Stacks and Queues

02-201 / 02-601

Another Slice and String Example

Repeated String Replacement

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

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

that say "change A into the given sequence of letters"

Example:

A → B–A–B
B → A+B+A

We want to start with some string (say "A") and repeatedly apply the rules:

A
B-A-B
A+B+A-B-A-B-A+B+A

All the rules get applied simultaneously.

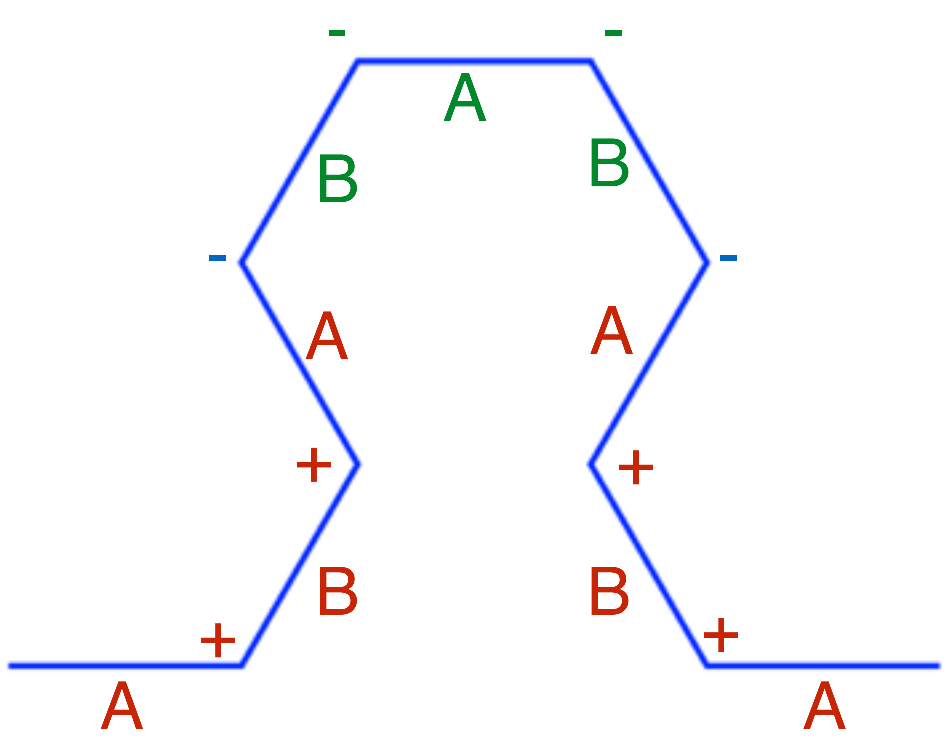
Lindenmayer Systems

A → B–A–B
B → A+B+A

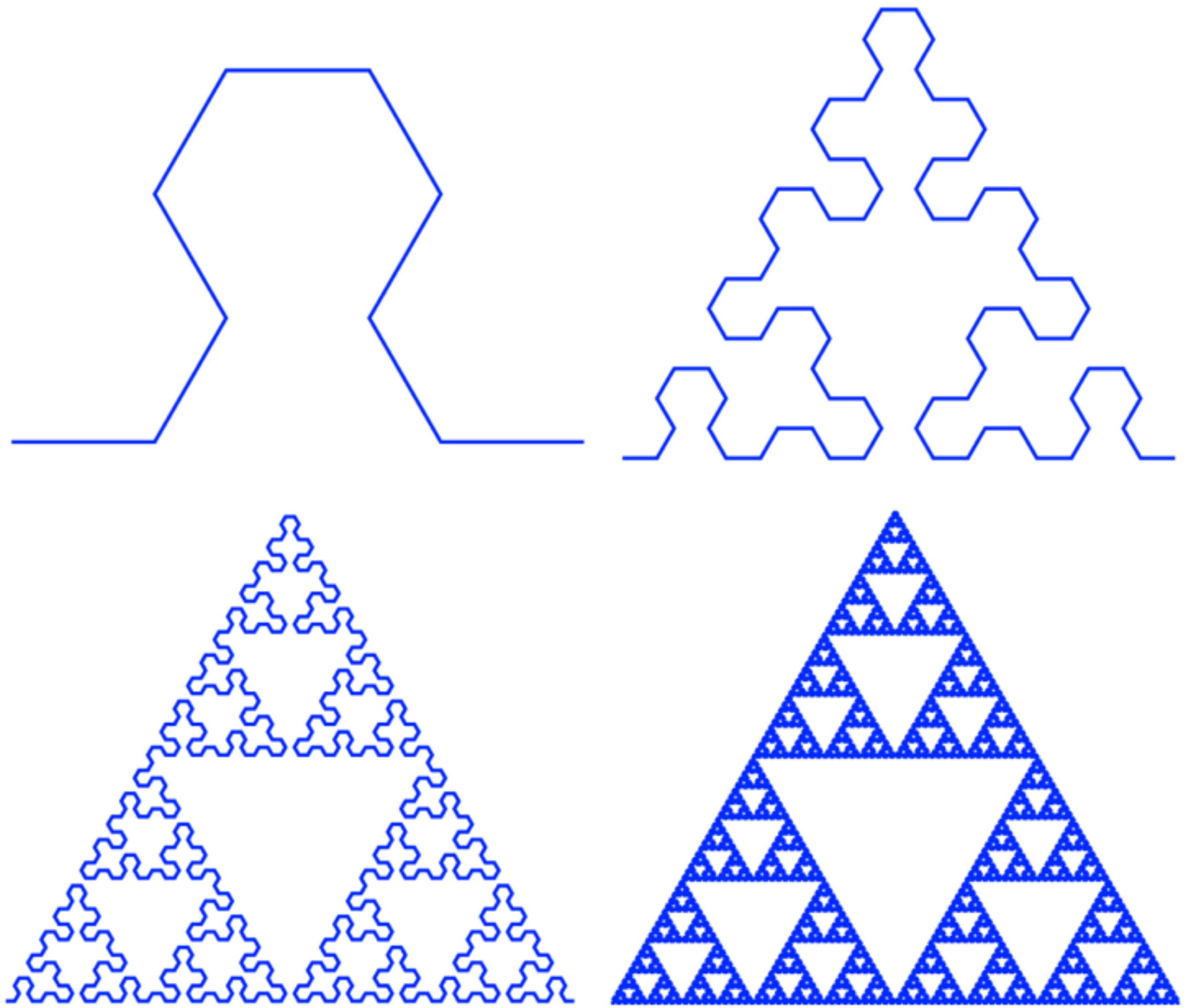
Suppose we give a meaning to each of the symbols that give instructions to a turtle sitting on a piece of paper:

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

A
B-A-B
A+B+A-B-A-B-A+B+A →



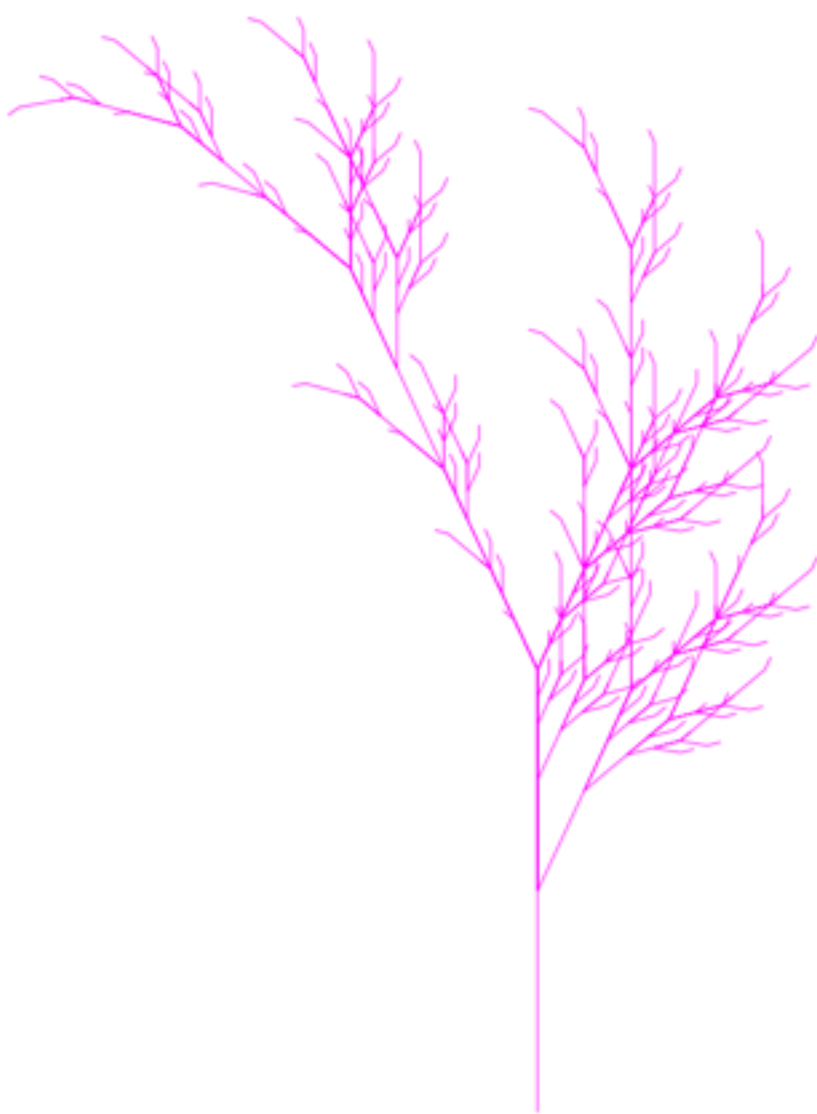
More iterations!



