04830180-编译实习

03. Vistor, Tools, and an Example

黄　骏

jun.huang@pku.edu.cn

高能效计算与应用中心

北京大学信息科学与技术学院

理科五号楼515S

Visitor

- In object-oriented programming, the **visitor** pattern is a way of *separating an algorithm from an object structure* on which it operates.
- Goal: it should be possible to define a new operation for (some) classes of an object structure without changing the classes.

When new operations are needed frequently, it's inflexible to add new subclasses or methods each time a new operation is required

Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Visitor

- In object-oriented programming, the **visitor** pattern is a way of *separating an algorithm from an object structure* on which it operates.
- Goal: it should be possible to define a new operation for (some) classes of an object structure without changing the classes.

When new operations are needed frequently, it's inflexible to add new subclasses or methods each time a new operation is required

Visitor lets you define a new operation without changing the classes of the elements on which it operates.

- *分离语法树的定义和对语法树的操作*

An Example: Summing an Integer List

```
interface List {}

class Nil implements List {}

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

Approach-1: instanceOf and Type Casts

```
List list;
int sum = 0;
boolean proceed = true;
while (proceed) {
        if (list instanceof Nil)
                proceed = false;
        else if (list instanceof Cons) {
                sum = sum + ((Cons) list).head;
                list = ((Cons) list).tail;
        }
}
```

Approach-1: instanceOf and Type Casts

```
List list;
int sum = 0;
boolean proceed = true;
while (proceed) {
        if (list instanceof Nil)
                proceed = false;
        else if (list instanceof Cons) {
                sum = sum + ((Cons) list).head;
                list = ((Cons) list).tail;
        }
}
```

- 优点：不需要改变Nil和Cons类的结构
- 缺点：繁琐的类型转换

Approach-2: Dedicated Methods

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

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

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

Approach-2: Dedicated Methods

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

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

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

- 优点：避免繁琐的类型转换
- 缺点：每新加一个对List-Object的操作，需要在Nil和Cons类中添加代码并重新编译。

Approach-3: The Visitor Pattern

- 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. A method for a class C takes an argument of type C

Approach-3: The Visitor Pattern

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

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

Approach-3: The Visitor Pattern

```
class Nil impelments List{
        public void accept(Visitor v) {
                v.visit(this);
        }
}
class Cons impelments List{
        int head; List tail;
        public void accept(Visitor v) {
                v.visit(this);
        }
}
```

Approach-3: The Visitor Pattern

```
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();
list.accept(sv);
```

- 避免繁琐的类型转换
- 将对Object的操作与类结构相分离
- 避免引入新操作时对类的改变和重编译

Outline

- Visitor Pattern
- JTB and JavaCC
- An Example

JavaCC

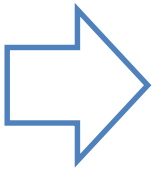
- JavaCC: Java Compiler Compiler
- ***JavaCC*** **是一个能自动生成词法分析器和语法分析器的程序**
  - 来源：http://javacc.dev.java.net
  - **java –cp [javacc.jar路径] javacc minijava.jj**

输入　一个 JavaCC 语法文件 *.jj

输出　一个包含词法分析器和语法分析器的程序

（由多个 Java文件组成）

```
int main() {
        return 0;
}
```

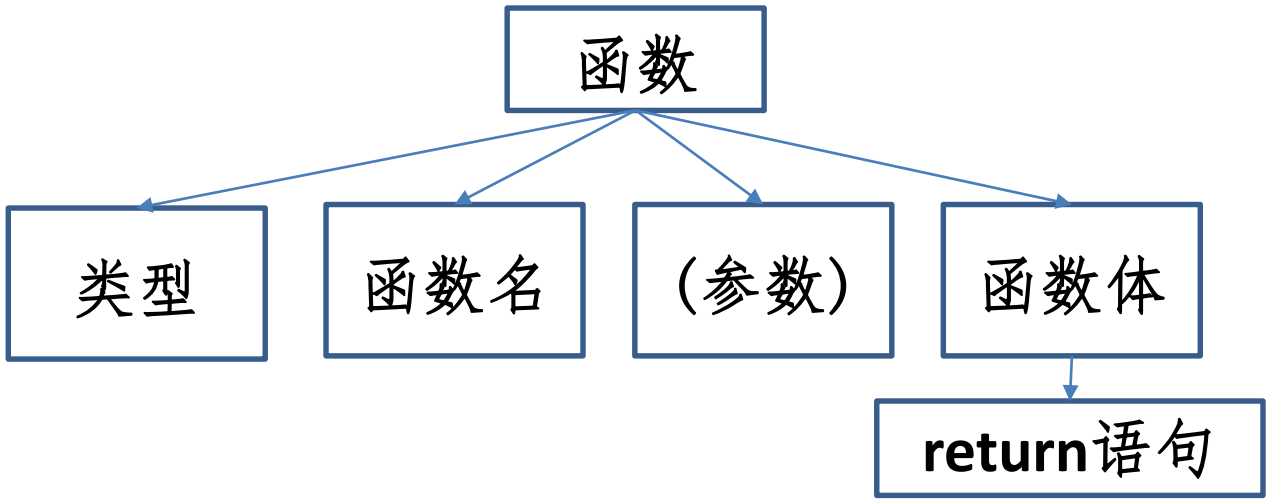


“int”, “ ”, “main”, “(”, “)”,
“ ”, “{”, “\n”, “\t”, “return”
“ ”, “0”, “ ”, “;”, “\n”,
“}”, “\n”, “” .

KWINT, SPACE, ID, OPAR, CPAR,
SPACE, OBRACE, SPACE, SPACE, KWRETURN,
SPACE, OCTALCONST, SPACE, SEMICOLON, SPACE,
CBRACE, SPACE, EOF .

