EDAN65: Compilers, Lecture 07 B

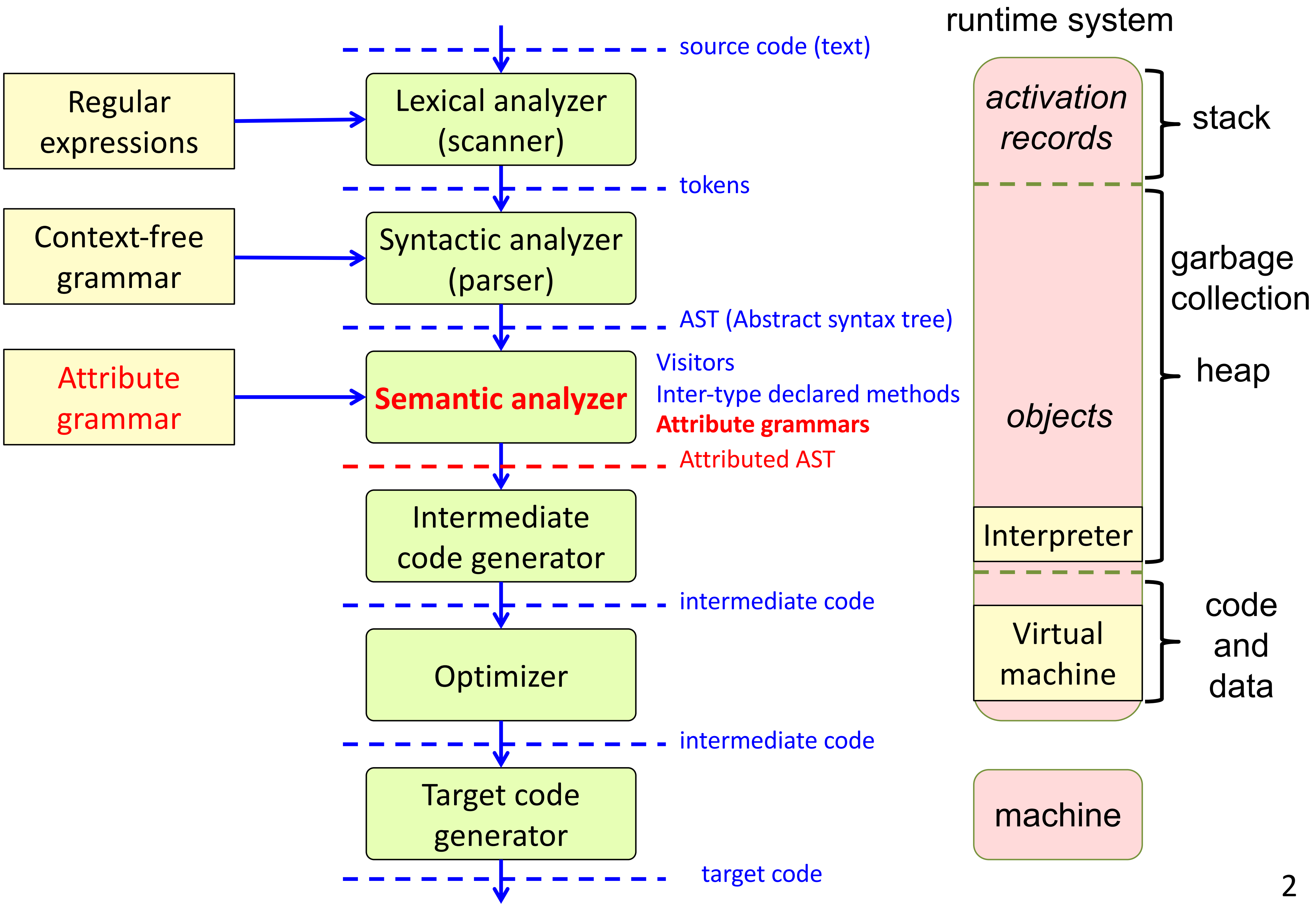
Introduction to Attribute Grammars

intrinsic, synthesized, inherited

Görel Hedin

Revised: 2023-09-12

This lecture



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.

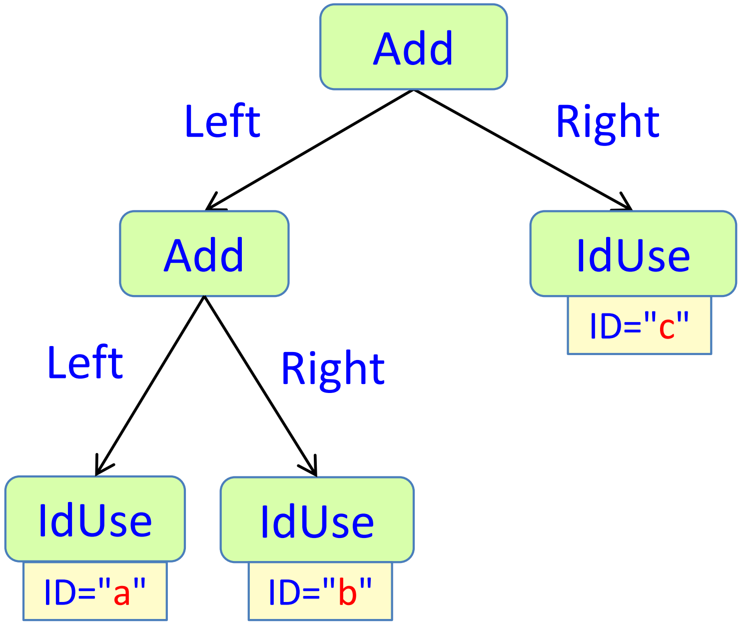
Abstract grammar

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)



