Compiler Construction

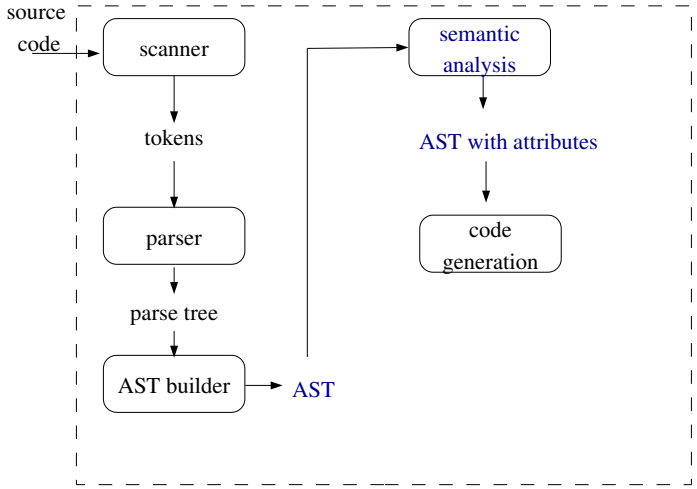
Computations on ASTs
Aspect oriented programming

Lennart Andersson

Revision 2012-01-31

2012

Semantic analysis



Examples of computations

- 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 a program
- Unparsing: compute a text representation of a program

An expression evaluator

Abstract grammar

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

Evaluator implementation

```
abstract class Expr {
  abstract int value();
}
abstract class BinExpr extends Expr { }
class Add extends BinExpr {




}
class Sub extends BinExpr {

}
class IntExpr extends Expr {




}
```


An unparser

```
abstract Stmt;
IfStmt : Stmt ::= Cond:Expr Then:Stmt [Else:Stmt];

abstract class Stmt {
  abstract void unparse(PrintStream s, String indent);
}
class IfStmt extends Stmt {
  void unparse(PrintStream s, String indent) {
    s.print(indent);
    s.print("if ");
    getCond().unparse(s, indent);
    s.println(" then ");
    getThen().unparse(s, indent + "  ");
    if (hasElse()) {
      s.println(" else ");
      getElse().unparse(s, indent + "  ");
    }
  }
```


The Interpreter design pattern

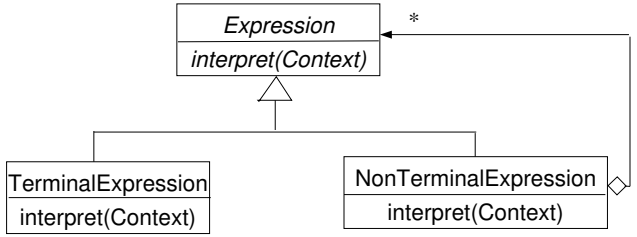
Intent

> *Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.*

Example

> *"Computer" in the OMD course (eda061/edaf10)*

Interpreter pattern



Example applications

The expression evaluator

- value() plays the role of the interpret method
- no Context was needed (but would be needed if we had a language with identifiers)
- Expr plays the role of the abstract Expression
- IntExpr plays the role of TerminalExpression
- Add and Sub play the role of NonTerminalExpression

The unparser

- unparse() plays the role of the interpret method
- Context consists of two parameters: (PrintStream s, String indent)
- Stmt and Expr play the role of abstract Expressions
- IfStmt plays the role of NonTerminalExpression
- ...

