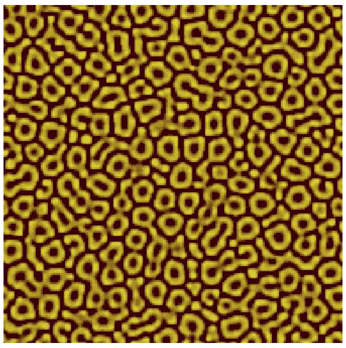
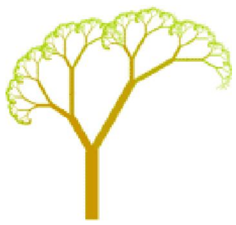
Lindenmayer Systems

Morphogenesis



Morphogenesis is the development of structural features and patterns in organisms. Formal interest in this area probably dates to the 19th century, when it was observed that the phyllotaxis (leaf arrangement) of many plants could be predicted using the ratios of successive Fibonacci numbers. The field received a true jump-start in 1952, however, when Alan Turing published "The Chemical Basis of Morphogenesis" [2], his last publication before his death in 1954. Turing proposed a reaction-diffusion model as a possible basis for the development of many features, including the arrangement of tentacles on the hydra and the patterns of colour seen on animal skin. Turing, more widely known for his work in cryptography and computing science, made use of an early computer (the Ferranti Mark I) to solve the differential equations that formed the basis of this model. (There is an applet on this site that will help you do the same.)

Lindenmayer Systems (L-systems)



The application of computing has continued to play an important role in the study of morphogenesis, although it was the introduction of Lindenmayer systems in 1968 [1] that marked the first use of not only computational power but some of the most fundamental theory of computing science in this field. The eponymic Aristid Lindenmayer was a theoretical biologist who proposed L-systems to formally describe the growth patterns of algae. It was later extended to include the branching structures of plants.

Formally, an L-system is a *language*, which means a set of strings that is made by the application of certain rules. (A string is a sequence of symbols—the text of this page is a string, for example.) Informally, L-systems consist of two important pieces: an axiom, and a set of productions. The axiom is the starting point, like a seed. The set of productions are the rules. By applying these rules, any other string in the language can be produced. When the productions are applied to the axiom, they produce more strings which (along with the axiom) are also in the language of that L-system. The rules can then be applied to each of *those* strings to produce even more strings that are in the language, and so on. The languages of most L-systems (the most useful ones, anyway) contain an infinite number of strings.

Let's look at a simple example:

```
axiom Y
Y → XYX
```

The first line above gives the axiom, and the next two lines are the productions. To apply the productions, you scan a string that is known to be the language. Anywhere you see a pattern on the left side of the → symbol, you replace it with the string on the right. Let's see how that works for this system. Start with the axiom, Y. That is our "order 0" string. When you look at the single production, you see that you can replace the Y with XYX by following the rule Y → XYX. That is the order 1 string. At order 2, you see that again the rule can only be applied in one place. We replace the inner Y of XYX with XYX again, and we get XXYXX. In general, at order *n*, this system will produce a string that consists of *n* X's, a Y, and *n* X's again.

What happens if you add the following production?

```
YX → XY
```

The system starts the same, producing Y and then XYX as before. At the next order, you run into a problem. Two different rules can be applied at the same spot. You can replace the Y with XYX, or you can replace the Y *and* the following X with XY. Which to choose? It turns out that both possibilities are allowed. Systems that allow rules like these tend to become very complex. While this ability can be useful, you can still do a lot of interesting stuff—and with a lot fewer headaches—if you use a subtype of L-system called a d0L-system. This kind of system has some restrictions on it that make it easier (and computationally feasible) to work with: First, the left hand side of a production can only be a single letter. Second, no two productions can have the same left-hand side.

These restrictions ensure that a given string in the language of a d0L-system can only be transformed into exactly *one* other string. No more ambiguity. Our use of the term "order *n* string" makes sense for these systems, because there will only be one of them. Here is an interesting d0L-system:

```
axiom B
A → AB
B → A
```

If you try writing the first eight orders of this system you'll get B, A, AB, ABA, ABAAB, ABAABABA, ABAABABAABAAB and ABAABABAABAABABAABABA. Notice a pattern? Consider the *length* of each of these strings. Can you see why the strings produced by this d0L system are called Fibostrings?

Imagine You're a Turtle

Other than a vague connection with phyllotaxis via the Fibostrings above, you may think we've gotten pretty far off track from describing the development of living organisms. Not so. Aside from producing the Fibostrings, the above system was used by Lindenmayer to characterize the growth patterns of certain kinds of algae. We can make things more interesting, though, by applying a geometric interpretation to the symbols of L-system strings.

