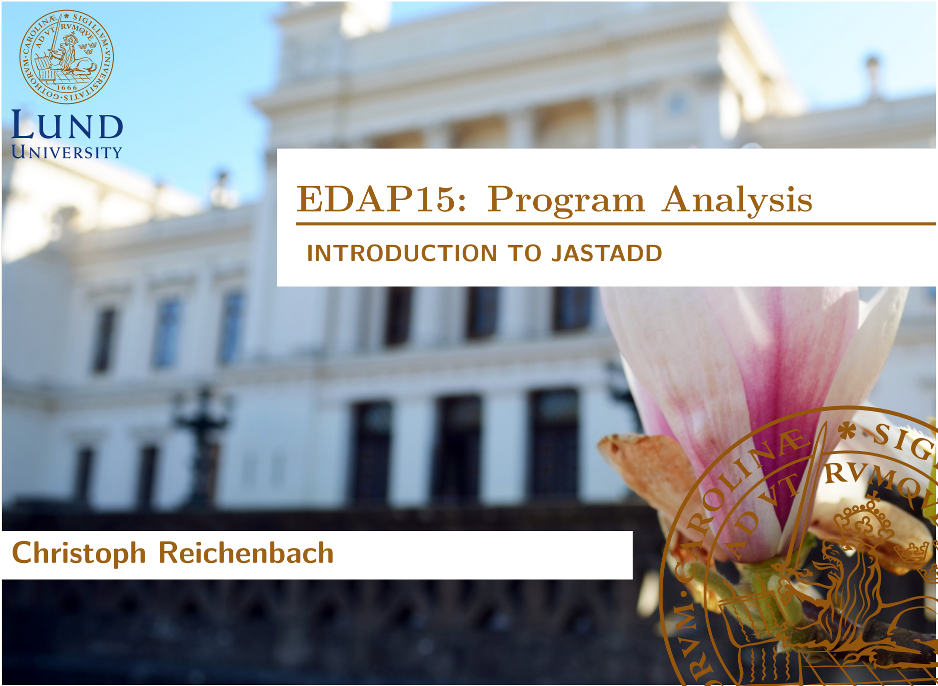


LUND UNIVERSITY

EDAP15: Program Analysis

INTRODUCTION TO JASTADD

Christoph Reichenbach

EDAN65: Compilers

Lectures on Abstract grammars, Reference Attribute Grammars, and JastAdd

Görel Hedin

Revised: 2023-09-05

Adapted for EDAP15: Program Analysis

Christoph Reichenbach

Revised: 2023-11-27

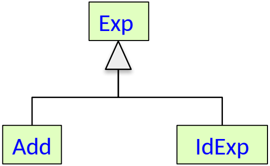
Abstract grammars

Abstract grammar vs. OO model

Abstract grammar	**OO model**	**Other terminology used (algebraic datatypes)**
nonterminal	superclass	type, sort
production	subclass	constructor, operator

Abstract grammar

```
Add: Exp -> Exp Exp
IdExp: Exp -> ID
```

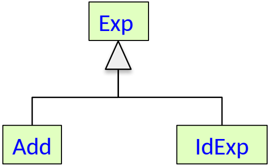


A canonical abstract grammar corresponds to a two-level class hierarchy!

Example Java implementation

Abstract grammar

Add: Exp -> Exp Exp
IdExp: Exp -> ID



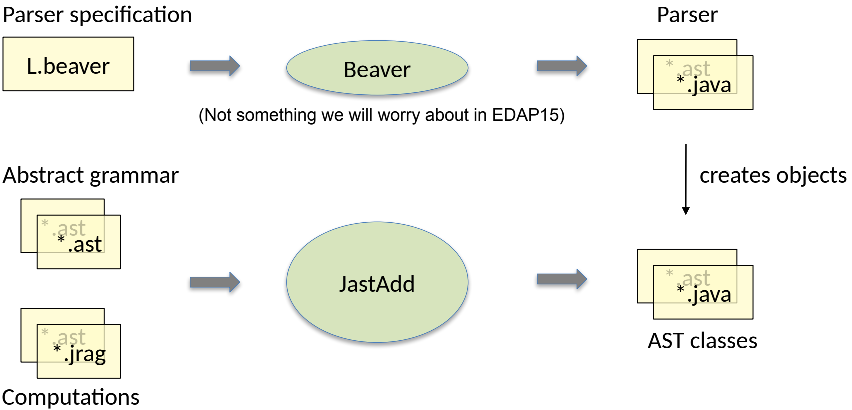

```
abstract class Exp {
}
class Add extends Exp {
  Exp exp1, exp2;
}
class IdExp extends Exp {
  String ID;
}
```

JastAdd

- A compiler generation tool. Generates Java code.
- Supports ASTs and modular computations on ASTs.
- JastAdd: "Just add computations to the ast"
- Independent of the parser used.
- Developed at LTH, see http://jastadd.org

JastAdd

- A compiler generation tool. Generates Java code.
- Supports ASTs and modular computations on ASTs.
- JastAdd: "Just add computations to the ast"
- Independent of the parser used.
- Developed at LTH, see http://jastadd.org



JastAdd abstract grammars

[abstract] *Class* [: *Superclass*] ::= *RightHandSide*;

JastAdd abstract grammars

[abstract] *Class* [: *Superclass*] ::= *RightHandSide*;

```
Program ::= Stmt*;
abstract Stmt;
Assignment : Stmt ::= IdExpr Expr;
IfStmt : Stmt ::= Expr Then:Stmt [Else:Stmt];
abstract Expr;
IdExpr : Expr ::= <ID:String>;
IntExpr : Expr ::= <INT:String>;
BinExpr : Expr ::= Left:Expr Right:Expr;
Add : BinExpr;
```

JastAdd abstract grammars

[abstract] *Class* [: *Superclass*] ::= *RightHandSide*;

```
Program ::= Stmt*;
abstract Stmt;
Assignment : Stmt ::= IdExpr Expr;
IfStmt : Stmt ::= Expr Then:Stmt [Else:Stmt];
abstract Expr;
IdExpr : Expr ::= <ID:String>;
IntExpr : Expr ::= <INT:String>;
BinExpr : Expr ::= Left:Expr Right:Expr;
Add : BinExpr;
```

Compared to canonical abstract grammars:

- Classes instead of nonterminals and productions
- Classes can be abstract (like in Java)
- Arbitrarily deep inheritance hierarchy (not just two levels)
- Support for *optional*, *list*, and *token* components
- Components can be named
- Right-hand side can be inherited from superclass (see BinExpr).
- No parentheses! You need to name all node classes in the AST.

Generated Java API, ordinary components

```
abstract Stmt;
WhileStmt : Stmt ::= Cond:Expr Stmt;
```

Generated Java API, ordinary components

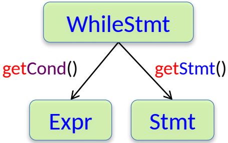
```
abstract Stmt;
WhileStmt : Stmt ::= Cond:Expr Stmt;
```

```
abstract class Stmt extends ASTNode {}

class WhileStmt extends Stmt {
  Expr getCond();
  Stmt getStmt();
}
```

Generated Java API, ordinary components

```
abstract Stmt;
WhileStmt : Stmt ::= Cond:Expr Stmt;
```

```
abstract class Stmt extends ASTNode {}

class WhileStmt extends Stmt {
  Expr getCond();
  Stmt getStmt();
}
```

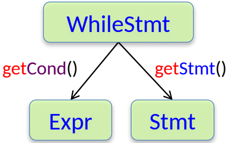


Example AST

Generated Java API, ordinary components

```
abstract Stmt;
WhileStmt : Stmt ::= Cond:Expr Stmt;
```

```
abstract class Stmt extends ASTNode {}

class WhileStmt extends Stmt {
  Expr getCond();
  Stmt getStmt();
}
```



Example AST

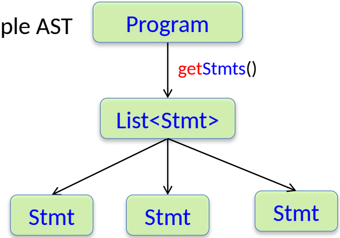
- A general class ASTNode is used as implicit superclass.
- A traversal API with *get* methods is generated.
- If component names are given, they are used in the API (getCond).
- Otherwise the type names are used (getStmt).

