Aristid Lindenmayer was a Hungarian biologist who, in 1968, invented an elegant mathematical system for describing the growth of plants and other multicellular organisms. This type of system, now called a Lindenmayer system, is a particular type of formal grammar.

Formal grammars

A formal grammar defines a set of productions (or rules) for constructing strings of characters. For example, the following very simple grammar defines three productions that allow for the construction of a handful of English sentences.

\[
S \rightarrow N \ V \\
N \rightarrow \text{our dog} \mid \text{the school bus} \mid \text{my foot} \\
V \rightarrow \text{ate my homework} \mid \text{swallowed a fly} \mid \text{barked}
\]

The first production, \( S \rightarrow N \ V \) says that the symbol \( S \) (a special start symbol) may be replaced by the string \( N \ V \). The second production states that the symbol \( N \) (short for “noun phrase”) may be replaced by one of three strings: our dog, the school bus, or my foot (the vertical bar (\( | \)) means “or”). The third production states that the symbol \( V \) (short for “verb phrase”) may be replaced by one of three other strings. The following sequence represents one way to use these productions to derive a sentence.

\[
S \Rightarrow N \ V \Rightarrow \text{my foot} \ V \Rightarrow \text{my foot swallowed a fly}
\]

The derivation starts with the start symbol \( S \). Using the first production, \( S \) is replaced with the string \( N \ V \). Then, using the second production, \( N \) is replaced with the string “my foot”. Finally, using the third production, \( V \) is replaced with “swallowed a fly.”

Formal grammars were invented by linguist Noam Chomsky in the 1950’s as a model for understanding the common characteristics of human language. Formal grammars are used extensively in computer science to both describe the syntax of programming languages and as formal models of computation. As a model of computation, a grammar’s productions represent the kinds of operations that are possible, and the resulting strings, formally called words, represent the range of possible outputs. The most general formal grammars, called unrestricted grammars are computationally equivalent to Turing machines. Putting restrictions on the types of productions that are allowed in grammars affects their computational power. The standard hierarchy of grammar types is now called the Chomsky hierarchy.

A Lindenmayer system (or L-system) is a special type of grammar in which
(a) all applicable productions are applied in parallel at each step, and
(b) some of the symbols represent turtle graphics drawing commands.

The parallelism is meant to mimic the parallel nature of cellular division in the plants and multicellular organisms that Lindenmayer studied. The turtle graphics commands represented by the symbols in a derived string can be used to draw the growing organism.

Instead of a start symbol, an L-system specifies an axiom where all derivations begin. For example, the following grammar is a simple L-system:

Axiom: F
Production: F → F-F++F-F

We can see from the following derivation that parallel application of the single production very quickly leads to very long strings:

\[
F \Rightarrow F-F++F-F \Rightarrow F-F++F-F-F-F++F-F-F-F++F-F++F-F+\cdot\cdot\cdot
\]

In the first step of the derivation, the production is applied to replace F with F-F++F-F. In the second step, all four instances of F are replaced with F-F++F-F. The same process occurs in the third step, and the resulting string grows very quickly. The number of strings generated from the axiom in a derivation is called the depth of the derivation. If we stopped the above derivation after the last string shown, then its depth would be four because four strings were generated from the axiom.

Repeated applications of the production(s) can be framed as a recursive algorithm. Like a recursive solution, we are applying the same algorithm (“apply all applicable productions”) at each stage of a derivation. At each stage, the inputs are growing larger, and we are getting closer to our desired depth. In other words, we can think of each step in the derivation as applying the algorithm to a longer string, but with the depth decreased by one. For example, in the derivation above, applying a derivation with depth four to the axiom F is the same as applying a derivation with depth three to F-F++F-F, which is the same as applying a derivation of length two to the next string, etc. In general, applying a derivation with depth d to a string is the same as applying a derivation with depth d − 1 to that string after the productions have been applied one time.
Part 1 Write a recursive function

```java
string derive(string str, Map<char, string> productions, int depth)
```

that returns the result of `depth` applications of the productions in the Map named `productions` to the string named `str`.