Count the number of identifiers

Abstract grammar

```
abstract Stmt;
IfStmt : Stmt ::= Cond:Expr Then:Stmt [Else:Stmt];
...
abstract Expr;
abstract BinExpr : Expr ::= Left:Expr Right:Expr;
Add : BinExpr;
Sub : BinExpr;
IntExpr : Expr ::= <INT:String>;
IdExpr : Expr ::= <ID:String>;
...
```


Implementation of the counter

```
class ASTNode {
  int countIds() {
    int count = 0;
    for (int k = 0; k < getNumChild(); k++) {
      count += getChild(k).countIds();
    }
    return count;
  }
}

class IdExp extends Expr {
  int countIds() {
    return 1;
  }
}
```


The Composite design pattern

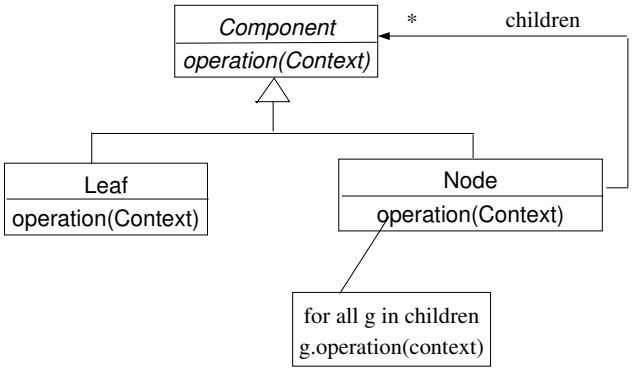
Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Example

Any recursive data type.

The Composite design pattern



Example applications

The counter

- countIds() plays the role of the operation method
- ASTNode plays the role of Component
- IdExp plays the role of Leaf
- Other classes play the role of Node. Rather than implementing count() in each of these classes, they inherit a default implementation of ASTNode.

Other typical applications

- Graphical objects that can be grouped hierarchically
- Line, Rectangle, Text, etc. play the role of Leaf
- Group plays the role of Node
- paint(Graphics) plays the role of an operation method

Modularization

How can new functionality be added to the AST classes?

- methods, instance variables, ...

Modify the AST classes?

- not modular
- risky to modify generated code
  — what happens if we need to regenerate the AST classes?

Modular techniques

- Intertype declarations (Static Aspect-Oriented Programming)
- Visitor pattern

Aspect-Oriented Programming

"Cross-cutting concerns"

- One aspect concerns many classes
- The aspect cross-cuts the normal language constructs and leads to tangled code

AOP techniques

- aspect constructs can modularize cross-cutting code, thereby avoiding tangled code
- A special kind of compiler, an aspect weaver, weaves together aspect code and ordinary code during compilation AOP systems

AOP systems

- AspectJ, http://www.eclipse.org/aspectj/, (from Xerox PARC)
- Hyper/J, http://www.research.ibm.com/hyperspace/ (IBM)
- AOSD homepage: http://aosd.net

Example of Tangled Code

Implementation of expression evaluator and unparser

```
abstract class Expr {
  abstract int value();
  abstract void unparse(Stream s, String indent);
}
abstract class BinExpr extends Expr {
  Expr getLeft() { ... }
  Expr getRight() { ... }
}
class Add extends BinExpr {
  int value() { ... }
  void unparse(Stream s, String indent) { ... }
}
class IntExpr extends Expr {
  String getINT() { ... }
  int value() { ... }
  void unparse(Stream s, String indent) { ... }
}
```


Intertype declarations (Static AOP)

Partial class definitions can be written in different modules
A preprocessor can weave together the code to complete (tangled) classes that are compiled by an ordinary compiler.

Example

```
class Add extends Expr {
  Expr expr1, expr2;
  void print() {
    expr1.print();
    System.out.print('+');
    expr2.print();
  }
}
class IntExpr extends Expr {
  int value;
  void print() {
    System.out.print(value);
  }
}
```


Extract functionality

```
class Add extends Expr {
  Expr expr1, expr2;
}
class IntExpr extends Expr {
  int value;
}
```

```
aspect Print {

  void Add.print() {
    expr1.print();
    System.out.print('+');
    expr2.print();
  }

  void IntExpr.print() {
    System.out.print(value);
  }

}
```

More functionality

```
                                  aspect Value {
class Add extends Expr {
  Expr expr1, expr2;
                                    int Add.value() {
                                      int n1 = expr1.value();
                                      int n2 = expr2.value();
                                      return n1+n2:
                                    }
}
class IntExpr extends Expr {
  int value;
                                    int IntExpr.value() {
                                      return value;
                                    }
}
                                  }
```


