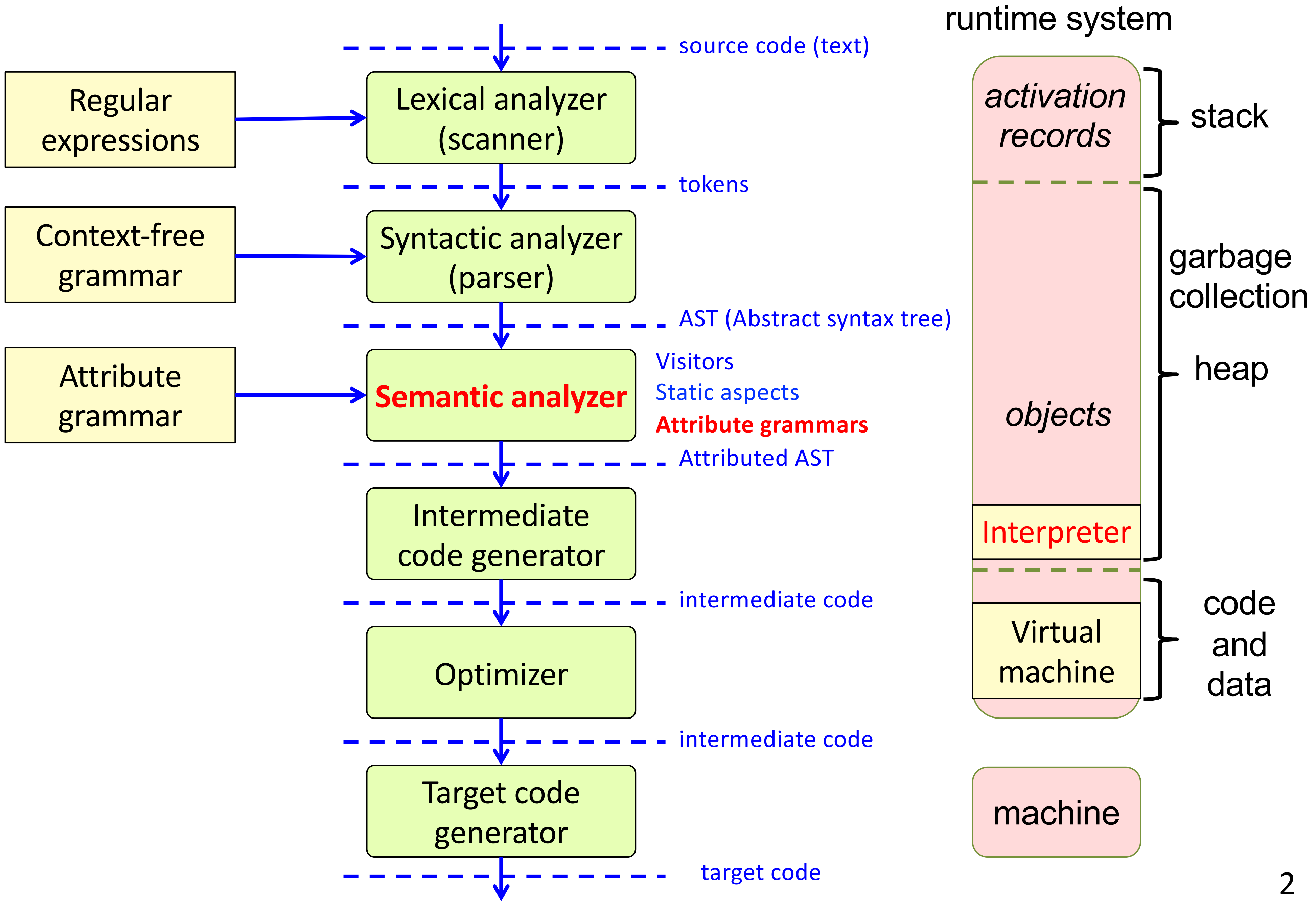
EDAN65: Compilers, Lecture 09

More on Attribute Grammars + interpretation

Görel Hedin

Revised: 2025-09-29

This lecture



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)