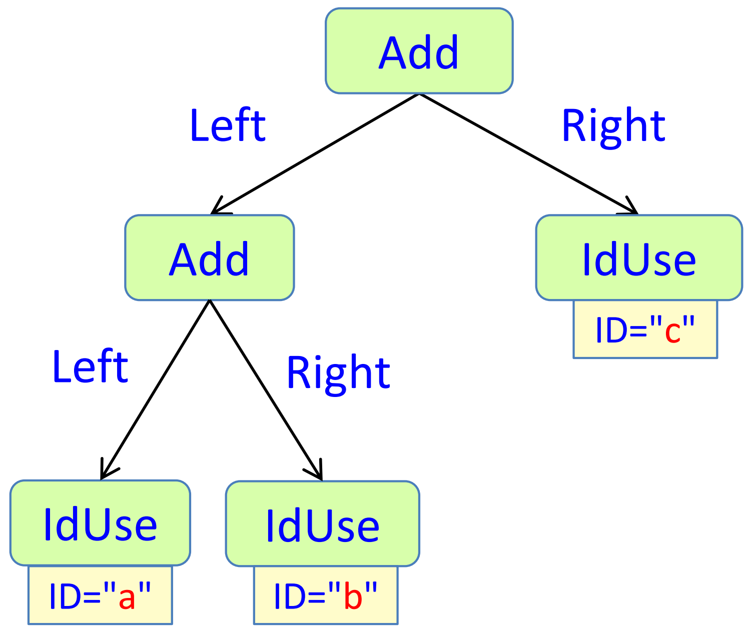
Abstract grammar

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:

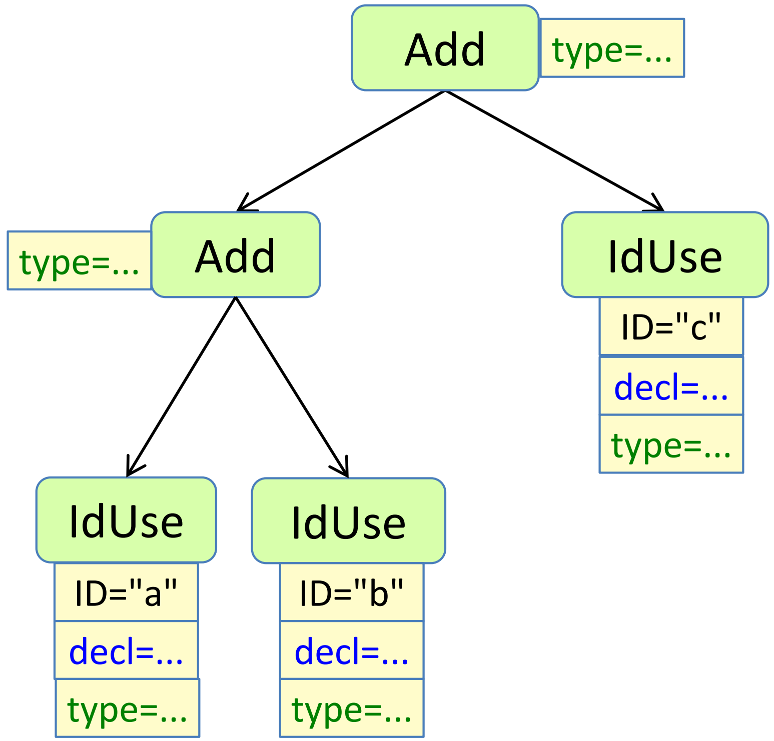
```
abstract Exp;
Add : Exp ::= Left:Exp Right:Exp;
IdUse : Exp ::= <ID:String>;
```

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)



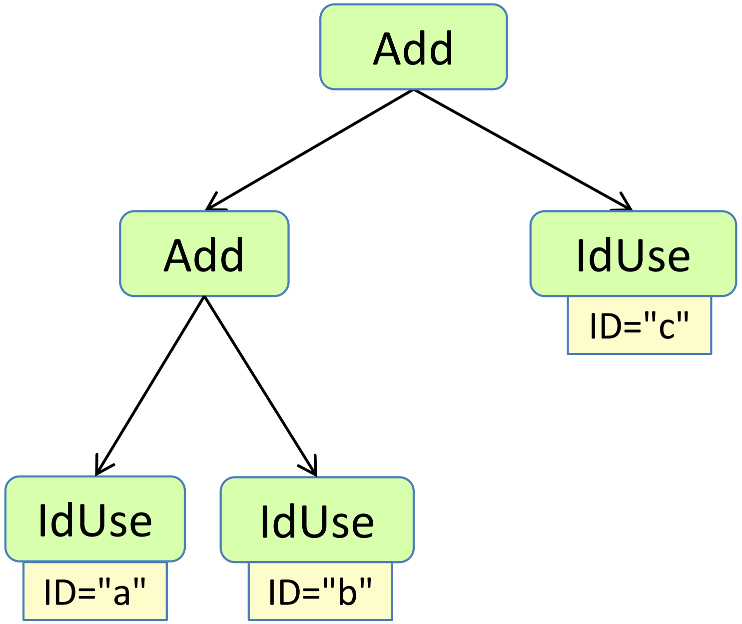
Each declared attribute ... ... will have instances in the AST