Generated Java API, lists

```
Program ::= Stmt*;
```

```
class Program extends ASTNode {
  int getNumStmt();   // 0 if empty
  Stmt getStmt(int i); // numbered from 0
  List<Stmt> getStmts(); // iterator
}
```

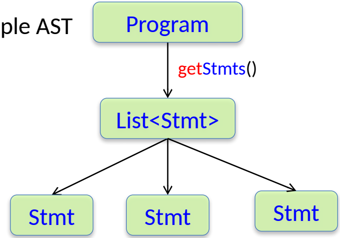
Example AST



Generated Java API, lists

```
Program ::= Stmt*;
```

Example AST




```
class Program extends ASTNode {
  int getNumStmt();   // 0 if empty
  Stmt getStmt(int i); // numbered from 0
  List<Stmt> getStmts(); // iterator
}
```

The list is represented by a List object that can be used as an iterator:

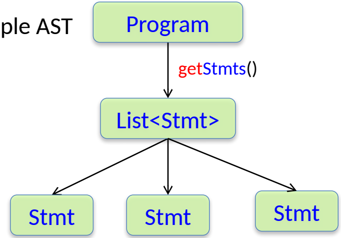
```
Program p = ...;
for (Stmt s : p.getStmts()) {
  ...
}
```

Generated Java API, lists

```
Program ::= Stmt*;
```

```
class Program extends ASTNode {
  int getNumStmt();   // 0 if empty
  Stmt getStmt(int i); // numbered from 0
  List<Stmt> getStmts(); // iterator
}
```

Example AST



The list is represented by a List object that can be used as an iterator:

```
Program p = ...;
for (Stmt s : p.getStmts()) {
  ...
}
```

Or access a specific statement:

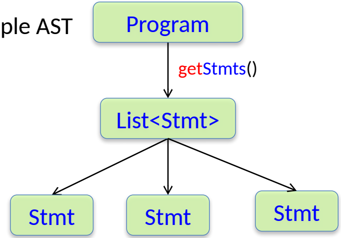
```
Program p = ...;
if (p.getNumStmt() >= 1) {
  Stmt s = p.getStmt(0);
  ...
}
```

Generated Java API, lists

```
Program ::= Stmt*;
```

```
class Program extends ASTNode {
  int getNumStmt();   // 0 if empty
  Stmt getStmt(int i); // numbered from 0
  List<Stmt> getStmts(); // iterator
}
```

Example AST



The list is represented by a List object that can be used as an iterator:

```
Program p = ...;
for (Stmt s : p.getStmts()) {
  ...
}
```

Or access a specific statement:

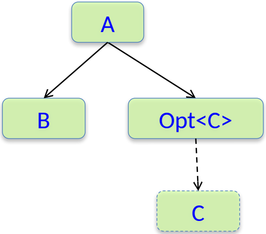
```
Program p = ...;
if (p.getNumStmt() >= 1) {
  Stmt s = p.getStmt(0);
  ...
}
```

Note! List is a JastAdd-specific class (like ASTNode and Opt). It is *not* the same class as java.util.List.

Generated Java API, optionals

A ::= B [C];

```
class A extends ASTNode {
  B getB();
  boolean hasC();
  C getC();      //Exception if not hasC()
}
```



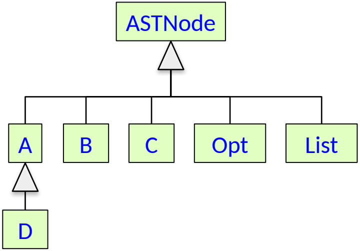
Example AST

- The traversal API includes a *has* method for the optional component.

General traversal

Abstract grammar

```
A ::= B [C];
B ::= ...;
C ::= ...;
D : A ::= ...;
```



Will stop also at Opt and List nodes.
Can be used for general traversal of the children of a node.

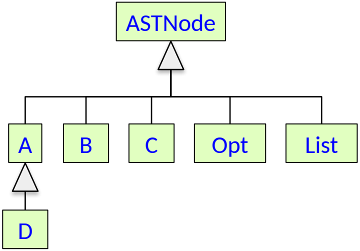
```
class ASTNode {
  Iterable astChildren(); //Iterator for the children
}
```

```
void ASTNode.m() {
  ...
  for (ASTNode child : astChildren()) { ... }
}
```

Low-level traversal API

Abstract grammar

```
A ::= B [C];
B ::= ...;
C ::= ...;
D : A ::= ...;
```



Will stop also at Opt and List nodes.
This low-level API is not recommended.
Use iterator or high-level API instead – much more readable.

```
class ASTNode {
  int getNumChild();
  ASTNode getChild(int i);
  ASTNode getParent(); // null for the root
}
```

Defining an abstract grammar

This is object-oriented modeling!

- What kinds of objects are there in the AST?
  E.g., Program, WhileStmt, Assignment, Add, …

- What are the generalized concepts (abstract classes)?
  E.g., Statement, Expression, …

- What are the components of an object?
  E.g., an Assignment has an Identifier and an Expression…

Defining an abstract grammar

This is object-oriented modeling!

- What kinds of objects are there in the AST?
  E.g., Program, WhileStmt, Assignment, Add, …

- What are the generalized concepts (abstract classes)?
  E.g., Statement, Expression, …

- What are the components of an object?
  E.g., an Assignment has an Identifier and an Expression…

```
Program ::= …;
abstract Statement;
abstract Expression;
abstract Declaration;
WhileStmt : Statement ::= …;
Assignment : Statement ::= Identifier Expression;
…
```

Defining an abstract grammar

This is object-oriented modeling!

- What kinds of objects are there in the AST?
  E.g., Program, WhileStmt, Assignment, Add, …

- What are the generalized concepts (abstract classes)?
  E.g., Statement, Expression, …

- What are the components of an object?
  E.g., an Assignment has an Identifier and an Expression…

```
Program ::= …;
abstract Statement;
abstract Expression;
abstract Declaration;
WhileStmt : Statement ::= …;
Assignment : Statement ::= Identifier Expression;
…
```

Teal syntax (used in the labs):

```
Program ::= …;
abstract Stmt;
abstract Expr;
abstract Decl;
WhileStmt : Stmt ::= …;
AssignStmt : Stmt ::= IdUse Expr;
…
```

Summary questions: Abstract syntax trees

- What is the correspondence between an abstract grammar and an object-oriented model?
- Orientation about JastAdd abstract grammars, traversal API.
- What are properties of a good abstract grammar?

The Expression Problem

	Hard to add computation	Easy to add computation
Easy to add language construct	**OOP**	**OOP with Static Aspects***
Hard to add language construct		**OOP with Visitor Pattern** **FP**

* Not Java, no separate compilation.

Ordinary programming

Example: Printing an AST

```
class Exp {
  abstract void print();
}
class Add extends Exp {
  Exp e1, e2;
  void print() {
    e1.print();
    System.out.print("+");
    e2.print();
  }
}
class IntExp extends Exp {
  int value;
  void print() {
    System.out.print(value);
  }
}
...
```

Ordinary programming

Example: Printing an AST

```
class Exp {
  abstract void print();
}
class Add extends Exp {
  Exp e1, e2;
  void print() {
    e1.print();
    System.out.print("+");
    e2.print();
  }
}
class IntExp extends Exp {
  int value;
  void print() {
    System.out.print(value);
  }
}
...
```

Pros:

- Straightforward code
- Modular extension in the language dimension (subclasses)

Cons:

- No modular extension in the operation dimension – all classes need to be modified.
- Tangled code – many different concerns in the same class.

