EDAN65: Compilers, Lecture 07 B

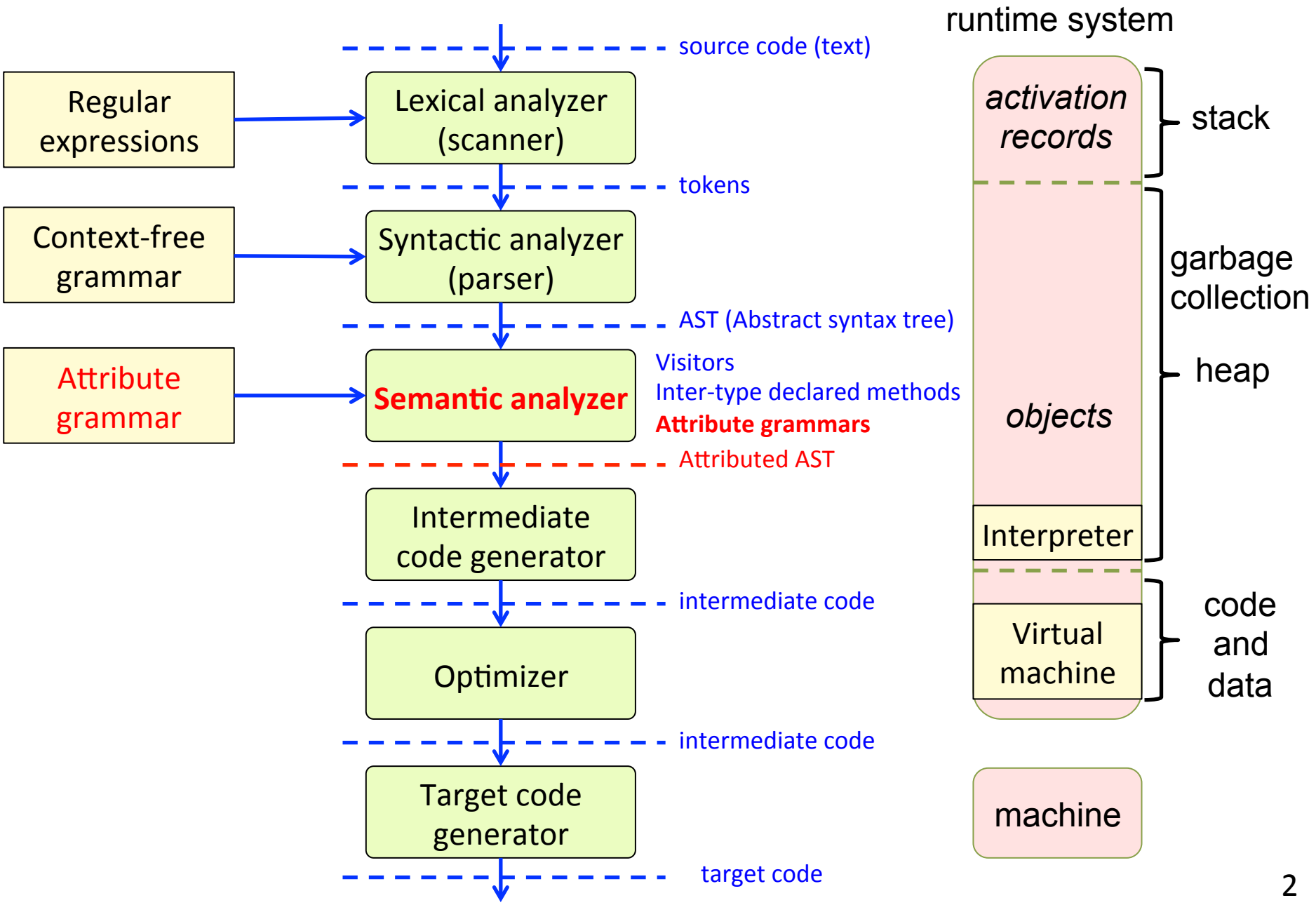
Introduction to Attribute Grammars

synthesized, inherited, broadcasting

Görel Hedin

Revised: 2016-09-19

This lecture



Computations on the AST

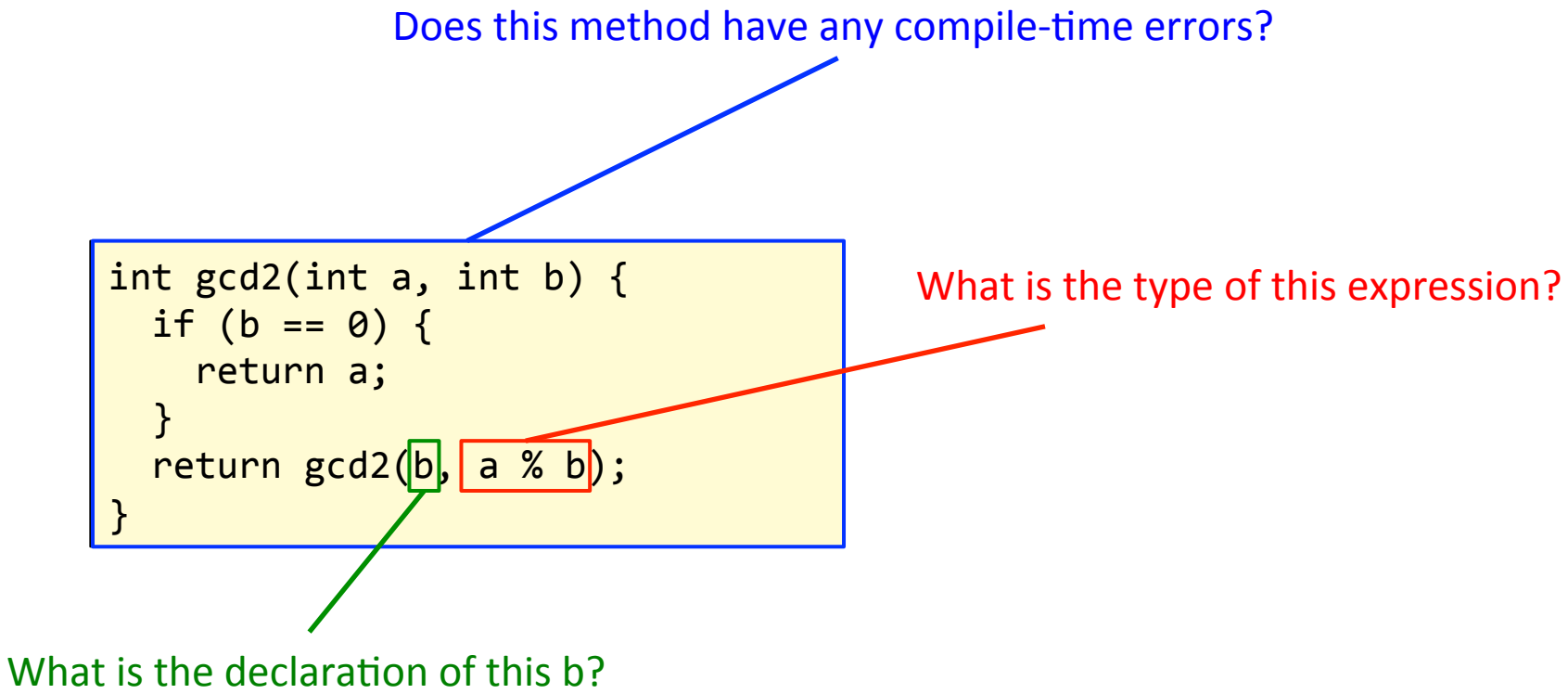
IMPERATIVE COMPUTATIONS

- Define methods that "do" something.
- Side-effects
  - Modify objects
  - Output to files
- Useful for
  - Execution/Interpretation
  - Unparsing
  - Printing error messages
- Technique
  - Inter-type declared methods
  - Visitors

DECLARATIVE COMPUTATIONS

- Define properties of nodes
- No side-effects
- Useful for computing
  - Name bindings
  - Types of expressions
