

# Lecture 10: Lindenmayer Systems

(Stacks, Queues, `append`, and list literals)

Lindenmayer systems or L-systems are a way to model complex shape construction simply. They are based on repeated string replacement

## Repeated string replacement

---

We're given a set of rules of the following form:

$A \rightarrow \text{some sequence of letters}$

which mean "change A into the given sequence of letters".

Example:

$A \rightarrow B-A-B$

$B \rightarrow A+B+A$

To "run" this set of rules, we start with some initial string, and repeatedly apply all the rules in parallel:

1. A
2. B-A-B
3. A+B+A-B-A-B-A+B+A
4. ...

## Lindenmayer systems

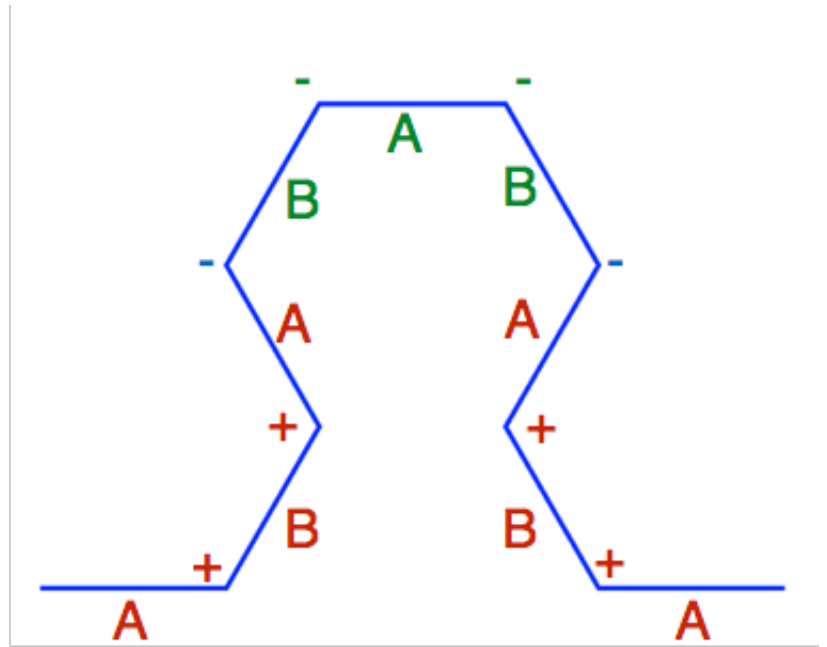
---

Suppose we give a meaning to each of the symbols that give instructions to a turtle sitting on a piece of paper, for example:

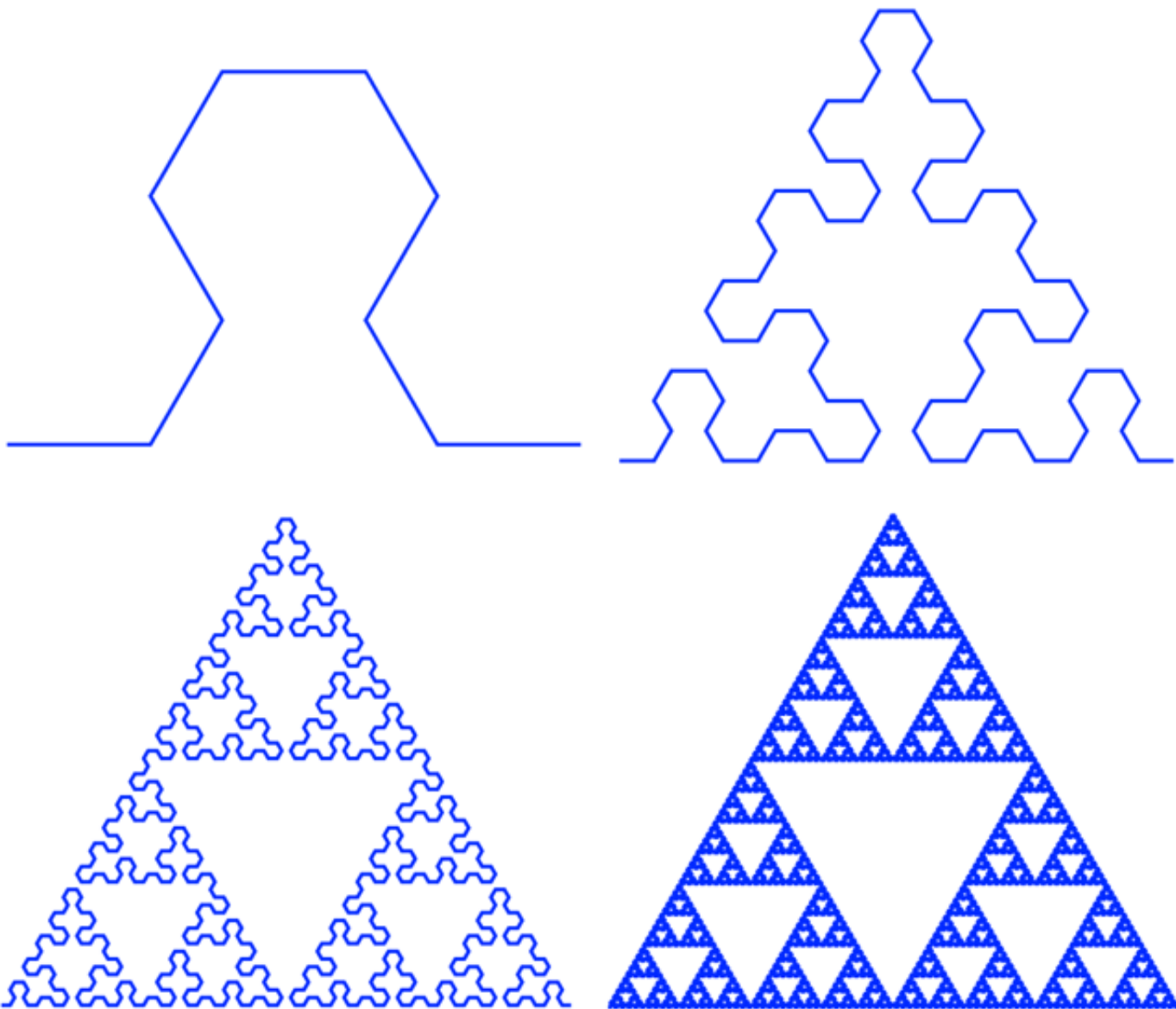
- A and B: draw line forward in the direction you're facing
- -: turn right by 60°
- +: turn left by 60°