Visitor solution

Example: Printing an AST

```
class Exp {
}
class Add extends Exp {
  Exp e1, e2;
  void accept(Visitor v) {
    v.visit(this);
  }
}
class IntExp extends Exp {
  int value;
  void accept(Visitor v) {
    v.visit(this);
  }
}
...
```

```
class UnparserVisitor implements Visitor {
  void visit(Add node) {
    node.e1.accept(this);
    System.out.print("+");
    node.e2.accept(this);
  }
  void visit(IntExpr node) {
    System.out.print(node.value);
  }
}
```

Visitor solution

Example: Printing an AST

```
class Exp {
}
class Add extends Exp {
  Exp e1, e2;
  void accept(Visitor v) {
    v.visit(this);
  }
}
class IntExp extends Exp {
  int value;
  void accept(Visitor v) {
    v.visit(this);
  }
}
...
```

```
class UnparserVisitor implements Visitor {
  void visit(Add node) {
    node.e1.accept(this);
    System.out.print("+");
    node.e2.accept(this);
  }
  void visit(IntExpr node) {
    System.out.print(node.value);
  }
}
```

Pros:

- Modular extension in the operation dimension (add new visitor).

Cons:

- Boilerplate code needed (accept and visit methods).
- Limited modular extensibility in the language dimension. Needs lots of boilerplate.

Static Aspect-Oriented Programming

Example: Printing an AST

```
class Exp {
}
class Add extends Exp {
  Exp e1, e2;
}
class IntExp extends Exp {
  int value;
}
...
```

```
aspect Unparser {
  abstract void Exp.print();
  void Add.print() {
    e1.print();
    System.out.print("+");
    e2.print();
  }
  void IntExp.print() {
    System.out.print(value);
  }
}
```

Static Aspect-Oriented Programming

Example: Printing an AST

```
class Exp {
}
class Add extends Exp {
  Exp e1, e2;
}
class IntExp extends Exp {
  int value;
}
...
```

```
aspect Unparser {
  abstract void Exp.print();
  void Add.print() {
    e1.print();
    System.out.print("+");
    e2.print();
  }
  void IntExp.print() {
    System.out.print(value);
  }
}
```

Pros:

- Straightforward code.
- Modular extension in the operation dimension (can be added in aspect).
- Modular extension in the language dimension (add new subclass, add operation code for those constructs in aspect).

Cons:

- Cannot use plain Java. Need more advanced language like AspectJ or JastAdd.
- Typically no separate compilation of modules. (Modules woven before compilation)

Inter-type declarations

The key construct in static AOP

```
class C {
    int x;
}
```

```
class D {
}
```

```
aspect A {
  T C.m() {
    x = ...;
    ...
  }
  int D.f = 3;
}
```

inter-type declared method

inter-type declared field

Inter-type declarations

The key construct in static AOP

```
class C {
    int x;
}
```

```
class D {
}
```

```
aspect A {
  T C.m() {        <-- inter-type declared method
    x = ...;
    ...
  }
  int D.f = 3;     <-- inter-type declared field
}
```

is equivalent to:

```
class C {
    int x;
    T m() {
        x = ...;
        ...
    }
}
```

```
class D {
    int f = 3;
}
```

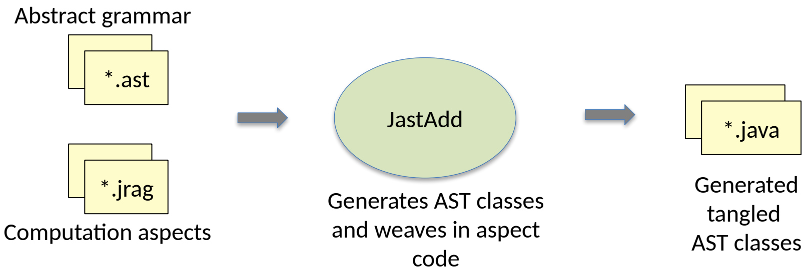
Recall: Dealing with the expression problem

- Edit the AST classes (i.e., actually not solving the problem)
  - Non-modular, non-compositional.
  - **It is always a VERY BAD IDEA to edit generated code!**
  - Sometimes used anyway in industry.
- Visitors: an OO design pattern.
  - Modularize operations through double dispatch.
  - Not full modularization, not composition.
  - Supported by many parser generators.
  - Reasonably useful, commonly used in industry.
- Static Aspect-Oriented Programming (AOP)
  - Also known as *inter-type declarations* (ITDs) or *introduction*
  - Use new language constructs (aspects) to factor out code.
  - Solves the expression problem in a nice simple way.
  - The drawback: you need a new language: AspectJ, JastAdd, …
- Advanced language constructs
  - Use more advanced language constructs: virtual classes in gbeta, traits in Scala, typeclasses in Haskell, …
  - Drawbacks: Much more complex than static AOP. You need an advanced language. Not much practical experience (so far).

This lecture: Static AOP

Static AOP in JastAdd

Static AOP in JastAdd



Example aspect: expression evaluation

Abstract grammar

```
abstract Exp;
abstract BinExp : Exp ::= Left:Exp Right:Exp;
Add : BinExp;
Sub : BinExp;
IntExp : Exp ::= <INT:String>;
```

Example aspect: expression evaluation

Abstract grammar

```
abstract Exp;
abstract BinExp : Exp ::= Left:Exp Right:Exp;
Add : BinExp;
Sub : BinExp;
IntExp : Exp ::= <INT:String>;
```

Aspect

```
aspect Evaluator {
  abstract int Exp.value();
  int Add.value() { return getLeft().value() + getRight().value(); }
  int Sub.value() { return getLeft().value() – getRight().value(); }
  int IntExp.value() { return String.parseInt(getINT()); }
}
```

Inter-type declarations: The value methods will be woven into the classes (Expr, Add, Sub, IntExpr).
Inter-type declarations are also known as *introductions*.

Another example: unparsing

Abstract grammar

```
abstract Exp;
abstract BinExp : Exp ::= Left:Exp Right:Exp;
Add : BinExp;
Sub : BinExp;
IntExp : Exp ::= <INT:String>;
```

Another example: unparsing

Abstract grammar

```
abstract Exp;
abstract BinExp : Exp ::= Left:Exp Right:Exp;
Add : BinExp;
Sub : BinExp;
IntExp : Exp ::= <INT:String>;
```

Aspect

```
aspect Unparser {
  abstract void Exp.unparse(Stream s, String indent);
  void BinExp.unparse(Stream s, String indent) {
    getLeft().unparse(s,indent);
    s.print(operatorString());
    getRight().unparse(s,indent);
  }
  abstract String BinExp.operatorString();
  String Add.operatorString() { return "+"; }
  String Sub.operatorString() { return "-"; }
  void IntExp.unparse(Stream s, String indent) { s.print(getINT()); }
}
```

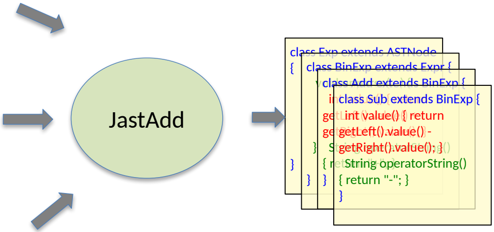
Weaving the classes in JastAdd

toy.ast

```
abstract Exp;
abstract BinExp : Exp ::= Left:Exp Right:Exp;
Add : BinExp;
Sub : BinExp;
IntExp : Exp ::= <INT:String>;
```

Evaluator.jrag

```
aspect Evaluator {
  abstract int Exp.value();
  int Add.value() { return getLeft().value() + getRight().value(); }
  int Sub.value() { return getLeft().value() – getRight().value(); }
  int IntExp.value() { return String.parseInt(getINT()); }
}
```