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)

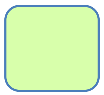
*** Treated in this lecture**

Introduction to attribute grammars

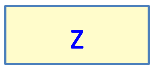
Simple example

attributes and equations

AST node



attribute

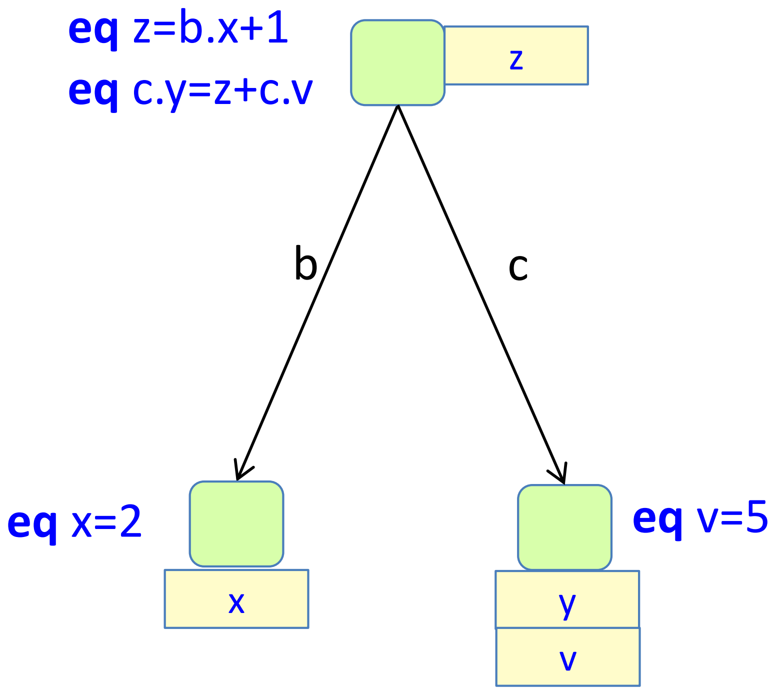


equation:

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

defined attribute

function of other attributes

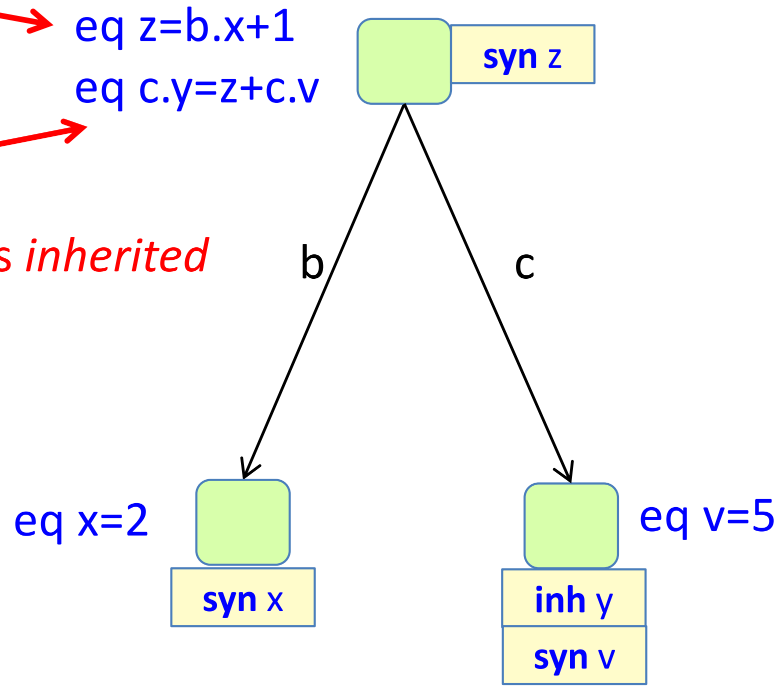


What is the value of y?
Solve the equation system!
(Easy! Just use substitution.)

