

ExtendJ

The Extensible Java Compiler

Jesper Öqvist, PhD Student
Lund University

About Me

- PhD student since 2013
- Google Intern (2015 & 2016)
- ExtendJ Maintainer (668 commits of 1158 total)
- Current main project: concurrent attributes and parallel compilation.

Other stuff:

- Java2C (Real-time programming course)

ExtendJ Motivation

- Creating semantic tools for Java is hard.
- However, semantic tools are very useful:
 - Static analysis, refactoring, metrics, testing, language extensions, semantic diff, dead code removal ...
- With an extensible compiler it's much easier to make such tools.

ExtendJ Background

An extensible Java compiler using JastAdd attribute grammars.

- **Declarative** side-effect free evaluation removes lots of headache
- **Modular** build with only the parts you need
- **Extensible** add your own module without touching old code
- **Demand evaluation** computes only attributes that are actually used

Previously known as “JastAddJ”, developed at Lund University by Torbjörn Ekman, et al.

ExtendJ Design

The basic design is very similar to the assignments for the course EDAN65 Compilers.

Extensions I have made

Java 7

- Master's Thesis project

Multiplicities

- Language extension for Java
- [Multitudes of Objects: First Implementation and Case Study for Java](#)

Regression Test Selector (AutoRTS)

- Unit testing tool

The Multiplicities Extension

Makes it easier to code with changing multiplicities.

Regular Java:

```
Person employee;  
employee = new Person("Bob");  
  
employee.fire();
```

The Multiplicities Extension

Makes it easier to code with changing multiplicities.

Java + Multiplicities:

```
@any Person employee;  
employee = new Person("Bob");  
employee += new Person("Eva");  
employee.fire(); // Call fire() on Bob and Eve.
```


Multiplicity Example 2

Multiplicity parameter:

```
void fire(@any Person people) {  
    for (Person person : people) {  
        person.fire();  
    }  
}
```

```
fire(new Person("Dave"));  
@any Person people = ...;  
fire(people);
```

Multiplicities Demo

Sacrificing a lamb to the demo gods...

Multiplicities Implementation

Deep integration with ExtendJ.

Error messages are reported for multiplicity errors, and bytecode is generated directly.

Implementation size is small considering the integration level:

- Static type checking - 1372 lines
- Bytecode generation - 1821 lines

Requirements for extensibility?

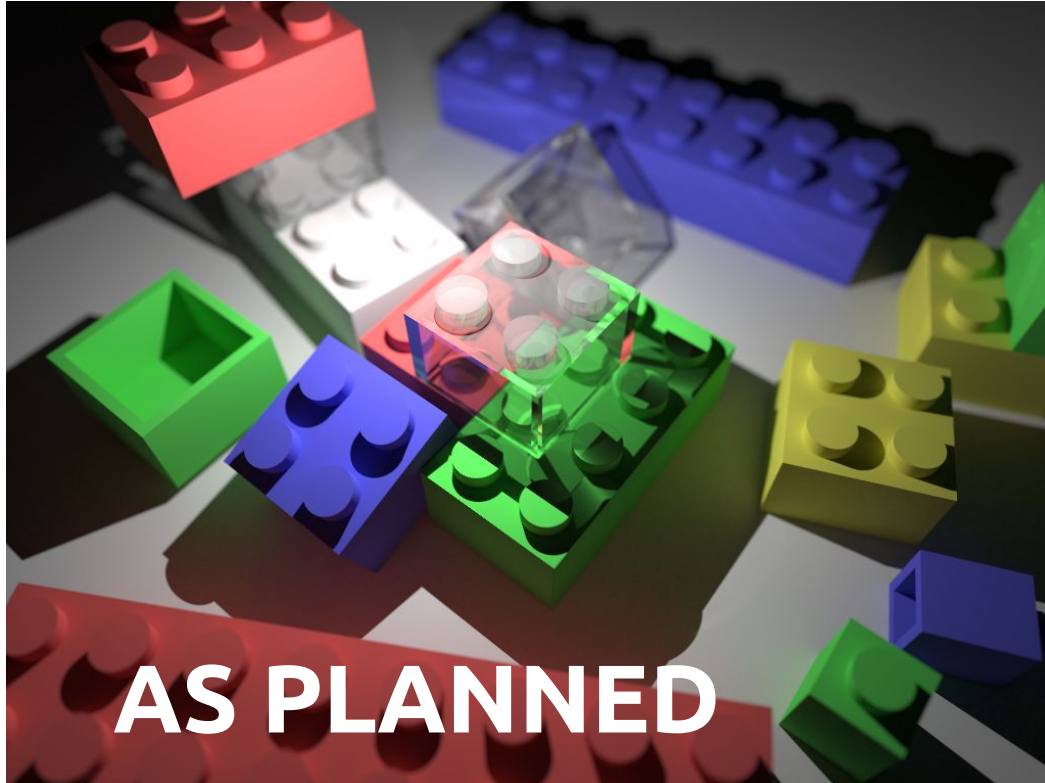
In general:

- Reusability
- Stability

ExtendJ:

- Side-effect free API
- Sensible abstractions & API (this is hard)

Extensible System



Extensible System



Note about Technical Debt

From Wikipedia:

“a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution”

Note about Technical Debt

From Wikipedia:

“a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution”

Naive idea: just do it right from the start!

Note about Technical Debt

From Wikipedia:

“a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution”

Naive idea: just do it right from the start!

Pitfall: overdesign for imagined future use - leads to Technical Debt anyway.

Note about Technical Debt

From Wikipedia:

“a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution”

Naive idea: just do it right from the start!

Pitfall: overdesign for imagined future use - leads to Technical Debt anyway.

Wisdom: make it simple first, then refactor when needed.

