EDAN65: Compilers, Lecture 08

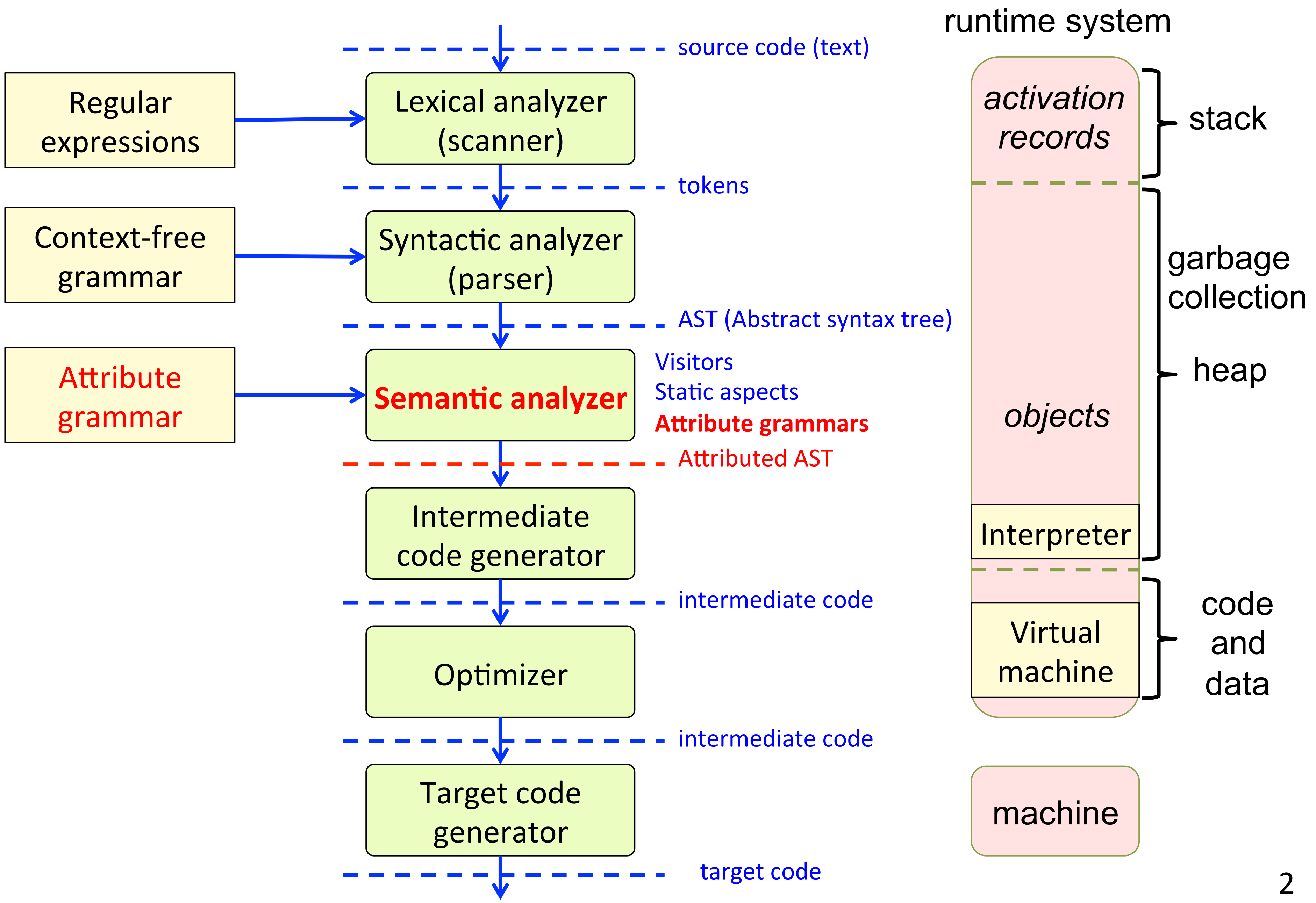
Reference Attribute Grammars

AG mechanisms, Semantic analysis

Görel Hedin

Revised: 2015-09-22

This lecture



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.

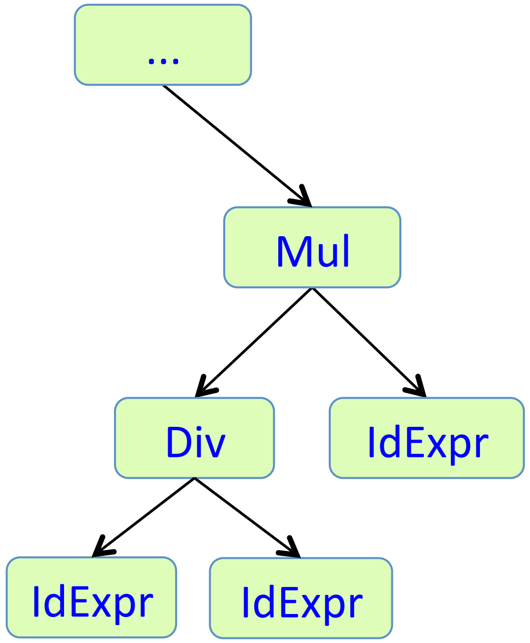
Parameterized – the attribute can have parameters

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)

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

Semantic analysis

```
class A {
  int f;
  int m1(int x) {
    return x * f;
  }
}

class B extends A {
  int m2() {
    System.out.println(m1(3));
  }
}
```

Name analysis: bind each use of an identifier to its declaration

Type analysis: compute the type of each expression

Error checking: find compile-time errors like missing declarations, wrong type, etc.

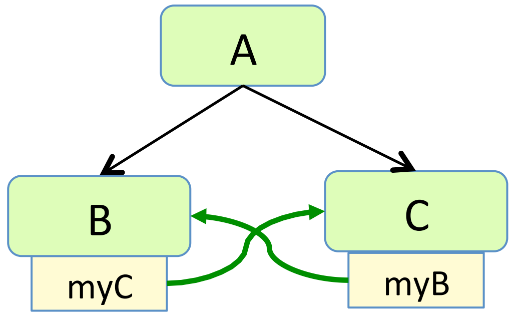
Reference and parameterized attributes

Name analysis

Reference attribute grammars (RAGs)

Can build **graphs** on top of the AST.

Trivial example:



JastAdd specification:

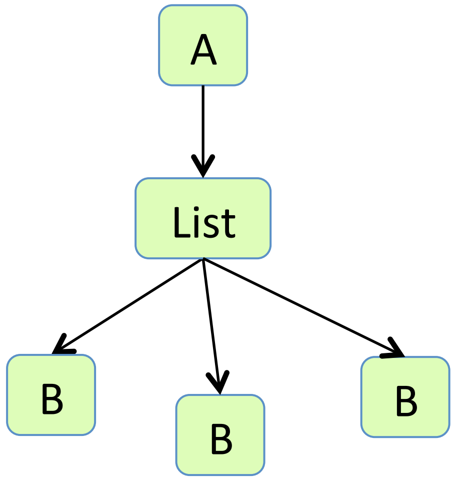
```
A ::= B C;
B;
C;
```

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

Parameterized attributes

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

Draw some isBefore attributes and their values!



An attribute can have parameters.

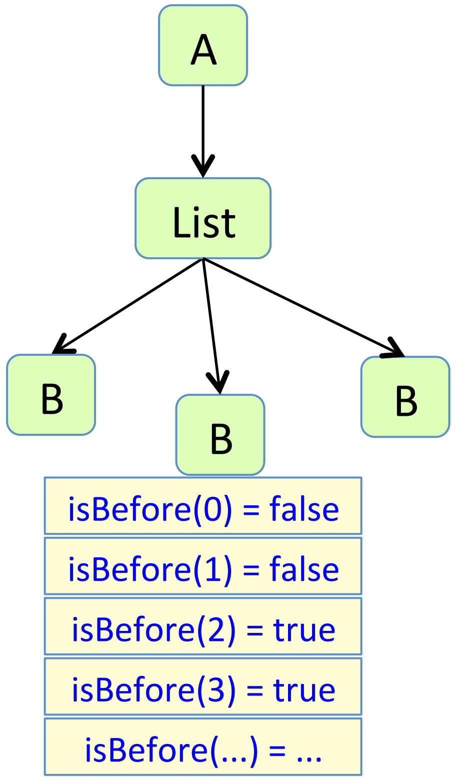
```
inh boolean B.isBefore(int i);
eq A.getB(int index).isBefore(int i) = index < i;
```

Parameterized attributes

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

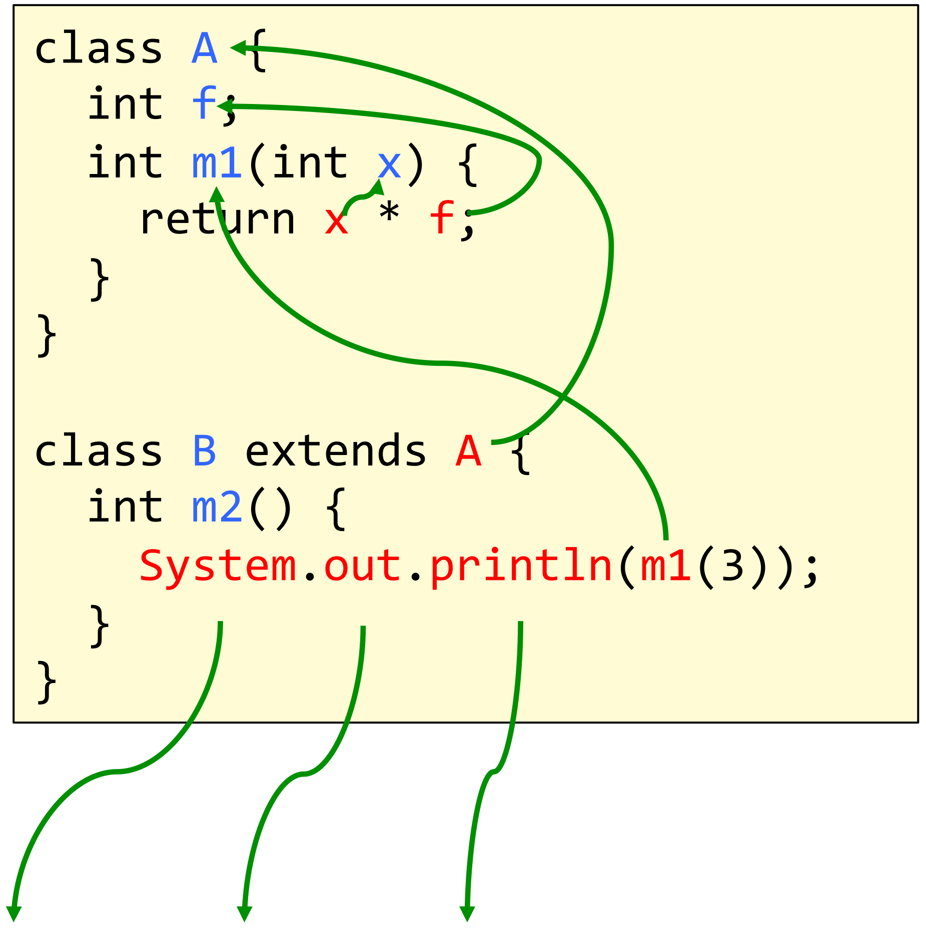
An attribute can have parameters.

```
inh boolean B.isBefore(int i);
eq A.getB(int index).isBefore(int i) = index < i;
```



There is one attribute instance for each possible parameter combination.
Potentially infinitely many attribute instances.
Only the accessed ones will be evaluated.

