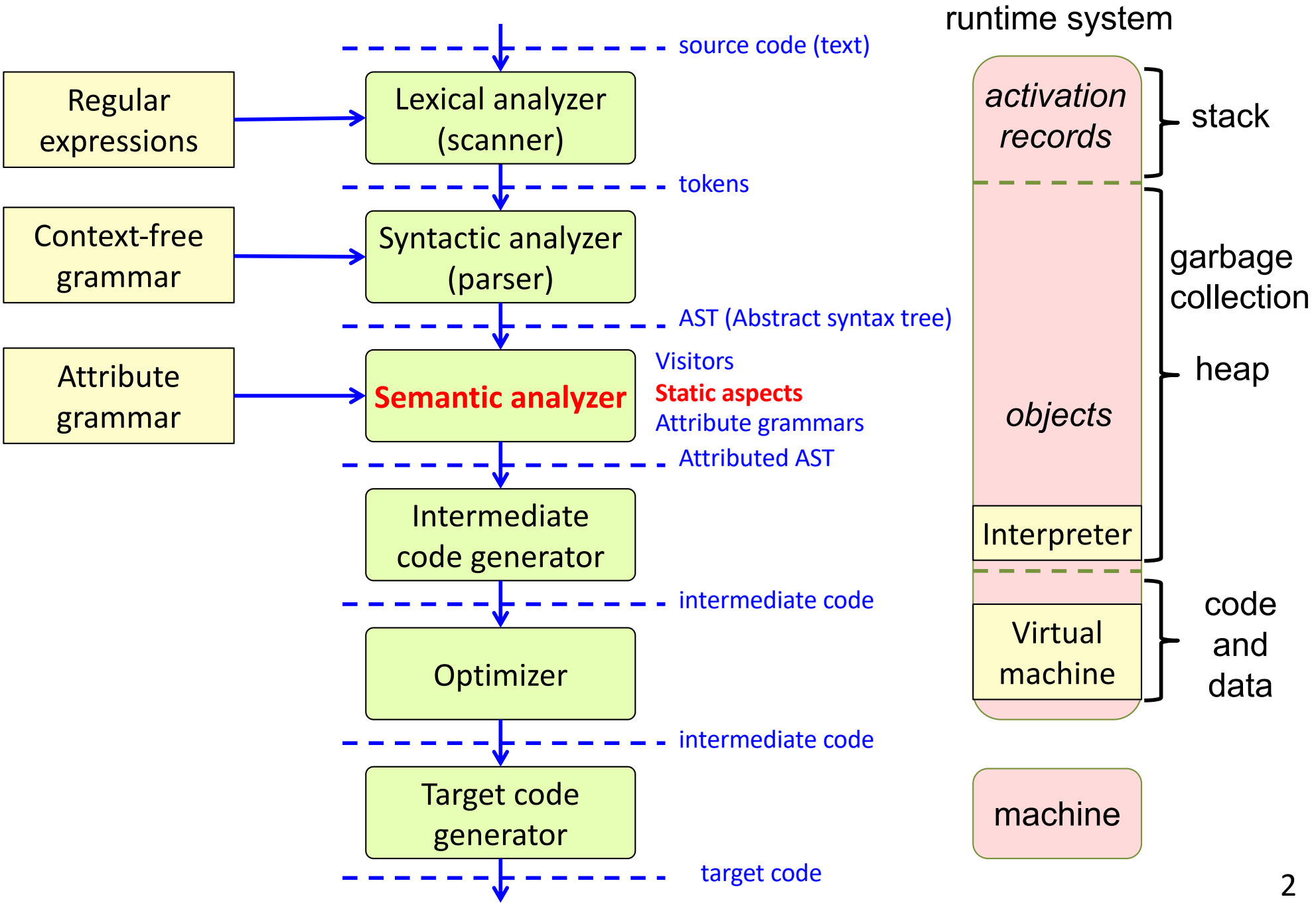
EDAN65: Compilers, Lecture 07 A

Static Aspect-Oriented Programming

Görel Hedin

Revised: 2020-09-21

This lecture



Recall

Semantic analysis

computations on the AST: name analysis, type analysis, error checking, ...

