Qualifying Exam

Nicholas Chen
nchen@uiuc.edu

JastAdd Extensible Java Compiler

Torbjörn Ekman and Görel Hedin

OOPSLA ‘07

What's the paper about?

- Building reliable compilers and analysis tools for a language is hard.
- Languages evolve and the tools for building compilers do not handle language evolution well.
- An extensible metacompiler tool embracing OOP, AOP and Attribute Grammars can alleviate these problems by maximizing modularity.
- JastAddJ compiler was created using such a metacompiler tool - it supports Java 1.4 with support for Java 1.5 and other analysis tools *added* as *modular extensions*.

Agenda

1. **What's JastAdd?**
2. Example of Using JastAdd
3. Novelties of JastAdd
4. Improving JastAdd

What's JastAdd

- Extensible Compiler – syntax and behavior
- Allows modularly expressing syntax and behavior
- **Not** a *lexer generator*
- **Not** a *parser generator*

Not an *all-in-one* solution for building language tools

Compiler	% pass	# pass	# fail
javac 1.4	99.0	4446	44
eclipse 1.4	98.1	4409	81
jastadd 1.4	99.5	4468	22

Jacks test suite

Compiler	junit	jhotdraw	JDK	ejc	jigsaw
polyglot2	✔	✔	✖	✖	✖
jaco	✔	✖	✖	✖	✖
jastadd 1.4	✔	✔	✔	✔	✔

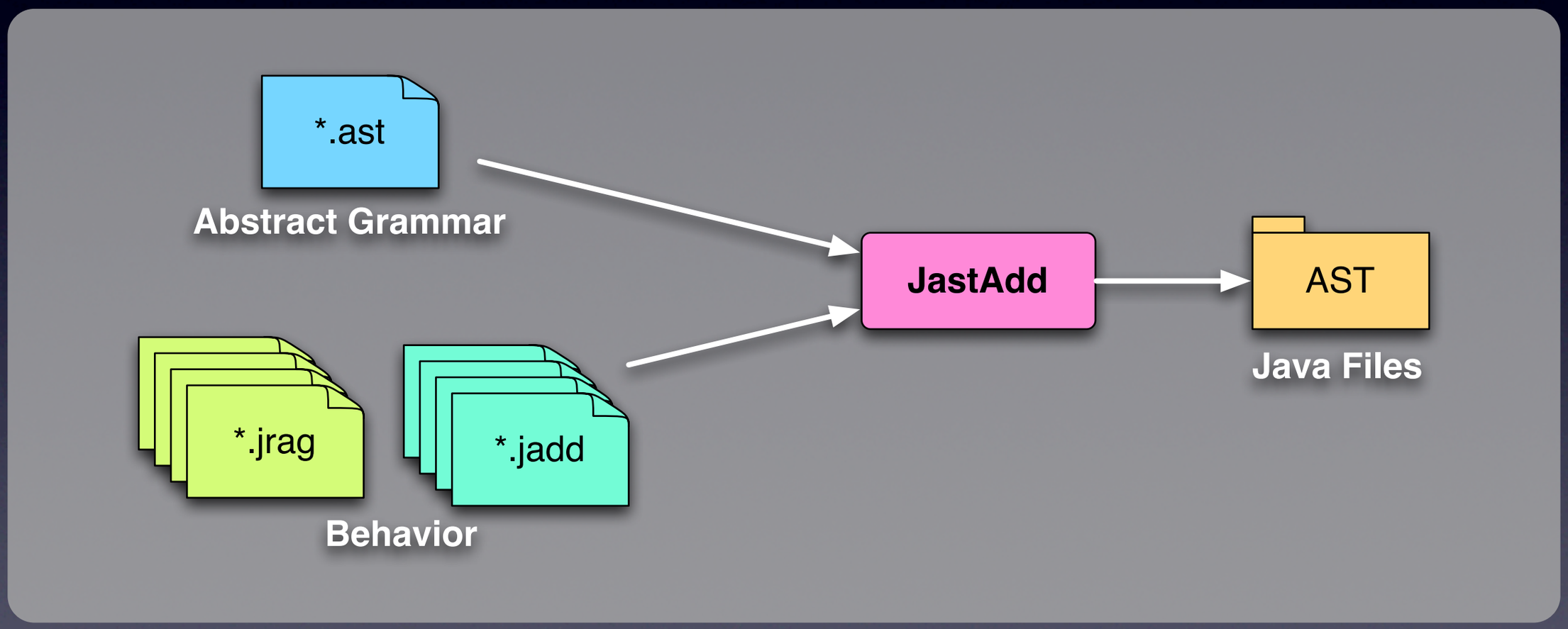
Compiling Java 1.4 Applications

Compiler	# KLOC
javac 1.5	30 (100%)
eclipse 1.5	83 (297%)
jastadd 1.5	21 (66%)

Source code size

1. What's JastAdd?
2. **Example of Using JastAdd**
3. Novelties of JastAdd
4. Improving JastAdd

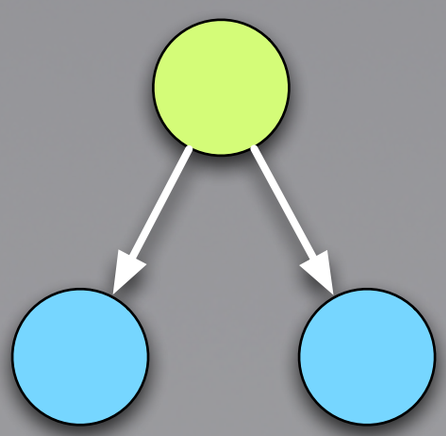
Generation Architecture



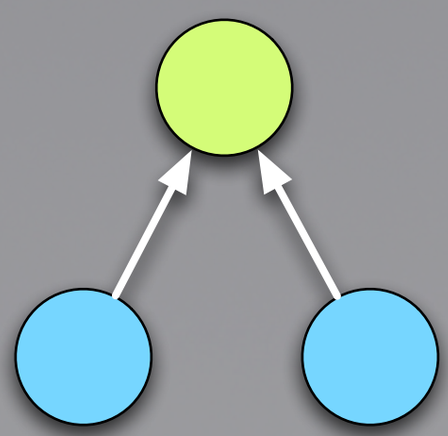
Attribute Grammars

- Introduced by Knuth in 1968
- A way to formalize the semantics of context free languages (CFG)