→ **Sierpinski triangle**

Another Example Lindenmayer System

X → F-[[X]+X]+F[+FX]-X
F → FF



- F: draw forward
- -: turn left 25°
- +: turn right 25°
- X: do nothing
- [: save the current position & direction
-]: restore the most-recently saved position & direction

save restore

X → F-[[X]+X]+F[+FX]-X

First Attempt to code Lindenmayer Systems

```
func lindenmayer(lhs, rhs []string, start string, steps int) {
    var curString, nextString = "", start

    for i := 0; i < steps; i++ {
        curString = nextString

        // apply every rule
        for i, a := range lhs {
            nextString = strings.Replace(nextString, a, rhs[i], -1)
        }

        fmt.Println(nextString)
    }
}

func main() {
    var lhs = []string{ "A", "B", "C"}
    var rhs = []string{ "BAB", "AC", "c" }

    lindenmayer(lhs, rhs, "A")
}
```

Problem! It doesn'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.

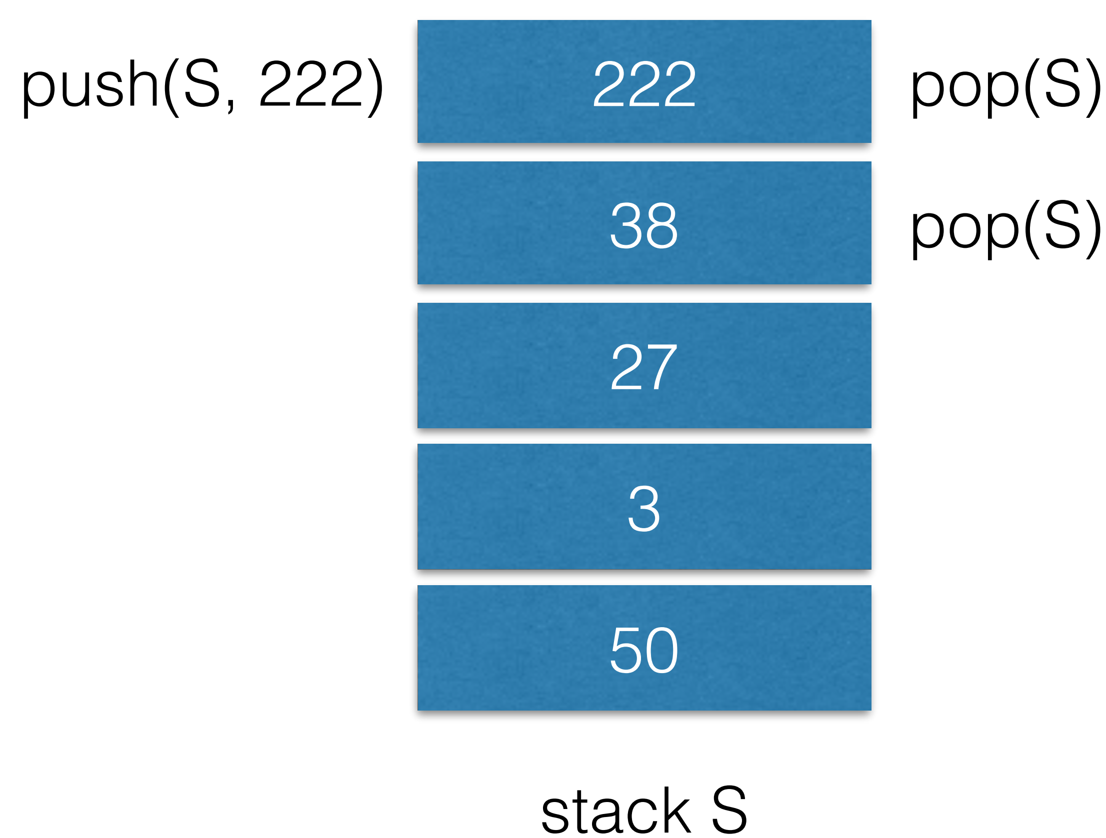
```
AcAAc
AcAAccAcAAcAcAAcc
AcAAccAcAAcAcAAcccAcAAccAcAAcAcAAccAcAAccAcAAcAcAAccc
AcAAccAcAAcAcAAcccAcAAccAcAAcAcAAccAcAAccAcAAcAcAAccccAcAAccAcAAcAcAAcccAcAAccAcAAcAcAAcc…
```