Expression problem

How can we add both operations and language constructs modularly?
We need extensibility in two dimensions: operations and language constructs.

Solutions to the expression problem

- Limited solution 1: Visitors (previous lecture)
- Solution 2: Static AOP (this lecture)

Ordinary programming

Example: Printing an AST

```
class Exp {
  abstract void print();
}
class Add extends Exp {
  Exp e1, e2;
  void print() {
    e1.print();
    System.out.print("+");
    e2.print();
  }
}
class IntExp extends Exp {
  int value;
  void print() {
    System.out.print(value);
  }
}
...
```

Pros/Cons?

Ordinary programming

Example: Printing an AST

```
class Exp {
  abstract void print();
}
class Add extends Exp {
  Exp e1, e2;
  void print() {
    e1.print();
    System.out.print("+");
    e2.print();
  }
}
class IntExp extends Exp {
  int value;
  void print() {
    System.out.print(value);
  }
}
...
```

Pros:

- Straightforward code
- Modular extension in the language dimension (subclasses)

Cons:

- No modular extension in the operation dimension – all classes need to be modified.
- Tangled code – many different concerns in the same class.

Visitor solution

Example: Printing an AST

```
class Exp {
}
class Add extends Exp {
  Exp e1, e2;
  void accept(Visitor v) {
    v.visit(this);
  }
}
class IntExp extends Exp {
  int value;
  void accept(Visitor v) {
    v.visit(this);
  }
}
...
```

```
class Unparser implements Visitor {
  void visit(Add node) {
    node.e1.accept(this);
    System.out.print("+");
    node.e2.accept(this);
  }
  void visit(IntExpr node) {
    System.out.print(node.value);
  }
}
```

Pros/Cons?

Visitor solution

Example: Printing an AST

```
class Exp {
}
class Add extends Exp {
  Exp e1, e2;
  void accept(Visitor v) {
    v.visit(this);
  }
}
class IntExp extends Exp {
  int value;
  void accept(Visitor v) {
    v.visit(this);
  }
}
...
```

```
class Unparser implements Visitor {
  void visit(Add node) {
    node.e1.accept(this);
    System.out.print("+");
    node.e2.accept(this);
  }
  void visit(IntExpr node) {
    System.out.print(node.value);
  }
}
```

Pros:

- Modular extension in the operation dimension (add new visitor).

Cons:

- Boilerplate code needed (accept and visit methods).
- Limited modular extensibility in the language dimension. Needs lots of boilerplate.

Static Aspect-Oriented Programming

Example: Printing an AST

```
class Exp {
}
class Add extends Exp {
  Exp e1, e2;
}
class IntExp extends Exp {
  int value;
}
...
```

```
aspect Unparser {
  abstract void Exp.print();
  void Add.print() {
    e1.print();
    System.out.print("+");
    e2.print();
  }
  void IntExp.print() {
    System.out.print(value);
  }
}
```

Pros/Cons?

Static Aspect-Oriented Programming

Example: Printing an AST

```
class Exp {
}
class Add extends Exp {
  Exp e1, e2;
}
class IntExp extends Exp {
  int value;
}
...
```

```
aspect Unparser {
  abstract void Exp.print();
  void Add.print() {
    e1.print();
    System.out.print("+");
    e2.print();
  }
  void IntExp.print() {
    System.out.print(value);
  }
}
```

Pros:

- Straightforward code.
- Modular extension in the operation dimension (can be added in aspect).
- Modular extension in the language dimension (add new subclass, add operation code for those constructs in aspect).

Cons:

- Cannot use plain Java. Need more advanced language like AspectJ or JastAdd.
- Typically no separate compilation of modules. (Modules woven before compilation)

Inter-type declarations

The key construct in static AOP

```
class C {
   int x;
}
```

```
class D {
}
```

```
aspect A {
  T C.m() {
    x = ...;
    ...
  }
  int D.f = 3;
}
```

inter-type declared method

inter-type declared field

Inter-type declarations

The key construct in static AOP

```
class C {
   int x;
}
```

```
class D {
}
```

```
aspect A {
  T C.m() {
    x = ...;
    ...
  }
  int D.f = 3;
}
```

inter-type declared method

inter-type declared field

is equivalent to:

```
class C {
   int x;
   T m() {
      x = ...;
      ...
   }
}
```

```
class D {
   int f = 3;
}
```