- Original formalism had *synthesized attributes* and *inherited attributes*

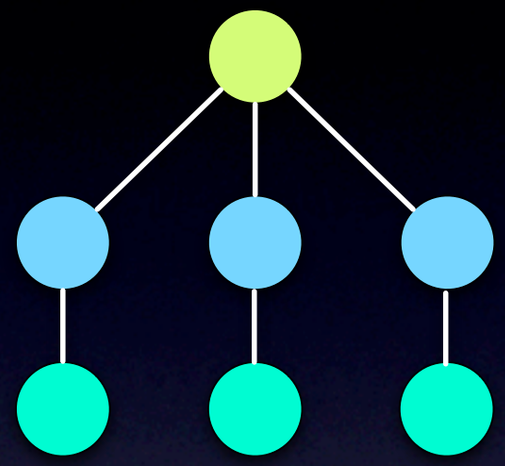


Inherited attributes transport information *down* the green node



Synthesized attributes transport information *up* from the blue node(s)

My Example – $a^n b^n c^n$ Sequence



Production	Semantic Rules
S → ABC	B.inhSize ← A.size C.inhSize ← A.size
A → **a**	A.size ← 1
A → A_1**a**	A.size ← A_1.size + 1
B → **b**	B.isValid ← B.inhSize == 1
B → B_1**b**	B.inhSize ← B_1.inhSize – 1

JastAdd Syntax – Abstract Grammar

```
abstract ALetterSequence;
abstract BLetterSequence;
abstract CLetterSequence;

ValidSequence ::= ASequence: ALetterSequence BSequence: BLetterSequence
CSequence: CLetterSequence;

ASingleChar: ALetterSequence ::= A;
AListChar: ALetterSequence ::= Seq: ALetterSequence A;

BSingleChar: BLetterSequence ::= B;
BListChar: BLetterSequence ::= Seq: BLetterSequence B;

CSingleChar: CLetterSequence ::= C;
CListChar: CLetterSequence ::= Seq: CLetterSequence C;

abstract Char;
A: Char;
B: Char;
C: Char;
```

JastAdd Behavior File *Attributes*

```
aspect LetterSequenceAttr {
    // Attribute for ALetterSequence
    syn int ALetterSequence.size();

    // Attributes for BLetterSequence
    inh int BLetterSequence.inhSize();
    syn boolean BLetterSequence.isValid();

    // Attributes for CLetterSequence
    inh int CLetterSequence.inhSize();
    syn boolean CLetterSequence.isValid();
}
```

JastAdd Behavior File *Equations*

```
aspect LetterSequenceEq {
    // Equations for ALetterSequence
    eq ASingleChar.size() = 1;
    eq AListChar.size() = getSeq().size() + 1;

    // Equations for BSingleChar
    eq BSingleChar.isValid() = inhSize() == 1;

    // Equations for BListChar
    eq BListChar.getSeq().inhSize() = inhSize() - 1;
    eq BListChar.isValid() = getSeq().isValid();

    // Equations for CSingleChar
    eq CSingleChar.isValid() = inhSize() == 1;

    // Equations for CListChar
    eq CListChar.getSeq().inhSize() = inhSize() - 1;
    eq CListChar.isValid() = getSeq().isValid();

    // Equations for ValidSequence
    eq ValidSequence.getBSequence().inhSize() = getASequence().size();
    eq ValidSequence.getCSequence().inhSize() = getASequence().size();
}
```

Embedding Java code

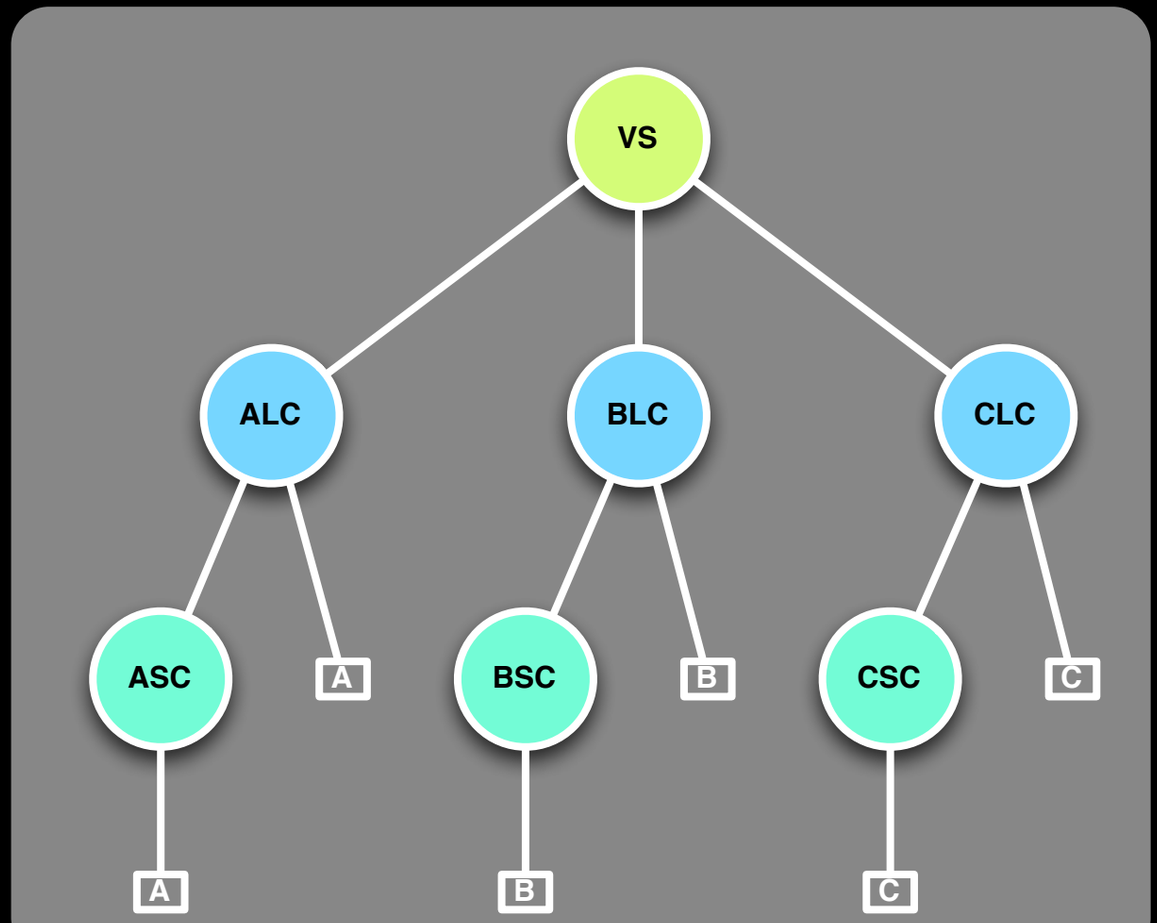
```
aspect LetterSequenceHelper {
    // Helper methods
    public boolean ValidSequence.isValid() {
        if(getASequence() == null
           || getBSequence() == null
           || getCSequence() == null)
            return false;

        return getBSequence().isValid()
               && getCSequence().isValid();
    }
}
```