  - Error information
- Technique
  - Attribute grammars

Example properties



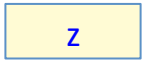
Attribute grammars:
Express these properties as *attributes* of AST nodes.
Define the attributes by simple directed *equations*.
The equations can be solved automatically.

Simple example

attributes and equations

AST node

attribute

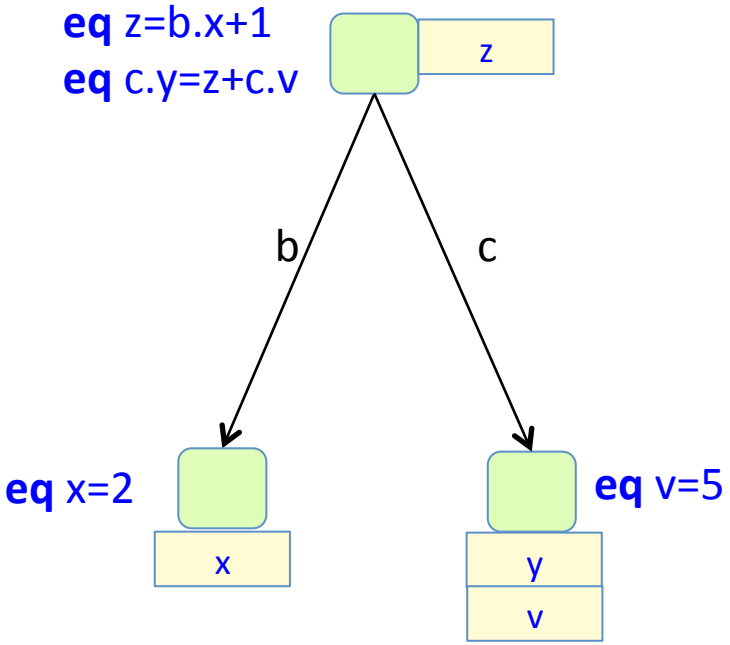


equation:

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

defined attribute

function of other attributes



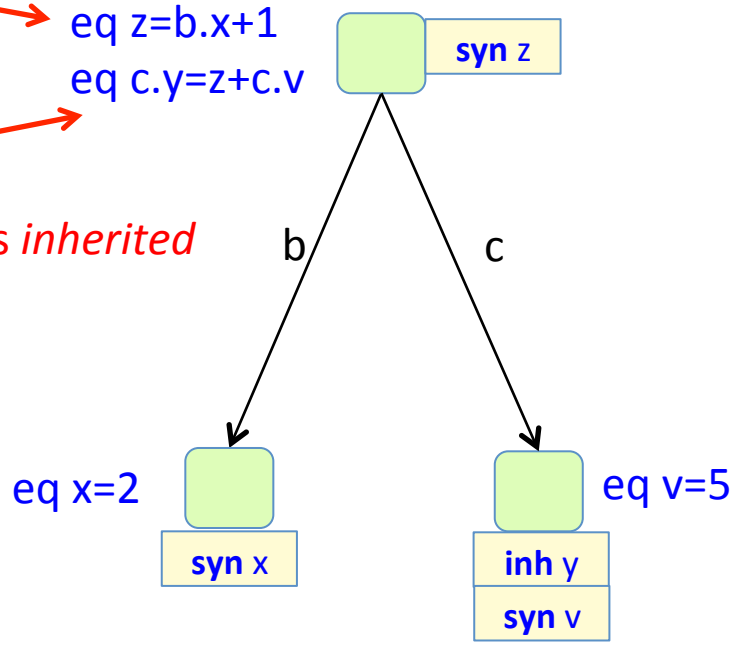
What is the value of y?
Solve the equation system!
(Easy! Just use substitution.)