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

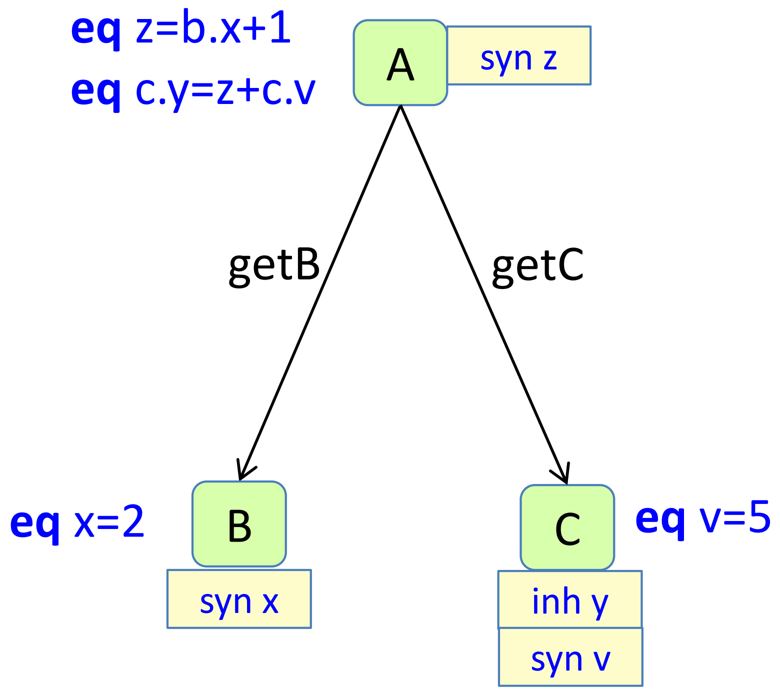
Abstract grammar:

```
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;
```

Well-formed attribute grammar

Abstract grammar:

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

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();
```

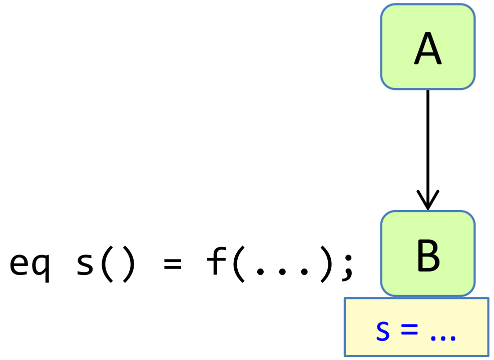
Not well defined. Circular definition.

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

Synthesized attributes

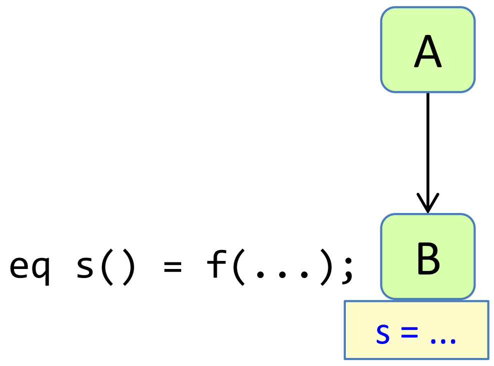
Synthesized attributes

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.

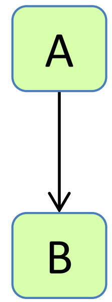
Synthesized attributes

simple example

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

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

Draw the attribute and its value!

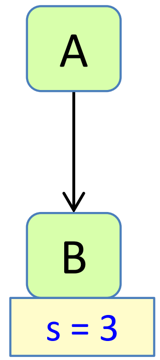


Synthesized attributes

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

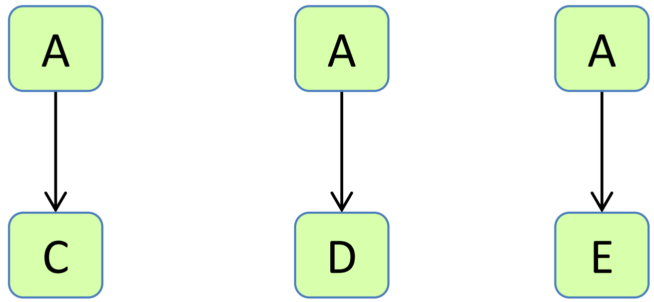
subtypes can have different equations

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

Different subclasses can have different equations.

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

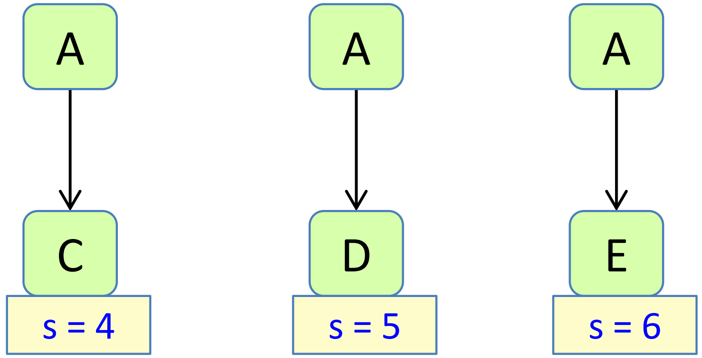
Three different ASTs.
Draw the attributes and their values!



Synthesized attributes

subtypes can have different equations

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



Different subclasses can have different equations.

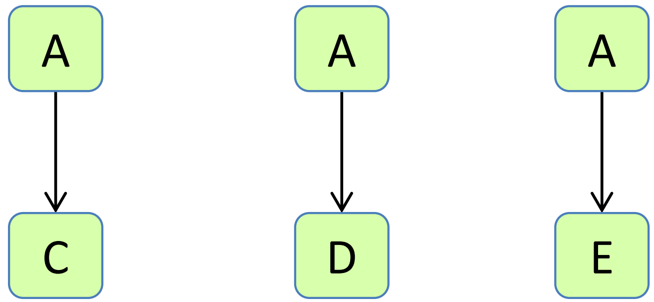
```
syn int B.s();
eq C.s() = 4;
eq D.s() = 5;
eq E.s() = 6;
```

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;
```