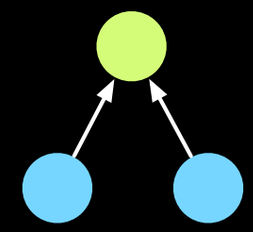
Testing

```
// Test "aabbcc"
public void testValidLonger() {
    // Parser usually does construction
    ValidSequence vs = new ValidSequence(
        new AListChar(
            new ASingleChar(new A()),
            new A()),
        new BListChar(
            new BSingleChar(new B()),
            new B()),
        new CListChar(
            new CSingleChar(new C()),
            new C()));

    assertTrue(vs.getBSequence().isValid());
    assertTrue(vs.getCSequence().isValid());
    assertTrue(vs.isValid());
}
```



Generating *Synthesized Attributes*



ListChar <: LetterSequence

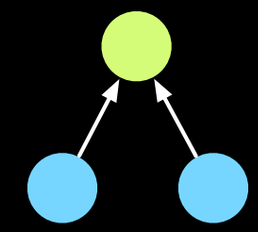
```
// Attribute for LetterSequence
syn int LetterSequence.size();

// Equation for ListChar
eq ListChar.size()
 = getSeq().size() + 1;
```

```
public abstract class LetterSequence
extends ASTNode implements Cloneable {
    // Declared in CFG.ast line 3
    public LetterSequence() {
        super();
    }

    // Declared in CFG1.jrag at line 3
    public abstract int size();
  ...
}
```

Generating *Synthesized Attributes*



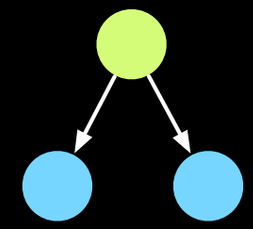
AListChar <: ALetterSequence

```
// Attribute for ALetterSequence
syn int ALetterSequence.size();

// Equation for AListChar
eq AListChar.size()
 = getSeq().size() + 1;
```

```
public class AListChar
extends ALetterSequence implements Cloneable {
    ...
    // Declared in CFG1.jrag at line 3
    protected boolean size_visited = false;
    public int size() {
        ...
        int size_value = size_compute();
        ...
        return size_value;
    }
    private int size_compute() {
        return getSeq().size()+1;
    }
}
```

Generating *Inherited Attributes*



BListChar <: BLetterSequence

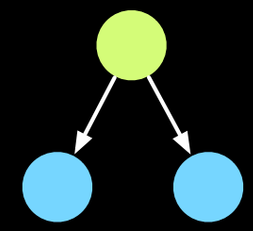
```
// Attribute for BLetterSequence
inh int BLetterSequence.inhSize();

// Equation for BListChar
eq BListChar.getSeq().inhSize()
 = inhSize() - 1;
```

```
public abstract class BLetterSequence
extends ASTNode implements Cloneable {
    // Declared in CFG.ast line 4
    public BLetterSequence() {
        super();
    }

    // Declared in CFG1.jrag at line 6
    public int inhSize()
     ...
        int inhSize_value =
     getParent().Define_int_inhSize(this, null);
     ...
        return inhSize_value;
    }
...
}
```

