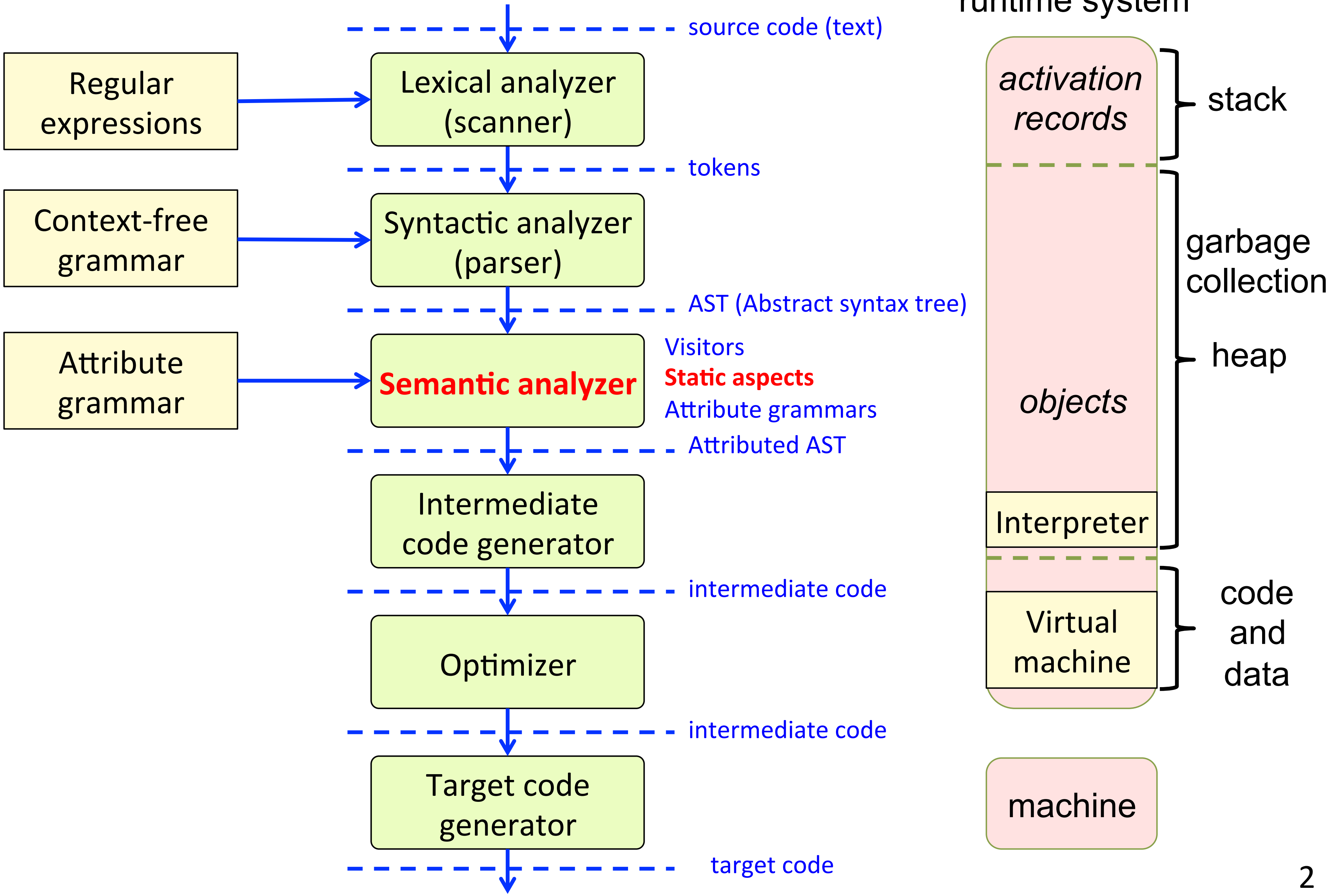
EDAN65: Compilers, Lecture 07 A

Static Aspect-Oriented Programming

Görel Hedin

Revised: 2015-09-21

This lecture



Recall

Semantic analysis

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

Expression problem

How can we add both computations and language constructs modularly?

