Polyglot Tutorial

PLDI 2014

Steve Chong, Harvard University
Chin Isradisaikul, Cornell University
Andrew Myers, Cornell University
Nate Nystrom, University of Lugano

Overview

Today:

- Polyglot architecture
- Run through of an example compiler extension
- Accrue

Download

You'll need Java 7 or later

Eclipse 3.4.2 or later is useful but not required.

Download the following zip file:

```
http://www.cs.cornell.edu/Projects/polyglot/pldi14/tutorial/polyglot-tutorial.zip
```

Unzip and import the included projects into Eclipse

Language extension

Language designers often create extensions to existing languages

- e.g., C++, PolyJ, Pizza, AspectJ, Jif, ESCJava, X10, Java5, Java6, Java7, Java8

Want to reuse existing compiler infrastructure as much as possible

Polyglot is a framework for writing compiler extensions for Java

Requirements

Language extension

- Modify both syntax and semantics of the base language
- Changes are not necessarily backward compatible

Goals:

- Easy to build and maintain extensions
- Extensibility should be **modular** and **scalable**
  - No changing base compiler, no code duplication
- Compilers for language extensions should be **open** to further extension

Polyglot base compiler

Base compiler is a complete Java 1.4 front end

Can reuse and extend through inheritance

53K lines of Java

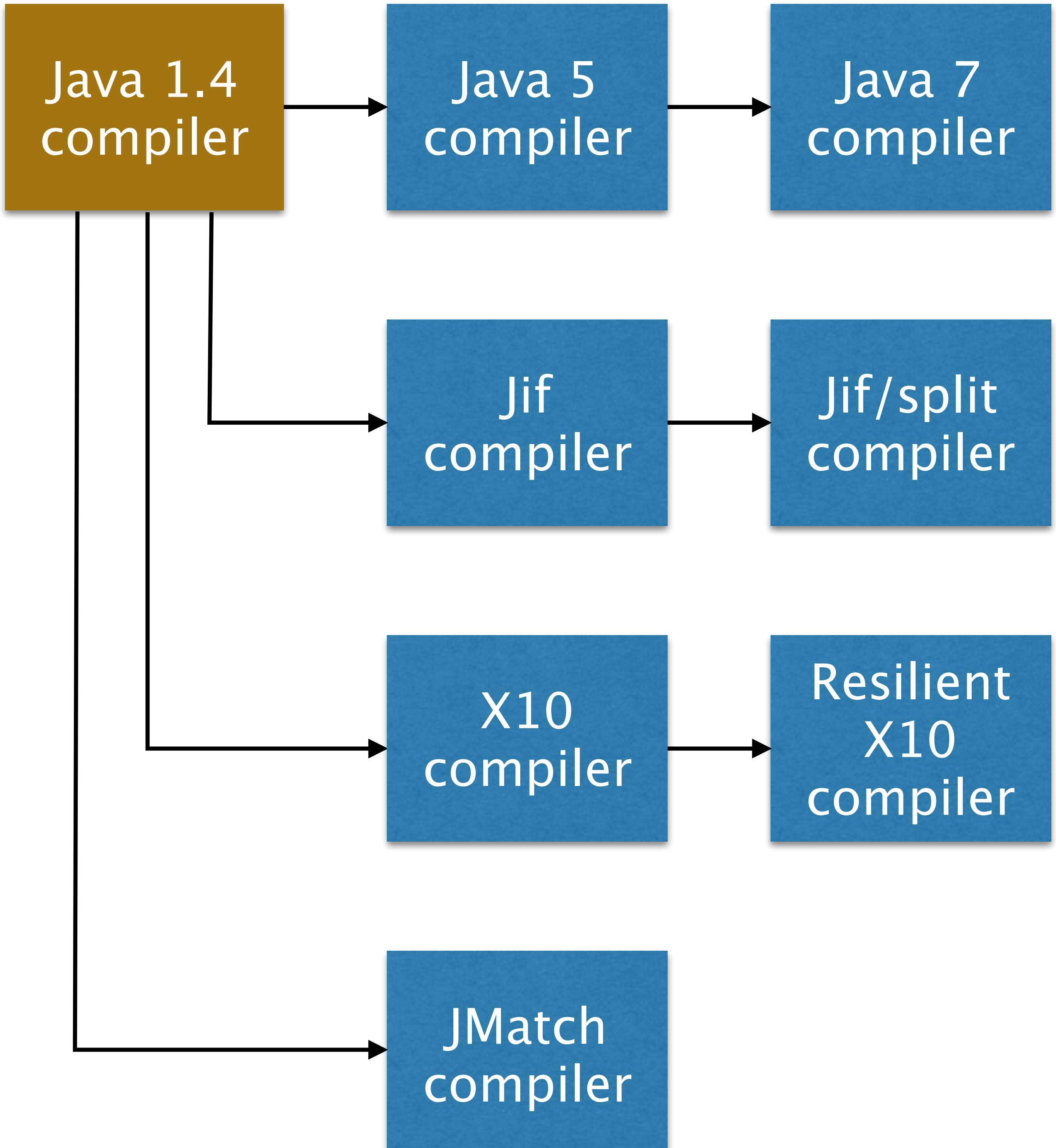
- Parsing, name resolution, inner class support, type checking, exception checking, uninitialized variable analysis, unreachable code analysis, …