Generating *Inherited Attributes*



BListChar <: BLetterSequence

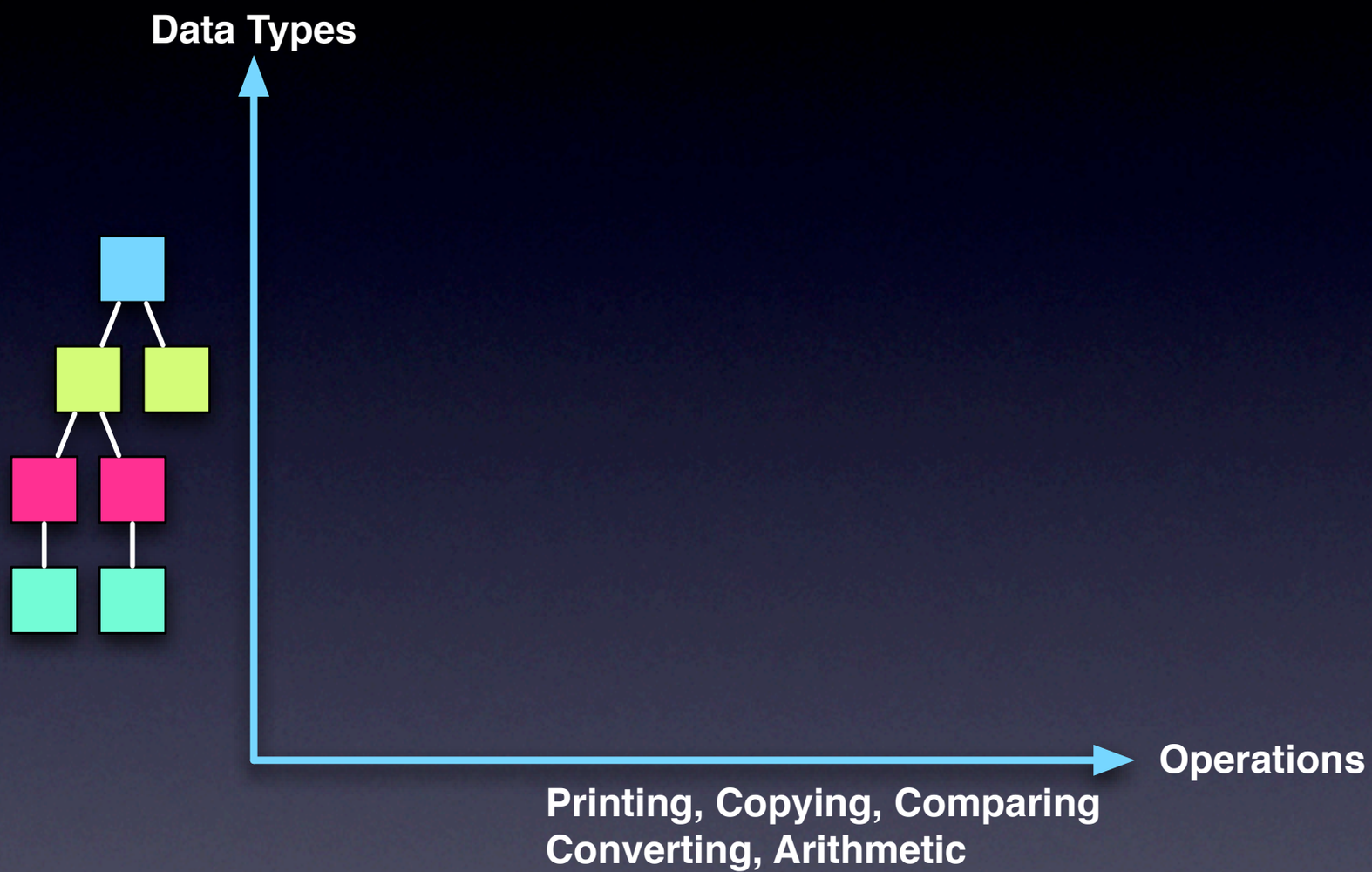
```
// Attribute for BLetterSequence
inh int BLetterSequence.inhSize();

// Equation for BListChar
eq BListChar.getSeq().inhSize()
 = inhSize() - 1;
```

```
public class BListChar extends
BLetterSequence implements Cloneable {
    // Declared in CFG1.jrag at line 23
    public int Define_int_inhSize
    (ASTNode caller, ASTNode child) {
        if(caller == getSeqNoTransform()) {
            return  inhSize() - 1;
        }
    ...
    }
    ...
}
```

1. What's JastAdd?
2. Example of Using JastAdd
3. **Novelties of JastAdd**
4. Improving JastAdd

Extensibility Problem



Can data types & set of operations over them be extended without modifying existing code?

Aspect Orientation

- Innovative use of Aspect-Oriented Programming *Paradigms*
- Using inter-type declarations helps introduce fields and methods for different analysis tools
- Allows for modularity by *weaving* syntax and attributes at generation time

Aspect Orientation

- Aspects vs. Visitor Pattern
  - methods can be factored out but fields still have to be directly declared in the classes
  - multiple visitors for different tasks cannot really share information modularly
  - generic method names that don't communicate intent