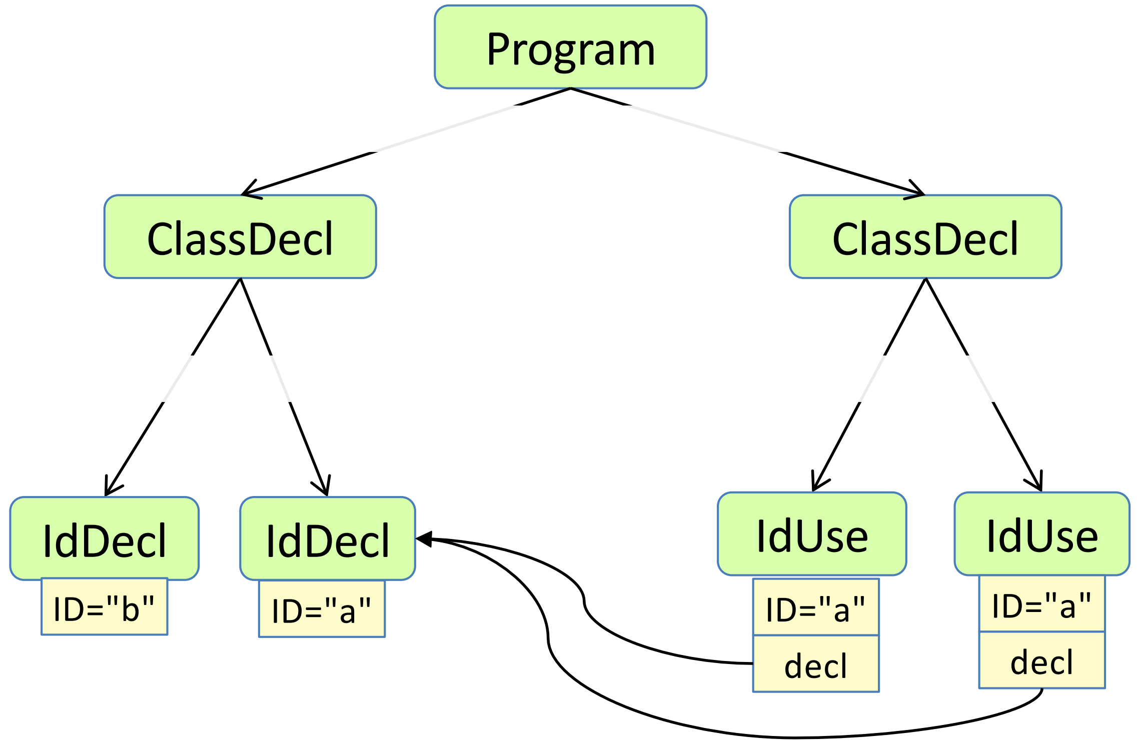
*** Treated in this lecture**

More examples of collection attributes

Example: uses of declaration

reversing references

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

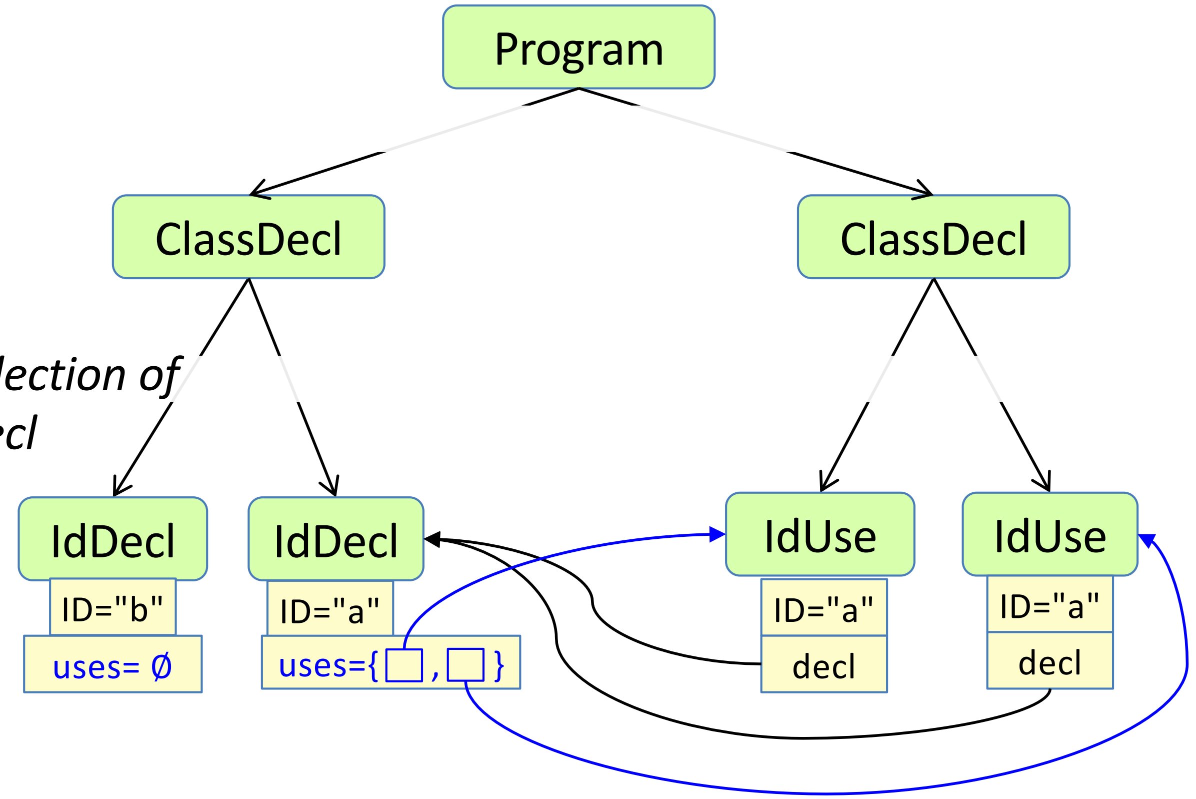


Example: uses of declaration

reversing references

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

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



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

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

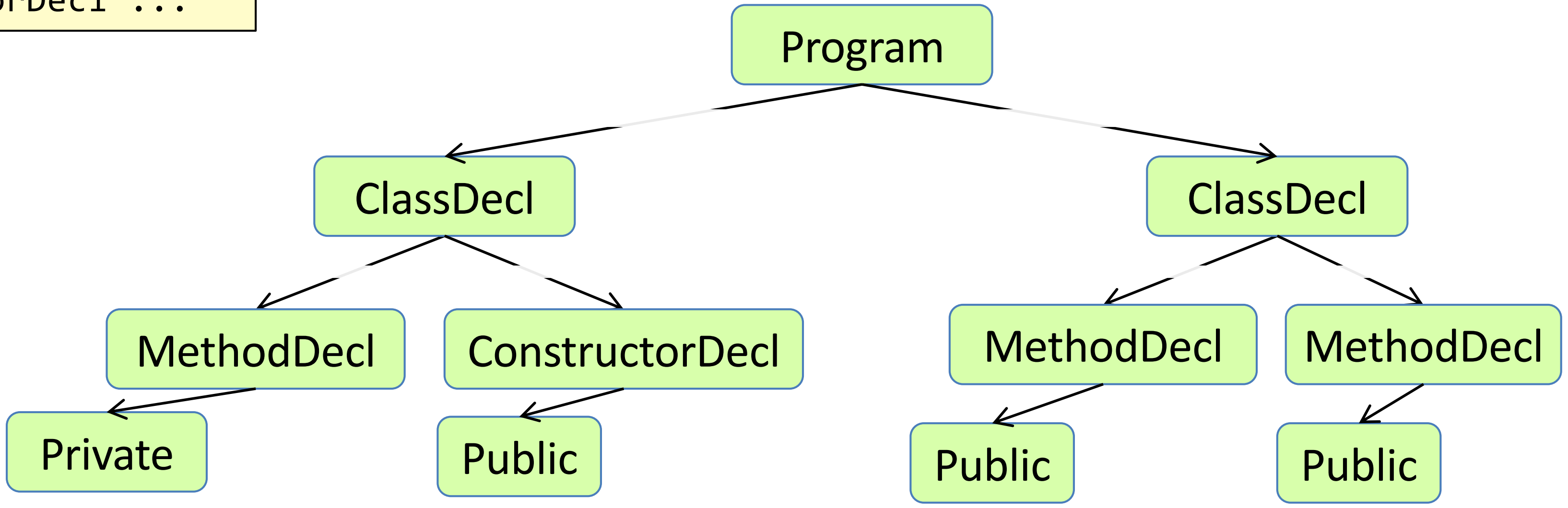
can skip because of defaults

JastAdd Java grammar

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

Example: NPM metric

conditional count



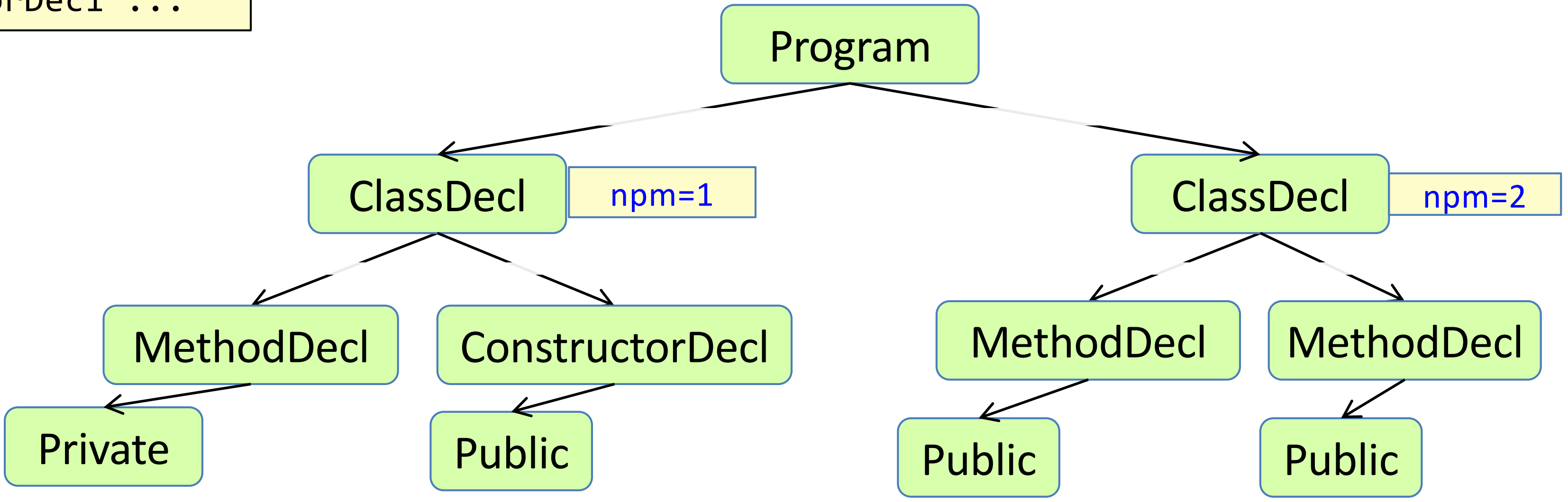
NPM – Number of Public Methods and constructors in a class

JastAdd Java grammar

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

Example: NPM metric

conditional count



NPM – Number of Public Methods and constructors in a class

```
class Counter {
  int count = 0;
  void add(int i) { count += i; }
}
```

can skip because of defaults

coll Counter ClassDecl.npm () ~~[new Counter()]~~ ~~with add~~ root ClassDecl;

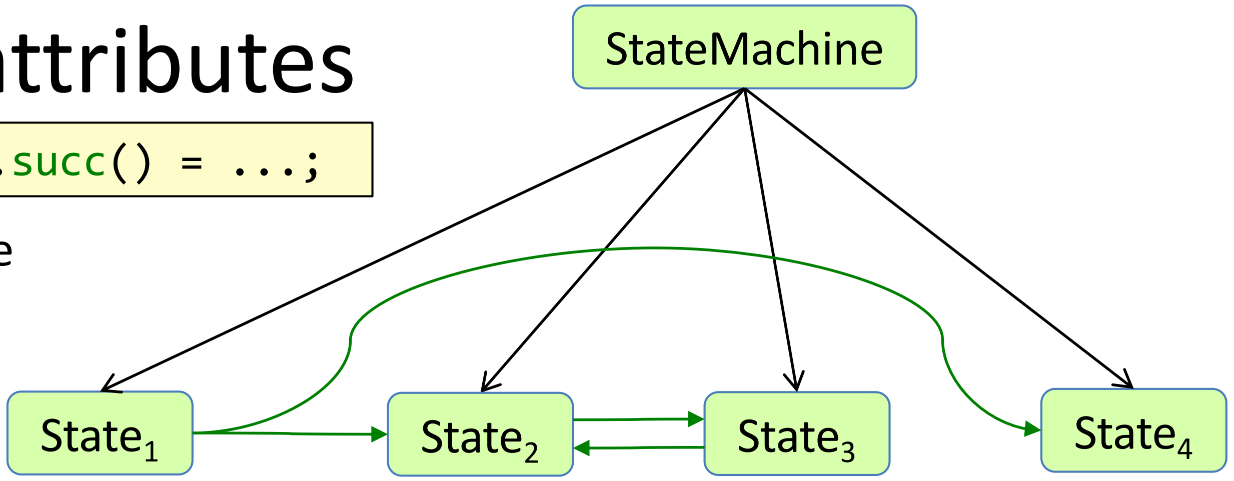
MethodDecl contributes 1 when isPublic() to ClassDecl.npm();
ConstructorDecl contributes 1 when isPublic() to ClassDecl.npm();

Circular attributes

Circular attributes

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

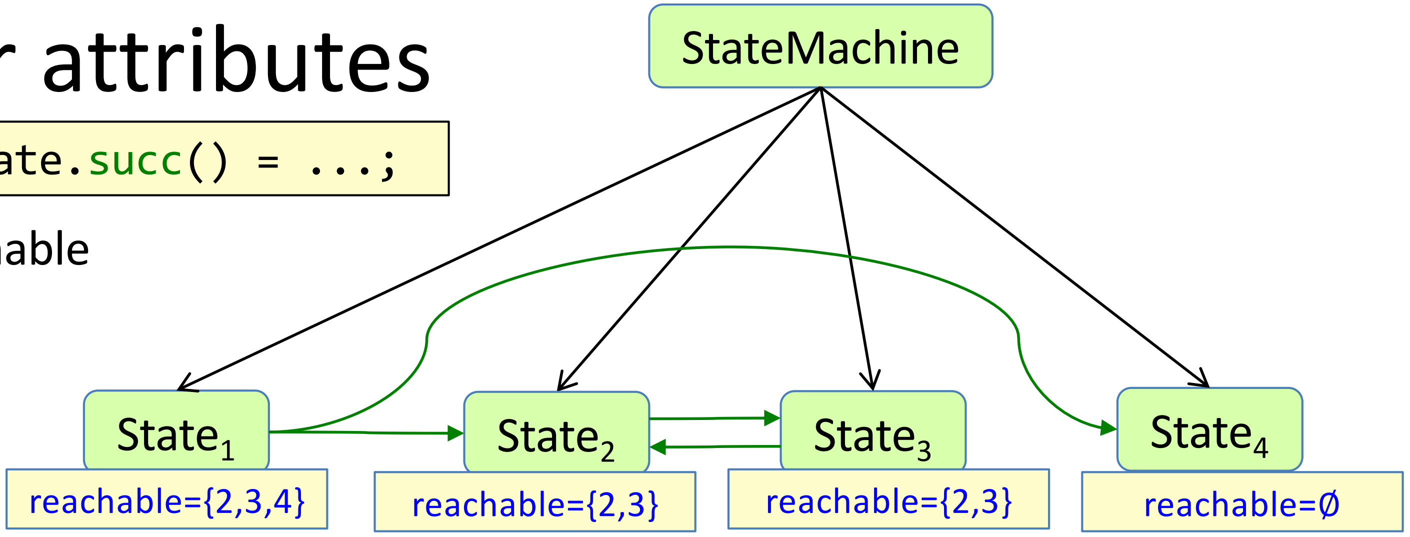
What states are reachable from state k?



