# Project phase 2 — Exercises in Java and PostScript programming — assigned Sunday 14 September, due Thursday 25 September

## Reading assignment

Read Chapters 1 through 5 of *The Java Programming Language, Third Edition*.

Read the following parts of the *PostScript Language Reference Manual, Second Edition*:

- all of Chapter 1
- all of Chapter 2
- from Chapter 3: Sections 3.1 through 3.6
- from Chapter 3: Section 3.7 (may omit 3.7.2, 3.7.4, 3.7.6, 3.7.7)
- from Chapter 3: Section 3.8 (may omit 3.8.4, 3.8.5)
- from Chapter 3: Section 3.11
- from Chapter 8: Section 8.1

## 2.1 Exercise in Java programming

**Task in brief:** Revise the code from your project Phase 0 to comply with more stringent specifications on input and output. Extend the code so that certain additional commands are recognized.

**Input specification:**

The calculator accepts commands from the standard input. Each command is a complete line of text.

The following commands need to be recognized (we give their operational semantics in terms of what they pop off the stack and what they push back onto the stack):

- control commands
  - `quit`: immediately quits the calculator **(this is new!)**
- stack manipulation commands
  - `exch`: pop *y*, pop *x*, push *y*, push *x*
  - $\varepsilon$ (an empty string as command, also known as "just hitting return"): pop *x*, push *x*, push *x*; this is like the `dup` operator in PostScript **(this is new!)**
- numbers: push a number
  - both integer and floating-point numbers (including floating-point numbers with exponents) must be accepted, but all numbers are internally stored and displayed as floating-point numbers (Java type `double`)
- binary arithmetic operators
  - `+`: pop *y*, pop *x*, push $x+y$
  - `-`: pop *y*, pop *x*, push $x-y$
  - `*`: pop *y*, pop *x*, push $xy$

- `/`: pop $y$, pop $x$, push $\frac{x}{y}$
- `^`: pop $y$, pop $x$, push $x^y$

- unary mathematical functions
  - `Q`: pop $x$, push $\sqrt{x}$
  - `L`: pop $x$, push $\ln x$
  - `E`: pop $x$, push $e^x$
  - `S`: pop $x$, push $\sin x$; angle $x$ is accepted in degrees
  - `C`: pop $x$, push $\cos x$; angle $x$ is accepted in degrees
  - `T`: pop $x$, push $\tan x$; angle $x$ is accepted in degrees
  - `IS`: pop $x$, push $\arcsin x$; angle is returned in degrees **(this is new!)**
  - `IC`: pop $x$, push $\arccos x$; angle is returned in degrees **(this is new!)**
- binary mathematical functions
  - `fT`: pop $y$, pop $x$, push $\arctan \frac{x}{y}$; angle is returned in degrees; this two-argument function can be implemented using the Java method `Math.atan2` **(this is changed from phase 0, to match Emacs Calc better!)**
- fancy geometric functions
  - `solvetriangle` **(this is new!)**:
    pop $y_3$, pop $x_3$, pop $y_2$, pop $x_2$, pop $y_1$, pop $x_1$, push $\alpha_1$, push $\alpha_2$, push $\alpha_3$, where the six operands specify the Cartesian coordinates of the three vertices of a triangle and the results are the three corresponding angles of the triangle (in degrees), as in Figure 1; the behavior of this function is undefined if the three vertices coincide or are colinear, and you can assume (as part of the specification) that this case is guaranteed not to happen

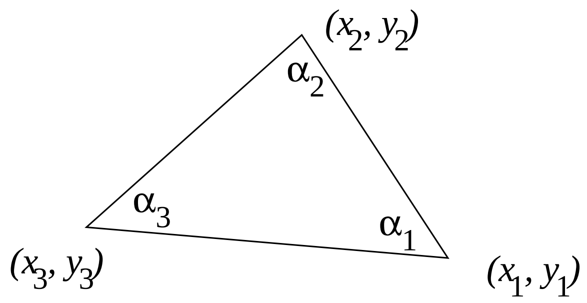

Figure 1: The vertices (inputs) and the angles (outputs) in a `solvetriangle` command.

