1 Part V: Abstract Syntax

- Abstract Syntax
- Abstract Syntax Trees
- Constructing Trees in the Parser
- Using Trees
  - Object Oriented Decomposition
  - Visitors

2 Syntax Trees

- In a multi - pass compiler the parser builds a syntax tree explicitely.
- All later phases of a compiler work on the *abstract syntax tree*, not the program source.
- The tree could be the concrete syntax tree (parse tree) corresponding to the context-free grammar.
- Usually, there is a better choice.

3 Abstract Syntax / Concrete Syntax

Compared to the concrete syntax tree, some simplifications are possible:

- No need for parentheses: $\mathsf{A} * (\mathsf{B} + \mathsf{C})$ becomes ...

- No need to maintain terminals **if** (x == 0) y = 1; **else** y = 2; becomes ...

4 Abstract Syntax Tree

- An abstract syntax tree is a tree with one kind of node for each alternative in the abstract syntax.
- It is simpler than a parse tree and therefore easier to use.
- It has all necessary information.

- We represent a tree using a set of Java classes, one for each alternative.
- Common abstract superclass: Tree.
- Each class represents subtrees as instance variables.
- Each class has a constructor to construct a node of the given kind.

5 Abstract Syntax Tree (2)

Abstract Syntax of Expressions

```
Expression = OPERATION Expression Expression Op
           | NUMLIT int
           ;
```

Parenthesis are not necessary in abstract syntax!


```
public abstract class Tree {
    public static class NumLit extends Tree {
        int num;
        public NumLit(int num) {
            this.num = num;
        }
    }
    public static class Operation extends Tree {
        Tree left, right;
        char op;
        public Operation(Tree left, Tree right, char op) {
            this.left = left;
            this.right = right;
            this.op = op;
        }
    }
}
```


6 Abstract Syntax Tree (3)

Repetition in the abstract syntax

```
Statement = STATEMENTLIST { Statement }
          | ...
          ;
```

is typically implemented by arrays.

```
public static class StatementList extends Tree {
    Tree[ ] stats;
    public StatementList(Tree[ ] stats) {
        this.stats = stats;
    }
}
```


7 Constructing Trees in the Parser

Concrete Syntax

```
E ::= T { "+" T }
T ::= NUMLIT
```

Top-Down Parser

```
void E() {
    T();
    while (token == PLUS) {
        token == nextToken();
        T();
    }
}
```

```
E ::= T { "+" T }
T ::= NUMLIT

void T() {
    if (token == NUMLIT) {
        token = nextToken();
    } else {
        error();
    }
}
```


8 Constructing Trees(2)

```
Tree E() {
    Tree t = T();
    while (token == PLUS) {
        int op = '+';
        token == nextToken();
        t = new Operation(t, T(), op);
    }
}
```



```
Tree T() {
    if (token == NUMLIT) {
        int i = INTEGER.parseInt(tokenChars);
        token = nextToken();
        return new NumLit(i);
    } else {
        error();
        return null;
    }
}
```


9 Using Trees

- The abstract syntax tree is the central data structure of later phases of the compiler.
- It is important to find a representation, which can be used in flexible ways.
- How do tree processors (compiler passes) access the tree?
- Simple (and crude) solution: use **instanceof** to find out the kind of the tree node and then cast to access tree elements.

```
if (tree instanceof NumLit) {
    return ((NumLit) tree).num;
}
```

- This is neither elegant nor efficient.
- Better solution: object-oriented decomposition
- Even better solution: Visitors.

10 Example: Expressions

- We now present both object-oriented decomposition and visitor access, using arithemetic expressions as an example.
- Two kind of nodes: Operation, NumLit
- Two kind of actions: eval, print
- Very simple example.
- Typical languages have 20 (Jex) - 40 (Java) or more kinds of nodes.
- A typical compiler has 5-10 processors.
- But the basic framework stays the same.

11 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().
- In our example: define methods eval() and print() in classes NumLit and Operation
- The methods eval() and print() are abstract in Tree, so they can be invoked on every tree.
- What they do will depend on the concrete kind of tree.

12 Object-oriented Decomposition

```
public abstract class Tree {
    public abstract void print();
    public abstract int eval();
    public static class NumLit extends Tree {
        int num;
        public NumLit(int num) { ... }
        public void print() {
            System.out.print("" + num);
        }
        public int eval() {
            return num;
        }
    }
```



```
    public static class Operation extends Tree {
        Tree left, right;
        char op;
        public Operation(Tree left,
                    Tree right, char op) { ... }
        public void print() {
            System.out.print("(");
            left.print();
            System.out.print(" " + op + " ");
            right.print();
            System.out.print(")");
        }
```

```
        public int eval() {
            int l = left.eval();
            int r = right.eval();
            switch(op) {
                case '+':
                    return l + r;
                case '-':
                    return l - r;
                case '*':
                    return l * r;
                case '/':
                    return l / r;
                default:
                    throw new InternalError();
            }
        }
    }
}
```


13 A Driver Class

```
class EvalTest {
    public static void main(String args[ ]) throws Exception {
        ...
        Tree tree = parser.parse();
        tree.print();
        System.out.println(" = " + tree.eval());
    }
}
```

```
java expression.EvalTest
(3 * (2 - 5)) = -9
```


14 A Typical Stack Trace

```
(#*).print()
(#-).print()
(#2).print()
System.out.print(2)
```

```
(3 * (
```


15 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 one is more common?
- Which one is easier to do?
- Add a new kind of node: add a new subclass.
- Add a new kind of processor method: add processor method to every subclass.

