Language Processors Lab 5 - Visitors

In this lab you will learn how to use **visitors**. You will be given a JavaCC file and a set of Java files that implement an arithmetical expression parser, calculator and printer. You will be asked to modify the files (JavaCC and Java) in order to implement a new arithmetical operator.

Downloading the files

Start a Unix shell window and move to your LanguageProcessors directory. Download the file `lab5.tar.gz` from CitySpace (week7) or from `http://www.soi.city.ac.uk/~sbbc287/lab5.tar.gz`. Unzip and untar the file with the following commands:

```
gunzip lab5.tar.gz       // this will genenerate lab5.tar
tar -xvf lab5.tar
```

The last command will generate the `lab5` directory. It is possible that, while downloading, the file system automatically unzips, and even expands (untars), the file. If that's the case, you may have to skip one or both of the above commands, though make sure to copy the new directory. Load java and javacc by executing the command `module add java javacc`.

The Program

The contents of the `lab5` directory are:

- `Exp.jj`: Defines the grammar for expressions and contains Java code that builds the abstract syntax tree (AST) of expressions.
- `Main.java`: Entry point of the program. It calls the parser, stores the AST (of class Exp), instantiates a calculator visitor which is then called to evaluate the AST.
- The `syntaxtree` directory: It contains the classes that implement the abstract syntax tree for expressions. The file `Exp.java` defines the abstract class `Exp`. All expressions - nodes of the AST - are subclasses of `Exp`: `PlusExp`, `MinusEx`, `TimesExp`, `DivideExp` and `NumExp`.
- The `visitor` directory:
  - `Visitor.java`: Defines the `Visitor` interface.
  - `Calc.java`: Implements the `Visitor` interface. Traverses an expression's AST and calculates "its value".
  - `AstPrint.java`: Implements the `Visitor` interface. Traverses an expression's AST and prints its contents.

Execute JavaCC, compile the generated java files and execute the program by typing the usual commands:

```
javacc Exp.jj            javac *.java            java Main
```

Test the program with a few examples. What happens? Why? The rest of this section provides a brief explanation of the code. Start the text editor and load `Main.java`. Inspect its contents:

```
1     Exp root;
2     ExpParser parser = new ExpParser(System.in);
3     try {
4       System.out.println("Type in an expression on a single line.");
5       root = parser.S();
6       Calc calculator = new Calc();
```

```
7       int value = root.accept(calculator);
8       System.out.println("Answer is "+value);
9       /* Add new code after this line */
10    } catch (ParseException e) {
11      System.out.println("Expression parser - error in parse");
12    }
```

Line (2) instantiates the parser and stores it in variable `parser`. Recall that the entry point of the grammar is `S()` (check this). Line (5) calls the parser and the result, an object of class `Exp` and root of the AST, is stored in `root`. Line (6) instantiates visitor `Calc` and stores it in variable `calculator`, which is then used in line (7) to visit the AST `root`. The result is stored in variable `value` and printed in the next line. All this code is enclosed by a try clause given that line (5) can throw a `ParseException` object.

Grammar and AST generation: `Exp.jj`

The JavaCC file `Exp.jj` implements the following grammar:

E → T (`+` T | `-` T)*
T → F (`*` F | `/` F)*
F → number | `(` E `)`

The main difference between this JavaCC file and last week's file is that, instead of parsing expressions and computing their values, the Java code inserted in the grammar builds an AST. First of all, the return value of all non-terminals now has type `Exp`. Second, the previous Java actions (see lab4) have been replaced by calls to constructors. For example, check the definition of E() in `Exp.jj`:

```
Exp E() :
{ Exp e; Exp t; }
{
  e=T() (  "+" t=T() { e=new PlusExp(e,t); }
         | "-" t=T() { e=new MinusExp(e,t); } )*
    { return e; }
}
```

Besides matching T() and the plus and minus operators, the code above gets the intermediate AST's (of type Exp) and uses them to build a new AST by calling the respective constructors, `PlusExp` or `MinusExp`.

Thus, at the end of parsing (if succesful), method `S()` returns an AST that represents the input. For an example, check line (5) in `Main.java`, above.

