Language Processors Lab Week 4 - A Simple Calculator

In this lab you will learn how to modify a grammar in JavaCC and how to introduce semantic actions to a .jj file. You will start with a JavaCC file that partially implements the grammar of arithmetical expressions introduced in class. You will complete the grammar and add Java code (semantic actions) that implements a simple calculator of expressions.

The Exp.jj file

Start a Unix shell window and move to your LanguageProcessors directory. Create a new directory `lab4` and move inside it. Download the files `Exp.jj` and `Main.java` from CitySpace (week4) - save it to `lab4`. Make sure you load java and javacc by executing the command:

```
module add java javacc.
```

Start the text editor and load Exp.jj and Main.jj. Inspect their contents: you will see that the JavaCC specification defines a grammar for arithmetical expressions and a parser for that grammar defined by class `Exp`, and that the `Main` class instantiates an `Exp` object, which is used to execute the method `S()`, the entry point to the grammar.

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. This time you have to type `java Main` in order to execute the program because the `Main` class defined in `Main.java` is the one carrying the `static main` method. Also note that in order to test a new expression you will have to run the program again.

What kind of expressions does it accept? Does it accept parenthesis? Check the grammar against your test results. For example, enter: `3+7*9` What happens? Why?

A grammar for arithmetical expressions: `Exp.jj`

Why are trying to implement a calculator for arithmetical expressions defined by the following grammar:

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

Check the grammar defined in `Exp.jj`. What is missing? In particular check the part of the grammar that specifies the non-terminal `T`. Note that we haven't included the part of the grammar the implements the sequence of factors (F) combined with the multiplication and division operators. Replace the specification of T with the following:

```
void T() :
{}
{
  F() ( "*" F() | "/" F() )*
}
```

Run javacc, compile the program and test it.

Adding semantic actions

The goal of this section is to implement a simple calculator for the expressions defined by the grammar by introducing Java code to different parts of the specification.

The JavaCC specification currently defines (at least) four Java methods, one per non-terminal, with the following signatures:

```
void S()        void E()        void T()        void F()
```

All of them are methods that take no arguments and return nothing. However, our calculator must return an integer. This means that we need to change the signature of the non-terminal methods so they return `int` instead. In particular, we need to:

- Change `S()` to return the value generated by `E()`:

  ```
  int S() :
  { int s; }
  {
      s=E() <EOL> { return s; }
    | <EOL>  | <EOF>
  }
  ```

  Note how we have introduced the variable s in order to store the value returned by `E()`.

- Change `F()` so it calculates the numeric value of the token `<NUM>` and returns it, or evaluates the expression between parenthesis and returns the result.

  ```
  int F() :
  { Token t; int result; }
  {
      t=<NUM> { return Integer.parseInt(t.image); }
    | "(" result=E() ")" { return result; }
  }
  ```

  In this case we need two intermediate variables, of types `Token` and `int`.

- Make sure that the intermediate non-terminals `E()` and `T()` pass the value to the top of the tree:

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

  int T() :
  { int t; }
  {
    t=F() ( "*" F() | "/" F() )* { return t; }
  }
  ```

- And finally, the driver method `main` in `Main.java` must print the result of the evaluation. In `Main.java`, replace `parser.S()` with:

  ```
  int result = parser.S();
  System.out.println("Answer is "+result);
  ```

The new code stores the result of `S()` and prints it out.

Run javacc, recompile and execute. What happens?
For example, what's the return value of `6 - 8*5`. It's 6. Check again the code and you will notice that we haven't implemented the actual operations. The current code extracts the value of integers but do not do anything with them, the value is just returned. You will always get the leftmost integer e.g. `5*6 + 7*9` will return 5. Try to understand why.

The fix:

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

int T() :
{ int t; int f; }
{
  t=F() (  "*" f=F() { t=t*f; }
         | "/" f=F() { t=t/f; } )*
    { return t; }}
```

Run javacc, recompile and test. Make sure you understand why it works.