ExtendJ Problems

ExtendJ can't satisfy all needs perfectly:

- **Quality** (bug free, no side effects, no technical debt)
- **Stability** (painless to upgrade)
- **Reusability** (deep integration)

Quality has been the no. 1 priority for a long time.

Reusability and stability will evolve from having more supported extensions.

Quality

High quality is **few bugs, good API**. Requires changing the internals of ExtendJ.

Threats to stability:

- Existing attributes change - renamed, repurposed, refactored, rewritten, replaced, changed behaviour, ...

Most non-bugfix changes likely break extensions.

Even pure bug fixes can break extensions.

Reusability

Ability to **reuse and alter** existing attributes.

Reusability almost for free with JastAdd:

- All attributes public
- The **refine** construct can be used to change nearly anything

Threats to stability:

- **Extensions depend on changing internals.**
- **No possibility for internal-only API.**

Stability

Obviously, stability is nice to have.

Right now, extensions should depend on specific revisions of ExtendJ.

Update only if you have a good test suite that can ensure no breakage.

Importance of Tests

Regression tests are very important for **Quality**.

- It's not possible to have full test coverage
- Writing feature tests is important, but won't cover everything

ExtendJ testing:

- 1442 regression tests
- 10 large benchmark applications
- Very large private test set

Java is not a simple language

// Will this compile?

```
public class Test {  
    class Inner {}  
}
```

```
class Outer extends Test.Inner {  
}
```


Java is not a simple language

// No enclosing instance for Test.Inner when accessed in super()

```
public class Test {  
    class Inner {}  
}
```

```
class Outer extends Test.Inner {  
}
```

Java is not a simple language

// Fixed!

```
public class Test {  
    class Inner {}  
}
```

```
class Outer extends Test.Inner {  
    Outer() {  
        new Test().super();  
    }  
}
```

Java is not a simple language

```
public class Test<T extends Integer> {  
    class C extends Test {}  
    C[] array = new C[10]; // Will this compile?  
}
```

Java is not a simple language

```
public class Test<T extends Integer> {  
    class C extends Test {}  
    C[] array = new C[10]; // Error: non reifiable element type.  
}
```

Java is not a simple language

```
public class Test {  
    I<?>[] array = new I<?>[10]; // Will this compile?  
}
```

```
interface I<T extends Number> {  
}
```

Java is not a simple language

```
public class Test {  
    I<?>[] array = new I<?>[10]; // OK! Wildcard type is reifiable.  
}
```

```
interface I<T extends Number> {  
}
```

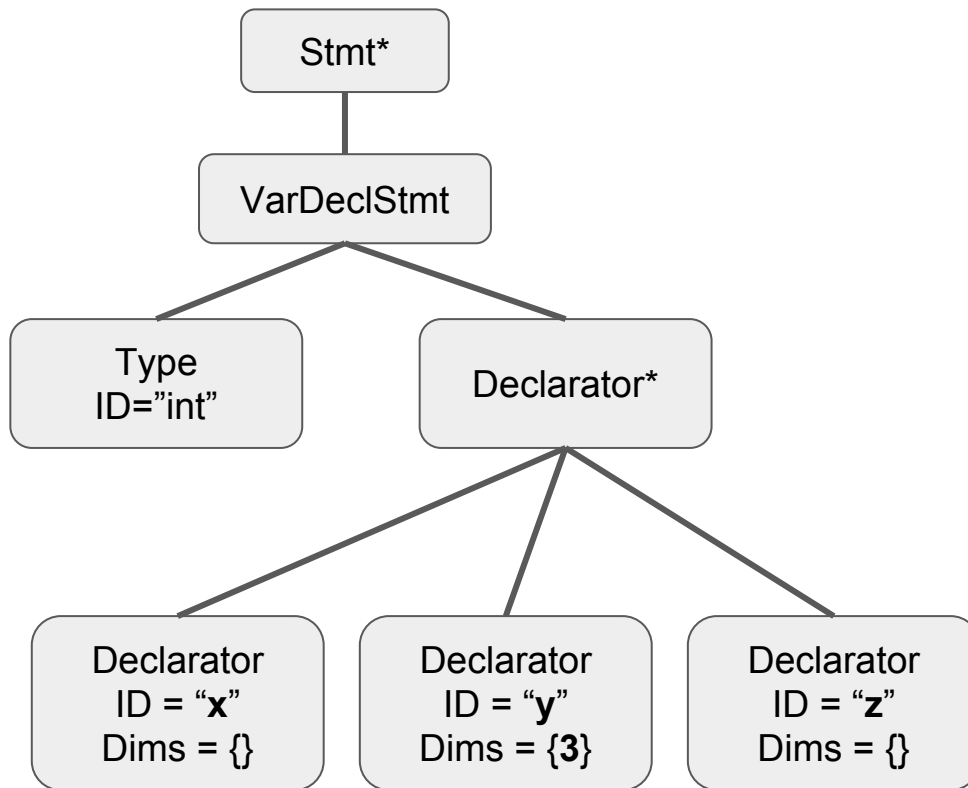
Side-effect freedom with JstAdd

Some transformations have to be done in the AST.

Instead of imperative tree transforms, we construct transformed trees inside the original tree using NTAs!