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>();
  result.addAll(succ());
  for (State s : succ())
    result.addAll(s.reachable());
  return result;
}
```

A circular attribute may depend (transitively) on itself.

Circular attributes - termination

Does this computation terminate?

Implementation using a circular attribute

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

A circular attribute may depend (transitively) on itself.

Circular attributes - termination

Does this 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

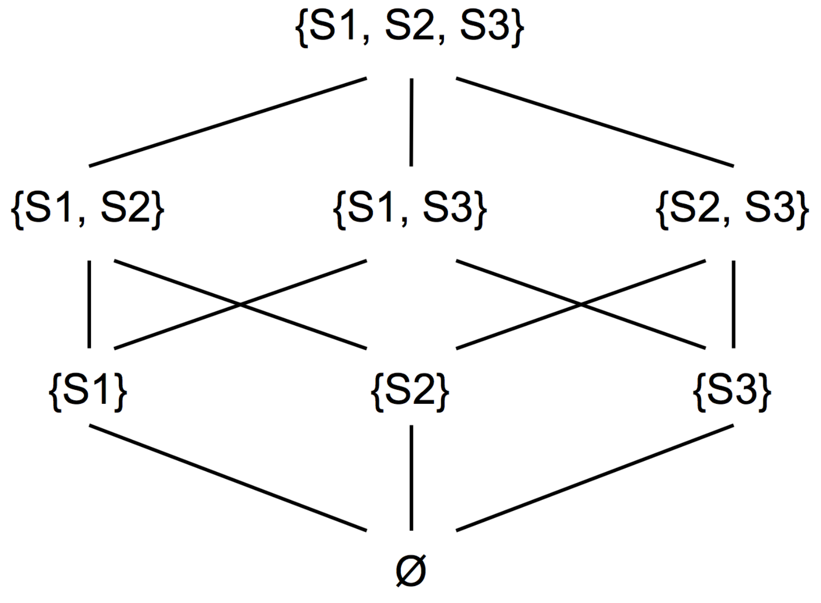
```
syn Set<State> State.reachable() circular [new HashSet<State>()] {
  HashSet<State> result = new HashSet<State>();
  result.addAll(succ());
  for (State s : succ())
    result.addAll(s.reachable());
  return result;
}
```

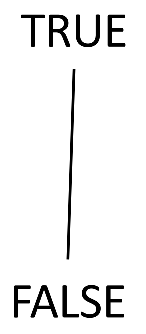
A circular attribute may depend (transitively) on itself.

Useful lattice types

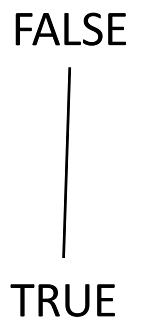
Set lattice
Start with the empty set.
Use the UNION operator.
Make sure there is a finite set
of possible values in a set.

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>();
  result.addAll(succ());
  for (State s : succ())
    result.addAll(s.reachable());
  return result;
}
```

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

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>();
  result.addAll(succ());
  for (State s : succ())
    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

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)
- ...

Program analysis

compute program properties

to find compile-time errors
to generate code
to optimize code
to find probable bugs or vulnerabilities
to support interactive tooling
to measure quality
...

Program analysis

compute program properties

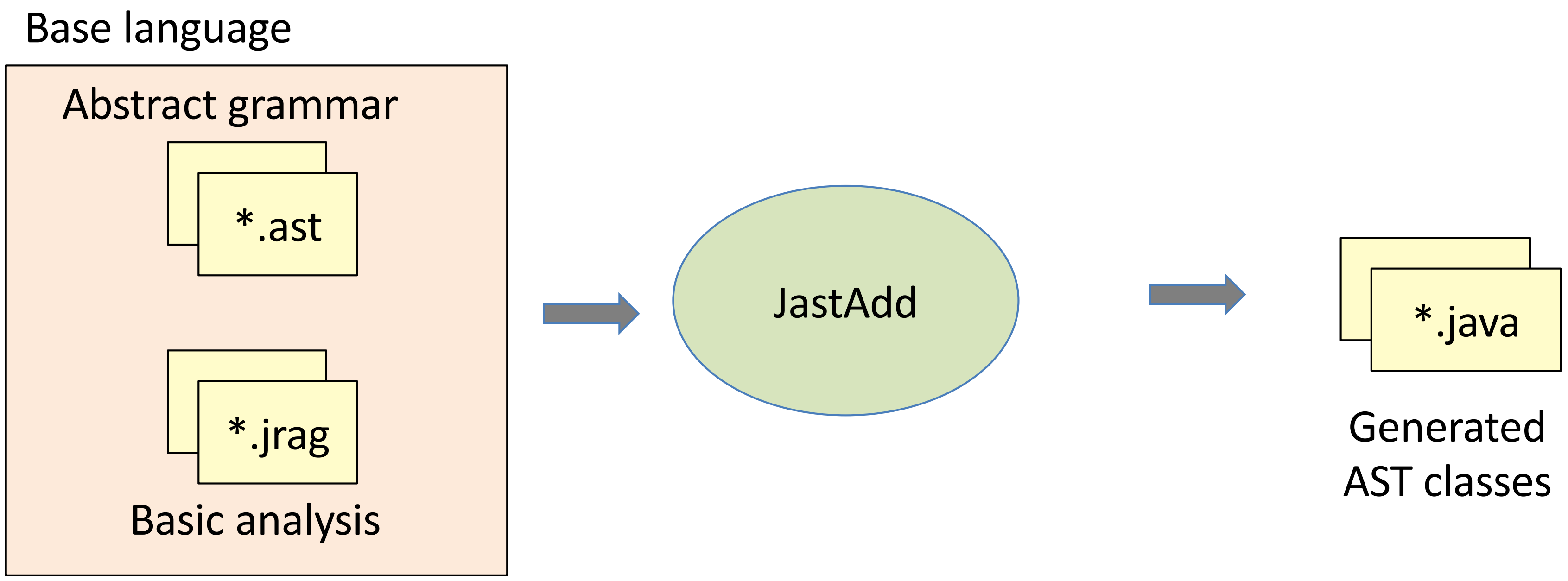
to find compile-time errors
to generate code
to optimize code
to find probable bugs or vulnerabilities
to support interactive tooling
to measure quality
…

Static	**Dynamic**
on the source code or on compiled code	on a running program
the analysis holds for all possible program runs	analysis of one particular run
many extend basic analyses like name and type analysis	

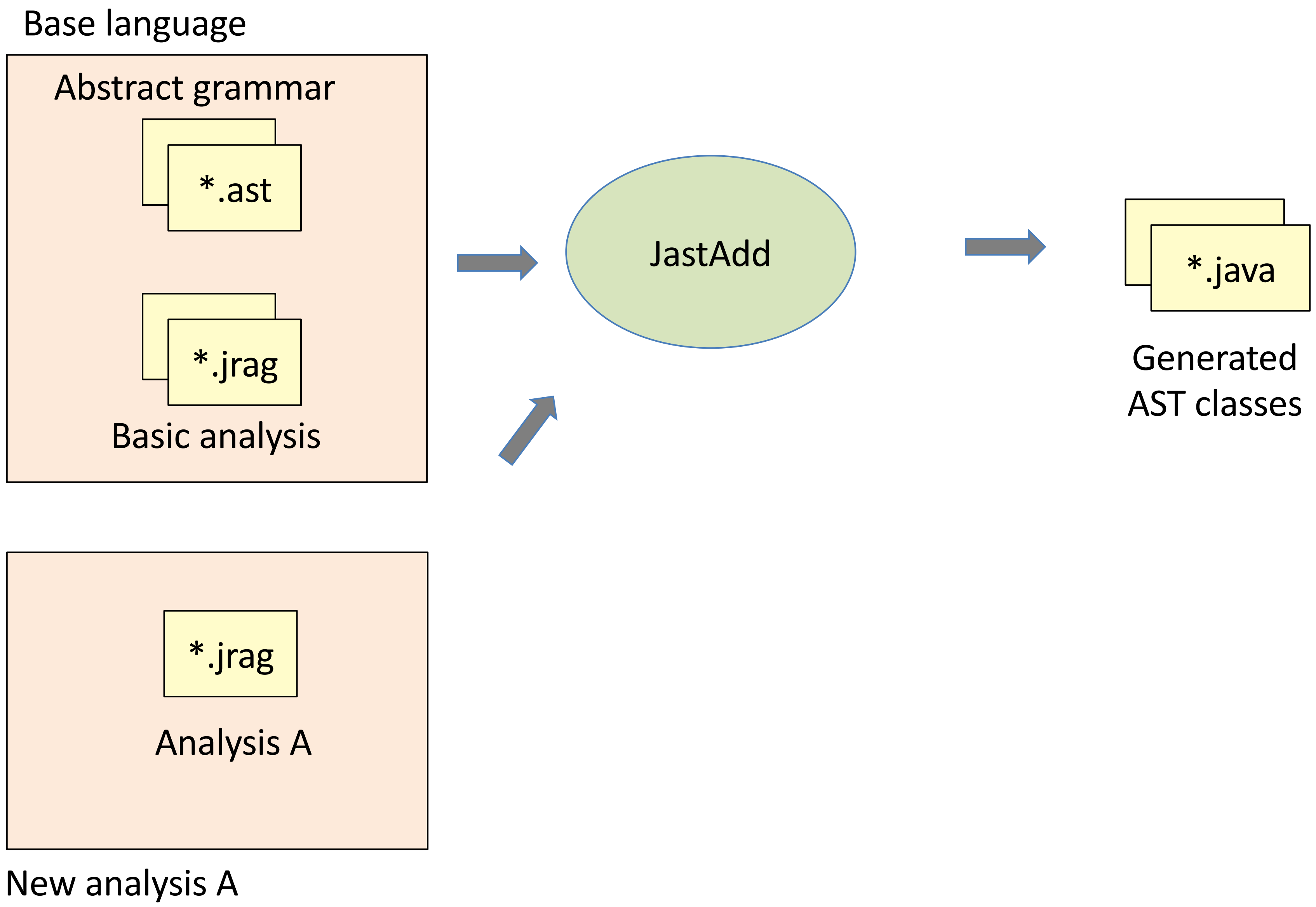
Example static analyses

name-analysis.jrag	
type-analysis.jrag	
control-flow.jrag	What are the possible successors of a given statement? Are there statements that are unreachable in a method?
data-flow.jrag	What statements affect the value of a variable at a given point? Are there statements that are unnecessary in the method?
call-graph.jrag	What methods are called by a given method? Are there methods that are never called?
metrics.jrag	Compute some useful metrics of a method, class or program.
....jrag	