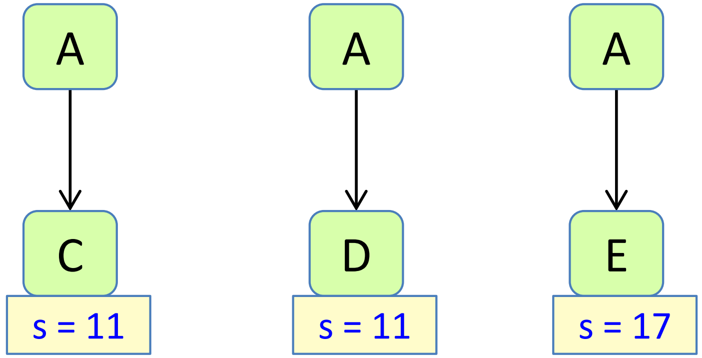


Synthesized attributes

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.

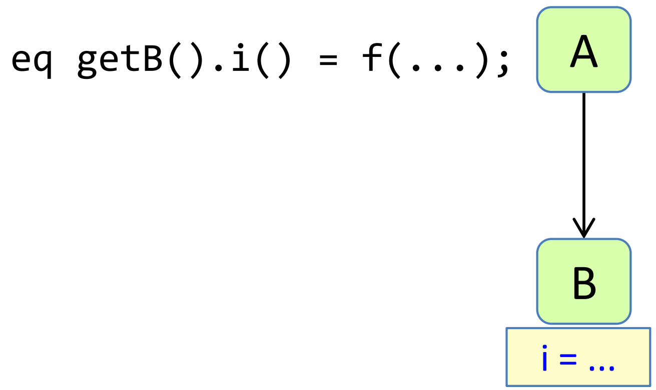
A synthesized attribute is similar to a side-effect free method, but:

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

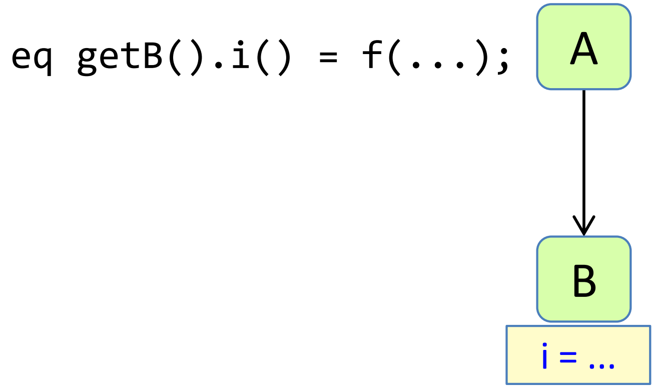
Inherited attributes

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.

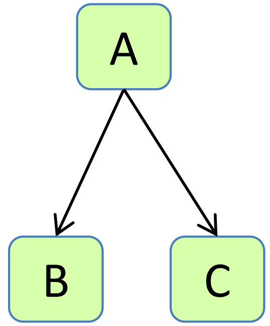
Inherited attributes

simple example

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

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

Draw the attribute and its value!

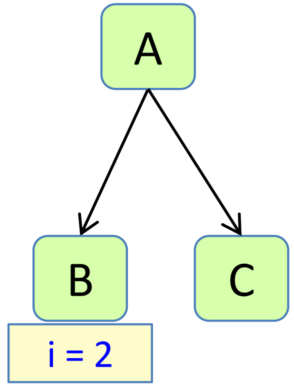


Inherited attributes

simple example

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

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

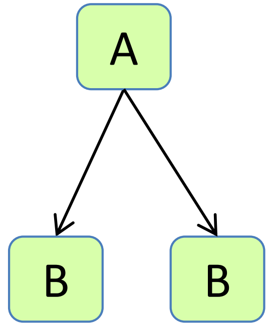


Inherited attributes

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.

