Language Processors Lab 2 - Week 2

READ: JavaCC Documentation.

GOAL: to implement a simple JavaCC-generated lexical analyser and use it as input to the RPN Calculator introduced in the previos lab.

The main functionality of the RPN Calculator was implemented by the StackMachine and RPN classes. The tokenisation (lexical analysis) of the input was performed by the Lexer class, which was called from the RPN class as shown by the following code extract:

```
Lexer lexer = new Lexer(e);      // Tokenises String e
Token token;
machine.clearStack();            // Clears contents of stack
while (lexer.hasNextToken()) {   // Loop while there's a token left
  token = lexer.getNextToken();  // gets next token from Lexer
  switch (token.kind) {          // inspects type of token
   case INTEGER:
      machine.pushInteger(Integer.parseInt(token.image));
      break;
   case PLUS: machine.plus(); break;
   case MINUS: machine.subtract(); break;
   case MULT: machine.multiply(); break;
   case DIV: machine.divide(); break;
 }
}
...
System.out.println("Result: "+machine.popInteger()); // prints top of the stack
```

We want to replace the functionality of the Lexer class with a Lexical Analyzer generated by JavaCC.

A Trivial Lexical Analyzer

Fire up a Unix shell window. Your current directory should be your home directory. Check the contents of your current directory with the `ls` command:

```
ls
```

If you have already created your `LanguageProcessors` directory, make it your current directory by executing `cd LanguageProcessors`. Otherwise do the following

```
mkdir LanguageProcessors
cd LanguageProcesoors
```

Once inside the `LanguageProcessors` directory, create the `lab2` directory and make it your current directory:

```
mkdir lab2
cd lab2
pwd
```

The result of the last command should be `<local_host>/<username>/LanguageProcessors/lab2`.

In this lab we will use three commands: `javacc`, `javac` (the java compiler) and `java` (the java virtual machine). You should already have `java` as part of your environment though `javac` may not be part of it. Check if you have both commands installed by typing:

```
javac -version
java -version
```

If any of the commands above fail, type the following:

```
module add java
```

Do the same with javacc by typing:

```
module add javacc
```

We will start with a simple JavaCC lexical analyzer which recognises a set number of token types. Make sure you are currently inside the `lab2` directory and copy the `LexTest.jj`, `StackMachine.java` and `StackMachineException.java` files from the module's Moodle lab2 space. You can see the contents of the JavaCC file with:

```
more LexTest.jj
```

In order to run `javacc` with LexTest.jj as input, type the following:

```
javacc LexTest.jj
```

This produces a Java program made up of various files. This program is a lexical analyser that recognizes the tokens specified in `LexTest.jj`. Now compile these Java classes with:

```
javac LexTest.java
```

And run the program with:

```
java LexTest
```

Type in some identifer names and integers and see what happens. For example, what is the output of the following inputs?

```
Enter input> + 8 9 xy z
Enter input > 134 / -
```

The JavaCC file - Explanation

Fire up your editor of choice (KEdit, Xemacs) and load `LexTest.jj`. Inspect `LexTest.jj` for full details. A JavaCC file has three parts:

- The first part, delimited by `PARSER_BEGIN(<parser_name>)` and `PARSER_END(<parser_name>)`, must include the Parser's class definition. The class should be named after the parser i.e. `<parser_name>`. In our case, `<parser_name>` is `LextTest`. JavaCC will generate a set of `.java` files, including `LexTest.java` with a default `main` method. We can re-define the `main` method - as shown in this example - as well as add more methods and fields. In our example, we have re-defined the main method in order to display the greeting message and call the parser (`TokenList()`).
- The second part is dedicated to lexical specifications using regular expressions. It defines the tokens recognised by the lexical analyser. In our example, the two forms of token (SKIP and TOKEN) are demonstrated, along with most of the kinds of JavaCC regular expressions, as well as the use of local definitions (prefixed by a `#` in a TOKEN definition).
  LexTest defines the following tokens: `END_INPUT, INTEGER_LITERAL, PLUS, MINUS` and `IDENTIFIER`.
- The third part contains the syntax specifications (grammar) of our language. Our example uses a very simple grammar:

```
(<INTEGER_LITERAL> |  <PLUS> |  <MINUS> | <IDENTIFIER> )* <END_INPUT>
```

which says: accept a sequence (any length, including zero) of these tokens, followed by the `<END_INPUT>` token. Notice that `TokenList()` - a non-terminal and entry point to the grammar - is divided into two parts: the first part is used for variable declarations (`Token t`) while the second for the syntax specification itself.