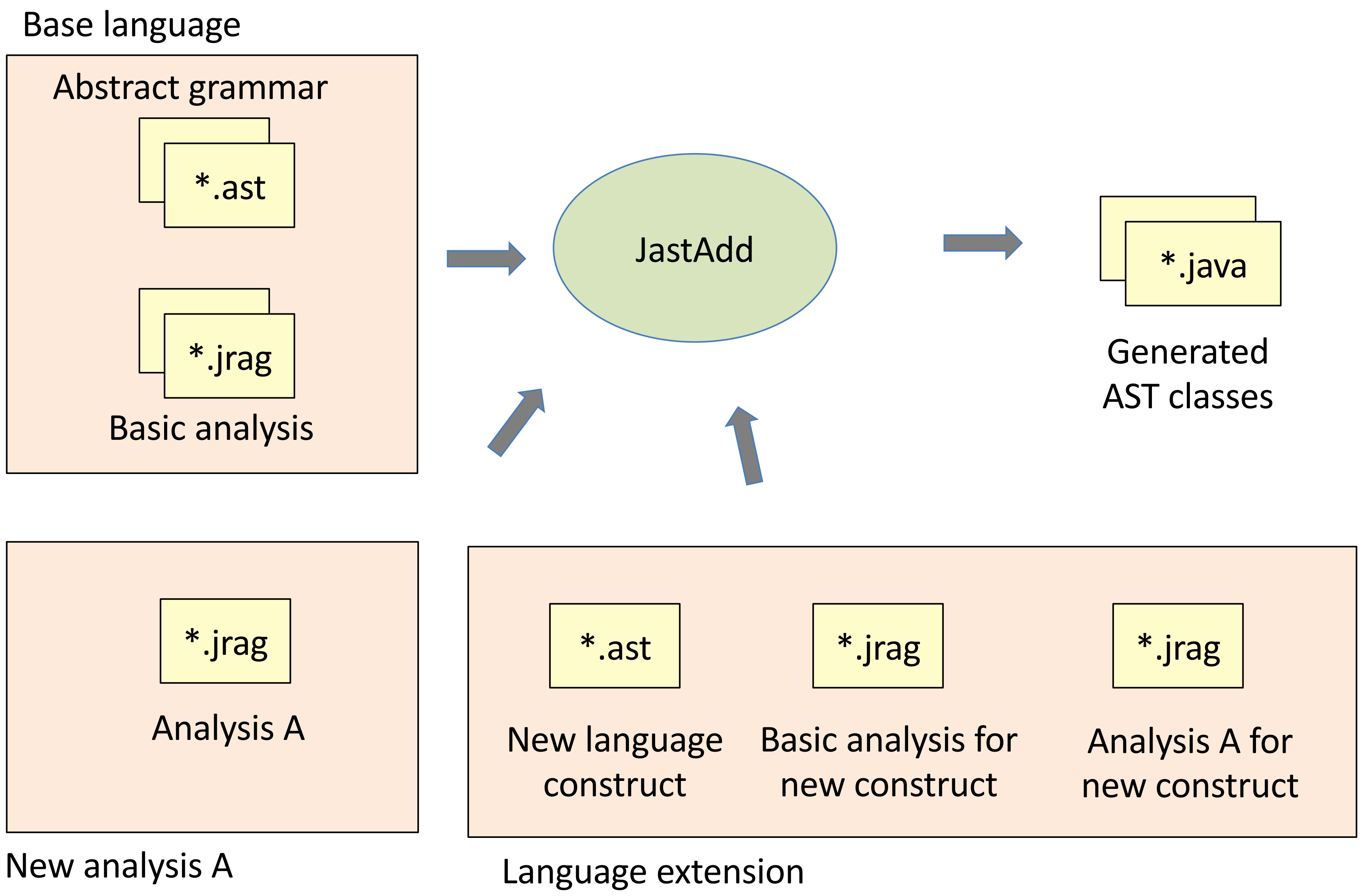
Modular extension in JastAdd



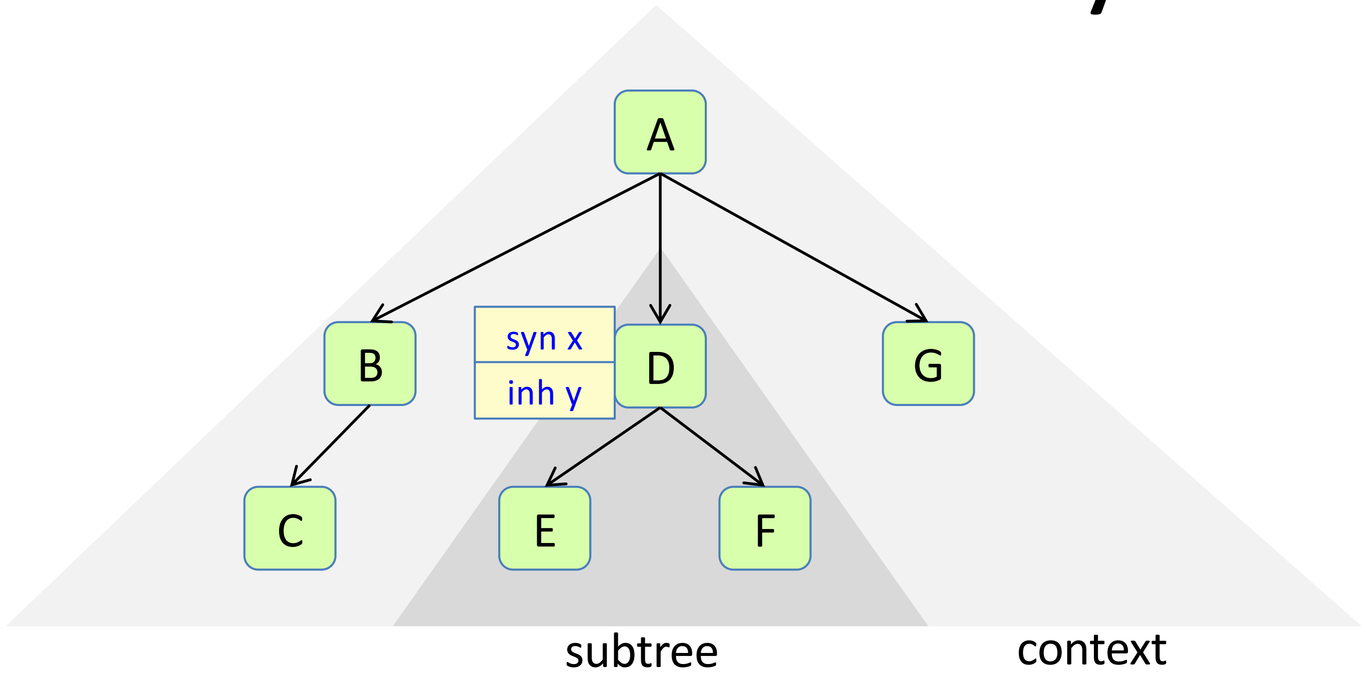
Modular extension in JastAdd



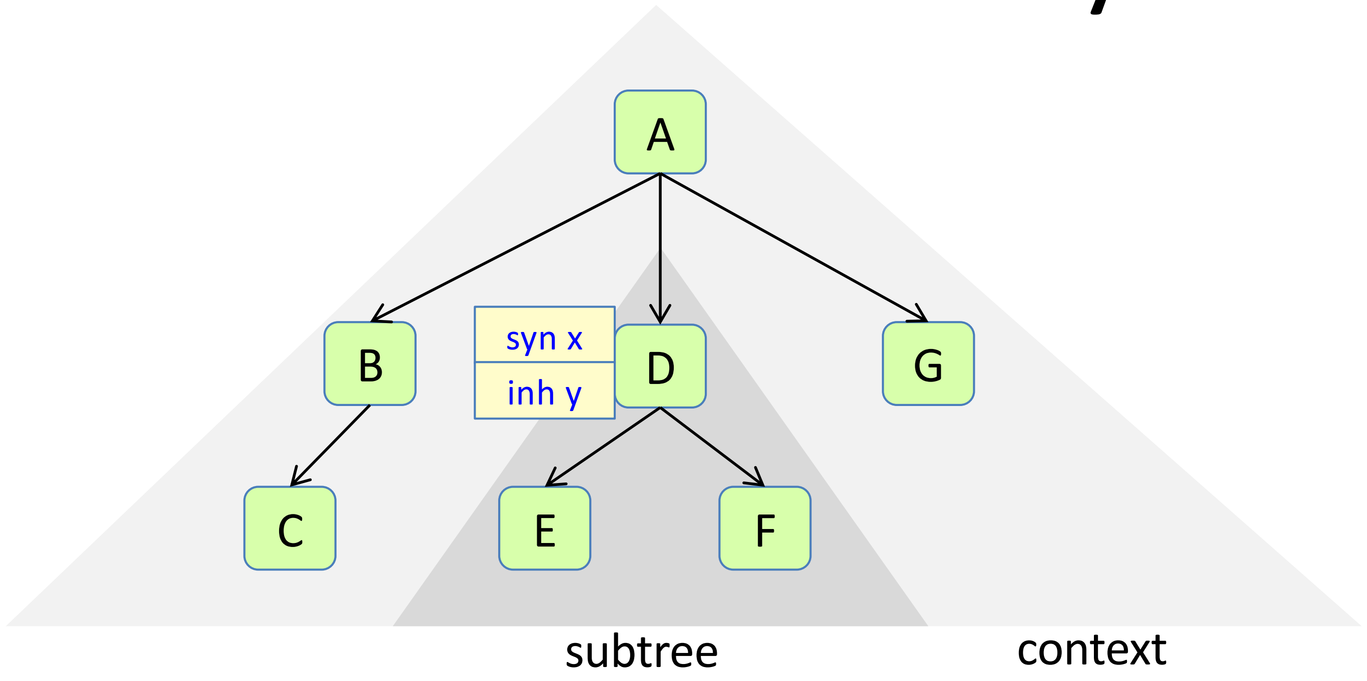
Modular extension in JastAdd



Think declaratively!