And … 25K more lines of Java

- Java5 and Java7 extensions

Polyglot family tree

XXX



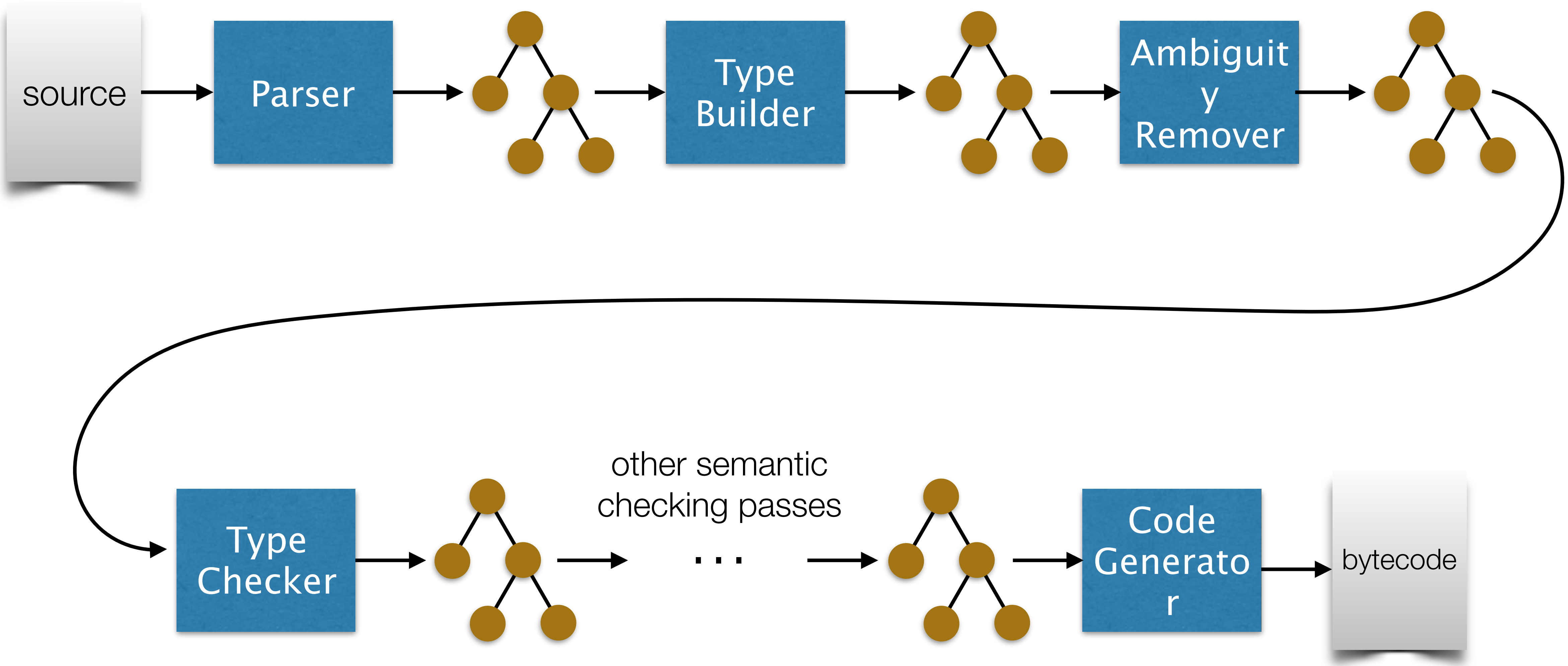
Architecture

Base compiler compiles Java 1.4 source to Java 1.4 source

Extensions implement new language features by …

- extending the parser
- adding new abstract syntax
- adding new types
- adding new compiler passes
- overriding existing passes
- translating new features into Java 1.4

Polyglot pass architecture



Overview of the base compiler

Let's do a quick run through the base compiler code

- parsing
- ASTs
- visitors
- types

But first, download:

http://www.cs.cornell.edu/Projects/polyglot/pldi14/tutorial/polyglot-tutorial.zip

Parsing (polyglot.parse)

The parser reads the source text and creates ASTs

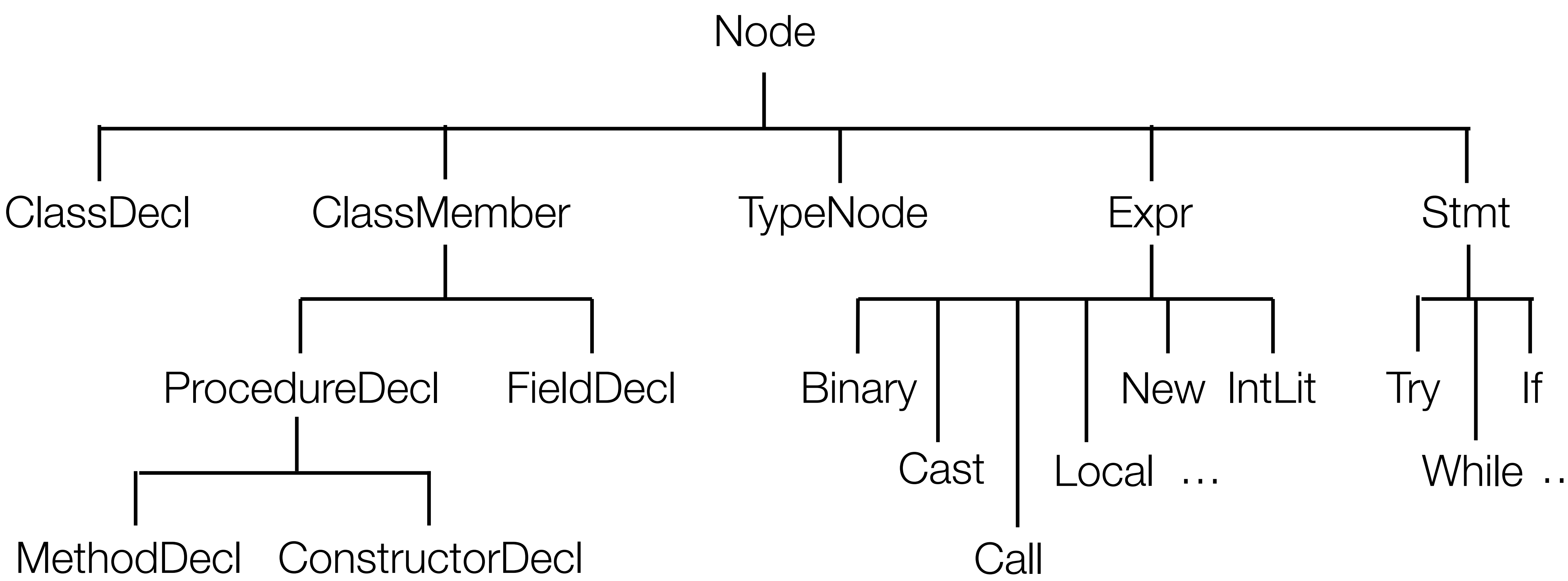
Base compiler Java parser implemented using LALR(1) JavaCUP parser generator

Extensions can use the Polyglot Parser Generator (PPG) to extend the parser by adding, modifying, or removing rules

ASTs (polyglot.ast)

- AST nodes created by factory methods in the NodeFactory
- AST nodes are persistent data structures
    - Methods to modify a node, return a modified copy of the node
    - We use clone() to ensure any fields added by subclasses are preserved

ASTs (polyglot.ast)