Unparser.jrag

```
aspect Unparser {
  abstract void Exp.unparse(Stream s, String indent);
  void BinExp.unparse(Stream s, String indent) {
    getLeft().unparse(s,ind);
    s.print(operatorString());
    getRight().unparse(s,ind);
  }
  abstract BinExp.operatorString();
  String Add.operatorString() { return "+"; }
  String Sub.operatorString() { return "-"}
  void IntExp.unparse(Stream s, String indent) { s.print(getINT()); }
}
```

Tangled generated code

Untangled source code

Features that can be factored out to aspects in JastAdd

- Methods
- Instance variables
- "implements" clauses
- "import" clauses
- attribute grammars (see later lecture)

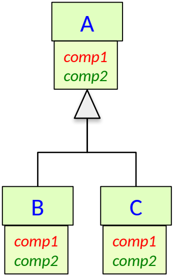
Static aspects vs Visitors

	Static aspects	Visitors
What can be factored out from AST classes?	instance variables methods implements clauses	only methods
Type safety?	full type precision	Casts may be needed, depending on framework
Method parameters	any number	only one
Ease of use?	Very simple	Clumsy, boilerplate code needed.
Arbitrary composition of modules?	Yes	No – you can extend a visitor, but then you need factories to create them. And you cannot not easily combine two extensions.
Separate compilation?	Not for JastAdd or AspectJ.	Yes
Mainstream OO language?	No – you need JastAdd, AspectJ, or similar	Yes, use Java or any other OO language.

Recall: The expression problem

How add both classes and computations in a modular way?

Ordinary OO

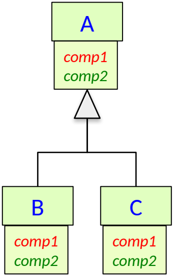


Classes can be added modularly, but not computations.
Simple code.

Recall: The expression problem

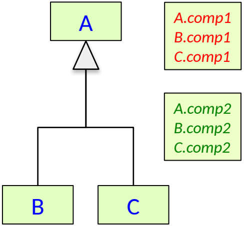
How add both classes and computations in a modular way?

Ordinary OO



Classes can be added modularly, but not computations. Simple code.

Aspects with inter-type declarations

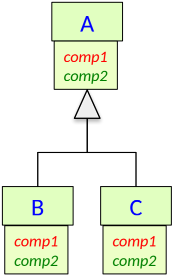


Fully modular. Simple code.

Recall: The expression problem

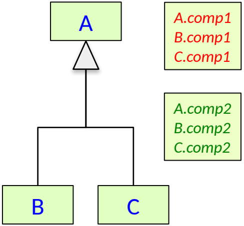
How add both classes and computations in a modular way?

Ordinary OO



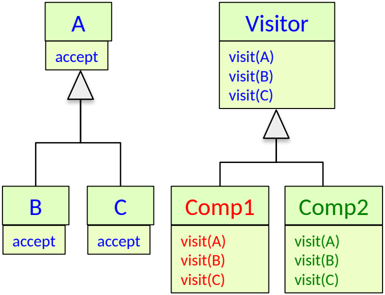
Classes can be added modularly, but not computations. Simple code.

Aspects with inter-type declarations



Fully modular. Simple code.

The Visitor design pattern



Computations can be added modularly, but not classes. Complex code.

Full Aspect-Oriented Programming

Full AOP = Static AOP + Dynamic AOP

Full Aspect-Oriented Programming

Full AOP	=	Static AOP	+	Dynamic AOP
		inter-type declarations		advice joinpoints pointcuts
		Modularize declarations.		Modularize instrumentation.

Full Aspect-Oriented Programming

- JastAdd supports only a small part of AOP, namely *static* AOP with inter-type declarations.

- Aspect-oriented programming is a wider concept that usually focuses on *dynamic* behavior: a general code instrumentation technique:
  - A joinpoint is a point at runtime where advice code can be added.
  - A pointcut is a set of joinpoints defined at compile-time, and that can be described in a simple way, e.g.,
    - all calls to a method m()
    - all accesses of a variable v
  - Advice is code you can specify in an aspect and that can be added at joinpoints, either after, before, or around the joinpoint.
  - Example applications:
    - Add logging of method calls in an aspect (instead of adding print statements all over your code)
    - Add synchronization code to basic code that is unsynchronized

Computations on the AST

IMPERATIVE COMPUTATIONS

DECLARATIVE COMPUTATIONS

Computations on the AST

IMPERATIVE COMPUTATIONS

- Computations that "do" something. (have an effect)
  - Modify state
  - Output to files
- Useful for
  - Interpretation
  - Printing error messages
  - Output of code
- Technique:
  - Methods, modularized with
    - Inter-type declarations, or
    - Visitors

DECLARATIVE COMPUTATIONS

- Computations of properties (of nodes in the AST)
  - No side-effects
- Useful for computing
  - Name bindings
  - Types of expressions
  - Error information
- Technique
  - Attribute grammars

Properties of AST nodes

INTRINSIC PROPERTIES

- Given directly by the AST:
  - children
  - token values (like the name of an identifier)

DERIVED PROPERTIES

- Computed using the AST. E.g.,
  - the type of an expression
  - the decl of an identifier
  - the code of a method
  - ...
- Can be defined using attribute grammars

Example derived properties

Does this method have any compile-time errors?

```
int gcd2(int a, int b) {
  if (b == 0) {
    return a;
  }
  return gcd2(b, a % b);
}
```

What is the type of this expression?

What is the declaration of this b?

Attribute grammars:
Express these properties as *attributes* of AST nodes.
Define the attributes by simple directed *equations*.
The equations can be solved automatically.

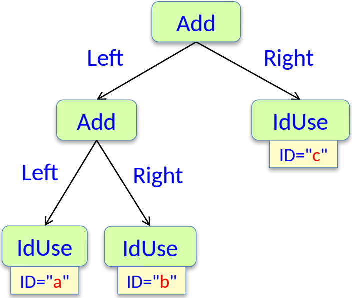
Abstract grammar

defines the *structure* of ASTs

Abstract grammar:

```
abstract Exp;
Add : Exp ::= Left:Exp
Right:Exp;
IdUse : Exp ::= <ID:String>;
```

Example AST for "a + b + c"
(an *instance* of the abstract grammar)



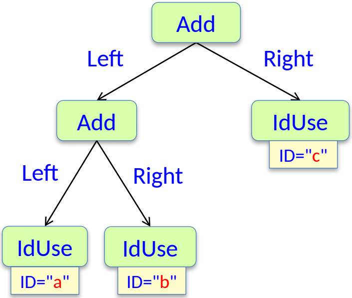
Abstract grammar

defines the *structure* of ASTs

Abstract grammar:

```
abstract Exp;
Add : Exp ::= Left:Exp
Right:Exp;
IdUse : Exp ::= <ID:String>;
```

Example AST for "a + b + c"
(an *instance* of the abstract grammar)



The terminal symbols (like ID) are **intrinsic** attributes – constructed when building the AST. They are not defined by equations.

Also the children can be seen as intrinsic attributes.

Attribute grammars

extends abstract grammars with attributes

Abstract grammar:

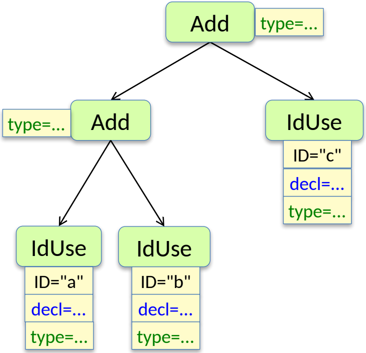
```
abstract Exp;
Add : Exp ::= Left:Exp
Right:Exp;
IdUse : Exp ::= <ID:String>;
```

Attribute grammar modules:

```
syn IdDecl IdUse.decl() = ...;
```

```
syn Type Exp.type();
eq  Add.type() = ...;
eq  IdUse.type() = ...;
```

Example AST for "a + b + c"
(an *instance* of the abstract grammar)



Each declared attribute ...