Solutions to the expression problem

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

Example: Printing an AST

Ordinary programming

```
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

Cons:
If we add a new operation, like computing the value, all classes need to be modified. We get tangled code – many different concerns in the same class.

Example: Printing an AST

Visitor solution

```
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 Evaluator 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 addition of new operation

Cons: Clumsy code with lots of boilerplate (accept and visit methods). Cannot extend visitors easily if the language is extended.

Example: Printing an AST

Static Aspect-Oriented Programming

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

```
aspect Evaluator {
  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 addition of new operation. No problem to extend the language – additional methods can be added in other aspect.

Cons: Cannot use Java. Need more advanced language like AspectJ or JastAdd.

Inter-type declarations

The key construct in static AOP

```
class C {
}
```

```
class D {
}
```

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

inter-type declared method

inter-type declared field

is equivalent to:

```
class C {
 T m() {
     …
   }
}
```

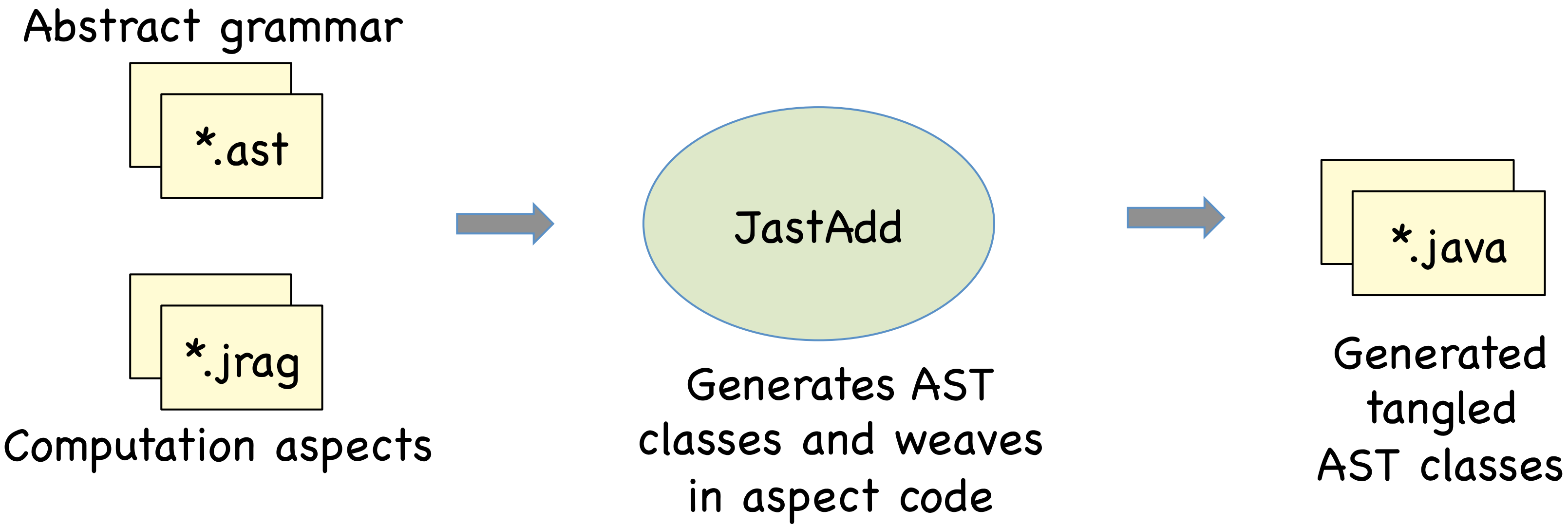
```
class D {
 int f = 3;
}
```

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 through clever indirect calls.
  - 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: More complex than static AOP. You need an advanced language. Not much practical experience (so far).

This lecture: Static AOP

Static AOP in JastAdd



Abstract grammar

*.ast

*.jrag

Computation aspects

JastAdd

Generates AST classes and weaves in aspect code

*.java

Generated tangled AST classes

Example aspect: expression evaluation

Abstract grammar

