EDAN65: Compilers, Lecture 08

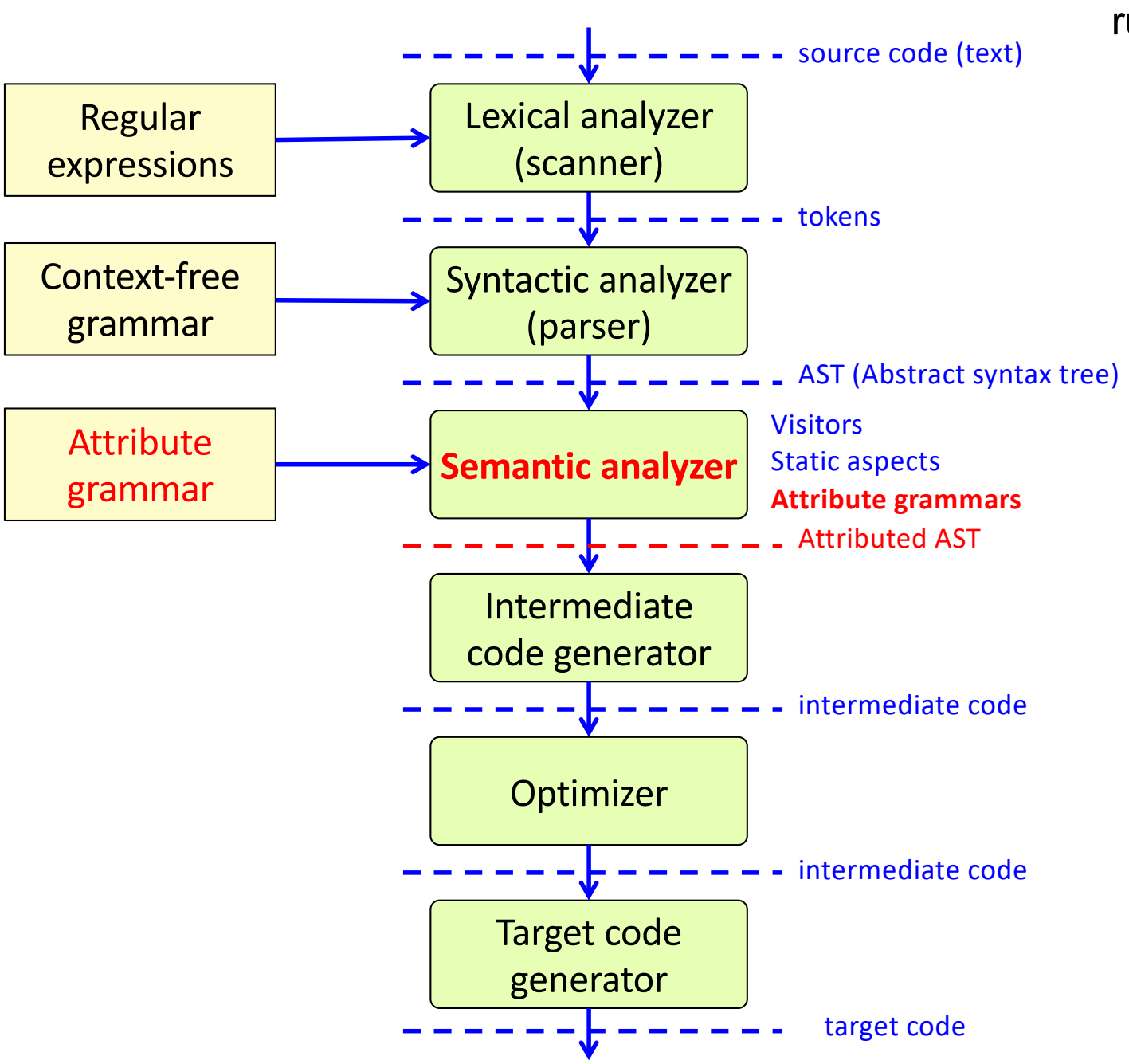
Reference Attribute Grammars

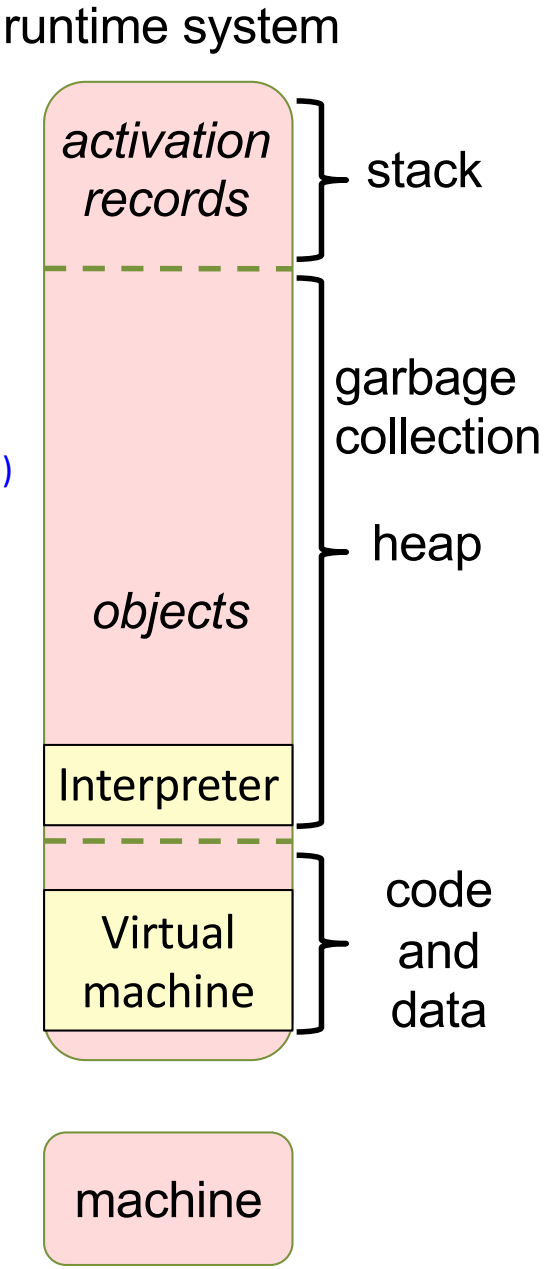
AG mechanisms, Semantic analysis

Görel Hedin

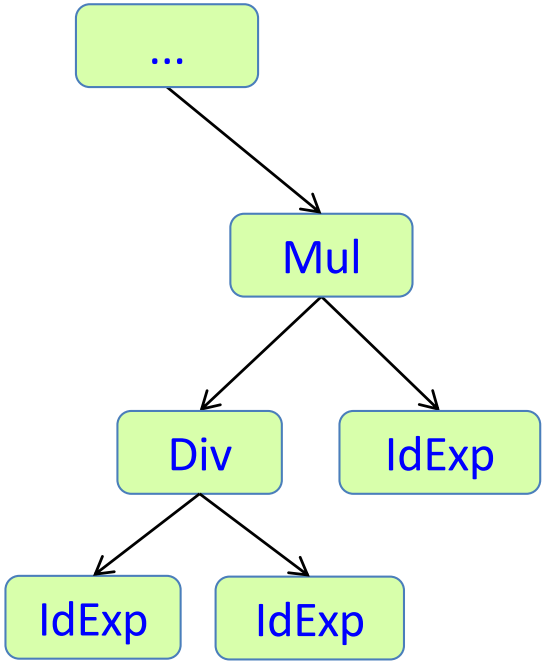
Revised: 2025-09-23

This lecture





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

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 * – an equation for a child holds for its 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)

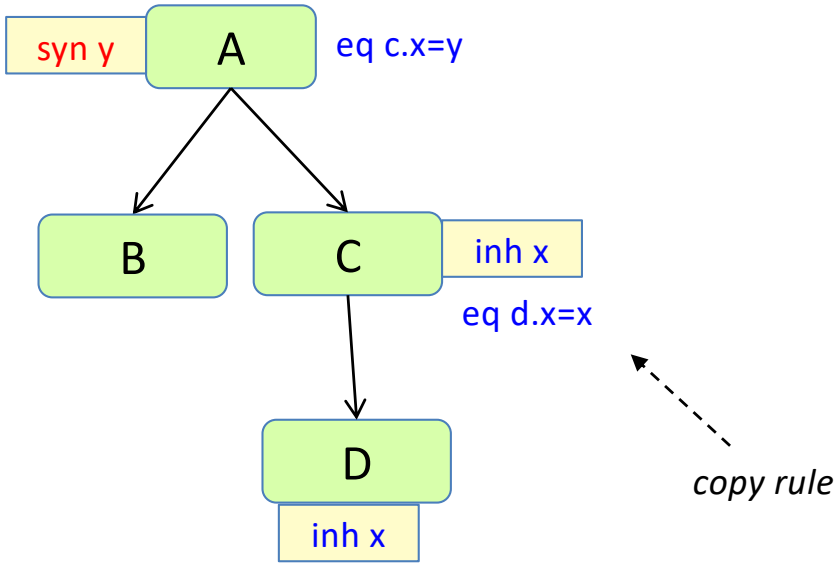
*** Treated in this lecture**

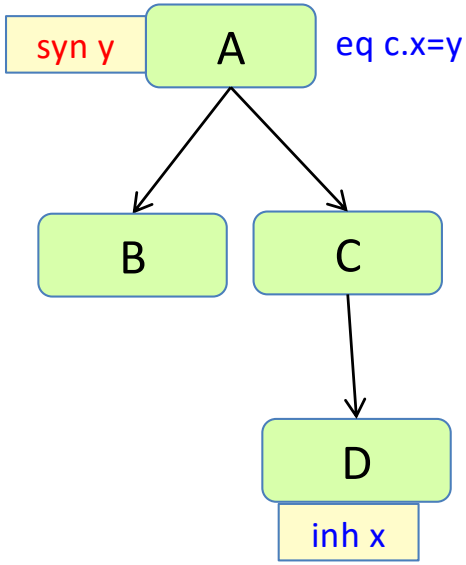
Broadcasting

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.





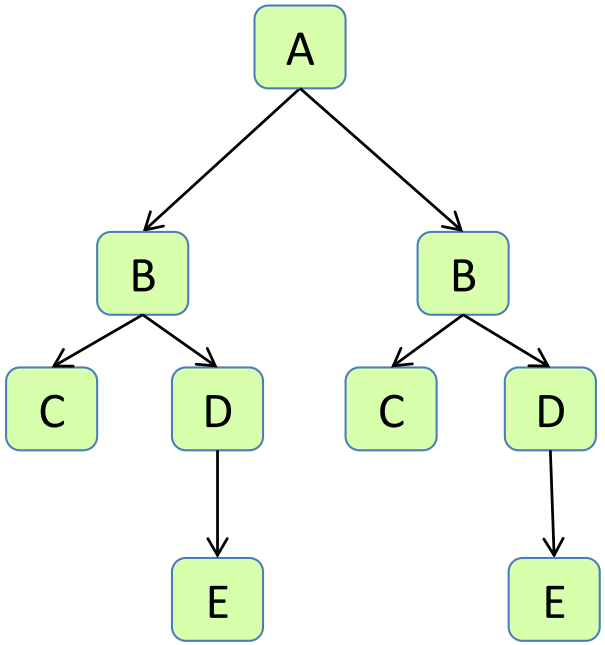
Most AG systems have some shorthand to avoid copy rules

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();
```

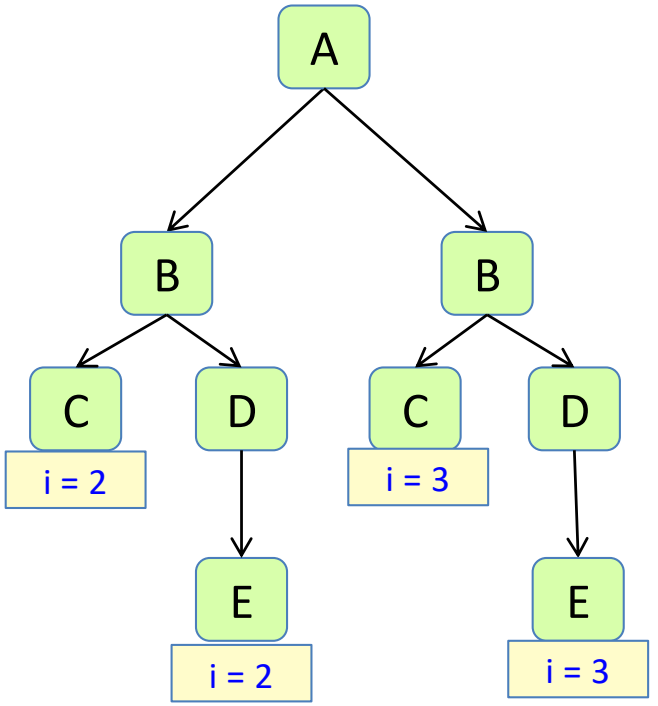
Inherited attributes

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();
```

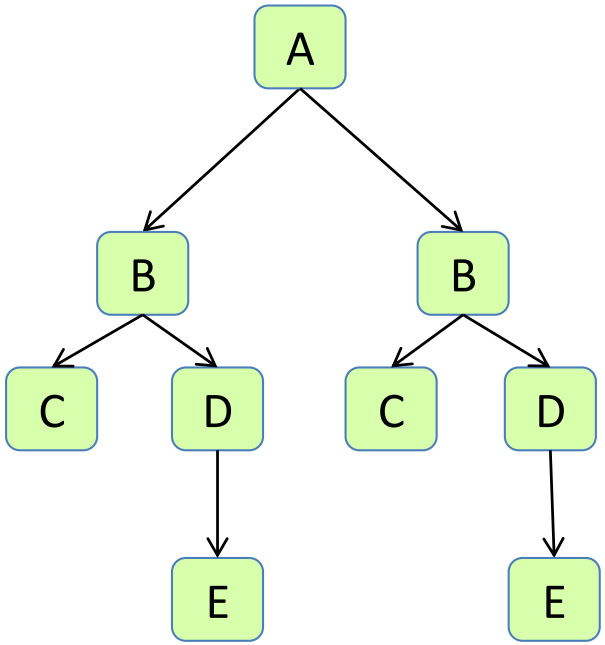


Inherited attributes

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!