Parsed Tree

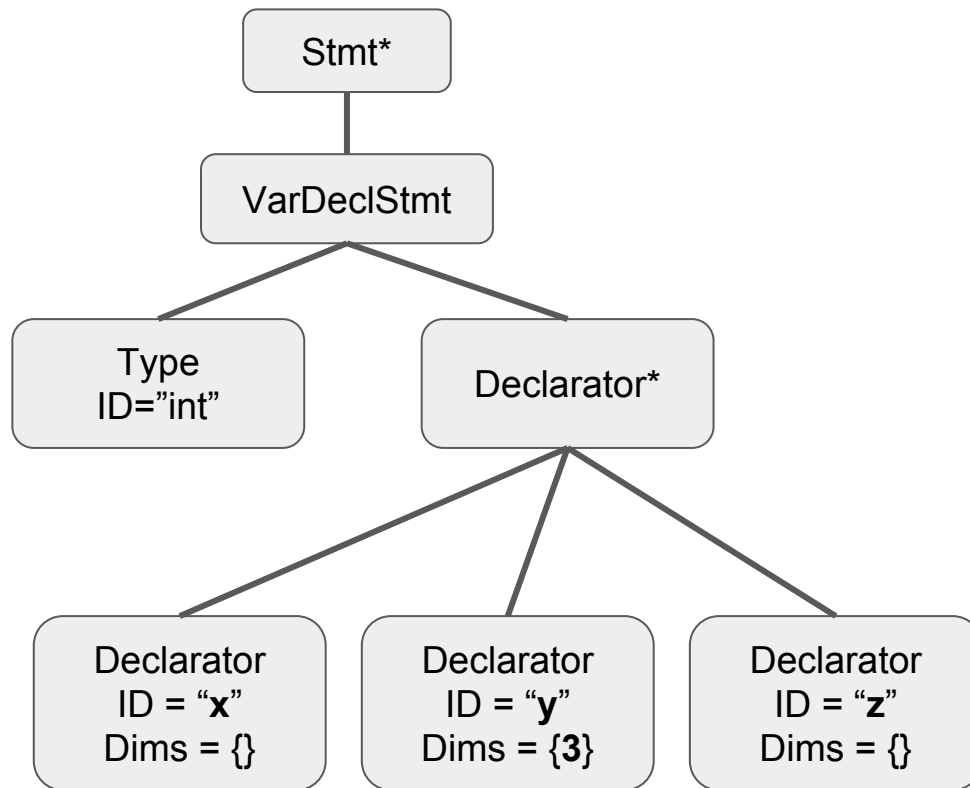
```
int x, y[3], z;
```



Parsed Tree

```
int x, y[3], z;
```

Need to combine
Type, Dims, ID

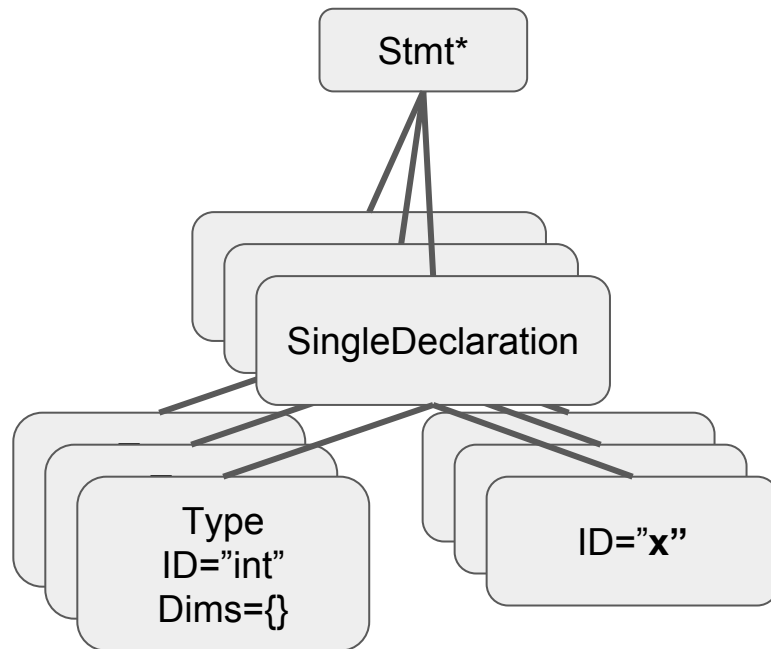


Rewritten Tree

```
int x, y[3], z;
```

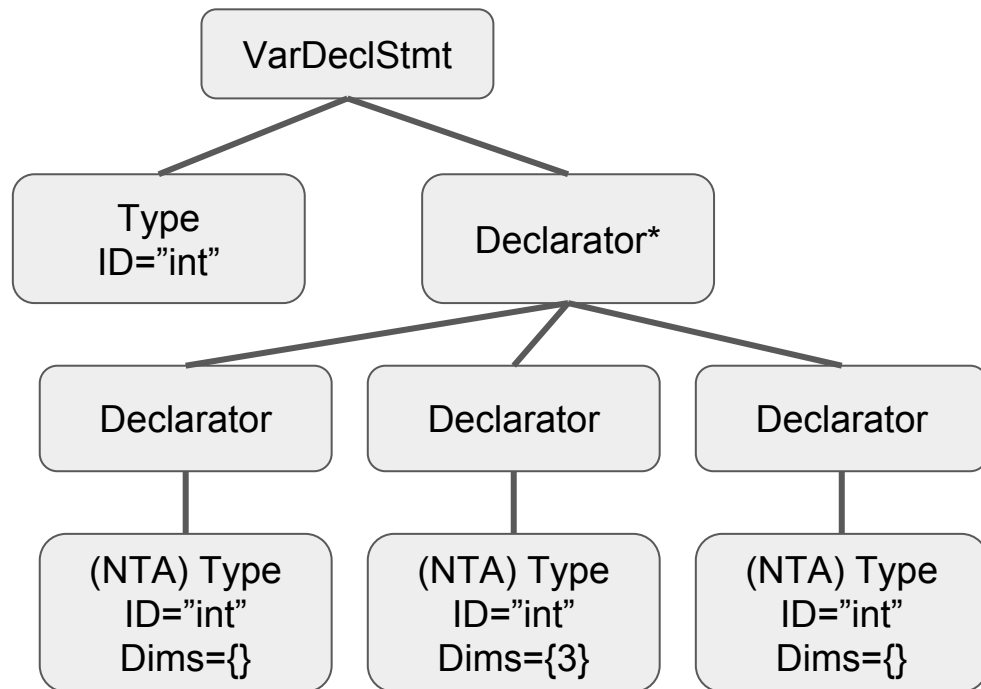
Equivalent:

```
int x;  
int y[3];  
int z;
```