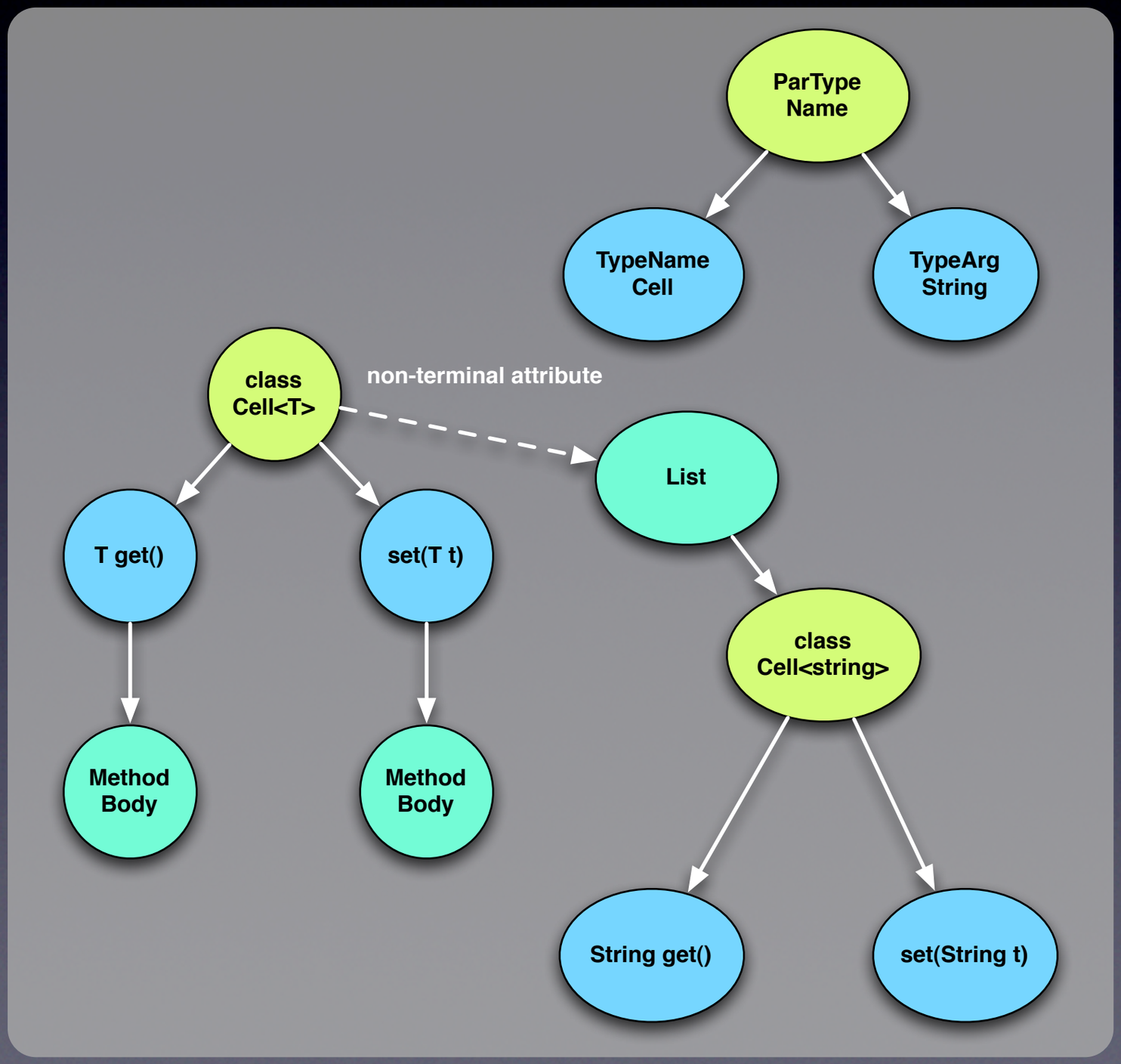
```
Object visit (Node node, Object arguments)

Type typeCheck (Type expectedType)
```

AST & Reference Attributes

- Allows *information sharing* through the AST by referencing relevant nodes
- Eliminates the need for multiple symbol tables
- Allows an *intuitive* manner to visualize bindings – it just points to the declaration site.

Non-terminal Attributes

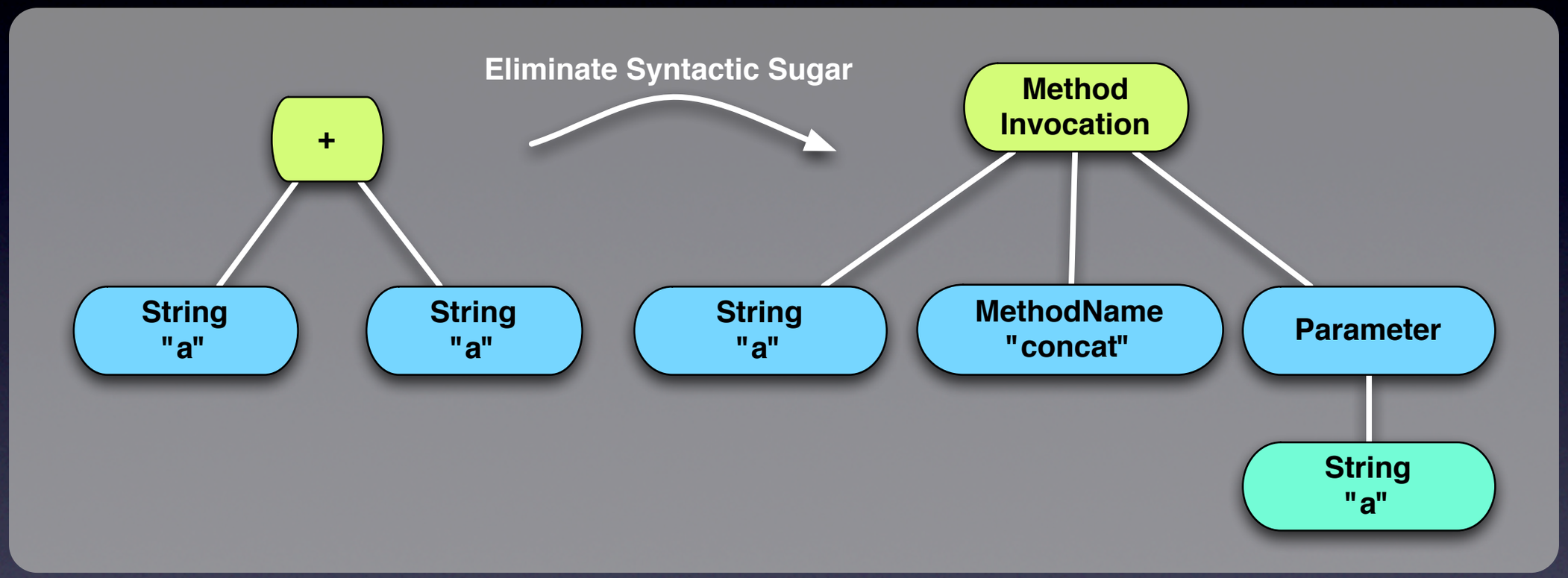


Non-terminal attributes allow subtrees to be created on-the-fly as necessary to represent new structure

```
A ::= B /C*/

syn C A.getCList() =
new List().
    add(new C()).
    add(new C());
```

Rewritable Reference Attributed Grammars




```
// Sample rewrite statement to transform tree
 rewrite Add {
    when (childType().equals(TypeSystem.STRING))
         to new MethodInvocation(leftOp,
                    new MethodName(“concat”),
                    new Parameter(rightOp))
}
```

1. What's JastAdd?
2. Example of Using JastAdd
3. Novelties of JastAdd
4. **Improving JastAdd**

Visual IDE

- Lacks a visual IDE; possible features for one:
  - Stepping through execution/ propagation of attributes visually
  - Syntax highlighting based on the grammar
  - Detects mistake in specification

My Evaluation of JastAdd

- The tools should not distract the developer with information overload.
- **The tools must be adaptive and work as the software being developed on evolves.**
- The tools should be non-intrusive; the developer should be able to use the tools with minimal changes to his existing software artifacts.
- The tools should not force the developer to use a new unfamiliar environment but should work with existing tools that the developer is familiar with.

