COM SCI 132 Week 2

Aidan Jan

April 8, 2024

The Java Compiler Compiler (JCC)

- 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 embeeded action code written in C.
- The lookahead can be changed by writing LOOKAHEAD(...).
- The whole input is given in just one file (not two).

Input format: one file containing a header, token specifications for lexical analysis, and 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
javac Main.java // Main.java contains a call of the parser
java Main < prog.f // parses the program prog.f
```

The Visitor Pattern

According to Gamma, Helm, Johnson, Vlissides: **Design Patterns**, 1995:

> 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.

First Approach: Instanceof and Type Casts

The running Java example: summing an integer list.

```
interface List {}

class Nil implements List {}

class Cons immplements List {
    int head;
    List tail;
}

List l;
```

Suppose we have a `List l` in the code to parse. Is `l` a `Nil` or a `Cons`?
One way to check is to do type casting:

```
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 += ((Cons) l).head;  // Notice these two type casts!
        l = ((Cons) l).tail;
    }
}
```

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 classcal approach is to use dedicated methods which both access and act on the subobjects.
Instead, of writing a `sum()` method that operates on list types, just simply add a .sum() method to the `List` interface. All classes that implement the interface can then override the method.

```
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.
Drawback: For each new operation on `List`-objects, write new dedicated methods and recompile all classes.

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 an argument.
- A Visitor contains a `visit` method for each class (overloading!) A 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);
}
```

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);
    }
}
```

Notice: the `accept` method in both classes is the same! Due to the way the Visitor is implemented, the method always calls `visit`. This is beneficial because it allows the compiler to easily parse classes by appending the same code to each class. In fact, the `accept` method is always implemented the way it is shown above, for any class!

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 += x.head;
        x.tail.accept(this);
    }
}
.....

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

Also notice: since the `accept` method is titled `accept` in every class, the `visit` method in the Visitor can just call the `accept` method, and it automatically calls the method for the correct class!

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 sturcture classes is powerful enough tot let visitors do their job. As a result, the patten often forces you to provide public operations that access internal state, which may compromise its encapsulation.

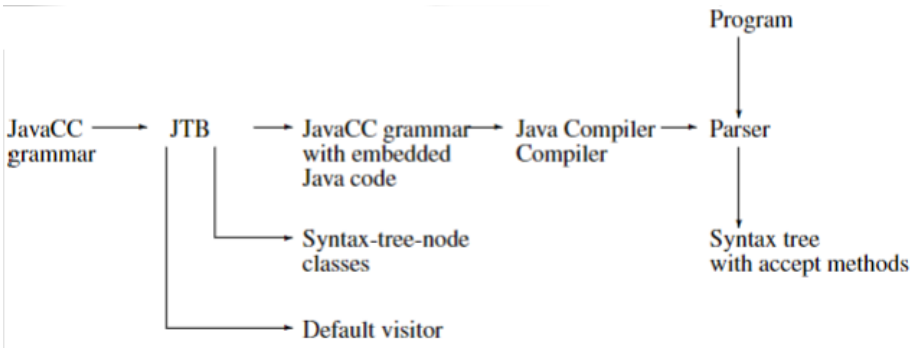
The Java Tree Builder

The Java Tree Builder (JTB) is a front for The Java Compiler Compiler. It supports the building of syntax trees which can be traversed using visitors.

JTB transforms a bare JavaCC grammar into three components:

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

The produced JavaCC grammar cam then be processed by the JCC 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.



Example (simplified)

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

```
public class Assignment implements Npde {
    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.

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.