Side-effect free transform with NTAs

```
int x, y[3], z;
```

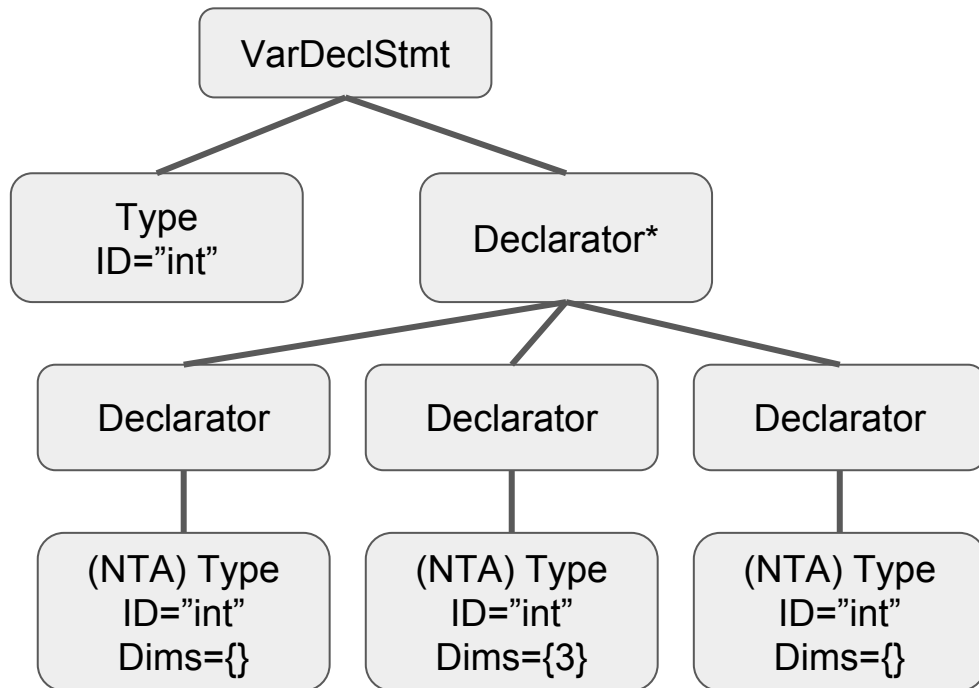


Side-effect free transform with NTAs

```
int x, y[3], z;
```

Type NTAs are easy to access from a declarator!

Type NTAs include the correct dimension without rewriting the tree!



Advantages of Side-Effect Freeness

- Execution order independence
- Non order-dependent API
- Concurrent evaluation possibilities

Modularization Problem

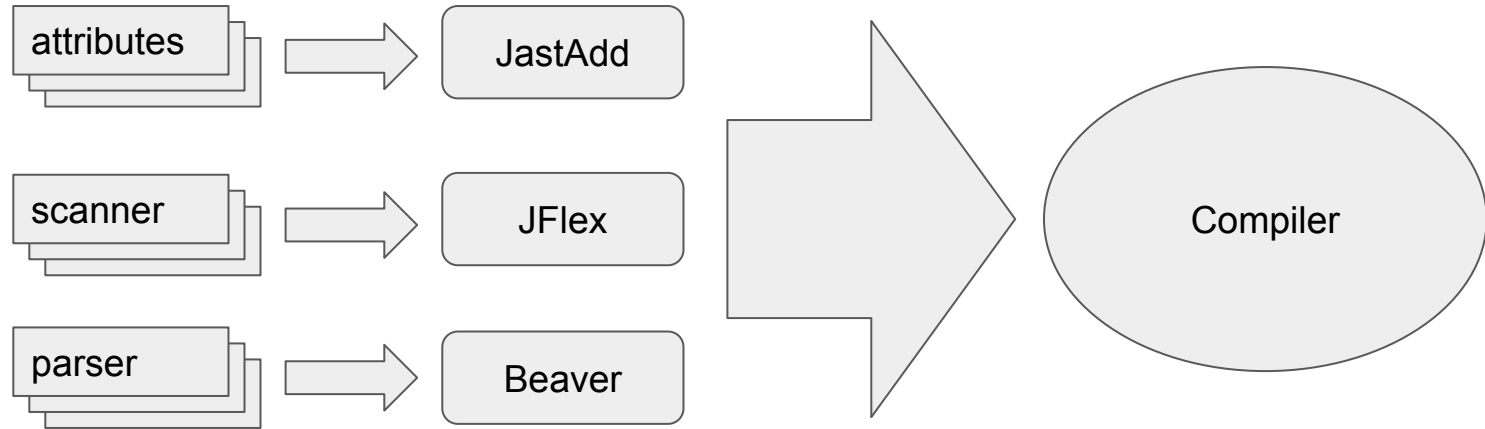
Extensions have inter-dependencies:

- Include dependencies
- Exclude conflicting objects

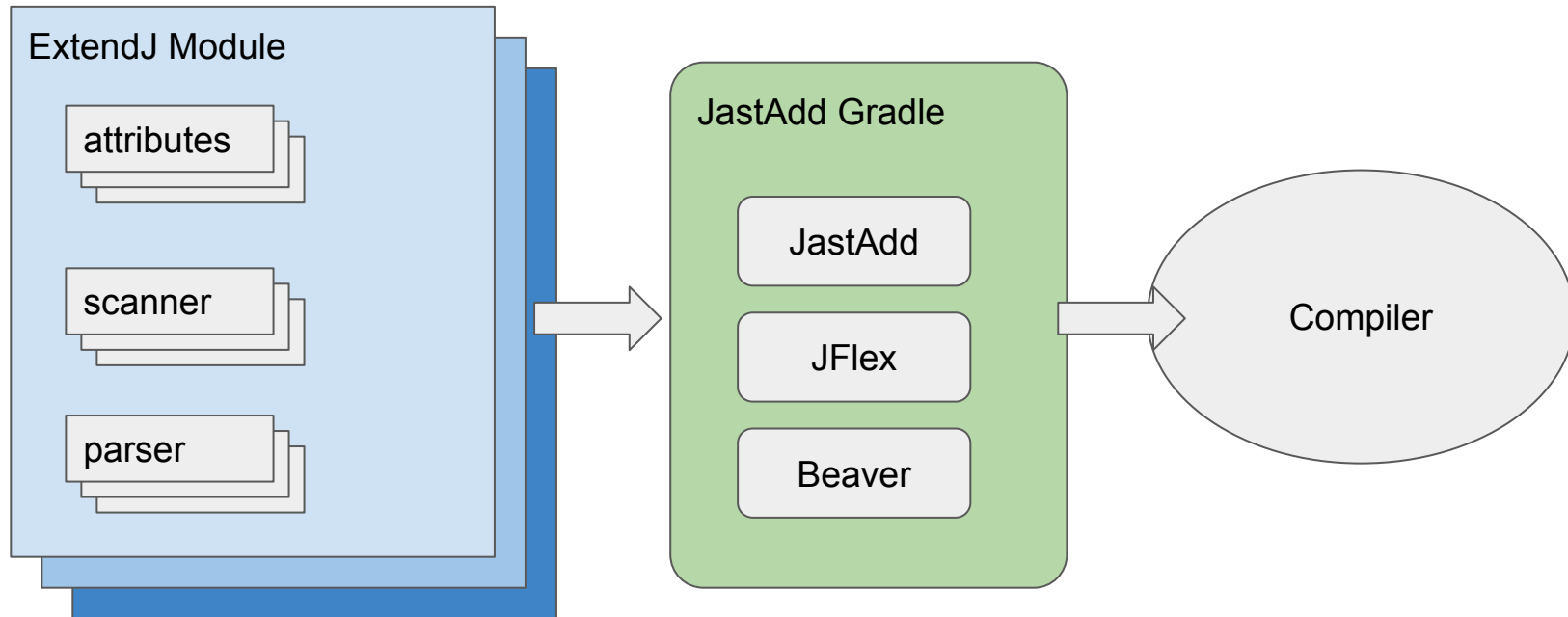
How can this be handled?

- Package everything into modules and declare dependencies.
- Let module system handle the build.

JastAdd Metacompilation



ExtendJ Modules



Module File (jastadd_modules)

```
module("java8 backend") {  
    moduleVariant "backend"  
  
    imports "java8 frontend"  
    imports "java7 backend"  
  
    jastadd {  
        include "backend/*.jadd"  
        include "backend/*.jrag"  
    }  
}
```