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

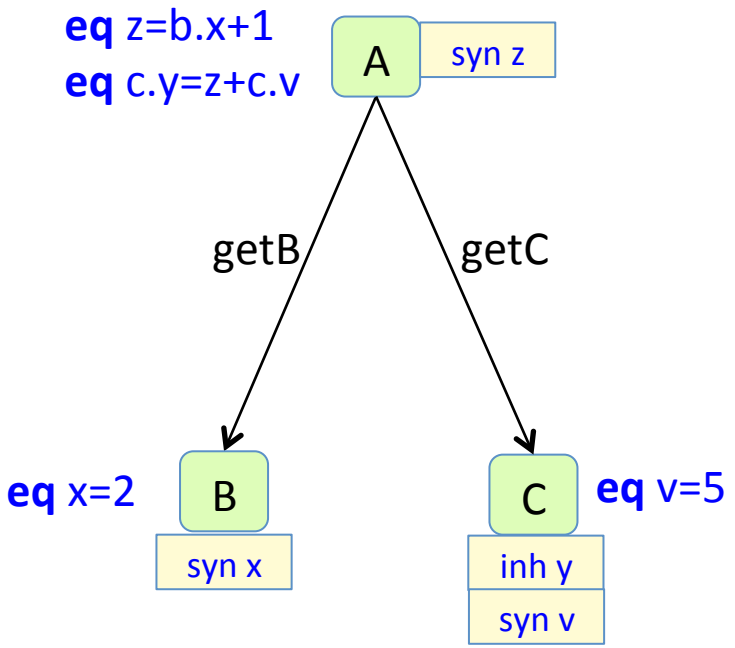
declaring attributes and equations in a (JastAdd) grammar

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().v() = 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.

Some shorthands

These rules:

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

are equivalent to:

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

and we could also use method body syntax:

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

Equations must be free from (externally visible) side effects

While this is formulated as a method, executing it has no side-effects, so this is fine.

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

It is also fine to have assignments to local variables, like this. The effect of changing r is not visible after executing the method.

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

Equations must be free from (externally visible) side effects

What is wrong with this attribute grammar?

```
syn int A.x() = Globals.variable;

syn int B.y() {
  Globals.variable++;
  return 3;
}
```

Equations must be free from (externally visible) side effects

What is wrong with this attribute grammar?

```
syn int A.x() = Globals.variable;

syn int B.y() {
  Globals.variable++;
  return 3;
}
```

Equations are not allowed to change other than local data. If they do, they are not equations.

Warning! JastAdd cannot discover if you have side-effects in your equations! If your definitions rely on global data that is changed, the wrong results will be computed.

Well-formed attribute 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().v() = z() + getC().v();
  eq  B.x() = 2;
  eq  C.v() = 5;
}
```

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

JastAdd checks this at compile time.

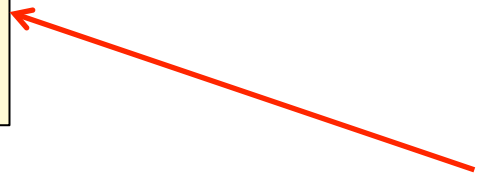
Well-defined attribute grammar

An attribute grammar is *well-defined*,
if it has a computable unique solution for any AST.

An ordinary attribute grammar is well-defined
if it is well-formed and *non-circular*.

Is this attribute grammar well-defined?