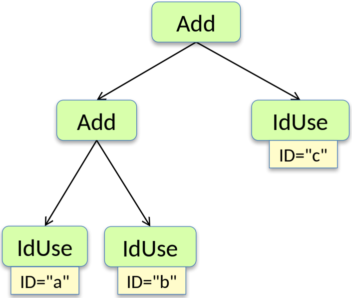
... will have instances in the AST

Attributes and equations

Abstract grammar:

```
abstract Exp;
Add : Exp ::= Left:Exp
Right:Exp;
IdUse : Exp ::= <ID:String>;
```

Example AST for "a + b + c"
(an *instance* of the abstract grammar)



Think of attributes as "fields" in the tree nodes.

```
syn Type ASTClass.attribute();
```

Each equation *defines* an attribute in terms of other attributes in the tree.

```
eq definedAttribute = function of other
attributes;
```

An *evaluator* computes the values of the attributes (solves the equation system).
Think of the equations as "methods" called by the evaluator.

Attribute mechanisms

Intrinsic* – given value when the AST is constructed (no equation)

Synthesized* – the equation is in the same node as the attribute

Inherited* – the equation is in an ancestor

Broadcasting* – the equation holds for a complete subtree

Reference* – the attribute can be a reference to an AST node.

Parameterized – the attribute can have parameters

NTA – the attribute is a "nonterminal" (a fresh node or subtree)

Collection – the attribute is defined by a set of contributions, instead of by an equation.

Circular – the attribute may depend on itself (solved using fixed-point iteration)

*** Treated in this lecture**

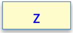
Introduction to attribute grammars

Simple example

attributes and equations

AST node

attribute

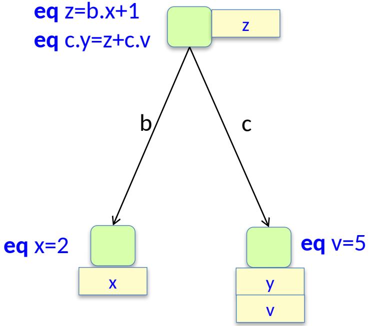


equation:

eq a0 = f(a1, ..., an)

defined attribute

function of other attributes



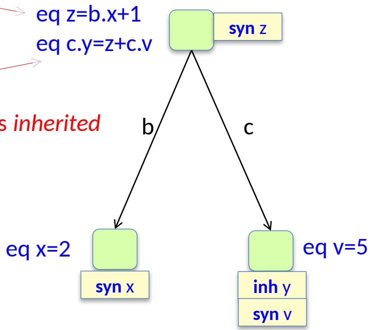
What is the value of y?
Solve the equation system!
(Easy! Just use substitution.)

Simple example

synthesized and inherited attributes

defines attribute in the node – the attribute is *synthesized*

defines attribute in the child – the attribute is *inherited*



Donald Knuth introduced attribute grammars in 1968.
The term "inherited" is *not* related to inheritance in object-orientation.
Both terms originated during the 1960s.

Simple example

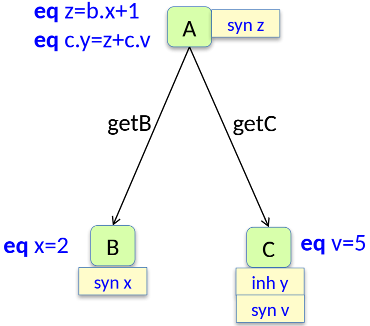
declaring attributes and equations in a (JastAdd) grammar

Abstract grammar:

```
A ::= B C;
B;
C;
```

Attribute grammar module:

```
aspect SomeAttributes {
  syn int A.z();
  syn int B.x();
  syn int C.v();
  inh int C.y();
  eq  A.z() = getB().x()+1;
  eq  A.getC().y() = z() +
getC().v();
  eq  B.x() = 2;
  eq  C.v() = 5;
}
```



uses inter-type declarations for attributes and equations

Note! The grammar is declarative. The order of the equations is irrelevant.
JastAdd solves the equation system automatically.

Shorthands and alternative forms

equation in attribute declaration, method body syntax

Canonical form:

```
syn int A.z();
eq  A.z() = getB().x()+1;
```

Alternative shorthand form with equation directly in attribute declaration:

```
syn int A.z() = getB().x()+1;
```

Alternative form with method body syntax:

```
syn int A.z() {
  return getB().x()+1;
}
```

Equations must be observationally pure

(free from externally visible side effects)

```
syn int A.z() {
  return getB().x()+1;
}
```

Equations must be observationally pure

(free from externally visible side effects)

Which of these examples are ok?

```
syn int A.z() {
  return getB().x()+1;
}
```

```
syn int A.z() {
  int r = 0;
  r = getB().x()+1;
  return r;
}
```

```
int B.f = 0;
syn int B.x() {
  f++;
  return f;
}
syn int B.y() {
  f++;
  return f;
}
```

Equations must be observationally pure

(free from externally visible side effects)

Which of these examples are ok?

OK – no side effects

```
syn int A.z() {
  return getB().x()+1;
}
```

OK – side effects, but only local

```
syn int A.z() {
  int r = 0;
  r = getB().x()+1;
  return r;
}
```

Not OK – visible side effects!

```
int B.f = 0;
syn int B.x() {
  f++;
  return f;
}
syn int B.y() {
  f++;
  return f;
}
```

Will give different results if evaluated more than once, and depending on order of evaluation.

Warning! JastAdd does not check observational purity

Abstract grammar:

```
A ::= B C;
B ::= D;
C ::= D;
D;
```

Well-formed attribute grammar

An AG is ***well-formed*** if there is
exactly one defining equation for each attribute in any AST.

Abstract grammar:

```
A ::= B C;
B ::= D;
C ::= D;
D;
```

Well-formed attribute grammar

An AG is ***well-formed*** if there is
exactly one defining equation for each attribute in any AST.
Which of these are well-formed?

```
syn int A.x();
```

```
inh int B.y();
eq A.getB().y() = 5;
```

```
syn int A.x();
eq A.x() = 3;
```

```
inh int D.z();
eq B.getD().z() = 7;
```

```
syn int A.x();
eq A.x() = 3;
eq A.x() = 17;
```

```
inh int D.z();
eq B.getD().z() = 7;
eq C.getD().z() = 11;
```

Abstract grammar:

```
A ::= B C;
B ::= D;
C ::= D;
D;
```

Well-formed attribute grammar

An AG is ***well-formed*** if there is
exactly one defining equation for each attribute in any AST.
Which of these are well-formed?

Not well formed

```
syn int A.x();
```

Well formed

```
inh int B.y();
eq A.getB().y() = 5;
```

Well formed

```
syn int A.x();
eq A.x() = 3;
```

Not well formed

```
inh int D.z();
eq B.getD().z() = 7;
```

Not well formed

```
syn int A.x();
eq A.x() = 3;
eq A.x() = 17;
```

Well formed

```
inh int D.z();
eq B.getD().z() = 7;
eq C.getD().z() = 11;
```

JastAdd checks well-formedness at generation time

Abstract grammar:

```
A ::= B C;
B ::= D;
C ::= D;
D;
```

Well-defined attribute grammar

*An AG is **well-defined*** if it is well-formed, and there is a unique solution that can be computed.

Abstract grammar:

```
A ::= B C;
B ::= D;
C ::= D;
D;
```

Well-defined attribute grammar

An AG is ***well-defined*** if it is well-formed, and there is a unique solution that can be computed. Which of these are well-defined?

```
syn int A.x() = 3;
```

```
syn int A.y() {
  int x = 0;
  while (true)
    x++;
  return x;
}
```

```
syn int A.s() = t();
syn int A.t() = s();
```

Abstract grammar:

```
A ::= B C;
B ::= D;
C ::= D;
D;
```

Well-defined attribute grammar

An AG is ***well-defined*** if it is well-formed, and there is a unique solution that can be computed. Which of these are well-defined?

```
syn int A.x() = 3;
```

Well defined

```
syn int A.y() {
  int x = 0;
  while (true)
    x++;
  return x;
}
```

Not well defined.
Computation does not terminate.

```
syn int A.s() = t();
syn int A.t() = s();
```

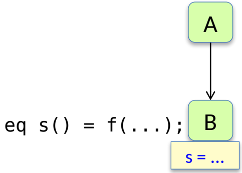
Not well defined. Circular definition.

