Chapter 5: JavaCC and JTB
The Java Compiler Compiler

- Can be thought of as “Lex and Yacc for Java.”

- It is based on LL(k) rather than LALR(1).

- Grammars are written in EBNF.

- The Java Compiler Compiler transforms an EBNF grammar into an LL(k) parser.

- The JavaCC grammar can have embedded action code written in Java, just like a Yacc grammar can have embedded action code written in C.

- The lookahead can be changed by writing `LOOKAHEAD(...)`.

- The whole input is given in just one file (not two).
The JavaCC input format

One file:

- header
- token specifications for lexical analysis
- grammar
Example of a token specification:

```java
TOKEN :
{
    < INTEGER_LITERAL: ( ["1"-"9"] ("0"-"9") )* | "0" ) >
}
```

Example of a production:

```java
void StatementListReturn() :
{
    {}
    {
        ( Statement() )* "return" Expression() ";"
    }
```
Generating a parser with JavaCC

javacc fortran.jj // generates a parser with a specified name
javac Main.java // Main.java contains a call of the parser
java Main < prog.f // parses the program prog.f
The Visitor Pattern

For object-oriented programming,

the Visitor pattern enables

the definition of a new operation

on an object structure

without changing the classes

of the objects.

Gamma, Helm, Johnson, Vlissides:
Design Patterns, 1995.
When using the **Visitor** pattern,

- the set of classes must be fixed in advance, and

- each class must have an accept method.
First Approach: Instanceof and Type Casts

The running Java example: summing an integer list.

```java
interface List {

    class Nil implements List {

        class Cons implements List {
            int head;
            List tail;
        }
    }
}
```
First Approach: Instanceof and Type Casts

List l;  // The List-object
int sum = 0;
boolean proceed = true;
while (proceed) {
    if (l instanceof Nil)
        proceed = false;
    else if (l instanceof Cons) {
        sum = sum + ((Cons) l).head;
        l = ((Cons) l).tail;
        // Notice the two type casts!
    }
}

**Advantage:** The code is written without touching the classes Nil and Cons.

**Drawback:** The code constantly uses type casts and instanceof to determine what class of object it is considering.
Second Approach: Dedicated Methods

The first approach is **not** object-oriented!

To access parts of an object, the classical approach is to use dedicated methods which both access and act on the subobjects.

```java
interface List {
    int sum();
}
```

We can now compute the sum of all components of a given `List-object l` by writing `l.sum()`.
Second Approach: Dedicated Methods

class Nil implements List {
    public int sum() {
        return 0;
    }
}

class Cons implements List {
    int head;
    List tail;
    public int sum() {
        return head + tail.sum();
    }
}

**Advantage:** The type casts and `instanceof` operations have disappeared, and the code can be written in a systematic way.

**Disadvantage:** For each new operation on `List`-objects, new dedicated methods have to be written, and all classes must be recompiled.
Third Approach: The Visitor Pattern

The Idea:

- Divide the code into an object structure and a Visitor (akin to Functional Programming!)
- Insert an `accept` method in each class. Each accept method takes a Visitor as argument.
- A Visitor contains a `visit` method for each class (overloading!) A method for a class $C$ takes an argument of type $C$.

```java
interface List {
    void accept(Visitor v);
}

interface Visitor {
    void visit(Nil x);
    void visit(Cons x);
}
```
Third Approach: The Visitor Pattern

- The purpose of the `accept` methods is to invoke the `visit` method in the Visitor which can handle the current object.

```java
class Nil implements List {
    public void accept(Visitor v) {
        v.visit(this);
    }
}

class Cons implements List {
    int head;
    List tail;
    public void accept(Visitor v) {
        v.visit(this);
    }
}
```
Third Approach: The Visitor Pattern

- The control flow goes back and forth between the visit methods in the Visitor and the accept methods in the object structure.

```java
class SumVisitor implements Visitor {
    int sum = 0;
    public void visit(Nil x) {}
    public void visit(Cons x) {
        sum = sum + x.head;
        x.tail.accept(this);  // Visit(x.tail)
    }
}

......
SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
```

Notice: The visit methods describe both 1) actions, and 2) access of subobjects.
Comparison

The Visitor pattern combines the advantages of the two other approaches.

