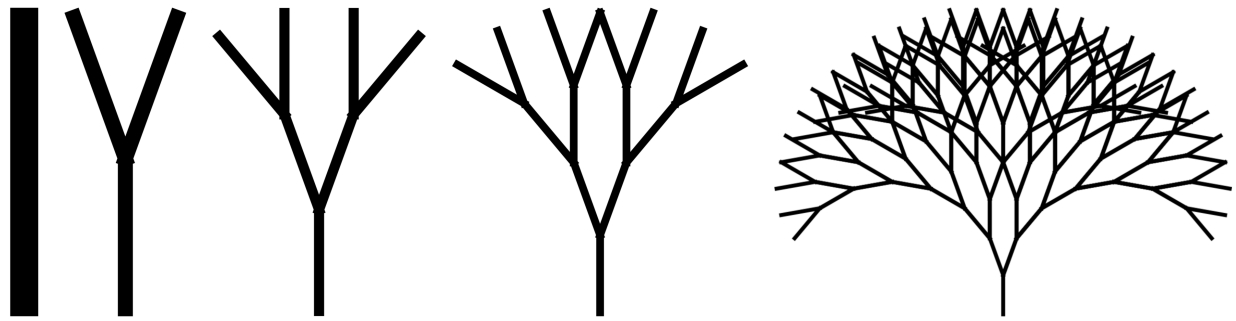
CS 245 Homework 3
Lindenmayer
due before December 14

1 Overview

A *Lindenmayer system* is a textual representation of a structure and rules for its growth. A plant, for instance, can be described in its youth as a straight line sticking out of the soil. As it grows, it bifurcates and bends. Each new stage of growth leads to more bifurcation and bending. Eventually, things get out of hand. The following image shows stages 0, 1, 2, 3, and 7 of an imagined plant's life:



The L-system that describes this plant has the following parameters:

Initial state: `F`
Theta: 10
Productions: `F` → `F[-F][+F]`

How does one interpret this system? An `F` means "go forward." In stage 0, all the plant does is go forward, so its initial state is simply `F`. To grow the plant, we apply the productions to the plant's current state. Any `F` gets replaced by `F[-F][+F]`. All characters that don't have explicit productions stay the same. The plant's state after one stage of growth, therefore, is `F[-F][+F]`.

Look at stage 1 in the image to see how we interpret `F[-F][+F]`. `F` means go forward, `[` means record the current position, `-` means turn left by *theta* degrees, `]` means return to the previously recorded position, and `+` means turn right by *theta* degrees.

Stage 2 is arrived at by applying the productions once again, but to the text of stage 1. We replace all the `F`s in `F[-F][+F]` with `F[-F][+F]`, leading to the text of stage 2:

```
F[-F][+F][-F[-F][+F]][+F[-F][+F]]
```

Your task in this assignment is to model an L-system and draw its current evolutionary stage using Java's 2-D drawing library. You will need to call upon several data structures to accomplish this task.

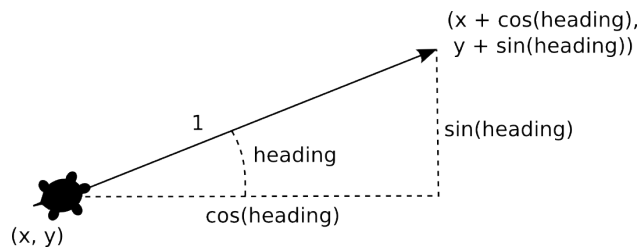
We provide a GUI interface in `LSystemViewer.java` that lets you interactively design and adjust an L-system. You will be tempted to run this GUI to test your code. Resist the temptation. It won't run until your classes have all the specified methods. You are advised to write unit tests as you go. Unit tests help you write far better software than interactive testing will.