16 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 K of trees a method called caseK that can process trees of that kind.
- The tree contains only a simple generic processor method which applies a given visitor object.

17 Visitable Trees for Expressions

```
public abstract class Tree {
    public abstract void apply(Visitor v);

    public static class NumLit extends Tree {
        int num;
        public NumLit(int num) { ... }
        public void apply(Visitor v) {
            v.caseNumLit(this);
        }
    }
```



```
    public static class Operation extends Tree {
        Tree left, right;
        char op;
        public Operation(Tree left, Tree right, char op) { ... }
        public void apply(Visitor v) {
            v.caseOperation(this);
        }
    }

    public interface Visitor {
        void caseOperation(Operation tree);
        void caseNumLit(NumLit tree);
    }
}
```


18 A Print Visitor

```
public class Printer implements Tree.Visitor {
    public static void print(Tree tree) {
        tree.apply(new Printer());
    }
    public void caseOperation(Tree.Operation tree) {
        System.out.print("(");
        print(tree.left);
        System.out.print(" " + tree.op + " ");
        print(tree.right);
        System.out.print(")");
    }
    public void caseNumLit(Tree.NumLit tree) {
        System.out.print("" + tree.num);
    }
}
```

19 A Typical Stack Trace

```
Printer.print(#*)
(#*).apply(new Printer())
new Printer().caseOperation(#*)
Printer.print(#–)
(#–).apply(new Printer())
new Printer().caseOperation(#–)
Printer.print(#2)
(#2).apply(new Printer())
new Printer().caseNumLit(#2)
System.out.print(2)
```

Each recursive call is implemented by three nested calls:

```
Printer.print(tree) → (#–).apply(new Printer())
                            → new Printer().caseOperation(#–)
```


20 Coding with Visitors

Make the tree good for visiting:

- Write an apply method for each node.
- Write an interface declaration for the tree visitors

Writing individual visitors:

- Write the caseXxx method for each node type xxx.
- Write one convenience routine (in the example print), which can be called from outside and for recursion.

21 An Evaluation Visitor

- Because we have only one general apply method, we have to pass the result differently.
- We keep it in a local instance variable val, that eval reads after apply finished.

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



```
    public void caseOperation(Tree.Operation tree) {
        switch (tree.op) {
            case '+':
                val = eval(tree.left) + eval(tree.right);
                break;
            case '–':
                val = eval(tree.left) – eval(tree.right);
                break;
            case '*':
                val = eval(tree.left) * eval(tree.right);
                break;
            case '/'::
                val = eval(tree.left) / eval(tree.right);
                break;
            default: throw new InternalError();
        }
    }
}
```

22 Driver Class for Visitors

```
class EvalTest {
    public static void main(String args[ ]) throws Exception {
        ...
        Tree tree = parser.parse();
        Printer.print(tree);
        System.out.println(" = " + Evaluator.eval(tree));
    }
}
```


23 Which one is better?

- Extensibility
  - OO Decomposition makes adding new kinds of nodes easy.
  - Visitors make adding of new processors easy.
- Modularity
  - OO allows sharing of data and code in a tree node between phases.
  - Visitors allow sharing of data and code between methods of same processor.
- Which is more important?
- Programming in a group
  - Is one person responsible for one kind of node?
  - Is one person responsible for one tree processor?
- SUN switched for the new Java compiler also because the old one was written object oriented.

24 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?

25 Extensibility

Compiler

- Operations
  - type-check
  - translate to Pentium
  - translate to SPARC
  - optimize
  - find uninitialized vars
- Kinds
  - Ident
  - Numeric literal
  - String literal
  - If statement

GUI

- Operations
  - redisplay
  - move
  - iconize
  - highlight
- Kinds
  - Scrollbar
  - Menu
  - Canvas
  - Dialogbox
  - Statusbar

26 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 routine is local and reuses the visitor.

- More efficient
- Allows visitor - global data

27 Optimization: Reusing the Visitor

```
public class Printer implements Tree.Visitor {
    public static void print(Tree tree) { tree.apply(new Printer()); }
    protected void printRec(Tree tree) { tree.apply(this); }
    public void caseOperation(Tree.Operation tree) {
        System.out.print("(");
        printRec(tree.left);
        System.out.print(" " + tree.op + " ");
        printRec(tree.right);
        System.out.print(")");
    }
    public void caseNumLit(Tree.NumLit tree) { ... }
}
```


28 Visitor Global Data

```
public class Printer implements Tree.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 caseOperation(Tree.Operation tree) {
        p.print("(");
        printRec(tree.left);
        p.print(" " + tree.op + " ");
        printRec(tree.right);
        p.print(")");
    }
    public void caseNumLit(Tree.NumLit tree) { ... }
}
```


29 Summary

- We use an abstract syntax to define the internal data structure of the compiler (the abstract syntax tree)
- Because it serves a different purpose, it is usually a good idea to choose it different from the concrete syntax.
- There are two ways of encoding the operations on the tree
  - object-oriented
  - visitors
- For compilers visitors are the better choice.