Introduction to JastAdd

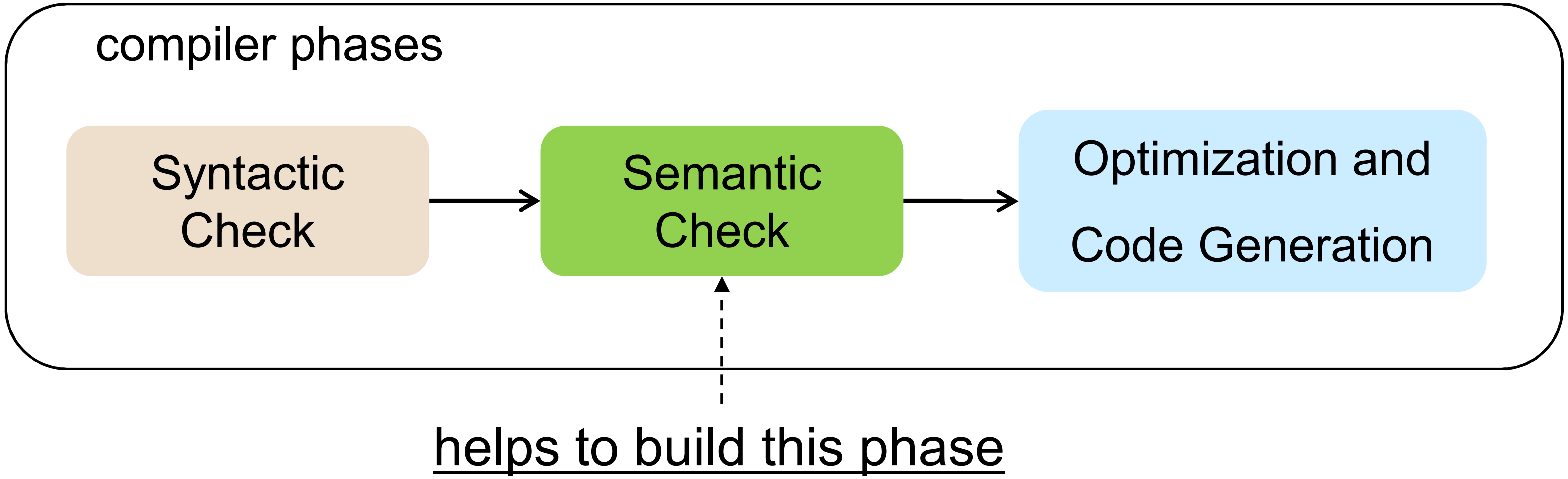
Hyunik Na
PL Lab@KAIST

ROSAEC 8th Workshop

2012.7.25~28

What Is JastAdd? (1/2)

- A meta-compiler for semantic checkers of language-based tools
  - e.g. compilers, program analyzers, language-sensitive editors