JastAdd checks circularity dynamically, at evaluation time.
JastAdd supports well-defined circular attributes by a special construction, see later lecture.

Synthesized attributes

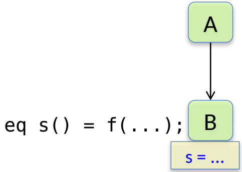
Synthesized attributes

Synthesized attribute:
The equation is in the *same* node as the attribute.



Synthesized attributes

Synthesized attribute:
The equation is in the *same* node as the attribute.



JastAdd syntax:

```
syn T B.s() = f(...);
```

this code is in the context of B

For properties that depend on information in the node (or its children).

Typically used for propagating information *upwards* in the tree.

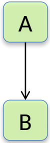
Synthesized attributes

simple example

```
A ::=
B;
B;
```

```
syn int B.s() = 3;
```

Draw the attribute and its value!

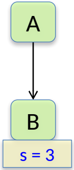


Synthesized attributes

simple example

```
A ::=
B;
B;
```

```
syn int B.s() = 3;
```



Or equivalently, write the declaration and equation separately.

```
syn int B.s();
eq  B.s() = 3;
```

Or equivalently, write the equation as a method body:

```
syn int B.s() {
  return 3;
}
```

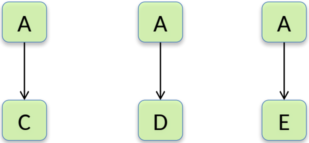
Nota bene!
The method body must be observationally pure.

Synthesized attributes

subtypes can have different equations

```
A ::= B;
abstract B;
C : B;
D : B;
E : D;
```

Three different ASTs.
Draw the attributes and their values!



Different subclasses can have different equations.

```
syn int B.s();
eq C.s() = 4;
eq D.s() = 5;
eq E.s() = 6;
```

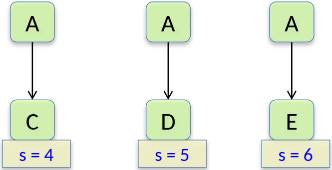
Synthesized attributes

subtypes can have different equations

```
A ::= B;
abstract B;
C : B;
D : B;
E : D;
```

Different subclasses can have different equations.

```
syn int B.s();
eq C.s() = 4;
eq D.s() = 5;
eq E.s() = 6;
```

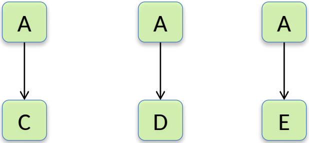


Synthesized attributes

an equation in the supertype can be overridden

```
A ::= B;
abstract B;
C : B;
D : B;
E : D;
```

```
syn int B.s() = 11;
eq E.s() = 17;
```

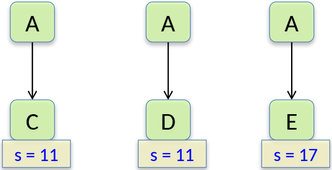


Synthesized attributes

an equation in the supertype can be overridden

```
A ::= B;
abstract B;
C : B;
D : B;
E : D;
```

```
syn int B.s() = 11;
eq E.s() = 17;
```



The equation in B holds for all subtypes, except for those overriding the equation.

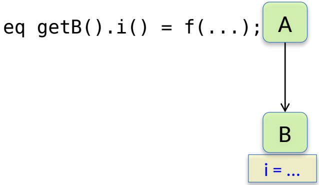
A synthesized attribute is similar to a side-effect free method, but:

- its value is cached (memoized) the first time it is accessed.
- circularity is checked at runtime (results in exception)

Inherited attributes

Inherited attributes

Inherited attribute:
The equation is in an ancestor



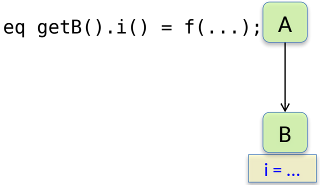
Inherited attributes

Inherited attribute:
The equation is in an ancestor

JastAdd syntax:

```
inh T B.s();
eq  A.getB().i() = f(...);
```

this code is in the context of A



For computing a property that depends on the *context* of the node.

Typically used for propagating information *downwards* in the tree.

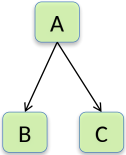
Inherited attributes

simple example

```
A ::= B C;
B;
C;
```

```
inh int B.i();
eq A.getB().i() = 2;
```

Draw the attribute and its value!

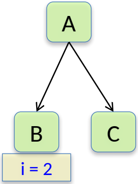


Inherited attributes

simple example

```
A ::= B C;
B;
C;
```

```
inh int B.i();
eq A.getB().i() = 2;
```

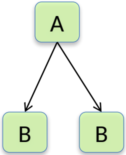


Inherited attributes

different equations for different children

```
A ::= Left:B Right:B;
B;
```

Draw the attributes and their values!



The parent can specify different equations for its different children.

```
inh int B.i();
eq A.getLeft().i() = 2;
eq A.getRight().i() = 3;
```

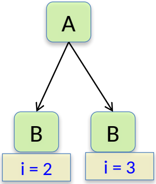
Inherited attributes

different equations for different children

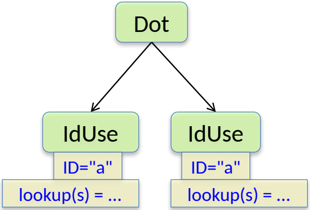
```
A ::= Left:B Right:B;
B;
```

The parent can specify different equations for its different children.

```
inh int B.i();
eq A.getLeft().i() = 2;
eq A.getRight().i() = 3;
```



This is useful, for example, when defining scope rules for qualified access. The lookup attributes should have different values for the different IdUses.



Inherited attributes

a subtype can override an equation

```
A ::= Left:B Right:B;
B;
A2 : A;
```

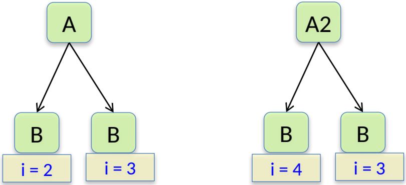
```
inh int B.i();
eq A.getLeft().i() = 2;
eq A.getRight().i() = 3;
eq A2.getLeft().i() = 4;
```

Inherited attributes

a subtype can override an equation

```
A ::= Left:B Right:B;
B;
A2 : A;
```

```
inh int B.i();
eq A.getLeft().i() = 2;
eq A.getRight().i() = 3;
eq A2.getLeft().i() = 4;
```

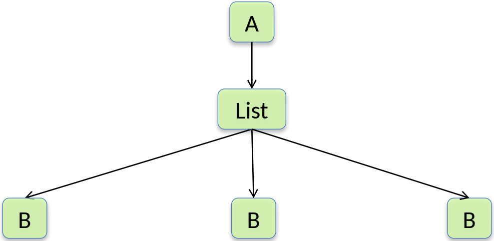


Inherited attributes
a list child has an index

```
A ::= B*;
B;
```

For list children, an index can be used in the equation

```
eq A.getB(int index).x() = (index+1) * (index+1);
inh int B.x();
```



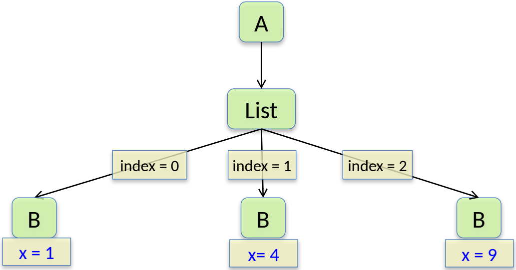
Inherited attributes

a list child has an index

```
A ::= B*;
B;
```

For list children, an index can be used in the equation

```
eq A.getB(int index).x() = (index+1) * (index+1);
inh int B.x();
```



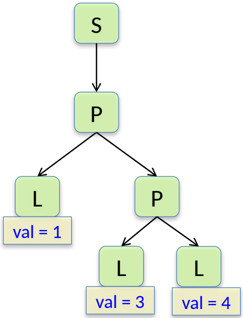
This is useful, for example, when defining name analysis with declare-before-use semantics.