```
abstract Expr;
BinExpr : Expr ::= Left:Expr Right:Expr;
Add : BinExpr;
Sub : BinExpr;
IntExpr : Expr ::= <INT:String>;
```

Aspect

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

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

Another example: unparsing

Abstract grammar

```
abstract Expr;
BinExpr : Expr ::= Left:Expr Right:Expr;
Add : BinExpr;
Sub : BinExpr;
IntExpr : Expr ::= <INT:String>;
```

Aspect

```
aspect Unparser {
  abstract void Expr.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 IntExpr.unparse(Stream s, String indent) { s.print(getINT()); }
}
```

Weaving the classes in JastAdd

toy.ast

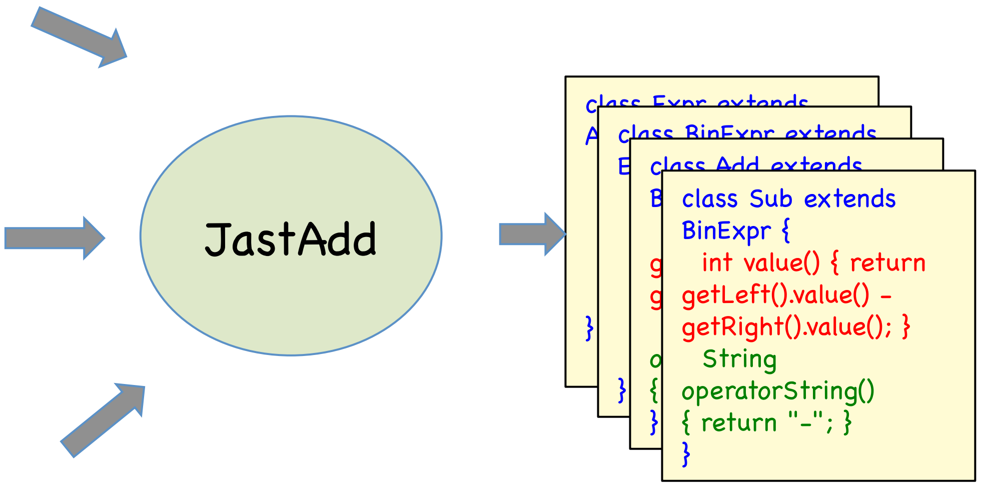
```
abstract Expr;
BinExpr : Expr ::= Left:Expr Right:Expr;
Add : BinExpr;
Sub : BinExpr;
IntExpr : Expr ::= <INT:String>;
```

Evaluator.jrag

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

Unparser.jrag

```
aspect Unparser {
  abstract void Expr.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 IntExpr.unparse(Stream s, String indent) { s.print(getINT()); }
}
```