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

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

```
eq B.getD().i() = i() + 5;
inh int B.i();
```

Inherited attributes

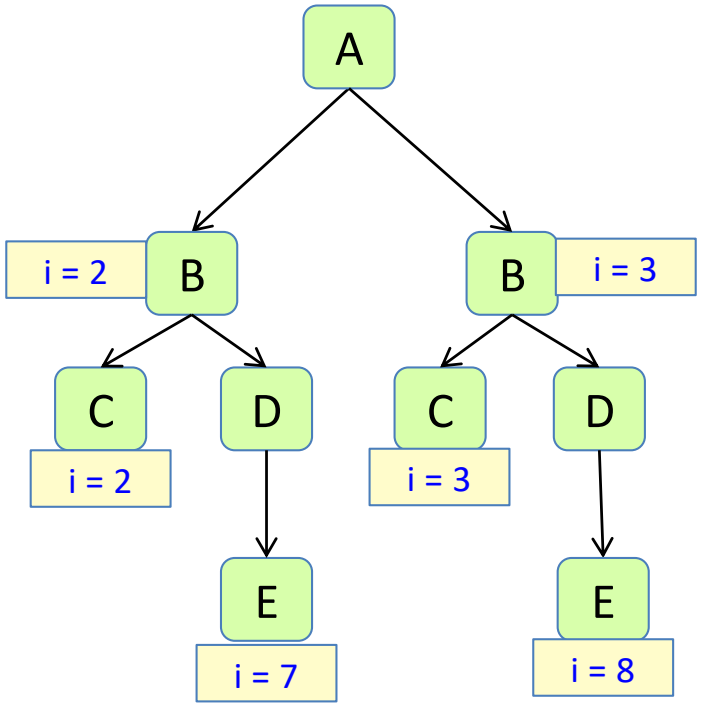
broadcasted equation can be overruled in subtree

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

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

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

```
eq B.getD().i() = i() + 5;
inh int B.i();
```

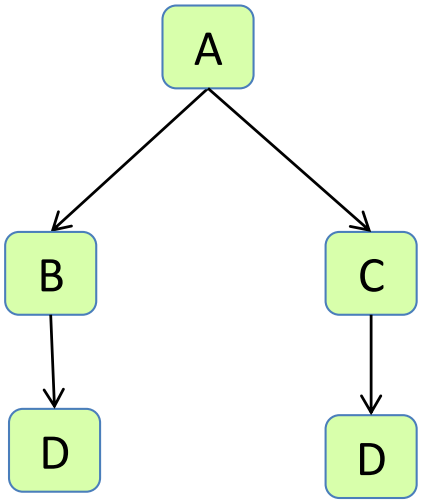


Inherited attributes

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();
```

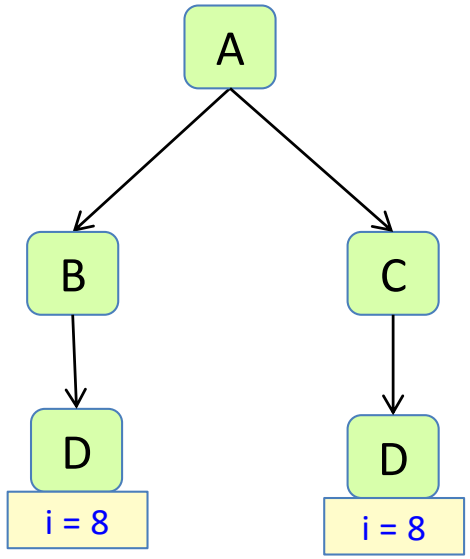
Inherited attributes

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:

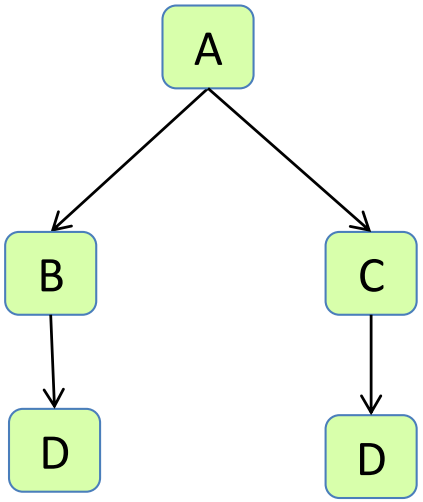
```
eq A.getB().i() = 8;
eq A.getC().i() = 8;
inh int D.i();
```

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;
```

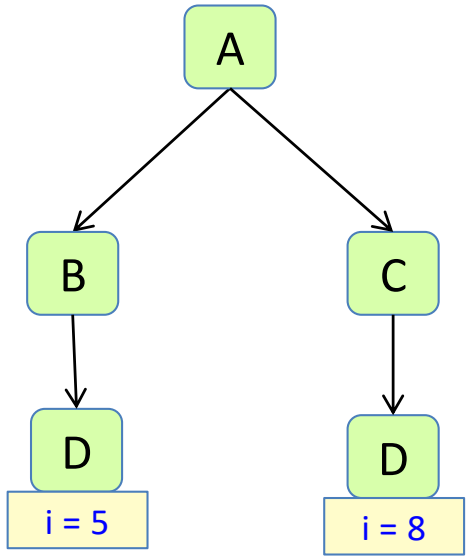


Inherited attributes

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;
```

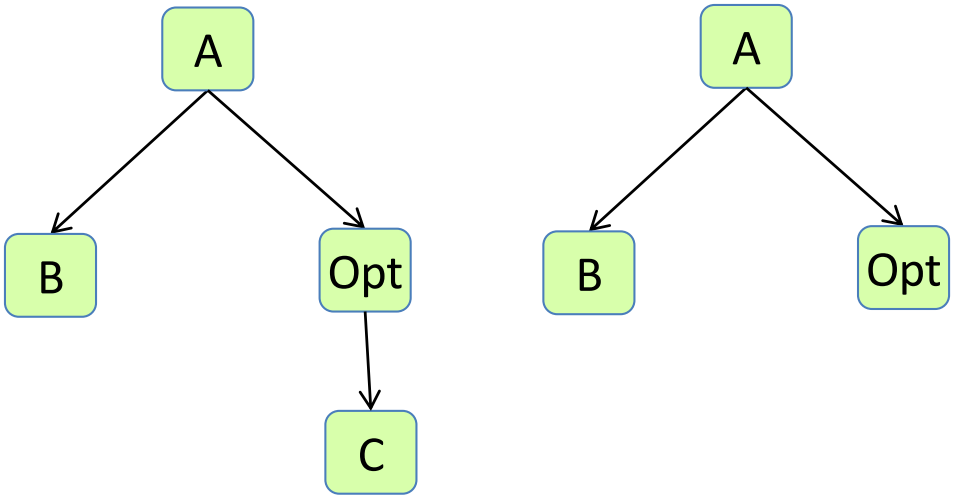


Inherited attributes

defining attributes for optional children

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

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

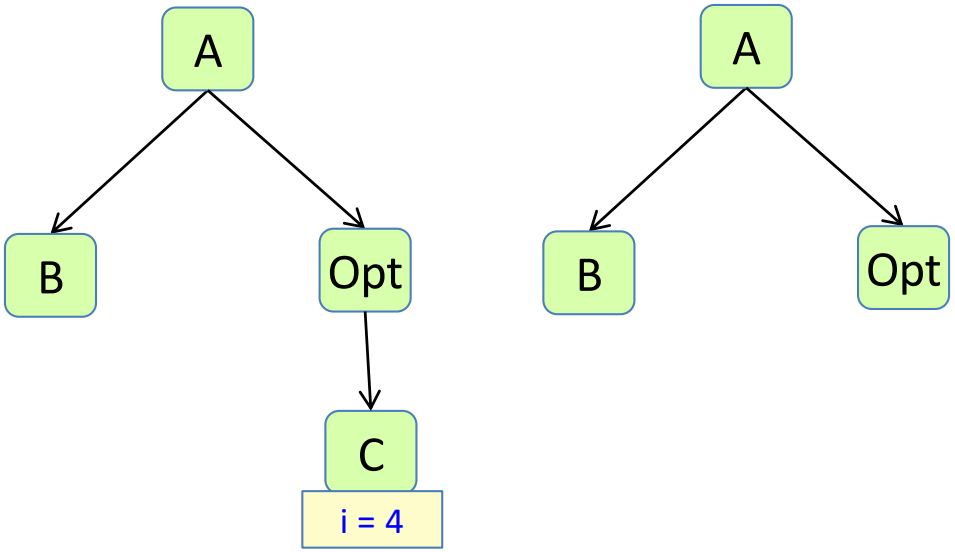


Inherited attributes

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.

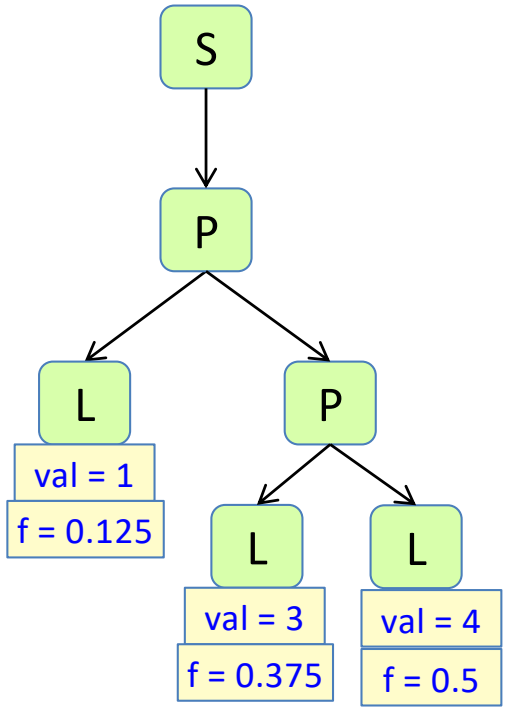
Fractions example revisited

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();
```

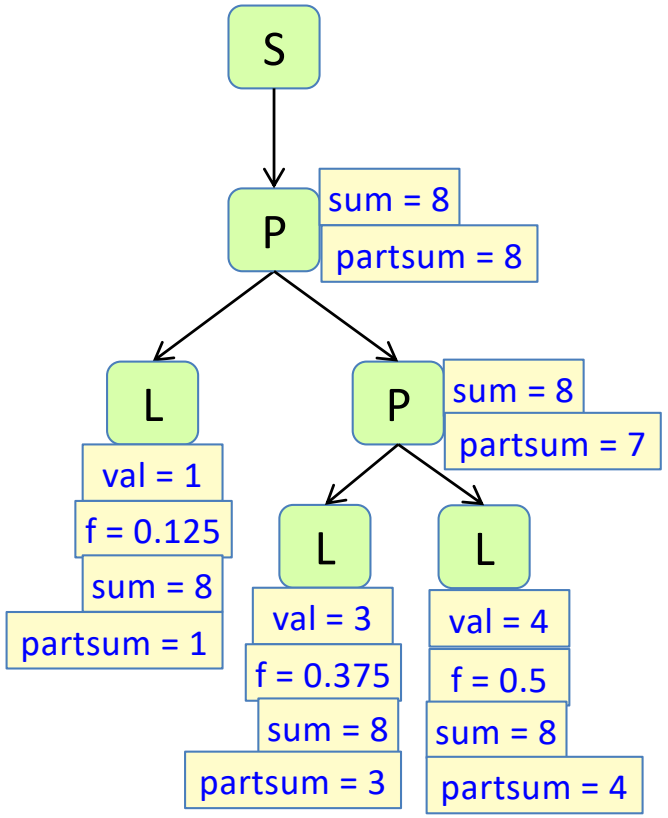


Fractions example

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();
```

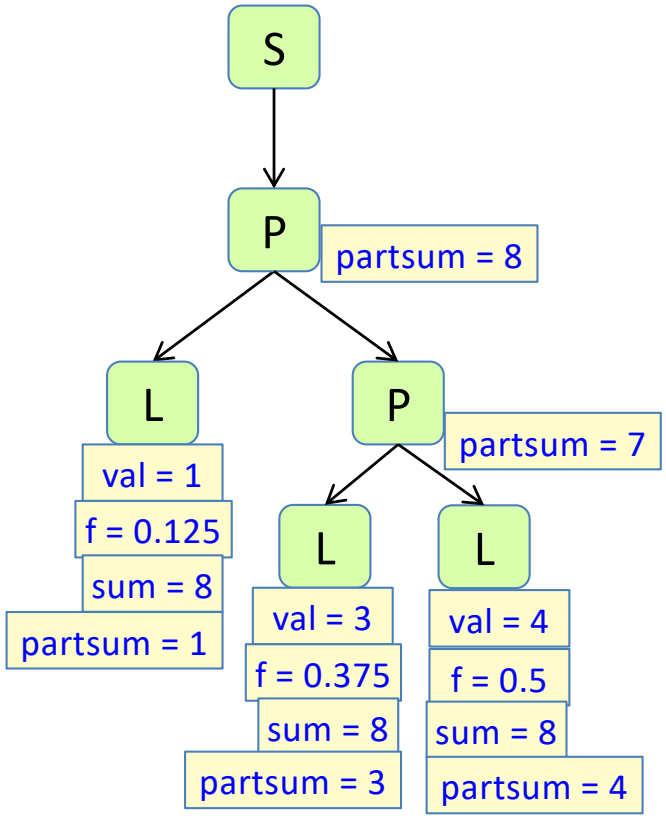


Fractions example

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();
```



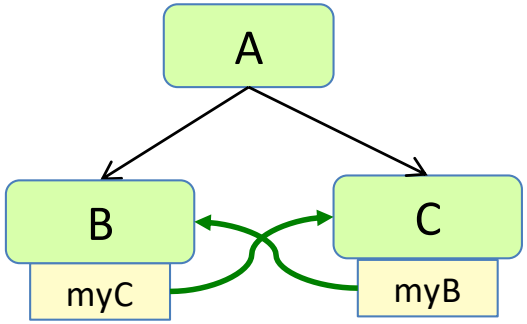
Because of broadcasting, the copy equations are unnecessary.

Reference attributes

Reference attributes

for defining graphs on top of the AST

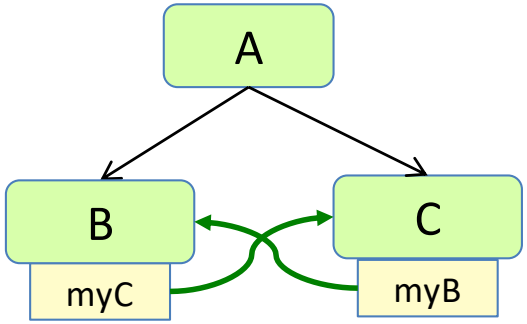
```
A ::= B C;
B;
C;
```



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.

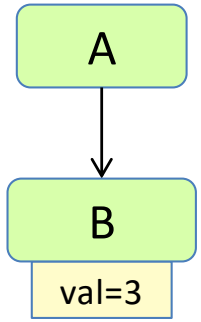
Parameterized attributes

Parameterized attributes

an attribute can have one or more parameters

Example: Find out if B's val is over some given limit

```
A ::= B;
B ::= <val:int>;
```



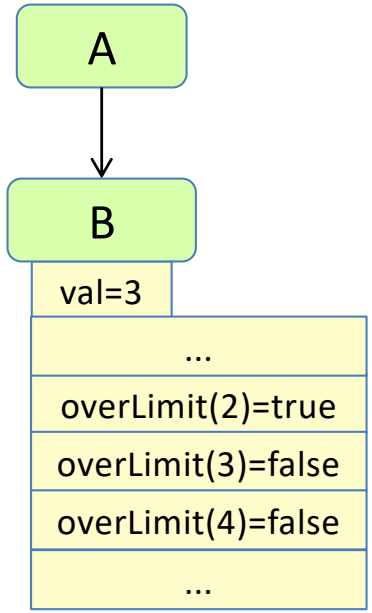
Parameterized attributes

an attribute can have one or more parameters

Example: Find out if B's val is over some given limit

```
A ::= B;
B ::= <val:int>;
```

```
syn boolean B.overLimit(int limit) =
      getval() > limit;
```



Unbounded number of attribute instances – one for each argument.
Similar to functions. But accessed values are cached.
Only accessed attribute instances will be evaluated.

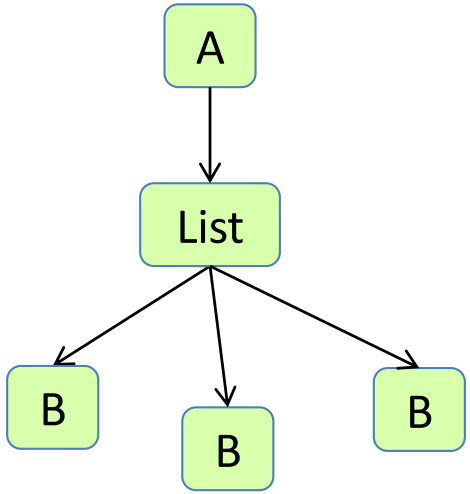
Parameterized attributes

list equations can use both index and parameters

Draw some isBefore attributes and their values!