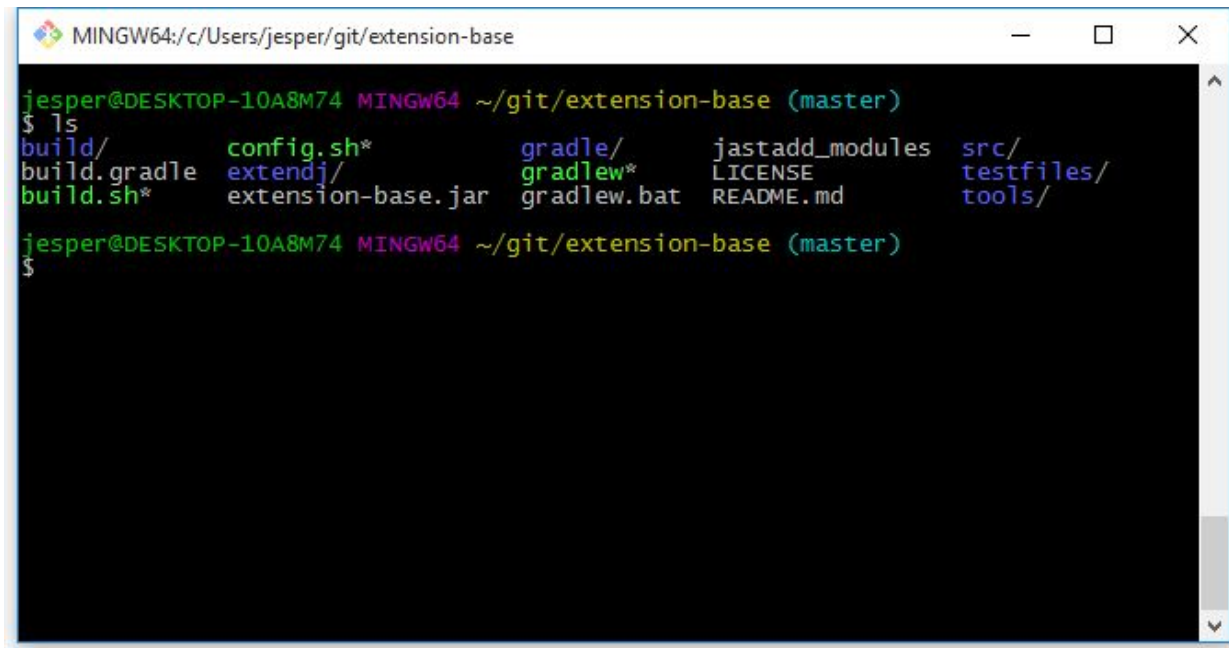
JastAddGradle

A Gradle plugin to make it easy to build JastAdd Projects

```
> gradle jar
```

Building an Extension - Demo

Extension Base project: <https://bitbucket.org/extendj/extension-base>



```
MINGW64:/c:/Users/jesper/git/extension-base

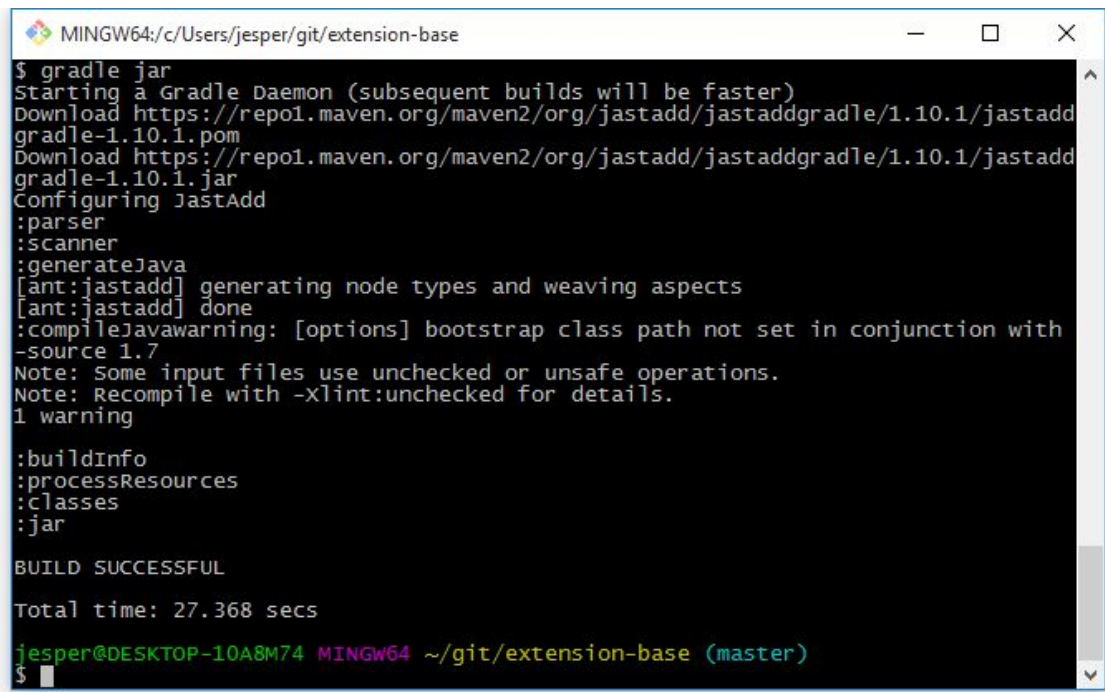
jesper@DESKTOP-10A8M74 MINGW64 ~/git/extension-base (master)
$ ls
build/          config.sh*      gradle/         jastadd_modules  src/
build.gradle    extendj/        gradlew*        LICENSE          testfiles/
build.sh*       extension-base.jar  gradlew.bat    README.md        tools/

jesper@DESKTOP-10A8M74 MINGW64 ~/git/extension-base (master)
$
```

Backup picture :)

Building an Extension - Demo

Extension Base project: <https://bitbucket.org/extendj/extension-base>



```
MINGW64:/c:/Users/jesper/git/extension-base
$ gradle jar
Starting a Gradle Daemon (subsequent builds will be faster)
Download https://repo1.maven.org/maven2/org/jastadd/jastaddgradle/1.10.1/jastadd
gradle-1.10.1.pom
Download https://repo1.maven.org/maven2/org/jastadd/jastaddgradle/1.10.1/jastadd
gradle-1.10.1.jar
Configuring JastAdd
:parser
:scanner
:generateJava
[ant:jastadd] generating node types and weaving aspects
[ant:jastadd] done
:compileJavawarning: [options] bootstrap class path not set in conjunction with
-source 1.7
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
1 warning

:buildInfo
:processResources
:classes
:jar

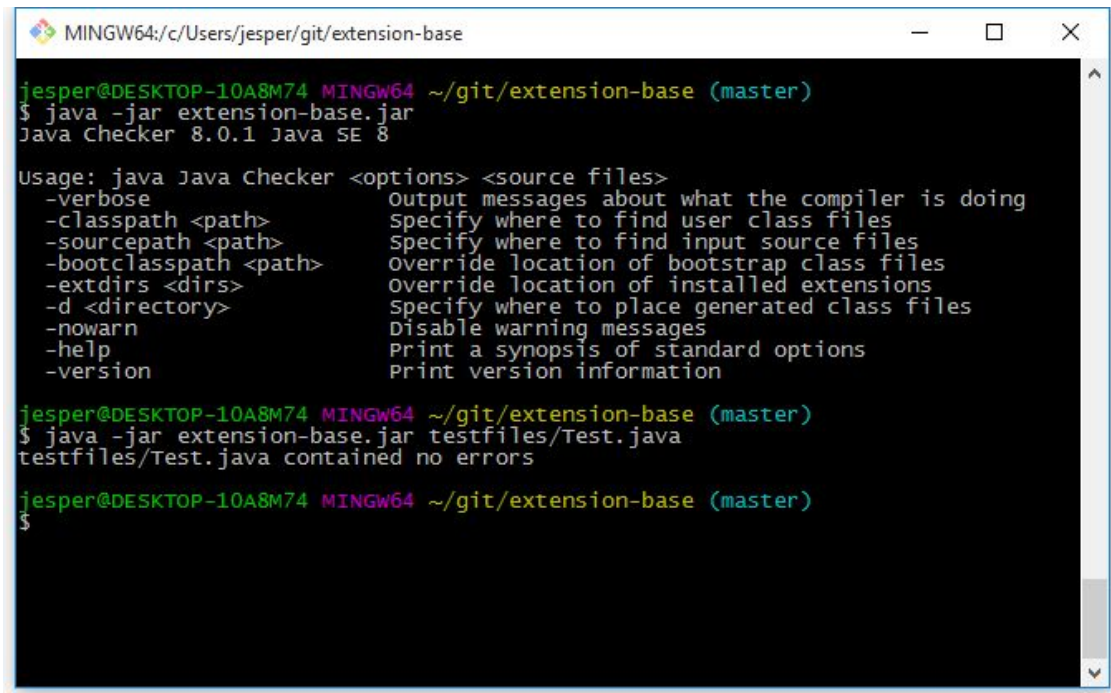
BUILD SUCCESSFUL

Total time: 27.368 secs

jesper@DESKTOP-10A8M74 MINGW64 ~/git/extension-base (master)
$
```