```
class Sub extends
BinExpr {
  int value() { return
getLeft().value() -
getRight().value(); }
  String
operatorString()
{ return "-"; }
}
```

Tangled generated code

Untangled source code

Features that can be inter-type declared or factored out to JastAdd aspects

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

Full Aspect-Oriented Programming

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

- Asepct-oriented programming is a wider concept that usually focuses on *dynamic* behavior:
  - 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

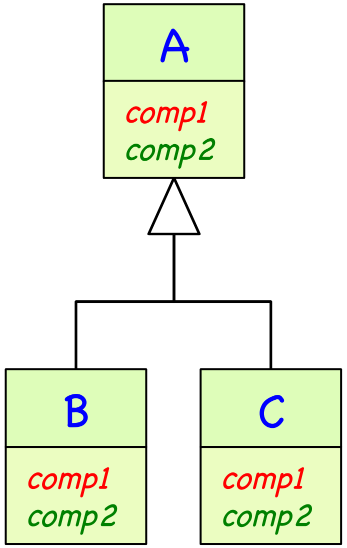
Static aspects vs Visitors

	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 not combine two.
Separate compilation?	Not for JastAdd. But could be implemented.	Yes
Mainstream OO language?	No – you need JastAdd, AspectJ, or similar	Yes, use Java or any other OO language.

Recall: The expression problem

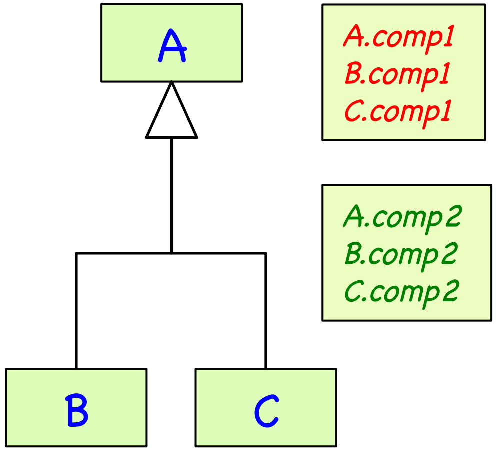
How add both classes and computations in a modular way?

Ordinary OO



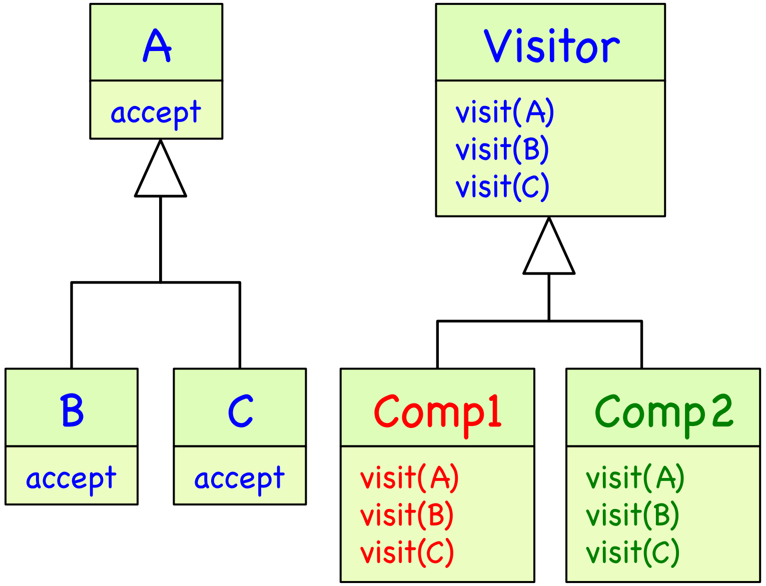
Classes can be added modularly, but not computations.

Aspects with inter-type declarations



Fully modular.

The Visitor design pattern