2 Turtle Geometry

As you interpret the current state of a system and draw it, you'll need to move a cursor around a 2-D canvas. Instead of issuing commands like "go to position $(5, 16)$", an L-system emits relative, first-person commands. Kind of like driving a turtle. The system tells the turtle to move forward and turn left or right. That's it. Historically, this method of describing movement has been called turtle geometry.

We model the turtle in this homework with an xy coordinate and a heading or direction. Turning the turtle left and right is straightforward. We just increment or decrement the instance variable representing the turtle's heading by the amount of rotation.

Moving the turtle forward requires some trigonometric manipulation, because what "forward" means changes with every turn. The turtle's position after a move depends on both its current position and its heading. We'll have to use the turtle's heading to determine how its x and y coordinates change. This turns out to be the same problem as finding the side lengths of a right triangle given one other angle. The following figure shows the turtle at position (x, y) with a particular heading:



The change in x is cos(heading) and the change in y is sin(heading).

3 Requirements

1. Write all code in package `hw3`.
2. Write a class `Turtle` that encapsulates a position and its heading. It has the following:
   (a) A constructor that accepts three `double` parameters, two for the xy position of the turtle and one for the angle (in degrees) in which the turtle is headed.
   (b) Methods `get{X,Y,Heading}` that return the turtle's x coordinate, y coordinate, and heading, respectively—all as `double`s.
   (c) Methods `set{X,Y,Heading}` that accept a single `double` parameter and set to the given value the turtle's x coordinate, y coordinate, and heading, respectively.
3. Write a class `LSystem` that encapsulates the initial state, theta, and productions of a Lindenmayer system. Many of its methods are called by the provided `LSystemViewer` class, though you should test them yourself. It has the following:
   (a) A constructor that accepts two parameters: the system's initial state (a `String`) and its theta value (a `double`). Systems constructed in this way start off with no productions.

(b) A constructor that accepts one parameter: a `File`. The first line of the file contains the system's initial state, followed by theta. Every subsequent token pair is a production. The first token is a single character, and the second is the text the character is to be replaced with during the system's evolution. The following example fits this format:

```
F-F-F-F
90
F F-F+F+FF-F-F+F
```

This constructor sets the initial state, theta, and productions to those found in the file. You may assume the file is well-formed. If the file doesn't exist or is improperly formatted, don't try to catch the resulting exceptions. Let them be thrown. Someone who can actually rectify any problems should deal with them.

(c) A method `getInitialState` that returns the system's initial state as a `String`.

(d) A method `setInitialState` that accepts a `String` parameter and updates the system's initial state to the given value.

(e) A method `getTheta` that returns the system's theta value as a `double`.

(f) A method `setTheta` that accepts a `double` parameter and updates the system's theta value to the given value.

(g) A method `addProduction` that accepts a `char` key parameter and a `String` replacement parameter. It adds a production to the system such that during an evolution of the system, all instances of the key in its current state get replaced by the replacement text.

(h) A method `getProduction` that accepts a `char` key parameter. It returns as a `String` the replacement text associated with the given key. If no explicit production has been added for this key, the key itself is returned (as a `String`).

(i) A method `removeProduction` that accepts a `char` key parameter. It removes any production that is keyed on the key parameter. Its behavior when no production exists with this key is unspecified.

(j) A method `getProductionKeys` that returns the keys of all productions in the system as a `Set<Character>`. (Hint: this is a one-liner. Check the `HashMap` documentation.)

4. Write an non-`static` inner class of `LSystem` named `LSystemEvolution`. It encapsulates an evolution of an `LSystem`. Because it is a non-`static` inner class of `LSystem`, each instance of `LSystemEvolution` is associated with an instance of `LSystem`. Therefore, it has direct access to any instance variables of the enclosing `LSystem`. You *do not* need to create instance variables for theta, productions, or initial state. That would be redundant. Using the enclosing system's values. `LSystemEvolution` has the following:

(a) A constructor that starts this evolution off in stage 0. At stage 0, the state of the evolution is the system's initial state.

(b) A method `toString` that returns the state of the evolution as a `String`.