```
int main() {
        return 0;
}
```

KWINT, SPACE, ID, OPAR, CPAR,
SPACE, OBRACE, SPACE, SPACE, KWRETURN,
SPACE, OCTALCONST, SPACE, SEMICOLON, SPACE,
CBRACE, SPACE, EOF



An Example: adder0.jj

```
options {
STATIC = false ;
}

PARSER_BEGIN(Adder)
  public class Adder {
    public static void main( String[] args ) throws ParseException, TokenMgrError {
      Adder parser = new Adder( System.in ) ;
      parser.Start() ;
    }
}
PARSER_END(Adder)

SKIP : {" " }
SKIP : { "\n" | "\r" | "\r\n" }
TOKEN : { < PLUS : "+" > }
TOKEN : { < NUMBER : (["0"-"9"])+ > }

void Start() :
{
}
{
  <NUMBER>
  (<PLUS><NUMBER>)*
  <EOF>
}
```

选项段，缺省为true：所有生成的方法、类声明为 “static”

类定义：定义一个名为“Adder” 的Java 类
这里所看到的不是Adder 类的全部，JavaCC 会添加其他代码
main 方法宣称可能在运行时隐式的抛出两个异常：
ParseException 和TokenMgrError；都会由JavaCC生成

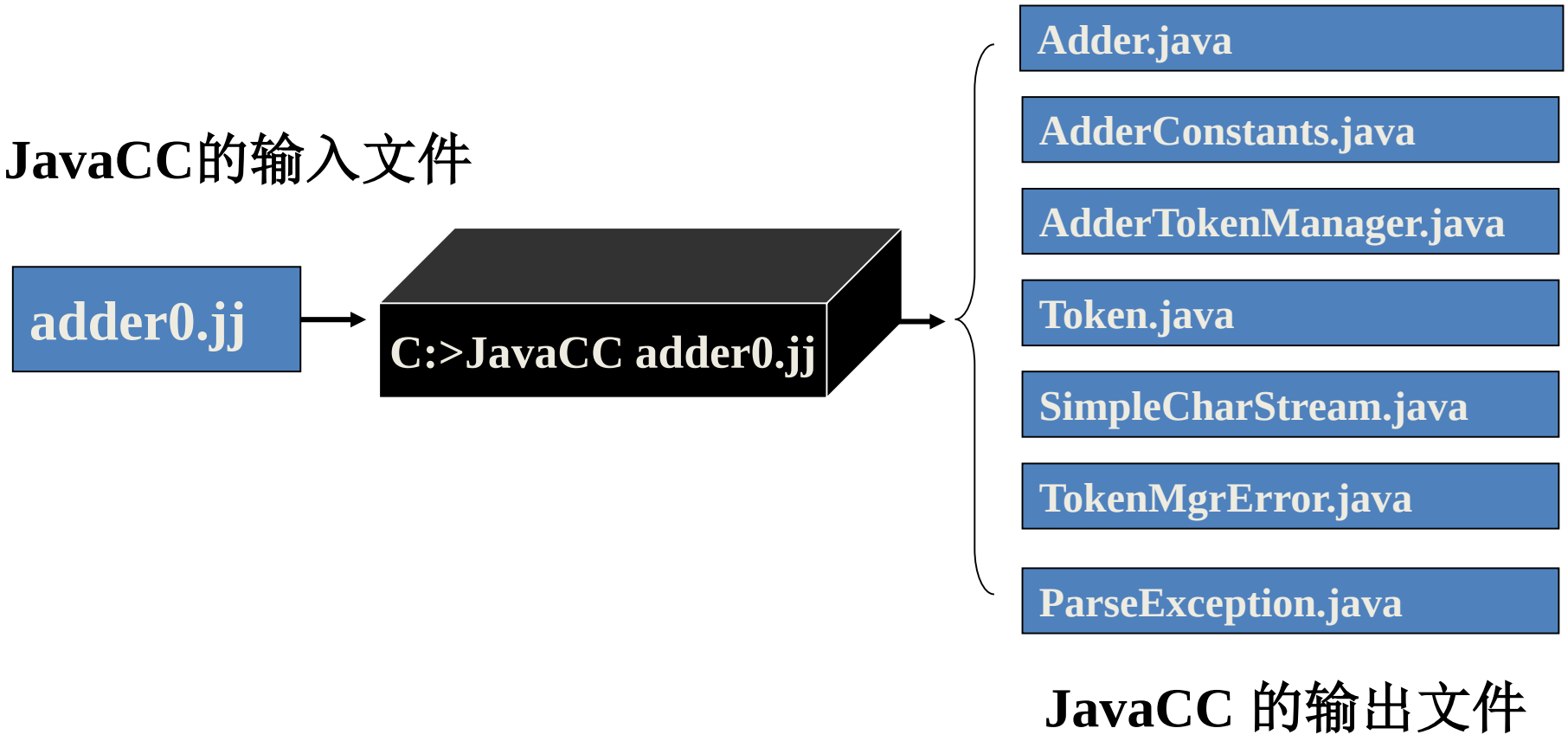
词法分析：
第一行：空格是一个token, 会被忽略。
第二行：各种换行符是token, 会被忽略。
第三行：一个单独的加号是一个名为PLUS 的token。
第四行：数字的语法，并为它们取名为NUMBER。

语法分析：

以NUMBER开头，
中间存在零个或多个由一个PLUS 后面跟一个NUMBER 组成的子序列
以EOF结束

一个例子：adder.jj

- 检查表达式是否为几个整数相加



- TokenMgrError
  - used for errors detected by the lexical analyser
  - “123-456\n”
- ParseException
  - used for errors detected by the parser
  - “+123+456\n”
- Token
  - a class representing tokens. Each Token object has an integer field kind that represents the kind of the token (PLUS, NUMBER, or EOF) and a String field image, which represents the sequence of characters from the input file that the token represents
- AdderTokenManager
  - Lexical analyser
- Adder
  - parser

JavaCC的局限性

- **JavaCC**不直接支持分析树或抽象语法树（**AST**）的生成
- 如果要完成这些功能，有两种方法：

  1）用户需要自己编写相应的代码