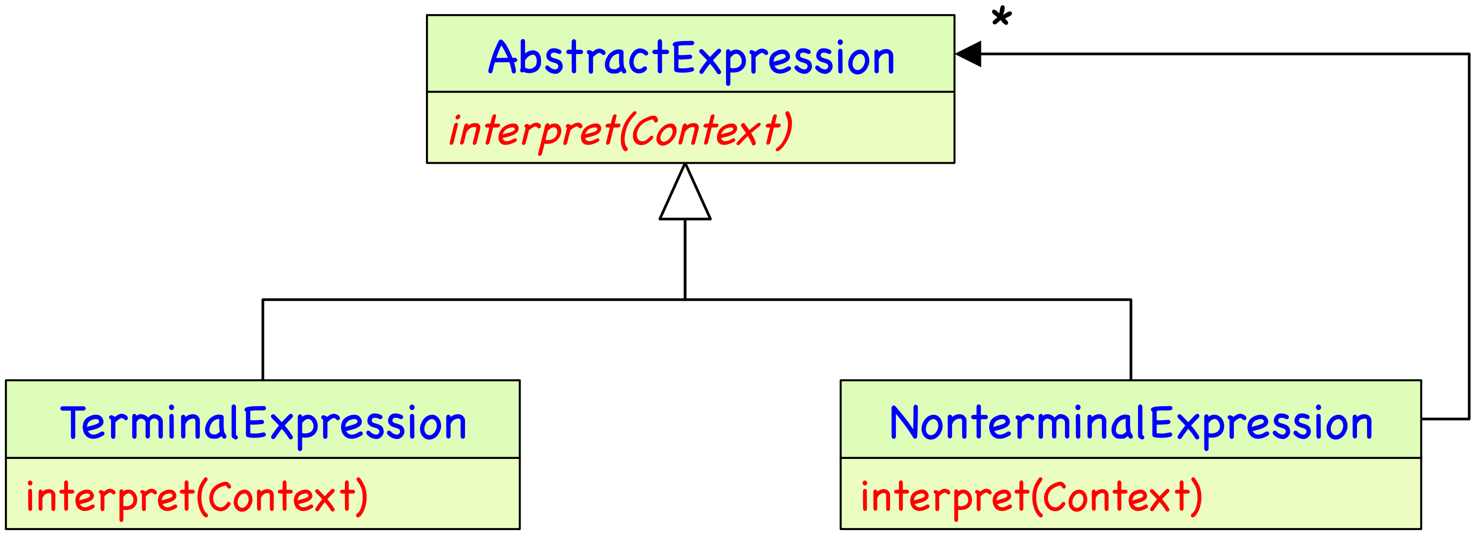


Computations can be added, but non-modular changes needed if classes are added. Complex code.

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.

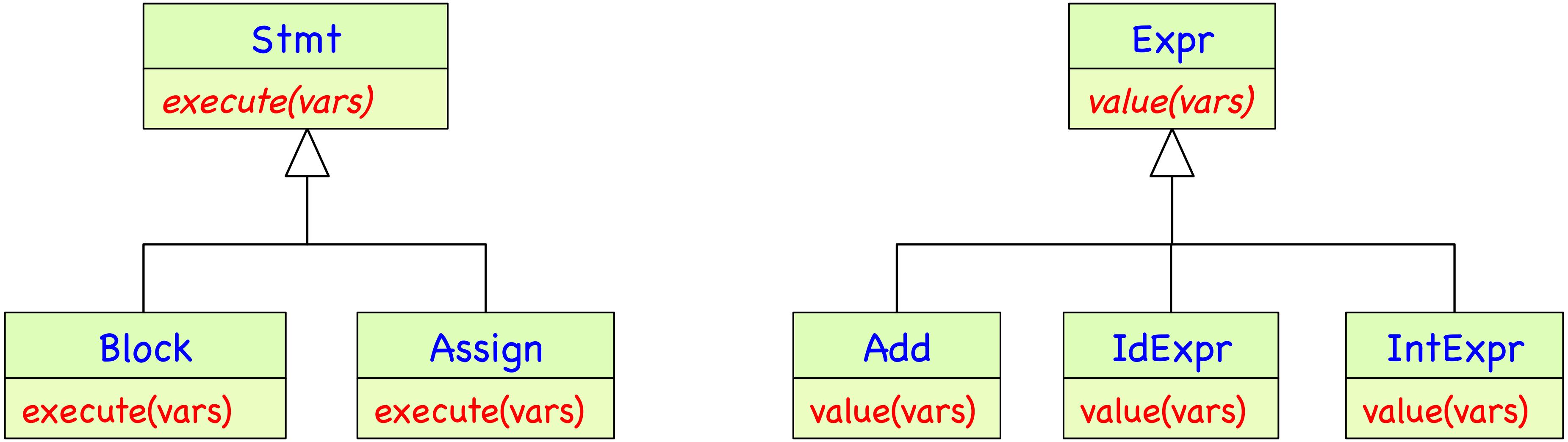
```
abstract Stmt;
Block : Stmt ::= Stmt*;
Assign : Stmt ::= <ID> Expr;
abstract Expr;
Add : Expr ::= Left:Expr Right:Expr;
IdExpr : Expr ::= <ID>;
IntExpr : Expr ::= <INT>;
```

Example use of Interpreter

Pattern roles:

context: *vars*

interpret: *execute* for statements, *value* for expressions



vars — a map keeping track of the current values of variables

execute — executes a Stmt, changing and using the vars map

value — returns the value of an Expr, using the vars map

Example implementation using JastAdd aspects

```
abstract Stmt;
Block : Stmt ::= Stmt*;
Assign : Stmt ::= <ID> Expr;
abstract Expr;
Add : Expr ::= Left:Expr Right:Expr;
IdExpr : Expr ::= <ID>;
IntExpr : Expr ::= <INT>;
```

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

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

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

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

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?