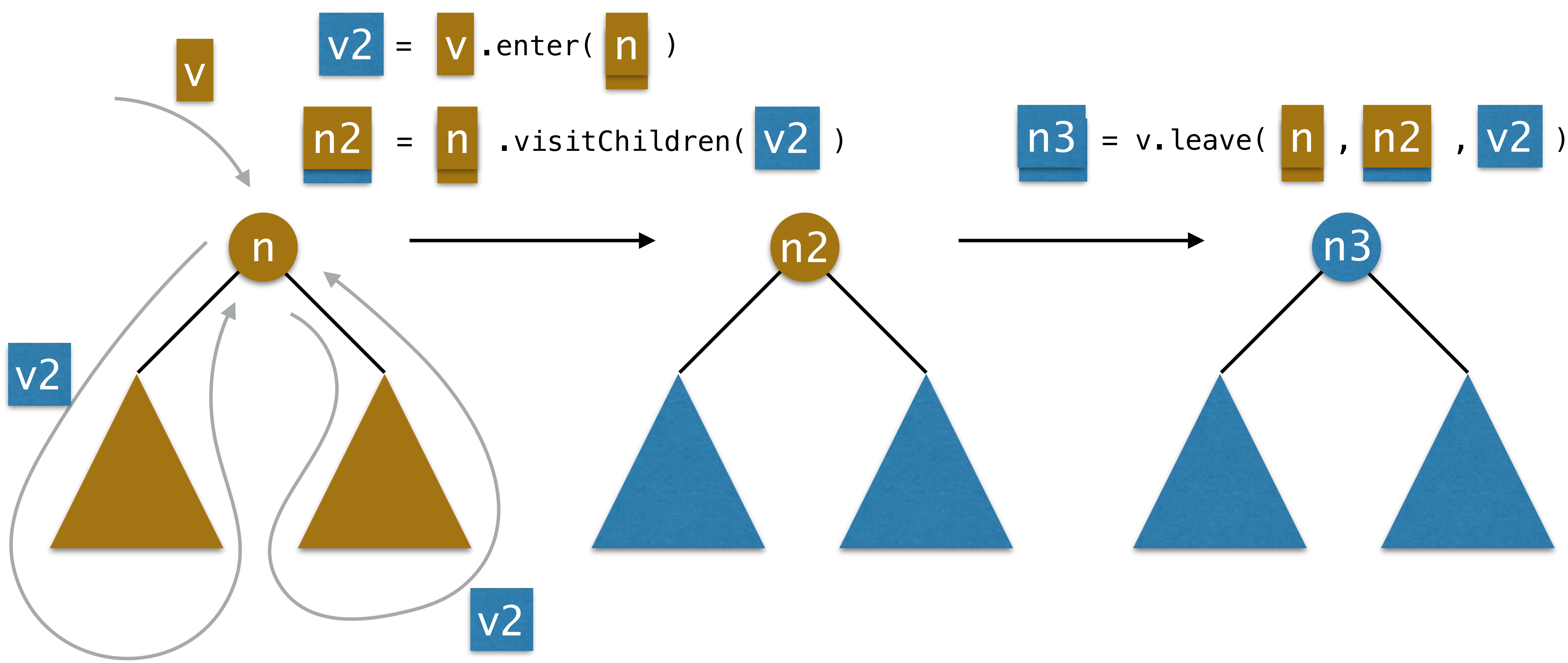
Visitors (polyglot.visit)

Visitors traverse the AST, executing operations at each node in the tree, returning a new node to reconstruct the tree if needed

Visitors follow the following protocol at each Node:

- override – if returns a Node, return that Node; otherwise, continue
- enter – returns a new Visitor used to visit children of this node
- (recursively visitChildren)
- leave – returns a new Node

NodeVisitor



TypeObjects and TypeSystem

TypeSystem implements methods for semantic checking

- `isSubtype(Type, Type)`
- `isCastValid(Type, Type)`
- `methodCallValid(MethodInstance, String, List<Type>)`
- …

Types (and other type-level entities) are represented as TypeObjects:

Type, PrimitiveType, ArrayType, ClassType, …

MethodInstance, FieldInstance, ConstructorInstance, …

TypeSystem also serves as a factory for TypeObjects

Extending Java with constant arrays

Covariant arrays

Java array types are **covariant**

$$Bird <: Animal \Rightarrow Bird[] <: Animal[]$$

```
class C {
    void foo() {
       Bird[] birds = { chicken, duck };
       Animal[] animals = birds;

    }
}
```

Covariant arrays

Java array types are **covariant**

$$Bird <: Animal \quad \Rightarrow \quad Bird[] <: Animal[]$$

```
class C {
    void foo() {
        Bird[] birds = { chicken, duck };
        Animal[] animals = birds;
        animals[0] = cow;  // what happens here?
    }
}
```

Covariant arrays

Java array types are **covariant**

$$Bird <: Animal \Rightarrow Bird[] <: Animal[]$$

```
class C {
    void foo() {
        Bird[] birds = { chicken, duck };
        Animal[] animals = birds;
        animals[0] = cow;  // throws ArrayStoreException
    }
}
```

Constant arrays

Let’s write a Polyglot extension to fix this problem

- Make traditional non-const arrays **invariant**
  - `Object[]`
- Introduce **covariant const** arrays
  - `Object const[]`

Make traditional arrays **invariant**

```
class C {
    void foo() {
        Bird[] birds = { chicken, duck };
        Animal[] animals = birds; // compile-time error
    }
}
```

Introduce **covariant const** arrays

```
class C {
    void foo() {
        Bird const[] birds = { chicken, duck };
        Animal const[] animals = birds;
        animals[0] = cow; // compile-time error
    }
}
```

Implementing the extension

1. Extend the parser with const array types
2. Add a const array AST node
3. Add a const array type
4. Prevent assignment to const arrays
5. Implement subtyping for const arrays
6. Change the subtyping rules for Java arrays
7. Translate const arrays to Java arrays

Testing

First let's write some tests …

Polyglot provides a testing harness (pth)

- give test inputs and specify tests that should pass, fail

Implementing the extension

1. Extend the parser with const array types
2. Add a const array AST node
3. Add a const array type
4. Prevent assignment to const arrays
5. Implement subtyping for const arrays
6. Change the subtyping rules for Java arrays
7. Translate const arrays to Java arrays

Parsing

The parser reads the source text and creates the AST

NodeFactory creates AST nodes

Parsing

Original rule for array types in java12.cup:

```
array_type ::=
    // array of primitive types such as int[] and char[][]
    primitive_type:a dims:b {:
        RESULT = parser.array(a, b.intValue());
    :}
    // array of object types such as String[] and Object[][]
  | name:a dims:b {:
        RESULT = parser.array(a.toType(), b.intValue());
    :}
;
```

Parsing

Extended rule in carray.ppg:

```
extend array_type ::= // RESULT of array_type is a TypeNode
    primitive_type:a CONST dims:b {:
        RESULT = parser.constArray(a, b);
    :}
  | name:a CONST dims:b {:
        RESULT = parser.constArray(a.toType(), b);
    :}
;
```

Parsing

Helper function for creating const array ASTs:

```
/**
 * Return a TypeNode representing a {@code dims}-dimensional constant array
 * of ultimate base {@code n}.
 */
public TypeNode constArray(TypeNode n, int dims) throws Exception {
    if (dims > 0)
        return nf.ConstArrayTypeNode(n.position(), constArray(n, dims - 1));
    return n;
}
```

Parsing exercise

Implement carray types by extending the rules for
array_type (as above) and *cast_expression*

Syntax for a const array type:
Type **const** [ ]

Examples:
int **const** [ ]
String **const** [ ] [ ] [ ]
java.lang.Number **const** [ ] [ ]

