Abstract Syntax & Visitors

Part IV

Repetition

- We divide the work of a compiler into several phases.
- The first two phases are:
  1. lexical analysis or scanning
  2. syntactical analysis or parsing
- These phases recognizes programs that are syntactically correct.
- The output of the syntactical analysis is usually an Abstract Syntax Tree (or AST), which is used by subsequent phases.

Using Trees

- The AST will be used by several phases of the compiler.
- How do tree processors (compiler phases) access the tree?
  - Simple solution: use **instanceof** to find the kind of a tree node and then use a cast
    `if (tree instanceof Lit) { return ((Lit) tree).num; }`
  - Better solution: object-oriented decomposition.
  - Even better solution: Visitors.

Object-oriented Decomposition

- Every tree processor `P` is represented by a dynamic method `P()` in every tree class.
- The method is abstract in class Tree, implemented in every subclass.
- To process a subtree, simply call its processor method `t.P()`.

Example: Expressions

- Two kinds of nodes: `Op`, `Lit`.
- Two kind of actions: `eval`, `print`.

```
public abstract class Tree {
  public static class Lit extends Tree {
    int num;
    public Lit(int num) {this.num = num;}
  }
 public static class Op extends Tree {
    Tree left, right; int Op;
    public OP(int Op, Tree left, Tree right) {
      this.Op = Op; this.right = right; this.left = left;
    }
  }
}
```


OO Decomposition

- Define methods `eval()` and `print()` in classes `Lit` and `Op`.
- The methods `eval()` and `print()` are abstract in `Tree`, so they can be invoked on every tree.
- What they do depend on the concrete kind of tree.

Part IV

OO Decomposition

```
public abstract class Tree {
  public abstract void print();
  public abstract int eval();
  public static class Lit extends Tree {
    int num;
    public Lit(int num) {this.num = num;}
    public void print() {
        system.out.print(" " + num);
    }
    public int eval() { return num; }
  }
…
```


Part IV

OO Decomposition

```
…
public static class Op extends Tree {
    Tree left, right; int Op;
    public OP(int Op, Tree left, Tree right) {
      this.Op= Op; this.right= right; this.left= left;}
    public void print() {
      System.out.print("(");
      left.print();
      printOp(Op);
      right.print();
      System.out.print(")");
  }
…
```


Part IV

OO Decomposition

```
…
    public int eval() {
      int l = left.eval();
      int r = right.eval();
      switch(Op) {
        case TOKENS.PLUS: return l + r;
        case TOKENS.MINUS: return l - r;
        case TOKENS.TIMES: return l * r;
        case TOKENS.DIV: return l / r;
        default: throw new InternalError();
      }
    }
  }
}
```


Part IV

Extensibility

- With an abstract syntax tree, there can be extensions in two dimensions:
  - Add a new kind of node.
  - Add a new kind of processor method.
- Which is more common?
  - New nodes ~ changes in the language.
  - New methods ~ added optimizations.
- Which is easier to do?
  - New nodes: add a new subclass.
  - New methods: add processor method to every subclass.

Part IV

Visitors

- The visitor design pattern allows simple extension by new processors.
- All methods of a processor are grouped together in a visitor object – it is easy to share common code and data.
- A visitor object contains, for each kind of node N, a method called caseN that can process trees of that kind.
- The tree contains only a simple generic processor method which applies a given visitor object.

Part IV

Example of Visitable Tree

```
public abstract class Tree {
  public abstract void apply(Visitor v);
  public static class Lit extends Tree {
    int num;
    public Lit(int num) {this.num = num;}
    public void apply(Visitor v) { v.caseLit(this); }
  }
  public static class Op extends Tree {
    Tree left, right; int Op;
    public OP(int Op, Tree left, Tree right) {
      this.Op = Op; this.right = right; this.left = left;}
    public void apply(Visitor v) { v.caseOp(this); }
  }
}
```

The Visitor

```
public interface Visitor {
  void caseLit(Lit tree);
  void caseOp(Op tree);
}
```


A Print visitor

```
public class Printer implements Visitor {
  public static void print(Tree tree) {
    tree.apply(new Printer());
  }
  public void caseLit(Tree.Lit tree) {
      System.out.print(" " + tree.num);
  }
  public void caseOP(Tree.Op tree) {
      System.out.print("(");
      print(tree.left);
      printOp(Op);
      print(tree.right);
      System.out.print(")");
  }
}
```


Coding with Visitors

- Make the tree ready for visits:
  - Write an `apply` method for each node.
  - Write an interface declaration for the visitor.
- Writing visitors:
  - Write the caseN method for each node type N.
  - Write one convenience routine, which can be called from the outside and for recursion. (in the example `print`).