Conclusion

JastAdd is a clean way to design compilers and language analysis tools in a modular fashion that handles extensibility well. This paper presents some of its novel techniques and how to reuse them.

Questions

Qualifying Exam

Nicholas Chen
nchen@uiuc.edu

Appendices

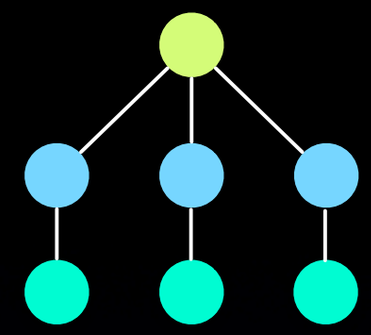
People behind JastAdd

- Görel Hodin and Torbjörn Ekman
  - Lund University, Sweden
- Various flavors of Reference Attributed Grammars
- Focus on using attribute grammar technologies

Evolution of JastAdd

- APPlication language LABoratory – AppLab (circa 1999)
  - Reference Attributed Grammars (RAG)
- JastAdd version 1 (circa 2002)
  - RAG + Aspect Orientation + Imperative Java
- JastAdd version 2 (circa 2006)
  - RAG, Rewritable Reference Attributed Grammar (ReRAG), Circular Reference Attributed Grammar (CRAG) + Aspect Orientation + Imperative Java

$a^n b^n c^n$ Sequence Example



Production	Semantic Rules
S → ABC	B.inhSize ← A.size C.inhSize ← A.size
A → **a**	A.size ← 1
A → A_1**a**	A.size ← A_1.size + 1
B → **b**	B.isValid ← B.inhSize == 1
B → B_1**b**	B.inhSize ← B_1.inhSize – 1
C → **c**	C.isValid ← C.inhSize == 1
C → C_1**c**	C.inhSize ← C_1.inhSize – 1

JastAdd Syntax – Abstract Grammar

```
abstract ALetterSequence;
abstract BLetterSequence;
abstract CLetterSequence;

ValidSequence ::= ASequence: ALetterSequence BSequence: BLetterSequence
CSequence: CLetterSequence;

ASingleChar: ALetterSequence ::= A;
AListChar: ALetterSequence ::= Seq: ALetterSequence A;

BSingleChar: BLetterSequence ::= B;
BListChar: BLetterSequence ::= Seq: BLetterSequence B;

CSingleChar: CLetterSequence ::= C;
CListChar: CLetterSequence ::= Seq: CLetterSequence C;

abstract Char;
A: Char;
B: Char;
C: Char;
```

JastAdd Behavior File *Attributes*

```
aspect LetterSequenceAttr {
    // Attribute for ALetterSequence
    syn int ALetterSequence.size();

    // Attributes for BLetterSequence
    inh int BLetterSequence.inhSize();
    syn boolean BLetterSequence.isValid();

    // Attributes for CLetterSequence
    inh int CLetterSequence.inhSize();
    syn boolean CLetterSequence.isValid();
}
```

JastAdd Behavior File *Equations*

```
aspect LetterSequenceEq {
    // Equations for ALetterSequence
    eq ASingleChar.size() = 1;
    eq AListChar.size() = getSeq().size() + 1;

    // Equations for BSingleChar
    eq BSingleChar.isValid() = inhSize() == 1;

    // Equations for BListChar
    eq BListChar.getSeq().inhSize() = inhSize() - 1;
    eq BListChar.isValid() = getSeq().isValid();

    // Equations for CSingleChar
    eq CSingleChar.isValid() = inhSize() == 1;

    // Equations for CListChar
    eq CListChar.getSeq().inhSize() = inhSize() - 1;
    eq CListChar.isValid() = getSeq().isValid();

    // Equations for ValidSequence
    eq ValidSequence.getBSequence().inhSize() = getASequence().size();
    eq ValidSequence.getCSequence().inhSize() = getASequence().size();
}
```