It is possible to capture the tokens recognised by the lexical analyser by assigning them to a Token object (`t = <PLUS>`), and to access and print their kind (from the table `tokenImage` indexed by Token field `kind`) and the corresponding string that was matched (from Token field `image`) by the analyser. These are fields that belong to the pre-defined JavaCC class `Token`.

It is also possible to add Java code by placing it between curly braces. For example, we have added code to print information about every matched token:

```
{ System.out.println ("token found: "+ tokenImage[t.kind]+
                      " (‘"+t.image+"’)"); }
```

The syntax definitions part of this JavaCC specification simply matches all defined tokens and prints them out.

Modifying the Analyser

We would like to add three more tokens in order to capture the multiplication operator, the division operator and real numbers of the form 0.45, 34.567, etc. We will use the following token specifications:

```
TOKEN           REGULAR EXPRESSION
-----           ------------------
MULT            *
DIV             /
FLOAT_LITERAL   <DIGIT>+ . <DIGIT>+
```

Three steps are required:

- **EXTEND** the specification of the PLUS and MINUS tokens with:

```
TOKEN : /* Operators */
{
  < PLUS: "+" >
 | < MINUS: "-" >
 | < MULT: "*" >
 | < DIV: "/" >
}
```

- **ADD** the following token specification:

```
TOKEN : /* Literal Floating Point */
{
  < FLOAT_LITERAL: (<DIGIT>)+ "." (<DIGIT>)+ >
}
```

- **UPDATE** the grammar with:

```
(t = <INTEGER_LITERAL> | t = <PLUS> | t = <MINUS> | t = <MULT> |
t = <DIV> | t = <IDENTIFIER> | t = <FLOAT_LITERAL>)
```

in order to inform the parser to accept the new tokens.

- Run `javacc LexTest.jj` and recompile the new files with `javac LexTest.java`. Run the new program with `java LexTest` and test a few examples that use the new tokens e.g. `89 7 zz / 34.56 -`.

Implementing the Calculator

Now we are ready to integrate our parser/lexer with the StackMachine in order to implement the RPN calculator.

Replace the specification of `TokenList` with:

```
void TokenList() :
{Token t;
StackMachine machine = new StackMachine();}
{
   (
     (t = <INTEGER_LITERAL> | t = <PLUS> | t = <MINUS> | t = <MULT> |
      t = <DIV> | t = <IDENTIFIER> | t = <FLOAT_LITERAL>)
{
  try {
  switch (t.kind) {          // inspects type of token
     case INTEGER_LITERAL:
        machine.pushInteger(Integer.parseInt(t.image));
        break;
     case PLUS: machine.plus(); break;
     case MINUS: machine.subtract(); break;
     case MULT: machine.multiply(); break;
     case DIV: machine.divide(); break;
     default: System.out.println("Unknown Token "+tokenImage[t.kind]); return;
   }
  } catch (StackMachineException sme) {
     System.out.println(sme);
        return;
  }
}
)* <END_INPUT>
{
  try {
    System.out.println("Result: "+machine.popInteger());
  } catch (StackMachineException sme) {
    System.out.println("Error while reading result\n"+sme);
  }
}
}
```

Run javacc once more, re-compile and execute the program. Test a few examples e.g. **34 56 78 + -**. Does it work? What happens when you use an identifier? What happens when you enter a floating point?

Exercise: Extend the calculator so it accepts our floating point tokens. Note that the stack machine only accepts integers as input to its operations.