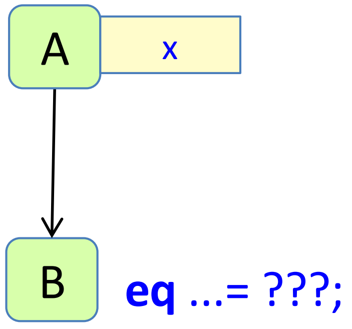
Think declaratively!



- What is the property you would like to compute? Declare as an attribute.
- What other properties would allow you to easily define its value? Declare as more attributes.
- Make an attribute *synthesized* if it depends on information in the subtree of the node.
- Make an attribute *inherited* if it *only* depends on the context (nodes outside the subtree).
- Don't think about the order of computation.

getParent antipattern

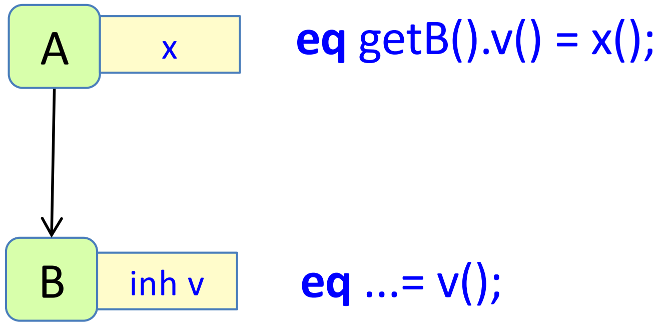
`A ::= B;`



Question:
Suppose B needs the value of A.x.
Is it a good idea to use getParent()?
Why not?

Answer:
B does not know the type of its parent – a cast would be needed, and a typecase if there is another rule C ::= B.

If another rule D ::= B is added, we would need to change the code for B – the extension will not be modular.



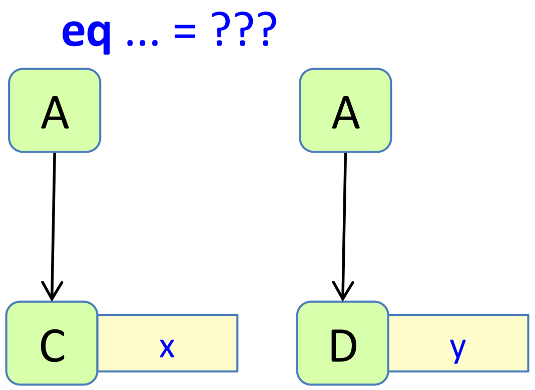
Refactored solution:
Let B define an inherited attribute v.
A (and C) will now need to supply an equation for v (modular additions).

If a D ::= B rule is added, D will also need to supply an equation for v. B will not have to be changed.

No typecases or casts needed.

instanceOf antipattern

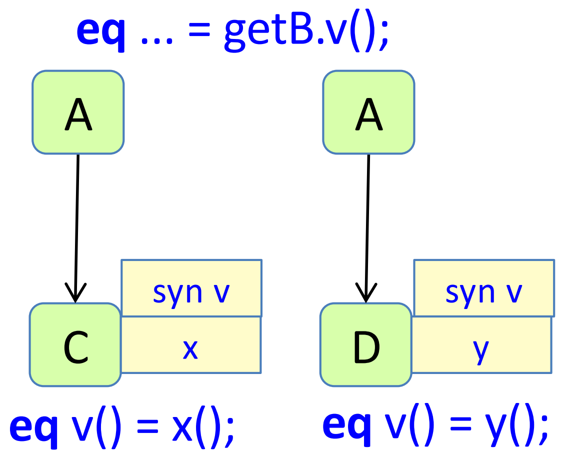
```
A ::= B;
abstract B;
C : B;
D : B;
```



Question:
Suppose A needs the value of C.x or D.y.
Is it a good idea to use instanceOf (typecase)? Why not?

Answer:
The type of A's child is B. It does not know what all the subtypes of B can be.

If A uses a typecase, this code would need to change if another subtype E : B is added – the extension will not be modular.



Refactored solution:
Introduce a synthesized attribute v in B.
A can now use the v attribute.
C and D can each supply an equation for v.

If an E : B rule is added, it is sufficient to add an equation for E.v. The code in A will not have to be changed.

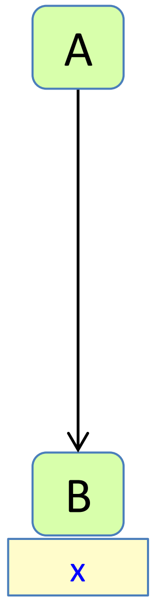
No typecases or casts needed.

Review of attribute mechanisms

Intrinsic
Synthesized
Inherited
Broadcasting
Reference
Parameterized
NTA
Collection
Circular

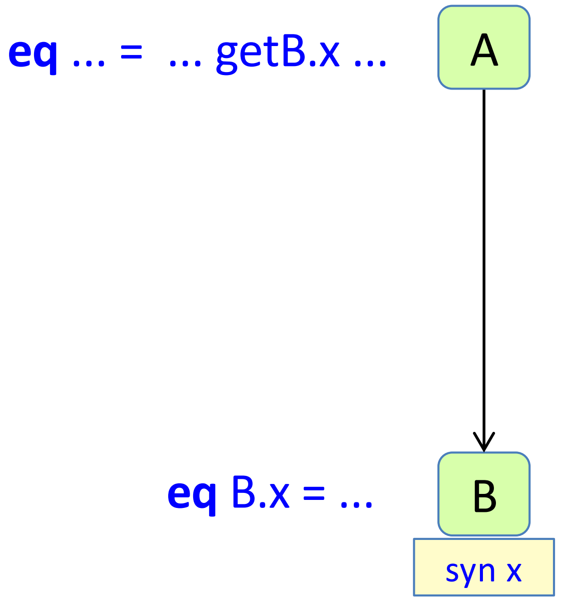
Intrinsic attribute

A ::= B
B ::= <x:int>



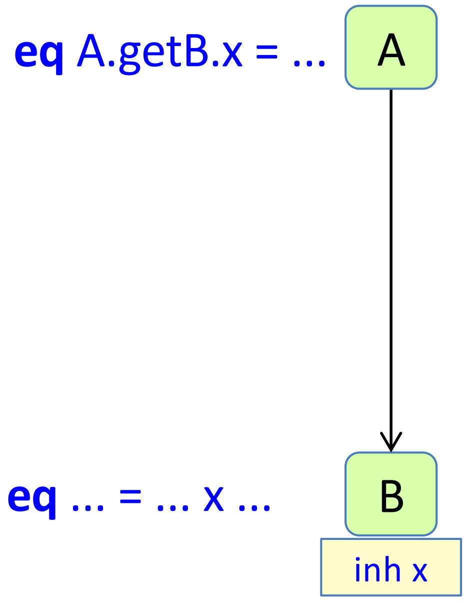
Defined in abstract grammar.
Given value when AST is constructed.

Synthesized attribute



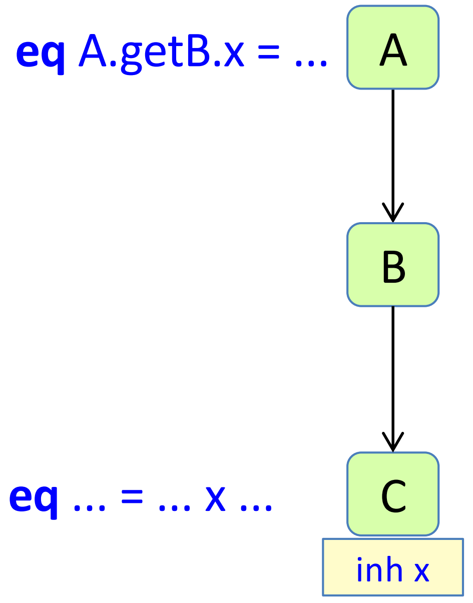
Define in the node itself. Use in parent.

Inherited attribute attribute



Use in the node itself. Define in a parent.

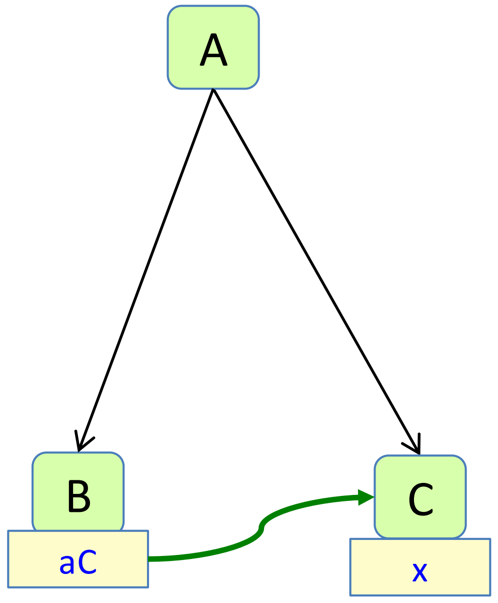
Broadcasting



The definition does not have to be in the immediate parent.