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

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

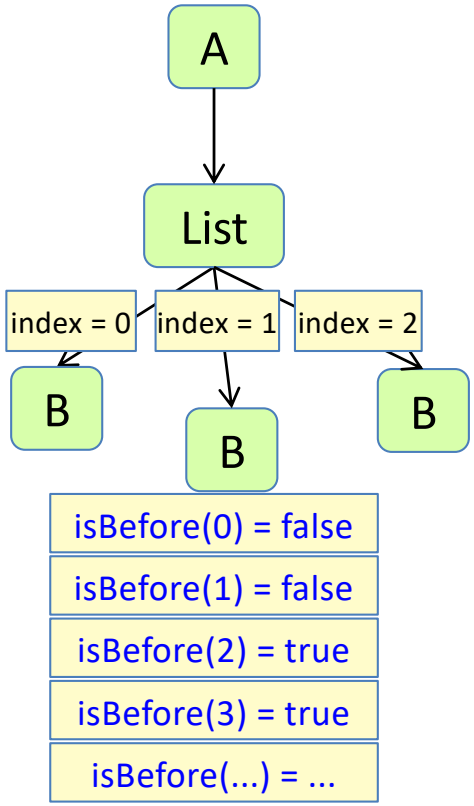


Parameterized attributes

list equations can use both index and parameters

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

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



Name analysis

Name analysis

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

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

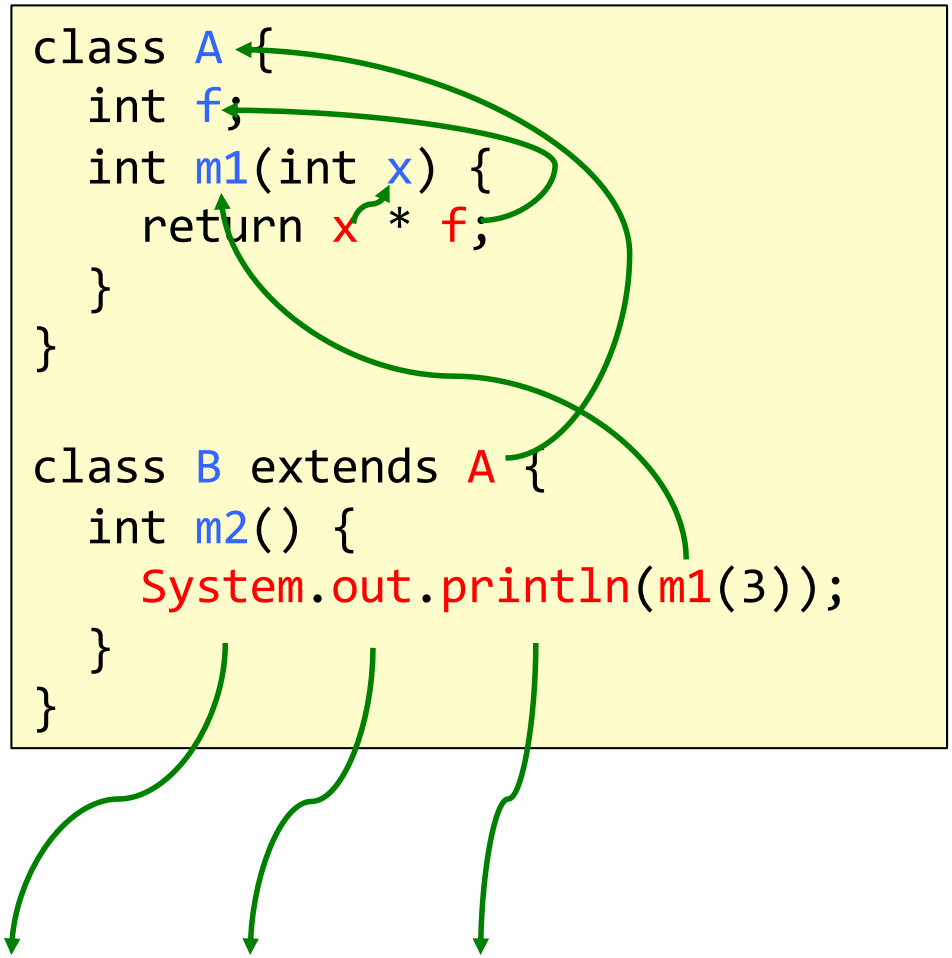
Name analysis: bind each use of an identifier to its declaration

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*.

Name analysis



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, imports
- 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: 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?

Which declaration of y is y bound to?

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.

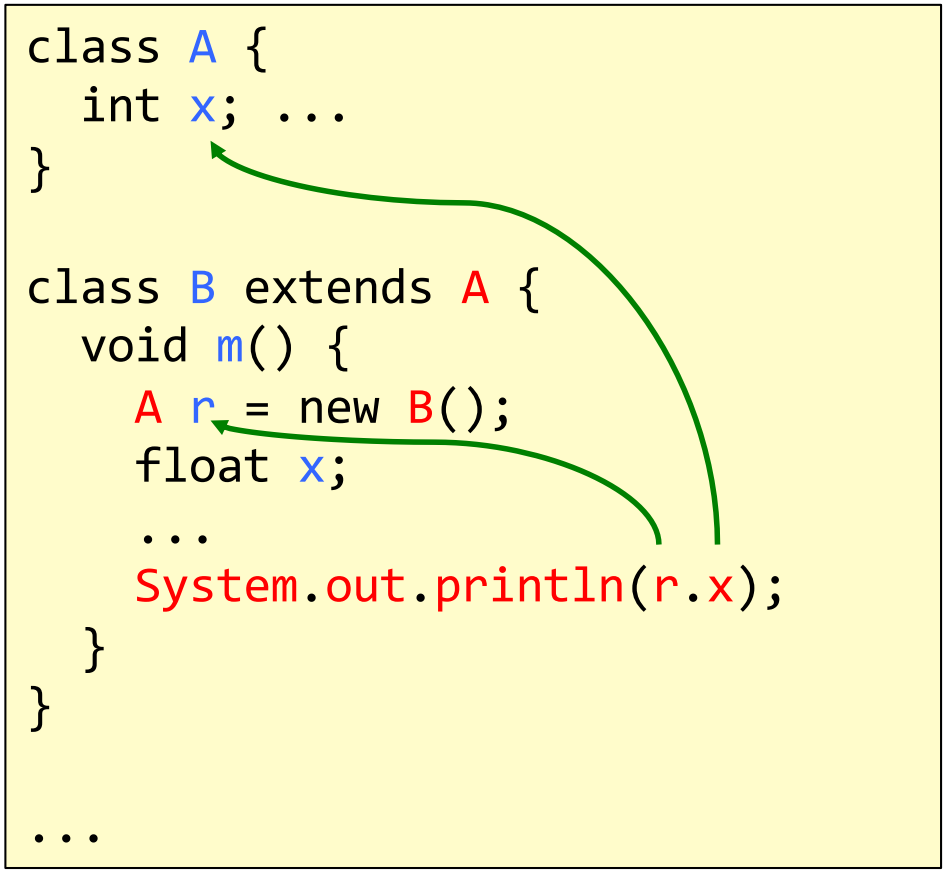
Name binding: Qualified access

```
class A {
  int x; ...
}

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

...
```

Qualified access (dot notation)

The binding of x depends on the *type* of r.

Name binding: Qualified access

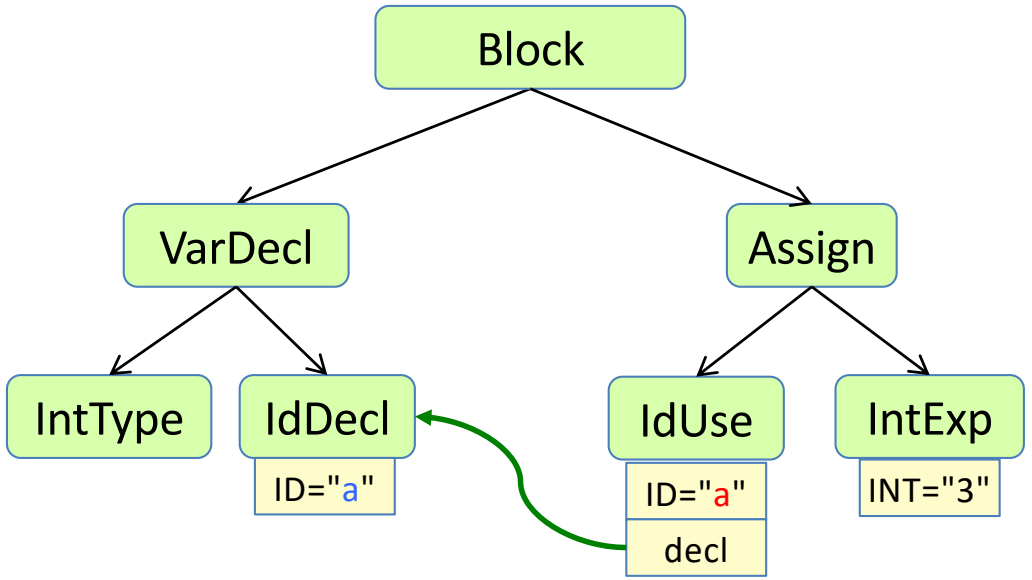


Qualified access (dot notation)

The binding of x depends on the *type* of r.

Recall:
Representing name bindings in an AST

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

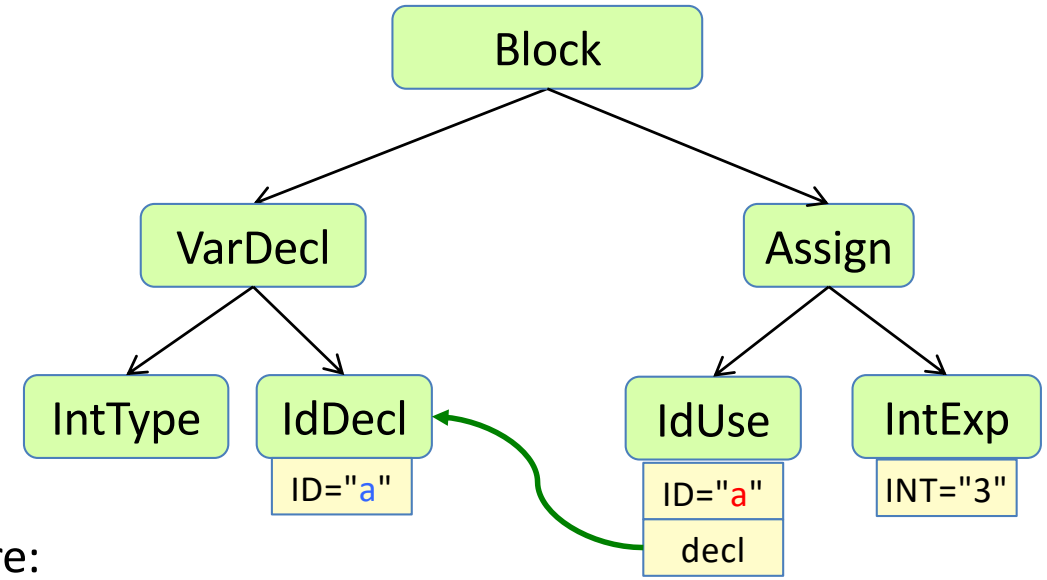


IdDecl for declared names
IdUse for used names

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 very complex.

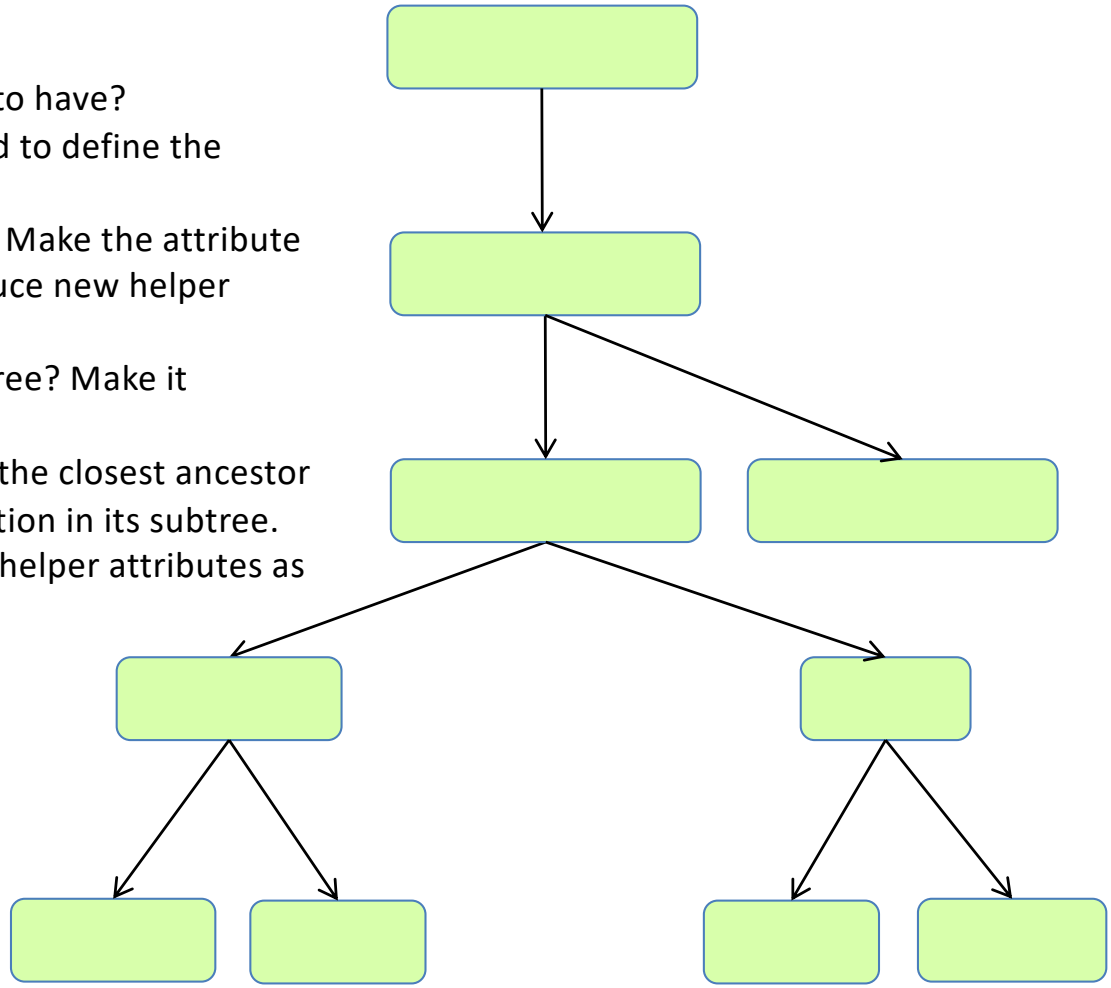
- What if we extend the language?
  Need to change the algorithm.

Solution: Attribute grammars – a declarative approach

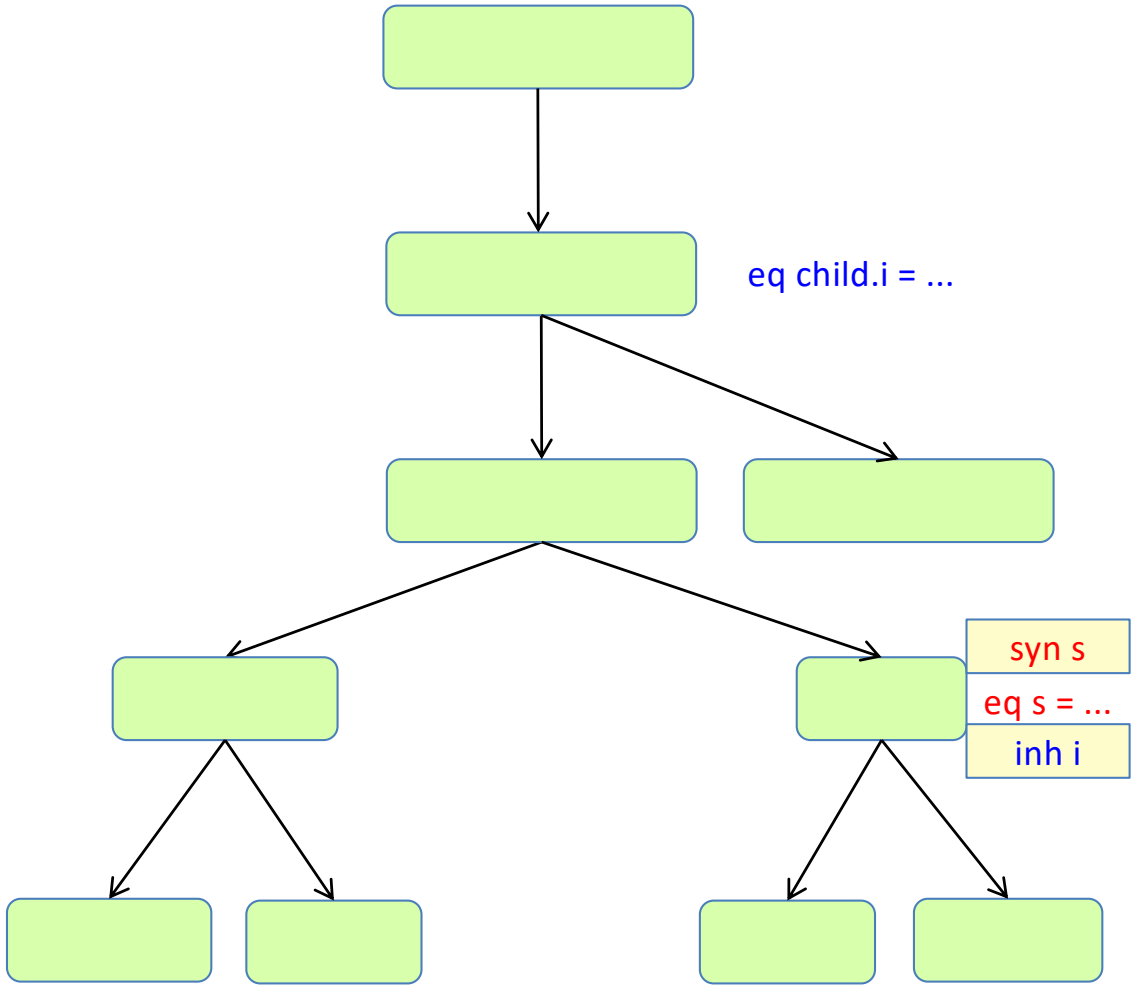
Attribute grammars

Think declaratively!!