Implementing the extension

1. Extend the parser with const array types
2. Add a const array AST node
3. Add a const array type
4. Prevent assignment to const arrays
5. Implement subtyping for const arrays
6. Change the subtyping rules for Java arrays
7. Translate const arrays to Java arrays

ASTs

1. Create an interface and class for the new AST node
2. Extend NodeFactory to create an instance of the class
3. Extend ExtFactory to provide a hook for future language extensions to override functionality

AST interface

```
package carray.ast;

import polyglot.ast.ArrayTypeNode;

/**
 * A {@code ConstArrayTypeNode} is a type node for a non-canonical
 * const array type.
 */
public interface ConstArrayTypeNode extends ArrayTypeNode {
}
```

AST class

```
package carray.ast;

import polyglot.ast.ArrayTypeNode_c;

/**
 * A {@code ConstArrayTypeNode} is a type node for a non-canonical
 * const array type.
 */
public class ConstArrayTypeNode_c extends ArrayTypeNode_c implements
        ConstArrayTypeNode {
    super(pos, base);
}
```

Exercise

Implement ConstArrayTypeNode_c.toString()

NodeFactory

AST nodes are created using a NodeFactory

NodeFactory attaches zero or more **extension object** to the new nodes

Extension objects provide a hook for language extensions to add functionality to the node

Extension objects are created with ExtFactory

Extending NodeFactory

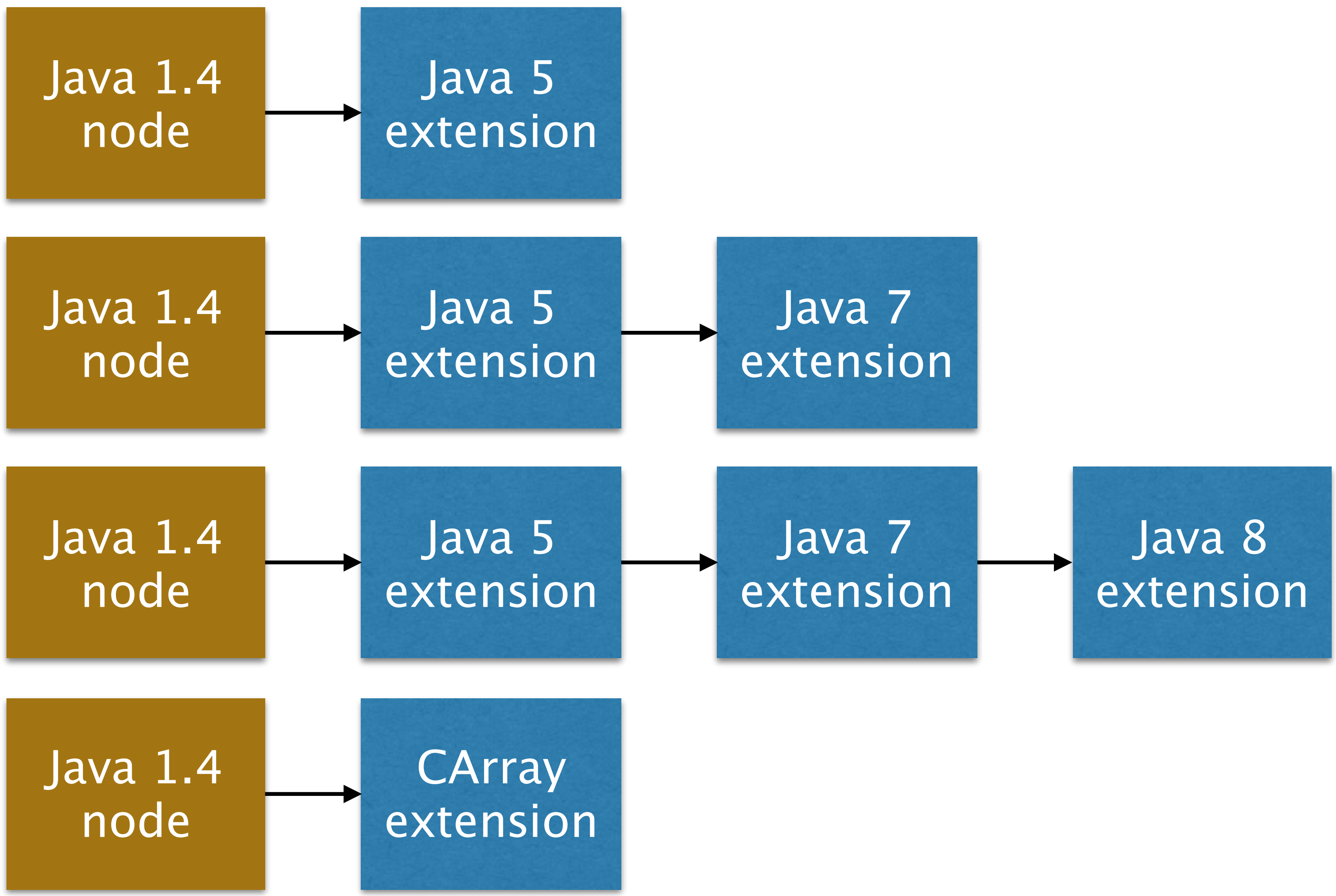
```
@Override
public ConstArrayTypeNode ConstArrayTypeNode(Position pos, TypeNode base) {
    ConstArrayTypeNode_c n = new ConstArrayTypeNode_c(pos, base);

    return n;
}
```

Extending NodeFactory

```
@Override
public ConstArrayTypeNode ConstArrayTypeNode(Position pos, TypeNode base) {
    ConstArrayTypeNode_c n = new ConstArrayTypeNode_c(pos, base);
    n = ext(n, extFactory().extConstArrayTypeNode());
    return n;
}
```

Extension chaining

An AST node can have zero or more **extension objects**



Extension objects

Used to add/modify operations on AST node types

Should only create a new extension object class if needed

By default: the extension object class for an AST is the same as for the superclass of the AST class.

