UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Fall 2009

Project #1: A Simple Drawing Program

Due: 5 October 2009 at midnight (beginning of 6 October)

1 Overview

In these days of mice, touch screens, GUIs, and Wiis, it's easy to forget that it is possible to describe drawings with text. The Postscript language and its compressed, streamlined version PDF, for example, can describe arbitrary pages of text, images, and line drawings. In this first project, you'll write a translator for a very simple diagram-description language into a small subset of Postscript. You can actually display or print the results using standard software.

The details of parsing modern programming languages is really the purview of CS164 (plug) so we'll finesse that problem here by using Lisp as an input language (you will, however, have to write a simple Lisp reader).

When finished, you should be able to put a program written according to the specifications of §2 into a file *INPUTFILENAME*.`drw`, and then enter the command

`java draw` *INPUTFILENAME*.`drw` *OUTPUTFILENAME.ps*

and have the program translated into a Postscript file that will produce the specified picture. In addition,

`java draw` *FILENAME*.`drw`

will instead actually execute the Postscript drawing (this requires some arcane machinery, but don't worry: we've done the hard work for you in the skeleton). You can use '`-`' in place of the name of an input or output file to refer to the standard input or standard output. Thus,

```
java draw - -
```

works as a filter, reading from the standard input and writing to the standard output. Finally, the plain command

```
java draw
```

should print out a helpful summary of these commands.

2 Input

The input language for your program has a Scheme-like syntax. That is, a program consists of a sequence of any number of S-expressions, where an S-expression is either

- A real (floating-point) or integer numeral with optional sign in Java format, representing a double-precision floating-point number in either case, or
- A symbol consisting of a sequence of non-whitespace characters other than parentheses or semicolons, and not starting with a numeral, or
- A left parenthesis, followed by zero or more S-expressions, followed by a right parenthesis.

Symbols are case-sensitive (unlike most Lisp dialects). Symbols and numbers must be separated from each other by whitespace. Other use of whitespace between is optional. Comments begin with a semicolon and end at the next end of line.

This syntax represents programs that manipulate *pictures* and numbers. A picture is something that can be drawn.

Not all S-expressions are valid in programs. The valid ones fall into several categories.

2.1 Primitive operations.

N where N is a numeral. The value of this expression is the number denoted.

X where X is a symbol. This is a variable. The value is the last value assigned to X.

`(:=` X E`)` where X is a symbol and E is an S-expression. Assigns the value of E to X.

2.2 Arithmetic.

In the following, A and B are numeric-valued expressions

`(+` A B`)` The sum of A and B.

`(-` A B`)` The difference of A and B.

`(*` A B`)` The product of A and B.

`(/` A B`)` The quotient of A and B.

`(sin` A`)` The sine of A degrees.

`(cos` A`)` The cosine of A degrees.

`(sqrt` A`)` The square root of A, which must be non-negative.

2.3 Creating pictures.

All commands other than `group` "capture" the current values of two parameters: the line width and the color, which the user may change using commands described in §2.4 below. Once created, a rectangle, circle, or line retains the values of these parameters from the time of its creation, and they affect how it is drawn.

`(rect` X Y W H`)` A picture consisting of the outline of a rectangle whose lower-left corner is at coordinates (X, Y), and which has a width of W and a height of H.

`(filledrect` X Y W H`)` As for `rect`, but denotes a filled rectangle, rather than the outline of its boundary.

`(circ` X Y R`)` A picture consisting of the outline of a circle whose center is at (X, Y) and whose radius is R.

`(filledcirc` X Y R`)` As for `circ`, but denotes a filled circle, rather than the outline of its boundary.

`(line` X_0 Y_0 X_1 Y_1`)` A picture consisting of a line from (X_0, Y_0) to (X_1, Y_1).

`(group` P_1 $\cdots P_n$`)` A picture consisting of $P_1, \ldots, P_n$, $n \geq 0$, all of which are pictures.

2.4 Parameters.

The following commands set certain parameters used by subsequent picture-creation commands. Although these commands resemble those in Postscript (see §3), they differ in one important way: the functions `rect`, `filledrect`, `circ`, `filledcirc`, and `line` all "capture" these parameters at the time they are evaluated. Any change to the parameters after that has no effect on previously created pictures, which, when and if they eventually get drawn, use the parameters in effect when they were created.

`(color` R G B`)` Set the current color to (R, G, B). The arguments must be numbers between 0 and 1, inclusive. Initially, $R = G = B = 0$ (i.e., the initial color is black).