<table>
<thead>
<tr>
<th></th>
<th>Frequent type casts?</th>
<th>Frequent recompilation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instanceof and type casts</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Dedicated methods</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>The Visitor pattern</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The advantage of Visitors: New methods without recompilation!

Requirement for using Visitors: All classes must have an accept method.

Tools that use the Visitor pattern:

- JJTree (from Sun Microsystems) and the Java Tree Builder (from Purdue University), both frontends for The Java Compiler Compiler from Sun Microsystems.
Visitors: Summary

- **Visitor makes adding new operations easy.** Simply write a new visitor.

- **A visitor gathers related operations.** It also separates unrelated ones.

- **Adding new classes to the object structure is hard.** Key consideration: are you most likely to change the algorithm applied over an object structure, or are you most likely to change the classes of objects that make up the structure.

- **Visitors can accumulate state.**

- **Visitor can break encapsulation.** Visitor’s approach assumes that the interface of the data structure classes is powerful enough to let visitors do their job. As a result, the pattern often forces you to provide public operations that access internal state, which may compromise its encapsulation.
The Java Tree Builder

JTB is a frontend for The Java Compiler Compiler.

JTB supports the building of syntax trees which can be traversed using visitors.

JTB transforms a bare JavaCC grammar into three components:

- a JavaCC grammar with embedded Java code for building a syntax tree;
- one class for every form of syntax tree node; and
- a default visitor which can do a depth-first traversal of a syntax tree.
The Java Tree Builder

The produced JavaCC grammar can then be processed by the Java Compiler Compiler to give a parser which produces syntax trees.

The produced syntax trees can now be traversed by a Java program by writing subclasses of the default visitor.
Using JTB

jtb fortran.jj    // generates jtb.out.jj
javacc jtb.out.jj // generates a parser with a specified name
javac Main.java   // Main.java contains a call of the parser
                  // and calls to visitors
java Main < prog.f // builds a syntax tree for prog.f, and
                  // executes the visitors
Example (simplified)

For example, consider the Java 1.1 production

```java
void Assignment() : {}
    { PrimaryExpression() AssignmentOperator()
      Expression() }
```

JTB produces:

```java
Assignment Assignment () :
{ PrimaryExpression n0;
  AssignmentOperator n1;
  Expression n2; {}
}
{n0=PrimaryExpression()
 n1=AssignmentOperator()
 n2=Expression()
  { return new Assignment(n0,n1,n2); }
}
```

Notice that the production returns a syntax tree represented as an Assignment object.
Example (simplified)

JTB produces a syntax-tree-node class for Assignment:

```java
public class Assignment implements Node {
    PrimaryExpression f0; AssignmentOperator f1;
    Expression f2;

    public Assignment(PrimaryExpression n0,
                       AssignmentOperator n1,
                       Expression n2)
    { f0 = n0; f1 = n1; f2 = n2; }

    public void accept(visitor.Visitor v) {
        v.visit(this);
    }
}
```

Notice the accept method; it invokes the method visit for Assignment in the default visitor.
Example (simplified)

The default visitor looks like this:

```java
public class DepthFirstVisitor implements Visitor {
    ...
    //
    // f0 -> PrimaryExpression()
    // f1 -> AssignmentOperator()
    // f2 -> Expression()
    //
    public void visit(Assignment n) {
        n.f0.accept(this);
        n.f1.accept(this);
        n.f2.accept(this);
    }
}
```

Notice the body of the method which visits each of the three subtrees of the Assignment node.
Example (simplified)

Here is an example of a program which operates on syntax trees for Java 1.1 programs. The program prints the right-hand side of every assignment. The entire program is six lines:

```java
public class VprintAssignRHS extends DepthFirstVisitor {
    void visit(Assignment n) {
        VPrettyPrinter v = new VPrettyPrinter();
        n.f2.accept(v); v.out.println();
        n.f2.accept(this);
    }
}
```

When this visitor is passed to the root of the syntax tree, the depth-first traversal will begin, and when Assignment nodes are reached, the method `visit` in `VprintAssignRHS` is executed.

Notice the use of `VPrettyPrinter`. It is a visitor which pretty prints Java 1.1 programs.

JTB is bootstrapped.