  2）利用 **JTB**（**Java Tree Builder)**

  自动生成分析树或抽象语法树

JTB（Java Tree Builder）

- 类似于**JJTree,** 采用 **Vsitor** 设计模式
- 将如下文件复制到当前工作目录：

  minijava.jj，jtb1.3.2.jar和javacc-x.0
- 执行如下命令：

  java -jar jtb132.jar minijava.jj
- 会生成**syntaxtree**和**visitor**两个目录以及文件 **jtb.out.jj**
  - **syntaxtree**目录下的每个文件**(**类**)**代表**AST**的一种**node**，代码中的注释会注明当前**node**由哪个产生式自动生成的**(**参考**jtb.out.jj)**
  - **visitor**目录包含了遍历**AST**所有结点的**Visitor(**访问者模式**)**接口，不论是构造符号表、类型检查还是代码生成，你只需实现完成相应功能的**Visitor**即可

- 继续执行命令：

  javacc jtb.out.jj
- 会生成前述的多个 Java 文件

后面就可以

- 自己写一个 Main 类
- 调用 javacc 生成的 parser 入口
- 用自己实现的Visitor遍历AST的每个结点
  – 得到想要的结果

Outline

- Visitor Pattern
- JTB and JavaCC
- An Example

Print Var Names

```
import visitor.*;
import syntaxtree.*;

class MyVisitor extends DepthFirstVisitor {
        public void visit(VarDeclaration n) {
                Identifier id = (Identifier)n.f1;
                System.out.println("VarName: "+id.f0.toString());
                n.f0.accept(this);
                n.f1.accept(this);
                n.f2.accept(this);
        }
}
```

```
public class Main {
        public static void main(String args[]){
                try{
                        InputStream in = new FileInputStream(args[0]);
                        Node root = new MiniJavaParser(in).Goal();
                        root.accept(new MyVisitor());
                }catch (ParseException e){
                        e.printStackTrace();
                }catch (TokenMgrError e){
                        e.printStackTrace();
                }catch (Exception e){
                        e.printStackTrace();
                }
        }
}
```