Weave

```
java -jar jastadd2.jar Expr.ast  Print.jadd Value.jadd
```

produces

```
class Add extends Expr {
  Expr expr1, expr2;
  void print() {
    expr1.print();
    System.out.print('+');
    expr2.print();
  }
  int value() {
    int n1 = expr1.value();
    int n2 = expr2.value();
    return n1+n2:
  }
}
```


The JastAdd system

Typed AST

- Generates AST classes with typed access methods from an abstract grammar

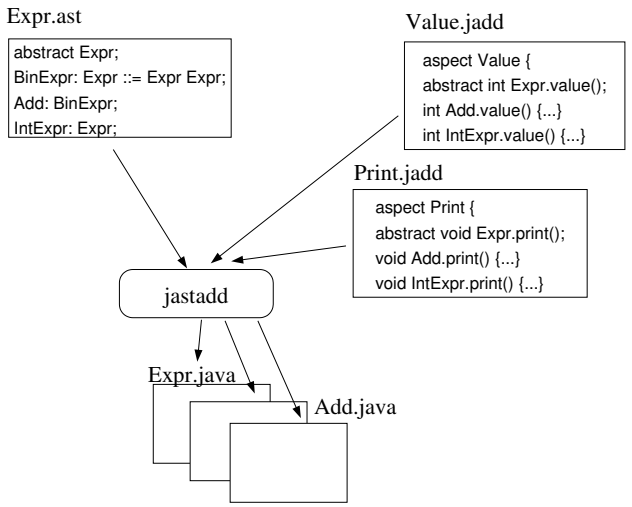
Intertype declarations

- partial class definitions for AST classes are written in .jadd files.
  JastAdd weaves in these partial definitions into the generated AST classes

Rewritable Reference Attribute Grammars

- Declarative computations using attributes and equations
- Declarative transformations of the AST
- Guest lecture by Emma Söderberg.

Modularizing our example



JastAdd aspect for the expression evaluator

```
aspect Value {
  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());
  }
}
```


What parts of AST classes can be factored out to .jadd files?

- Methods
- Instance variables
- "implements" clauses
- "import" clauses

Example (instance variable)

Add an integer representation of INT values (in addition to the String representation)

```
aspect IntValue {
  int Expr.value;
  ...
}
```


Visitors

How to modularize in Java (or any other OO language) if we do not have access to AOP mechanisms?

Example

How can we factor out the code for print()?

```
class Add extends Expr {
  Expr expr1, expr2;
  void print() {
    expr1.print();
    System.out.print('+');
    expr2.print();
  }
}
class IntExpr extends Expr {
  int value;
  void print() {
    System.out.print(value);
  }
}
```


Move functionality

```
class Add extends Expr {
  Expr expr1, expr2;
  void print() {
    expr1.print();
    System.out.print('+');
    expr2.print();
  }
}
class IntExpr extends Expr {
  int value;
  void print() {
    System.out.print(value);
  }
}
```

```
class Visitor {
  void visit(Add node) {
    node.expr1.print();
    System.out.print('+');
    node.expr2.print();
  }

  void visit(IntExpr node){
    System.out.print(
      node.value);
  }
}
```


Missing functionality

```
class Add extends Expr {
  Expr expr1, expr2;
  }
}
class IntExpr extends Expr {
  int value;
}
```

```
class Visitor {
  void visit(Add node) {
    node.expr1.print();
    System.out.print('+');
    node.expr2.print();
  }

  void visit(IntExpr node){
    System.out.print(
      node.value);
  }
}
```