Name analysis



```
class A {
  int f;
  int m1(int x) {
    return x * f;
  }
}

class B extends A {
  int m2() {
    System.out.println(m1(3));
  }
}
```

Name analysis: bind each use of an identifier to its declaration

Name binding: a reference from a use to its declaration

Scope of a declaration: the parts of the program where it is visible.

Name binding rules: also known as *scope rules* or *visibility rules*.

Typically, there are rules for

- blocks, nesting, inheritance
- name collisions, shadowing
- declaration order (insignificant or declare-before-use?)
- visibility restrictions (private, public, ...)
- qualified access (a.b)
- overloading, namespaces
- ...

Name binding: Blocks

```
class A {
  int f;
  int m1(int x) {
    int a = 4;
    int f = a + 5;
    return x * f + b;
  }
}

class B extends A {
  int m2() {
    System.out.println(m1(3));
  }
}

...
```

Block: a syntactic unit containing declarations and statements.

Can be **nested**. Called *block strucure* or *lexical nesting*.

Declarations in inner blocks **shadow** declarations in outer blocks

Declaration ordering can be

- **insignificant**, or
- **declare-before-use**

Name binding: Inheritance

```
class A {
  int x;
  class AA {
    int y;
  }
}

class B extends A {
  int y, z;
  class BB extends AA {
    int v;
    void m() {
      int w = x + y + z + v;
    }
  }
}

...
```

Inheritance and block nesting can be combined.

In what block order should we look for the declaration of x?

m
BB
AA
B
A
globally

Which declaration of y is y bound to?

AA.y
since inheritance binds tighter than lexical nesting.

Name binding: Qualified access

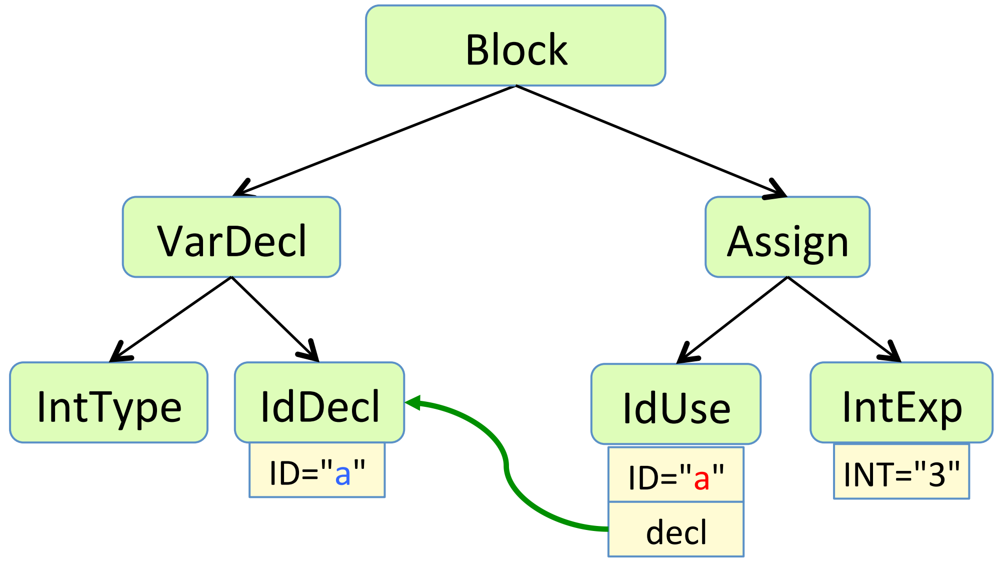
```
class A {
  int m() { ... };
}

class B extends A {
  void m() {
    A a = new B();
    System.out.println(a.m());
  }
}

...
```

Qualified access (dot notation)

The binding of m depends on the *type* of a.

Recall: Representing name bindings in an AST

```
{
  int a;
  a = 3;
}
```

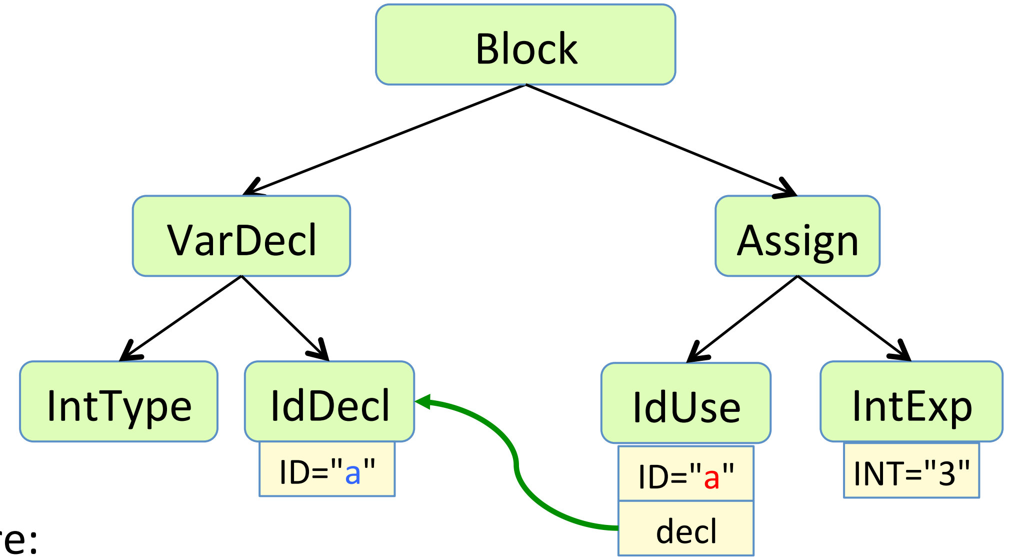


IdDecl for declared names
IdUse for used names

An attribute **decl** represents the name binding.

Recall:
Computing name bindings imperatively

```
{
  int a;
  a = 3;
}
```



Use a **symbol table** data structure:
For each block, a **map** from visible names to declarations.
Use a **stack** of maps to handle nested blocks.

Algorithm:
Traverse the AST
push/pop symbol table when entering/leaving a block
add/lookup identifiers when encountering IdDecls/IdUses

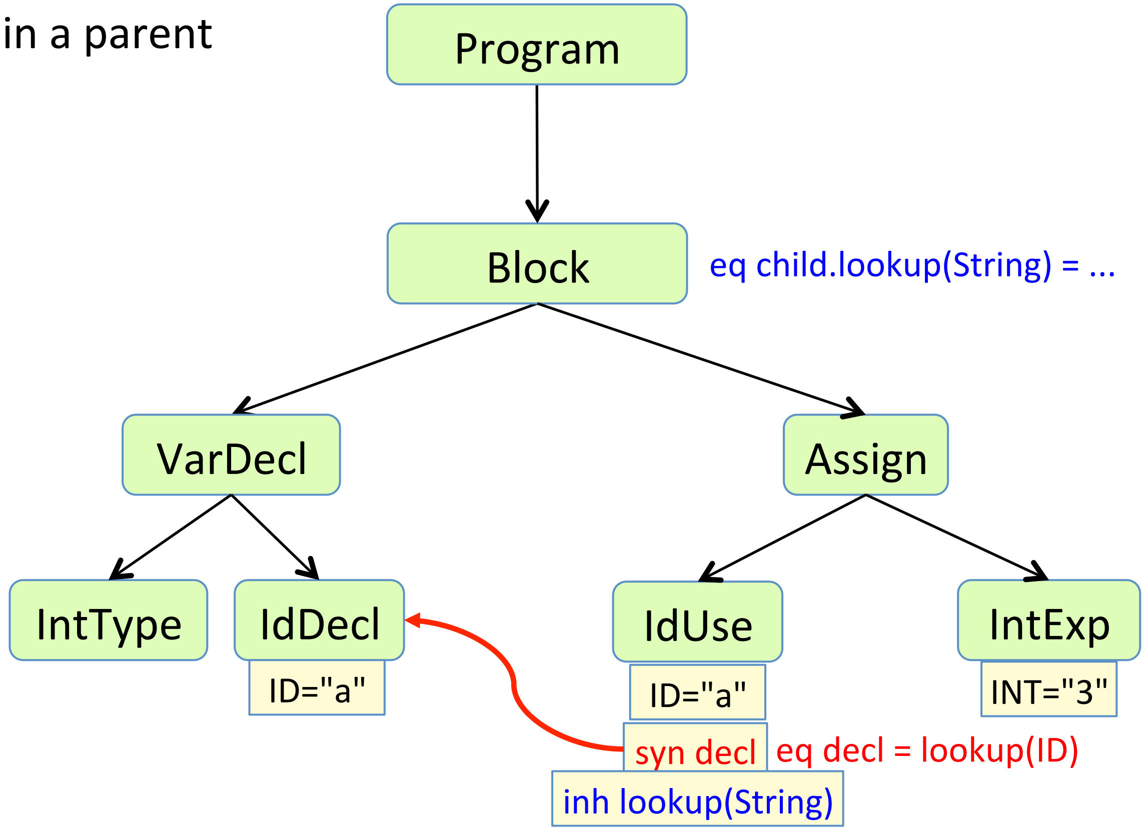
Problems with the imperative approach

- Need to write an algorithm that computes things in the right order.
- What if we have more complex name binding rules?
  Need a more elaborate symbol table.
  The algorithm may get complex.
- What if we extend the language?
  Need to change the algorithm.