1. What attributes would I like the nodes to have?
2. Where is the information that is needed to define the attribute?
3. Is some of it in the node or its subtree? Make the attribute **synthesized**. Write its **equation**. Introduce new helper attributes as needed.
4. Is all of it outside the node and its subtree? Make it **inherited**.
5. To define an inherited attribute, locate the closest ancestor which has some of the needed information in its subtree. Put the **equation** there. Introduce new helper attributes as needed.



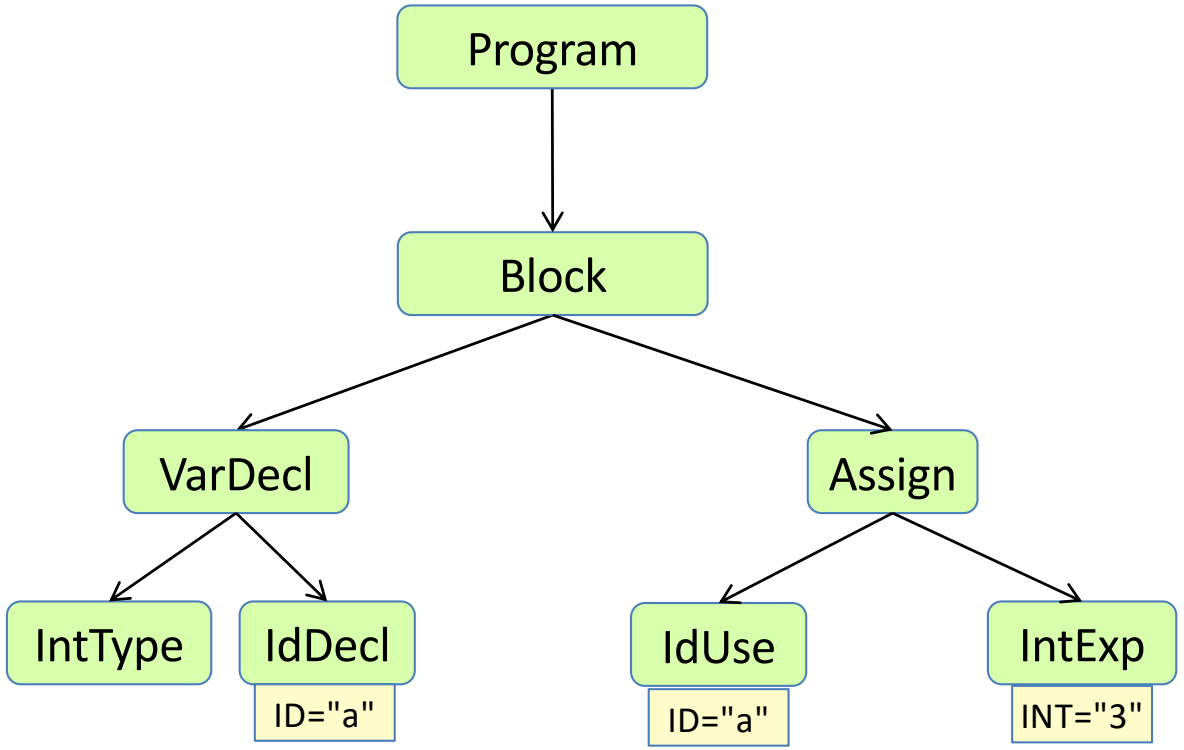
Attribute grammars



Synthesized – value defined in the node
Inherited – value defined in an ancestor

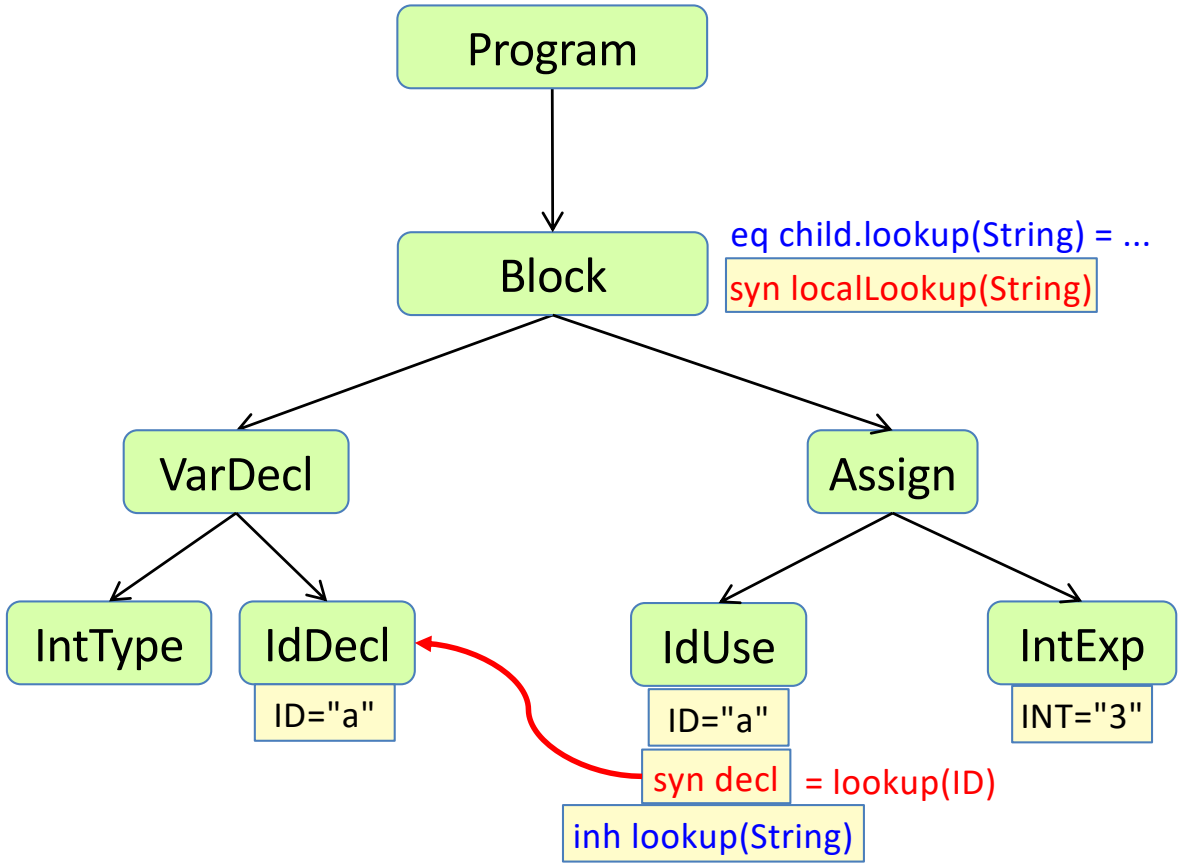
The Lookup Pattern for Name analysis

Name analysis using RAGs



Synthesized – value defined in the node
Inherited – value defined in an ancestor

Name analysis using RAGs



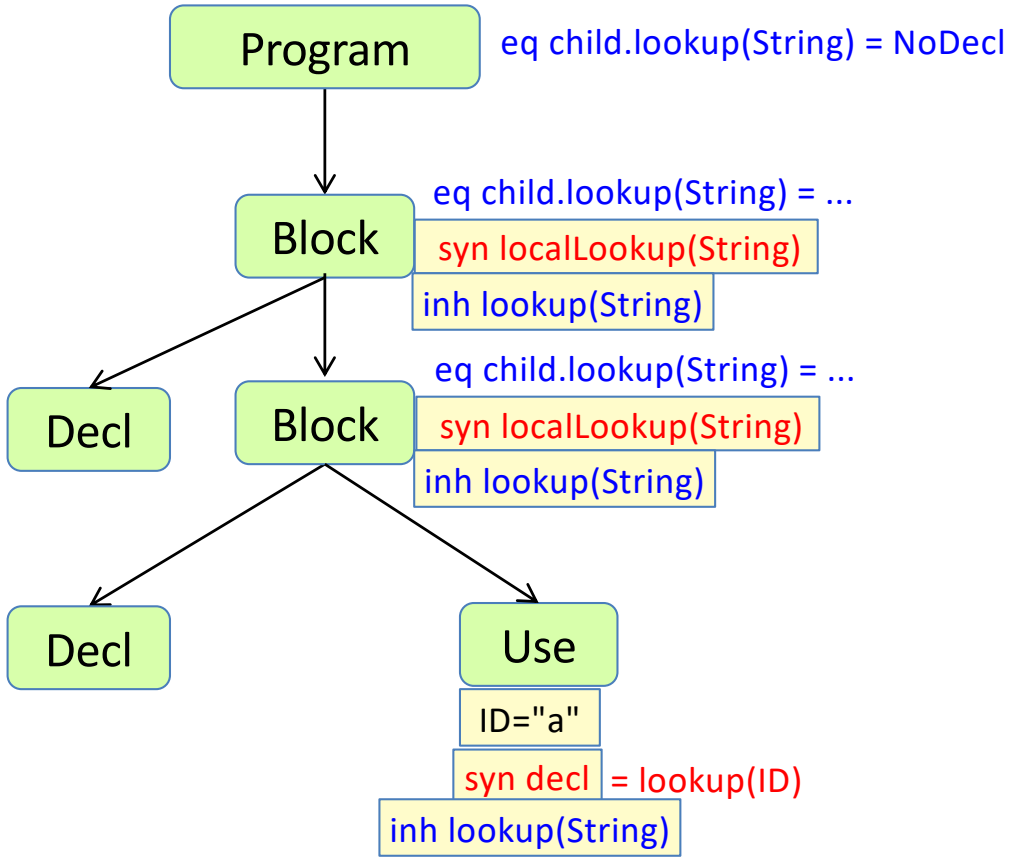
Synthesized – value defined in the node
Inherited – value defined in an ancestor

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

syn decl – the name binding
inh lookup(String) – finds the declaration
syn 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, ...

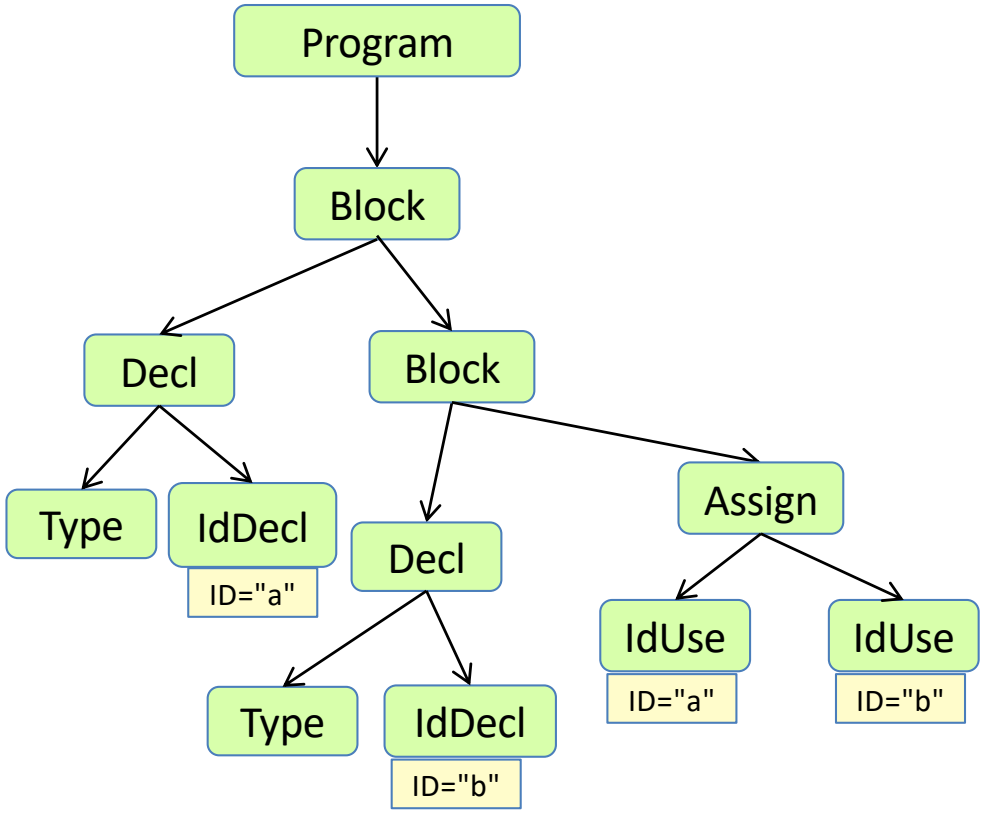
The **Lookup pattern** for name analysis in RAGs



Example implementation in JastAdd

Abstract grammar:

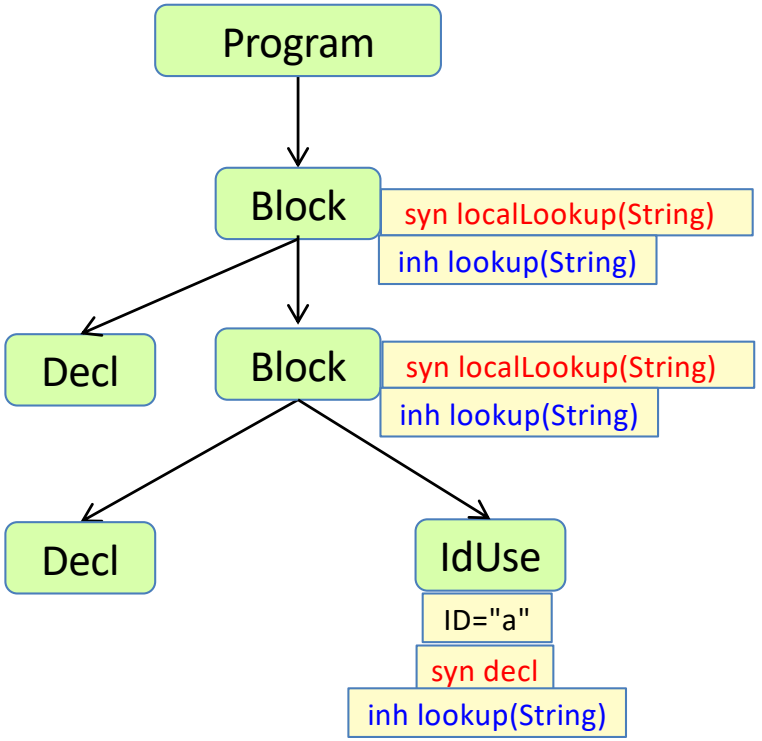
```
Program ::= Block;
Block : Stmt ::= Decl* Stmt*;
Decl ::= Type IdDecl;
IdDecl ::= <ID:String>;
Type;
abstract Stmt;
Assign : Stmt ::= To:IdUse From:IdUse;
IdUse ::= <ID:String>;
```



Example implementation in JastAdd

Abstract grammar:

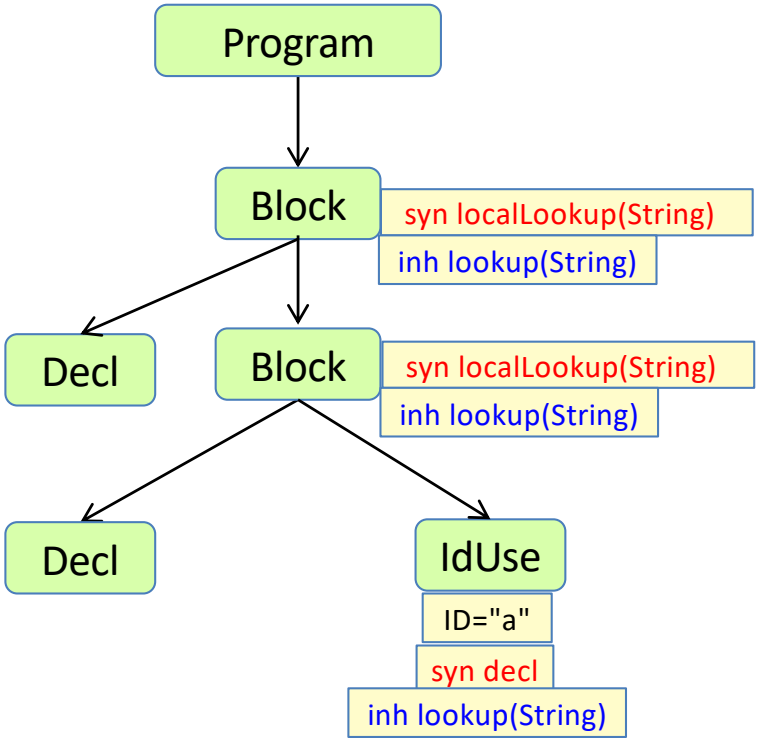
```
Program ::= Block;
Block : Stmt ::= Decl* Stmt*;
Decl ::= Type IdDecl;
IdDecl ::= <ID:String>;
Type;
abstract Stmt;
Assign : Stmt ::= To:IdUse From:IdUse;
IdUse ::= <ID:String>;
```



