Language Processors Lab 5 - Building an Abstract Syntax Trees (ASTs)

In this lab you will learn how to use JavaCC semantic actions to build Abstract Syntax Trees. You will be given a JavaCC file that implements an arithmetical expression calculator using JavaCC semantic actions (the solution to previous week's lab), and a set of Java classes that implement ASTs for arithmetical expressions. You will be asked to modify the JavaCC file to inlcude the correct constructor calls (that build the AST) and modify the AST classes in order to implement a new calculator. You will then be asked to add 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 Moodle (week5) 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 evaluates arithmetical expressions while they are parsed.
- `Main.java`: Entry point of the program. It calls the parser which, in turn, avaluates the parsed arithmetical expression.
- 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`.All classes declare the method:

  ```
  public int evaluate()
  ```

  which will contain code that implements the new calculator. It currently returns 0. These classes are currently not used by `Exp.jj`; they will by included in the program in the next section/

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

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

Test the program with a few examples. Double-check that the Calculator returns the correct results.

Building Abstract Syntax Trees fron JavaCC

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

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

The JavaCC file contains semantics actions (Java code) that implement the evaluation of arithmetical expressions into integer values. For example, the specification of `E()`:

```
int E() :
{ int e; int t; }
{
  e=T() (  "+" t=T() { e=e+t; }
         | "-" t=T() { e=e-t; } )*
    { return e; }
}
```

returns an `int`: the result of adding (or subtracting) the `T()`'s matched by the parser (which also return integers).

We want to change these specifications so they instantiate Java objects (of class Exp) instead. Our first step is to change the return type of all specifications from `int` to `Exp`. Modify `S()` so it looks like:

```
Exp S() :    // Note the return type Exp.
{ Exp s; }   // The temporary variable is also of type Exp
{
    s=E() <EOL> { return s; }
  | <EOL> | <EOF>
}
```

`E()` must return an object of class `Exp` that is instantiated to either a `PlusExp` or a `MinusExp`. These are tree nodes that contain two branches each: the left- and right-hand side operators. Change `E()` to:

```
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`.

Similarly, `T()` must instantiate `TimesExp` and `DivideExp` objects:

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

The `F()` specification - which denotes integer constants and expressions in parenthesis - must return a `NumExp` object, or the object denoting the expression in parenthesis. Change `F()` to the following:

```
Exp F() :    // Don't forget the new return type!!
{ Token t; Exp result; }  // and the new type of result!
{
    t=<NUM> { return new NumExp(t.image); }
  | "(" result=E() ")" { return result; }
}
```

Note that `Exp.jj` has to import all the Java classes needed to build the AST. Add the following import declaration (`syntaxtree` is the package that contains all the AST classes) so the first lines of `Exp.jj` look like:

```
PARSER_BEGIN(ExpParser)

import syntaxtree.*;
public class ExpParser {
```

Finally, we must update `Main.java` so it reflects the new type returned by hte parser. We will also call the `evaluate()` method on the returned AST. Update `Main` so it looks like (DON'T include the line numbers!!):

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

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) invokes method `evaluate()` which, in theory, should evluate the parsed arithmetical expression. 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.

Run javacc, re-complie the files and execute the program. What happens? You will note that the value of the evaluated arithmetical expressions is always 0.

Implementing the Calculator as tree traversal

So far the parser returns a Java object: the abstract syntax tree of the parsed expression. Expression evaluation can be implemented by traversing the AST and performing arithmetical operations at each node of the tree. Let's have a look at the `PlusExp.java` file:

```
package syntaxtree;

public class PlusExp extends Exp {
  public Exp e1, e2;
  public PlusExp(Exp a1, Exp a2) { e1=a1; e2=a2; }
  public int evaluate() { return 0; }

}
```

Note that the class has two fields of class `Exp`, `e1` and `e2`, storing the ASTs of the left- and right-hand-side operators, respectively. There is also the method `evaluate`, intended to calculate the integer value of the expression. It currently returns 0.
Modify the method so it computes the integer value of its operators, and adds them together:

```
  public int evaluate() {
      return e1.evaluate() + e2.evaluate();
  }
```

Evaluation on a branch terminates when the traversal hits an AST leaf i.e. a numerical constant. This case is taken care of by the `NumExp` class. Open the `NumExp.java` file and modify the `evaluate` method so it looks like:

```
public class NumExp  extends Exp {
    public int v;  // the numeric value of string s
    public String s;
    public NumExp (String s) { v=Integer.parseInt(s); }
    public int evaluate() {
      return v;
    }
}
```

The explanation is straightforward: a number evaluates to itself.

Update the `evaluate()` method in `MinusExp`, `TimesExp` and `DivideExp`, so they correctly implement the evaluation of arithmetical expression.

Run javacc, recompile and run your new program. Try a few exmples that test all operators.

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`.

  Exercise: Complete the `PowerExp.java` file.