Example:

- In the Java compiler, ArrayTypeNode is a subclass of TypeNode
- In CArray, the extension object for an ArrayTypeNode can be from the same class as extension objects for TypeNode

Extension chaining

For new kinds of AST nodes, extension objects forward operations to the base language

If the next extension knows about CArray, extConstArrayTypeNode is invoked

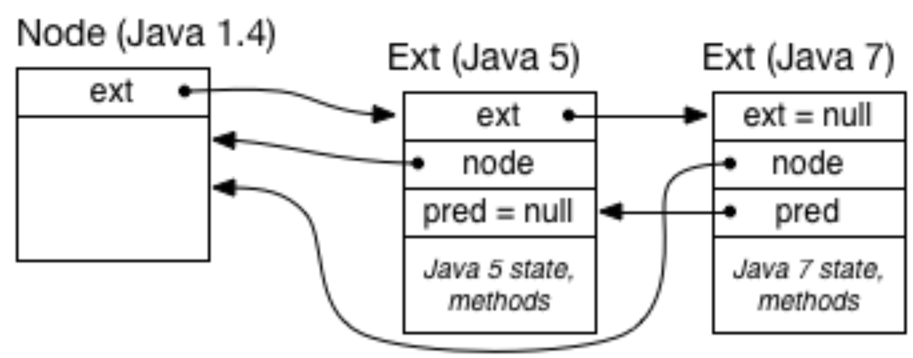
If the next extension does not know about CArray, extArrayTypeNode is invoked

CArrayOps

```
// Interface for any new operations here.
public interface CArrayOps extends NodeOps { }


public class CArrayExt extends Ext_c implements CArrayOps {
    ...
}
```

Extension objects

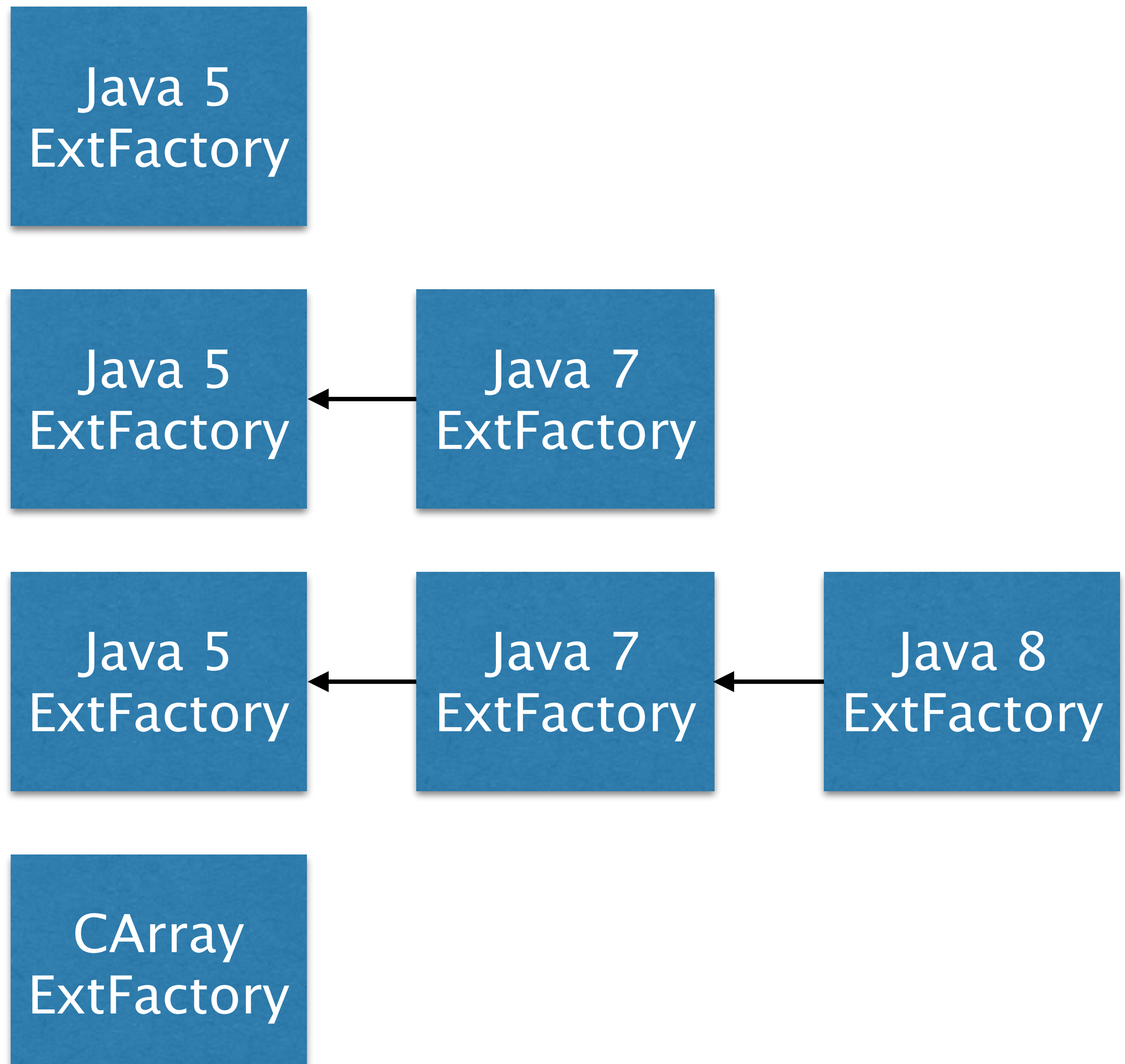
Extension objects are stored in a doubly-linked list



Extension objects are created by a per-language ExtFactory

Extension chaining

Extension objects are created by an ExtFactory, which are themselves chained (from the extension up)



Language dispatchers

Every language extension has a **language dispatcher**

The language dispatcher maps a node to the extension object for that language

`NodeOps(n)` (returns `n`)

`CArrayOps(n)` (returns `n`'s `CArrayOps ext`)

Every visitor has a language dispatcher.

For example, the TypeChecker visitor indirects calls to `typeCheck` for node `n` through the dispatcher

```
tc.lang().typeCheck(n, v)  ->  NodeOps(n).typeCheck(v)
```

CArrayLang

```
public interface CArrayLang extends JLang { }
```

CArrayLang_c

```
public class CArrayLang_c extends JLang_c implements CArrayLang {
    public static final CArrayLang_c instance = new CArrayLang_c();

    public static CArrayLang lang(NodeOps n) {
        while (n != null) {
            Lang lang = n.lang();
            if (lang instanceof CArrayLang) return (CArrayLang) lang;
            if (n instanceof Ext)
                n = ((Ext) n).pred();
            else return null;
        }
        throw new InternalCompilerError("Impossible to reach");
    }

    protected CArrayLang_c() {
    }

    protected static CArrayExt carrayExt(Node n) {
        return CArrayExt.ext(n);
    }

    // Get the Node operations of n (returns this language's ext for n)
    protected NodeOps NodeOps(Node n) {
        return carrayExt(n);
    }

    // Get the Node operations of n (returns this language's ext for n)
    protected CArrayOps CArrayOps(Node n) {
        return carrayExt(n);
    }
}
```

CArrayExt

```
public class CArrayExt extends Ext_c implements CArrayOps {
    private static final long serialVersionUID = SerialVersionUID.generate();

    public static CArrayExt ext(Node n) {
        Ext e = n.ext();
        while (e != null && !(e instanceof CArrayExt)) {
            e = e.ext();
        }
        if (e == null) {
            throw new InternalCompilerError("No CArray extension object for node "
                                                    + n
                                                    + " ("
                                                    + n.getClass()
                                                    + ")",
                                            n.position());
        }
        return (CArrayExt) e;
    }

    @Override
    public final CArrayLang lang() {
        return CArrayLang_c.instance;
    }
}
```

Extending the type system

Implementing the extension

1. Extend the parser with const array types
2. Add a const array AST node
3. Add a const array type
4. Implement subtyping for const arrays
5. Change the subtyping rules for Java arrays
6. Translate const arrays to Java arrays

Extending the type system

1. Create a Type subclass for the new type.
2. Extend TypeSystem to provide a factory method for the new type.
3. Extend the AST node classes to use the new type.