An Evaluation Visitor

- The apply method has the signature `void apply(Visitor)` so we can not return the result of the evaluation to the caller of `apply`.
- Instead we keep the result in a local variable.

An Evaluation Visitor

```
public class Evaluator implements Visitor {
  int val;
  public static int eval(Tree tree) {
    Evaluator ev = new Evaluator();
    tree.apply(ev);
    return ev.val;
  }
  public void caseLit(Tree.Lit tree) {
      val = tree.num;
  }
…
```


An Evaluation Visitor

```
…
  public void caseOP(Tree.Op tree) {
    int lval = eval(tree.left)
    int rval = eval(tree.right);
    switch(tree.Op) {
      case TOKENS.PLUS:  val = lval + rval; break;
      case TOKENS.MINUS: val = lval - rval; break;
      case TOKENS.TIMES: val = lval * rval; break;
      case TOKENS.DIV:   val = lval / rval; break;
      default: throw new InternalError();
    }
  }
}
```

Which solution is better?

- Extensibility
  - OO Decomposition makes adding new kinds of nodes easy.
  - Visitors makes adding of new processors easy.
- Modularity
  - OO allows sharing of data and code in a tree node between phases.
  - Visitors allows sharing of data and code between methods of the same processor.
- Programming in a project:
  - Is one person responsible for one kind of node or is one person responsible for one tree processor?

Trees in Other Contexts

- Trees with multiple kinds of nodes arise not only in compilation.
- They are also found in text layout, structured documents such as HTML or XML, graphical user interfaces.
- Components of a GUI:
  - Which method of tree access is used for GUI components?
  - Which kind of extension is more common?

Extensibility

- Compiler
  - Operations
    - Type-check
    - Pentium back-end
    - SPARC back-end
    - optimize
    - ...
  - Kinds
    - Id
    - Lit
    - Exprs
    - Statements
- GUI
  - Operations
    - redisplay
    - move
    - iconize
    - highlight
  - Kinds
    - Scrollbar
    - Menu
    - Canavas
    - Dialogbox
    - ...

Optimization: Reusing the Visitor

- Creating a new visitor object for every invocation is expensive.
- One routine is globally available and creates a new visitor.
- Another is local and reuses the visitor.
- More efficient and allows visitor global data.

Reusing the Visitor

```
public class Printer implements Visitor {
  public static void print(Tree tree){
    tree.apply(new Printer()); }
  protected void printRec(Tree tree) { tree.apply(this);}
  public void caseLit(Tree.Lit tree) {
    System.out.print(" " + tree.num); }
  public void caseOP(Tree.Op tree) {
    System.out.print("("); printRec(tree.left);
                           printOp(Op);
                           printRec(tree.right);
    System.out.print(")");
  }
}
```


Visitor Global Data

```
public class Printer implements Visitor {
  PrintStream p;
  public Printer(PrintStream p) {this.p = p;}

  public static void print(Tree tree, PrintStream p) {
    tree.apply(new Printer(p)); }
  protected void printRec(Tree tree){ tree.apply(this);}

  public void caseLit(Tree.Lit tree){p.print(" "+tree.num);}
  public void caseOP(Tree.Op tree) {
    p.print("("); printRec(tree.left);
                  printOp(Op,p);
                  printRec(tree.right); p.print(")");
  }
}
```

Summary

- In an OO setting there are basically two ways of encoding operations on a tree:
  - Object-oriented Decomposition.
  - Visitors.
- For compilers visitors are the better choice.

Semantic Analysis

Part V

Semantic Analysis

- We have used a context-free grammar for parsing.
- Programming languages are not really context-free, we can define new names.

Programming Languages are not really context-free

- In many languages the user can declare variables, and types.
- Whether a variable is declared then depends on the *context*.
- Example:

```
{
 int x;
 x = i + 1;
}
```


Programming languages are not really context-free

- We solve this problem in two steps:
  - We define a context-free superset of the language which can be parsed easily.
  - We write a special purpose context-sensitive analyzer that weeds out illegal programs.

Type- and Name Analysis

- The purpose of *type analysis* is to find out the type of expressions and variables, by finding out which variable declaration a certain variable name refers to.
- The *name analysis* finds out if there are uses of identifiers which are not defined, or if there are illegal double definitions.
- Some languages allows *type-inference* where the type of an identifier can be inferred from its use, without any declaration.