Then you can think of a particular string as a sequence of commands:

A+B+A-B-A-B-A+B+A →



And if you generate a really long string you get:



which is called the Sierpinski triangle.

## Another example

Different rules and interpretations give different pictures. If we have the rules:

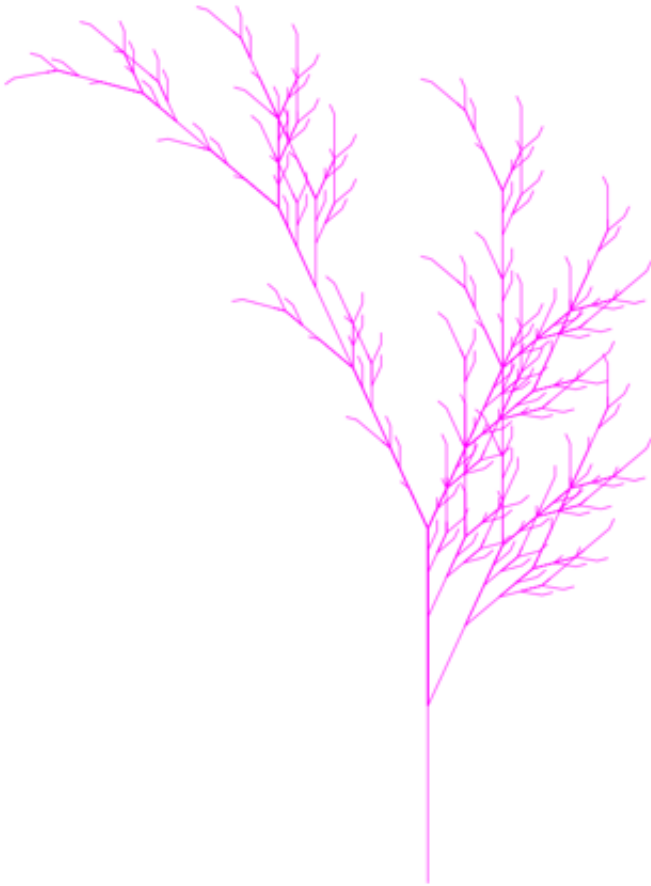
$$X \rightarrow F-[[X]+X]+F[+FX]-X \quad F \rightarrow FF$$

with the interpretation:

- F: draw forward
- -: turn left 25°
- +: turn right 25°
- X: do nothing

- [: save the current position & direction
- ]: restore the most-recently saved position & direction

Then you get the picture:



which is a pretty nice simulation of a plant!

Notice that very "complex" behavior arises out of something pretty simple.