ConstArrayType

```
package carray.types;

/**
 * A {@code ConstArrayType} represents an array of base java types,
 * whose elements cannot change after initialization.
 */
public interface ConstArrayType extends ArrayType { }
```

ConstArrayType

```
package carray.types;

public class ConstArrayType_c extends ArrayType_c implements ConstArrayType {
    private static final long serialVersionUID = SerialVersionUID.generate();

    /** Used for deserializing types. */
    protected ConstArrayType_c() {
    }

    public ConstArrayType_c(TypeSystem ts, Position pos, Type base) {
        super(ts, pos, base);
    }

    ...
}
```

Exercise

Implement ConstArrayType_c by extending ArrayType_c

Implement a factory method for ConstArrayType in CArrayTypeSystem

Using the ConstArrayType

- Type building
- Type checking

Type building

The TypeBuilder constructs type objects from the AST

Invokes the buildTypes method on each AST node

Updates the **system resolver**, which maps fully qualified names (e.g., “java.util.HashMap”) to a TypeObject (e.g., a ClassType)

Exercise

Implement buildTypes for ConstArrayTypeNode

Model solution on ArrayTypeNode

Type building

```
@Override
public Node buildTypes(TypeBuilder tb) throws SemanticException {
    // Ask the type system to create a type object for constant array
    // of the base type.  At this point, the base type might not be
    // canonical.  The disambiguation phase will canonicalize the base type.
    CArrayTypeSystem ts = (CArrayTypeSystem) tb.typeSystem();
    return type(ts.constArrayOf(position(), base().type()));
}
```

Disambiguation

Unlike ASTs, type objects are mutable, updated during disambiguation

A type object is **canonical** if all ambiguities have been resolved

The AmbiguityRemover resolves names in the AST, updating type objects

Object => java.lang.Object

Disambiguation

The AmbiguityRemover …

… may cause other compilation units to be loaded

… runs multiple times, incrementally rewriting the AST

Example: ArrayTypeNode

```
Node disambiguate(AmbiguityRemover v) {
    TypeSystem ts = v.typeSystem();
    NodeFactory nf = v.nodeFactory();
    if (base.isCanonical())
        return nf.CanonicalTypeNode(ts.arrayOf(base.type()) );
    else
        return this; // will rerun the pass later
}
```

Exercise

Implement disambiguate for ConstArrayTypeNode

Model solution on ArrayTypeNode

Type checking

The TypeChecker pass type checks the AST

Invokes the typeCheck method on each AST node

Type checking

We need to:

- disallow assignment to constant arrays
    - override typeCheck in extension object for ArrayAccessAssign
- implement covariant subtyping for const arrays
    - override isImplicitCastValidImpl in ConstArrayType
- implement invariant subtyping for non-const arrays
    - introduce CArrayArrayType to replace ArrayType
    - override isImplicitCastValidImpl in CArrayArrayType

Implementing the extension

1. Extend the parser with const array types
2. Add a const array AST node
3. Add a const array type
4. Prevent assignment to const arrays
5. Implement subtyping for const arrays
6. Change the subtyping rules for Java arrays
7. Translate const arrays to Java arrays

Implementing typeCheck

Override typeCheck in the extension object for ArrayAccessAssign

```
public Node typeCheck(TypeChecker tc) throws SemanticException {
    // n is a[i] = v;
    ArrayAccessAssign n = (ArrayAccessAssign) this.node();
    // left is a[i].
    ArrayAccess left = n.left();
    // array is a.
    Expr array = left.array();

    if (array.type() instanceof ConstArrayType) {
        throw new SemanticException("Cannot assign an element of a const array.",
                                    n.position());
    }

    // Let the base language deal with the default type checking.
    return superLang().typeCheck(n, tc);
}
```

Implementing subtyping

Let’s look at how assignment is type-checked in Assign_c

```
if (op == ASSIGN) {
    if (!ts.isImplicitCastValid(s, t) &&
        !ts.typeEquals(s, t) &&
        !ts.numericConversionValid(t,
                 tc.lang().constantValue(right, tc.lang()))) {

        throw new SemanticException("Cannot assign " + s + " to " + t
                + ".", position());
    }

    return type(t);
}
```

Need to change isImplicitCastValid to support constant arrays

ArrayType_c

```
@Override
public boolean isImplicitCastValidImpl(Type toType) {
    if (toType.isArray()) {
        if (base().isPrimitive() || toType.toArray().base().isPrimitive()) {
            return ts.typeEquals(base(), toType.toArray().base());
        }
        else {
            return ts.isImplicitCastValid(base(), toType.toArray().base());
        }
    }

    // toType is not an array, but this is.  Check if the array
    // is a subtype of the toType.  This happens when toType
    // is java.lang.Object.
    return ts.isSubtype(this, toType);
}
```

Subtyping rules

$$\frac{A <: B}{A \text{ const } [\,] <: B \text{ const } [\,]}$$

Const arrays are covariant

$$\frac{}{A[\,] <: A[\,]}$$

Non-const arrays are invariant

$$\frac{}{A\ [\,] <: A \text{ const } [\,]}$$

Can assign a non-const array to a “const” array

Implementing the extension

1. Extend the parser with const array types
2. Add a const array AST node
3. Add a const array type
4. Implement subtyping for const arrays
5. Change the subtyping rules for Java arrays
6. Translate const arrays to Java arrays

Covariant const arrays

ConstArrayType_c

```
public boolean isImplicitCastValidImpl(Type toType) {
    if (!toType.isArray()) {
        // ?1 = ?2 const[]
        // This const array type is assignable to ?1 only if ?1 is Object.
        // Let the base language check this fact.
        return super.isImplicitCastValidImpl(toType);
    }

    // From this point, toType is an array.
    if (toType instanceof ConstArrayType) {
        // ?1 const[] = ?2 const[]
        // Let the base language check whether ?2 is assignable to ?1.
        return super.isImplicitCastValidImpl(toType);
    }

    // From this point, toType is a non-const array.
    // ?1[] = ?2 const[]
    // We cannot assign a const array to a non-const array.
    return false;
}
```

$$\frac{A <: B}{A\ \text{const}\ [\,] <: B\ \text{const}\ [\,]}$$

Implementing the extension

1. Extend the parser with const array types
2. Add a const array AST node
3. Add a const array type
4. Prevent assignment to const arrays
5. Implement subtyping for const arrays
6. Change the subtyping rules for Java arrays
7. Translate const arrays to Java arrays