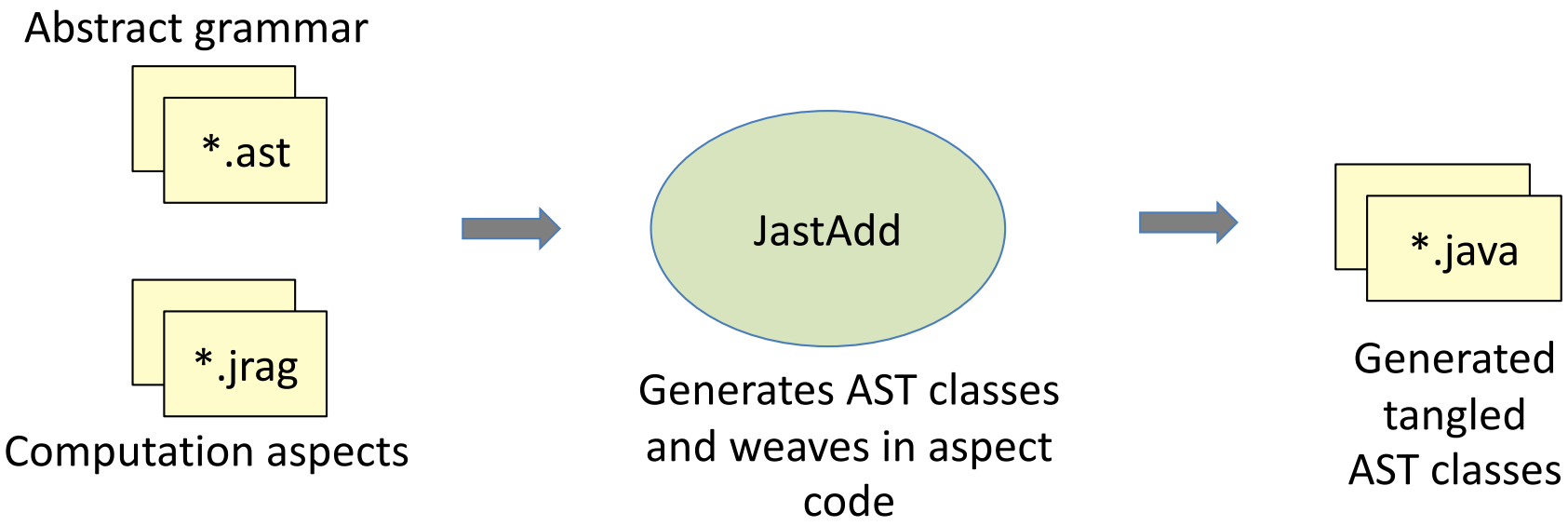
Recall: Dealing with the expression problem

- Edit the AST classes (i.e., actually not solving the problem)
  - Non-modular, non-compositional.
  - **It is always a VERY BAD IDEA to edit generated code!**
  - Sometimes used anyway in industry.
- Visitors: an OO design pattern.
  - Modularize operations through double dispatch.
  - Not full modularization, not composition.
  - Supported by many parser generators.
  - Reasonably useful, commonly used in industry.
- Static Aspect-Oriented Programming (AOP)
  - Also known as *inter-type declarations* (ITDs) or *introduction*
  - Use new language constructs (aspects) to factor out code.
  - Solves the expression problem in a nice simple way.
  - The drawback: you need a new language: AspectJ, JastAdd, ...
- Advanced language constructs
  - Use more advanced language constructs: virtual classes in gbeta, traits in Scala, typeclasses in Haskell, ...
  - Drawbacks: Much more complex than static AOP. You need an advanced language. Not much practical experience (so far).