```
aspect SomeAttributes {
  syn int A.c() = d();
  syn int A.d() = c();
}
```



Circular attribute grammar. Well-formed, but not well-defined.

JastAdd checks circularity at runtime.

It is possible to allow circular attributes, but they will then
have to be explicitly declared as circular. See later lecture.

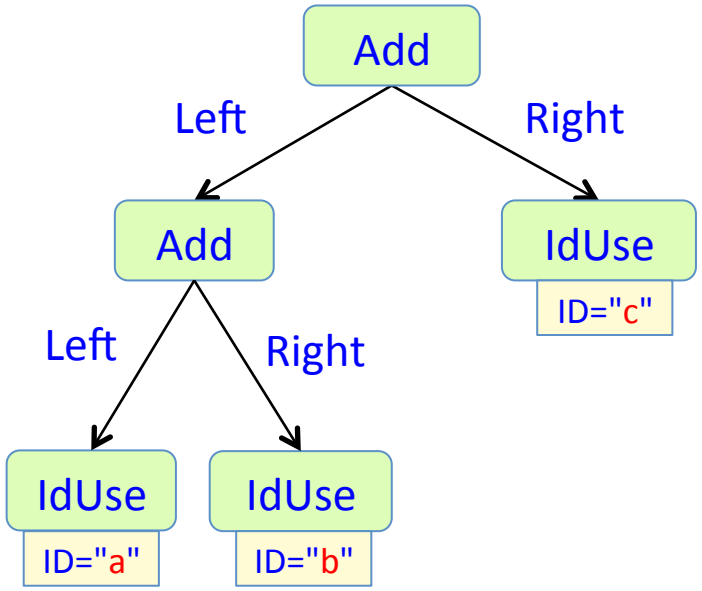
Abstract grammar

defines the *structure* of ASTs

Abstract grammar:

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

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.

Attribute grammars

extends abstract grammars with attributes

Abstract grammar:

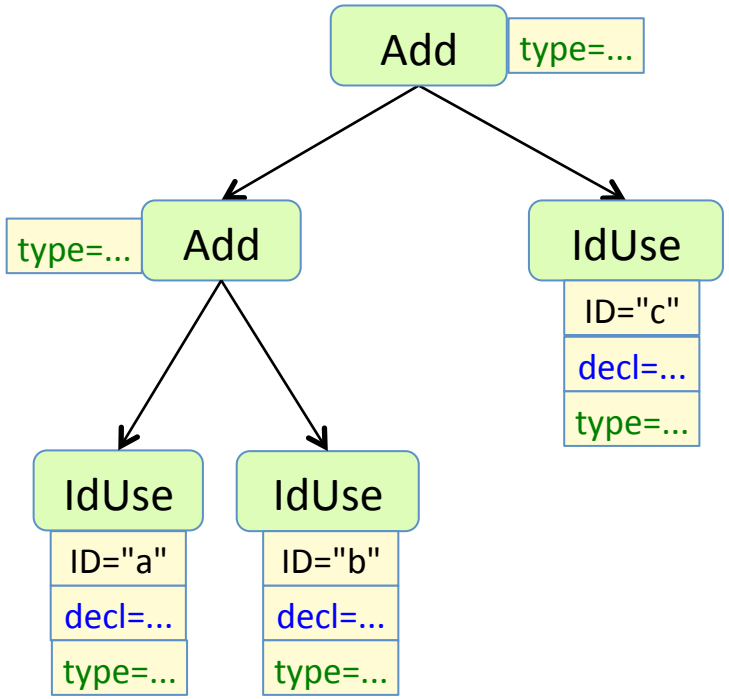
```
abstract Exp;
Add : Exp ::= Left:Exp Right:Exp;
IdUse : Exp ::= <ID>;
```

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 ...

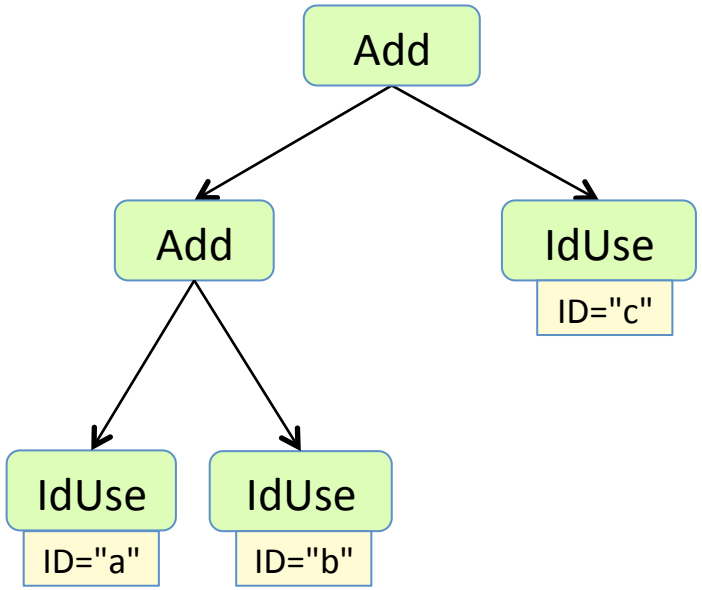
... will have instances in the AST

Attributes and equations

Abstract grammar:

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

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

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)

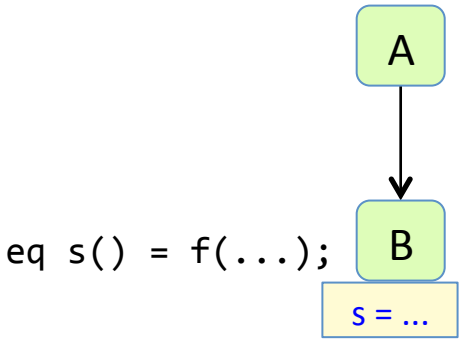
Collection – the attribute is defined by a set of contributions, instead of by an equation.

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

*** Treated in this lecture**

Synthesized attributes

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



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

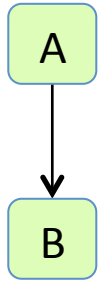
Typically used for propagating information *upwards* in the tree.

Synthesized attributes, example 1

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

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

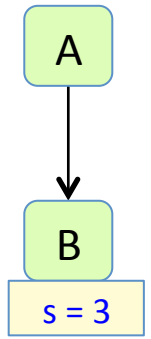
Draw the attribute and its value!



Synthesized attributes, example 1

```
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 free of externally visible side-effects.

Don't do this!

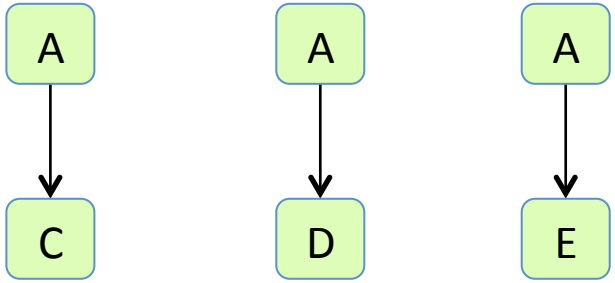
```
int B.counter = 0; // Ordinary field
syn int B.s() {
  counter++; // Visible side-effect
  return counter;
}
```

Warning!
Side-effects are not checked by JastAdd.
The attributes will get inconsistent values.

Synthesized attributes, example 2

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

Three different ASTs.
Draw the attributes and their values!

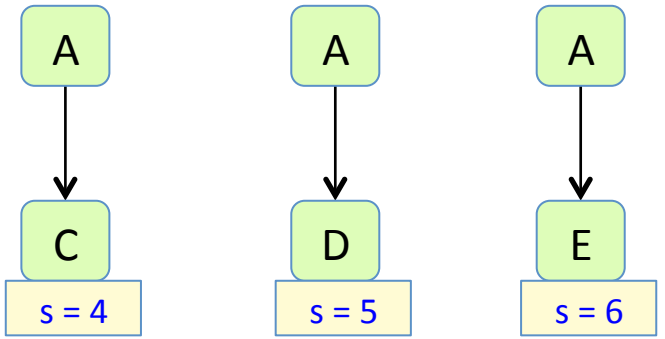


Different subclasses can have different equations.

