DCFL Project part c: Lindenmayer System

- Due Friday, December 4.
- Input:
  - Axiom and Rules of a Deterministic Context Free Language.
  - Number of Generations (>= 0)
  - An Initial Angle (in degrees)
  - A Turn Angle (in degrees)
- Output:
  - A frame that contains a specific graphical interpretation of the string generated from the input.
Lindenmayer System API

- **class DCFL extends JFrame**

  Constructor:
  
  ```java
  DCFL(String axiom, String[] rules)
  // same as part 1
  ```

- **Methods:**
  
  ```java
  String produce(int level) // same as part 2
  void setInitialAngle(double degrees)
  // [0, 360] where 0 is right, 90 is up, 180 left, ...
  void setTurnAngle(double degrees)
  // [0, 180] the angle is degrees right of current heading.
  void draw(String lindenmayerString)
  void draw()
  ```

Turtle Graphics

- The turtle moves with commands that are relative to its own position, such as:
  1. "Move forward 10 spaces" and
  2. "Turn left 90 degrees".

- The Turtle moves on a 2D surface and carries a pen.

- The state of the turtle has three attributes:
  1. Position
  2. Orientation
  3. Color

  ![Turtle Graphics Examples]
  
  - Forward 1
  - Right 90° Forward 1
  - Right 90°
**L-System – Graphical Interpretation of Symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>Draw a straight line in the direction of the current heading. $(x, y, \theta, c) \rightarrow (x', y', \theta, c)$, where $x' = x + d \cos(\theta)$ and $y' = y + d \sin(\theta)$.</td>
</tr>
<tr>
<td>h</td>
<td>Same as f.</td>
</tr>
<tr>
<td>g</td>
<td>Same as f except no line is drawn.</td>
</tr>
<tr>
<td>k</td>
<td>Draw line with length a function of the symbol's age. The k symbol cannot be a variable.</td>
</tr>
<tr>
<td>+</td>
<td>Turn clockwise by angle $\delta$. $(x, y, \theta, c) \rightarrow (x, y, \theta + \delta, c)$.</td>
</tr>
<tr>
<td>-</td>
<td>Turn counter-clockwise by angle $\delta$. $(x, y, \theta, c) \rightarrow (x, y, \theta - \delta, c)$.</td>
</tr>
<tr>
<td>[</td>
<td>Push current state, $(x, y, \theta, c)$, onto stack.</td>
</tr>
<tr>
<td>]</td>
<td>Pop current state, $(x, y, \theta, c)$, from stack.</td>
</tr>
<tr>
<td>a, b, c...e</td>
<td>Change the color $(x, y, \theta, c) \rightarrow (x, y, \theta, c')$.</td>
</tr>
</tbody>
</table>

**LSystem Dragon Curve Example**

```java
String axiom = "f";
String[] rules = {"f=f-h", "h=f+h"};
DCFL dcfl = new DCFL(axiom, rules);
String dragon0 = dcfl.produce(0);
dcfl.setInitialAngle(90.0);
dcfl.setTurnAngle(90.0);
dcfl.draw(dragon0);
Thread.sleep(10000);
String dragon1 =
    dcfl.produce(1);
dcfl.draw(dragon1);
```

| Gen 0: f | Gen 1: f-h |
Dragon Curve – Generation 2 and 3

2: \[ f-h-f+h \]

3: \[ f-h-f+h-f-h+f+h \]

Dragon Curve – Generation 4 and 5

4: \[ f-h-f+h-f-h+f+h-f-h-f+h+f-h+f+h \]
Dragon Curve (gen 3 through 7, & 12)

Frame Size

- The initial outside frame size must be 800 x 500 pixels.
- The drawing must be scaled, without distortion, to the frame size with a few pixels of blank space around the drawing.
- When the user resizes the frame, the drawing must be resized.
Moving the Turtle

The graphical interpretations of $f$, $g$, and $h$ use the equations:

\[
\begin{align*}
    x' &= x + d \cos \theta \\
    y' &= y + d \sin \theta
\end{align*}
\]

Map World Coordinates to Screen

1. World coordinates are floating point numbers, screen coordinates are integer, pixel values.
2. In the world coordinate system, let $x_0=0, y_0=0, d=1$.
3. **Pass 1 (No Drawing):** Walk through the command string updating $(x,y)$, and keeping track of the minimum and maximum values of both $x$ and $y$ throughout the path.
4. Using the extreme values of $x$ and $y$ from step 3, calculate the scaling factor and the $x$ and $y$ offsets for mapping world coordinates to screen coordinates.
5. **Pass 2 (Draw):** Walk through the command string a second time recalculating all of the same $(x,y)$ to $(x',y')$ pairs. Additionally, convert each set of points to screen coordinates and render each line.
Example Showing World Coordinates

\[(x_7, y_7) = ?\]
\[(x_8, y_8) = ?\]
\[(x_9, y_9) = ?\]

\[MinX = ?\]
\[MaxX = ?\]
\[MinY = ?\]
\[MaxY = ?\]