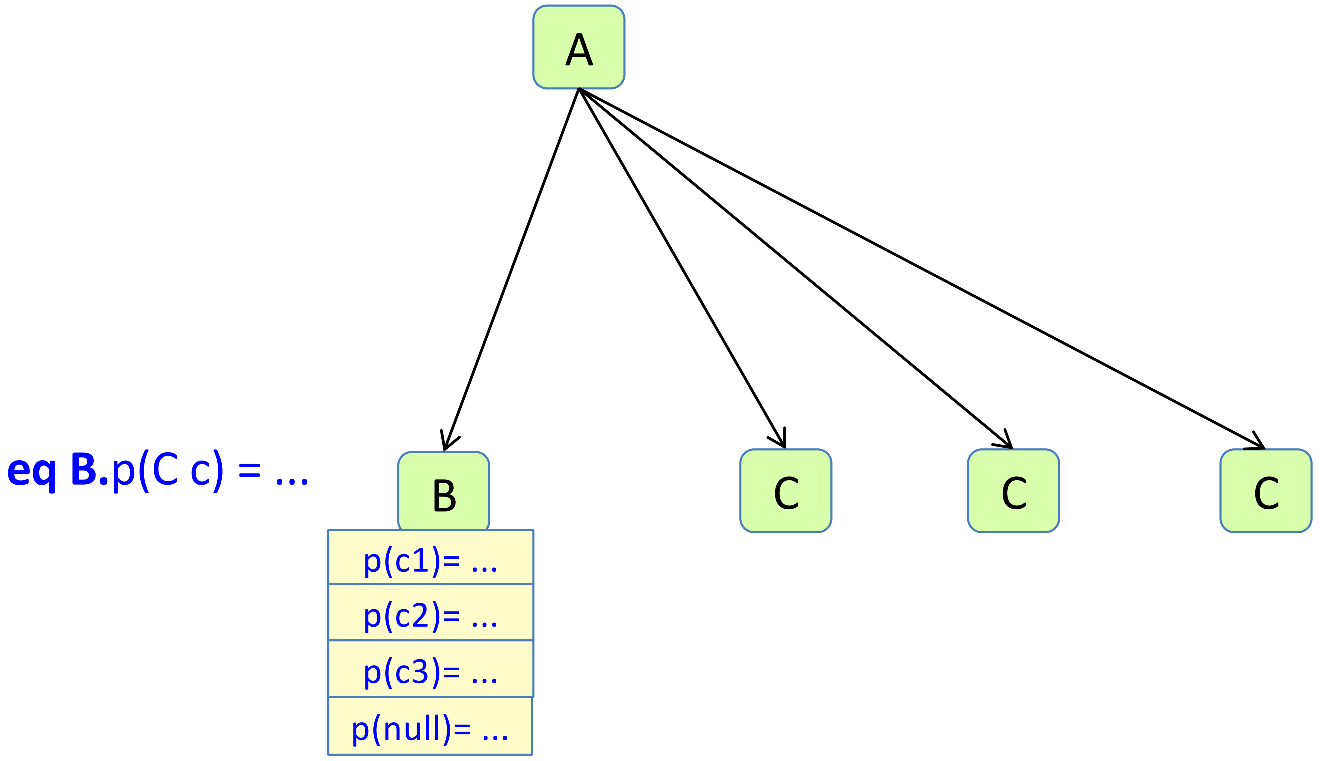
Reference attributes

eq B.aC = ...
eq ... = ... aC.x ...



An attribute can be a reference to another node.
Attributes of that node can be accessed.

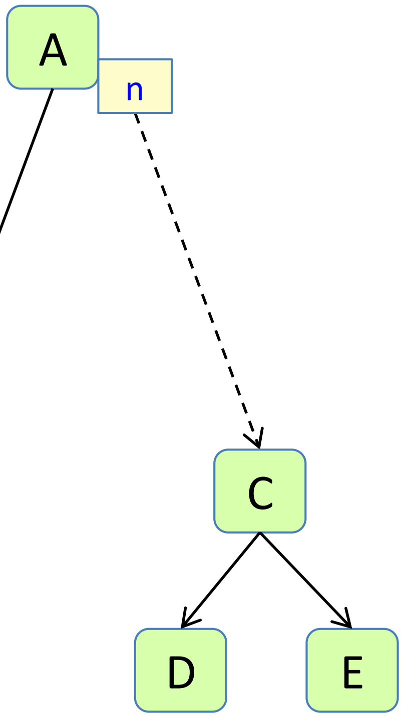
Parameterized attributes



An attribute can have parameters.
There is one attribute instance for each possible parameter combination.

Nonterminal attributes (NTAs)

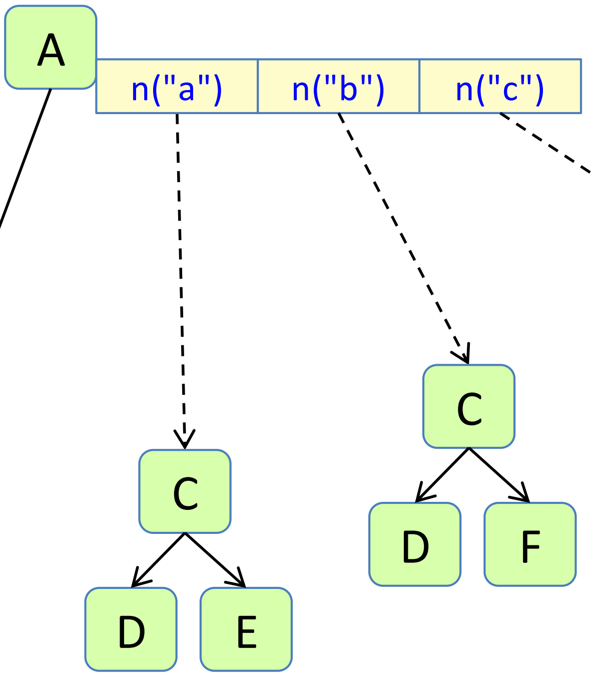
syn nta C A.n = ...



An attribute can be a new fresh subtree.

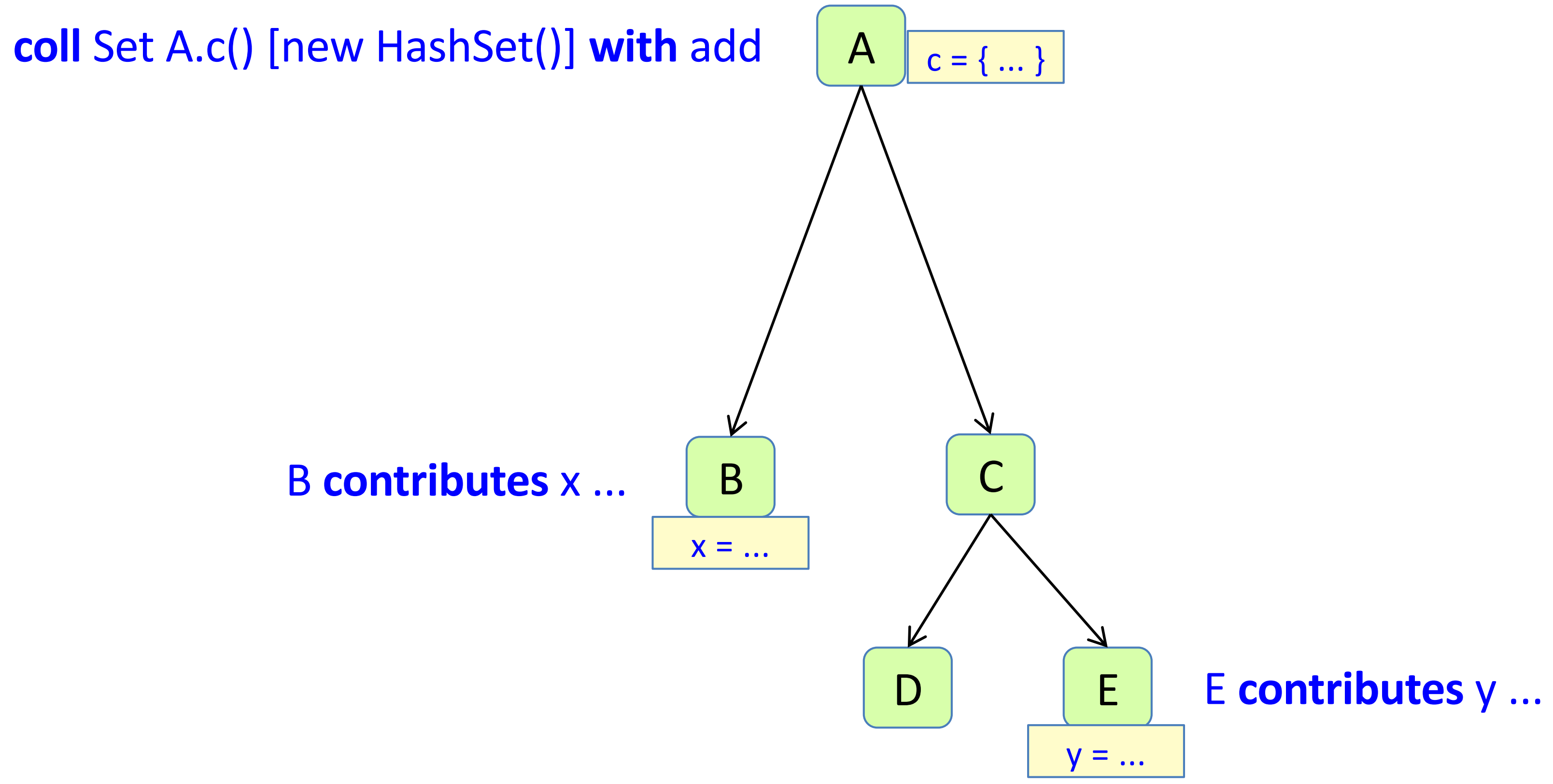
Parameterized nonterminal attribute

syn nta C A.n(String s) = ...



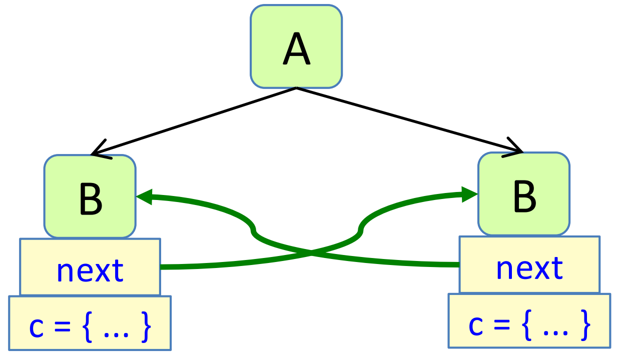
An NTA can be parameterized.

Collection attributes



A collection is a combination of contributions.