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

Synthesized attributes, example 2

```
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;
```

Note that equations can override equations in superclasses,
in analogy to how methods can override methods in OO languages.

JastAdd checks that each concrete class has equations for all its synthesized attributes.

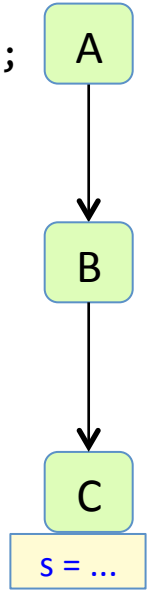
A synthesized attribute is similar to a side-effect free method, but:

- its value is cached (memoized)
- circularity is checked at runtime (results in exception)

Inherited attributes

Inherited attribute:
The equation is in an ancestor

```
eq getB().s() = f(...);
```



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

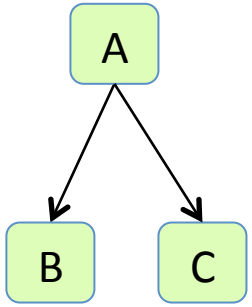
Typically used for propagating information *downwards* in the tree.

Inherited attributes, example 1

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

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

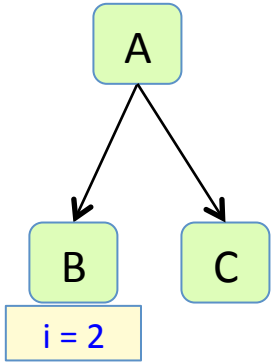
Draw the attribute and its value!



Inherited attributes, example 1

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

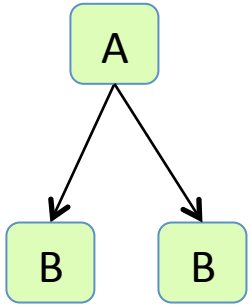
```
inh int B.i();
eq A.getB().i() = 2;
```



Inherited attributes, example 2

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

Draw the attributes and their values!



The parent can specify different equations for its different children.

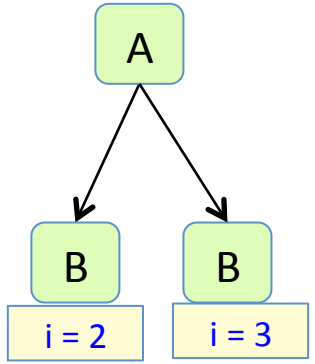
```
inh int B.i();
eq A.getLeft().i() = 2;
eq A.getRight().i() = 3;
```

Inherited attributes, example 2

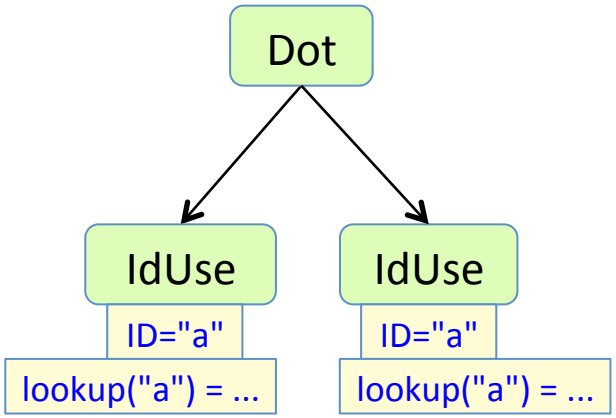
```
A ::= Left:B Right:B;
B;
```

The parent can specify different equations for its different children.

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



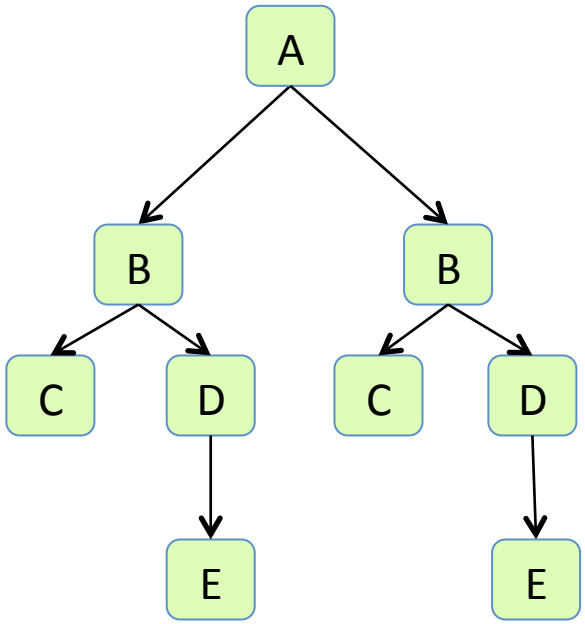
This is useful, for example, when defining scope rules for qualified access. The lookup attributes should have different values for the different IdUses.