Building an Extension - Demo

Extension Base project: <https://bitbucket.org/extendj/extension-base>



```
MINGW64: c:/Users/jesper/git/extension-base
jesper@DESKTOP-10A8M74 MINGW64 ~/git/extension-base (master)
$ java -jar extension-base.jar
Java Checker 8.0.1 Java SE 8

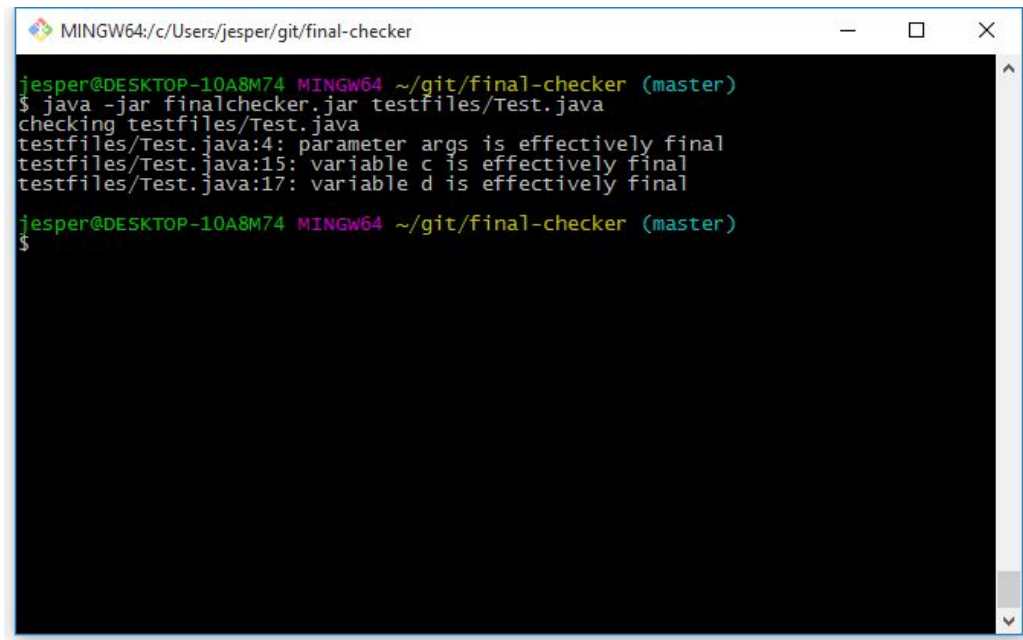
Usage: java Java Checker <options> <source files>
  -verbose                Output messages about what the compiler is doing
  -classpath <path>       Specify where to find user class files
  -sourcepath <path>      Specify where to find input source files
  -bootclasspath <path>   Override location of bootstrap class files
  -extdirs <dirs>         Override location of installed extensions
  -d <directory>          Specify where to place generated class files
  -nowarn                 Disable warning messages
  -help                   Print a synopsis of standard options
  -version                Print version information

jesper@DESKTOP-10A8M74 MINGW64 ~/git/extension-base (master)
$ java -jar extension-base.jar testfiles/Test.java
testfiles/Test.java contained no errors

jesper@DESKTOP-10A8M74 MINGW64 ~/git/extension-base (master)
$
```

Final Checker Demo

Final checker project: <https://bitbucket.org/extendj/final-checker>



```
MINGW64; c:/Users/jesper/git/final-checker

jesper@DESKTOP-10A8M74 MINGW64 ~/git/final-checker (master)
$ java -jar finalchecker.jar testfiles/Test.java
checking testfiles/Test.java
testfiles/Test.java:4: parameter args is effectively final
testfiles/Test.java:15: variable c is effectively final
testfiles/Test.java:17: variable d is effectively final

jesper@DESKTOP-10A8M74 MINGW64 ~/git/final-checker (master)
$
```

API Overview

`Program ::= CompilationUnit*; // Program is a list of Java source files.`

`CompilationUnit ::= TypeDecl*; // ClassDecl, EnumDecl, InterfaceDecl.`

`TypeDecl ::= BodyDecl*; // MethodDecl, ConstructorDecl, FieldDecl.`

`abstract Stmt;`

`IfStmt : Stmt ::= Condition:Expr Then:Block [Else:Block];`

<http://jastadd.org/releases/jastaddj/7.1/doc/index.html>

API Overview

Attributes are used to look up semantic information. Some examples:

```
decl().type(); // Get the type of a declaration.
```

```
lookupType("java.util", "Collection"); // Lookup specific type.
```

```
lookupMethod(access); // Find method for access.
```

```
decl().hostType();
```


Extra Slides

Final Checker Example <https://bitbucket.org/extendj/final-checker>

A simple extension to check for variables that could safely be declared 'final'.