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);
}

inh IdDecl Block.lookup(String s);

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

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

The Local Map pattern

replace repeated search by map

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 null;
}
```

What happens if there are 1000 elements in the declaration list, and one use for each?
What is the complexity?

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 null;
}
```

What happens if there are 1000 elements in the declaration list, and one use for each?
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) {
  return localMap().get(s);
}

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

Non-terminal attributes (NTAs)

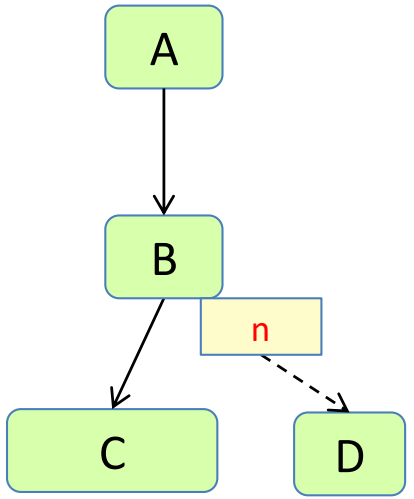
also known as *higher-order attributes*

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

The right-hand side of its defining equation must be a **fresh (new) 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



an NTA

Non-terminal attributes (NTAs)

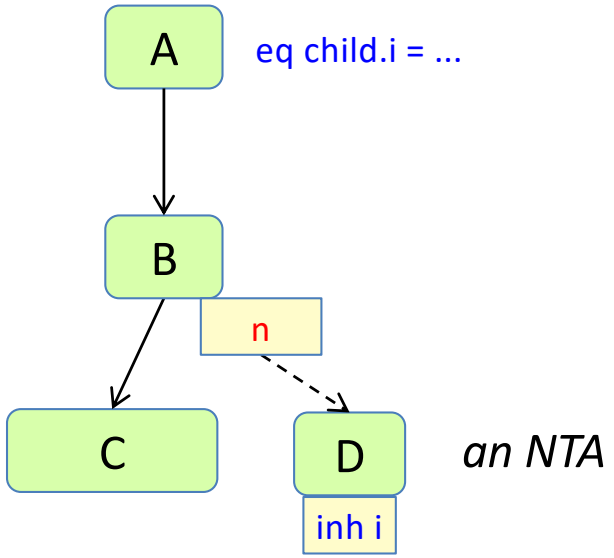
also known as *higher-order attributes*

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

The right-hand side of its defining equation must be a **fresh (new) 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 D B.n() = new D();
```

An NTA is also known as a *higher-order* attribute:
It can itself have attributes.
The owning node (or its ancestors) must define the inherited attributes of the NTA.

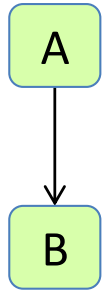
*Warning! Remember to use **fresh** objects!*
If you reuse existing nodes for NTAs, the AST will be inconsistent.
JastAdd does not check this.

NTA example

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

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

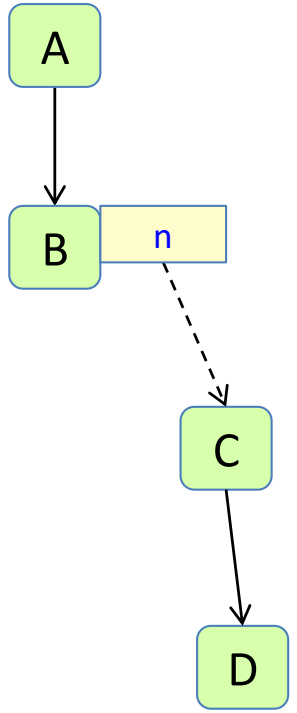
Draw the n attribute and its value!



NTA example

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

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



Nonterminal attributes (NTAs)

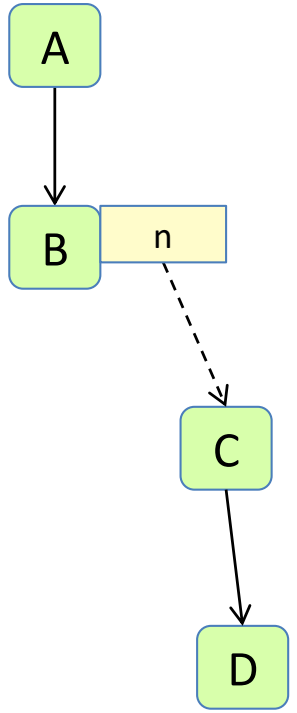
```
A ::= B;
B;
C ::= D;
D;
```

```
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;
```

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



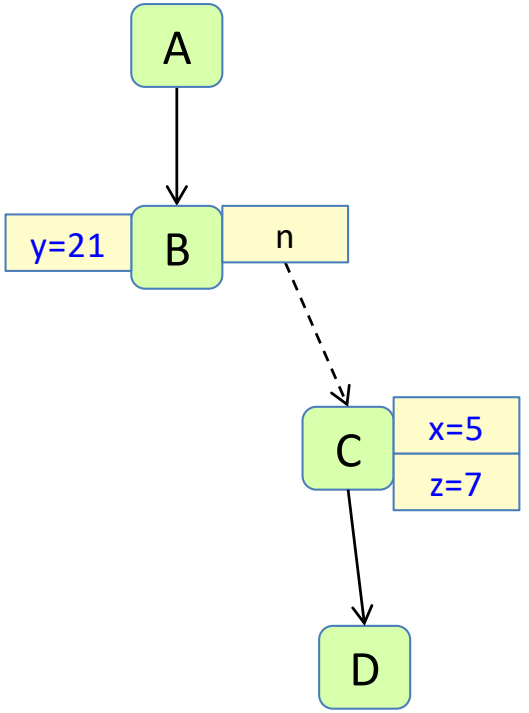
Nonterminal attributes (NTAs)

```
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

The **Null object pattern**

Null is not a very attractive way of representing
a missing declaration. Use a **real object** instead.
Give the object suitable properties (attributes).
The code becomes simpler.

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

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

The **Null object pattern**

Null is not a very attractive way of representing a missing declaration. Use a **real object** instead. Give the object suitable properties (attributes). The code becomes simpler.

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

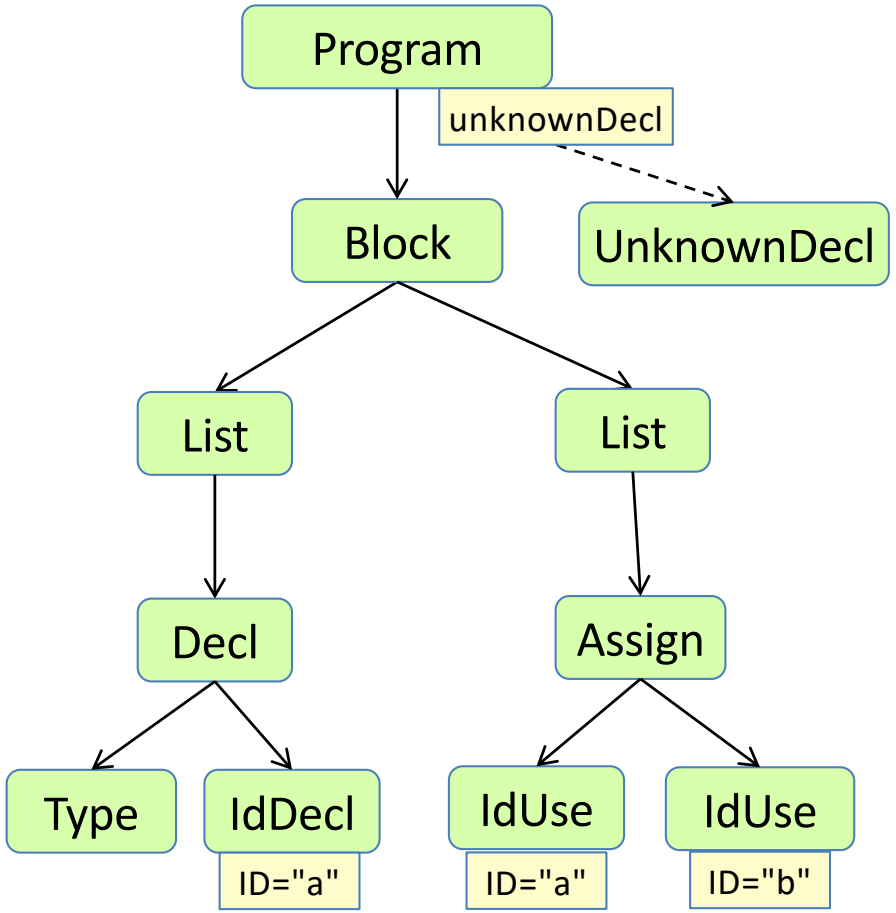
See https://en.wikipedia.org/wiki/Null_Object_pattern

But how can we implement null objects like unknownDecl() in attribute grammars?

```
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();
}
```

Use an NTA for the Null object

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



Use an NTA for the Null object

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

Extend the abstract grammar:

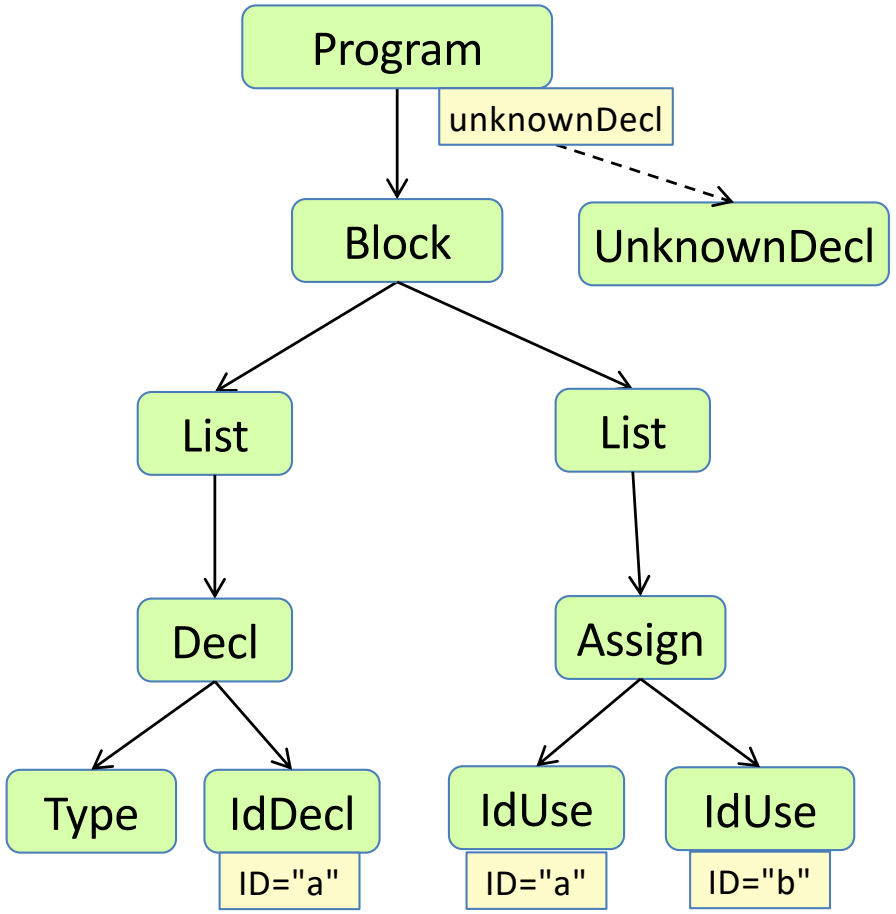
```
UnknownDecl : IdDecl;
```

Add the NTA:

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

Implement the special behavior:

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

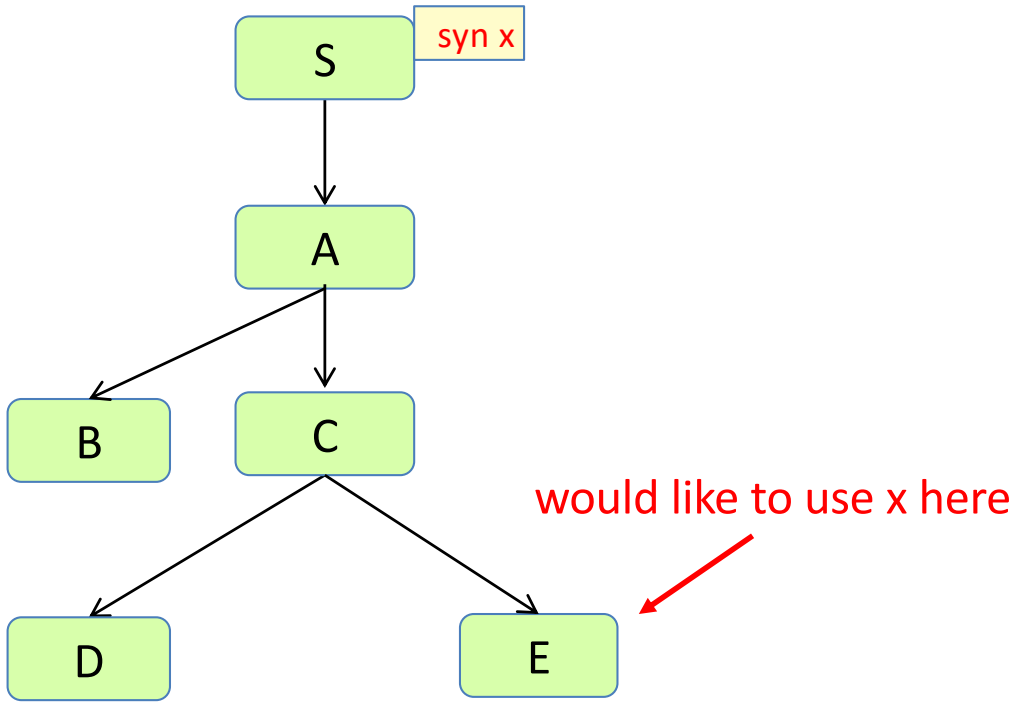


But how can we make the UnknownDecl object known throughout the AST?

The Root Attribute Pattern

The **Root Attribute** pattern

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



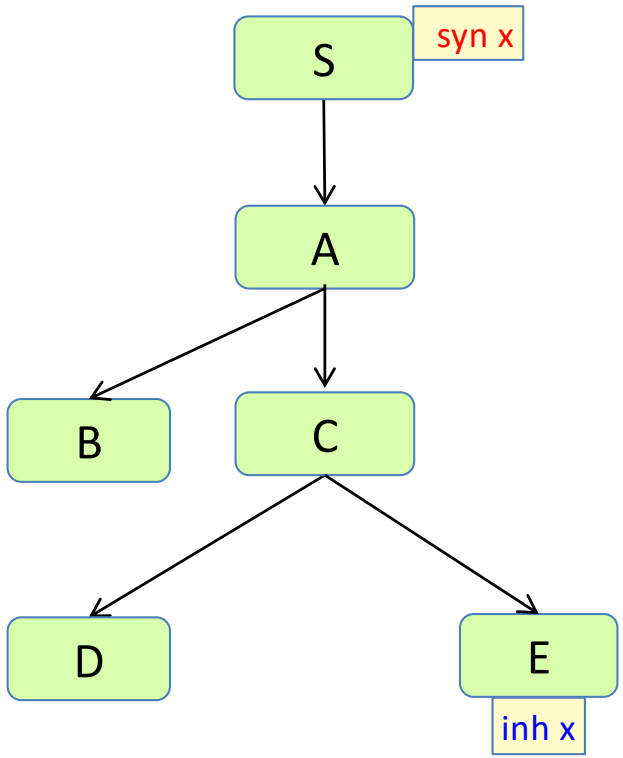
The **Root Attribute** pattern

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.

```
eq  S.getA().x() = x();

inh T E.x();
```



The **Root Attribute** pattern

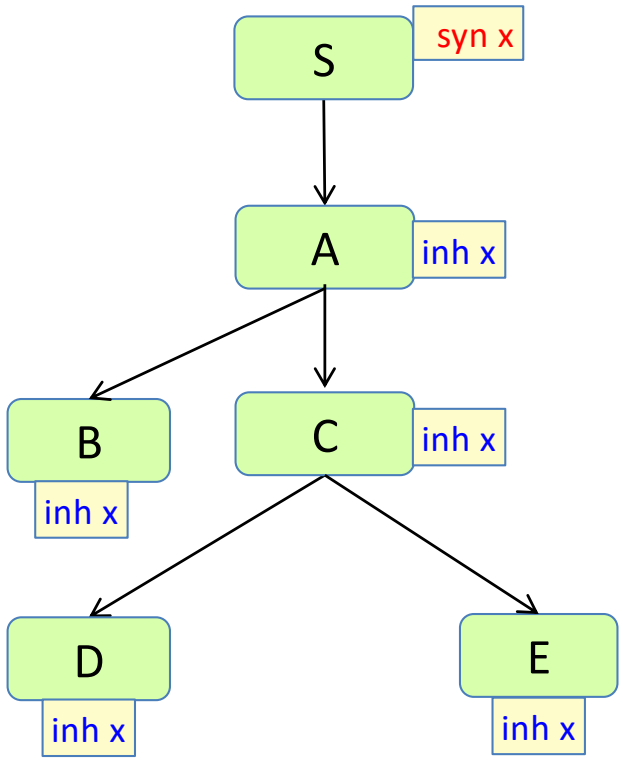
variant: expose the attribute in ASTNode

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.