`(linewidth` W`)` Set the current line width to W, which must be a non-negative number. A line width of 0 means "as thin a visible line as possible[1]." The initial value of the line width is 1.

2.5 Operations on pictures.

None of the following operations are affected by the current settings of the parameters in §2.4. Those that create new pictures copy these parameters from their inputs.

`(move` P X Y`)` The picture resulting from moving P, a picture, X units in the x-direction and Y units in the y-direction, non-destructively.

`(rotate` P D`)` The picture resulting from rotating P, a picture, D degrees counterclockwise around the origin, non-destructively.

`(scale` P S`)` The picture resulting from non-destructively scaling P, a picture, by a factor of S, a number. All lengths in P (including all distances to the origin) are multiplied by S in the resulting picture.

`(draw` P`)` Draws P, a picture. If P is a group, then this draws each picture in the group in the order they were listed. The order is important, since lines and filled areas are opaque, so that it is the last one drawn over any given point that determines that point's color (conveniently, this is as in Postscript).

2.6 Control structures.

`(for` I L U E_1 $\cdots E_n$`)` Execute statements $E_1 \cdots E_n$ multiple times, first with I, a variable, set to L, an integer, then to $L + 1$, etc., up to and including U. Does nothing if $U < L$ or $n = 0$. Leaves I as the maximum of L and U.

[1] Conveniently, this is its meaning in Postscript, too.

3 Output

The Postscript program that you produce as output will have the following form:

```
%!PS-Adobe-2.0

commands

showpage
```

Postscript is a *postfix language,* meaning that each command consists of a sequence of arguments followed by an operator, with no parenthetical grouping or punctuation. This is the same style as a stack-based calculator or the programming language Forth. Instead of writing `(3+4)*(5+6)`, one writes

```
3 4 add 5 6 add mul
```

You can think of `3` as "pushing the integer value 3 on the stack" and `add` as "popping the last two values off the stack, adding them, and pushing the result back on."

Commands are in free format, meaning that operands and operators are separated from each other by arbitrary whitespace (blanks, tabs, and newlines). A '`%`' marks the beginning of a comment, which runs to the next end of line. Your program may insert whitespace and comments as you see fit (aside from the first one).

Postscript uses a coordinate system in which 1 unit is 1/72 inch (this is approximately one *point.* The origin of a page, $(0, 0)$, is initially the lower left corner, with x coordinates increasing to the right and y coordinate increasing upward. To draw something, one builds a *path,* a sequence of lines and curves, and then either *strokes* it (drawing a line having the current color), or *fills* it (painting the interior of the path with the current color and pattern). The Postscript interpreter maintains a *graphics state,* which for our purposes consists of

- A *current point* (xy position), which is initially null (absent);
- A *current path;*
- A *current color,* which for our purposes consists of a 3-tuple (r, g, b), where $0 \leq r, g, b \leq 1$ indicate the intensities of red, green, and blue light, respectively. The condition $r = g = b$ indicates a shade a gray, with $r = g = b = 0$ being black and $r = g = b = 1$ being white.
- A current line thickness, which indicates the thickness of the lines drawn when stroking (as opposed to filling) a path. Lines are centered on the path.

The set of possible commands and operands is a very small subset of what full Postscript allows. In this subset, operands are all decimal numbers (basically as in Java, including decimal fractions and possibly trailing exponents of 10, as in `1.3e-3`, which means 0.0013.

x y `moveto` Set the current point to (x, y).

x y `lineto` Add a line segment to the current path starting at the current point and going to (x, y), which becomes the new current point. An error if there is no current point.

Δ_x Δ_y `rmoveto` Set the current point to $(x+\Delta_x, y+\Delta_y)$, where (x, y) is the current point, which must be defined.

Δ_x Δ_y `rlineto` Add a line segment to the current path starting at the current point, (x, y), and going to $(x + \Delta_x, y + \Delta_y)$, which becomes the new current point. An error if there is no current point.

x y r θ_1 θ_2 `arc` Add an arc segment to the current path. The segment is the piece of a circle with radius r and center (x, y) that runs counterclockwise, starting at an angle of θ_1 degrees counterclockwise from the positive x-axis and ending at an angle of θ_2 degrees. If there is a current point, first inserts a line segment from the current point to the beginning of the arc (the point at θ_1 degrees). Since this results in an arc with a sort of tail on it, you probably want to avoid using this command when there is a current point in this project.