Circular attributes



The c attributes depend on each other

```
syn Set B.c() circular [new HashSet()] = ... next().c() ...;
```

A circular attribute depends (transitively) on itself.
Typically, several attributes depend on each other along a cyclic structure.
The evaluation algorithm uses fixed point iteration.

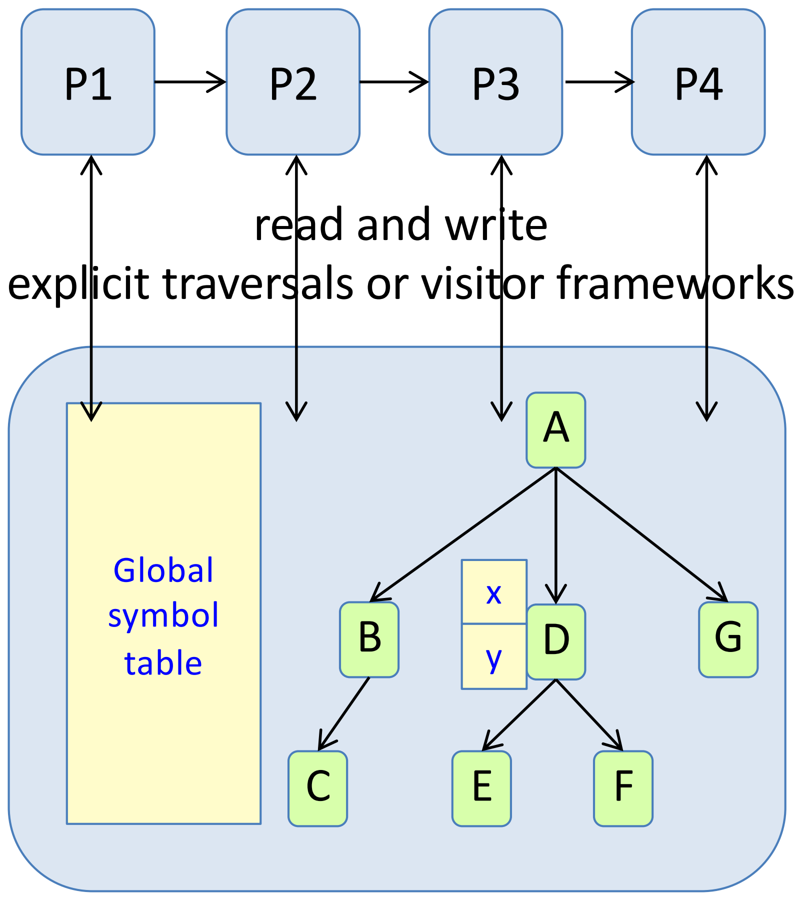
Attribute grammar systems

System	Developed at	Characteristics	Example applications
JastAdd	Lund University, Sweden http://jastadd.org	Generates Java	• Java compiler *ExtendJ* (Lund U) • Modelica compiler *OCT – Optimica Compiler Toolkit* (Modelon AB) • AspectJ compiler *abc*, built as extension to ExtendJ (U. of Oxford, McGill U.) • Java bytecode analysis framework *Soot*, uses ExtendJ as its frontend (McGill U., Paderborn U.) • Robotics applications (TU Dresden)
Silver	University of Minnesota, USA https://melt.cs.umn.edu/silver/	Generates Java	• Java compiler (*ableJ*, U. of Minnesota) • C compiler (*ableC*, U. of Minnesota) • Promela compiler (U. of Minnesota)
Kiama	Macquarie University, Australia https://github.com/inkytonik/kiama	Scala library	• Skink: Static analysis of LLVM • Cooma: Research on capability-based programming
JavaRAG	Lund University, Sweden https://bitbucket.org/javarag/javarag/	Java library	• Parts of the compiler for CAL (an actor language for stream processing)
RACR	Technical University of Dresden, Germany https://github.com/christoff-buerger/racr	Scheme library	
uuagc	University of Utrecht, The Netherlands https://hackage.haskell.org/package/uuagc	Generates Haskell	

Compiler architecture

Traditional pass-oriented

Passes compute data in explicit order

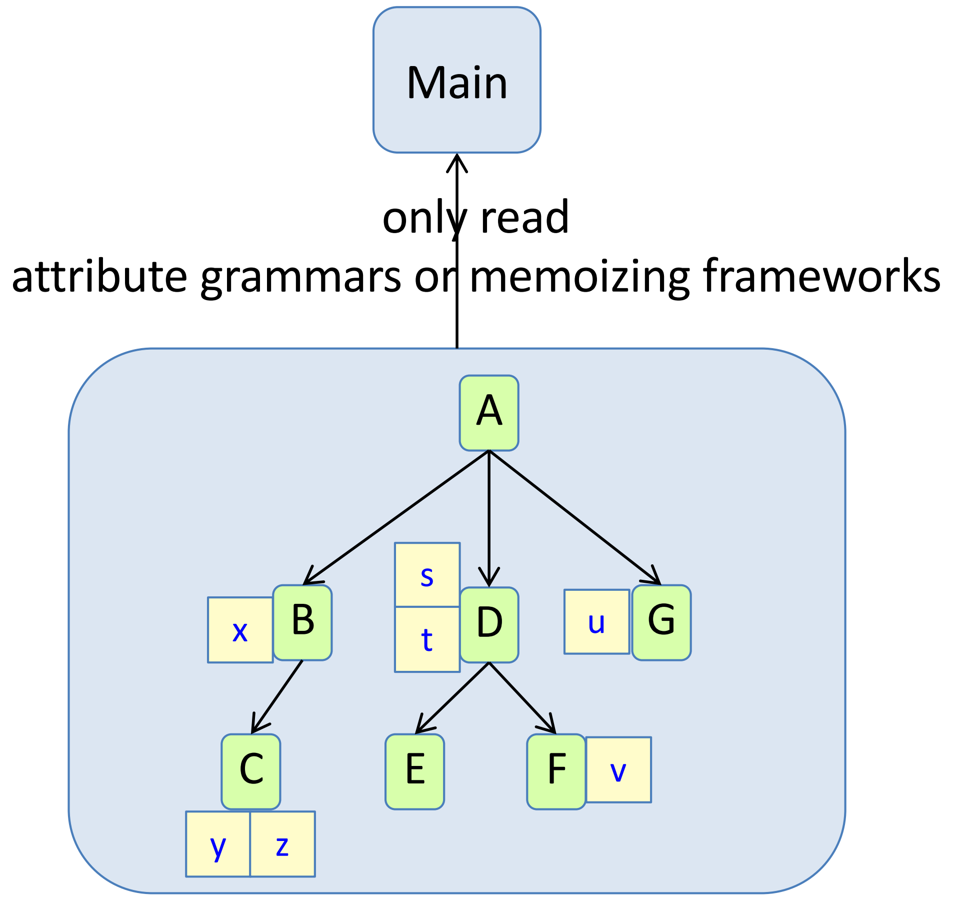


Examples (most compilers):

- OpenJDK Java compiler (visitors)
- Clang C compiler (visitors)
- ...

Query-based

Computation order is implicit



Examples (some newer compilers, in particular for IDE integration):

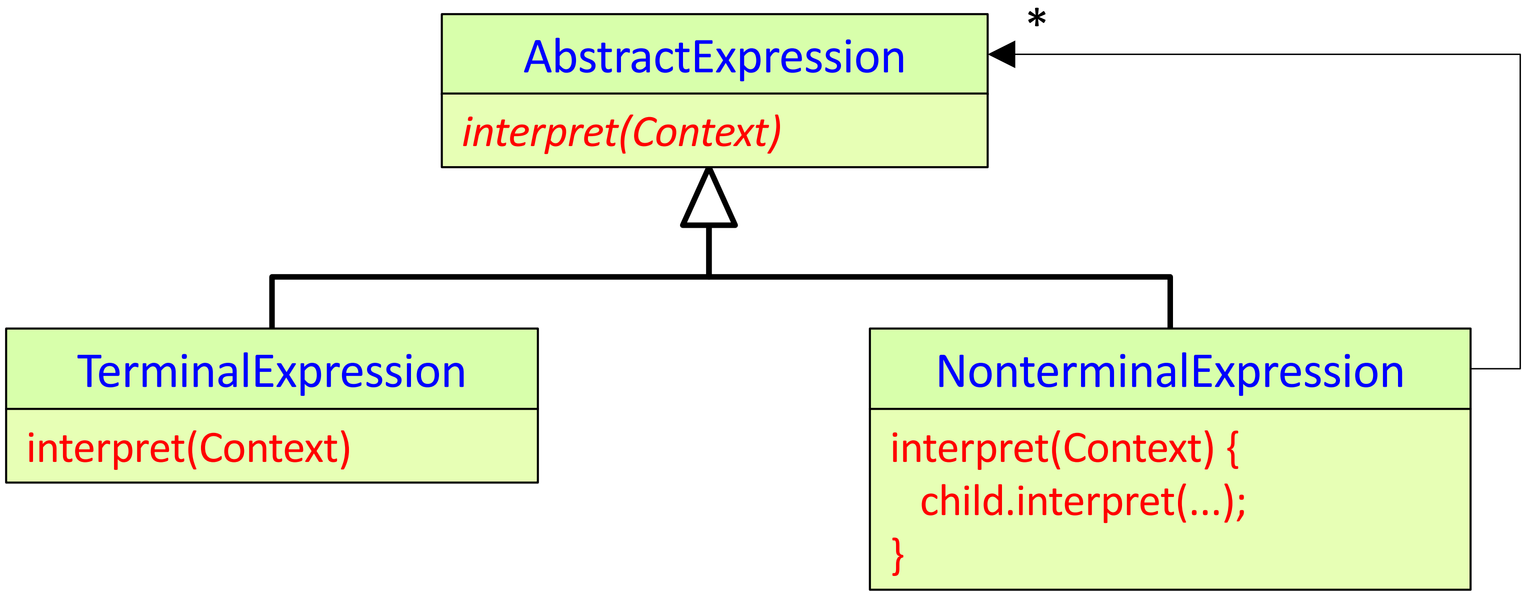
- ExtendJ (attribute grammars)
- Roslyn (query API)
- Rustc, RustAnalyzer (memoizing framework)
- ...