This lecture: Static AOP

Static AOP in JastAdd

Static AOP in JastAdd



Example aspect: expression evaluation

Abstract grammar

```
abstract Exp;
abstract BinExp : Exp ::= Left:Exp Right:Exp;
Add : BinExp;
Sub : BinExp;
IntExp : Exp ::= <INT:String>;
```

Example aspect: expression evaluation

Abstract grammar

```
abstract Exp;
abstract BinExp : Exp ::= Left:Exp Right:Exp;
Add : BinExp;
Sub : BinExp;
IntExp : Exp ::= <INT:String>;
```

Aspect

```
aspect Evaluator {
  abstract int Exp.value();
  int Add.value() { return getLeft().value() + getRight().value(); }
  int Sub.value() { return getLeft().value() – getRight().value(); }
  int IntExp.value() { return String.parseInt(getINT()); }
}
```

***Inter-type declarations**:* The value methods will be woven into the classes (Expr, Add, Sub, IntExpr).
Inter-type declarations are also known as *introductions*.

Another example: unparsing

Abstract grammar

```
abstract Exp;
abstract BinExp : Exp ::= Left:Exp Right:Exp;
Add : BinExp;
Sub : BinExp;
IntExp : Exp ::= <INT:String>;
```

Another example: unparsing

Abstract grammar

```
abstract Exp;
abstract BinExp : Exp ::= Left:Exp Right:Exp;
Add : BinExp;
Sub : BinExp;
IntExp : Exp ::= <INT:String>;
```

Aspect

```
aspect Unparser {
  abstract void Exp.unparse(Stream s, String indent);
  void BinExp.unparse(Stream s, String indent) {
    getLeft().unparse(s,indent);
    s.print(operatorString());
    getRight().unparse(s,indent);
  }
  abstract String BinExp.operatorString();
  String Add.operatorString() { return "+"; }
  String Sub.operatorString() { return "-"; }
  void IntExp.unparse(Stream s, String indent) { s.print(getINT()); }
}
```

Weaving the classes in JastAdd

toy.ast

```
abstract Exp;
abstract BinExp : Exp ::= Left:Exp Right:Exp;
Add : BinExp;
Sub : BinExp;
IntExp : Exp ::= <INT:String>;
```

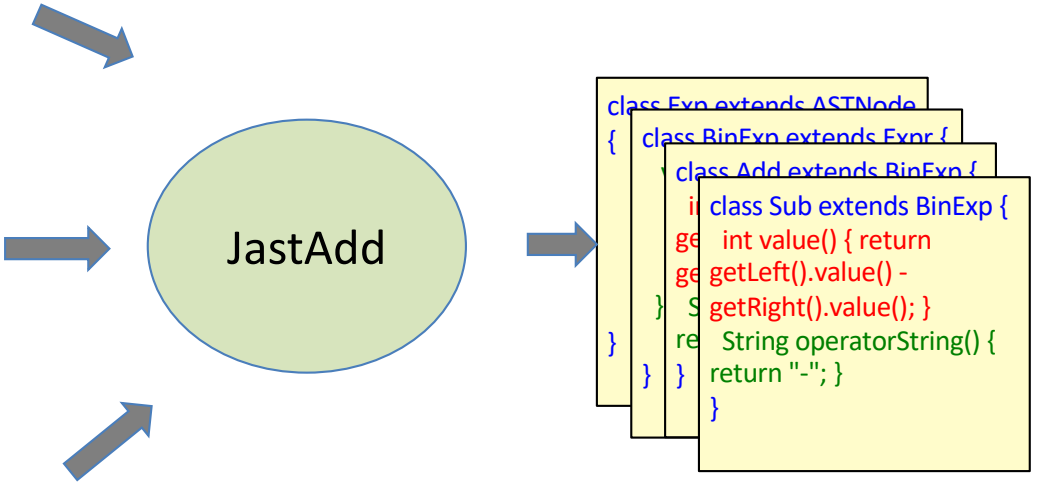
Evaluator.jrag

```
aspect Evaluator {
  abstract int Exp.value();
  int Add.value() { return getLeft().value() + getRight().value(); }
  int Sub.value() { return getLeft().value() – getRight().value(); }
  int IntExp.value() { return String.parseInt(getINT()); }
}
```

Unparser.jrag

```
aspect Unparser {
  abstract void Exp.unparse(Stream s, String indent);
  void BinExp.unparse(Stream s, String indent) {
    getLeft().unparse(s,ind);
    s.print(operatorString());
    getRight().unparse(s,ind);
  }
  abstract BinExp.operatorString();
  String Add.operatorString() { return "+"; }
  String Sub.operatorString() { return "-"}
  void IntExp.unparse(Stream s, String indent) { s.print(getINT()); }
}
```