Example: Fractions

Goal

Compute *f* for each L, where *f* is L's fraction of the sum of all *val* attributes.

```
S ::= N;
abstract N;
P : N ::= Left:N Right:N;
L : N ::= <val:int>;
```

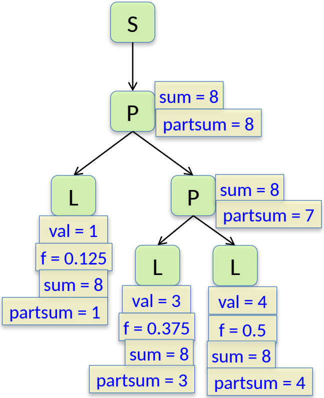


Goal

Compute *f* for each L, where *f* is L's fraction of the sum of all *val* attributes.

```
S ::= N;
abstract N;
P : N ::= Left:N Right:N;
L : N ::= <val:int>;
```

```
syn float L.f() = getval()/sum();
inh int N.sum();
eq  int P.getLeft().sum() = sum();
eq  int P.getRight().sum() = sum();
eq  int S.getN().sum() =
getN().partsum();
syn int N.partsum();
eq  P.partsum() =
       getLeft().partsum() +
       getRight().partsum();
eq  L.partsum() = getval();
```

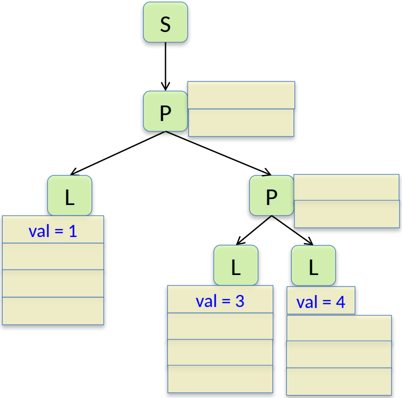


Demand evaluation and memoization

```
S ::= N;
abstract N;
P : N ::= Left:N
Right:N;
L : N ::= <val:int>;
```

```
S root = ...;
L leaf1 = root...; L leaf2 = root...;
System.out.println(leaf1.f());
System.out.println(leaf2.f());
```

```
syn float L.f() = sum()/getval();
inh int N.sum();
eq  int P.getLeft().sum() = sum();
eq  int P.getRight().sum() = sum();
eq  int S.getN().sum() =
getN().partsum();
syn int N.partsum();
eq  P.partsum() =
       getLeft().partsum() +
       getRight().partsum();
eq  L.partsum() = getval();
```



Recursive evaluation algorithm with memoization

If not cached
  find the equation
  compute its right-hand side
  cache the value
fi
Return the cached value

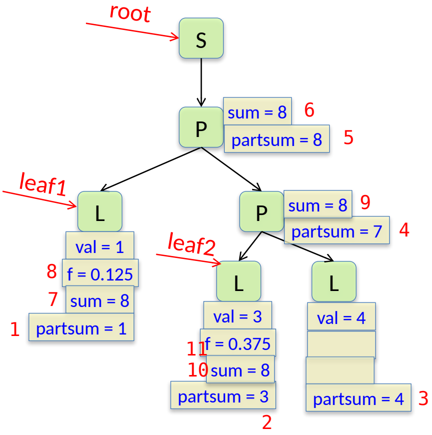
```
S ::= N;
abstract N;
P : N ::= Left:N
Right:N;
L : N ::= <val:int>;
```

```
S root = ...;
L leaf1 = root...; L leaf2 = root...;
System.out.println(leaf1.f());
System.out.println(leaf2.f());
```

```
syn float L.f() = sum()/getval();
inh int N.sum();
eq  int P.getLeft().sum() = sum();
eq  int P.getRight().sum() = sum();
eq  int S.getN().sum() =
getN().partsum();
syn int N.partsum();
eq  P.partsum() =
       getLeft().partsum() +
       getRight().partsum();
eq  L.partsum() = getval();
```

Recursive evaluation algorithm
with memoization

```
If not cached
  find the equation
  compute its right-hand side
  cache the value
fi
Return the cached value
```



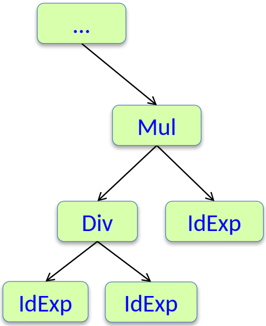
memoization order

Summary questions

- What is an attribute grammar?
- What is an intrinsic attribute?
- What is an externally visible side-effect? Why are they not allowed in the equations?
- What is a synthesized attribute?
- What is an inherited attribute?
- What is the difference between a declarative and an imperative specification?
- What is demand evaluation?
- Why are attributes cached?

You can now do all of Assignment 3.
But it is recommended to do the 7B quiz first!

Example computations on an AST



Name analysis: find the declaration of an identifier

Type analysis: compute the type of an expression

Expression evaluation: compute the value of a constant expression

Code generation: compute an intermediate code representation of the program

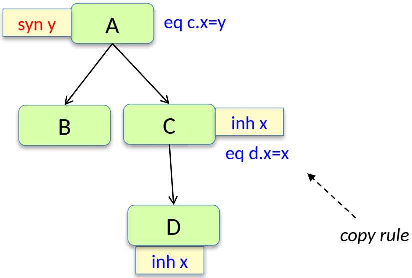
Unparsing: compute a text representation of the program

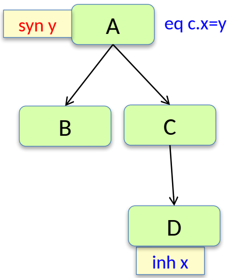
Broadcasting

Broadcasting of inherited attributes

Traditional AG:
Equation for inherited attribute
must be in the immediate parent.
Leads to "**copy rules**".

JastAdd:
Equation for inherited attribute
is "broadcasted" to complete subtree.
No "copy rules" are needed.





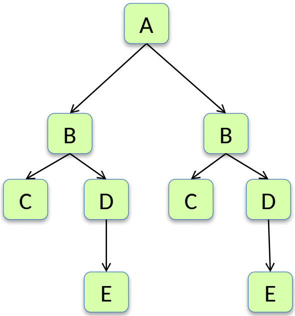
Most AG systems have some shorthand to avoid copy rules

Inherited attributes

broadcasting: equations hold for complete subtrees

```
A ::= Left:B Right:B;
B ::= C D;
C;
D ::= E;
E;
```

Draw the attributes and their values!



The equations hold for the complete children subtrees.

```
eq A.getLeft().i() = 2;
eq A.getRight().i() = 3;
inh int C.i();
inh int E.i();
```

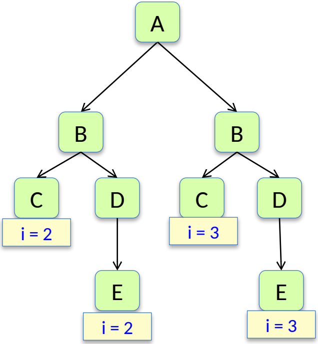
Inherited attributes

broadcasting: equations hold for complete subtrees

```
A ::= Left:B Right:B;
B ::= C D;
C;
D ::= E;
E;
```

The equations hold for the complete children subtrees.

```
eq A.getLeft().i() = 2;
eq A.getRight().i() = 3;
inh int C.i();
inh int E.i();
```

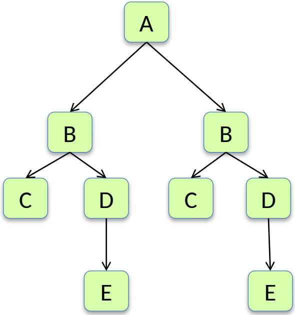


Inherited attributes

broadcasted equation can be overruled in subtree

```
A ::= Left:B Right:B;
B ::= C D;
C;
D ::= E;
E;
```

Draw the attributes and their values!