Solution: Attribute grammars – a declarative approach

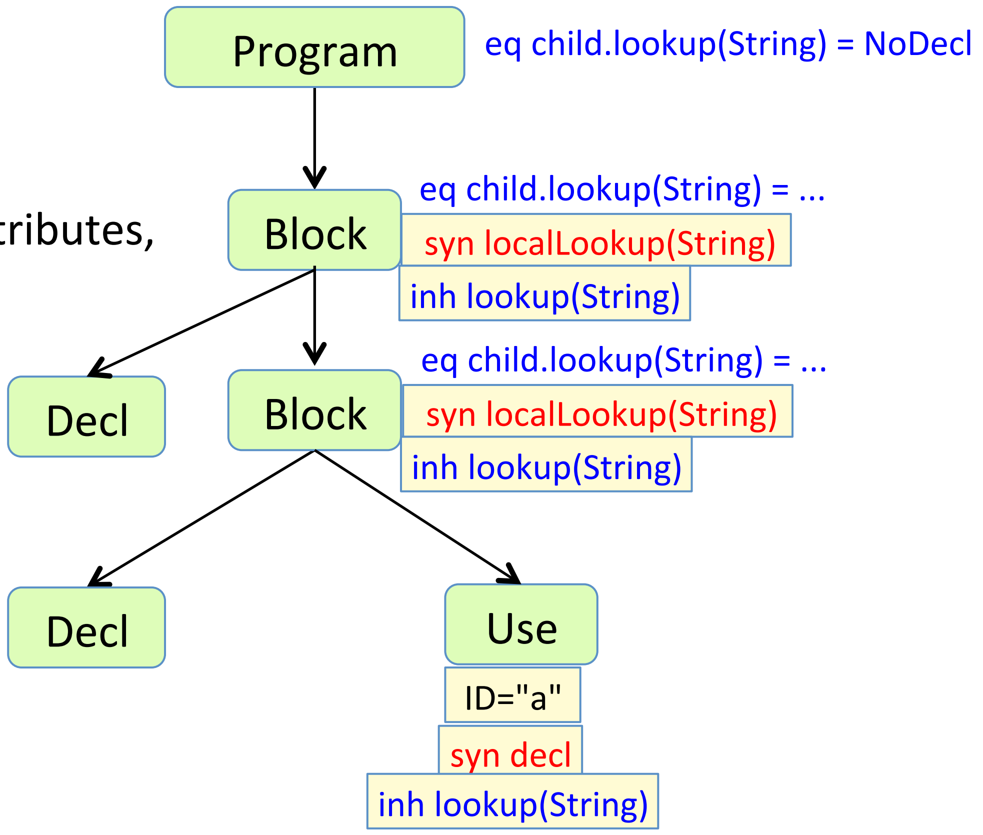
Name analysis using RAGs

Think declaratively!!
What attributes would I like the nodes to have?
Synthesized – defined in the node
Inherited – defined in a parent



The **Lookup pattern** for name analysis in RAGs

decl – the name binding
lookup(String) – finds the declaration
localLookup(String) – looks locally
eq child.**lookup**(String) –
delegates to localLookup and lookup attributes, according to scope rules.

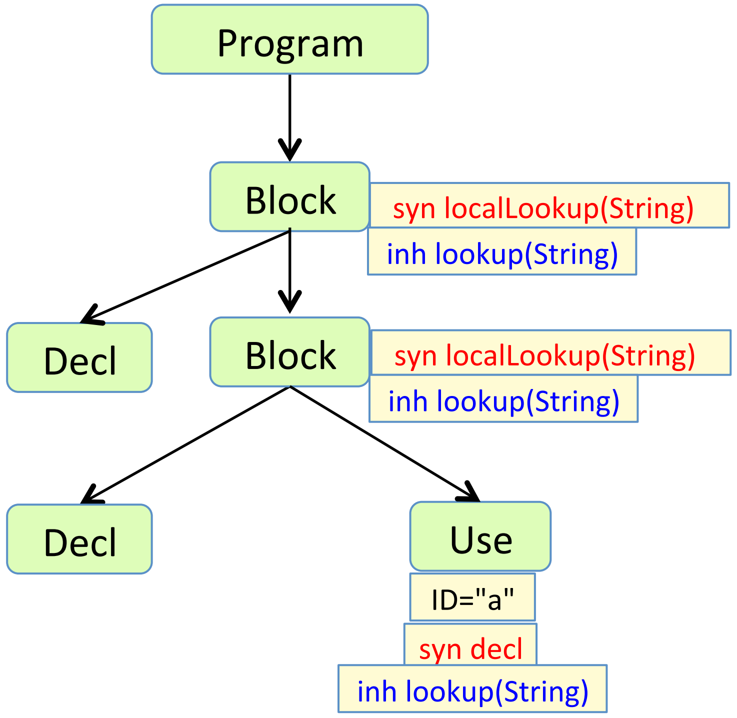


General pattern for name analysis.
Can handle block structure, inheritance, qualified access, ...

Example implementation in JastAdd

Abstract grammar:

```
Program ::= Block;
Block : Stmt ::= Decl* Stmt*;
Decl ::= Type IdDecl;
IdDecl ::= <ID:String>;
Type;
abstract Stmt;
Assign : Stmt ::= To:IdUse From:IdUse;
IdUse ::= <ID:String>;
```



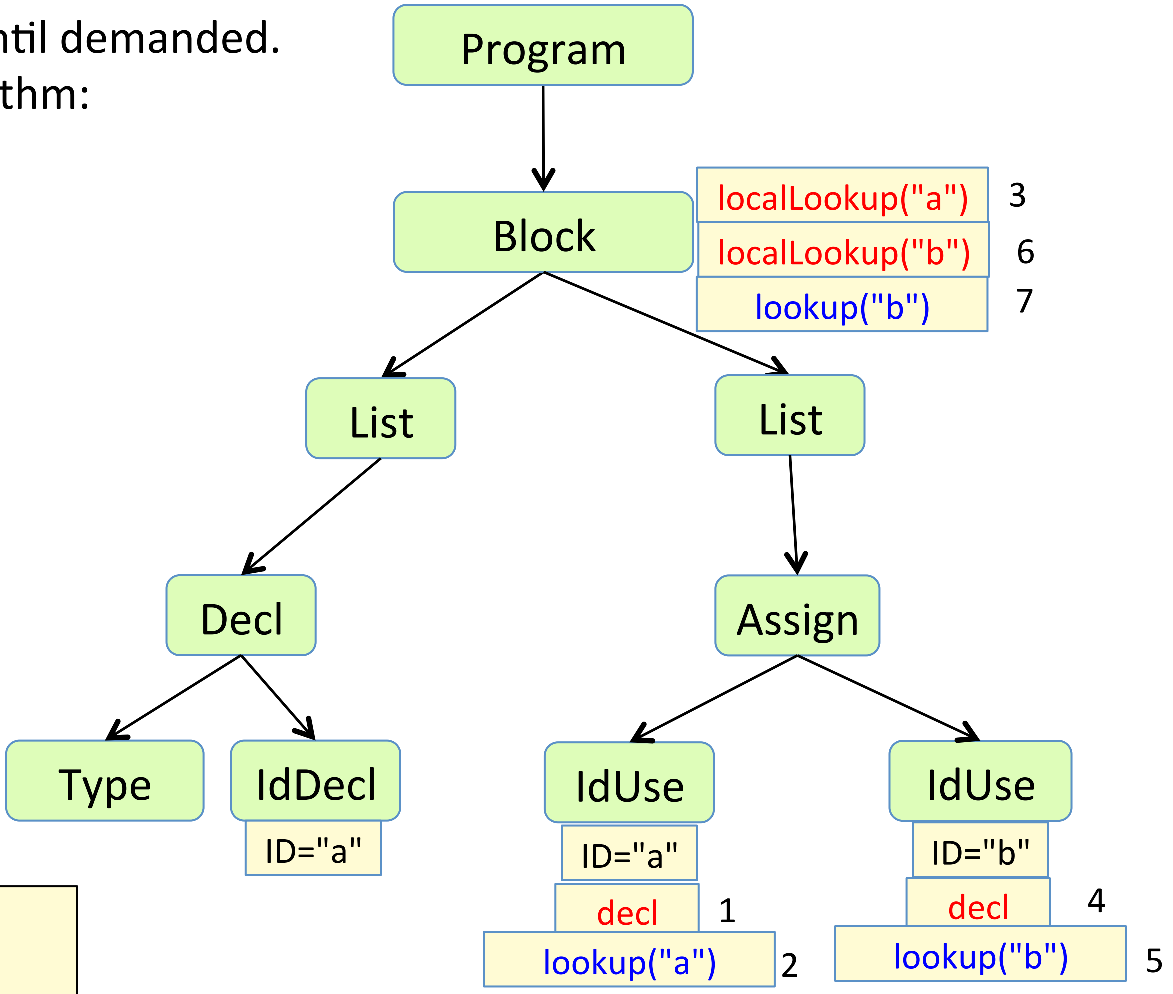
Attributes and equations:

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

inh IdDecl IdUse.lookup(String s);

eq Block.getStmt().lookup(String s) {
  IdDecl d = localLookup(s);
  if (d != null) return d;
  return lookup(s);
}

syn IdDecl Block.localLookup(String s) {
  for (Decl d: getDecls()) {
    if (d.getIdDecl().getID().equals(s))
      return d.getIdDecl();
  }
  return null;
}

inh IdDecl Block.lookup(String s);

eq Program.getBlock().lookup(String s) {
  return null;
}
```

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());
```

More efficient implementation of localLookup

```
syn IdDecl Block.localLookup(String s) {
  for (Decl d: getDecls()) {
    if (d.getIdDecl().getID().equals(s))
      return d.getIdDecl();
  }
  return unknownDecl();
}
```

What happens if there are 1000 elements in the declaration list?
What is the complexity?

Linear search for each declaration gives quadratic time complexity: O(n^2)

More efficient:
Use a local hashmap which is built on the first access.
After that each access is done in constant time.
Resulting complexity: O(n)

```
syn IdDecl Block.localLookup(String s) {
  IdDecl result = localMap().get(s);
  return (result!=null)?result:unknownDecl();
}

syn Map<String,IdDecl> Block.localMap() {
  Map<String,IdDecl> map = new HashMap<String,IdDecl>();
  for (Decl d: getDecls()) {
    IdDecl id = d.getIdDecl();
    map.put(id.getID(), id);
  }
  return map;
}
```

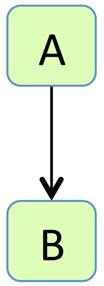
Nonterminal attributes

The Null Pattern

Nonterminal attributes (NTAs)

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

Draw the n attribute and its value!



A nonterminal attribute is both an attribute and a nonterminal (AST node).

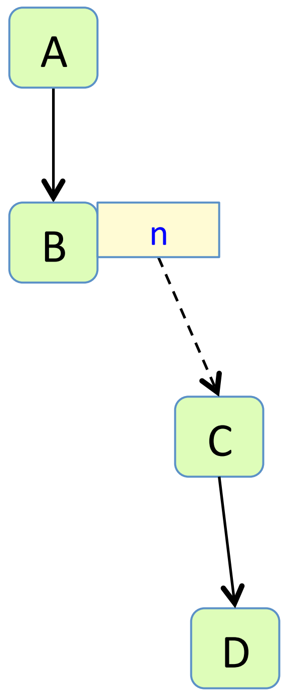
The value must consist of *fresh* nodes.
(If you reuse existing nodes, the attributes may get inconsistent values.)

```
syn nta C B.n() = new C(new D());
```

Use for *reifying* information (making implicit information explicit).
For example, primitive types, predefined functions, ...