Invariant non-const arrays

CArrayArrayType_c

```
public boolean isImplicitCastValidImpl(Type toType) {
    if (!toType.isArray()) {
        // ?1 = ?2 const[]
        // This const array type is assignable to ?1 only if ?1 is Object.
        // Let the base language check this fact.
        return super.isImplicitCastValidImpl(toType);
    }

    // From this point, toType is an array.
    if (toType instanceof CArrayArrayType) {
        // ?1[] = ?2[]
        // Non-const arrays are invariant, so we need to check that ?1 = ?2.
        return ts.typeEquals(base, toType.toArray().base());
    }
```

A[] <: A[]

```
    // From this point, toType is a const array.
    // ?1 const[] = ?2[]
    // We can assign a non-const array to a const array only if ?2 is a
    // subtype of ?1. Java arrays have this semantics, so let the base
    // language deal with this.
    return super.isImplicitCastValidImpl(toType);
}
```

A [] <: A const []

Exercise: casting

Can also cast from a non-const array to a const array

Implement isCastValidImpl in ConstArrayType_c

Implementing constant arrays

Override typeCheck for assignment to disallow assigning to constant arrays

Creating an extension object

```
package carray.ast;

import polyglot.ast.Ext;
import polyglot.ast.ExtFactory;

public final class CArrayExtFactory_c extends CArrayAbstractExtFactory_c {
   public CArrayExtFactory_c() {
        super();
    }

    public CArrayExtFactory_c(ExtFactory nextExtFactory) {
        super(nextExtFactory);
    }

    @Override
    protected Ext extNodeImpl() {
       return new CArrayExt();  // default extension
    }

    @Override
    protected Ext extArrayAccessAssignImpl() {
       return new CArrayArrayAccessAssignExt();
    }
}
```

CArrayArrayAccessAssignExt

```
@Override
public Node typeCheck(TypeChecker tc) throws SemanticException {
    // Suppose n is a[2] = 3;
    ArrayAccessAssign n = (ArrayAccessAssign) this.node();
    // Then left is a[2].
    ArrayAccess left = n.left();
    // And array is a.
    Expr array = left.array();

    // If the type of the array is a ConstArrayType, then this assignment
    // is illegal.
    if (array.type() instanceof ConstArrayType) {
        throw new SemanticException("Cannot assign into a const array.",
                                    n.position());
    }

    // Let the base language deal with the default type checking.
    return superLang().typeCheck(n, tc);
}
```

CArrayArrayAccessAssignExt

```
@Override
public Node typeCheck(TypeChecker tc) throws SemanticException {
    // Suppose n is a[2] = 3;
    ArrayAccessAssign n = (ArrayAccessAssign) this.node();
    // Then left is a[2].
    ArrayAccess left = n.left();
    // And array is a.
    Expr array = left.array();

    // If the type of the array is a ConstArrayType, then this assignment
    // is illegal.
    if (array.type() instanceof ConstArrayType) {
        throw new SemanticException("Cannot assign into a const array.",
                                    n.position());
    }

    // Let the base language deal with the default type checking.
    return superLang().typeCheck(n, tc);
}
```

Translation

Implementing the extension

1. Extend the parser with const array types
2. Add a const array AST node
3. Add a const array type
4. Prevent assignment to const arrays
5. Implement subtyping for const arrays
6. Change the subtyping rules for Java arrays
7. Translate const arrays to Java arrays

Translation

Polyglot will pretty print Java ASTs

To implement translation to Java, we can override pretty-printing

Less error-prone is to translate ASTs into base language ASTs

Translation

To translate ASTs, we add a pass before the CodeGenerated pass

CodeGenerated requires a type-checked AST.

Rather than generate type-correct ASTs during translation, we can generate untyped ASTs and run the type checker again