Mapping World Coordinates to the Screen

\[rangeX = maxX - minX = 0 - (-2) = 2\]

\[rangeY = maxY - minY\]
\[= 1 - (-2) = 3\]
**Step 1: Shift Real Coordinates to Positive Values**

- \((-1,1)\), \((0,1)\), \((0,0)\), \((-2,-2)\)
- \(\text{rangeX} = 2.0\)
- \(\text{rangeY} = 3.0\)
- \(\text{shiftX} = -\text{minX} = 2.0\)
- \(\text{shiftY} = -\text{minY} = 2.0\)

<table>
<thead>
<tr>
<th>Real Coordinates</th>
<th>Shifted Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>((0.0, 0.0))</td>
<td>((2.0, 2.0))</td>
</tr>
<tr>
<td>((0.0, 1.0))</td>
<td>((2.0, 3.0))</td>
</tr>
<tr>
<td>((-1.0, 1.0))</td>
<td>((1.0, 3.0))</td>
</tr>
<tr>
<td>((-2.0, -2.0))</td>
<td>((0.0, 0.0))</td>
</tr>
</tbody>
</table>

**Step 2: Find the Scaling Factor**

- \(\text{rangeX} = 2.0\)  \(\text{pixelWidth} = 50\)
- \(\text{rangeY} = 3.0\)  \(\text{pixelHeight} = 50\)
- \(\text{scaleX} = \frac{\text{pixelWidth}}{\text{rangeX}} = 25.0\)
- \(\text{scaleY} = \frac{\text{pixelHeight}}{\text{rangeY}} = 16.67\)
- \(\text{scale} = \text{Math.min}(\text{scaleX}, \text{scaleY})\)

<table>
<thead>
<tr>
<th>Real Coordinates</th>
<th>Shifted and Scaled Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>((0, 0))</td>
<td>((2.0, 2.0) \times 16.67 = (33, 33))</td>
</tr>
<tr>
<td>((0, 1))</td>
<td>((2.0, 3.0) \times 16.67 = (33, 50))</td>
</tr>
<tr>
<td>((-1, 1))</td>
<td>((1.0, 3.0) \times 16.67 = (17, 50))</td>
</tr>
<tr>
<td>((-2, -2))</td>
<td>((0.0, 0.0) \times 16.67 = (0, 0))</td>
</tr>
</tbody>
</table>
Visualizing: Steps 1 and 2

Both dimensions were scaled by the smaller of \( \text{scale}_X \) and \( \text{scale}_Y \). Therefore:

- The Y-dimension fills the draw area.
- The X-dimension does not.

Step 3: Centering

The mapping thus far gives:

\[
(\text{minX}+\text{shiftX})\times\text{scale} = 0
\]
\[
(\text{minY}+\text{shiftY})\times\text{scale} = 0
\]

Let

\[
\text{xx} = (\text{maxX}+\text{shiftX})\times\text{scale}
\]
\[
\text{yy} = (\text{maxY}+\text{shiftY})\times\text{scale}
\]

\[
\text{extraSpace}_X = \text{pixelWidth} - \text{xx}
\]
\[
\text{extraSpace}_Y = \text{pixelHeight} - \text{yy}
\]

\[
\text{leftBorder} = \text{extraSpace}_X/2
\]
\[
\text{rightBorder} = \text{extraSpace}_Y/2
\]
Transformation Equations for Steps 1, 2 & 3

The mapping thus far gives:

\[
\text{screenX} = (\text{realX} + \text{shiftX}) \times \text{scale} + \text{leftBorder}
\]

\[
\text{invertY} = (\text{realY} + \text{shiftY}) \times \text{scale} + \text{rightBorder}
\]

Finally, to flip the image to conform to the upside-down computer coordinate system:

\[
\text{screenY} = \text{pixalHeight} - \text{invertY}
\]

Angle Measures

- Input angles will be in degrees.
- \text{Math.sin( double angRadians)}
- \text{Math.cos( double angRadians)}
- \text{Math.toRadians( double angdeg)}
Java, Doubles and Trigonometry

What is going on here? Math.sin works fine. My code is attached.

```
public static void main(String[] args)
{
    System.out.println(Math.PI);
    System.out.println(Math.cos(Math.PI/2));
    System.out.println(Math.sin(Math.PI/2));
}
```

Output:
3.141592653589793
6.123233995736766E-17
1.0

3.141592653589793 is only an approximation of π.
6x10^-17 is approximately zero.

