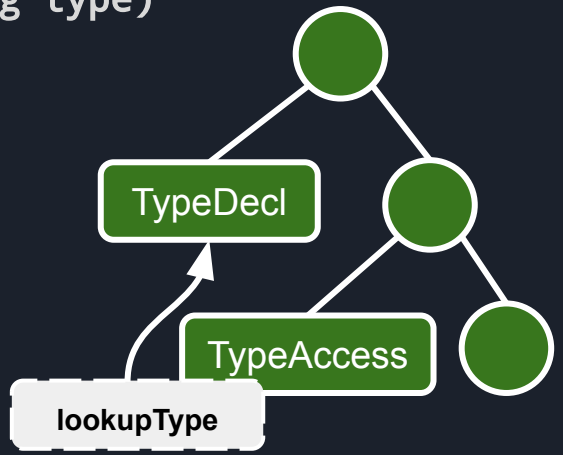
```
lookupVariable(String name)
lookupMethod(String name)
lookupConstructor(String name)
```



# Type analysis

Type lookup works like name lookup:

```
lookupType(String pkg, String type)
```



# Type Analysis: Subtyping

Double dispatch:

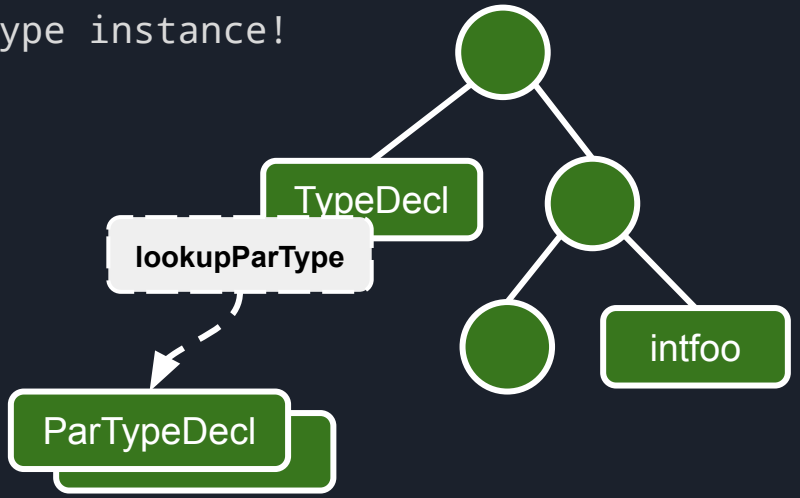
```
syn boolean TypeDecl.subtype(TypeDecl other);
eq ClassDecl.subtype(TypeDecl other) =
        other.subtypeClassDecl(this);

syn boolean TypeDecl.subtypeClassDecl(ClassDecl other) = false;
eq ClassDecl.subtypeClassDecl(ClassDecl other) = …;
```

# Type Analysis: Implicits

Higher-order attributes for implicit types
(primitives, parameterized types)

```
class Foo<T> {}
Foo<Integer> intfoo;  // Need type instance!
```

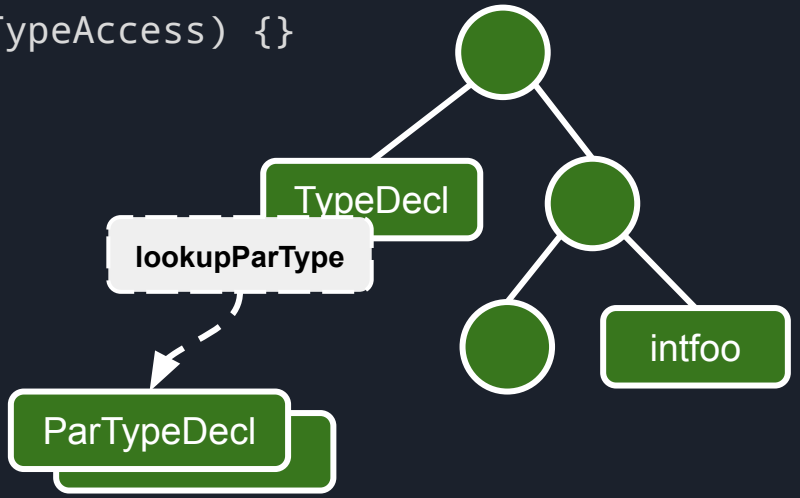


# Type Analysis: Implicits

Higher-order attributes for implicit types
(primitives, parameterized types)

```
nta ParTypeDecl
    TypeDecl.lookupParType(ParTypeAccess) {}

ParClassDecl : ClassDecl ::=
    Parameterization;
```



# Extensibility

Extensibility is easy with JastAdd:

- Fine-grained control with Aspect Oriented Programming and Attributes.
- Extensions can change everything!

However, this leads to fragile extensions:

- Everything is exposed for extension
  => can not change internals without affecting existing extensions!

Open problem: ExtendJ needs to be refactored to continue development, but how do we do this without hurting existing users?

# Parallelizing ExtendJ

I have been working on parallelizing ExtendJ using concurrent attribute evaluation.

The compiler runs twice as fast (in error checking) with parallelization.

# Demo

<Demo> Parallel execution.

extendj.org

# Building Extensions

I wrote a Gradle plugin to easily build extensions with ExtendJ!

Gradle plugin: JastAddGradle

The extension is specified in a module specification, which is used to compile together with some base modules from ExtendJ.

# Extensions: Getting Started

A small template project to get started with building an extension:

Compiler Template Project: https://bitbucket.org/extendj/compiler-template

# Demo

<Demo> Compiler Template.

https://bitbucket.org/extendj/compiler-template

extendj.org

# Thank You

Thanks for listening!

Learn more:

**extendj.org**
**jastadd.org**

extendj.org