

## CS6013 - Modern Compilers: Theory and Practise

## Introduction to Tools

V. Krishna Nandivada

IIT Madras

## 1 Introduction to Tools

- JavaCC
- Visitor Pattern
- Java Tree Builder



## The Java Compiler Compiler (JavaCC)

- 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).



## JavaCC input

## One file

- header
- token specification for lexical analysis
- grammar

## Example of a token specification:

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

## Example of a production:

```
void StatementListReturn() :
{
{
  ( Statement() )* "return" Expression() ";"
}
```



# Generating a parser with JavaCC

```
javacc fortran.jj // generates a parser with a specified name  
  
// Sample Main.java  
public class Main {  
    public static void main(String [] args) {  
        try {  
            new FortranParser(System.in).Goal();  
            System.out.println("Program parsed successfully");  
        }  
        catch (ParseException e) {  
            System.out.println(e.toString());  
        }  
    }  
}
```

```
javac Main.java // Main.java contains a call of the parser  
java Main < prog.f // parses the program prog.f
```



# Outline

- 1 Introduction to Tools
  - JavaCC
  - **Visitor Pattern**
  - Java Tree Builder



# The Visitor Pattern

- The visitor design pattern is a way of separating an algorithm from an object structure on which it operates.
- Implication: the ability to add new operations to existing object structures without modifying those structures.
- Interesting in object oriented programming and software engineering.

## Requirements

- The set of classes must be fixed in advance, and
- each class must have an `accept` method.



# Motivate Visitor by summing an integer list

```
interface List {}  
  
class Nil implements List {}  
  
class Cons implements List {  
    int head;  
    List tail;  
}
```



## 1/3 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!
    }
}
```

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

**Drawback:** The code constantly uses explicit type cast and `instanceof` operations.



## 2/3 approach: dedicated methods

- The first approach is NOT object-oriented!
- Classical method to access parts of an object: dedicated methods which both access and act on the subobjects.

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

- We can now compute the sum of all components of a given List-object `ll` by writing `ll.sum()`.



## 2/3 approach: dedicated methods (contd)

```
class Nil implements List {
    public int sum() {
        return 0;
    }
}
class Cons implements List {
    int head;
    List tail;
    public int sum() {
        return head + tail.sum();
    }
}
```

- **Adv:** The type casts and `instanceof` operations have disappeared, and the code can be written in a systematic way.
- **Drawback:** For each new operation, new dedicated methods have to be written, and all classes must be recompiled.



## 3/3 approach: Visitor pattern

### The Idea:

- Divide the code into an object structure and a Visitor.
- 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 `visit` method for a class `C` takes an argument of type `C`.

```
interface List {
    void accept(Visitor v);
}
interface Visitor {
    void visit(Nil x);
    void visit(Cons x);
}
```



## 3/3 approach: Visitor pattern

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

```
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);
    }
}
```



## 3/3 approach: Visitor pattern

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

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

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



## 3/3 approach: Visitor pattern control flow

```
interface List {
    void accept(Visitor v);
}
interface Visitor {
    void visit(Nil x);
    void visit(Cons x);
}
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);
    }
}
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);
    }
}
.....
SumVisitor sv = new SumVisitor();
l.accept(sv);
```



## Comparison

#	detail	Frequent type casts	Frequent recompilation
1.	Instanceof + type-cast	Yes	No
2.	Dedicated methods	No	Yes
3.	Visitor pattern	No	No

- The Visitor pattern combines the advantages of the two other approaches.
- **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), the Java Tree Builder (from Purdue University), both frontends for The JavaCC from Sun Microsystems.
- ANTLR generates default visitors for its parse trees.



## 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 like 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.



## Outline

- 1 Introduction to Tools
  - JavaCC
  - Visitor Pattern
  - Java Tree Builder



## Fun Assignment 1

- Write the three versions of code corresponding to each of the above discussed approaches.
- Populate the lists with 'N' number of elements.
- Print the Sum of elements.
- Convince yourself about the programmability with Visitor pattern.
- See which of the three approaches is more efficient?
- Vary 'N' - 10; 100; 1000; 100,000; 10,00,000.
- Make a table and report the numbers.
- Write a paragraph or two reasoning about the performance.
- Mention any thoughts on performance improvement.

The best answer(s) will be [recognized](#).



## Java Tree builder

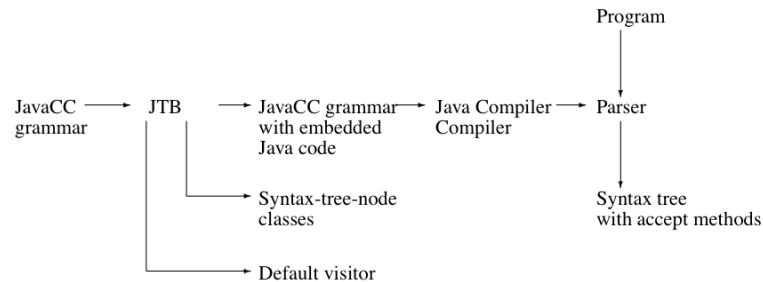
- The Java Tree Builder (JTB) has been developed at Purdue (my ex group).
- JTB is a frontend for The Java Compiler Compiler.
- JTB supports the building of syntax trees which can be traversed using visitors. Q: Why is it interesting?
- JTB transforms a bare JavaCC grammar into three components:
  - 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.



## Invoking JTB

```
jtb fortran.jj //generates jtb.out.jj
javacc jtb.out.jj //generates a parser with a specified name
// Sample Main.java:
public class Main {
    public static void main(String [] args) {
        try {
            Node root = new FortranParser(System.in).Goal();
            System.out.println("Program parsed successfully");
            root.accept(new GJNoArguDepthFirst());
        }
        catch (ParseException e) {
            System.out.println(e.toString());
        }
    }
}
```

```
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
```



## (simplified) Example

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

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

    ...

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

Notice the `accept` method; it invokes the method `visit` for `Assignment` in the default visitor.



## (simplified) Example

The default visitor looks like this:

```
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.



What have we do today?

- JavaCC
- Visitor pattern
- JTB

Reading/ToDo:

- Visitor pattern (from the Design patterns book)
- Download and play with JTB, JavaCC