Live Coding: Updated (Correct?) Lindenmayer Program

Stacks

Stack Data Structure

- push(S, Item): put an item Item onto the top of the stack S.
- Item = pop(S): set Item to the item at the top of the stack S and remove the top item.



push(S, 50); push(S, 3); push(S, 834); pop(S); push(S, 27); push(S, 5555); pop(S); push(S, 38)

How would you reverse a list of integers?

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

`var list []int` `var reversedList []int`

```
func reverse(in []int) []int {

   S := createStack()
   for _, v := range in {
      push(S, v)
   }

   var v int
   var out []int = make([]int,0)

   for len(S) != 0 {
      S, v = pop(S)
      out = append(out, v)
   }
   return out
}
```

Each time through the green loop, 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
S						

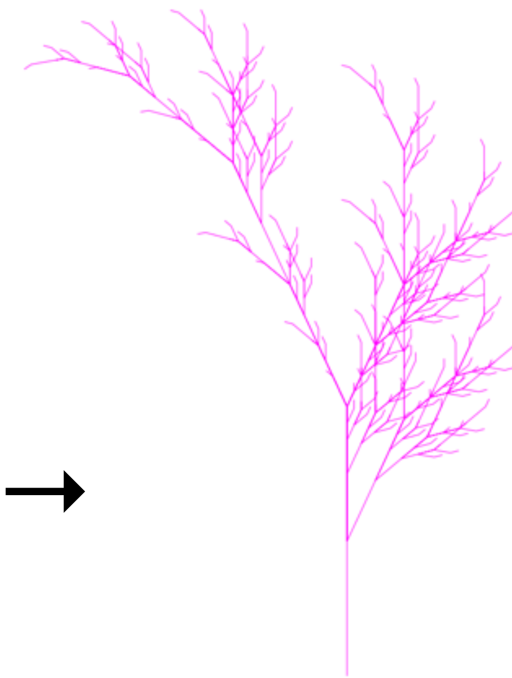
64 78 33 21 60 -30 -1

out

How would you implement “[“ and “]” when drawing the Lindenmayer system we saw?

- F: draw forward
- -: turn left 25°
- +: turn right 25°
- X: do nothing
- **[: save the current position & direction**
- **]: restore the last saved position & direction**

F-[[X]+X]+F[+FX]-X →



When you see [the 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+]

(80, 80) 300°

(12,700) 100°

(102, 34) 325°

(50,50) 75°

stack S

(The string and angles in this example aren’t real — they’re just placeholders)

Stack Implementation

```
func createStack() []int {
   return make([]int, 0)
}
```

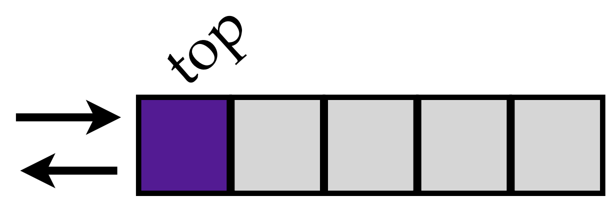
```
func push(S []int, item int) []int {
   return append(S, item)
}
```

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

```
func main() {
    S := createStack()

    S = push(S, 1)
    S = push(S, 10)
    S = push(S, 13)
    fmt.Println(S)

    S, item := pop(S)
    fmt.Println(item)

    S, item = pop(S)
    fmt.Println(item)

    S, item = pop(S)
    fmt.Println(item)
}
```

Stacks vs. Queues

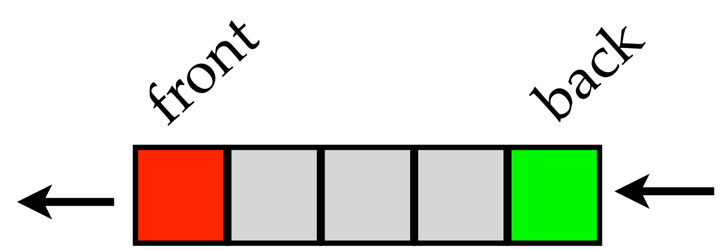
Stack:
aka LIFO



push, pop

LIFO = last-in, first-out

Queue:
aka FIFO



enqueue, dequeue

FIFO = first-in, first-out

More Example Uses

- 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.

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.