Inherited attributes, example 3

```
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.

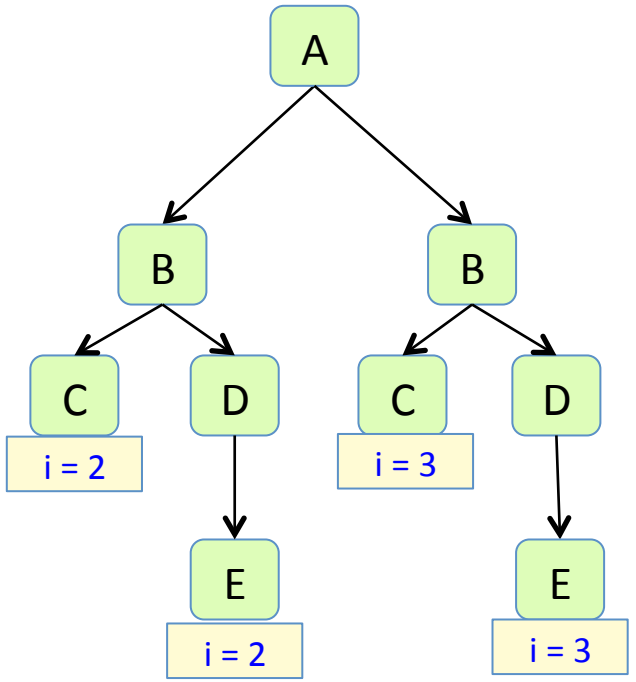
```
eq A.getLeft().i() = 2;
eq A.getRight().i() = 3;
inh int C.i();
inh int E.i();
```

Inherited attributes, example 3

```
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();
```

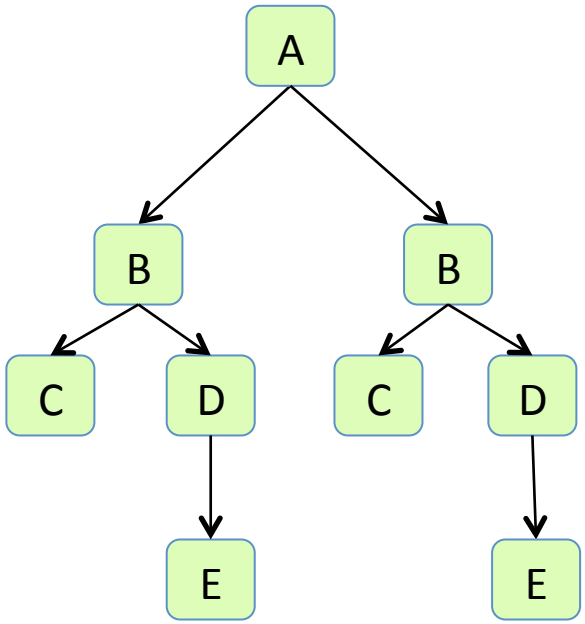


This is called *broadcasting*.

Inherited attributes, example 4

```
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 holds.

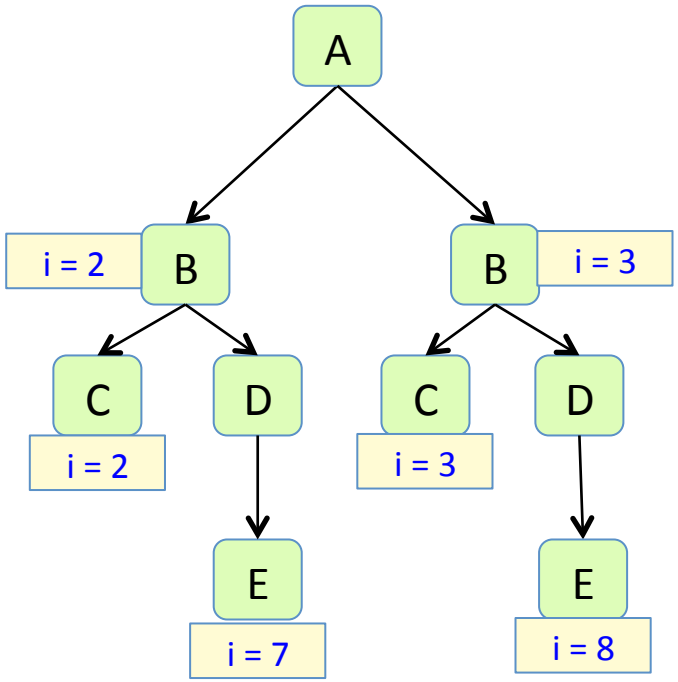
```
eq A.getLeft().i() = 2;
eq A.getRight().i() = 3;
eq B.getD().i() = i() + 5;
inh int B.i();
inh int C.i();
inh int E.i();
```

Inherited attributes, example 4

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

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

```
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, example 5

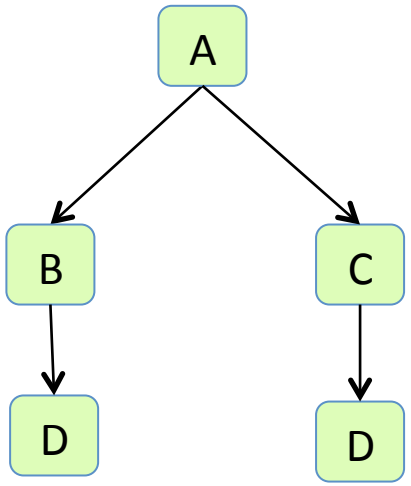
```
A ::= B C;
B ::= D;
C ::= D;
D;
```

There must be an equation for each attribute in any possible AST.

What is the problem with this grammar?

```
eq B.getD().i() = 6;
inh int D.i();
```

Draw the attributes and their values!



Inherited attributes, example 5

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

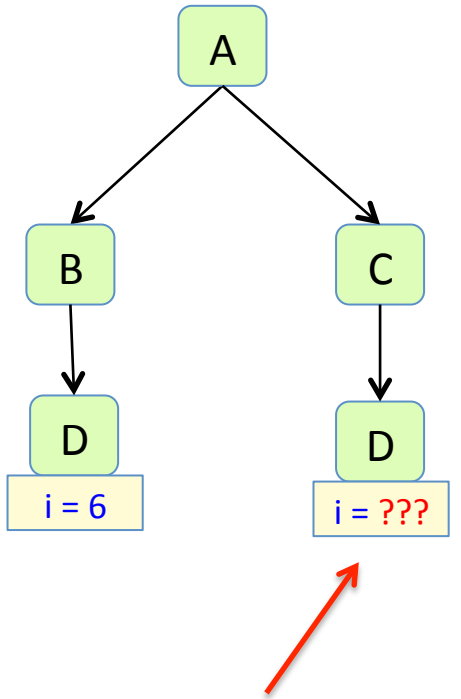
There must be an equation for each attribute in any possible AST.

What is the problem with this grammar?