This is done using "turtle graphics," an odd name which comes from the Logo programming language. They work like this: Imagine you're a turtle on an infinitely large piece of paper. Not just any turtle, but one that can understand simple instructions and that has a basic understanding of geometry. Finally, imagine that you (the unusually bright turtle, that is) also have a pen in your mouth.

Have you got that image in mind? Good.

Now suppose I start giving you instructions like this:

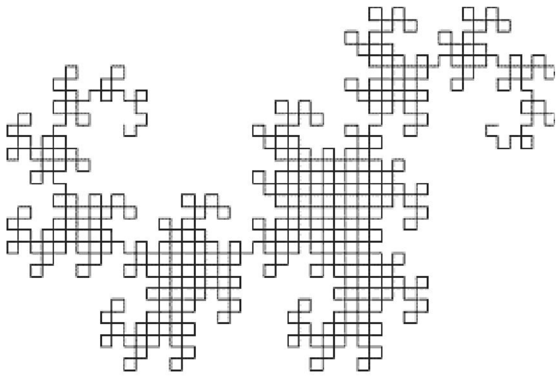
1. Touch the pen to the paper.
2. Move forward 1 step, turn left 90°.
3. Move forward 1 step, turn left 90°.
4. Move forward 1 step, turn left 90°.
5. Move forward 1 step.

In this case, you'd have drawn a square. What a good little turtle you are.

Getting back to our L-system strings, you can assign commands like the ones above to symbols in our L-system. Then by generating a string of a certain order, you have also described instructions for drawing a picture—instructions that can be carried out by our ever-faithful turtle. Here is an example:

```
axiom FX
X → X+YF+
Y → -FX-Y
F → ε
```

The special symbol ε stands for the empty string, which means that we will delete any F's from the result when we rewrite the string. If the turtle interprets F to mean "move forward one step with the pen down" and - and + to mean "turn right 90°" and "turn left 90°", respectively, then the turtle will draw this picture from the order 10 string:

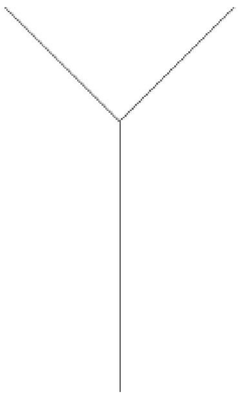

Dragon curve from an L-system

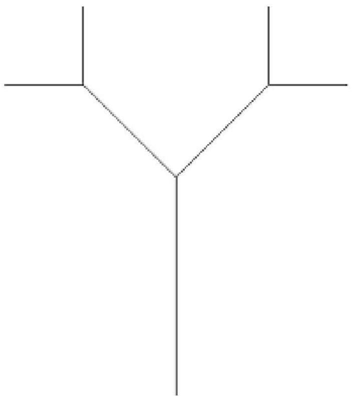
If you are familiar with fractals, you might recognize that picture as the famous Dragon Curve. But, while fractals are interesting enough in their own right, we were talking about morphogenesis. As I said earlier, L-systems came to be applied to the description of plants. To do this, we have to extend the turtle's capabilities a little. First, we will add a way to tell the turtle to change how far it goes in one step. Second, we will have to give the turtle a *stack*. A stack is a way for the turtle to remember what it is doing. It can then go off and do something else for a bit, and when it comes back it will pick up right where it left off. What's more, it can be interrupted in the middle of one of these interruptions, tend to that, and return to the first interruption, tend to that, and *then* return to what it was originally doing, ad infinitum. The device that the turtle uses to keep track of all its jobs is called a stack because, like a stack of dishes, the next item to come off the stack is always the item that was put there most recently. The "something else" that the turtle gets up to will usually be (in the case of plants) a smaller branch that leads off of the current one.

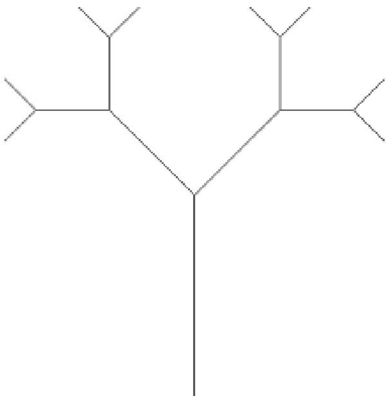
To get us started, here is a very simple *recursive* definition of a tree (recursive means that it is defined in terms of itself):

1. A tree is a trunk with two shorter branches coming out of the top of it at a right angle to each other.
2. A branch is a tree.