Nonterminal attributes (NTAs)

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



A nonterminal attribute is both an attribute and a nonterminal (AST node).

The value must consist of *fresh* nodes.
(If you reuse existing nodes, the attributes may get inconsistent values.)

```
syn nta C B.n() = new C(new D());
```

Use for *reifying* information (making implicit information explicit).
For example, primitive types, predefined functions, ...

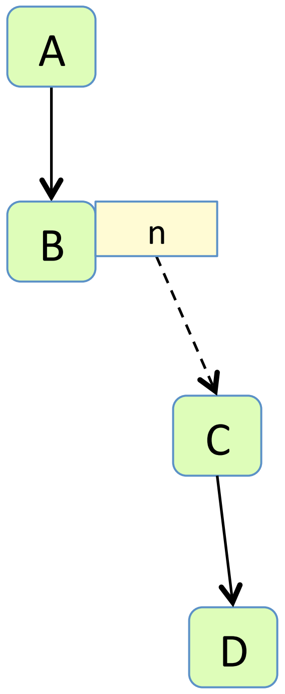
Warning!
If you reuse existing nodes for NTAs, the AST will be inconsistent.
JastAdd does not check this.

Nonterminal attributes (NTAs)

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

```
syn nta C B.n() = new C(new D());
```

Draw the x, y, and z attributes and their values!



An NTA may itself have attributes.

```
inh int C.x();
eq B.n().x() = 5;
syn int B.y() = n().z() * 3;
syn int C.z() = x() + 2;
```

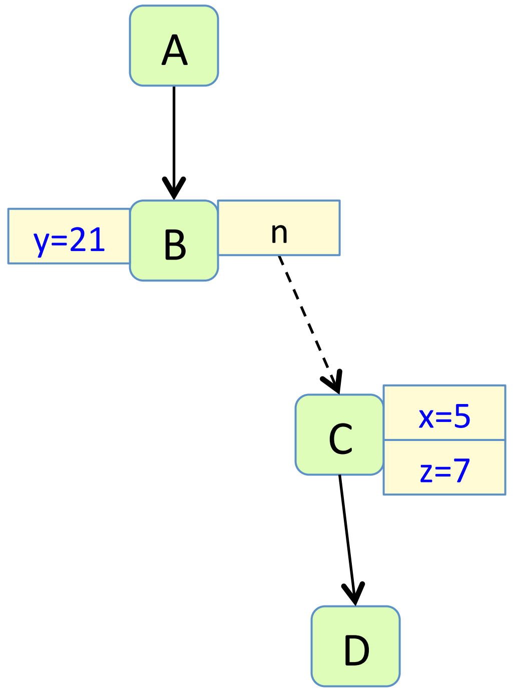
Nonterminal attributes (NTAs)

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

```
syn nta C B.n() = new C(new D());
```

An NTA may itself have attributes.

```
inh int C.x();
eq B.n().x() = 5;
syn int B.y() = n().z() * 3;
syn int C.z() = x() + 2;
```



The **Null object pattern**

Use a **real object** instead of null.
Give the object suitable behavior.
The code becomes simpler.

In RAGs: use null objects for missing declarations, unknown types, etc.

But how can we implement unknownDecl()?

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

inh IdDecl IdUse.lookup(String s);

eq Block.getStmt().lookup(String s) {
  IdDecl d = localLookup(s);
  if (!d.isUnknown()) return d;
  return lookup(s);
}

syn IdDecl Block.localLookup(String s) {
  for (Decl d: getDecls()) {
    if (d.getIdDecl().getID().equals(s))
      return d.getIdDecl();
  }
  return unknownDecl();
}

inh IdDecl Block.lookup(String s);

eq Program.getBlock().lookup(String s) {
  return unknownDecl();
}
```

See http://en.wikipedia.org/wiki/Null_Object_pattern

Non-terminal attributes (NTAs)

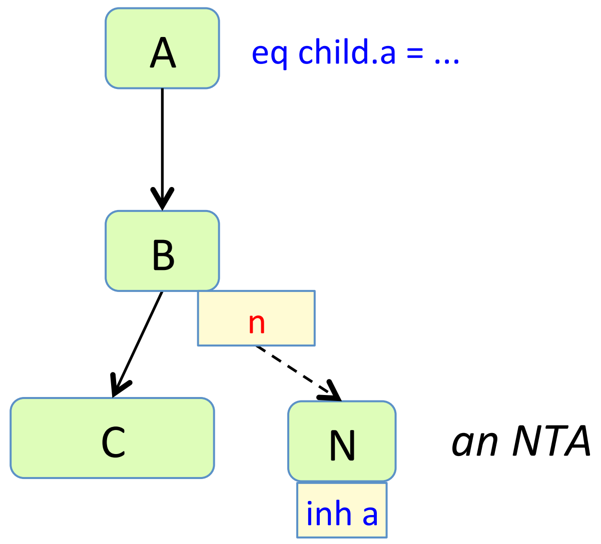
An NTA is **both** a **node** and an **attribute**.

The right-hand side of its defining equation must be a **fresh object** (not part of any AST).

Useful for *reifying* implicit constructs (make them explicit in the AST), like:

- Missing declarations
- Unknown types
- Primitive types and functions

```
syn nta N B.n() = new N();
```



An NTA can have attributes.
The owning node (or its ancestors) must define the inherited attributes of the NTA.

Non-terminal attributes (NTAs)

An NTA is **both** a **node** and an **attribute**.

The right-hand side of its defining equation must be a **fresh object** (not part of any AST).

Add an UnknownDecl object to the AST using a non-terminal attribute

Extend the abstract grammar:

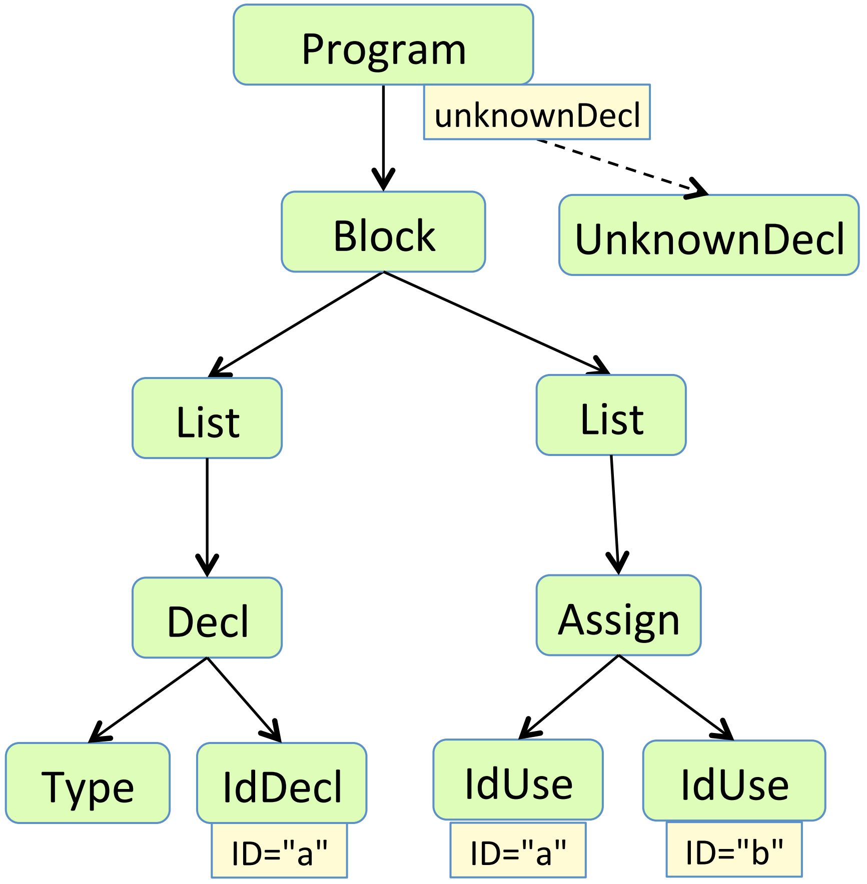
```
UnknownDecl : IdDecl;
```

Add the NTA:

```
syn nta UnknownDecl Program.unknownDecl() =
  new UnknownDecl("<Unknown>");
```

Implement the special behavior:

```
syn boolean IdDecl.isUnknown() = false;
eq UnknownDecl.isUnknown() = true;
```



But how can we make the UnknownDecl object known throughout the AST?

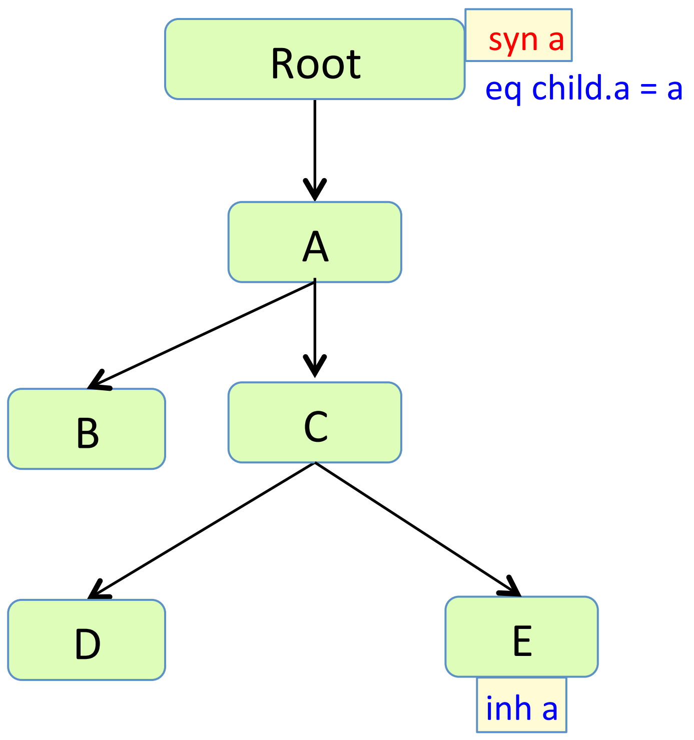
The **Root Attribute** pattern

Intent:
Make an attribute in the root visible throughout the AST.

Solution:
Add an equation in the root, propagating the value to the children.

Expose the attribute by declaring it as inherited where it is needed.

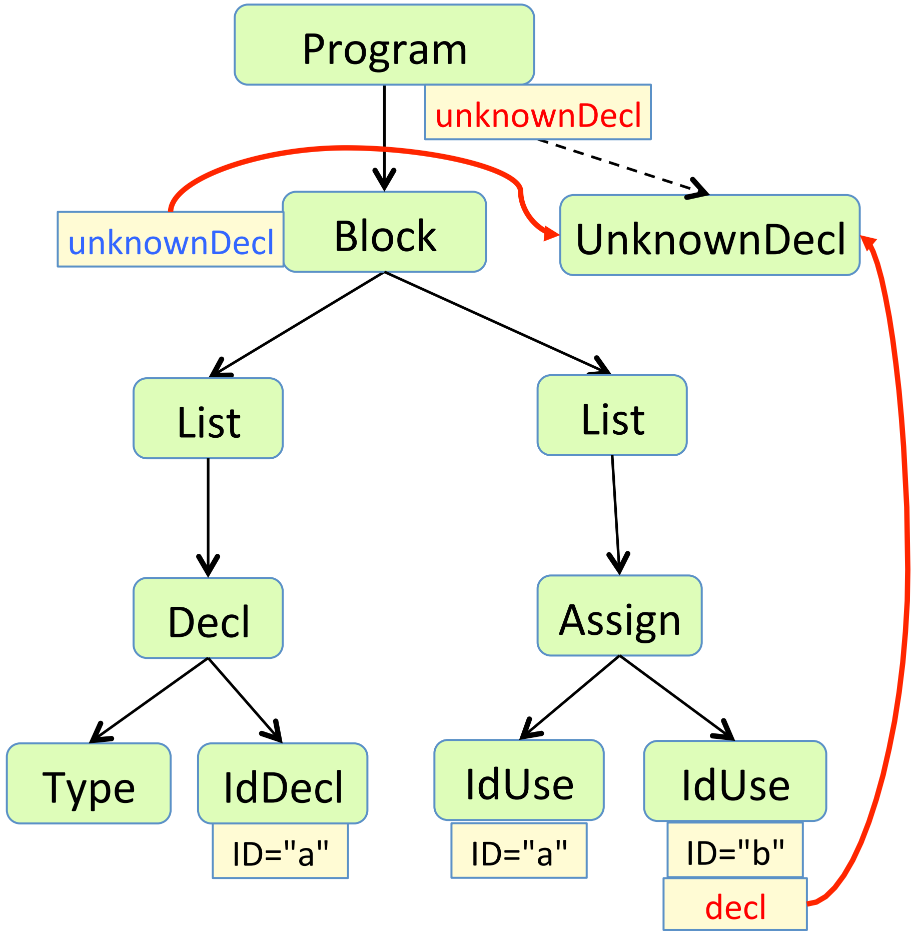
Or declare it in ASTNode. Then it will be visible in all nodes.



Implementing unknownDecl

Apply the Root Attribute pattern:

```
eq Program.getBlock().unknownDecl() =
  unknownDecl();

inh UnknownDecl Block.unknownDecl();
```



Type analysis

Type analysis

Type analysis: compute the type of each expression

Add a type attribute to Expr

```
syn Type Expr.type();
```

Implement it for IdUses

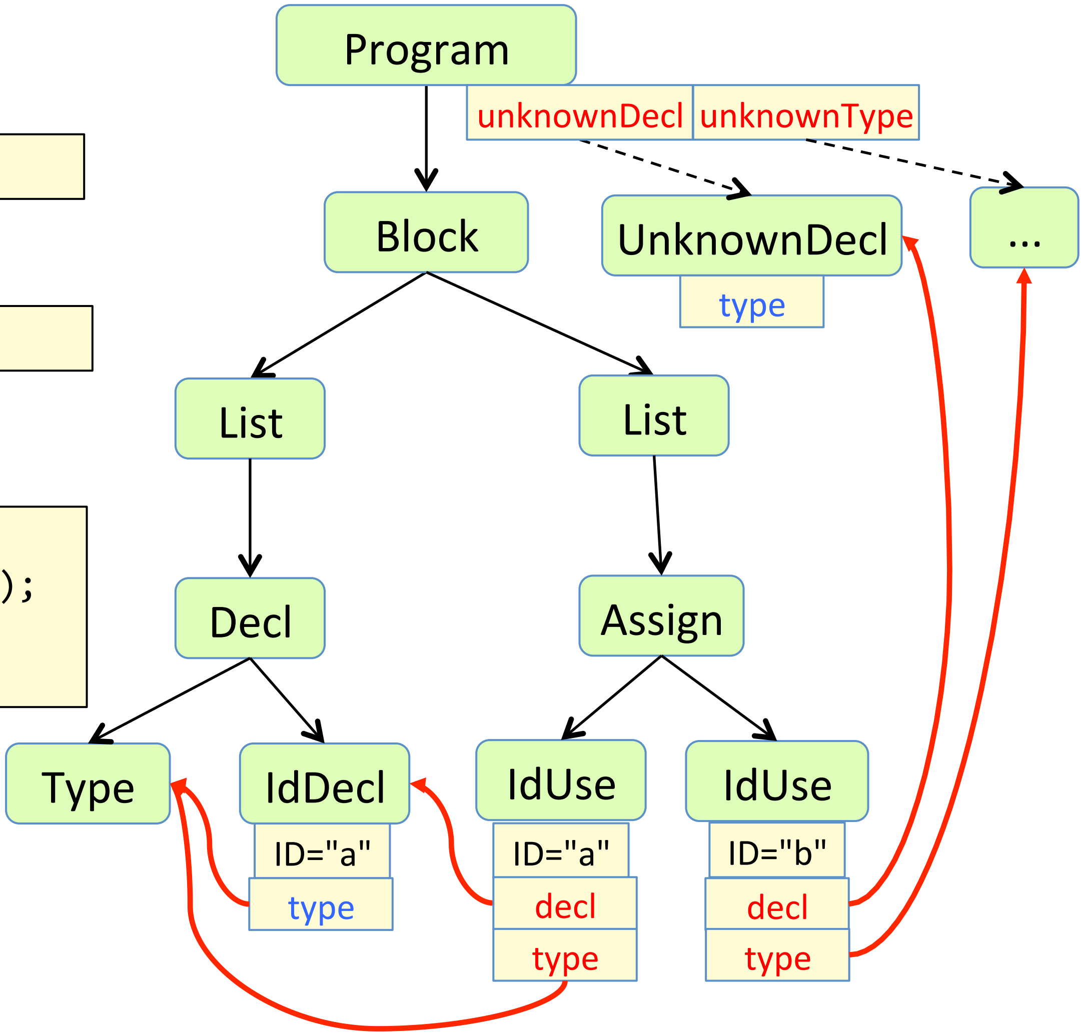
```
eq IdUse.type() = decl().type();
```

Define the type attribute for IdDecls

```
inh Type IdDecl.type();
eq Decl.getIdDecl().type() = getType();
eq Program.unknownDecl().type() =
  unknownType();
```

Define unknownType as an NTA

```
...
```



Type checking

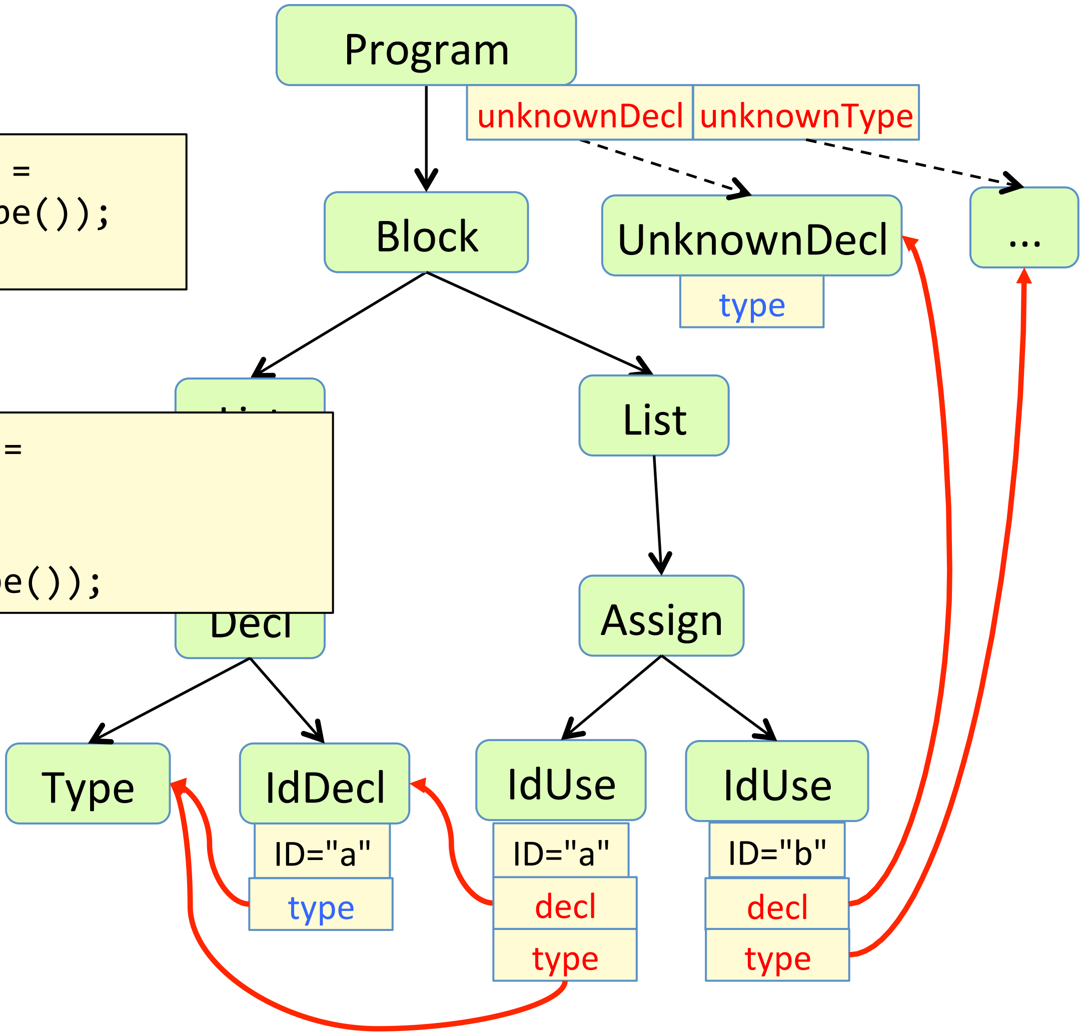
Type checking: Check if types are used correctly

First attempt:

```
syn boolean Assign.compatibleTypes() =
  getTo().type().equals(getFrom().type());
```

Second attempt:

```
syn boolean Assign.compatibleTypes() =
  getFrom().type().isUnknownType() ||
  getTo().type().isUnknownType() ||
  getTo().type().equals(getFrom().type());
```



Problem with first attempt: Missing declaration errors will give type checking errors as well.
Would be nicer to view unknownType as compatible with all other types.

Collection attributes

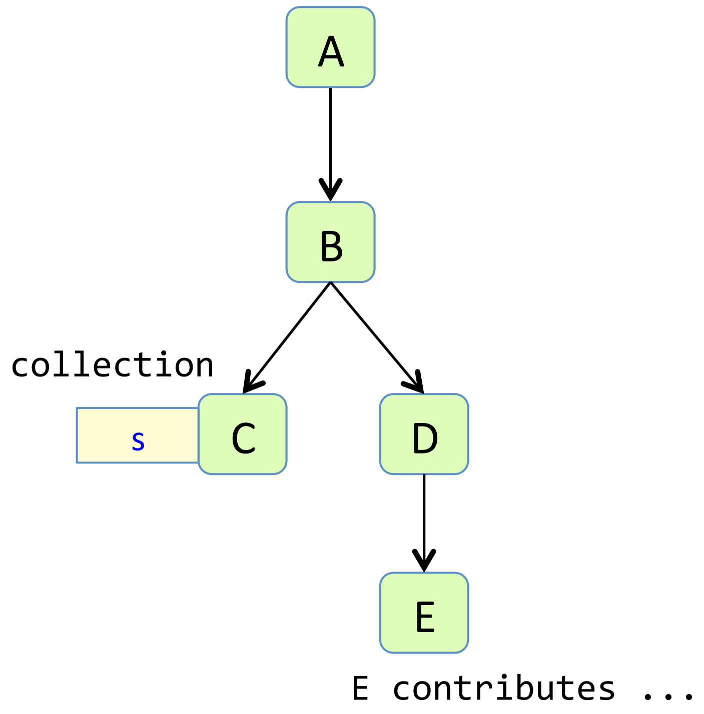
Error checking

Collection attributes

A collection attribute is defined by contributions, instead of by a single equation.

Use for values combined from many small parts spread out over the tree.

Example uses:
- collect compile-time errors in a program
- collect what uses are bound to a specific declaration
- count the number of if-statements in a method



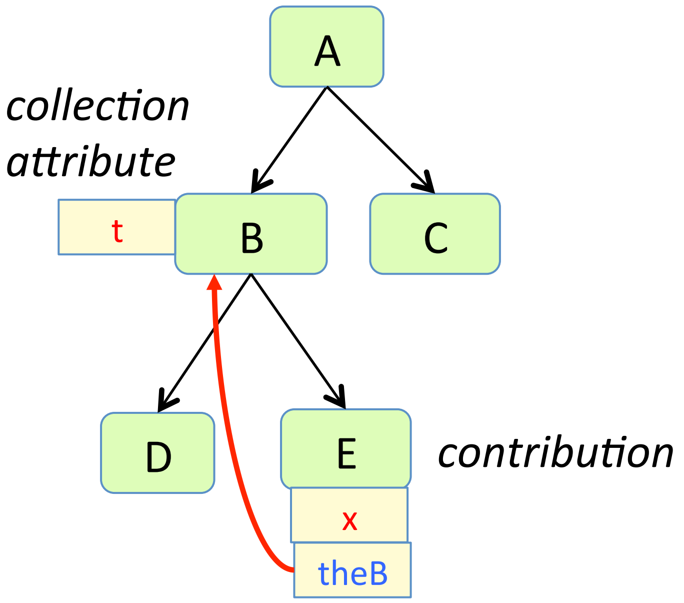
When a collection attribute is accessed, the attribute evaluator will automatically traverse the AST and find the contributions.

Collection attributes

An **collection** attribute has a composite value.

Contribution rules can declare elements that should contribute to the composite value.

The attribute evaluator will automatically traverse the AST starting from a given root, and add the contributions, using a method m which must be **commutative**.



Declare the collection:

```
coll T B.t() [new T()] with m root B;
```

Declare a contribution
(suppose E has an attribute x):

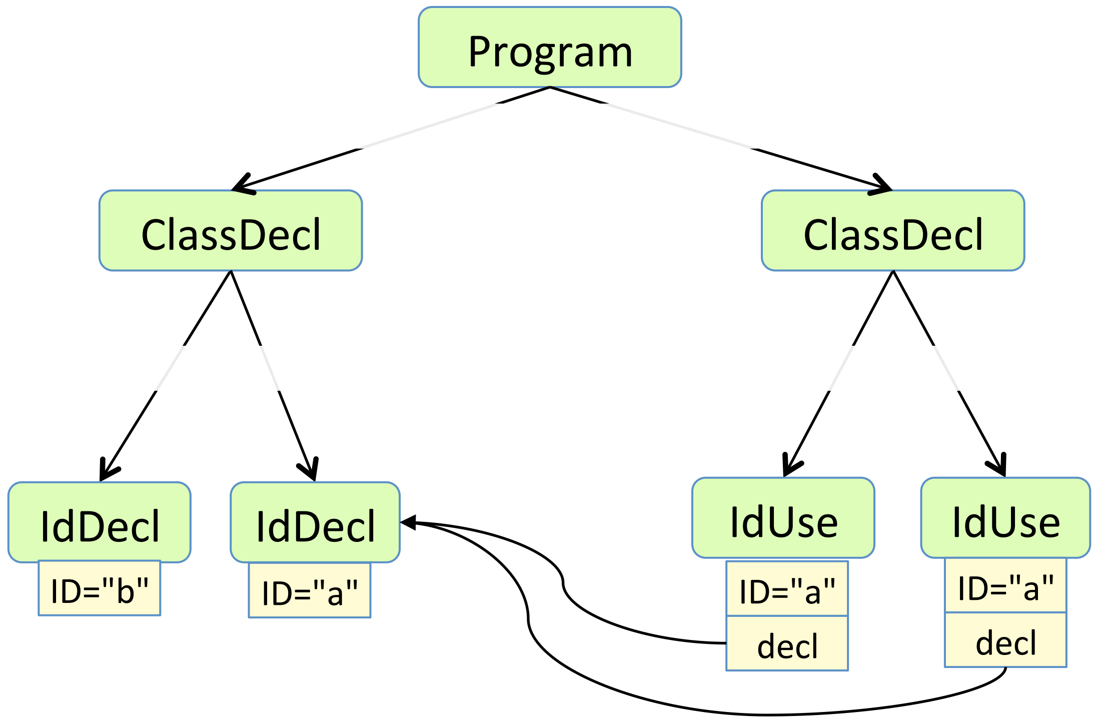
```
E contributes x
  when condition
  to B.t()
  for theB();
```

Propagate theB (Root Attribute pattern):

```
inh B E.theB();
eq B.getChild().theB() = this;
```

Collection attributes, example 1

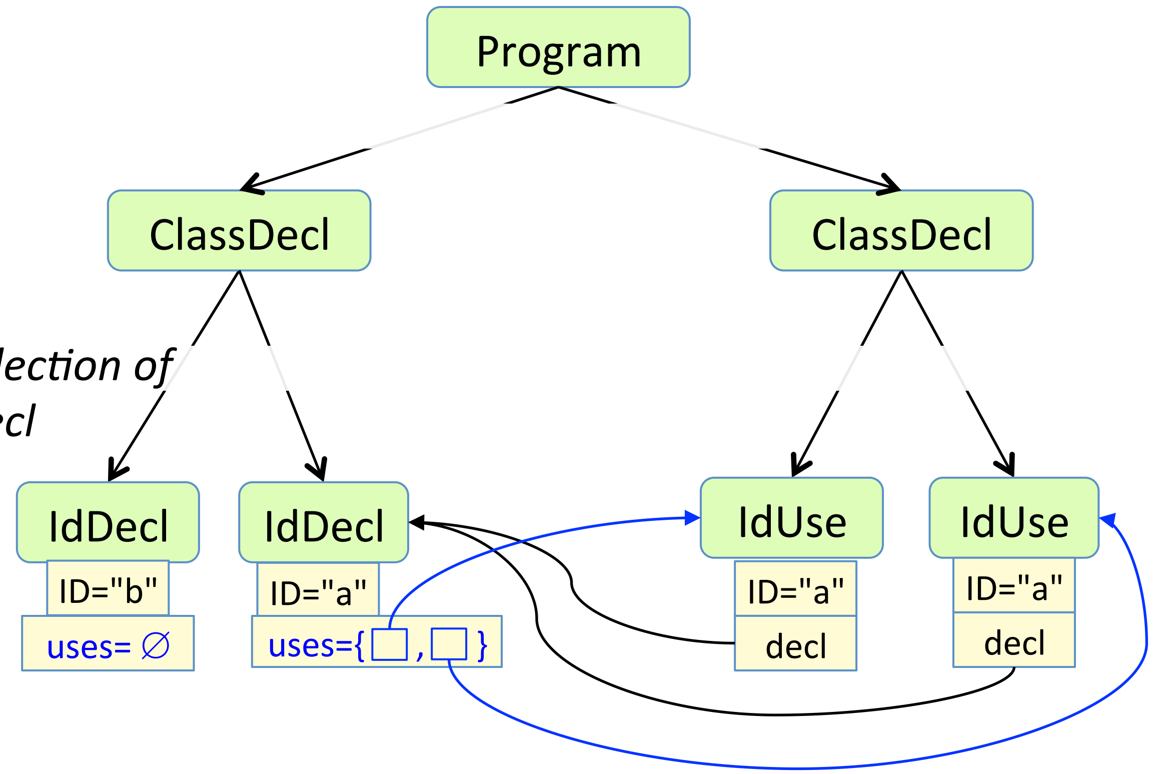
```
...
IdDecl ::= <ID:String>;
IdUse ::= <ID:String>;
```



Collection attributes, example 1