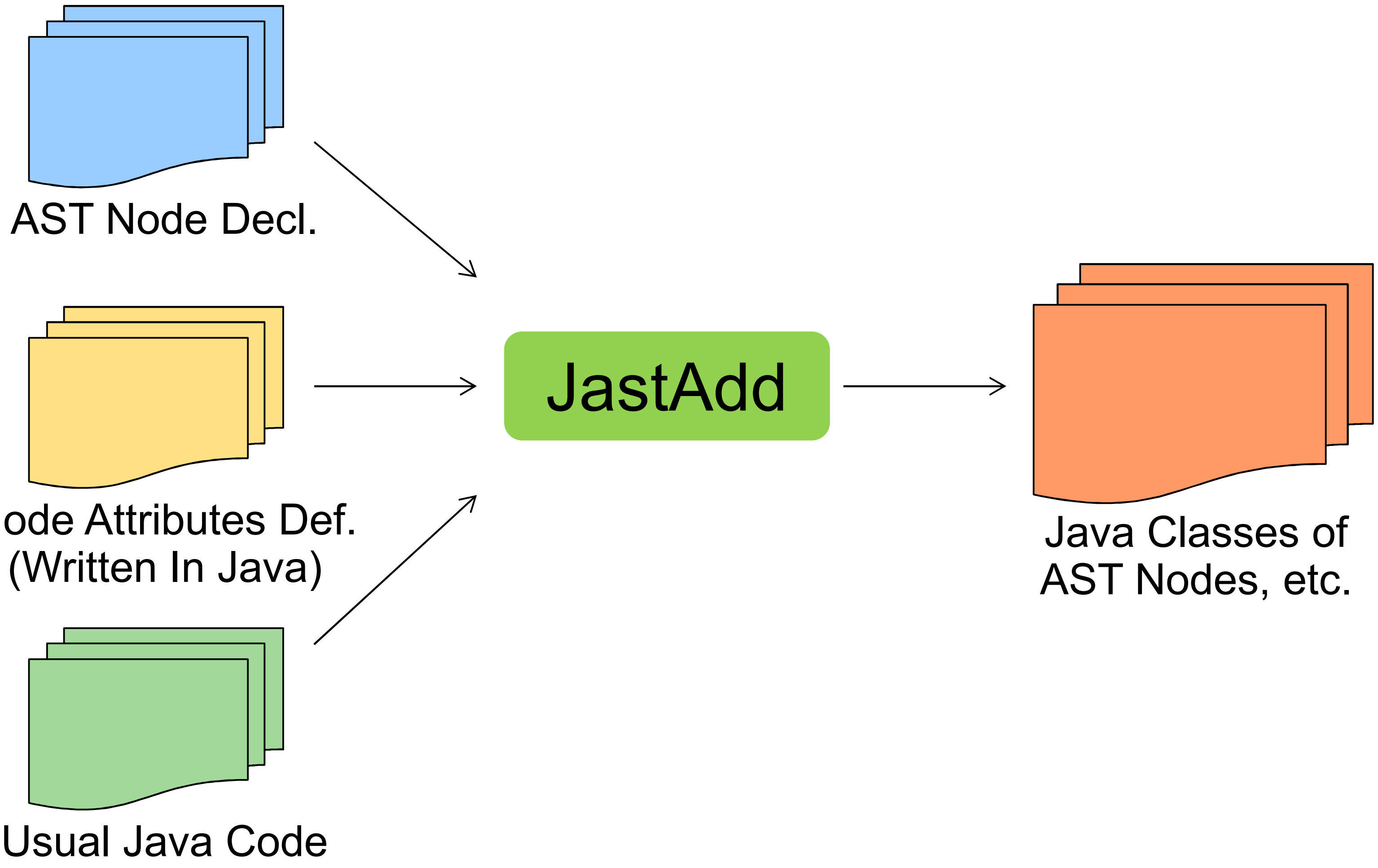
- Developed by Görel Hedin and her group in Lund University, Sweden since early 2000’s

What Is JastAdd? (2/2)

- Abstract Syntax Tree (AST) manipulator generator
  - Simple declaration of AST nodes
  - Various kinds of AST node attributes
  - AST rewriting

- Its motto: “Every computation on AST”
  - No separate symbol tables
  - No separate intermediate representations
  - No separate control flow graphs or call graphs
    - Superimpose them on AST if necessary

Input and Output



AST Node Decl.

Node Attributes Def.
(Written In Java)

Usual Java Code

JastAdd

Java Classes of
AST Nodes, etc.

Simple Declaration of AST Nodes: Syntax-based

Syntax	Description	Generated Java Class
T;	AST node T	class T extends ASTNode { … }

M: N ::= A B C;	M is a kind of N.
	M has children A, B and C.
	Amounts to the productions:
	"N → M"
	"M → A B C"
		class M extends N {
		A getA();
		B getB();
		C getC();
		…
		}

M: N ::= [B] C* <D>	M has an optional child B, children of Cs, and a string token D	class M extends N {
		boolean hasB();
		B getB();
		
		int getNumC();
		C getC(int i);
		