An equation can be overruled in a subtree.
The nearest equation applies.

```
eq A.getLeft().i() = 2;
eq A.getRight().i() = 3;
eq B.getD().i() = i() + 5;
inh int B.i();
inh int C.i();
inh int E.i();
```

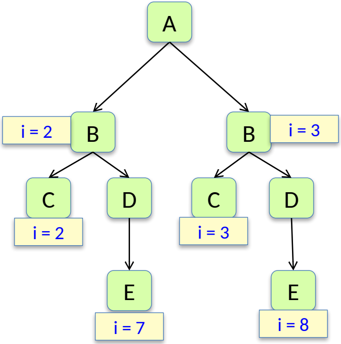
Inherited attributes

broadcasted equation can be overruled in subtree

```
A ::= Left:B Right:B;
B ::= C D;
C;
D ::= E;
E;
```

An equation can be overruled in a subtree.
The nearest equation applies.

```
eq A.getLeft().i() = 2;
eq A.getRight().i() = 3;
eq B.getD().i() = i() + 5;
inh int B.i();
inh int C.i();
inh int E.i();
```

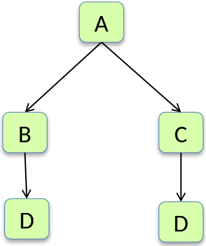


Inherited attributes

shorthand for equation applying to all children

```
A ::= B C;
B ::= D;
C ::= D;
D;
```

Draw the attributes and their values!



The parent can write an equation that applies to *all* children.

```
eq A.getChild().i() = 8;
inh int D.i();
```

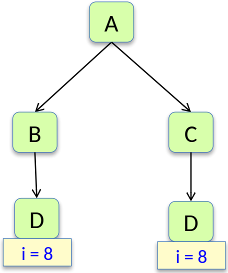
Inherited attributes

shorthand for equation applying to *all* children

```
A ::= B C;
B ::= D;
C ::= D;
D;
```

The parent can write an equation that applies to all children.

```
eq A.getChild().i() = 8;
inh int D.i();
```



This is equivalent to writing an equation for each child:

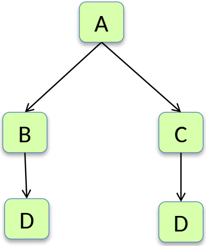
```
eq A.getB().i() = 8;
eq A.getC().i() = 8;
inh int D.i();
```

Inherited attributes

overruling is possible for getChild too

```
A ::= B C;
B ::= D;
C ::= D;
D;
```

```
eq A.getChild().i() = 8;
inh int D.i();
eq B.getD().i() = 5;
```

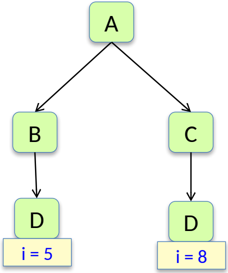


Inherited attributes

overruling is possible for getChild too

```
A ::= B C;
B ::= D;
C ::= D;
D;
```

```
eq A.getChild().i() = 8;
inh int D.i();
eq B.getD().i() = 5;
```

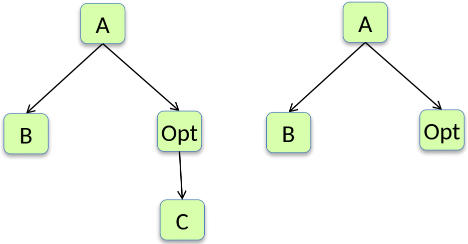


Inherited attributes

defining attributes for optional children

```
A ::= B [C];
B;
C;
```

```
eq A.getC().i() = 4;
inh int C.i();
```

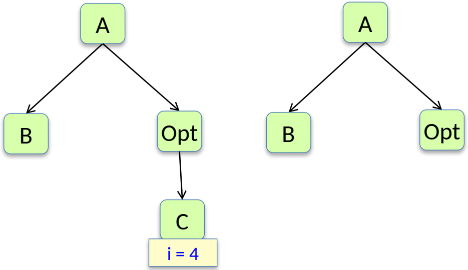


Inherited attributes

defining attributes for optional children

```
A ::= B [C];
B;
C;
```

```
eq A.getC().i() = 4;
inh int C.i();
```



The equation applies if there is a C node.

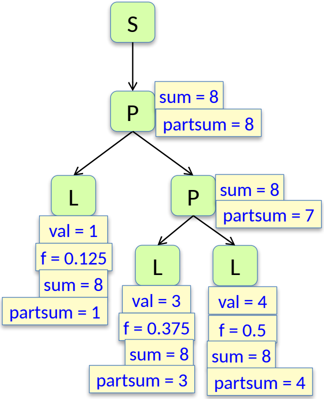
Fractions example revisited

Fractions example

Compute *f* for each L, where *f* is L's fraction of the sum of all *val* attributes.

```
S ::= N;
abstract N;
P : N ::= Left:N Right:N;
L : N ::= <val:int>;
```

```
syn float L.f() = sum()/getval();
inh int N.sum();
eq  int P.getLeft().sum() = sum();
eq  int P.getRight().sum() = sum();
eq  int S.getN().sum() =
getN().partsum();
syn int N.partsum();
eq  P.partsum() =
       getLeft().partsum() +
       getRight().partsum();
eq  L.partsum() = getval();
```

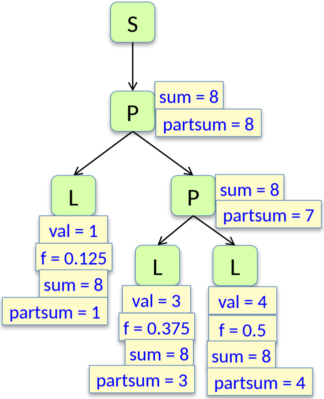


Fractions example

Compute *f* for each L, where *f* is L's fraction of the sum of all *val* attributes.

```
S ::= N;
abstract N;
P : N ::= Left:N Right:N;
L : N ::= <val:int>;
```

```
syn float L.f() = sum()/getval();
inh int N.sum();
eq  int P.getLeft().sum() = sum();
eq  int P.getRight().sum() = sum();
eq  int S.getN().sum() =
getN().partsum();
syn int N.partsum();
eq  P.partsum() =
       getLeft().partsum() +
       getRight().partsum();
eq  L.partsum() = getval();
```



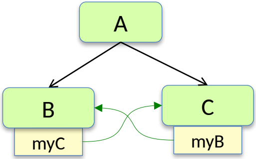
Because of broadcasting, the copy equations are unnecessary.

Reference attributes

Reference attributes

for defining graphs on top of the AST

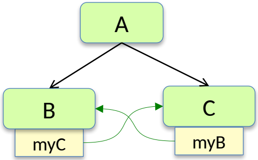
```
A ::= B C;
B;
C;
```



Reference attributes

for defining graphs on top of the AST

```
A ::= B C;
B;
C;
```



Attribute grammar

```
aspect Graph {
  inh C B.myC();
  inh B C.myB();
  eq  A.getB().myC() =
getC();
  eq  A.getC().myB() =
getB();
}
```

Note!
The defined structure is cyclic, but the attribute dependencies are not circular.

Summary questions:
reference attributes, name analysis

- What is broadcasting?
- What is a reference attribute grammar?
- What is a reference attribute?