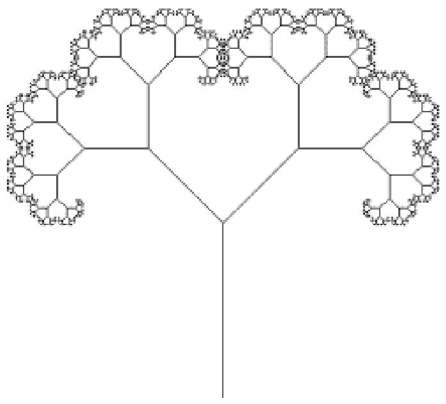
Let's make a tree from this definition:


A tree is a trunk with two branches...


Each branch is a tree...


Each of those branches is a tree...

After several more steps we'll get this:



Which is indeed rather tree-like. If you noticed the similarity between this process and replacing the symbols to rewrite an L-system at a higher order, then pat yourself on the back. All we need is a symbol for the "shorter" part of the definition, and a way to represent starting and stopping the interruptions ("pushing" and "popping" the stack), and we can write the above definition in L-system form. We'll use S, [, and] for these, respectively, and we'll redefine - and + to turns of 45° rather than 90°. Then we have:

```
axiom FX
X → S[-FX]+FX
```

The axiom is similar to step 1 of our definition. The F draws the trunk and the X represents the branches. This is our basic definition of a tree; everywhere we write FX, a tree will sprout. The X isn't actually expanded into branches until the next order, but if we *assume* that the X part knows what it's doing, then we can see that this will indeed draw a tree as defined above. The order 1 string of the system will be FS[-FX]+FX. We can read this as: move forward (the trunk); shorten the length of each move; stop what you're doing; turn right; draw a tree (remember that we said every FX will sprout a new tree); return to where you were when you were last interrupted; turn left; draw a tree. The place "where you were last interrupted" is just after the point where we drew the trunk (from the order 0 part). The result will look like the picture on the left of the figure above.

Consider what will happen with the next order string, FS[-FS[-FX]+FX]+FS[-FX]+FX. The main trunk (0) is drawn; the turtle goes right; a new trunk (1) is drawn; the turtle goes right again and draws another trunk (2) and an X to be expanded next time. Now it returns to where it was last interrupted, trunk (1). It then turns left and draws a trunk (3) and an X to be expanded next time. Now it returns to where it was last interrupted again, which is at the main trunk (0). It turns left, draws a trunk (4), and then repeats the same process as for trunk (2). The result is the middle picture of the figure above. As an exercise, you might try deriving the order 3 string yourself and then trying to draw it by following the turtle rules. Verify that the result looks like the rightmost picture.

Didn't You Mention an Applet?

Impatient little turtle, aren't you?

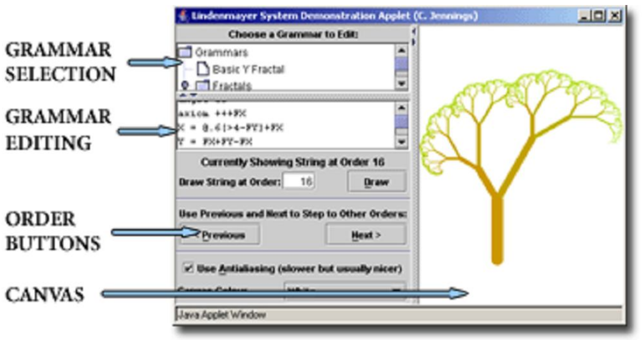
But I did indeed, and you're nearly ready to try it out (and probably already have). But first I need to explain a few differences between what we've been working with and how the program works. First, since → is not very convenient to type on most keyboards, the program uses = instead. Second, to specify how far the turtle should turn with each - or +, you use an *angle n* command. The turtle will turn in 360/*n*° increments. Third, the letter G will cause the turtle to move forward without drawing. Fourth, as with →, typing ε is inconvenient for most people. To specify the empty string, you can either leave the right side of the production blank or use the _ character. Lastly, to change the line length, use @ followed by a number. The current line length will be multiplied by the number you supply. (Note: Even if you normally use "," to indicate a decimal point, use "." in the program. One half is "@0.5".) The grammar for our tree system works out to:

```
angle 8
axiom FX
X = @.6[-FX]+FX
```

There are some more commands as well, for doing things like drawing in colour. But you seem anxious to start, so go ahead. There is a table further down that explains the rest of the commands when you're ready.