Untangled source code



Tangled generated code

Features that can be factored out to aspects in JastAdd

- Methods
- Instance variables
- "implements" clauses
- "import" clauses
- attribute grammars (see later lecture)

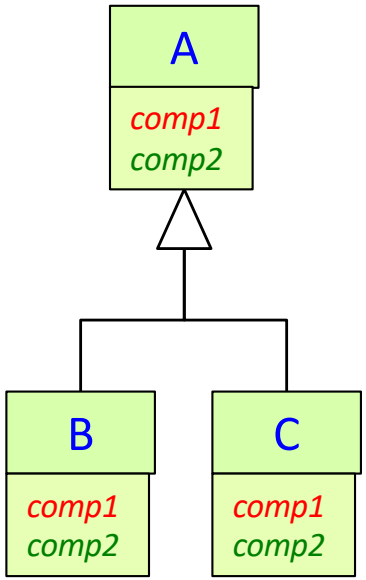
Static aspects vs Visitors

	Static aspects	Visitors
What can be factored out from AST classes?	instance variables methods implements clauses	only methods
Type safety?	full type precision	Casts may be needed, depending on framework
Method parameters	any number	only one
Ease of use?	Very simple	Clumsy, boilerplate code needed.
Arbitrary composition of modules?	Yes	No – you can extend a visitor, but then you need factories to create them. And you cannot not easily combine two extensions.
Separate compilation?	Not for JastAdd or AspectJ.	Yes
Mainstream OO language?	No – you need JastAdd, AspectJ, or similar	Yes, use Java or any other OO language.

Recall: The expression problem

How add both classes and computations in a modular way?

Ordinary OO

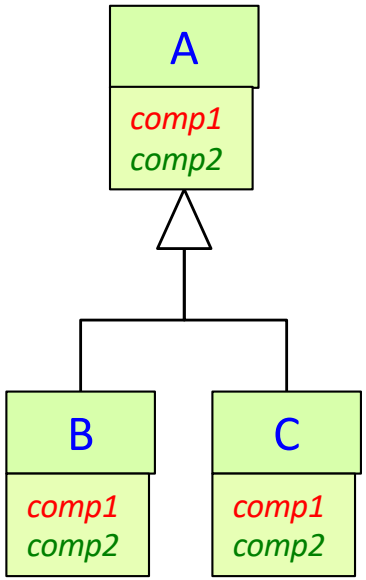


Classes can be added modularly, but not computations.
Simple code.

Recall: The expression problem

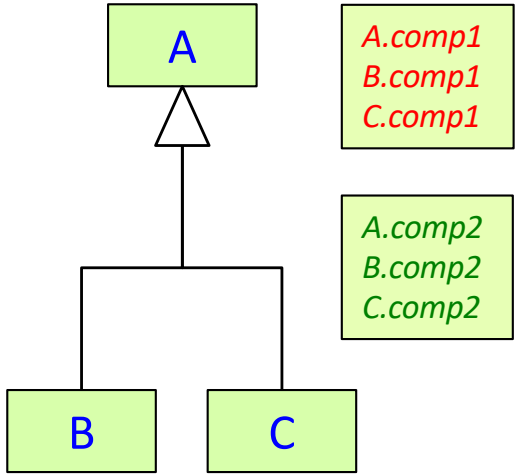
How add both classes and computations in a modular way?

Ordinary OO



Classes can be added modularly, but not computations.
Simple code.

Aspects with inter-type declarations

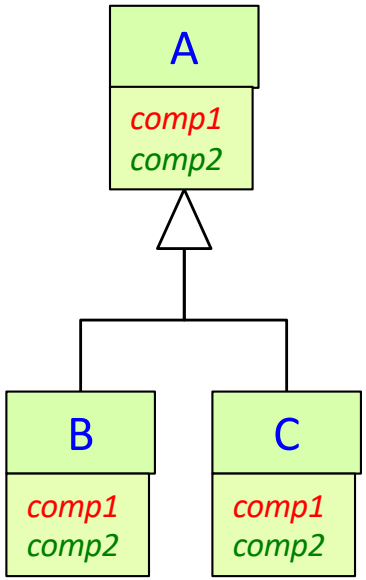


Fully modular.
Simple code.