Part IV

Scope

- Every identifier in a program has a *scope*, i.e., an area of program text, in which it can be referred to.
- Each language has its own *scoping-rules* that defines the scope of identifiers.

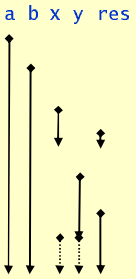
Part IV

Scope in Expressive

- Expressive has only one type of identifiers – variables (no functions, modules, or user defined types).
- The scoping rules are quite simple: the scope of a variable extends (lexically) from its first definition to the end of the program.
- Except in an if-statement where a variable defined in the 'true'-branch does not extend into the 'else'-branch.

Part IV

Example of Scopes




```
readint a.
readint b.
if a < b {
  x:= a.
  res:= x * x.
} else {
  y:= b.
  res:= y * y.
}.
print res.
```

Coding with Visitors

- Make the tree ready for visits:
  - Write an `apply` method for each node.
  - Write an interface declaration for the visitor.
- Writing visitors:
  - Write the caseN method for each node type N.
  - Write one convenience routine, which can be called from the outside and for recursion. (in the example `print`).

An Evaluation Visitor

```
public class Evaluator implements Visitor {
  int val;
  public static int eval(Tree tree) {
    Evaluator ev = new Evaluator();
    tree.apply(ev);
    return ev.val;
  }
```

Extensibility

- Compiler
  - Operations
    - Type-check
    - Pentium back-end
    - SPARC back-end
    - optimize
    - …
  - Kinds
    - Id
    - Lit
    - Exprs
    - Statements
- GUI
  - Operations
    - redisplay
    - move
    - iconize
    - highlight
  - Kinds
    - Scrollbar
    - Menu
    - Canavas
    - Dialogbox
    - …

Reusing the Visitor

```
public class Printer implements Visitor {
  public static void print(Tree tree){
    tree.apply(new Printer()); }
  protected void printRec(Tree tree) { tree.apply(this);}
  public void caseLit(Tree.Lit tree) {
    System.out.print(" " + tree.num); }
  public void caseOP(Tree.Op tree) {
    System.out.print("("); printRec(tree.left);
```

Using Trees

- The AST will be used by later phases of the compiler.
- How do tree processors (compiler phases) access the tree?
  - Simple solution: use **instanceof** to find the kind of a tree node and cast
    ```
    if (tree instanceof Lit) { return ((Lit) tree).num;}
    ```
  - Better solution: object-oriented decomposition.
  - Even better solution: Visitors.

Example: Expressions

- Two kinds of nodes: `Op`, `Lit`.
- Two kind of actions: `eval`, `print`.

```
public abstract class Tree {
  public static class Lit extends Tree {
    int num;
    public Lit(int num) {this.num = num;}
  }
 public static class Op extends Tree {
```

OO Decomposition

```
…
    public int eval() {
      int l = left.eval();
      int r = right.eval();
       switch(Op) {
         case TOKENS.PLUS: return l + r;
         case TOKENS.MINUS: return l - r;
         case TOKENS.TIMES: return l * r;
         case TOKENS.DIV: return l / r;
         default: throw new InternalError();
       }
    }
  }
}
```


Visitors

- The visitor design pattern allows simple extension by new processors.
- All methods of a processor are grouped together in a visitor object – it is easy to share common code and data.
- A visitor object contains, for each kind of node N,

Coding with Visitors

- Make the tree ready for visits:
  - Write an `apply` method for each node.
  - Write an interface declaration for the visitor.
- Writing visitors:
  - Write the caseN method for each node type N.
  - Write one convenience routine, which can be called from the outside and for recursion. (in the example `print`).

An Evaluation Visitor

```
public class Evaluator implements Visitor {
  int val;
  public static int eval(Tree tree) {
    Evaluator ev = new Evaluator();
    tree.apply(ev);
    return ev.val;
  }
```

Extensibility

- Compiler
  - Operations
    - Type-check
    - Pentium back-end
    - SPARC back-end
    - optimize
    - …
  - Kinds
    - Id
    - Lit
    - Exprs
    - Statements
- GUI
  - Operations
    - redisplay
    - move
    - iconize
    - highlight
  - Kinds
    - Scrollbar
    - Menu
    - Canavas
    - Dialogbox
    - …

Reusing the Visitor

```
public class Printer implements Visitor {
  public static void print(Tree tree){
    tree.apply(new Printer()); }
  protected void printRec(Tree tree) { tree.apply(this);}
  public void caseLit(Tree.Lit tree) {
    System.out.print(" " + tree.num); }
  public void caseOP(Tree.Op tree) {
    System.out.print("("); printRec(tree.left);
```