Any other input is considered a syntax error, and must be ignored by the calculator: an error message is printed, but the stack is left intact.

**Computation:**

If there are not enough operands on the stack for a given operation, the calculator prints `"error: stack underflow"` and does not change the stack. Otherwise it performs the indicated computation.

**Output specification:**

The calculator does not display any kind of prompt.

Upon accepting user input and computing the next stack state, the calculator displays the stack as follows: The string `"______________"` is printed first on a separate line. Then each stack element is printed on a separate line, from the bottom of the stack to the top of the stack. In front of each stack element, its position in the stack is indicated, counting from the top of the stack, where position 1 corresponds to the top of the stack. In other words, the entire stack is displayed upside-down, just like in Emacs Calc.

**Example:**

If the following commands are given in standard input:

```
1
2
3
4
+
+
+


+
+

S
exch
C
4
*
E
^

0
exch
-
fT
C
IC
S
IS



exch
100
+
exch


3
Q
100
*
+
solvetriangle
```

```
quit
```

then the following is printed to standard output:

```
_______________
1:  1.0
_______________
2:  1.0
1:  2.0
_______________
3:  1.0
2:  2.0
1:  3.0
_______________
4:  1.0
3:  2.0
2:  3.0
1:  4.0
_______________
3:  1.0
2:  2.0
1:  7.0
_______________
2:  1.0
1:  9.0
_______________
1:  10.0
_______________
2:  10.0
1:  10.0
_______________
3:  10.0
2:  10.0
1:  10.0
_______________
2:  10.0
1:  20.0
_______________
1:  30.0
_______________
2:  30.0
1:  30.0
_______________
2:  30.0
1:  0.49999999999999994
_______________
2:  0.49999999999999994
1:  30.0
_______________
2:  0.49999999999999994
1:  0.8660254037844387
_______________
```

```
3:  0.49999999999999994
2:  0.8660254037844387
1:  4.0
_______________
2:  0.49999999999999994
1:  3.464101615137755
_______________
2:  0.49999999999999994
1:  31.94774550588494
_______________
1:  2.414183675246273E-10
_______________
2:  2.414183675246273E-10
1:  2.414183675246273E-10
_______________
3:  2.414183675246273E-10
2:  2.414183675246273E-10
1:  0.0
_______________
3:  2.414183675246273E-10
2:  0.0
1:  2.414183675246273E-10
_______________
2:  2.414183675246273E-10
1:  -2.414183675246273E-10
_______________
1:  135.0
_______________
1:  -0.7071067811865475
_______________
1:  135.0
_______________
1:  0.7071067811865476
_______________
1:  45.00000000000001
_______________
2:  45.00000000000001
1:  45.00000000000001
_______________
3:  45.00000000000001
2:  45.00000000000001
1:  45.00000000000001
_______________
4:  45.00000000000001
3:  45.00000000000001
2:  45.00000000000001
1:  45.00000000000001
_______________
4:  45.00000000000001
3:  45.00000000000001
2:  45.00000000000001
```

```
1:  45.00000000000001
_______________
5:  45.00000000000001
4:  45.00000000000001
3:  45.00000000000001
2:  45.00000000000001
1:  100.0
_______________
4:  45.00000000000001
3:  45.00000000000001
2:  45.00000000000001
1:  145.0
_______________
4:  45.00000000000001
3:  45.00000000000001
2:  145.0
1:  45.00000000000001
_______________
5:  45.00000000000001
4:  45.00000000000001
3:  145.0
2:  45.00000000000001
1:  45.00000000000001
_______________
6:  45.00000000000001
5:  45.00000000000001
4:  145.0
3:  45.00000000000001
2:  45.00000000000001
1:  45.00000000000001
_______________
7:  45.00000000000001
6:  45.00000000000001
5:  145.0
4:  45.00000000000001
3:  45.00000000000001
2:  45.00000000000001
1:  3.0
_______________
7:  45.00000000000001
6:  45.00000000000001
5:  145.0
4:  45.00000000000001
3:  45.00000000000001
2:  45.00000000000001
1:  1.7320508075688772
_______________
8:  45.00000000000001
7:  45.00000000000001
6:  145.0
5:  45.00000000000001
```

```
4:  45.00000000000001
3:  45.00000000000001
2:  1.7320508075688772
1:  100.0
_______________
7:  45.00000000000001
6:  45.00000000000001
5:  145.0
4:  45.00000000000001
3:  45.00000000000001
2:  45.00000000000001
1:  173.20508075688772
_______________
6:  45.00000000000001
5:  45.00000000000001
4:  145.0
3:  45.00000000000001
2:  45.00000000000001
1:  218.20508075688772
_______________
3:  90.0
2:  60.00000000000001
1:  29.999999999999993
```

