Language Processors Lab Week 4
Parsing Expressions - A Simple Calculator

In this lab you will learn how to work with grammars in JavaCC and how to introduce semantic actions to a .jj file. You will start with a JavaCC file that partially implements a grammar for arithmetical expressions. You will complete the grammar and add Java code (semantic actions) that implements a simple calculator of expressions. We will then find out that the grammar, though correct, is not useful for the standard evaluation of arithmetical expressions. A second grammar will solve the problem.

The Files

Start a Unix shell window and move to your `LanguageProcessors` directory. Create a new directory `lab4` and move inside it. Download the files `Exp1.jj`, `Exp2.jj`, `Calculator1.java` and `Calculator2.java` from Moodle (Lab4), and save them into your `lab4` directory. Make sure you load java and javacc by executing the command:

```
module add java javacc.
```

Start the text editor and load `Exp1.jj` and `Calculator1.java`. Inspect their contents: you will see that the JavaCC specification defines an incomplete grammar for arithmetical expressions and a parser for that grammar defined by class `Exp1`, and that the `Calculator1` class instantiates an `Exp1` object (the parser), 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 Exp1.jj
javac Calculator1.java
java Calculator1
```

Test the program with a few examples. This time you have to type `java Calculator1` in order to execute the program because the `Calculator1` class defined in `Calculator1.java` is the one carrying the `static main` method.

What kind of expressions does it accept? Does it accept parenthesis? Does it accept operators? Check the grammar against your test results. For example, enter: 5, `((5))`, 5 + 6 What happens? Why?

A grammar for arithmetical expressions: `Exp1.jj`

Modifying the grammar

The grammar specified by `Exp1.jj` implements part of the general grammar for arithmetical expressions indicated below:

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

In particular, it specifies the last two rules. Check `Exp1.jj`. The entry point of the grammar is `S()`, which says that a program is made of non-terminal `E` followed by the end-of-line token. The incomplete specification of `E()` indicates that an expression is made of non-terminal `F`: a number or another expression in parenthesis.

In this section we will try to specify the rules of the grammar above. We will find out the we have to adapt them in order to avoid left-recursion and conflicts. Let's start by adding the addition operator. Add the first rule of the grammar by modifying `E()` to:

```
void E() :
{}
{
  E() "+" E()
  |
  F()
}
```

Run JavaCC. What happens? JavaCC has detected the presence of left-recursion. You should get a message like:

```
Reading from file Exp1.jj . . .
Error: Line 28, Column 1: Left recursion detected: "E... --> E..."
```

We have to remove the first `E()` (the one that generates left-recursion) without modifying the meaning of the initial grammar (we are only considering the plus operator first). Change the specification of `E()` to:

```
void E() :
{}
{
  F() "+" E()
  |
  F()
}
```

Run JavaCC again. What happens? It still doesn't like it, the message error should contain:

```
A common prefix is: "(" <NUM>
         Consider using a lookahead of 3 or more for earlier expansion.
```

JavaCC complains that is has no way to decide (with just a single token lookahead) between the first and second rule since both start with `F()`. Let's try the following:

```
void E() :
{}
{
  F() ( "+" E() )?
}
```

Run JavaCC. Does it work? It should! The grammar says that an expression is made of a factor `F` followed - optionally - by the plus operator and another expression (the recursion is on the right). Now complete the grammar to include all four operators:

```
void E() :
{}
{
  F() ( "+" E() | "-" E() | "*" E() | "/" E() )?
}
```

Run JavaCC, compile with `javac Calculator1.java` and execute with `java Calculator1`. The program should accept any combination of correctly formed arithmetical expressions e.g. `5`, `(4+3)*7`, `7 - 8/2`, etc.

Implementing the Calculator - 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) three Java methods, one per non-terminal, with the following signatures:

```
void S()        void E()          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. Don't forget to change the return type.

```
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`, in order to capture the token returned by `<NUM>` and the numeric value returned by `E()`, respectively.

- Add Java code to the rule that specifies `E()` so that it computes the correct value of the expression.

```
int E() :
{ int e, result;}
{
  result=F()
  ("+" e=E() { result += e; }
  | "-" e=E() { result -=e;}
  | "*" e=E() { result *=e;}
  | "/" e=E() { result /= e; }
  )?
  { return result; }
}
```

  We have introduced two integer variables, `result` and `e`, in order to store the value of the left-hand side of the expression matched by `F()` and the right-hand side (if any) matched by `E()`, respectively. The variable `result` is updated with the correct result of the operation only if the right-hand side exists. Note that the latter is included inside the optional part of the grammar.

- And finally, the driver method `main` in `Calculator1.java` must print the result of the evaluation. In `Calculator1.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. Try a few examples e.g.

`5` `(5+2)*7` `23+7+4` `23+7-5` `10-7+3` `10*7-2`

Are the results what you expected? They should, except for the last two. The expression `10-7+3` is parsed as `10-(7+3)`, and `10*7-2` as `10*(7-2)`, instead of the standard rules for grouping and operator priority. However, the parser is doing exactly what we asked for (check the grammar). Thus, we need to change the grammar.

A modified grammar for arithmetical expressions: `Exp2.jj`

We need to re-write the grammar in order to force the parser to group expressions using the standard priority of arithmetical operators. The new grammar should look like:

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

The grammar above is implemented by `Exp2.jj`, as well as most of the functionality needed for the Calculator. For example, check the new specification for `E()`:

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

Note the presence of the star (Kleene closure) and the Java code that implements addition and substraction. Variable `e` works as an accumulator which is updated everytime there's a match inside the Kleene closure. Run JavaCC, compile and execute the program:

`javacc Exp2.jj` `javac Calculator2.java` `java Calculator2`

Test the program. Does `10-7+3` work? Yes. What about `10*7-2`? It returns 8! Why? Check the definition for `T()`. The grammar is correct but the semantic actions (Java code) are missing. In our example, instead of computing `10*7` it ignores the 7 (and the multiplication) and returns 10 instead.

Fix it!