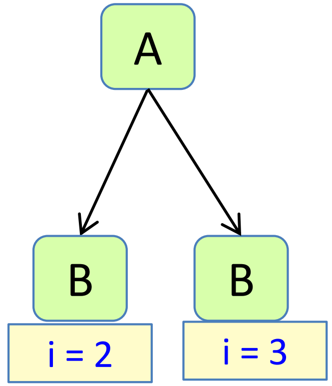
```
inh int B.i();
eq A.getLeft().i() = 2;
eq A.getRight().i() = 3;
```

Inherited attributes
different equations for different children

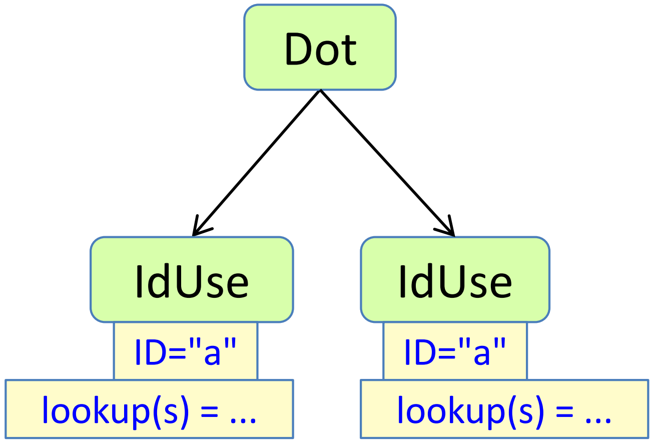
```
A ::= Left:B Right:B;
B;
```

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.

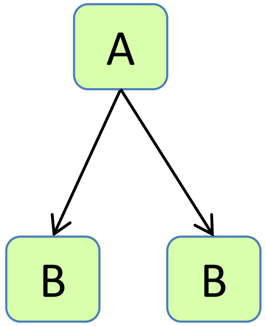


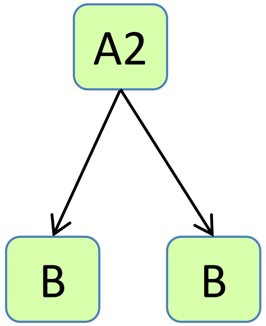
Inherited attributes

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;
```



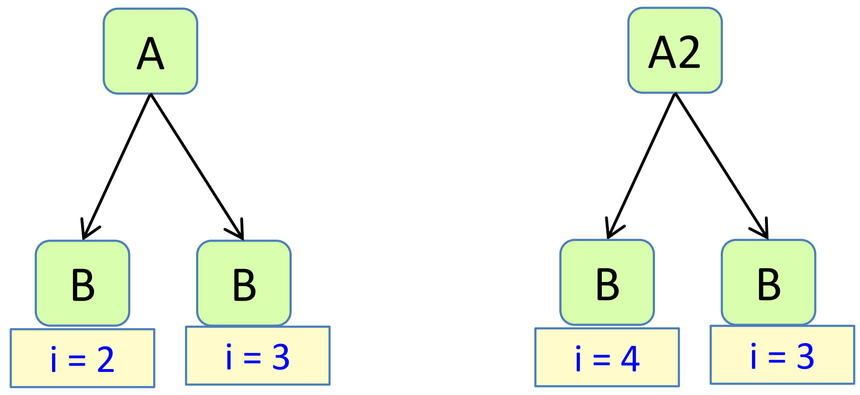


Inherited attributes

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;
```