## Coding up L-systems

### Attempt #1 --- BUGGY

We use a pair of lists ( `lhs` and `rhs` ) to represent the rules: one list holds the left-hand sides, and one holds the right hand sides.

```

1 // BAD CODE BAD CODE BAD CODE BAD CODE
2
3 // simulate the given L-system rules for steps steps starting from the
4 // given "start" string
5 func badLindenmayer(lhs, rhs []string, start string, steps int) {
6     var curString, nextString = "", start
7
8     for i := 0; i < steps; i++ {
9         curString = nextString
10
11         // apply every rule
12         for i := 0; i < len(lhs); i++ {
13             nextString = strings.Replace(nextString, lhs[i], rhs[i], -1)
14         }
15
16         fmt.Println(nextString)
17     }
18 }
19
20 func main() {
21     var lhs = []string{ "A", "B", "C" }
22     var rhs = []string{ "BAB", "AC", "c" }
23
24     badLindenmayer(lhs, rhs, "A", 3)
25 }
26 // BAD CODE BAD CODE BAD CODE BAD CODE

```

**Test yourself!** (before reading ahead) What is wrong with the above code?

When run, the above code will print out:

```

1 AcAAc
2 AcAAccAcAAcAcAAcc
3 AcAAccAcAAcAcAAcccAcAAccAcAAcAcAAccAcAAccAcAAcAcAAccc
4 AcAAccAcAAcAcAAcccAcAAccAcAAcAcAAccAcAAccAcAAcAcAAcccAcAAccAcAAcAcAAccc...

```

which is wrong because the code didn't apply all the rules at once! After replacing the first A with BAB, it will replace the Bs with AC, and then replace the Cs with c all in the first step.

**Attempt #2 --- correct**

```

1 // Return the Rhs for a given Lhs
2 func getRhsFor(char string, lhs, rhs []string) (string, bool) {
3     for i := 0; i < len(lhs); i++ {
4         if lhs[i] == char {
5             return rhs[i], true
6         }
7     }
8     return "", false
9 }
10
11 // applyRules will take string "start" and apply each of the given rules
12 // in parallel, returning the new string.
13 func lindenmayer(lhs, rhs []string, start string) string {
14     var out string
15     for _, char := range start {
16         charRhs, existed := getRhsFor(string(char), lhs, rhs)
17         if existed {
18             out = out + charRhs
19         } else {
20             out = out + string(char)
21         }
22     }
23     return out
24 }
25
26 func main() {
27     var lhs = []string{ "A", "B", "C" }
28     var rhs = []string{ "BAB", "AC", "c" }
29
30     for i := 0; i < 3; i++ {
31         lindenmayer(lhs, rhs, "A", 3)
32     }
33 }

```

## Stacks

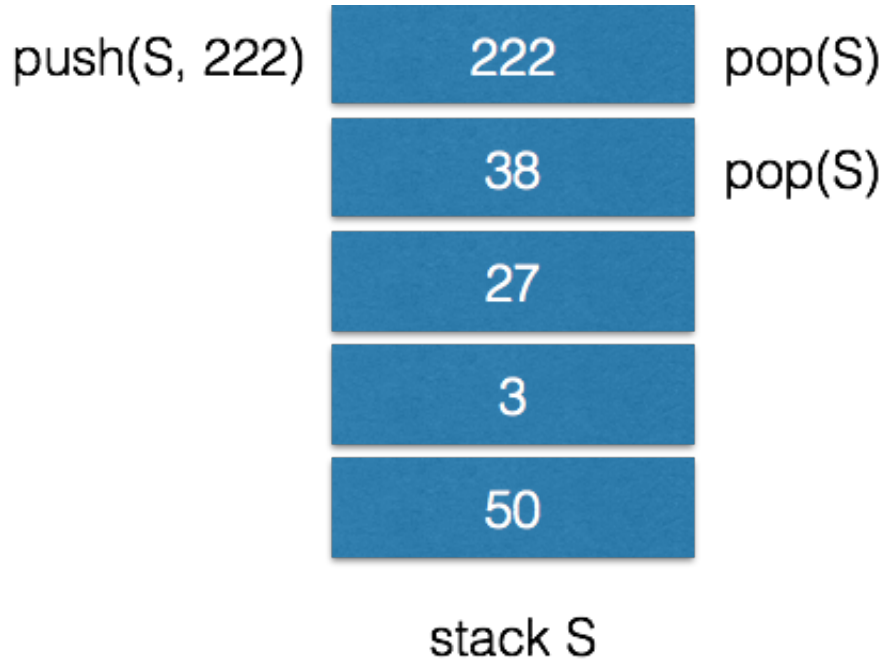
A [stack](#) is an example of an [abstract data type](#): it is a way of organizing data that support a set of well-defined operations.

For a stack, we have the following operations:

- `S := createStack()` creates a new, empty stack.