As mentioned above, each symbol in an L-system represents a turtle graphics command:

- F means “move forward”
- - means “turn left”
- + means “turn right”

Interpreted in this way, every derived string represents a sequence of instructions for a turtle to follow. The distance moved for an F symbol can be chosen when the string is drawn. But the angle that the turtle turns when it encounters a - or + symbol must be specified by the L-system. For the L-system above, we will specify an angle of 60 degrees:

Axiom: F
Production: F → F-F++F-F
Angle: 60 degrees

An annotated sketch of the first string derived from this L-system (F-F++F-F) is shown below.

Starting on the left, we first move forward. Then we turn left 60 degrees and move forward again. Next, we turn right twice, a total of 120 degrees. Finally, we move forward, turn left again 60 degrees, and move forward one last time. Does this look familiar? As shown in Figure 1, the strings derived from this L-system produce Koch curves. Indeed, Lindenmayer systems produce fractals!
Here is another example:

Axiom: \( FX \)

Productions:
- \( X \rightarrow X-YF \)
- \( Y \rightarrow FX+Y \)

Angle: 90 degrees

This L-system produces a well-known fractal known as a dragon curve, shown in Figure 2.

**Part 2** Write a function

\[
\text{void drawLSystem(string str, float angle, float distance, Point position, float heading)}
\]

that draws the picture described by the given L-system string. Your function should correctly handle the special symbols \( F \), \( + \), and \( - \), but ignore any other symbols. The parameters \text{angle} and \text{distance} give the angle (in degrees) that the “turtle” turns in response to a \( + \) or \( - \) command, and the distance the “turtle” draws in response to an \( F \) command, respectively. The last two parameters specify the initial \text{position} and \text{heading} of the “turtle,” before drawing commences.

Obviously, we do not have a turtle graphics library available to us. Instead, you will need to mimic the turtle graphics commands with the \text{drawLine} method of \text{GWindow}, and a little trigonometry.

**Part 3** Apply your \text{drawLSystem} function to each of the following strings:

1. \( F-F++F++F-F++F-F++F-F++F-F++F-F++F-F \) (\text{angle} = 60 degrees, \text{distance} = 20)
2. \( FX-YF-FX+YF-FX-YF+FX+YF-FX-YF-FX+YF+FX-YF+FX+YF \) (\text{angle} = 90 degrees, \text{distance} = 20)

**Part 4** Write a function

\[
\text{void lsystem(string axiom, Map<char, string> productions, int depth, float angle, float distance, Point position, float heading)}
\]
that calls the `derive` function with the first three parameters, and then calls your `drawLSystem` function with the new string and the last four parameters. This function combines all of your previous work into a single L-system generator.

**Part 5** Call your `lsystem` function on each the following L-systems. (The position values assume an 800 × 800 window. Adjust as appropriate.)

1. **Axiom:** F 
   **Production:** F → F-F++F-F 
   **Angle:** 60 degrees 
   distance = 10, position = (700, 400), heading = 0, depth = 4

2. **Axiom:** FX 
   **Productions:** X → X-YF, Y → FX+Y 
   **Angle:** 90 degrees 
   distance = 5, position = (400, 400), heading = 0, depth = 12

3. **Axiom:** F-F-F-F 
   **Production:** F → F-F+F+FF-F-F+F 
   **Angle:** 90 degrees 
   distance = 3, position = (500, 300), heading = 0, depth = 3

4. **Axiom:** F-F-F-F 
   **Production:** F → FF-F-F-F-F-F+F 
   **Angle:** 90 degrees 
   distance = 5, position = (400, 600), heading = 0, depth = 3

Aristid Lindenmayer was specifically interested in modeling the branching behavior of plants. To accomplish this, we need to introduce two more symbols: `[` and `]`. For example, consider the following L-system:

**Axiom:** X 
**Productions:** X → F[-X]+X, F → FF 
**Angle:** 30 degrees 

These two new symbols involve the use of a stack. In a Lindenmayer system, the `[` symbol represents a push operation and the `]` symbol represents a pop operation. More specifically,

- `[` means “push the turtle’s current position and heading on a stack,” and
- `]` means “pop a position and heading from the stack and set the turtle’s current position and heading to these values.”
Let’s now return to the Lindenmayer system above. Applying the productions of this Lindenmayer system twice results in the following string.

\[ X \Rightarrow F[-X]+X \Rightarrow FF[-F[-X]+X]+F[-X]+X \]

The X symbols are used only in the derivation process and do not have any meaning for turtle graphics, so we simply skip them when we are drawing. So the string \( FF[-F[-X]+X]+F[-X]+X \) represents the simple “tree” below. On the left is a drawing of the tree; on the right is a schematic we will use to explain how it was drawn.