Embedding Java code

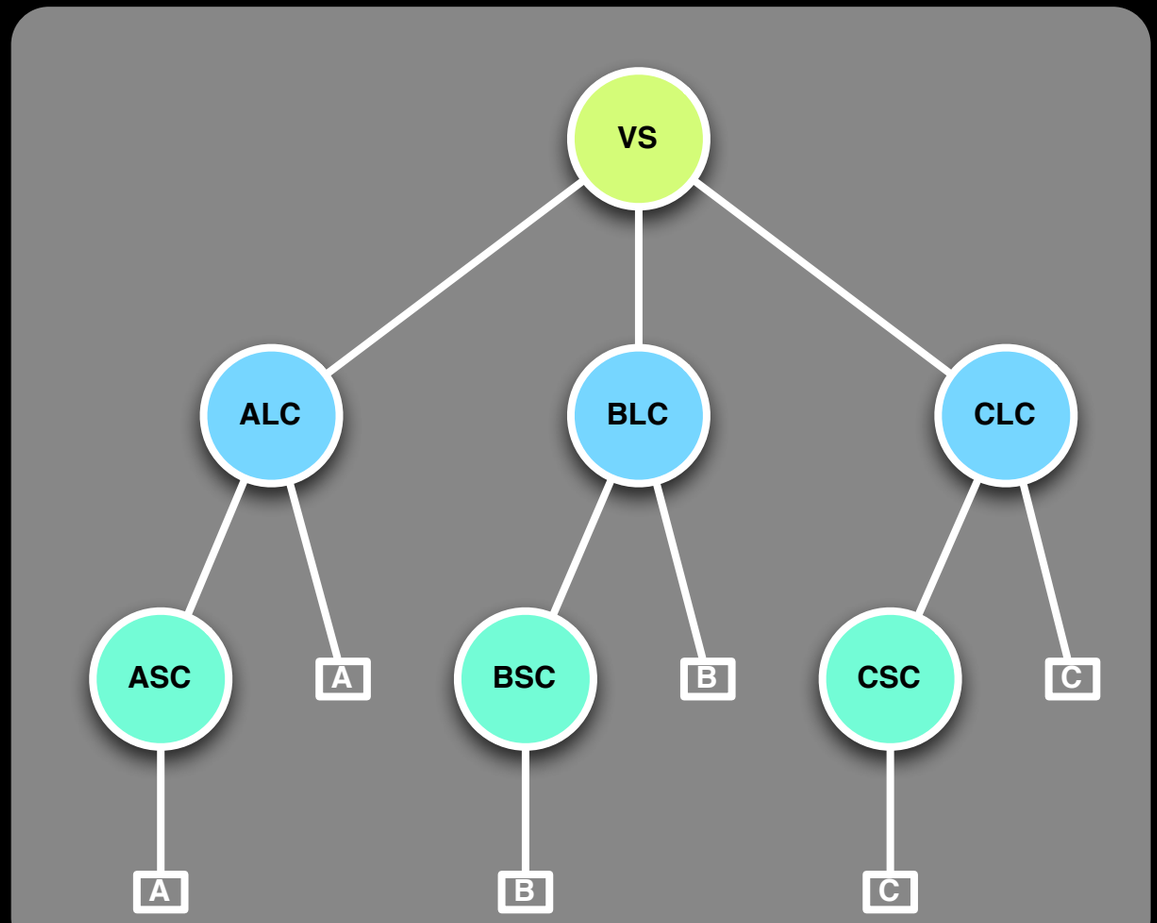
```
aspect LetterSequenceHelper {
    // Helper methods
    public boolean ValidSequence.isValid() {
        if(getASequence() == null
           || getBSequence() == null
           || getCSequence() == null)
            return false;

        return getBSequence().isValid()
               && getCSequence().isValid();
    }
}
```

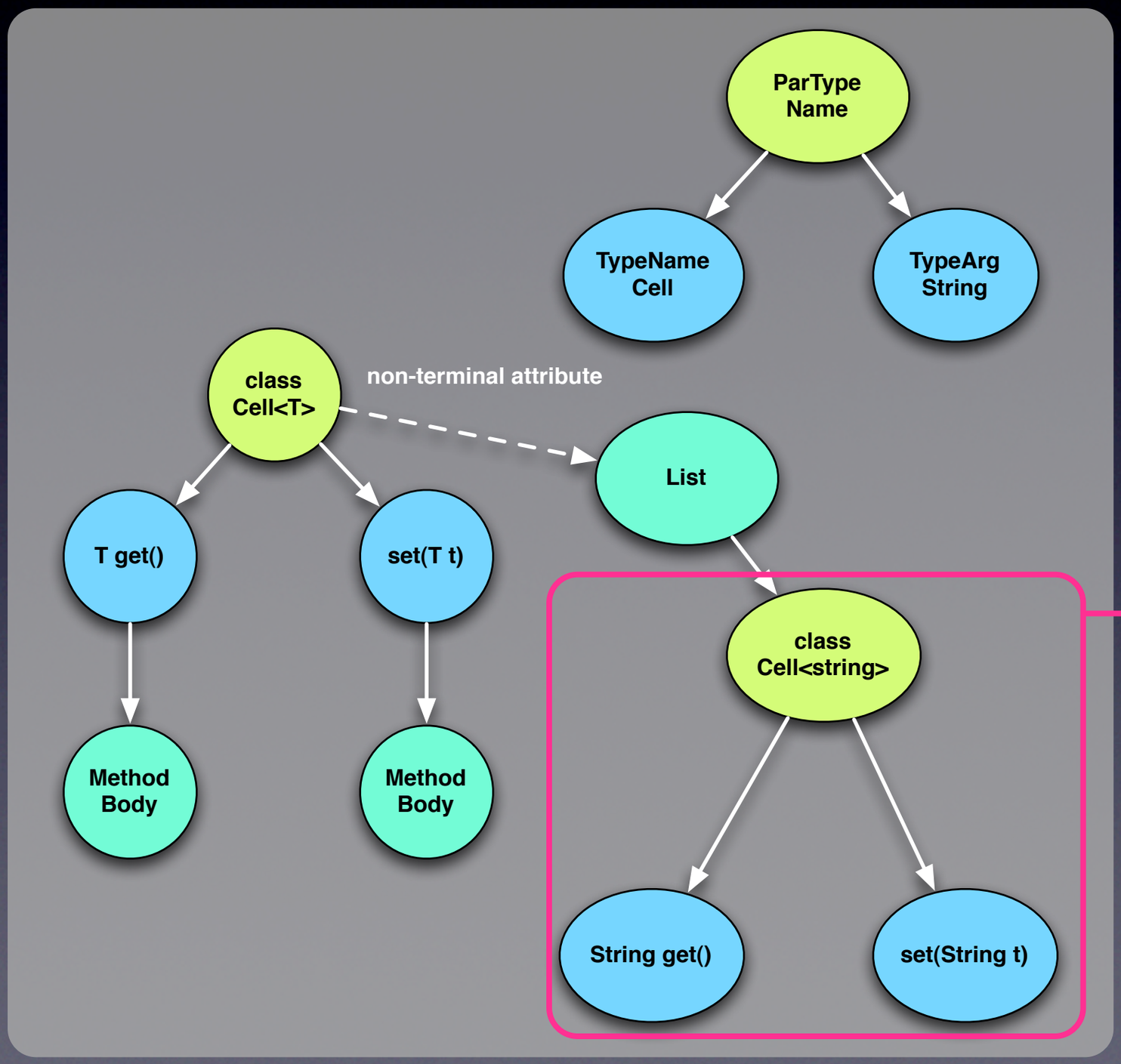
Testing

```
// Test "aabbcc"
public void testValidLonger() {
    // Parser usually does construction
    ValidSequence vs = new ValidSequence(
        new AListChar(
            new ASingleChar(new A()),
            new A()),
        new BListChar(
            new BSingleChar(new B()),
            new B()),
        new CListChar(
            new CSingleChar(new C()),
            new C()));

    assertTrue(vs.getBSequence().isValid());
    assertTrue(vs.getCSequence().isValid());
    assertTrue(vs.isValid());
}
```