`stroke` Draw lines with the current line width and color over all the segments in the current path. Set the current path to empty, and set the current point to undefined.

`fill` Close the current path (that is, connect the current point to the last position set by `moveto` or `rmoveto`) and fill the interior of the current path with the current color. Set the current path to empty and the current point to undefined. For our purposes, the concept of "interior" probably coincides with your intuition[2].

W `setlinewidth` Set the current line width to W. The initial value of the line width is 1.

R G B `setrgbcolor` Set the current color to (R, G, B).

4 Handling Errors

When the input file contains erroneous input, we're not going to be terribly particular about what error messages you produce, except that

- They must be *your* error messages. Stack traces from exceptions automatically produced by Java aren't acceptable. The autograder will interpret them as errors you didn't catch. Just because Java catches a particular error by throwing an exception doesn't mean you are forced to end your program with an exception. See §1.15 in *A Java Reference*.
- All error messages must be printed on `System.err`, not `System.out` (and certainly not in your output file!). This is standard practice in Unix-like systems.
- All error messages must contain the literal word "error," with any capitalization, somewhere in their first line. This is how the autograder will know that you found *some* error in tests where it is expecting one.
- Every program exits with an integer *exit status,* which is supposed to indicate whether something went wrong. If the input is correct and all processing is normal, the exit status should be 0. Otherwise, it should be something else (may I suggest 1). Exiting a Java program by simply running off the end of the main method results in an exit code of 0. To exit with code N, use the library method `System.exit(`N`)`.

5 What to Turn In

You will need to turn in all your Java files, a user manual, and an `INTERNALS` file. You may change *any* of the code we've given you, as long as

- You don't add any more classes to the default package (i.e., more besides `draw.java`);

[2]Technically, a point is in the interior of the current path if it has a *nonzero winding number*. The winding number of a point is computed by taking a ray from that point to infinity (that is not tangent to any line segment in the path) and, starting at the point with a count of 0, adding one for each time the path crosses the ray running left to right and subtracting one each time the path crosses the ray running right to left.

- `java draw` works as described above.
- `java drawing.Testing` runs all your JUnit tests (which, when you are finished, should all pass by the way). Part of your grade depends on the thoroughness of your testing.

The user manual describes how to use the program. Mostly, we've already said that here, but we'd like you to present it in your own words as well. And of course, if in your zeal you added some features, you'll certainly want to describe those. The user manual should say nothing about implementation—about *how* your program works. That is for the INTERNALS manual.

The purpose of your `INTERNALS` file is to give a description of the structure of your project to future maintainers—people who take over the code from you and go on to fix (the few remaining) bugs or add enhancements. This description will give them a good idea of everything is supposed to work together, so that it is easier for them to read and understand your code. You don't need to echo all the comments: those details that are easily understood just by reading the comments on methods needn't be duplicated in `INTERNALS`, which is more of an overview.

We will expect your finished programs to be workmanlike—consistently indented, well-commented, readably spaced. Don't leave debugging print statements lying around. In fact, don't use them; learn to use the debugger (either `gjdb` or that of `Eclipse` or your favorite Java-system vendor).

6 Advice

First, get started immediately, of course. Don't just jump in and code, though. Make sure you understand the specifications first, and plan out how you're going to meet them. Figure out how to break this problem down into small pieces, and how to implement and test them one piece at a time. Know in detail how you're going to do something before writing a line of Java code for it.

Read the skeleton files and *understand* as much as possible. Don't allow things to remain mysterious to you, or they'll surely bite you at some point. We've put some stuff in the skeleton files precisely to get you to ask questions and (especially) to browse the Java library documentation. Again, however, as long as your program behaves properly, we don't care whether you use any of our skeleton files.

The Java library can help you. Look at `java.util.Scanner` and see Chapter 9 of *A Java Reference.* You'll be able to find a use for `java.util.HashMap` as well.

I suggest you write the user manual early on. First, it gets that task over with. Second, by restating the problem, you might get to understand it better. The same comment goes for the `INTERNALS` manual.

The JUnit philosophy describes a test-early-and-often attitude. You can write tests of the overall system *before* you write a line of code.

Above all, it is always fair to ask for help and advice. We don't *ever* want to hear about how you've been beating your head against the wall over some problem for hours. If you can't make progress, don't waste your time guessing or bleeding: ask. If nobody's available to ask, do something else (or get some sleep).