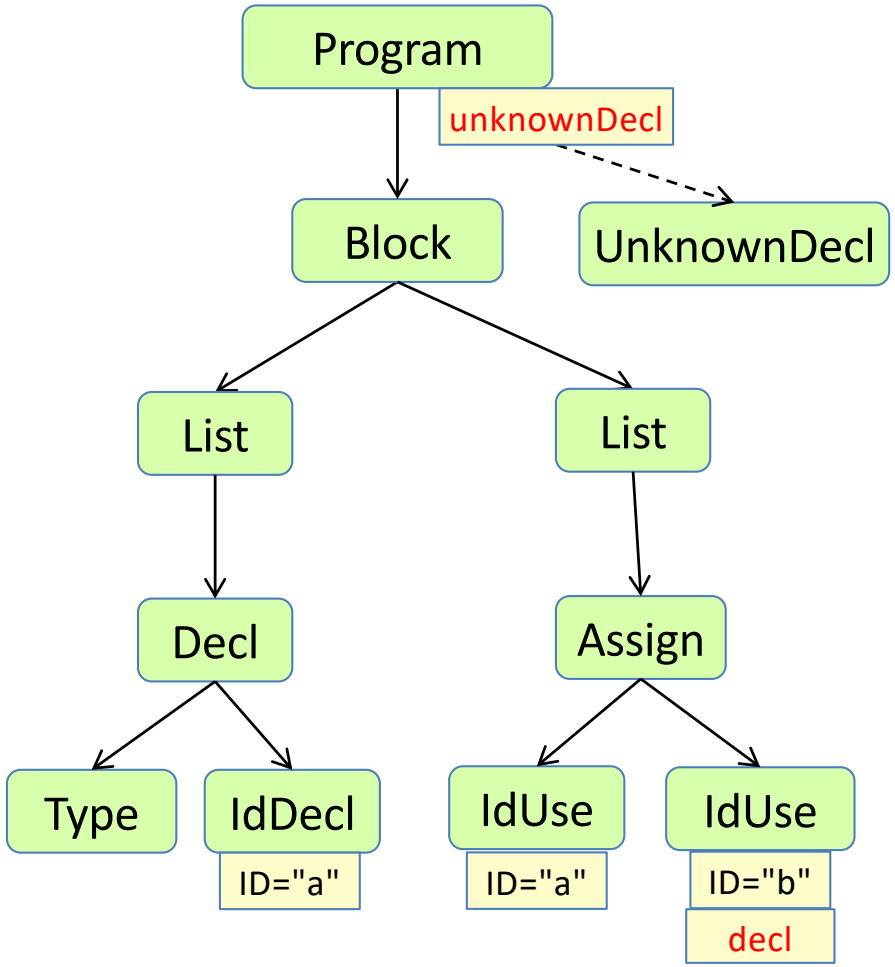
Exposing it in ASTNode will make it available in all nodes.




```
eq  S.getChild().x() = x();

inh T ASTNode.x();
```

Propagating unknownDecl

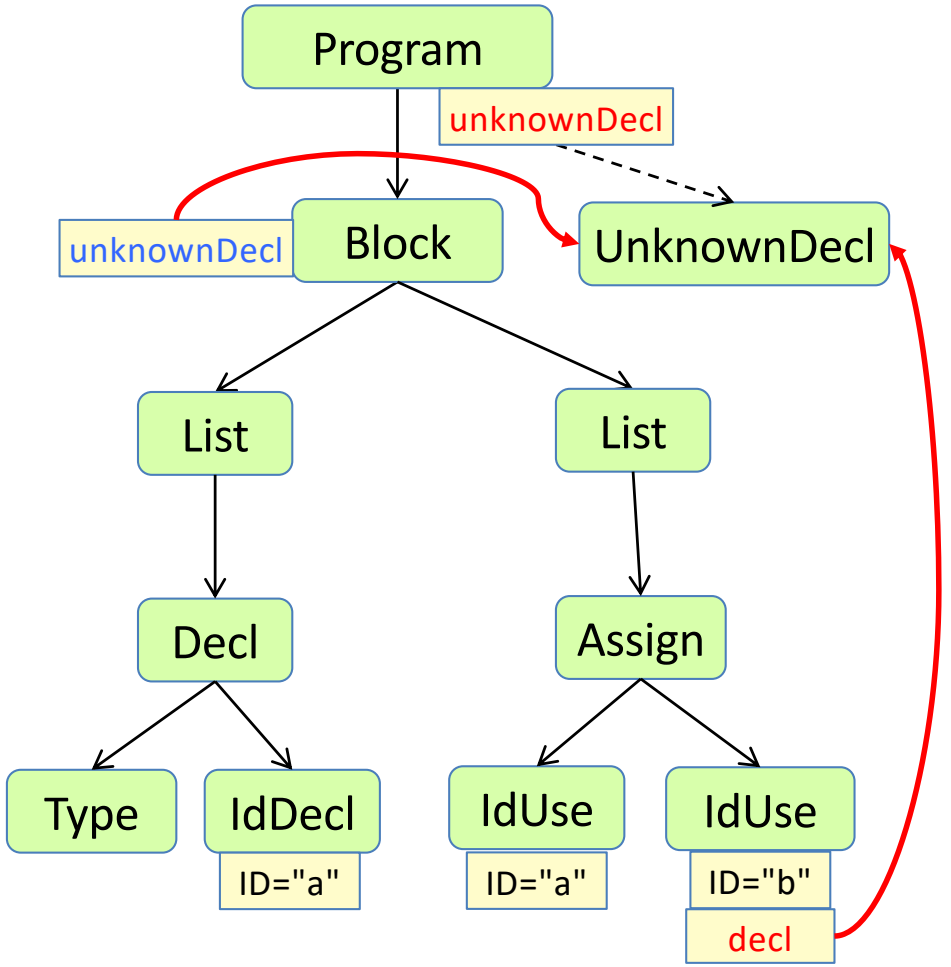
using the Root Attribute Pattern



Propagating unknownDecl

using the Root Attribute Pattern

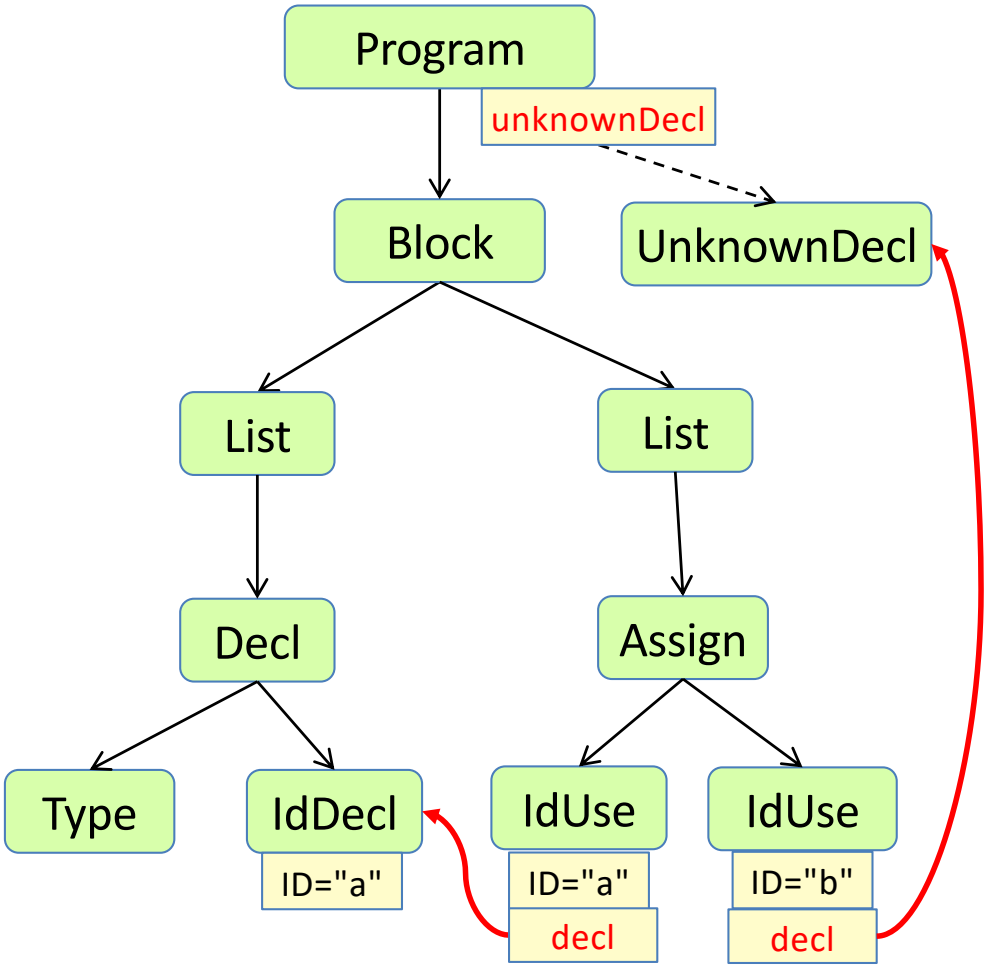
```
eq Program.getBlock().unknownDecl() =
  unknownDecl();

inh UnknownDecl Block.unknownDecl();
```



Type analysis

Type analysis

Type analysis: compute the type of each expression



Type analysis

Type analysis: compute the type of each expression

Add a type attribute to Expr

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

Implement it for IdUses

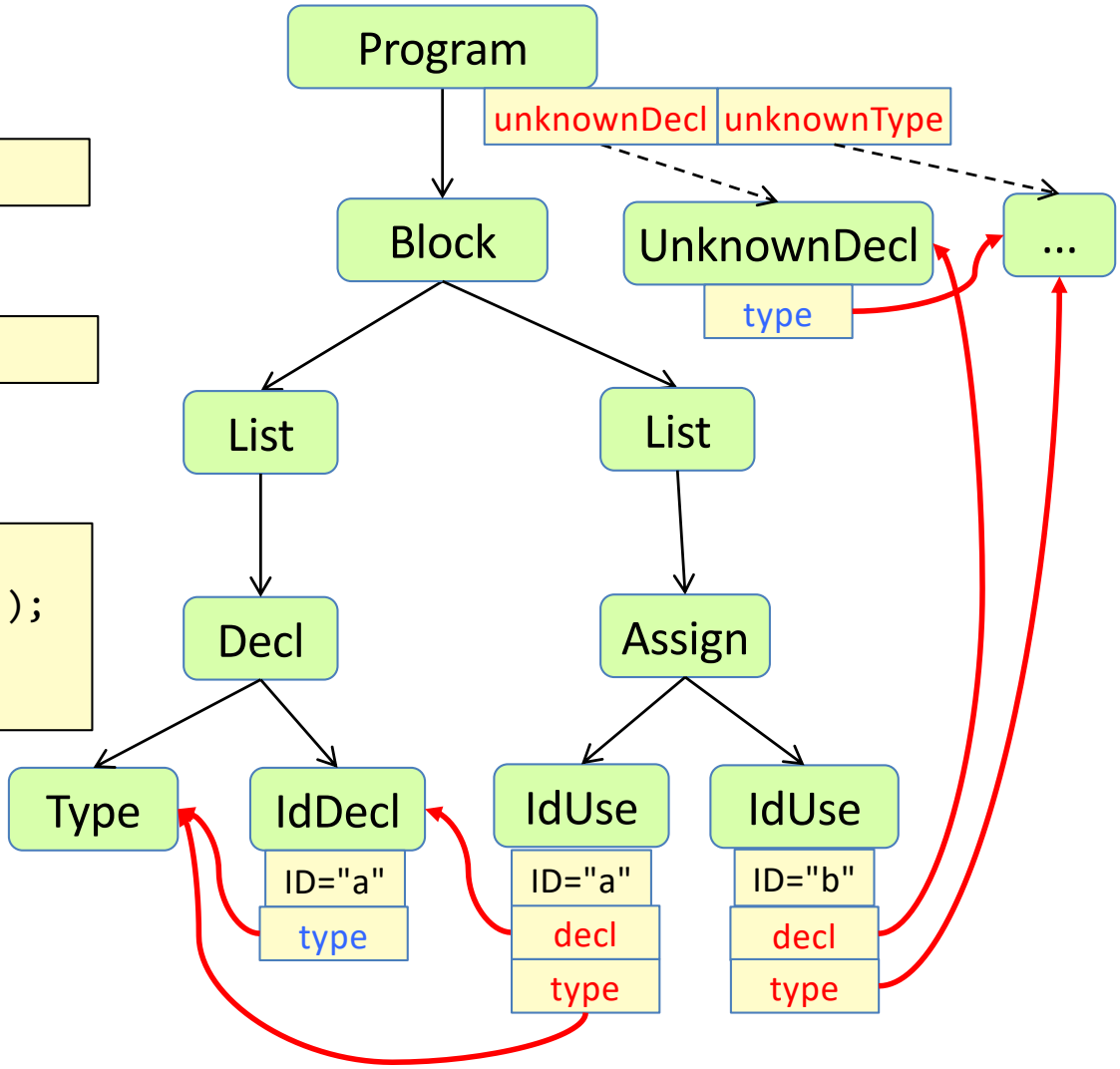
```
eq IdUse.type() = decl().type();
```

Define the type attribute for IdDecls

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

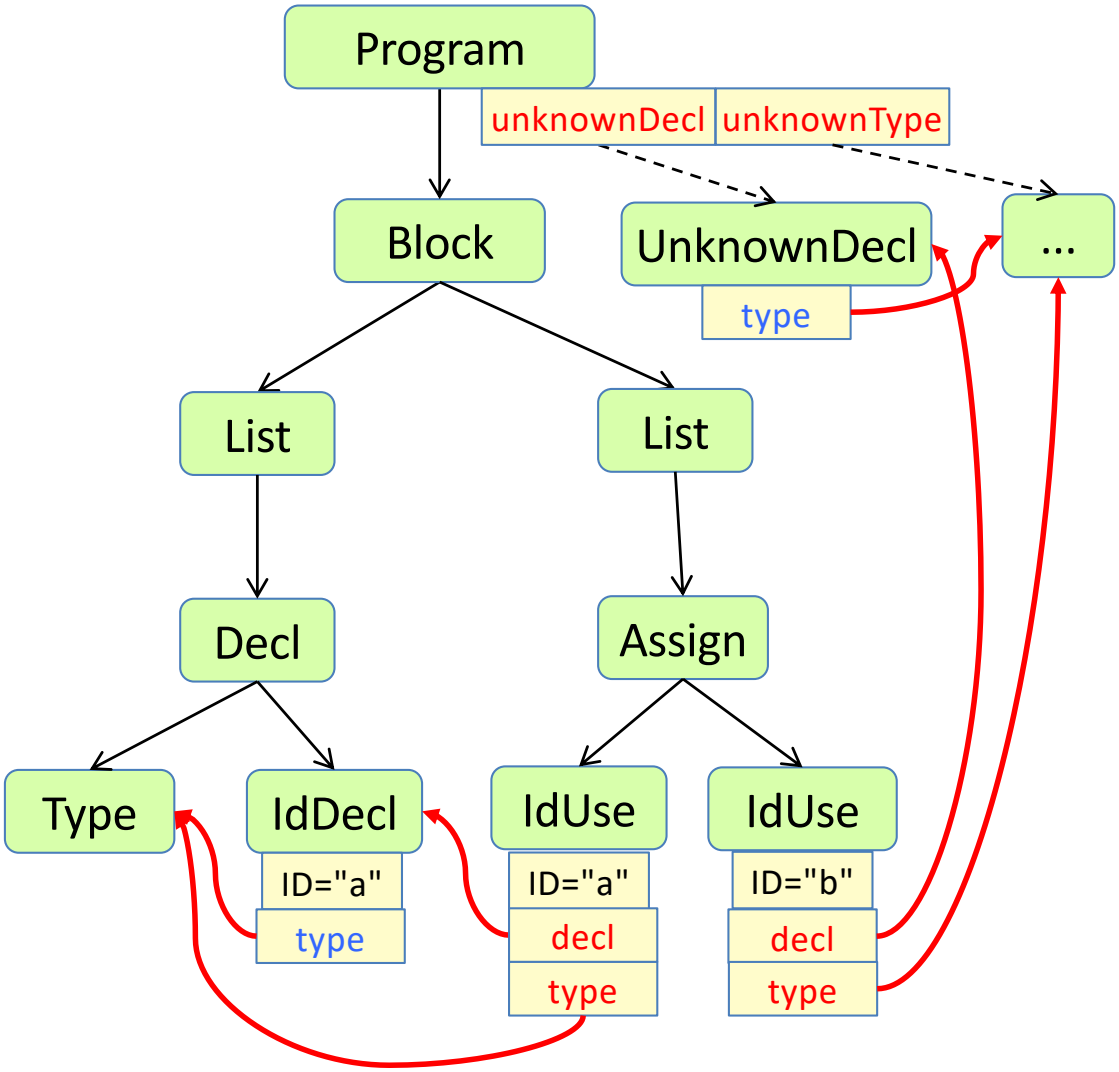
Define unknownType as an NTA

```
...
```