```
...
IdDecl ::= <ID>;
IdUse ::= <ID>;
```

A "uses" attribute contains the collection of IdUses referring to the IdDecl

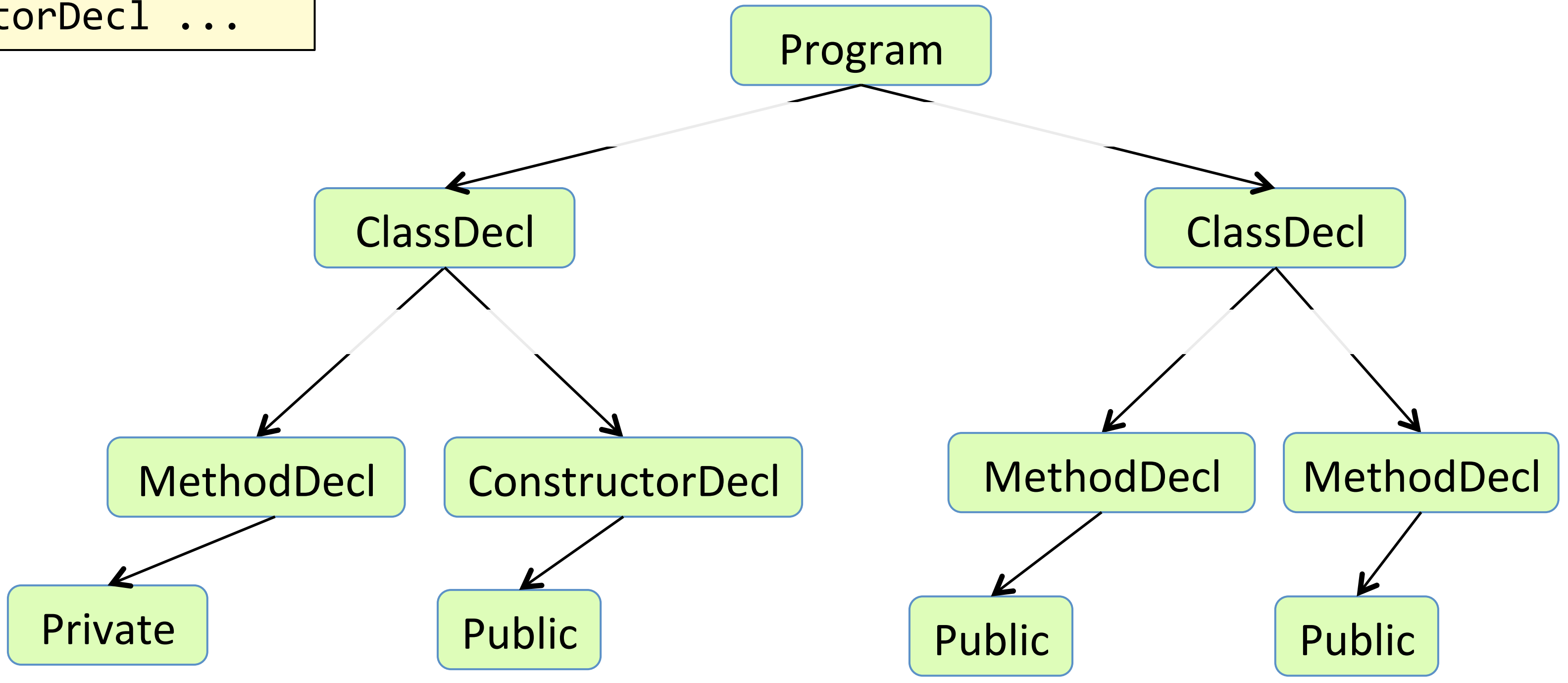


```
coll Set<IdUse> IdDecl.uses() [new Set<IdUse>()] with add;

IdUse contributes this
to IdDecl.uses()
for decl();
```

Collection attributes, example 2

JastAdd Java grammar

```
ClassDecl ...
MethodDecl ...
ConstructorDecl ...
```

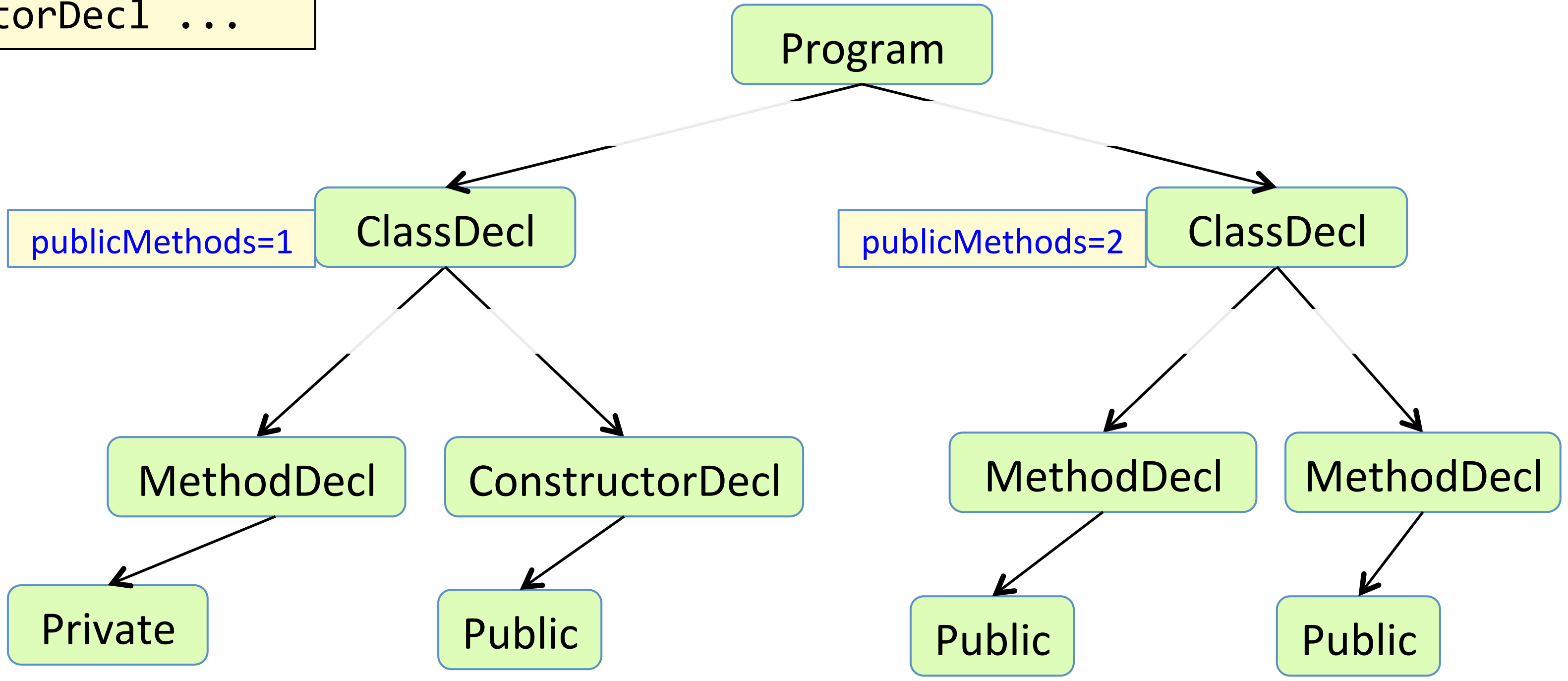


Contributions can be conditional.
Example: NPM metric- Number of Public Methods and constructors in a class

Collection attributes, example 2

JastAdd Java grammar

```
ClassDecl ...
MethodDecl ...
ConstructorDecl ...
```



Contributions can be conditional.

Example: NPM metric- Number of Public Methods and constructors in a class

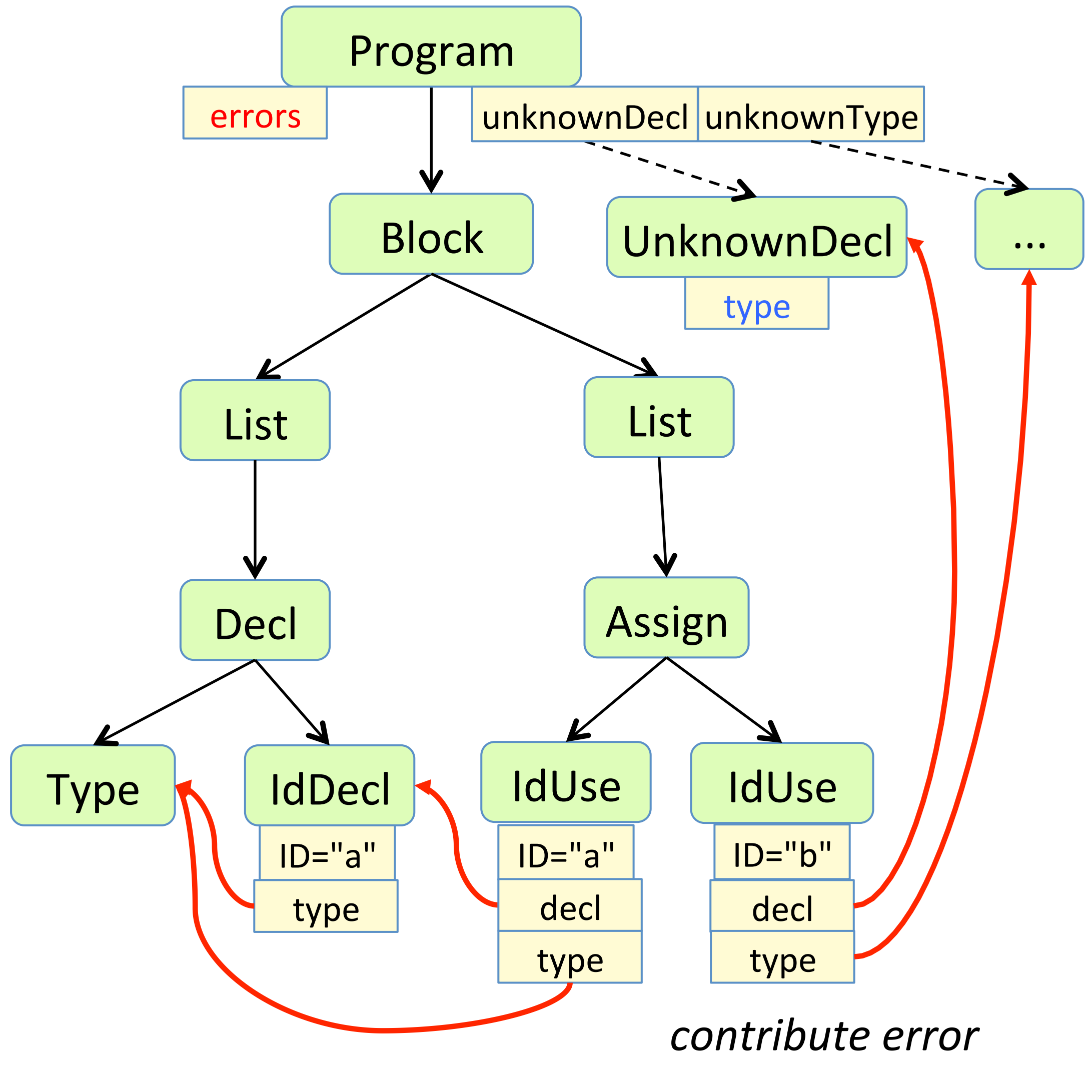
```
coll Counter ClassDecl.publicMethods () [new Counter()] with add;