L-System: Open Dragon generation 7

```
String axiom = "f";
String[] rules =
{  "f=f-h",
   "h=f+h"
};
DCFL dclf =
    new DCFL(axiom, rules);
String gen7 = dclf.produce(7);
dclf.setInitialAngle(90.0);
dclf.setTurnAngle(80.0);
dclf.draw(gen7);
```

Generation 7
L-System: Open Dragon generation 12

```java
String axiom = "f";
String[] rules = {
    "f=f-h",
    "h=f+h"
};
DCFL dcf1 = new DCFL(axiom, rules);
String gen = dcf1.produce(12);
dcf1.setInitialAngle(90.0);
dcf1.setTurnAngle(80.0);
dcf1.draw(gen);
```

Sierpiński Triangle

```java
String axiom = "f";
String[] rules = {
    "f = h - f - h",
    "h = f + h + f"
};
DCFL dcf1 = new DCFL(axiom, rules);
String gen = dcf1.produce(9);
dcf1.setInitialAngle(0.0);
dcf1.setTurnAngle(60.0);
dcf1.draw(gen);
```

The Sierpiński triangle is a fractal named after Wacław Sierpiński who described it in 1915. It was originally constructed as a curve (as it is with this L-system). It can also be constructed using the "Chaos Game", or by using an Iterated Function System.
Koch Snowflake

String axiom = "f++f++f";
String[] rules = {
   "f=af-cf++f-af", "a=", "c=");
DCFL dcfl = new DCFL(axiom, rules);
String gen = dcfl.produce(2);
dcfl.setInitialAngle(90.0);
dcfl.setTurnAngle(60.0);
dcfl.draw(gen);

Koch's method of construction:
1. Start with an equilateral triangle.
2. Divide each segment into three segments of equal length.
3. Draw a new equilateral triangle that has the middle segment from step 2 as its base and points outward.
4. Remove the line segment that is the base of the triangle from step 2.

Koch Snowflake with Mitsubishi Diamonds

String axiom = "af++f++f----bh++h++h";
String[] rules = {
   "f=f-f++f-f", "h=h+h--h+h"
};
DCFL dcfl = new DCFL(axiom, rules);
String gen = dcfl.produce(3);
dcfl.setInitialAngle(0.0);
dcfl.setTurnAngle(60.0);
dcfl.draw(gen);

Note: The colors of this drawing, as with all lines drawn by f and h symbols, are not specified. Colors a and b can be any colors that are different from each other and different from the background.
Space-Filling Peano Curve (gen 1-3)

```
String[] rules =
{ "f = a f + c h + + h - a f - - f f - c h +",
  "h = - a f + c h h + + h + a f - - f - c h",
  "a =",
  "c =" }
;
DCFL dcf1 = new DCFL("f", rules);
String gen = dcf1.produce(3);
dcf1.setInitialAngle(90.0);
dcf1.setTurnAngle(60.0);
```

Space-Filling Peano Curve (gen 4-5)

Space-filling curves or Peano curves are curves, first described by Giuseppe Peano (1858 – 1932), whose ranges contain the entire 2-dimensional unit square (or the 3-dimensional unit cube). The idea of a 1-dimensional object being space filling was found to be highly counterintuitive.

In their limit, the Koch Snowflake, the Sierpiński Triangle, and Peano curves all have infinite length, are continuous, and are nowhere differentiable.
Set Java VM Maximum Heap Size

- When a Java program runs, the application variable space is allocated from a block of memory called the heap.
- Garbage Collection returns variables space that is no longer referenced to the heap.
- By default, the Java Virtual Machine starts up with a maximum heap size of 128 Megabytes.
- For security reasons, once the Java Virtual Machine starts, its maximum heap size cannot be increased.
- The maximum heap size of the Java Virtual Machine can be specified at startup with the Java VM option: `-mx500M`
- Where `-mx500M` sets the heap size to 500 Megabytes.