(c) A method `getStage` that returns the stage of evolution as an `int`. An evolution starts off at stage 0, but other methods may increase or decrease this number.

(d) A method `evolve` that advances the evolution to the next stage. The new state is formed by replacing each symbol in the originating state with its replacement text, as specified by the productions. For example, if the originating state is `F+F` and we have production `F` $\rightarrow$ `fF-Ff`, the new state is `fF-Ff+fF-Ff`.

(e) A method `evolveTo` that accepts an evolutionary stage as an `int`. It moves this evolution into the given stage. (Hint: use `evolve`, if you can. If you can't, make it so you can.) If the stage value is less than 0, throw an `IllegalArgumentException`.

(f) A method `reevolve` that starts the evolution back to the starting state and evolves it again to its current stage. What's the point of this method? Suppose the system's initial state, theta, or productions have changed. Since our evolution was arrived at using the old initial state, theta, and productions, we'd like to have some way to make the evolution reflect those changes. By reevolving under the new system, we'll see those changes.

(g) A method `draw` that accepts a `Graphics2D` parameter and draws the evolution in its current state. The turtle that traces out the drawing is initially at $(0, 0)$ with a heading of 90 degrees (headed to the top of your screen). Process each character/command in the current evolution to move the turtle around. The only characters in the current state that you need to consider when drawing are the following:

- `+` Turn the turtle clockwise by theta, i.e., decrease its heading by theta.
- `-` Turn the turtle counter-clockwise by theta, i.e., increase its heading by theta.
- `f` Move the turtle one unit in its current heading, but do not draw a line marking the turtle's trail.
- `g` Same as `f`. This copycat command exists to facilitate certain tricks in the productions.
- `F` Move the turtle one unit in its current heading, and draw a line marking the turtle's trail from its old position to its new. Lines can be drawn by casting your `Graphics` object to a `Graphics2D` object and issuing a call like `g2.draw(new Line2D.Double(x1, y1, x2, y2)`.
- `G` Same as `F`.
- `[` Push a copy of the current turtle onto the turtle stack, so that it may be restored later. This allows one to branched drawing—after finishing one part of a fork, we can easily return to the bifurcation point with a `]`.
- `]` Pop a turtle off of the turtle stack and use it for subsequent drawing.

This method also returns a `Rectangle2D.Double` representing the bounds of the drawing. Through it, one can determine the minimum and maximum x and y coordinates that you visited as the evolution was drawn. These bounds are used by `LSystemViewer` to center and scale your creations to fit neatly in the canvas. (Hint: to construct this rectangle, maintain minima and maxima for the x and y coordinates. Every time you update the turtle's position during the drawing, see if you need to update these values.)

Since `LSystemEvolution` is an inner class, you will first need an instance of `LSystem` in order to make an instance of `LSystemEvolution`. To create an instance of an inner class outside of the outer class, you must qualify the `new` operator:

```
LSystem lsystem = new LSystem("F+F", 90);
lsystem.addProduction('F', "F-FF");
LSystemEvolution evolution = lsystem.new LSystemEvolution();
```

4 Submission

This homework is part of a regular assignment and is graded by hand and with help from the SpecChecker. Your assignment is expected to fully meet the requirements above and the following:

1. Variable names should be meaningful and accurate.
2. Non-obvious parts of your code should be commented.
3. Code should be cleanly formatted and indented.

Your work will also be inspected for plagiarism. Please do your own work. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. (If you find violators of this rule, please let me know.) Write your own code.

Put the SpecChecker in your Build Path. Run it as a Java Application (not a JUnit Test) and fix problems until all tests pass. Upload the resulting ZIP file to the W drive.

5 Files

- `http://www.twodee.org/teaching/cs245/2013C/homework/lindenmayer/speccheck_hw3.jar`
- `http://www.twodee.org/teaching/cs245/2013C/homework/lindenmayer/LSystemViewer.java`