IN2009 Tutorial 2

In this tutorial you will develop, first an interpreter, and then a compiler, for a simple calculator language. A calc program has the following form:

- All calc programs start with an INT.
- The initial INT is followed by a sequence of zero or more *Operation*s, where an *Operation* is any one of:

  EQUALS
  PLUS INT
  MINUS INT
  TIMES INT

- An INT is a standard decimal integer literal (negative literals are allowed).
- EQUALS is the "**=**" character.
- PLUS, MINUS, TIMES are the characters "**+**", "**-**", "*****", respectively.
- The effect of running a calc program is to start with the initial integer value and calculate the running sum obtained by performing each operation in turn. Operations are echoed to the terminal so that the user can see what is being done (the EQUALS operation prints the current value of the running sum and outputs a line ending). Below are two example calc programs (on the left) and the output that they produce (on the right). Note that the line-structure of the input file is ignored.

A calc program

```
3 + 7
=
+ 5
+ 5 = + 11 +
-1
=
```

The output

```
3 + 7 = 10
 + 5 + 5 = 20
 + 11 + -1 = 30
```

```
32 = + 8 - 2
* 6 =
+ 1
= =
```

```
32 = 32
 + 8 - 2 * 6 = 228
 + 1 = 229
 = 229
```

Preparation

Step 1: Download the Jar file Lex.jar from Moodle. Put it somewhere convenient (the same place as your SSM.jar, for example).

Step 2: Download and extract the IntelliJ project Tutorial2.zip. The configuration of this project will need to be edited, so that the file paths are correct for your local set up. The next two steps walk you through the necessary edits. First open the project in IntelliJ.

Step 3: *Do this is so that IntelliJ can access the SSM and Lex libraries when it compiles your Java code*. Edit the project dependencies configuration as follows:

File ⇒ Project Structure… ⇒ Project Settings ⇒ Modules ⇒ Dependencies

Delete (-) the existing entries for SSM.jar and Lex.jar. Then:

+ ⇒ JARs or Directories…

navigate to where you have saved *your* Jar files and select both SSM.jar and Lex.jar. OK.

Step 4: *Do this so that you can run your code from the command-line*. Configure the CLASSPATH in IntelliJ Terminal preferences so that it has at least three entries: one for your copy of `SSM.jar`, one for your copy of `Lex.jar`, and one for your `out/production/Tutorial2` folder.

IntelliJ's Green Hammer is your friend. If you edit your Java code in IntelliJ and then immediately run it from the command-line, your Java code will not have been compiled, so your changes will not take effect. (When you run your code from IntelliJ, it automatically forces a recompile, but it is not automatic when you run from the command line.) Find your green hammer and use it to force a rebuild after you make edits in IntelliJ and before you run from the command-line.

Tasks

1. Study the code in TokensPrinter. Don't make any changes just yet. Compile it (use your green hammer!) and run it on the test files in the data folder (ignore the "expected output" files for now). For example:

   ```
   java calc.TokensPrinter data/test0.calc
   ```

   What happens?

2. Add the missing token definitions in TokensPrinter and re-run it on all the test files. When you have the correct token definitions, running it on `test0.calc` etc should print a list of tokens and exit normally. None of the negative test files contain valid calc programs, but only some of them contain *lexical* errors: which ones?

3. Make a copy of TokensPrinter and rename it as Calculator (still in the calc package). Modify the code in Calculator so that, instead of printing the tokens, it examines each token and carries out the appropriate calculator operation. Note:

   - You will need a variable to keep track of the running sum. This will be initialised with the value of the initial INT and then updated by the subsequent Operations.
   - You can test the `type` field of a token, to see what kind of token it is, by using Java's == operator (as a rule, this is not recommended for Strings, but it will be fine in this case). As an example:

     ```
     if (lex.tok().type == "EQUALS") {
        System.out.println(" = " + running_sum);
        lex.next();
     }
     ```

   - For INT tokens you will need the string that was actually matched (stored in the token's `image` field) so that you can convert it to a Java int value, eg:

     ```
     int i = Integer.parseInt(lex.tok().image);
     ```

   - Always remember to call `lex.next()` after you have processed a token.

   Initially, just aim to get your Calculator working for addition, and assume that the input file is a valid calc program (don't check for errors). Test its behaviour on files `test0.calc` and `test1.calc`. Invent some more tests of your own.

4. Extend your Calculator so that it implements multiplication and subtraction too. Test it!

5. Add error checks. If your Calculator encounters the wrong kind of token at any stage, it should halt by throwing an Error with a helpful error message. Test it using *all* the test files (including the negative tests).

6. *Challenge*. Make a copy of Calculator and rename it as Compiler. Delete the running sum variable from Compiler. Modify the code so that instead of actually doing the operations, it outputs SSM assembly code. Start small! Don't worry about multiplication and subtraction to start with, and only produce minimal output. As an example:

   ```
   if (lex.tok().type == "EQUALS") {
      System.out.println("push runningSum");
      System.out.println("push load");
      System.out.println("push 3");
      System.out.println("push sysc");
      lex.next();
   }
   ```

   Of course, to test your compiler you will need to assemble and execute the SSM code that it generates. You can either copy your compiler's output from the terminal window, and then paste it into a file, or (better) redirect the compiler's output into a file directly:

   ```
   java calc.Compiler data/test1.calc > data/test1.ssma
   java Assemble data/test1.ssma
   java Exec a.out
   ```