- `push(S, x)` puts item `x` onto the top of stack `S`.
- `x = pop(S)` takes the most-recently added item off of the stack and assigns it to `x`.
- `size(S)` returns the number of items currently on the stack.

A good mental image is a stack of cafeteria plates:



### Example: a way to reverse a list of integers

Suppose you are given a list of integers and you want to reverse the order of the items in the list:

`-1, -30, 60, 21, 33, 78, 64` → `64, 78, 33, 21, 60, -30, -1`

```
var list []int
```

```
var reversedList []int
```

If you had functions for `createStack`, `push`, `pop`, and `size`, then you could reverse a list of integers using the following code:

```

1 func reverseList(in []int) []int {
2     S := createStack()
3     for i := 0; i < len(in); i++ {
4         push(S, v)
5     }
6
7     var v int
8     var out []int = make([]int, 0)
9
10    for size(S) != 0 { // ***
11        S, v = pop(S)
12        out = append(out, v)
13    }
14    return out
15 }

```

Each time through the loop marked with a `***`, the top of the stack is removed and added to the end of out:

64							
78	78						
33	33	33					
21	21	21	21				
60	60	60	60	60			
-30	-30	-30	-30	-30	-30		
-1	-1	-1	-1	-1	-1	-1	
<hr style="width: 20%; margin: 0 auto;"/>							
S							
	64	78	33	21	60	-30	-1

## Implementing stacks

We could use a list to hold the items in a stack:

```

1 func createStack() []int {
2     return make([]int, 0)
3 }

```



Pushing puts the item at the end of the list, and returns the new list:

```
1 func push(S []int, item int) []int {
2     return append(S, item)
3 }
```

Note that we have to return the updated list because `append` returned a new list.

Popping removes the last item, and returns the new list:

```
1 func pop(S []int) ([]int, int) {
2     if len(S) == 0 {
3         panic("Can't pop empty stack!")
4     }
5     item := S[len(S)-1]
6     S = S[:len(S)-1]
7     return S, item
8 }
```


**New Syntax!** `L[x:y]` creates a sublist of the list `L`. The sublist contains elements from indices `x` up to but not including `y`.

**Test yourself!** If `L = [0,1,2,3,4,5]` what is `L[2:5][0]` ?

## Stacks vs. Queues

A queue is another abstract data type that operates just slightly differently from a stack: rather than returning things most-recently-added-first, a queue returns items in the order that they were added (just like a "queue" at the post office):

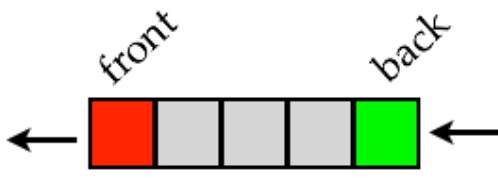
Stack:  
*aka LIFO*



push, pop

LIFO = last-in, first-out

Queue:  
*aka FIFO*



enqueue, dequeue

FIFO = first-in, first-out

Stacks useful to save subproblems to solve later. Every time you type in Microsoft Word, it adds what you typed to a stack. Control-Z pops the last thing you did and undoes it.

Queues useful for processing events. Every time you click your mouse, where you clicked is added to a queue. The computer processes the clicks in the order you did them.

## Implementing [ and ] for the plant example

---

Recall the "plant" L-system from above. It had two operations: [ and ] where [ saved the current position and direction and ] restored the most-recently-saved position and direction.

How could we implement these?

When we encounter a ] we want the most-recently-saved state, we can use a stack!

- When you see [ save the current position and direction onto a stack
- When you see ] pop the top position and direction from the stack and set the current position and direction to them

F-[[X[+X]][-[X]+]X-]X+

(The string and angles in this example don't match — they're just placeholders)



stack S

## Summary

Lindenmayer systems are a cute idealization of branching and evolving systems.

*Stacks* are a data structure that is like a list except you can only access one end of the list with:

- push: add something to the top of the list
- pop: remove the top thing on the list

Queues are lists where we add things to one end and take things from the other. Queues keep the items in order.

Strings behave a lot like lists except that you can't modify their elements.

Literal lists can be specified using the syntax `[ ]TYPE{value1, value2, ...}`.

## Glossary

- Lindenmayer system or L-system: A modeling system based on repeated replacement.
- Stack: A data structure that holds items, and returns them in the reverse order in which you added them (last in first out)
- Queue: A data structure that holds items and returns them in the same order in which you added them (first in first out)
- List literal: If you know the items you want to have in your list (to start at least), you can use a list literal of the form `[ ]TYPE{item1, item2, ...}` to specify them. For example `[ ]float64{1.0, 2.0, 3.14}`

- append: The builtin-Go function that adds an item to the end of the list.