The Interpreter design pattern

The interpreter design pattern

Commonly used for many computations in a compiler.
Here explained using Ordinary OO. Modularize using AOP or Visitors.

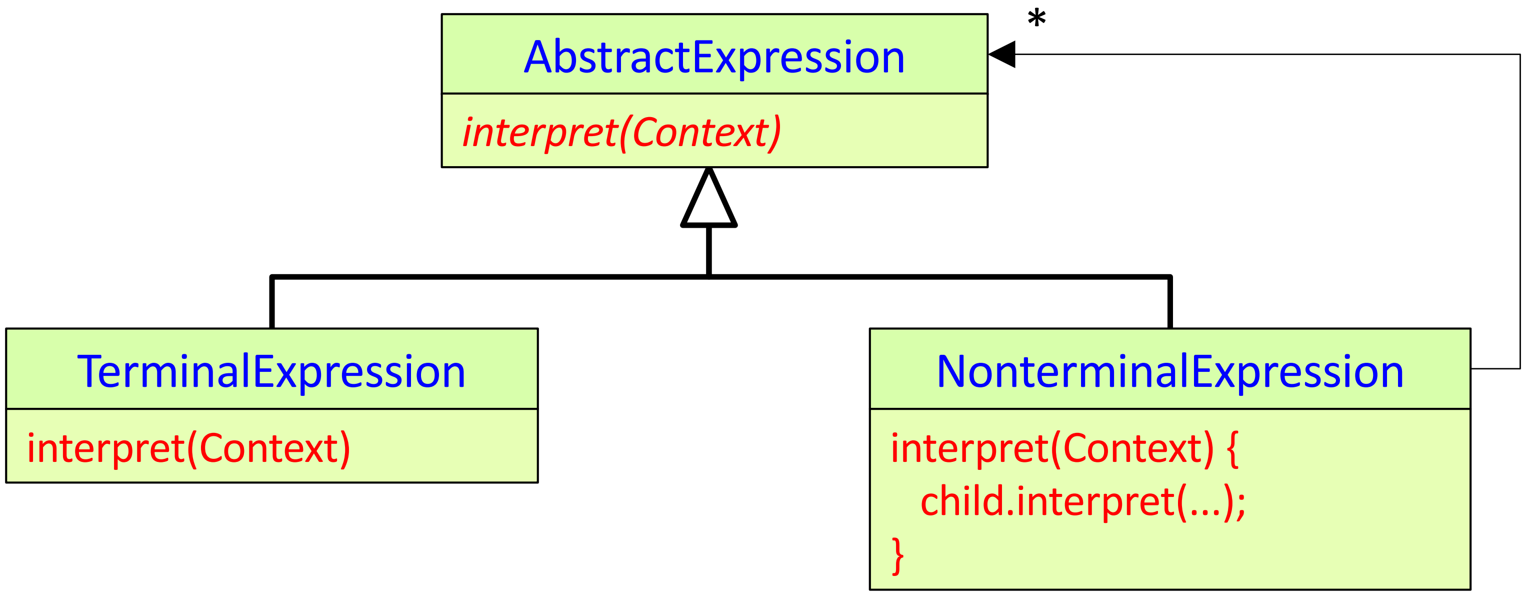
Intent: *Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.*
[Gamma, Helm, Johnson, Vlissides, 1994]



The interpreter design pattern

Commonly used for many computations in a compiler.
Here explained using Ordinary OO. Modularize using AOP or Visitors.

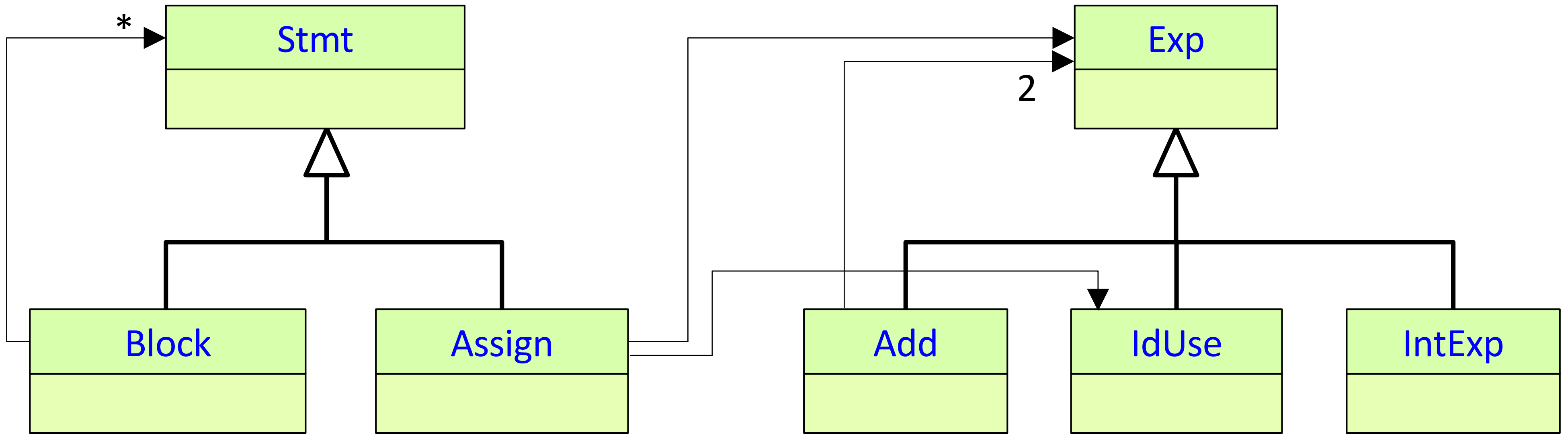
Intent: *Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.*
[Gamma, Helm, Johnson, Vlissides, 1994]



AbstractExpression, TerminalExpression, NonterminalExpression, interpret, and Context are just ROLES in the pattern.
In our programs, we will use our own names.

Example use of Interpreter

```
abstract Stmt;
Block : Stmt ::= Stmt*;
Assign : Stmt ::= IdUse Exp;
abstract Exp;
Add : Exp ::= Left:Exp Right:Exp;
IdUse : Exp ::= <ID>;
IntExp : Exp ::= <INT>;
```



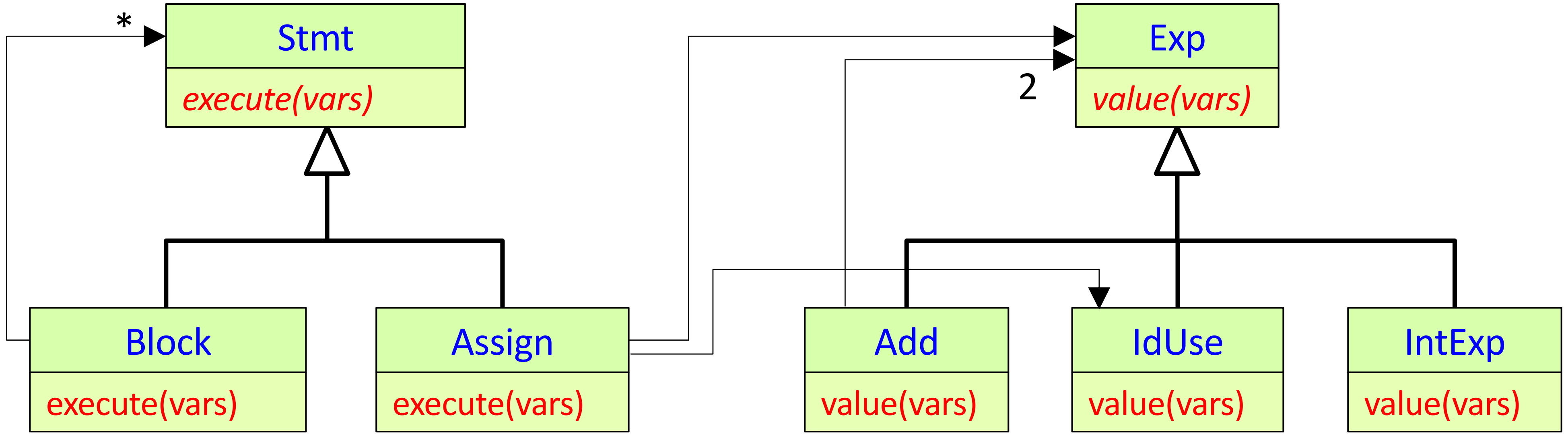
```
abstract Stmt;
Block : Stmt ::= Stmt*;
Assign : Stmt ::= IdUse Exp;
abstract Exp;
Add : Exp ::= Left:Exp Right:Exp;
IdUse : Exp ::= <ID>;
IntExp : Exp ::= <INT>;
```

Example use of Interpreter

Pattern roles:

context: *vars*

interpret: *execute*, *value*



vars	a map String -> Value, keeping track of the current values of variables
execute	executes a Stmt, changing and using the vars map
value	executes an Exp and returns its value, making use of the vars map

Example implementation using JastAdd aspects

```
abstract Stmt;
Block : Stmt ::= Stmt*;
Assign : Stmt ::= IdUse Exp;
abstract Exp;
Add : Exp ::= Left:Exp Right:Exp;
IdUse : Exp ::= <ID:String>;
IntExp : Exp ::= <INT:String>;
```

```
aspect Interpreter {
  abstract void Stmt.execute(Map<String, Integer> vars);

  void Block.execute(Map<String, Integer> vars) {
    for (Stmt s : getStmts()) { s.execute(vars); }
  }
  void Assign.execute(Map<String, Integer> vars) {
    int value = getExp().value(vars);
    vars.put(getIdExp().getID(), value);
  }

  abstract int Exp.value(Map<String, Integer> vars);

  int Add.value(Map<String, Integer> vars) {
    return getLeft().value(vars) + getRight().value(vars);
  }
  int IdUse.value(Map<String, Integer> vars) {
    return vars.get(getID());
  }
  int IntExp.value(Map<String, Integer> vars) {
    return String.parseInt(getINT());
  }
}
```

Summary questions:

- Give examples of properties that can be computed using collection attributes.
- 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.
- How does the Interpreter design pattern work?