The Calculator implemented as a Visitor

The Visitor Pattern allows us to create new operations based on tree traversal *without* changing the abstract syntax tree classes. That is, all new operations (defined as visitors) performed on ASTs are implemented by separate classes. The implementation of a visitor takes three steps:

- Define the visitor's interface. In our case, we want to define a visitor that visits all nodes of the AST and,a fter each visit, returns an integer. Therefore, we define a Java `interface` that declares a `visit` method for each class that is part of the `Exp` AST. The `visit method` returns `int`.

```
public interface Visitor {
  public int visit(PlusExp n);
  public int visit(MinusExp n);
  public int visit(TimesExp n);
```

```
    public int visit(DivideExp n);
    public int visit(NumExp n);
  }
```

This interface can be found in `Visitor.java`.

- In order to be able to traverse the AST, each AST class has to accept the visitor. In our case, each AST class has to accept an object that implements the interface `Visitor`. We do this by adding the method `accept`, with argument of type `Visitor`, to each AST class. For example, we add the following to `PlusExp` (check the actual code in `PlusExp.java`!):

  ```
  public int accept(Visitor v) { return v.visit(this); }
  ```

  Note that the `visit` method calls back the visitor `v`, passing itself as argument. If you check back the definition of `Visitor` above, there must be a `visit` method per AST class. In a sense, the node is saying to the visitor "you can visit me, but do it somewhere else".

- Next, we need to create the actual visitor. We do this by defining a new class e.g. `Calc`, that implements the `Visitor` interface:

  ```
  public class Calc implements Visitor { ... }
  ```

  and by providing an implemention to each of the `visit` methods declared in `Visitor`. In our case, we want to compute the value of the expression (like a calculator). For example, the value of a `NumExp` must be number itself. Thus, the implementation of `visit` for `NumExp` must be:

  ```
  public int visit (NumExp n) { return Integer.parseInt(n.f0); }
  ```

  The value of a `PlusExp` is the addition of the values of its left (e1) and right (e2) operators. We do this by calling `accept` (traversing) on e1 and e2, and adding the returned values:

  ```
  public int visit (PlusExp n) { return n.e1.accept(this) + n.e2.accept(this); }
  ```

Check the rest of `Calc.java` and the Java files in directory `syntaxtree` for the complete implementation.

Adding the power operator

We want to be able to write and evaluate expressions that contain the power operator `^`. For example, we want to type `2^5 + 10` and get back 42. In order to do this, follow these steps:

- Modify the grammar. The power operator should have precedence to the plus and minus operator. For example, when we write `2^5 + 10`, we mean `(2^5) + 10` and not `2^(5 + 10)`. Therefore, you need to add a line to the definition of T() in `Exp.jj`:

  ```
  Exp T() :
  { Exp t; Exp f; }
  {
    t=F() (  "*" f=F() { t=new TimesExp(t,f); }
           | "/" f=F() { t=new DivideExp(t,f); }
           | "^" f=F() { t=new PowerExp(t,f); } )*  // definition of power
      { return t; }
  }
  ```

Note that we are making a call to the constructor of the `PowerExp` class, which does not exist yet. That's our next step.

- Create a new file, `PowerExp.java`, with the following contents:

```
package syntaxtree;

import visitor.*;

public class PowerExp extends Exp {
  public Exp e1, e2;
  public PowerExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
  public int accept(Visitor v) {
    return v.visit(this);
  }
}
```

- Add a declaration of the `visit` method for the new `PowerExp` class inside the `Visitor` interface:

```
public interface Visitor {
...
  public int visit(PowerExp n);   // new line in Visitor.java

}
```

- And, finally, write the implementation of `visit` for the new class. Insert the following method definition in `Calc.java`:

```
  public int visit (PowerExp n) {
      return (int) java.lang.Math.pow(n.e1.accept(this),n.e2.accept(this));
  }
```

  Note that we are using the definition of power provided by the java.lang.Math library.

All the steps above must be done before the next compilation.
Execute javacc, compile the files and execute the program. Test it, for example, with `2^5 + 10`. Does it work? It should.

Printing the output

The `visitor` directory contains another file, `AstPrint.java`. This file implements a visitor that traverses the AST and prints its contents. In order to use it, add the following lines to `Main.java`:

```
/* Add new code after this line */
 AstPrint printer =  new AstPrint();
 System.out.println("The expression is: ");
 root.accept(printer);
```

Compile again. What happens?
You get a compilation error because the visitor `AstPrint` has not implemented `visit` for `PowerExp`. Provide an implementation!