MethodDecl contributes 1 when isPublic() to ClassDecl.publicMethods() for hostType();
ConstructorDecl contributes 1 when isPublic() to ClassDecl.publicMethods() for hostType();
```

Error checking

Error checking: collect all errors

We would like an attribute errors in the root, containing all error messages.

We would like an easy way to "contribute" different kinds of errors from different nodes in the AST.



Example: Collecting errors

Error checking: collect all errors

Declare the errors collection:

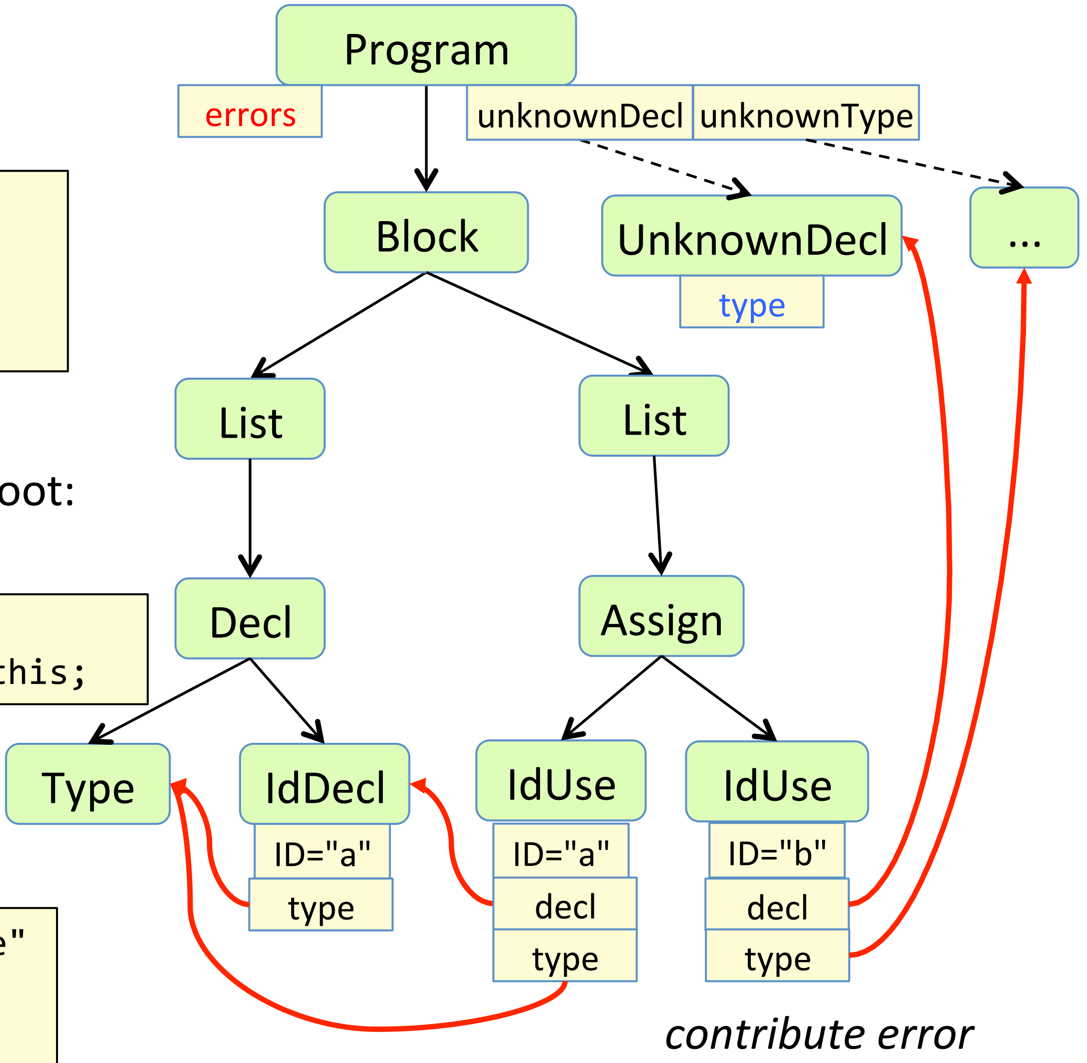
```
coll Set<String> Program.errors()
  [new HashSet<String>()]
  with add
  root Program;
```

Propagate a reference to the Program root:
(Root Attribute pattern):

```
inh Program ASTNode.theProgram();
eq Program.getChild().theProgram() = this;
```

Contribute an error

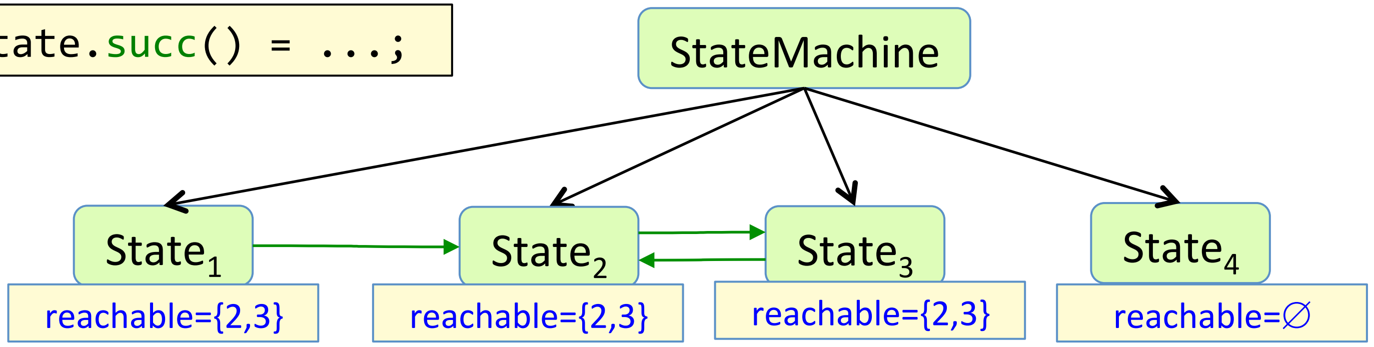
```
IdUse contributes "Undeclared variable"
  when decl().isUnknown()
  to Program.errors()
  for theProgram();
```



Circular attributes

Circular attributes

```
syn Set<State> State.succ() = ...;
```



What states are reachable from state k?   Mathematical definition:

$$reachable_k = succ_k \cup \bigcup_{s_j \in succ_k} reachable_j$$

Implementation using a circular attribute

```
syn Set<State> State.reachable() circular [new HashSet<State>()] {
  HashSet<State> result = new HashSet<State>();
  for (State s : succ()) {
    result.add(s);
    result.addAll(s.reachable());
  }
  return result;
}
```

A circular attribute may depend (transitively) on itself.

Circular attributes - termination

Does the computation terminate?

Yes!
The values (sets of states) can be arranged in a lattice.
The lattice is of finite height (the number of states is finite).
The equations are monotonic: they use set union.

Warning! JastAdd does not check this property. If you use non-monotonic equations or values that can grow unbounded, you might get nontermination.

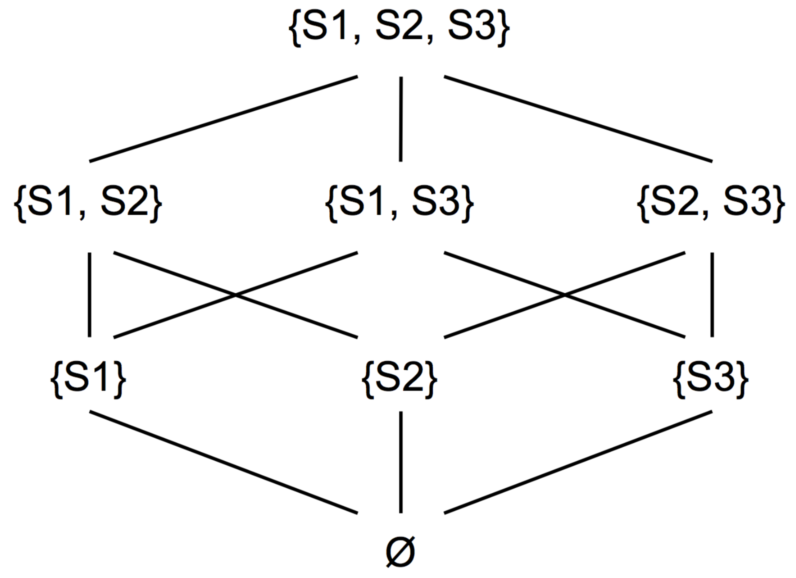
Implementation using a circular attribute

```
syn Set<State> State.reachable() circular [new HashSet<State>()] {
  HashSet<State> result = new HashSet<State>();
  for (State s : succ()) {
    result.add(s);
    result.addAll(s.reachable());
  }
  return result;
}
```

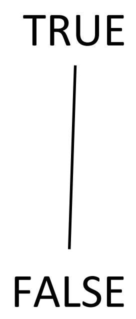
A circular attribute may depend (transitively) on itself.

Useful lattice types

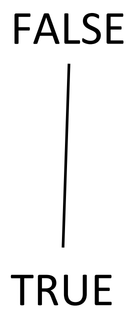
Set lattice
Start with the empty set.
Use the UNION operator.
Make sure there is a finite set of possible values in a set.



Boolean lattices
The lattice is of finite height: only two possible elements



Start with FALSE
Use the OR operator



Start with TRUE
Use the AND operator

Circular attributes – beware of externally visible side effects!

It is ok to use local side effects:

```
syn Set<State> State.reachable() circular [new HashSet<State>()] {
  HashSet<State> result = new HashSet<State>();
  for (State s : succ()) {
    result.add(s);
    result.addAll(s.reachable());
  }
  return result;
}
```

Only the local object is changed. There are no externally visible side effects. This is fine!

Warning! If you by mistake change the value of an attribute, e.g.

```
... s.reachable().add(...) ...
```

JastAdd does not detect this error, and inconsistent attribution may result.

There are many fixed-point problems in compilers and program analysis tools

- Cyclic class hierarchy: find out if a class inherits from itself
- Definite assignment: find out if every variable is guaranteed to have been assigned a value before it is used.
- Call graph analysis: for example, find methods that are never called (dead code)
- Data flow analysis: for example, find variables that are never used (dead code)
- Nullable, FIRST, and FOLLOW (if your "compiler" is actually a parser generator)
- ...

Summary questions:
reference attributes, name analysis

- What is a reference attribute grammar?
- What is a reference attribute?
- What is a parameterized attribute?
- What is name analysis?
- What is a name binding?
- What does scope mean?
- Give examples of some typical name binding rules.
- What does ”declare-before-use” mean?
- What is qualified access?
- How does the Lookup pattern work?

Summary questions:

NTAs, type checking

- What is a nonterminal attribute (NTA)?
- What is the Null Object pattern?
- How does the Root Attribute pattern work?
- Why is it useful to implement missing declarations and unknown types as AST nodes?
- What is type analysis and type checking?
- How can unnecessary error propagation be avoided?

Summary questions:

Collection attributes, Error checking, Circular attributes

- What is a collection attribute?
- How can a collection of error message be implemented?
- What is a circular attribute?
- How is a circular attribute evaluated?
- How can you know if the evaluation of a circular attribute will terminate?
- Give examples of properties that can be computed using circular attributes.