Goals and passes

Compiler maintains a dependency graph of **goals**

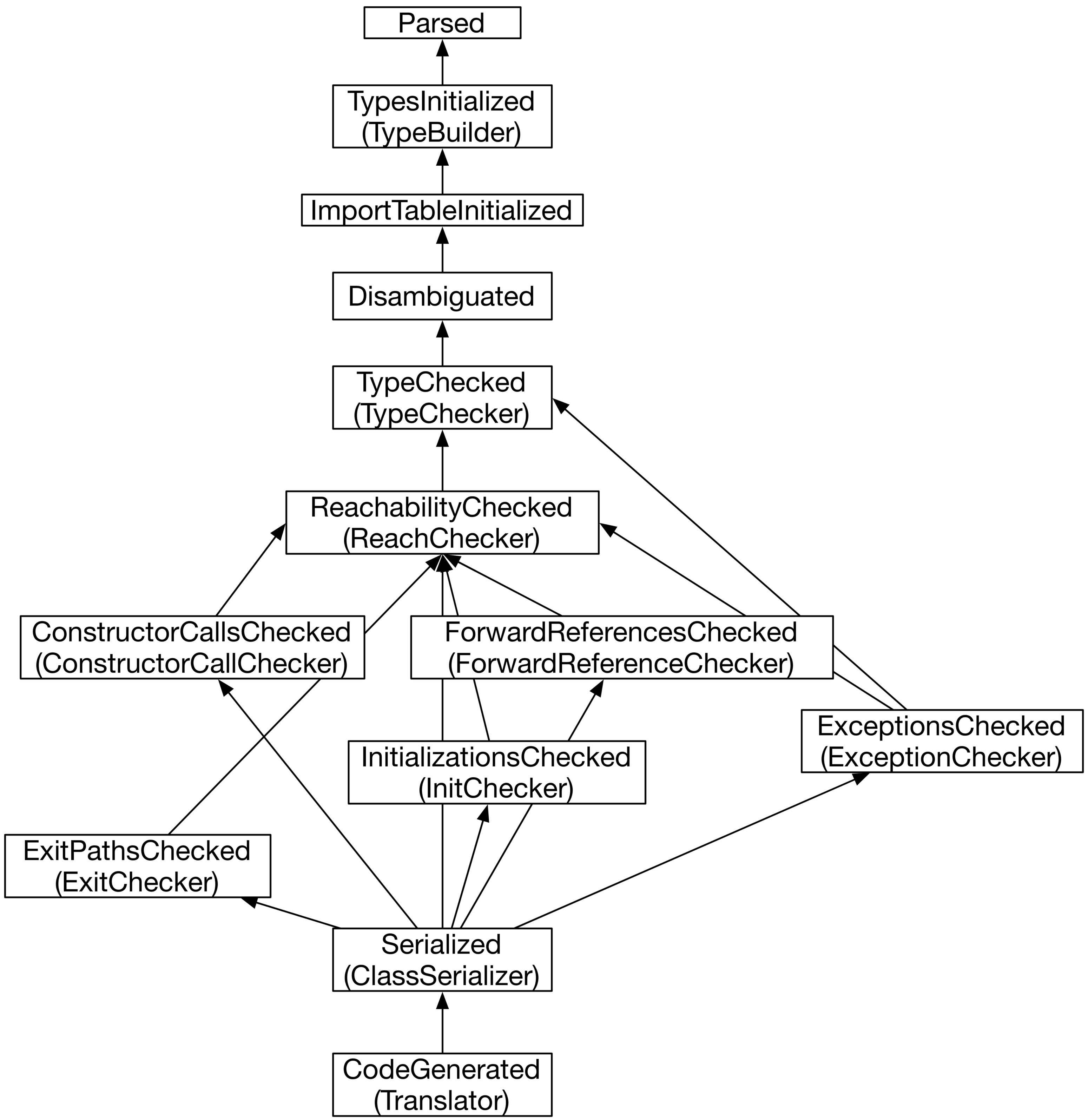
Goals have **prerequisites**

To fulfill a goal, all prerequisites must be reached and the **pass** for the goal must complete

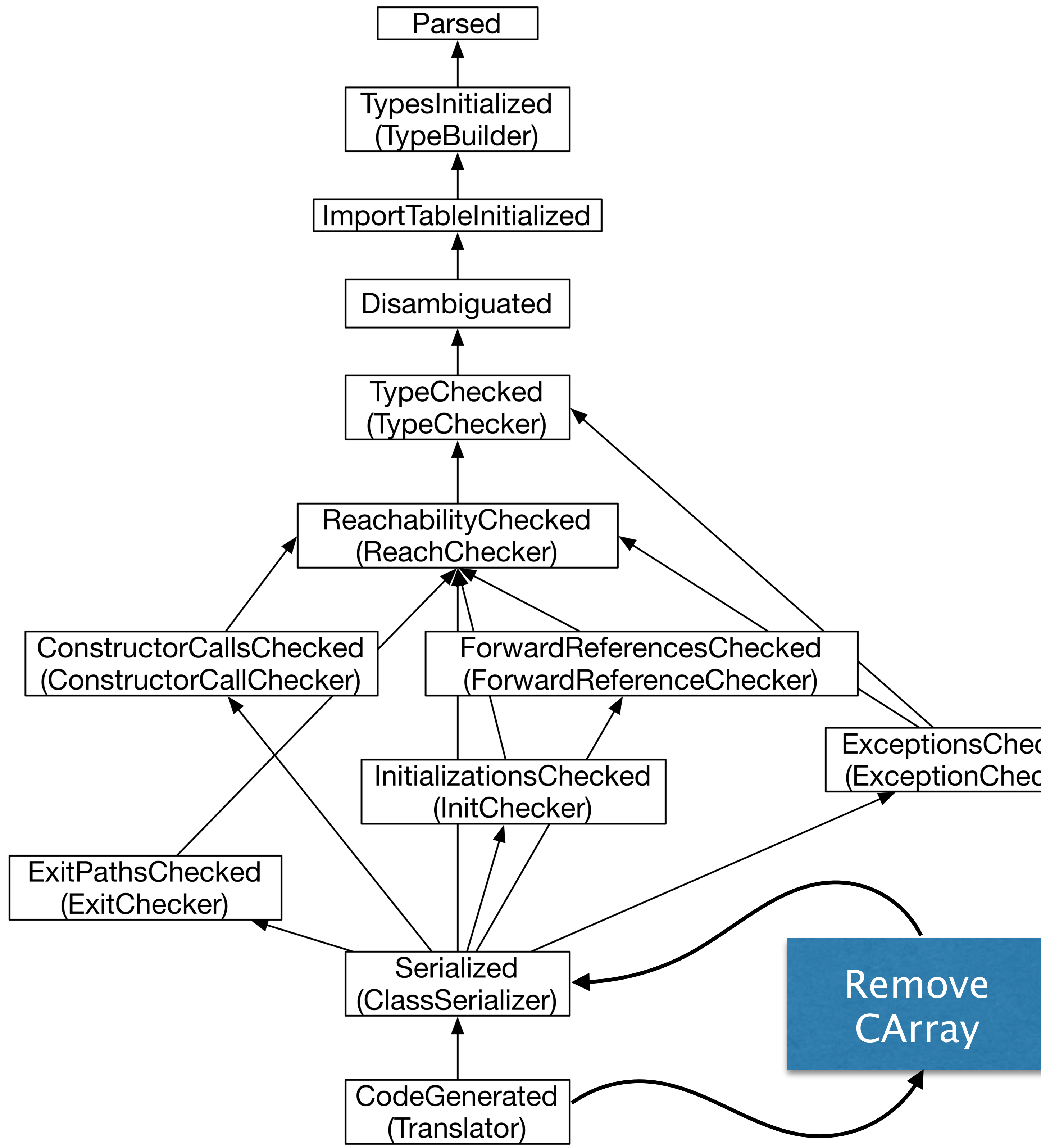
Typical goals:

- type check a compilation unit
- generate code for a compilation unit
- resolve a single name reference
- barrier (block until all compilation units have reached a goal)

Goals and passes



Scheduling translation



Translating to Java arrays

```
class CArrayRewriter extends ExtensionRewriter {
    ...
    public TypeNode typeToJava(Type t, Position pos) throws SemanticException {
        // Convert constant arrays to regular Java 1.4 arrays
        if (t instanceof ConstArrayType) {
            // X const[] --> X[]
            ConstArrayType at = (ConstArrayType) t;
            Type base = at.base();
            NodeFactory nf = to_nf();
            return nf.ArrayTypeNode(pos, typeToJava(base, base.position()));
        }
        return super.typeToJava(t, pos);
}
```

Supporting separate compilation

To support separate compilation, need to store extension-specific type information in Java .class files

Before code generation, the serialization pass encodes the type information into fields of the generated code

For this to work, TypeObjects must implement java.lang.Serializable and cannot have (non volatile) references to non serializable objects

Another translation

Instanceof

A problem with translating constant arrays to Java arrays is that `instanceof` behaves incorrectly

```
Bird const[] birds = …
birds instanceof Animal const []   // should return false!
```

But in our translation:

```
Bird[] birds = …
birds instanceof Animal[]   // returns true
```

Another translation

Idea: wrap const array objects in an object

Extends the Java5 extension and generates Java5

See carray-full.zip for the complete implementation

Contributing

If you want to contribute to Polyglot

- http://www.cs.cornell.edu/Projects/polyglot
- https://github.com/polyglot-compiler/polyglot

Email:

- nate.nystrom@usi.ch
- andru@cs.cornell.edu
- chong@seas.harvard.edu