Delegate

```
class Add extends Expr {
  Expr expr1, expr2;
  void accept(Visitor v) {
    v.visit(this);
  }
}
class IntExpr extends Expr {
  int value;
  void accept(Visitor v) {
    v.visit(this);
  }
}
```

```
class Visitor {
  void visit(Add node) {
    node.expr1.accept(this);
    System.out.print('+');
    node.expr2.accept(this);
  }

  void visit(IntExpr node){
    System.out.print(
      node.value);
  }
}
```

Why not call visit directly?

```
class Add extends Expr {
  Expr expr1, expr2;
  void accept(Visitor v) {
    v.visit(this);

  }
}
class IntExpr extends Expr {
  int value;
  void accept(Visitor v) {
    v.visit(this);
  }
}
```

```
class Visitor {
  void visit(Add node) {
    visit(node.expr1);
    System.out.print('+');
    visit(node.expr2);
  }

  void visit(IntExpr node){
    System.out.print(
      node.value);
  }
}
```


Generalise

```
class Add extends Expr {
  Expr expr1, expr2;
  void accept(Visitor v) {
    v.visit(this);

  }
}
class IntExpr extends Expr {
  int value;
  void accept(Visitor v) {
    v.visit(this);
  }
}
```

```
interface Visitor {
  void visit(Add node);
  void visit(IntExpr node);
}

class PrintVisitor
       implements Visitor {

  void visit(Add node) {
    node.expr1.accept(this);
    System.out.print('+');
    node.expr1.accept(this);
  } ...
}
```


Generalise with parameter and return

```
interface Visitor {
  Object visit(Add node, Object data);
  Object visit(IntExpr node, Object data);
}

class IntExpr extends Expr {
  Expr expr1, expr2;
  Object accept(Visitor v, Object data) {
    return v.visit(this, data);             }
  }
}

class PrintVisitor extends Visitor {
  Object visit(IntExpr node, Object data) {
    System.out.print(node.value);
    return null;
  }
```


Another visitor

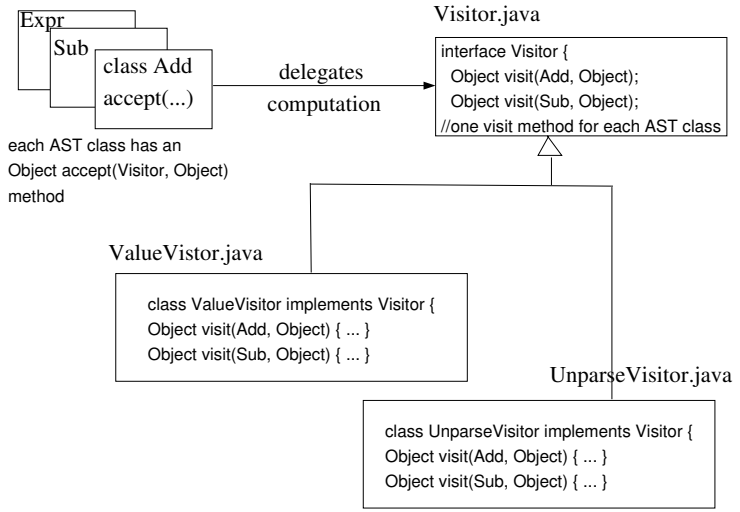
```
class ValueVisitor extends Visitor {
  Object visit(Add node, Object data) {
    int n1 = (Integer) node.expr1.accept(this, data);
    int n2 = (Integer) node.expr2.accept(this, data);
    return new Integer(n1+n2);
  }
  Object visit(IntExpr node, Object data) {
    return new Integer(node.value);
  }
}


  Expr expr = new Add(... );
  expr.accept(new PrintVisitor(), null);
  int value = expr.accept(new ValueVisitor(), null);
```

The Visitor Pattern

Intent

Represent an operation to be performed on the elements of an object structure.
Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Sketch



Interface Visitor

```
interface Visitor {
  Object visit(Add node, Object data);
  Object visit(Sub node, Object data);
  Object visit(IntExpr node, Object data);
  ...
}
```

- The visit method is overloaded for different AST argument types
- Each method returns an untyped object
- Each method has an untyped argument (data)