Recall: The expression problem

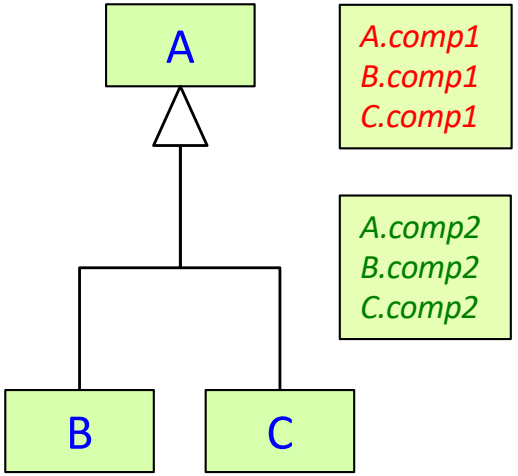
How add both classes and computations in a modular way?

Ordinary OO



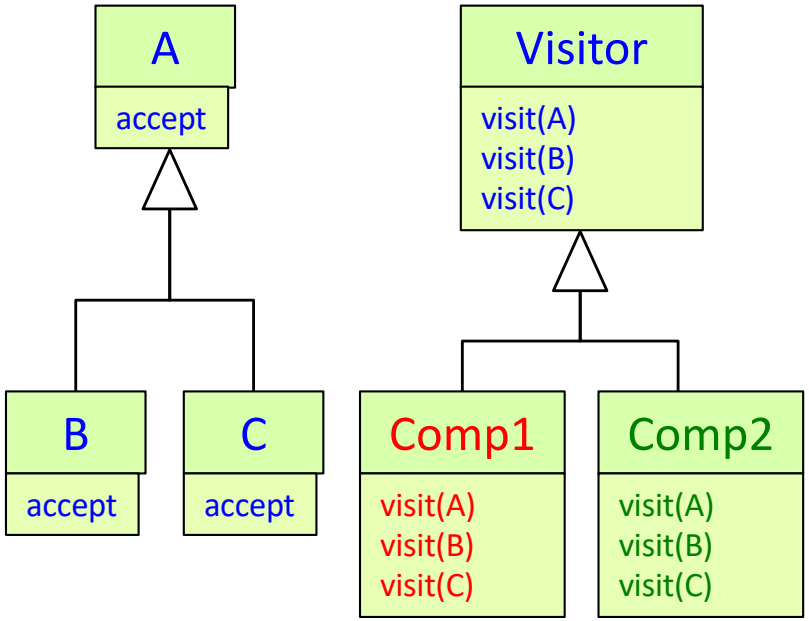
Classes can be added modularly, but not computations. Simple code.

Aspects with inter-type declarations



Fully modular. Simple code.

The Visitor design pattern



Computations can be added, but non-modular changes usually needed if classes are added. Complex code.

Full Aspect-Oriented Programming

Full AOP = Static AOP + Dynamic AOP

Full Aspect-Oriented Programming

Full AOP = Static AOP + Dynamic AOP

inter-type declarations

advice
joinpoints
pointcuts

Modular code.

Modular instrumentation.

Full Aspect-Oriented Programming

- JastAdd supports only a small part of AOP, namely *static* AOP with inter-type declarations.

- Aspect-oriented programming is a wider concept that usually focuses on *dynamic* behavior: a general code instrumentation technique:
  - A joinpoint is a point during execution where advice code can be added.
  - A pointcut is a set of joinpoints that can be described in a simple way, e.g.,
    - all calls to a method m()
    - all accesses of a variable v
  - Advice is code you can specify in an aspect and that can be added at joinpoints, either after, before, or around the joinpoint.
  - Example applications:
    - Add logging of method calls in an aspect (instead of adding print statements all over your code)
    - Add synchronization code to basic code that is unsynchronized

Summary questions

- What are different ways of solving the Expression Problem?
- What is an intertype declaration?
- What is aspect-oriented programming?
- How does static AOP differ from dynamic AOP?
- Implement a computation over the AST using static aspects.
- What are advantages and disadvantages of static AOP as compared to Visitors?