		String getD();
		…
		}

Examples from a Java Compiler

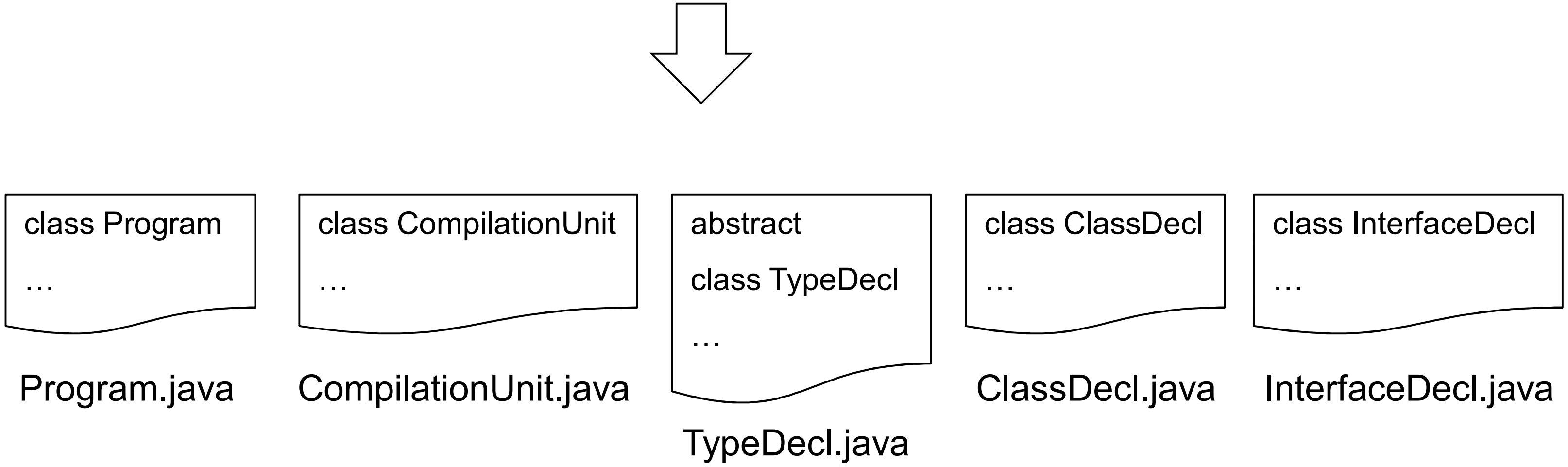
```
Program ::= CompilationUnit*;
CompilationUnit ::= <PackageName> ImportDecl* TypeDecl*;

abstract TypeDecl;
ClassDecl: TypeDecl ::= Modifiers <ID> [Super] Impl* BodyDecl*;
InterfaceDecl: TypeDecl ::= Modifiers <ID> Super* BodyDecl*;
```

Examples from a Java Compiler

```
Program ::= CompilationUnit*;
CompilationUnit ::= <PackageName> ImportDecl* TypeDecl*;

abstract TypeDecl;
ClassDecl: TypeDecl ::= Modifiers <ID> [Super] Impl* BodyDecl*;
InterfaceDecl: TypeDecl ::= Modifiers <ID> Super* BodyDecl*;
```



class Program
…

Program.java

class CompilationUnit
…

CompilationUnit.java

abstract
class TypeDecl
…

TypeDecl.java

class ClassDecl
…

ClassDecl.java

class InterfaceDecl
…

InterfaceDecl.java

AST Node Attributes

- Central in semantic checks
  - Semantic checks are computing and verifying node attribute values

- Translated to methods and code for tree traversal, fixed point iteration, collection management, etc in generated node classes

- Various Kinds of AST node attributes
  - Synthesized/Inherited attributes
  - Non-terminal attributes
  - Circular attributes
  - Collection attributes

- Features
  - Stateless: should return the same value on every lookup
  - May be cached : when declared 'lazy'
  - May have parameters like methods

Synthesized Attribute

- Computed within the subtree rooted at the node
- Translated to a simple method of the generated node class
  - may be abstract, overridden or "inherited" through class hierarchy

Example: Casting Conversion in Java

```
syn boolean TypeDecl.castingConversionTo(TypeDecl type);

eq ClassDecl.castingConversionTo(TypeDecl type) {
    if(type.isArrayDecl())
      return isObject();
    else if(type.isClassDecl())
      return instanceOf(type) || type.instanceOf(this);
    else if(type.isInterfaceDecl())
      return instanceOf(type) || !isFinal();
    else
      return false;
}
```

Inherited Attribute

- Computed by an ancestor node in the AST
  - NOTE: Inherited through AST, not through class hierarchy

- Translated to an upward AST traversal code which finds the first ancestor node that can compute the attribute

- Notation

```
inh T N.x();                        // declaration
eq N2.getChild().x() {              // definition
   return Java-expr;
}
```

Example: Type Lookup in Java

```
inh TypeDecl Expr.lookupType(String name);
inh TypeDecl Stmt.lookupType(String name);

eq Block.getStmt().lookupType(String name)
{   // search local classes, and send request upward on failure   }

eq TypeDecl.getBodyDecl().lookupType(String name)
{   // search nested classes, and send request upward on failure  }

eq CompilationUnit.getChild().lookupType(String name)
{   // search top-level TypeDecl’s in the unit,
    // and send request upward on failure   }

eq Program.getChild().lookupType(String name)
{   // search whole compilation units   }
```

Non-terminal Attribute

- An AST node generated on the fly during semantic check, not during parsing
- Mainly, for “derived types”
  - C[ ] is a non-terminal attribute of C
  - C<Integer, Boolean> and C<Float, Double> are non-terminal attributes of C<X,Y>
- Notation

```
syn nta N2 N.x() = new N2(…);            // may be either syn or inh
inh nta N2 N.x() = new N2(…);
```

Circular Attribute

- An attribute whose value depends on itself.
  - Starts with given initial value, and continues until no change

- Examples
  - "In-set" and "out-set" of a data-flow analysis
  - Does a class extend itself either directly or indirectly?