Type checking

Type checking: Check if types are used correctly



Type checking

Type checking: Check if types are used correctly

First attempt:

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

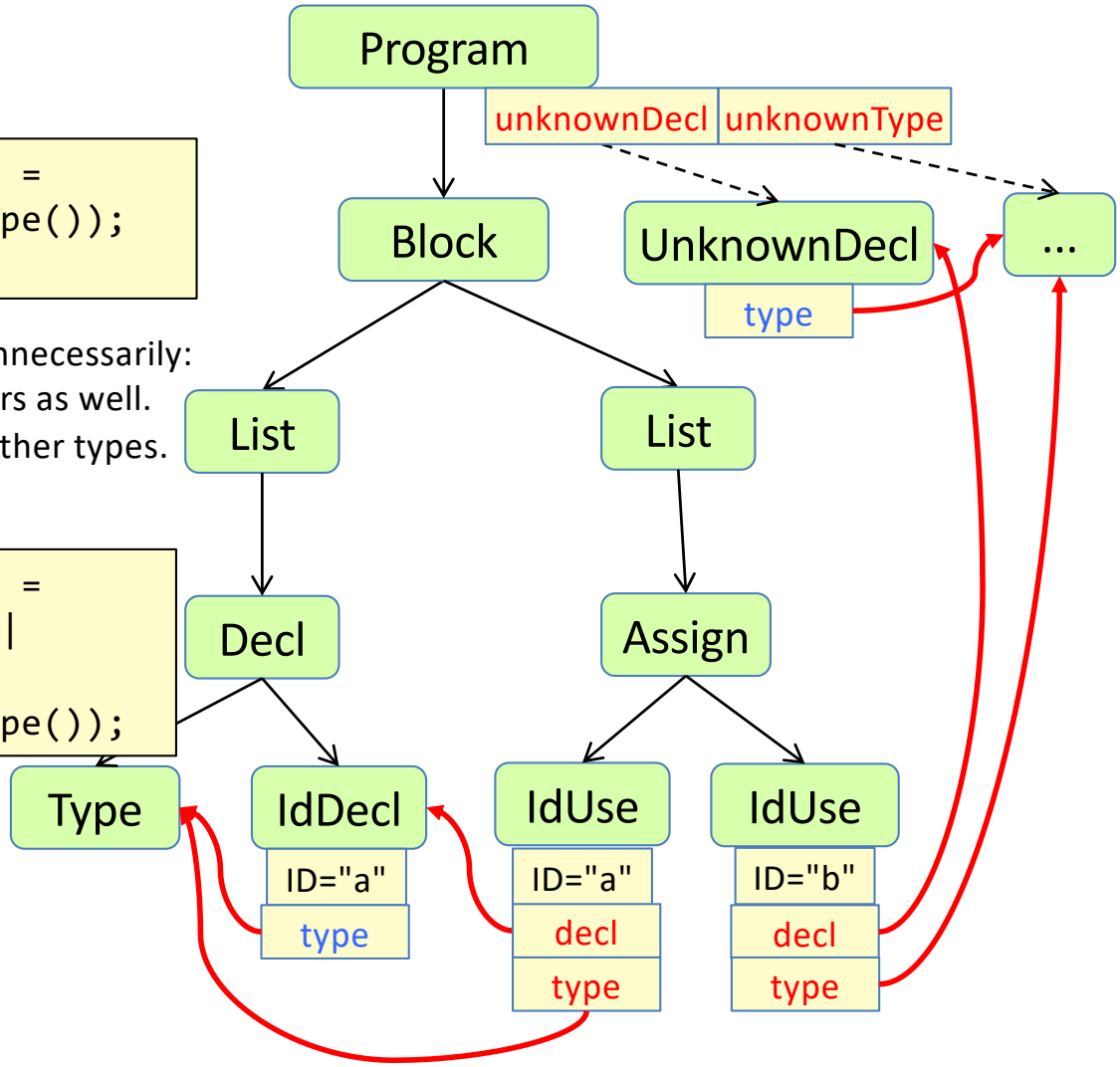
Problem with first attempt: Errors are propagated unnecessarily: Name declaration errors will give type checking errors as well. Nicer to view unknownType as compatible with all other types.

Second attempt:

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

Problem with second attempt: Have to repeat error checking code in all constructs that contain expressions.

Third attempt:
Use the Expected Type pattern.



The Expected Type pattern

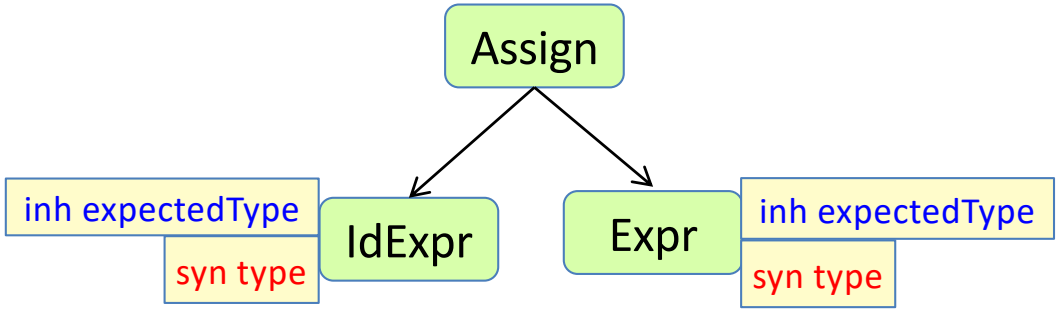
The **Expected Type** pattern

Intent:
Simplify type checking by placing the check in the expression (instead of in each construct that contains expressions)

Solution:
Add an inherited attribute *expectedType* to expressions.

Write the check once – in the expression.

A construct containing an expression needs to supply an equation for its expected type.




```
eq  Assign.getLeft().expectedType() = unknownType();
eq  Assign.getRight().expectedType() = getLeft().type();

inh Type Expr.expectedType();
syn Type Expr.type();
syn boolean Expr.compatibleTypes() = ...
```

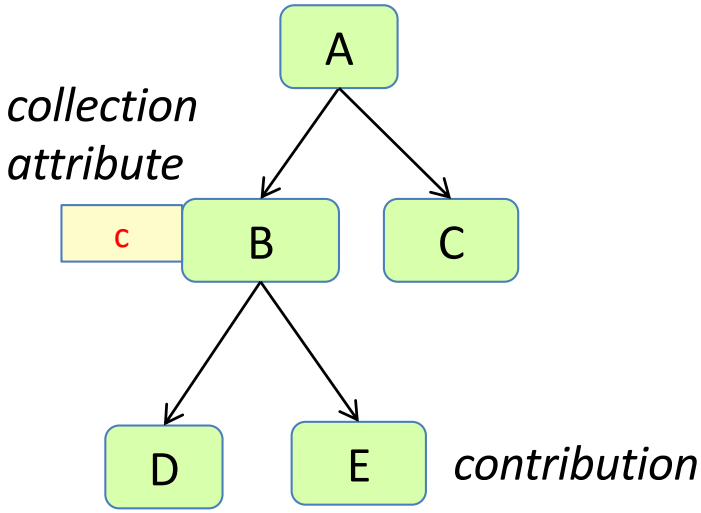
Collection attributes

Collection attributes

motivation

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

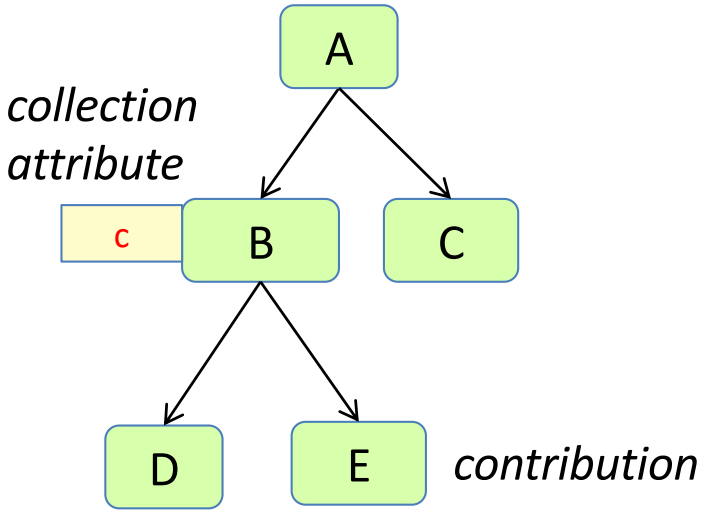
Use for values combined from many small parts spread out over the tree.



Collection attributes

motivation

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 attribute

structure

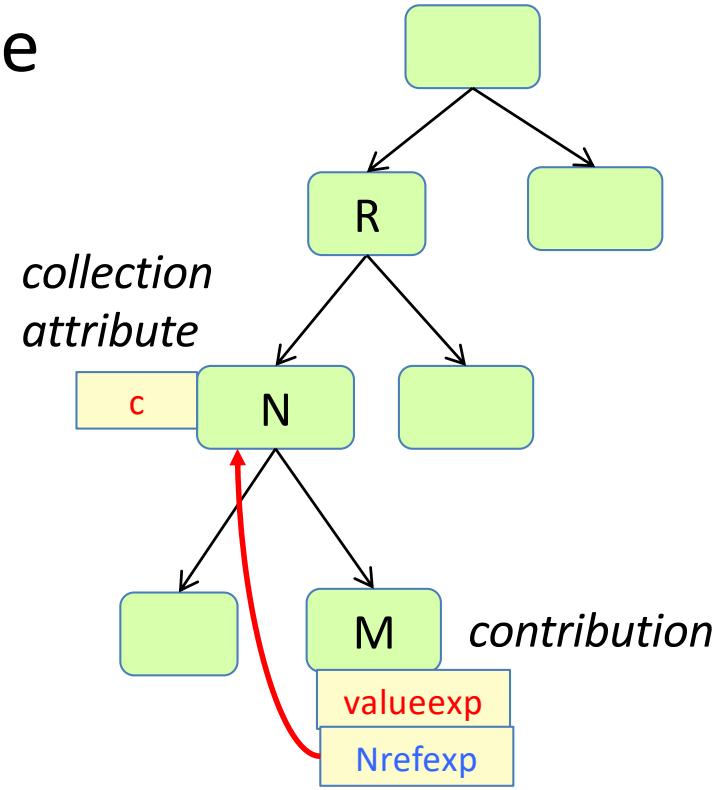
Declaration of **collection** attribute c in node type N:

```
coll T N.c() [freshexp] with m root R;
```

The method m must be **commutative**

A **contribution** from a node type M:

```
M contributes valueexp
  when condition
  to N.c()
  for Nrefexp
```



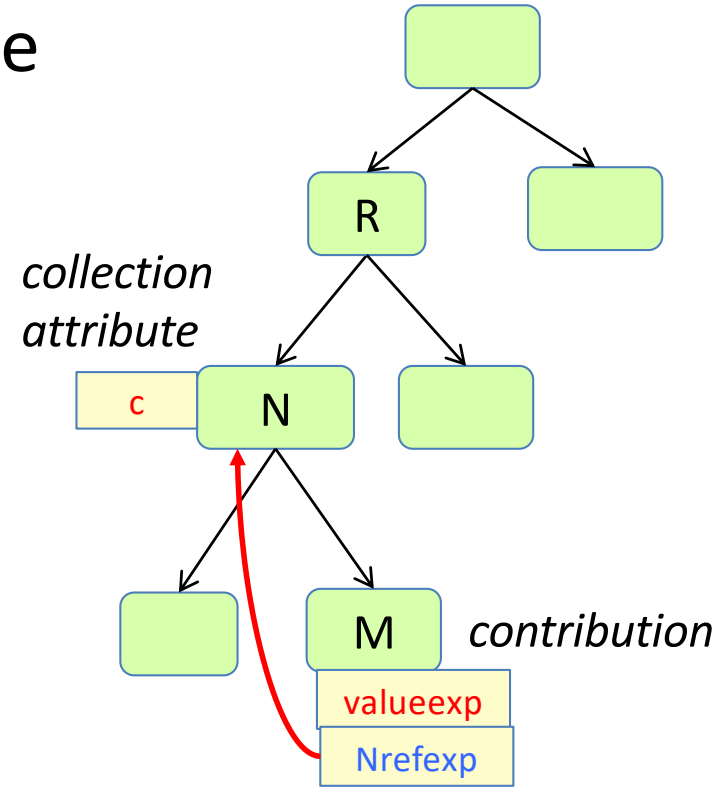
Collection attribute

structure

Declaration of **collection** attribute c in node type N:

```
coll T N.c() [freshexp] with m root R;
```

- T is the type of c
- *freshexp* is a fresh T object (empty collection)
- m is a **commutative** mutating method used for adding contributions to c
- R is an AST node type, identifying the subtree where contributions can be



A **contribution** from a node type M:

```
M contributes valueexp
  when condition
  to N.c()
  for Nrefexp
```

- *valueexp* is the value to be contributed
- *condition* is a condition indicating if *valueexp* should be added or not
- *Nrefexp* is a reference to an N node

Evaluation algorithm

When c is accessed for the first time:

- the empty collection is created using *freshexp*
- the subtree at the upward nearest R is traversed, and all contributions are added to c
- c is cached

Collection attribute

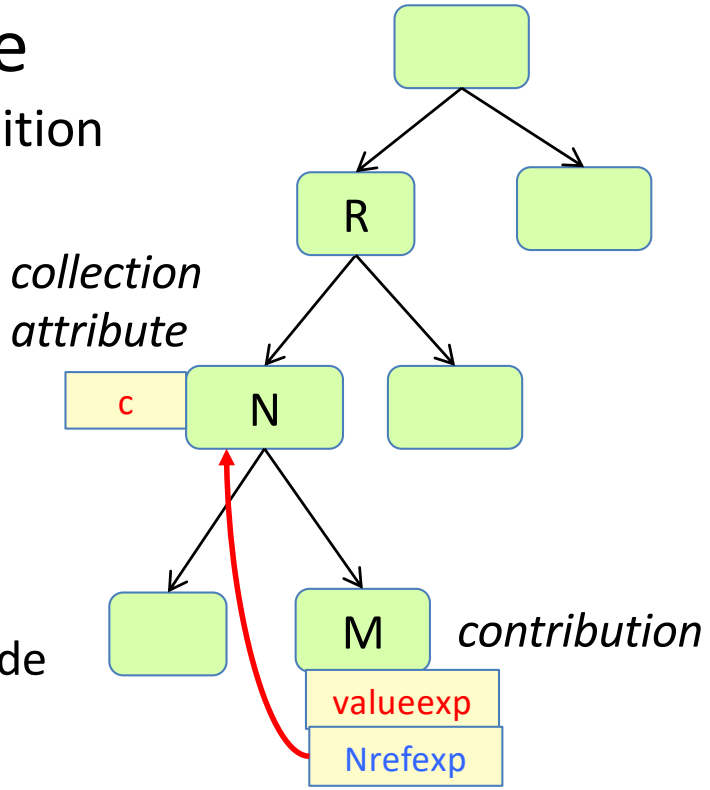
optional elements in the definition

Declaration of **collection** attribute c in node type N:

```
coll T N.c() [freshexp] with m root R;
```

The method m must be **commutative**

- if "[freshexp]" is left out, the default constructor for T will be used.
- if "with m" is left out, the method name "add" is used
- if "root R" is left out, R is set to the type of the root node