Grading L-System Part 3

<table>
<thead>
<tr>
<th>Test Description</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>All source code is present and submission is easy to run and test.</td>
<td>3</td>
</tr>
<tr>
<td>Code is neat and well commented.</td>
<td>3</td>
</tr>
<tr>
<td>Dragon Curve (level 0, 1, 2, 3, 4)</td>
<td>5</td>
</tr>
<tr>
<td>Dragon Curve (level 12)</td>
<td>3</td>
</tr>
<tr>
<td>Open Dragon (level 7 and 12)</td>
<td>4</td>
</tr>
<tr>
<td>Sierpiński Triangle (level 9)</td>
<td>2</td>
</tr>
<tr>
<td>Koch Snowflake (level 1, 2, 3, and 4)</td>
<td>8</td>
</tr>
<tr>
<td>Koch Snowflake with Mitsubishi (level 4)</td>
<td>2</td>
</tr>
<tr>
<td>Peano Curve (levels 1, 2, 3, 4, and 5)</td>
<td>10</td>
</tr>
<tr>
<td>Image is sized to fit frame</td>
<td>2</td>
</tr>
<tr>
<td>Image resizes as frame is resized</td>
<td>3</td>
</tr>
<tr>
<td>Image is centered when frame is resized</td>
<td>5</td>
</tr>
</tbody>
</table>

Total Points: 50

- Any test that does not complete after 5 minutes on Torin's PC is considered to have failed.
- Any test that runs out of memory fails.
- Mirror images of given output receive full credit.
### L-System: Age, Stack and Color

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>Draw a straight line in the direction of the current heading. $(x, y, \theta, c) \rightarrow (x', y', \theta, c)$, where $x' = x + d \cos(\theta)$ and $y' = y + d \sin(\theta)$.</td>
</tr>
<tr>
<td>h</td>
<td>Same as f.</td>
</tr>
<tr>
<td>g</td>
<td>Same as f except no line is drawn.</td>
</tr>
<tr>
<td>k</td>
<td>Draw line with length a function of the symbol's age. The k symbol cannot be a variable.</td>
</tr>
<tr>
<td>+</td>
<td>Turn clockwise by angle $\delta$. $(x, y, \theta, c) \rightarrow (x, y, \theta + \delta, c)$.</td>
</tr>
<tr>
<td>-</td>
<td>Turn counter-clockwise by angle $\delta$. $(x, y, \theta, c) \rightarrow (x, y, \theta - \delta, c)$.</td>
</tr>
<tr>
<td>[</td>
<td>Push current state, $(x, y, \theta, c)$, onto stack.</td>
</tr>
<tr>
<td>]</td>
<td>Pop current state, $(x, y, \theta, c)$, from stack.</td>
</tr>
<tr>
<td>a, b, c...e</td>
<td>Change the color $(x, y, \theta, c) \rightarrow (x, y, \theta, c')$.</td>
</tr>
</tbody>
</table>

---

### void setGrowth(double growth)

In addition to the constructor and methods needed in part 3, part 4 requires

```java
void setGrowth(double growth)
```

Where the value of growth must be no less than 0.1 and no greater than 10.
L-System – k Symbol

Every new k symbol moves the turtle the same distance as does every f, g, and h symbol (therefore, in the equation below a new k symbol has age=0).

Every generation that a k symbol ages, its length is multiplied by the value of growth (=age^s where s, the growth value, is an input parameter).

Thus, a growth value > 1 makes older k symbols longer, and a growth value < 1 makes older k symbols shorter.

The color drawn by a k is also a function of its age: New k symbols are drawn in light greens. Older k symbols are drawn with darker greens and the oldest k symbols are drawn in browns.

\[(x, y, \theta, c) \rightarrow (x', y', \theta, c)\]

where

\[x' = x + s^{age}d \cos \theta \quad y' = y + s^{age}d \sin \theta\]

s is the growth value, and age is the age of the k symbol.

L-System Color Specifications

- There must be at least 6 different colors from dark brown to light green for k symbols of different ages.
- If you have defined more generation colors for k than there are generations in a given run, then use any subset of your generation colors.
- The colors used for a, b, c, d and e can be hard coded, and may be any set of 5 distinct colors.
- When drawing a k symbol, its age overrides any color set by a, b, c, d, or e, but does not change the color state of the turtle.
- Background color is not specified.
Purple Yarrow L-System

Axiom: \( ef \)
Rules: \( f=k[+f][-f] \)
Initial Angle: 90°
Turn Angle: 20°
Growth: 2.0

Gen 1 has a \( k \) of age 0. This \( k \) is green and the same length as an \( f \).

Gen 2 has a one gen old \( k \) and two new \( k \) symbols. Each new \( k \) is green and the same length as an \( f \). The old \( k \) is brown and twice as long as the \( f \).