```
eq B.getD().i() = 6;
inh int D.i();
```

Where can we add an equation to solve the problem?

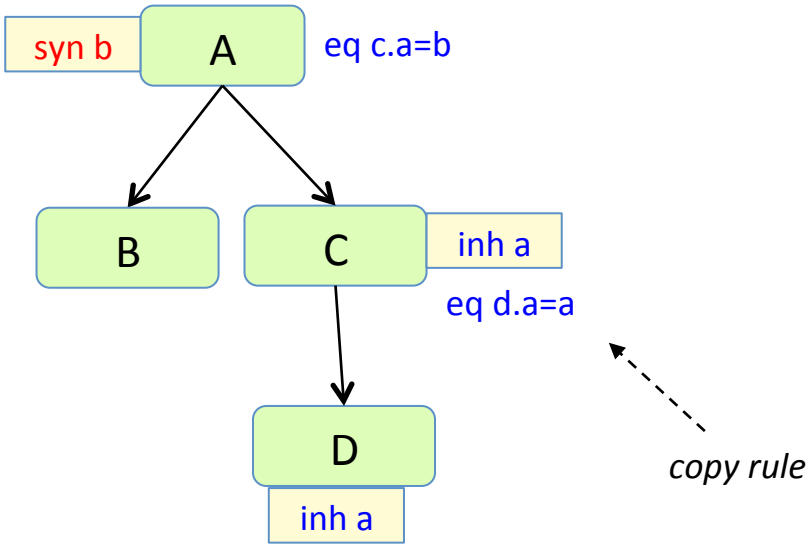
In C or A. Or in their superclass ASTNode.



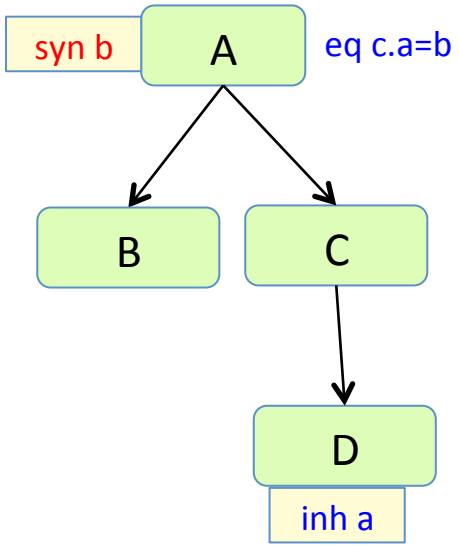
This attribute has no equation!
JastAdd will find this and report an error.

Broadcasting of inherited attributes

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



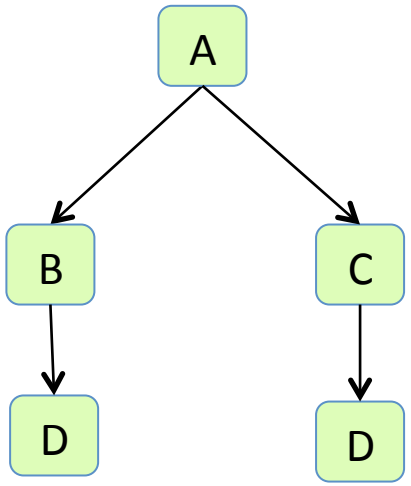
JastAdd:
Equation for inherited attribute
is "broadcasted" to complete subtree.
No "copy rules" are needed.



Inherited attributes, example 6

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

Draw the attributes and their values!



The parent can write an equation that holds for all children.

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

Inherited attributes, example 6

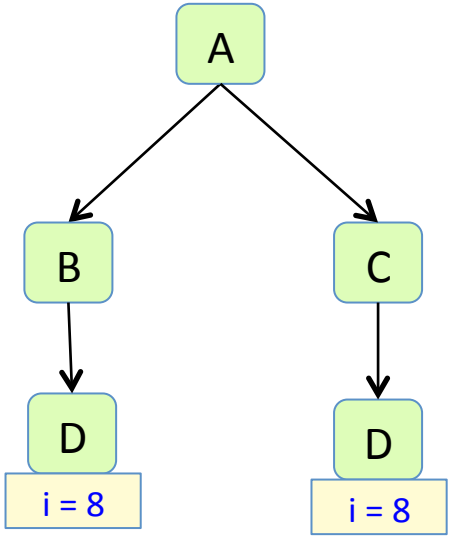
```
A ::= B C;
B ::= D;
C ::= D;
D;
```

The parent can write an equation that holds for all children.

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

This is equivalent to writing an equation for each child:

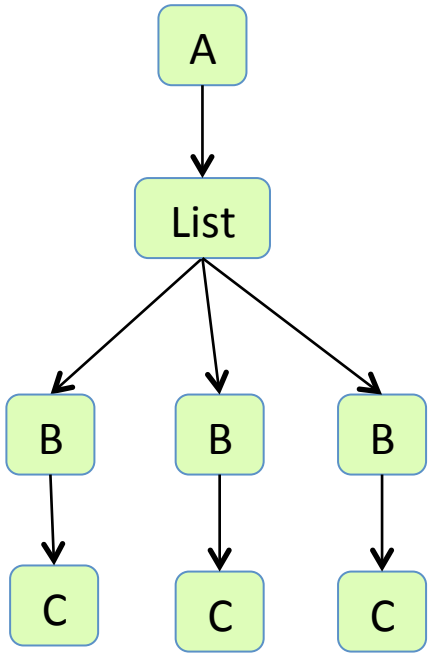
```
eq A.getB().i() = 8;
eq A.getC().i() = 8;
inh int D.i();
```



Inherited attributes, example 7

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

Draw the attributes and their values!



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

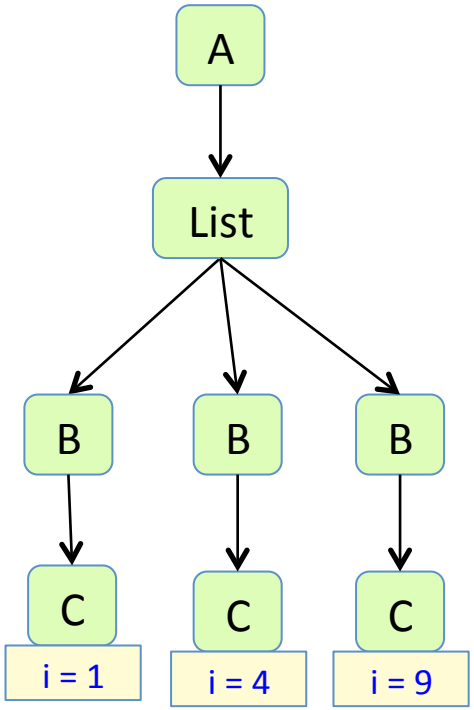
```
eq A.getB(int index).i() = (index+1) * (index+1);
inh int C.i();
```

Inherited attributes, example 7

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

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