A **contribution** from a node type M:

```
M contributes valueexp
  when condition
  to N.c()
  for Nrefexp
```

- if "when condition" is left out, the value will always be added
- "for Nrefexp" can be left out if N=R

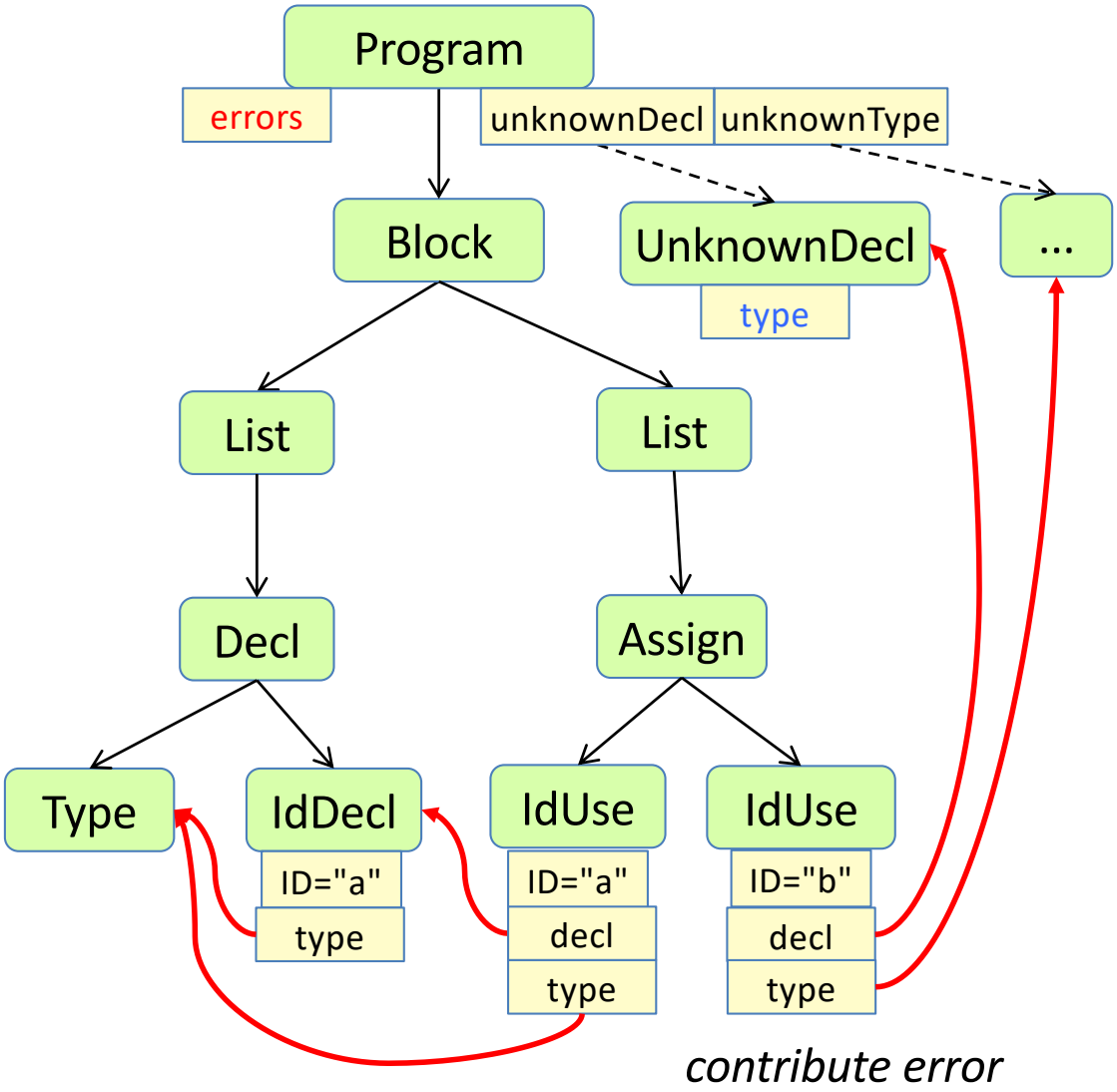
Collect errors

Example: Collect errors

Error checking: collect all errors

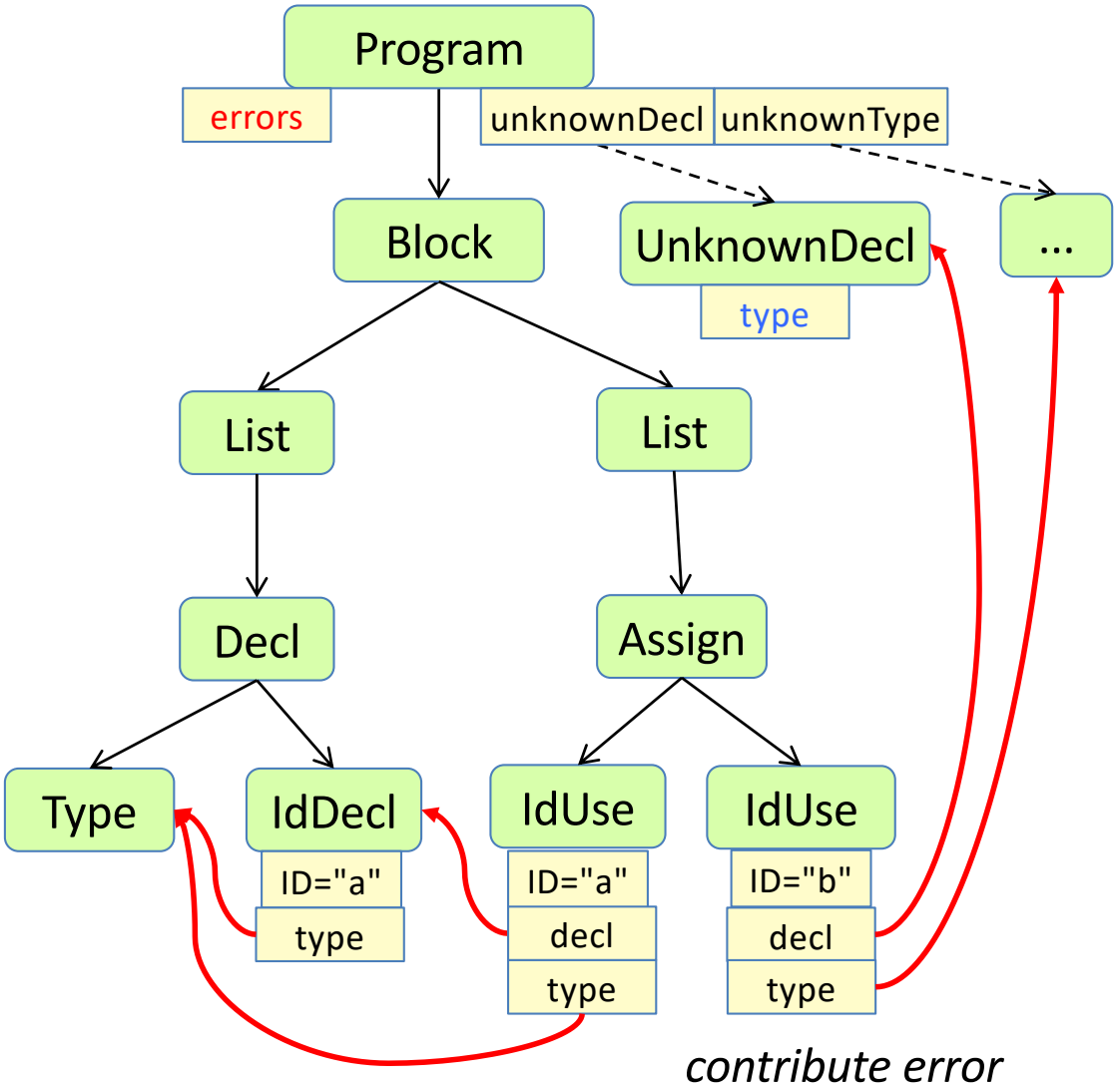
We would like an attribute errors in the root, containing all error messages.

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



Example: Collect errors

Error checking: collect all errors

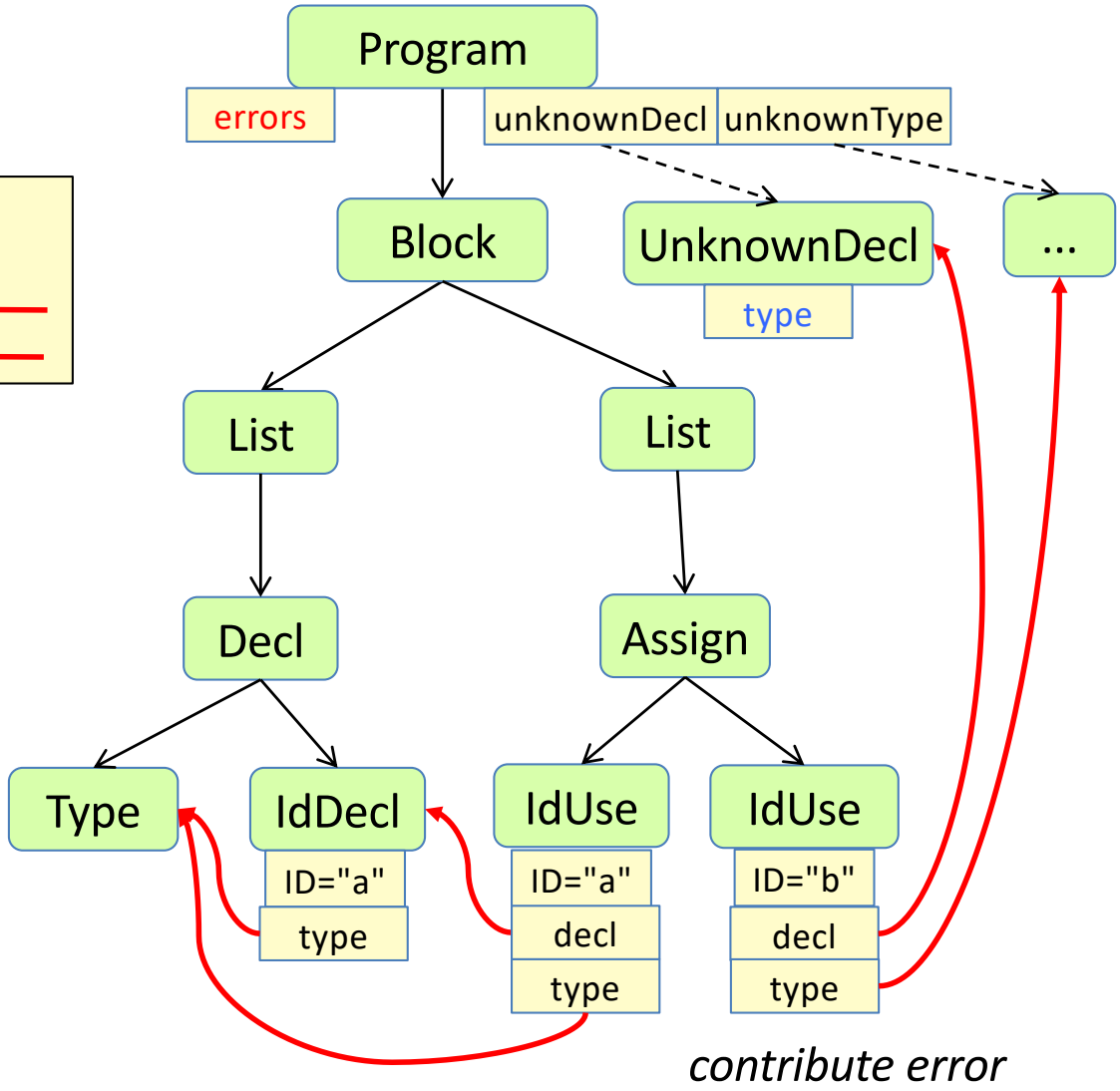


Example: Collect errors

Error checking: collect all errors

Declare the errors collection:

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



because of defaults, these optional parts can be skipped in this case

Example: Collect errors

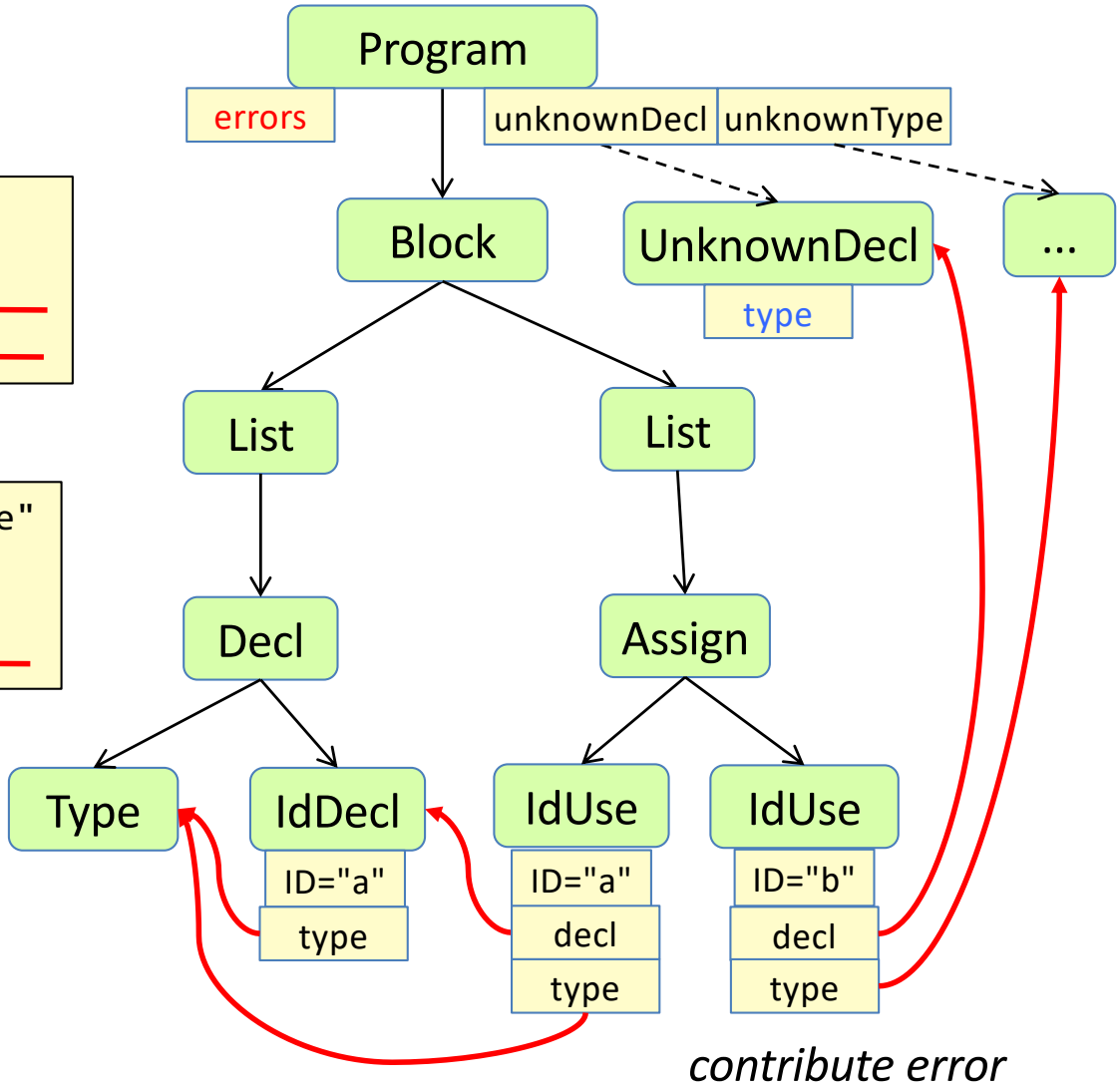
Error checking: collect all errors

Declare the errors collection:

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

Contribute an error

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



because of defaults, these optional parts can be skipped in this case

The compiler main program becomes very simple when using AGs:

```
// Sketch:
public class Compiler {
   public static void main(String[] args) {
      // Construct the AST:
      Program program = new Parser().parse(new Scanner());
      // Use attributes:
      if (program.errors().isEmpty()) {
         printCode(program.code());
      }
      else {
         printErrors(program.errors());
      }
   }
}
```

All attributes (e.g., **code** and **errors**) are automatically available as soon as the AST (**program**) has been constructed by the parser.
Calling an attribute causes it to be evaluated (on-demand evaluation).

Summary questions:

reference attributes, name analysis

- What is broadcasting?
- 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, collection attributes, error 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?
- How dows the Expected Type pattern work?
- What is a collection attribute?
- How can a collection of error message be implemented?