Language Processors Lab 3

Note: Read this through *before* logging in.

The goal for this lab is to see some JavaCC parsers working and to introduce to you how to put actions into the grammars.

Expression parser

Move to the directory in which you want to do your work and copy my directory and set up the shell to use JavaCC etc with the commands:

```
cp -R /soi/sw/courses/daveb/IN2009/bnfonly .
cd bnfonly
module add java soi javacc/3.2
```

The directory contains a JavaCC specification (tokens and grammar) for a simple expression language in file `Exp.jj`. The file `Main.java` contains a class Main with the `main()` method that sets everything up and calls the parser.

To create the parser, first run JavaCC with:

```
javacc Exp.jj
```

This creates a series of Java files containing classes that together go to make up the parser and lexical analyser. You do not need to look at these (indeed, some are will make no sense to you), but you may wish to look at `Token.java` to see how tokens are stored and the fields you can access to examine them. You can compile the whole thing with:

```
javac Main.java
```

and the run it with

```
java Main
```

If you type in a legal expression (eg `3+4*5+9`), the parser will print out a message about a successful parse. Try putting in an syntactically illegal expression (ie one with a syntax error in it). Then try putting in a lexical error (ie one with a character in it that is not included in the lexical specification or grammar).

Note that this parser does nothing other than recognise that the input tokens match the grammar JavaCC specification (or not).

The EBNF version of the expression parser is at `/soi/sw/courses/daveb/IN2009/ebnfonly`. Copy it and try it in the same way as above.

Adding actions

Although we haven't studied it in detail yet, we've mentioned in passing that you can embed actions in a JavaCC grammar. Copy the directory `/soi/sw/courses/daveb/IN2009/braces` in the same way as above. The file `Simple.jj` contains a grammar for matching braces `{}` and counting the levels of nesting (ie prints out 3 if you type in `{{{}}}`). The file `DoSimple.java` contains the class `DoSimple` and the `main()` method. Compile with:

```
javacc Simple.jj
javac DoSimple.java
```

and run with:

```
java DoSimple
```

and type in nested braces followed by end-of-file (`Ctrl-D` on Linux, `Ctrl-Z` on Windows).

Now look at the grammar in `Simple.jj`. If we take out the action code, a grammar that just parses matching braces is on the left side here:

```
void Input() :                              | void Input() :
{}                                          | { int count; }
{                                           | {
  MatchedBraces() <EOF>                     |   count=MatchedBraces() <EOF>
                                            |     { System.out.println("The levels of nesting is "+count); }
}                                           | }
                                            |
void MatchedBraces() :                      | int MatchedBraces() :
{}                                          | { int nested_count=0; }
{                                           | {
 <LBRACE> [ MatchedBraces() ] <RBRACE>      |   <LBRACE> [ nested_count=MatchedBraces() ] <RBRACE>
                                            | { nested_count = nested_count + 1; return nested_count; }
}                                           | }
```

The grammar with actions added is on the right. You will notice that we can put in some Java action code at the end of each definition and it gets executed when that definition is matched during the parse. We can also indicate that a definition will return a value (eg `int MatchedBraces() :`) and return it in the action code. We can also declare local variables at the head of definitions that can then be used in the definition both to assign values from matched non-terminals to (eg `nested_counted=MatchedBraces()`) and also in the action code in Java statements.

Note that this example does not show it, but non-terminal definitions can also be made to accept parameters or arguments and hence be passed values when they are used in other definitions.

Making the expression grammar a calculator

Change the EBNF version of the expression grammar in `/soi/sw/courses/daveb/IN2009/ebnfonly` so that the definitions return values and contain assignments so as to make the parser into a calculator that computes and prints out the value of the expression that is typed in.

As a start, the definition for `F()` is:

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

MiniJava parser

You may also like to take a first look at the MiniJava parser. There's a few things we must still cover before you can fully understand it, but you should be able to follow most of it.

It's in `/soi/sw/courses/daveb/IN2009/minijava/chap3`.