The turtle starts at the origin, marked \( O \), with a heading of 90 degrees (north). The first two F symbols move the turtle forward from the origin to point \( a \) and then point \( b \). The next symbol, [, means that we push the current position and heading (\( b \), 90 degrees) on the stack.

The next two symbols, -F, turn the turtle left 30 degrees (to a heading of 120 degrees) and move it forward, to point \( c \). The next symbol is another [, which pushes the current position and heading, (\( c \), 120 degrees), on the stack. So now the stack contains two items—(\( b \), 90 degrees) and (\( c \), 120 degrees)—with the last item on top.

The next three symbols, -X], turn the turtle left another 30 degrees (to a heading of 150 degrees), but then restore its heading to 120 degrees by popping (\( c \), 120 degrees) from the stack.

The next three symbols, +X], turn the turtle 30 degrees to the right (to a heading of 90 degrees), but then pop (\( b \), 90 degrees) from the stack, moving the turtle back to point \( b \), heading north.
Figure 3: Two trees from *The Algorithmic Beauty of Plants* (p. 25).

(So, in effect, the previous six symbols, \([-X]+X\) did nothing.) The next two symbols, \(+F\), turn the turtle 30 degrees to the right (to a heading of 60 degrees) and move it forward to point \(d\). Similar to before, the last six symbols, \([-X]+X\), while pushing states onto the stack, have no visible effect.

Continued applications of the productions in the L-system above will produce strings that draw a sequence of increasingly complex trees. More involved L-systems will produce much more interesting trees. For example, the following two L-systems produce the trees in Figure 3.

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Productions</th>
<th>Angle:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X)</td>
<td>(X \rightarrow F-[[X]+X]+F[+FX]-X) (F \rightarrow FF)</td>
<td>25 degrees</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Production</th>
<th>Angle:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(F)</td>
<td>(F \rightarrow FF-[-F+F+F]+[+F-F-F])</td>
<td>22.5 degrees</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Part 6** Augment the `drawLSystem` function so that it correctly draws L-system strings containing the \([\) and \(]\) characters. Do this by incorporating a single stack into your function, as we described above. Test your function with the three tree-like L-systems above.

The `drawLSystem` function can be implemented without an explicit stack by using recursion. Think of the `drawLSystem` function as drawing the figure corresponding to a string situated inside matching square brackets.

**Part 7** Write a recursive function

```c
int drawLSystemR(GWindow gw, string str, int startIndex, float angle, float distance)
```

that draws the same figure as your previous `drawLSystem` function, but without an explicit stack. The additional parameter `startIndex` is the index of the first character after the left square bracket (\([\)`. The function will return the index of the matching right square bracket (\(]\). (We can pretend that there are imaginary square brackets around the entire string for the initial call of the function, so we initially pass in 0 for `startIndex`.)
The recursive function will iterate over the indices of the characters in string, starting at startIndex. (Use a while loop, for reasons we will see shortly.) When it encounters a non-bracket character, it should do the same thing it did earlier. When the function encounters a left bracket, it will save the turtle’s current position and heading, and then recursively call the function with startIndex assigned to the index of the character after the left bracket. When this recursive call returns, the current index should be set to the index returned by the recursive call, and the function should reset the turtle’s position and heading to the saved values. When it encounters a right bracket, the function will return the index of the right bracket.

For example, the string below would be processed left to right but when the first left bracket is encountered, the function would be called recursively with index 5 passed in for startIndex.

\[
\text{drawLSystem(..., 5, ...)}
\]

This recursive call will return 26, the index of the corresponding right bracket, and the + symbol at index 27 would be the next character processed in the loop. The function will later make two more recursive calls, marked with the two additional braces above.

**Part 8** Use your program to draw the following additional Lindenmayer systems. For each one, set distance = 5, position = (400, 700) (assuming an 800 × 800 window), heading = 90, and depth = 6.

**Axiom:** X

**Productions:**
- \( X \rightarrow F[+X]F[-X]+X \)
- \( F \rightarrow FF \)

**Angle:** 30 degrees

**Axiom:** H

**Productions:**
- \( H \rightarrow HFX[+H][-H] \)
- \( X \rightarrow X[-FFF][+FFF]FX \)

**Angle:** 25.7 degrees