Exploring d0L-systems

Once Java has started up and downloaded the applet, a button will appear below that opens its main window. This window is split in two halves; you choose the details of the L-system you wish to work with on the left half, and the results are drawn on the right half (the "canvas"). There are three basic steps for working with the applet:


GRAMMAR SELECTION
GRAMMAR EDITING
ORDER BUTTONS
CANVAS
Lindenmayer System Demonstration Applet (C. Jennings)
Choose a Grammar to Edit:
Grammars
Basic Y Fractal
Fractals
Currently Showing String at Order 16
Draw String at Order: 16
Draw
Use Previous and Next to Step to Other Orders:
< Previous
Next >
Use Antialiasing (slower but usually nicer)
Java Applet Window

```
axiom +++FX
X = @.6[>4-FY]+FX
Y = FX+FY-FX
```

1. Select a grammar. The grammar selection box contains a collection of over 50 (mostly well-known) grammars divided into loose categories. In addition, there is a "Custom" selection that allows you to enter your own.
2. View or edit the grammar if desired. When a grammar is selected, it will appear in the editing field immediately below. Edited grammars replace the "Custom" selection; the original version is unaffected.
3. Expand the string. Recall that the string produced by your L-system starts as the "axiom" at order 0, and then proceeds to the next order by replacing every occurrence of a symbol according to its associated production (if any). Most grammars require you to step through a few orders before they start to look interesting. Press the Next button to expand the string produced by your L-system to a higher order. The string will be rewritten to the next-higher order and then the result will be drawn on the canvas. (The rewriting happens behind the scenes, you don't actually see the string.) You can also use the Previous button to return to a previous order.

Generally, L-systems grow in size extremely quickly. Since applets are only allowed to use a fairly small amount of memory for security reasons, it is easy to request an order that is too high for the applet to compute. When this happens, a message will appear on the canvas to tell you. Just use the Previous button to go back down an order or choose a new grammar to explore.

Requires Java.

Note: It may take a minute for the launch button to appear above.

Grammar Description

The command set is designed to be similar to that of the fractal software FRACTINT, although the applet's grammar language has both some added and some missing features compared to FRACTINT's L-system support. One important difference is that this parser is case-sensitive, while FRACTINT is not. Another is that the D/M form of the F/G commands is not implemented. This form is rarely used, although / and \ have been reserved, so they might be implemented in future. I must also give credit to others for the included grammars, many of which were copied or adapted from the FRACTINT or other L-system collections of many individuals.

d0L-system Grammar Description Language

Preamble	
angle *n*	Sets the amount of a single turn using + or - to 360/*n*°.
order *n*	Sets the default order to draw this system at to *n*.
rotate *n*	The entire image will be rotated *n*° before drawing.
Productions	
axiom *s*	The initial string produced by the L-system at order 0.

p = *s*	When the L-system is rewritten at a higher order, every instance of the letter *p* will be replaced by the string *s*. Note that *p* must be a single character, and cannot be one of the special symbols =, +, -, !, |, [,], <, >, @, /, \, _, c, or the space character. Rewriting is case-sensitive, so, *e.g.*, x = xXx is different from x = xxx.
Drawing	
F	The turtle will move forward one stroke with the pen down, *i.e.*, it will draw as it moves. Note that *f* (the lowercase form) has no special meaning. Because F is not a reserved special symbol, it can (and often does) appear on the left side of a production.
G	The turtle will move forward one stroke with the pen up. Note that *g* has no special meaning. Because G is not a reserved special symbol, it can (and often does) appear on the left side of a production.
+	The turtle turns one step to the "left" without moving.
-	The turtle turns one step to the "right" without moving.
!	Reverses the meaning of "left" and "right."
|	Turn around (the turtle turns as close to 180° as possible).
@*n*	Multiply the current stroke length by *n*, where *n* is a decimal number. The value *n* may be preceded by an I, in which case 1/*n* is used instead, or it may be preceded by a Q, in which case the square root of *n* is used. Example: @IQ2 multiplies the current line length by 1/√2.
>, >*n*	The > command changes the colour of the drawing pen. The turtle has a set of 256 pens which progress smoothly through the colours brown, green, blue, indigo, violet, red, orange, and back to brown. This palette has been chosen to make it easy to draw both plant-like structures and freeform designs with nice results. Using a > moves the pen one colour to the right (towards the right end of the above list). Optionally, the > may be followed by a number *n*, in which case the colour changes by *n* steps instead of 1.
<, <*n*	As above, with the colour change moving toward the left instead of the right.
c*n*	Change directly to pen colour *n*.
[	Store the current state of the turtle, including position, direction, stroke length, and pen colour on a stack.
]	Restores the state of the turtle to be exactly as it was when the most recent [command was processed. Note that the number of [and] commands should match.
/, \	Reserved. These symbols are reserved and may be used in a future version.
Comments	
;	A semicolon causes the rest of the current line (including the ; itself) to be ignored.

Bibliography

[1] Aristid Lindenmayer. **Mathematical models for cellular interaction in development.** *Journal of Theoretical Biology,* 18:280–315, 1968.

[2] Alan M. Turing. **The chemical basis of morphogenesis.** *Philosophical Transactions of the Royal Society of London. B* **327**, 37–72 (1952).

Return to Home Page Return to Tickle Trunk Send Feedback

April 14, 2002—Updated January 10, 2010