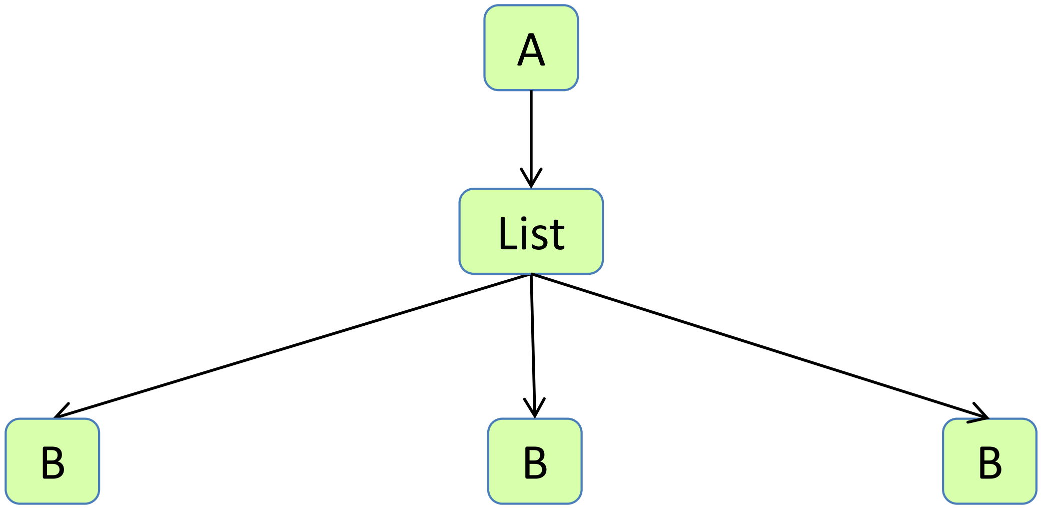
Inherited attributes

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();
```

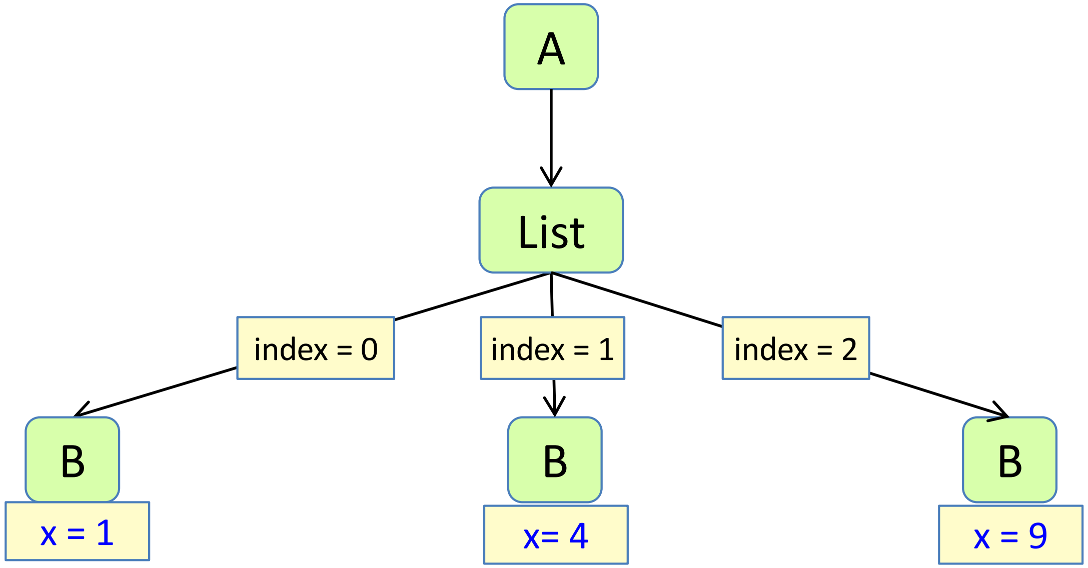


Inherited attributes
a list child has an index

```
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();
```



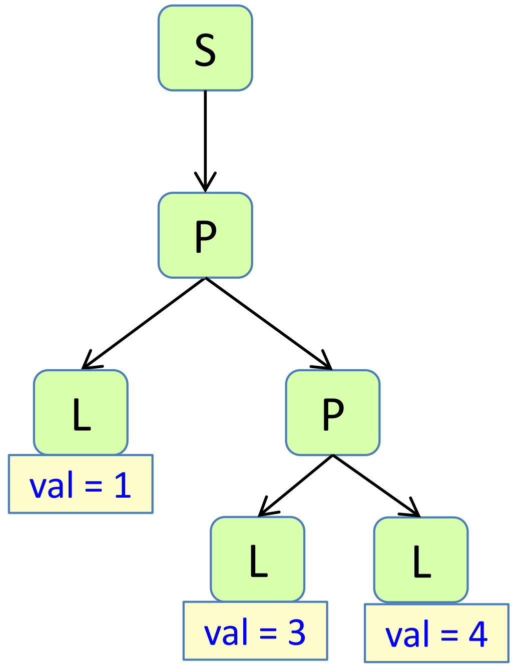
This is useful, for example, when defining name analysis with declare-before-use semantics.

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>;
```