Purple Yarrow L-System – \( k \) and Age

Axiom: \( ef \)
Rules: \( f=k[+f][-f] \)

Color Key: age=0, age=1, age=2, age=3

Gen 1: \( ek[+f][-f] \)

Gen 2: \( ek[+k[+f][-f]][-k[+f][-f]] \)

Gen 3: \( ek[+k[+k[+f][-f]][-k[+f][-f]]][-k[+k[+f][-f]][-k[+f][-f]]] \)

- In generation 1, \( e \) is left over from the axiom and is now one generation old.
- The \( f \) from the axiom was replaced with the red text: \( k[+f][-f] \) – all zero generations old.
- Variables are replaced each generation, thus they never age.
Purple Yarrow L-System – k and Age

Axiom: ef
Rules: f=k[+f][-f]

Gen 1: ek[+f][-f]
Gen 2: ek[+k[+f][-f]][-k[+f][-f]]
Gen 3: ek[+k[+k[+f][-f]][-k[+f][-f]]][-k[+k[+f][-f]][-k[+f][-f]]]

All f symbols should be drawn one unit long and in the color of the current state of the turtle (in this case color e).
k symbols of age 0: growth⁰ = 1 unit long in light green.
k symbols of age 1: growth¹ = 2 units long in medium green.
k symbols of age 2: growth² = 4 units long in dark green.

L-System Purple Yarrow: Gens 3 & 8
L-System Test Case: 45-45-90 Tree

axiom: xy
Rules:
  y = [-xy] [zxy]
  z = +k
  x = k [++k++k++k]
Initial Angle: 90°
Turn Angle: 45°
Growth: $\sqrt{2}$
Generations 2

Symbols of age 0 are colored black,
0: xy
1: k [++k++k++k] [-xy] [zxy]
2: k [++k++k++k] [-k [++k++k++k] [-xy] [zxy]] [+kk[++k++k++k] [-xy] <zxy]]
L-System: 45-45-90 Tree – Generation 4

axiom: xy
Rules:
\[ y = [-xy][zxy] \]
\[ z = +k \]
\[ x = k[+++k++k++k] \]
Initial Angle: 90°
Turn Angle: 45°
Growth: $\sqrt{2}$
Generations 4

L-System: 45-45-90 Tree – Generation 5

axiom: xy
Rules:
\[ y = [-xy][zxy] \]
\[ z = +k \]
\[ x = k[+++k++k++k] \]
Initial Angle: 90°
Turn Angle: 45°
Growth: $\sqrt{2}$
Generations 5
L-System: 45-45-90 Tree – Generation 9

axiom: xy
Rules:

\[
y = [-xy][zxy]
z = +k
x = k[++k++k++k]
\]

Initial Angle: 90°
Turn Angle: 30°
Growth: \( \sqrt{2} \)
Generations 12

L-System: 30-60-90 Tree

axiom: xy
Rules:

\[
y = [-xy][zxy]
z = +k
x = k[++k++k++k]
\]

Initial Angle: 90°
Turn Angle: 30°
Growth: \( \sqrt{2} \)
Generations 12
L-System Output: Fern (Generation 1, 2 & 3)

Axiom: f
Rules:
f=k[+f][--f]k[+++f][--------f]--[f]--f]--f]-f

Initial Angle: 90°
Turn Angle: 8°
Growth: 2.5

Note: The main stems drawn with k symbols and need to show color aging. The tips are drawn with f and can be any color (except the background color). All of the tips must be the same color.

L-System: Fern – generation 5

Axiom: f
Rules:
f=k[+f][--f]k[+++f]
   k[--------f]--f]
   k[+f][-f]-kf

Initial Angle: 90°
Turn Angle: 8°
Growth: 2.5
Generations: 5
L-System: Grass

Axiom: \( x \)
Rules:
- \( x=df-[x]+x+f[+afx]-x \)
- \( f=ff \)
Initial Angle: 65.0°;
Turn Angle: 25.0°;
Generations: 6

Note: There are no \( k \) symbols used in this system. Thus, all color comes from the \( a \) and \( d \) symbols. \( k \) used light brown and green, but they can be any colors that are different from each other and different from the background.

Turtle class (gets pushed and popped)

```java
class Turtle
{
    public double x=0;
    public double y=0;
    public char c='a';
    public double heading = 0;

    public void setTurtle (Turtle source)
    {
        this.x = source.x;
        this.y = source.y;
        this.heading = source.heading;
        this.c = source.c;
    }
}
```