Both files are provided to you.

**Benchmark:** A compiled Java program, `Calc.class`, is provided to you as a benchmark. This program performs according to the specification above (to the best of our knowledge). Your calculator program should work exactly the same.

For instance, executing:

```
java Calc <example.in >example.out
```

is how we produced the file `example.out`, and executing

```
java YourCalcProgram <example.in >example.out
```

must result in an identical `example.out` file. (Use `diff` to check this.)

**Testing:** It is very important to test your code. Provide *extensive* tests. Test cases should be designed *carefully* to cover all control-flow paths in your code. (The supplied `example.in` tests a few features, but is far from being exhaustive.) Provide the results of running your test cases in your README file.

**Discussion:** The goal of this exercise is to practice code extension and revision. If you were to write the program from phase 0 again, would you do it differently? Was your design easy to revise and extend? Write about this in your README file.

**Effort:** We estimate code size to be about 350 lines of Java, and about an hour of *programming* beyond Phase 0.

## 2.2 Exercise in PostScript programming

Write a PostScript program that reverses the order of the characters in a text file.

Supposing that the program is in a file `reverse.ps`, it should be invoked like this:

```
gs reverse.ps <infile.txt 2>outfile.txt
```

For reference, you are given an example pair of files `infile.txt` and `outfile.txt`.

Explain why we have `2` in the command above.

Note that executing the program above should have roughly the same effect as the Unix command

```
cat infile.txt | rev | tac >outfile.txt
```

but it is OK if there is a discrepancy for newline characters at the end of the file.

In the interest of good programming style (see *PostScript Language Tutorial and Cookbook* and *PostScript Language Program Design*), strive to use the operand stack efficiently and to avoid using `def` to mimic local variables.

**Effort:** This can be done in less than 20 lines of PostScript by taking advantage of the PostScript execution model; or in no more than 80 lines otherwise.

## 2.3 Exercise in PostScript programming

Write a PostScript program that behaves similar to the Java program in section 2.1.

In other words, supposing that your PostScript program is in a file `calc.ps`, executing

```
gs calc.ps <example.in 2>example.out
```

has nearly the same effect as executing

```
java Calc <example.in >example.out
```

The difference is that the PostScript version does not need to be able to recognize floating-point numbers in the input. (Notice that the supplied example file, `example.in`, does not use floating-point numbers.) All integers in the input should still be converted internally to floating-point numbers, i.e. PostScript real numbers.

Because of differences in the accuracy of floating-point arithmetic built into Java and into PostScript, it is OK if the two calculators compute slightly different results (for instance, 29.999999999999993 vs. 30.0).

In the interest of good programming style (see *PostScript Language Tutorial and Cookbook* and *PostScript Language Program Design*), strive to use the operand stack efficiently and to avoid using `def` to mimic local variables.

**Effort:** This can be done in about 650 lines of PostScript. Count on spending at least a full day of programming beyond the Java version, because you must rewrite a moderate-size program at a rather low level of abstraction. It is very difficult to debug PostScript code, so you must instead think ahead and write procedures using the techniques of preconditions, postconditions, and loop invariants (which you learned about in CS251) so that you are sure of their correctness.

## How to turn in

Turn in your code by running

*˜barrick/handin your-file*

on a regular UNM CS machine.

You should use whatever filename is appropriate in place of your-file. You can put multiple files on the command line, or even directories. Directories will have their entire contents handed in, so please be sure to clean out any cruft.