- Notation

```
syn T N.x() circular [init-val];        // declaration
eq N.x() = Java-expr;                   // definition
```

Collection Attribute

- An attribute whose value is a collection with many contributor nodes

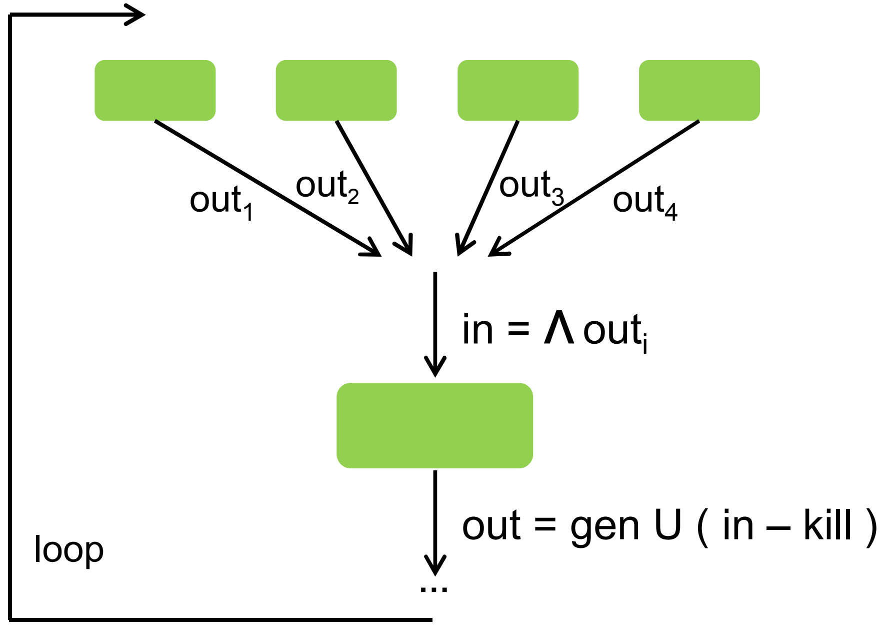
- Examples
  - Predecessor nodes in a control-flow graph when successors are known
  - Use sites of a variable declaration

- Notation

```
// collection attribute declaration
coll T N.x() [coll-init] with adder-name;

// contributor declaration
N2 contributes val-expr
when cond-expr
to N.x()
for N-ref-expr;
```

Example of Circular and Collection Attributes: Data-flow Analysis



- 'in' and 'out' are circular attributes of a node
  - in depends on out and vice versa
  - possible loops in CFG
- Usually, they are collections of values
  - i.e. collection attributes

Example of Circular and Collection Attributes: Data-flow Analysis

```
// Intra-procedural data flow analysis with ∧ = ∪

syn Set CFGNode.out() circular [Set.emptySet()] =
  gen().union( in().compl( kill() ) );

coll Set CFGNode.in() circular [Set.emptySet()] with add;
CFGNode contributes out() to CFGNode.in() for each succ();
```

AST Rewrite

- Sometimes, precise AST cannot be built during parsing
  - For example, “pkg.class.fld” is parsed to

    ```
    Dot(Name(“pkg”), Dot(Name(“class”), Name(“fld”)))
    ```

    But, it should be converted to

    ```
    Dot(PkgAcc(“pkg”), Dot(TypeAcc(“class”), VarAcc(“fld”)))
    ```

- Also, useful for desugaring and implicit construct

- Notation

```
rewrite N {
   when (cond-exp)
   to N2 {
      return Java-exp;
   }                     // There may be multiple when-to clauses
   …                     // whose conditions are checked in order.
}
```

Examples from a Java Compiler

```
rewrite AmbiguousName {
  to Access {
    if( !lookupVariable(name()).isEmpty() )
       return new VarAccess( name() );
    else if( !lookupType(name()).isEmpty() )
       return new TypeAccess( name() );
    else
       return new PackageAccess( name() );
  }
}

rewrite ConstructorDecl {
  when( !hasConstructorInvocation() && !hostType().isObject() )
  to ConstructorDecl {
    setConstructorInvocation(
      new ExprStmt(new SuperConstructorAccess("super", new List())));
    return this;
  }
}
```

Conclusion

- JastAdd is a framework to implement semantic check phase of a language-based tools
  - Various features for computations on AST
    - caching, inh, fixed point iteration, collections, AST nodes on the fly, etc
  - Features are modular and orthogonal ➔ easy extension

- Further reading
  - Görel Hedin, “An Introductory Tutorial on JastAdd Attribute Grammars”, GTTSE 2009 (and other papers it sites)
  - Reference manual for JastAdd (http://jastadd.org/web/)
  - JastAddJ source code