Final means that a variable can only be assigned once.

```
public class Test {  
  
    // 'args' is never assigned, so it is effectively final.  
    public static void main(String[] args) {  
        int a = args.length;  
        for (int i = 0; i < a; ++i) {  
            a = m(i, a);  
        }  
    }  
    ...  
}
```

Final Checker Example

The final checker extension is a good starting point to make your own extension.

```
static int m(final int i, int a) {  
    a = i;  
    int c;  
    c = a; // only assigned once  
    int d = a + i;  
    return d;  
}
```

```
11:21:05 jesper@yolo ~/git/final-checker $ java -jar finalchecker.jar Test.java  
checking Test.java  
Test.java:4: parameter args is effectively final  
Test.java:15: variable c is effectively final  
Test.java:17: variable d is effectively final  
11:21:25 jesper@yolo ~/git/final-checker $ █
```

Final Checker Implementation

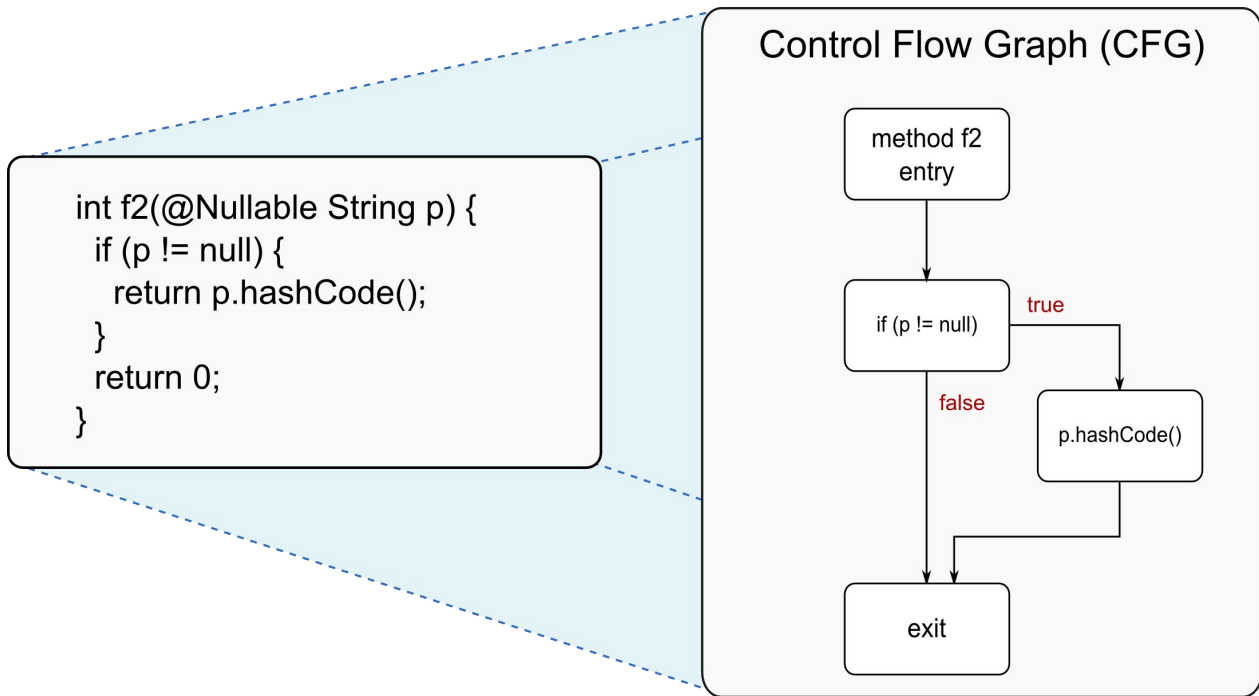
```
coll java.util.List<Variable> CompilationUnit.effectivelyFinalVariables()  
  [new LinkedList<Variable>()]  
  with add  
  root CompilationUnit;
```

```
VariableDecl contributes declaration()  
  when !declaration().isFinal() && declaration().isEffectivelyFinal()  
  to CompilationUnit.effectivelyFinalVariables()  
  for compilationUnit();
```

```
ParameterDeclaration contributes this  
  when !isFinal() && isEffectivelyFinal()  
  to CompilationUnit.effectivelyFinalVariables()  
  for compilationUnit();
```

Control Flow Graphs (CFGs)

A CFG, captures the possible control flow in a program:



SimpleCFG

Control flow graphs are useful for static analysis.

We have an intraflow module already for ExtendJ, but it generated dense graphs - collected control flow between every expression.

SimpleCFG is a new CFG module where you instead select which statements or expressions should appear in the graph.


The idea is to insert nonterminal attributes where needed to represent CFG nodes.

Nullable Dereference

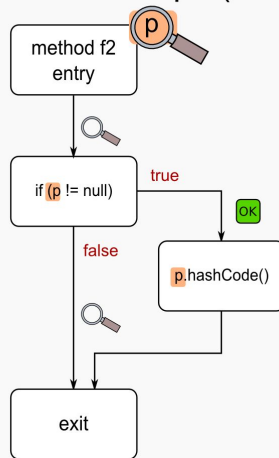
An analyzer testing for dereferences of parameters declared as @Nullable

Uses CFG module to do control flow sensitive analysis.

```
int f2(@Nullable String p) {  
    if (p != null) {  
        return p.hashCode();  
    }  
    return 0;  
}
```



Control Flow Graph (CFG)



Control flow dependent analysis:

- search in the CFG after dereferences of **p**
- skip branches guarded by null tests

Benefits of implementing analysis in ExtendJ

- Pick only the parts of the compiler that are needed
- I stripped out most type lookup for the Nullable Dereference analyzer
- Code generation not necessary
- Analysis is very fast!

Extension Architecture