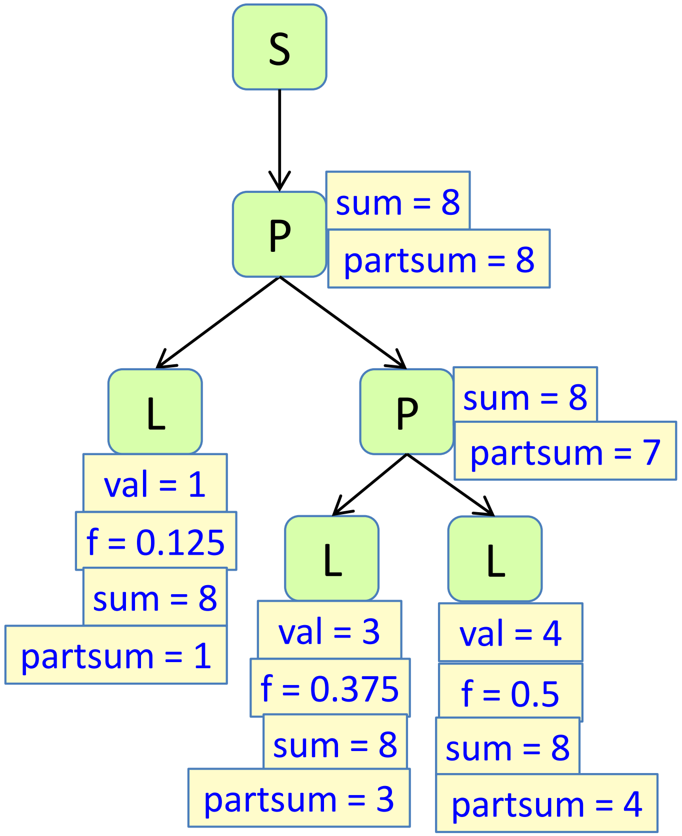


Goal

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();
```

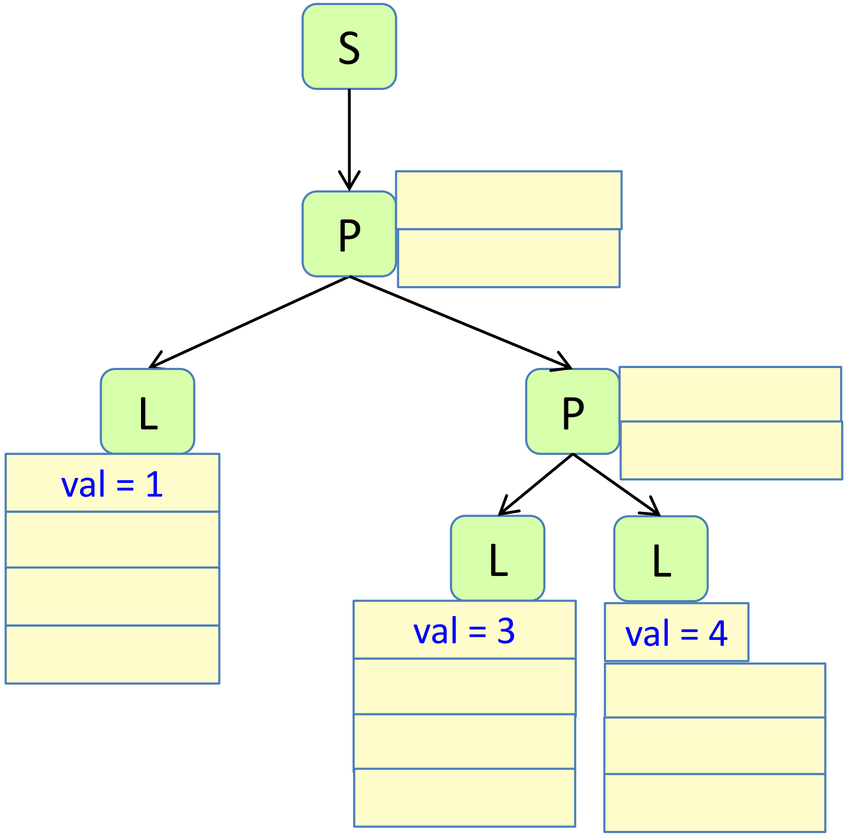


Demand evaluation and memoization

```
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();
```



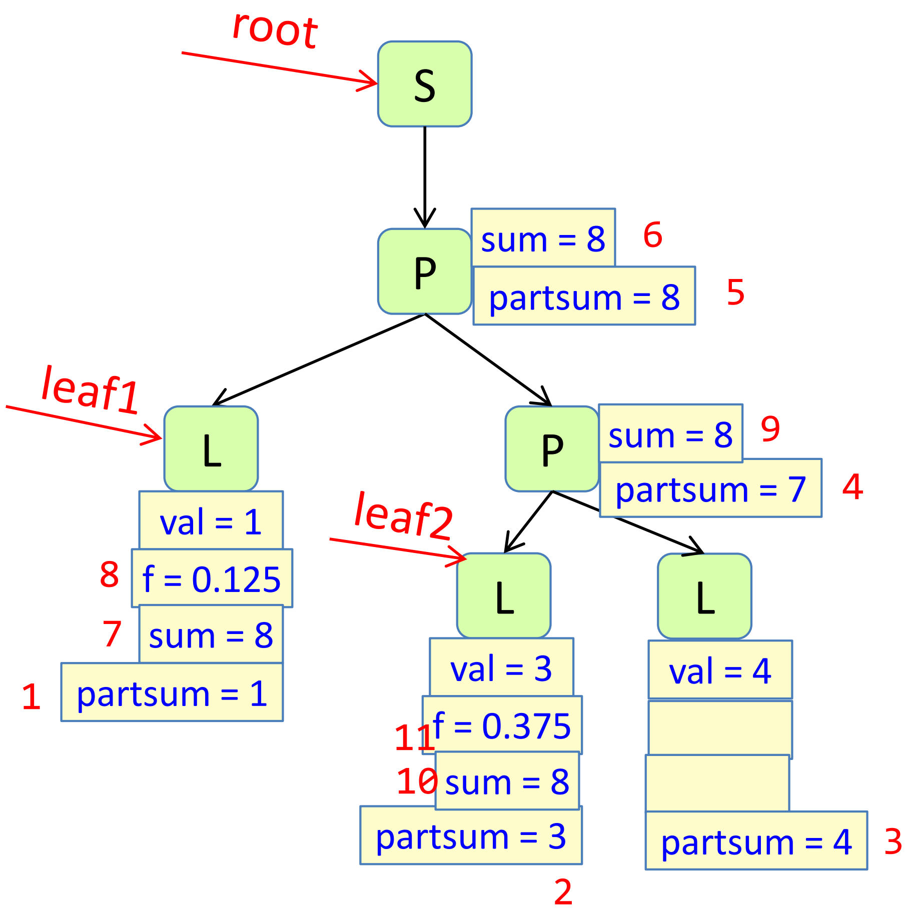
Recursive evaluation algorithm
with memoization

```
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();
```



memoization order

Recursive evaluation algorithm with memoization

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

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!