```
eq A.getB(int index).i() = (index+1) * (index+1);
inh int C.i();
```



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

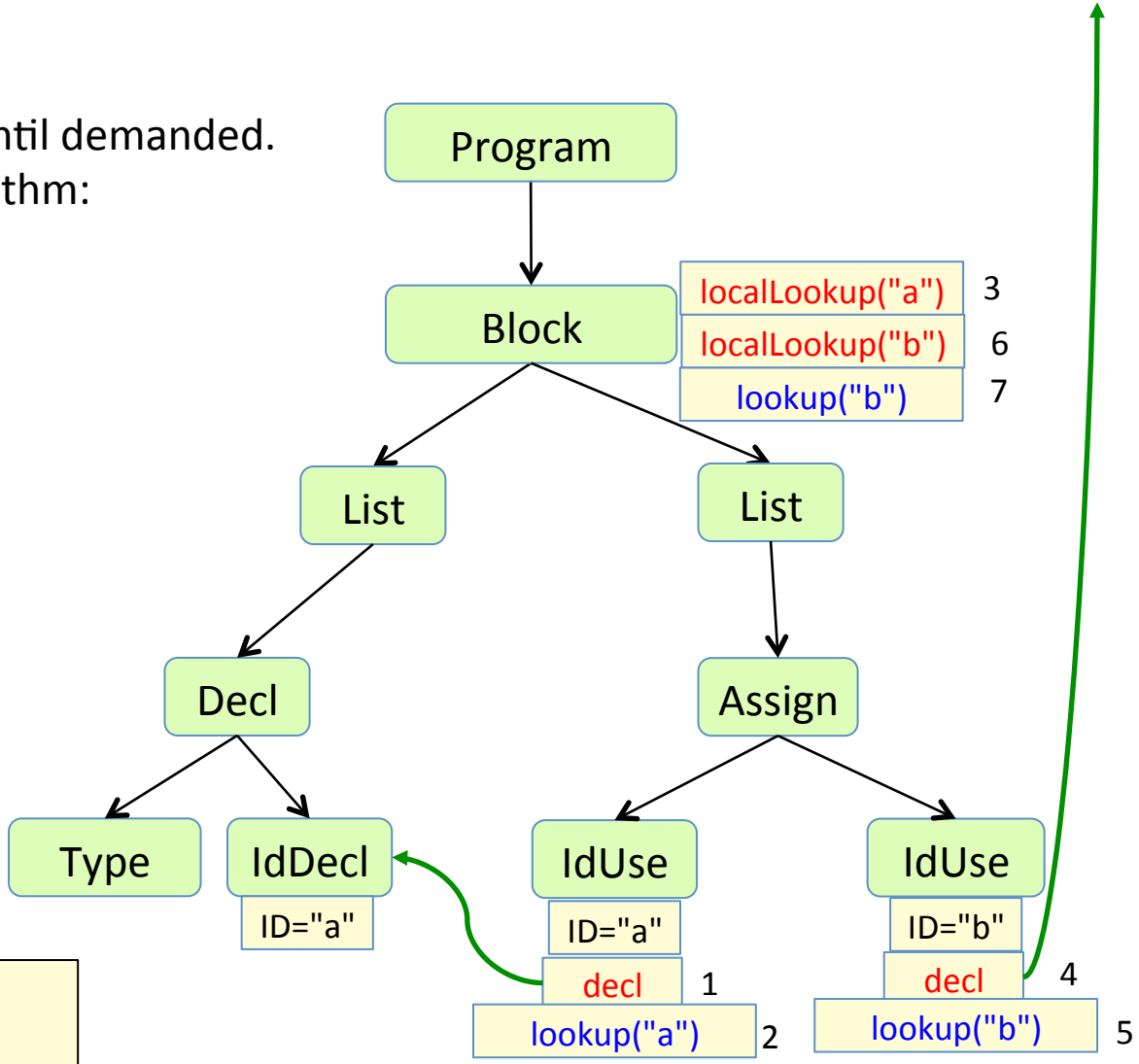
Demand evaluation

Attributes are not evaluated until demanded.
Simple recursive caching algorithm:

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

Example program demanding attributes:

```
Program p = ...
Assign a = p.getBlock().getStmt(0);
System.out.println(a.getTo().decl());
System.out.println(a.getFrom().decl());
```



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 broadcasting?
- 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!