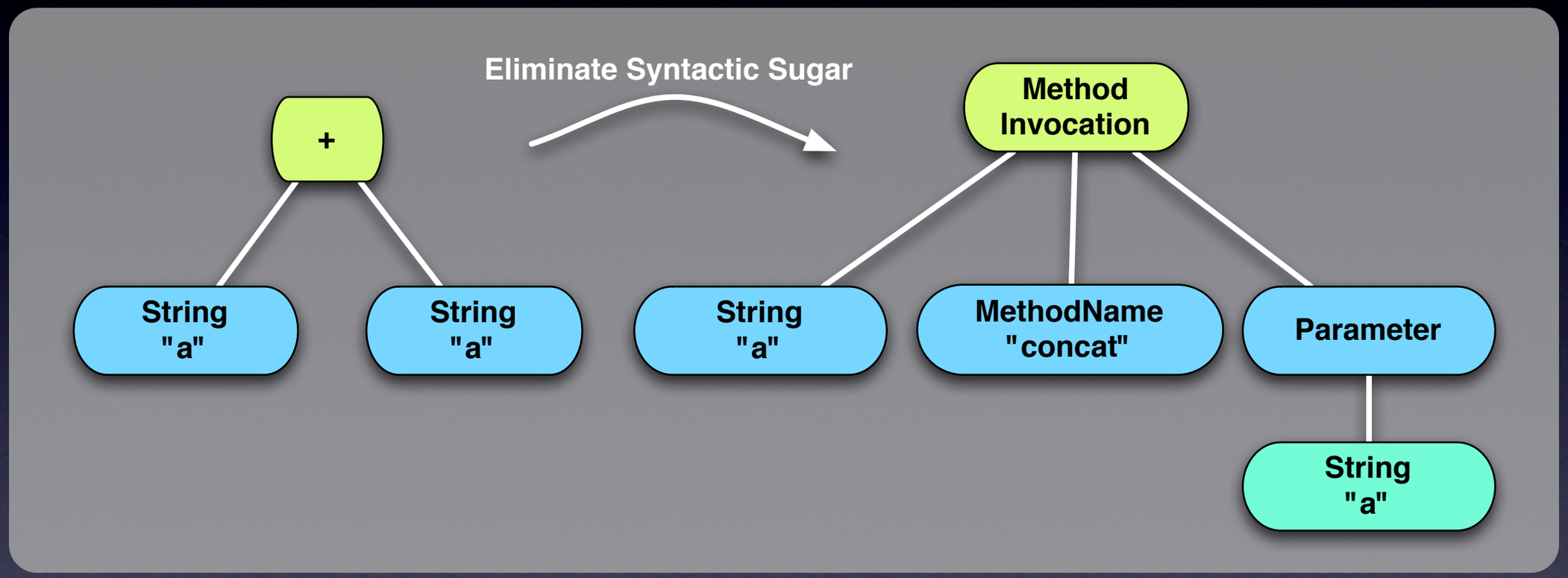
Non-terminal Attributes



Non-terminal attributes allow subtrees to be created on-the-fly as necessary to represent new structure

We want the *normal* representation of a class i.e. class SomeName. But there is no explicit declaration of it.
So, we construct one using non-terminal attributes

Rewritable Reference Attributed Grammars



- Rewritten AST reflects semantics rather than syntax
- Simplifies other computations like code generation
- Brings *consistency* and *symmetry* to AST

Aspect Orientation

Not only supports addition but
also allows for refining equations

```
aspect A {
    void C.m() { ... }
}

aspect B {
    refine A void C.m() { // similar to overriding
        ...
        A.C.m(); // similar to call to super
        ...
    }
}
```

Pumping Lemma

For any context-free grammar G, there is an integer K, depending on G, such that any string generated by G which has length greater than K can be written in the form $uvxyz$ s.t.

1. $|vy| \geq 1$
2. $|vxy| \leq K$

and uv^nxy^nz is in the language generated by G for all $n \geq 0$

Context Free Grammar

$$G = (V, \Sigma, R, S)$$

1. V is a finite set of non-terminal characters.
2. Σ is a finite set of terminals.
3. S is the start variable.
4. R is a relation from V to $(V \bigcup \Sigma)^*$ s.t. $\exists w \in (V \bigcup \Sigma)^* : (S, w) \in R$.

Expression Problem

- Can data model & set of operations over it be extended without modifying existing code?
- Two axes of extensibility
  - *Data-centered Expression*
    - Subclassing
  - *Operation-centered Expression*
    - Visitor Pattern

K in Maude

- A way to define and interpret programming languages
- Uses the Maude parser
- Interpretation done using rewriting of a continuation structure
- Can add multiple analysis over the same grammar *but one at a time*