Visitor support in AST nodes

Method accept that delegates the computation to a Visitor

```
abstract class Expr {
  abstract Object accept(Visitor v, Object data);
}
abstract class BinExpr extends Expr {
  Object accept(Visitor v, Object data) {
    return v.visit(this, data);
  }
}
class Add extends BinExpr {
  Object accept(Visitor v, Object data) {
    return v.visit(this, data);
  }
}
class IntExpr extends Expr {
  Object accept(Visitor v, Object data) {
    return v.visit(this, data);
  }
}
```

The evaluator as a visitor

```
class Evaluator implements Visitor {









}
```


To note about this example . . .

- the argument data was not needed for the computation
- the result (of type int) had to be wrapped in an Integer object
- Casts are needed in some places.
- the method value hides implementation detail and makes it easy to call the visitor from a client

Example of use

```
Expr e = ...;
print("the value is ");
println(Evaluator.value(e));
```


The unparser as a visitor

```
class Unparser implements Visitor {









}
```


To note about this example . . .

- the argument data is used for representing the argument indent and needs to be cast to String before use
- the original argument PrintStream s is stored in the Visitor object and can be used directly by all visit methods
- the result value is not used
- the method unparse hides implementation detail and makes it easy to call the visitor from a client

Example of use

```
Program p = ...;
PrintStream s = ...;
Unparser.unparse(p, s);
```

One more example

Count the number of identifiers in a program

```
abstract Stmt;
IfStmt : Stmt ::= Cond:Expr Then:Stmt [Else:Stmt];
...
abstract Expr;
abstract BinExpr : Expr ::= Left:Expr Right:Expr;
Add : BinExpr ::= ;
Sub : BinExpr ::= ;
Int : Expr ::= <INT:String>;
IdExpr : Expr ::= <ID:String>;
...
```

How can the Visitor be implemented?

TraversingVisitor

```
class TraversingVisitor implements Visitor {
  Object visit(IfStmt node, Object data) {
    node.getCond().accept(this, data);
    node.getThen().accept(this, data);
    if (node.hasElse()) {
      node.getElse().accept(this, data);
    }
  }
  Object visit(Add node, Object data) {
    node.getLeft().accept(this, data);
    node.getRight().accept(this, data);
  }
}
```

The code above is independent of the computation and could have been generated from the abstract grammar (but this is currently not done in JJTree or JastAdd).

CountIdentifiers as a visitor

```
class CountIdentifiers extends TraversingVisitor {




}
```

Example of use

Intertype declarations vs. Visitor

	intertype declarations	Visitor
what can be modularized?	instance variables, methods, implements clauses	only methods
types for arguments and return values	arbitrary	Object visit(..., Object) (one untyped argument and one result)
separate compilation?	no – preprocessor required	yes
pure Java?	no – requires additional tools	yes

Using the modularization techniques

Interpretation
Unparsing
Metrics
Semantic Analysis

- Name analysis – connect an identifier to its declaration
- Type analysis ? compute the type of an expression
- ...

Code generation

- Compute the size needed for objects and methods
- Generate instructions
- ...

Interpreter, AST

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


Interpreter, methods

```
abstract class Expr extends ASTNode {
    abstract int value(Map<String, int> map);
}

class Add extends Expr {
    int value(Map<String, int> map) {
        return getLeft().value(map) + getRight().value(map)
    }
}
class IdExpr extends Expr {
    int value(Map<String, int> map) {
        return map.get(getID());
    }
}
```


Interpreter, methods

```
abstract class Stmt extends ASTNode {
    abstract void execute(Map<String, int> map);
}

class Block extends Stmt {
    void execute(Map<String, int> map) {
        for (int i=0, i>getNumStmt(), i++) {
            getStmt(i).execute(map);
        }
    }
}
class Assignment extends Stmt {
    void execute(Map<String, int> map) {
        int value = getExpr().value(map);
        map.put(getID(), value));
    }
}
```