Announcements

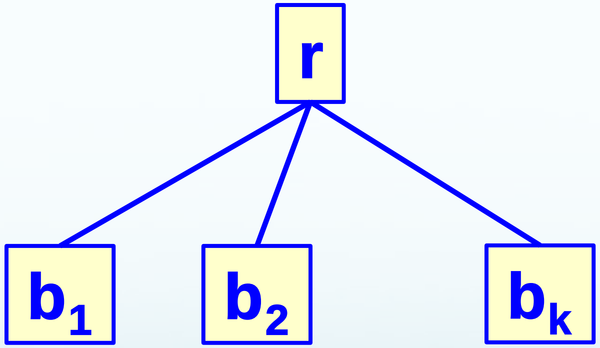
- Form teams and work on project 4
  - Check instructions on home page => projects

About Trees and Recursion

- Summing all nodes
- Expression evaluation
- Dragon curve pattern
- L-systems

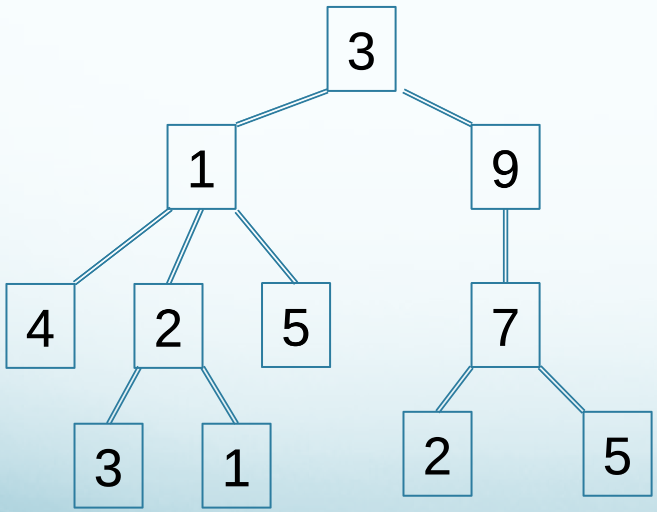
Tree Encoding

- **[r, b_1, …, b_k]** encodes the node r and its descendants
- Nesting builds up the tree



Summing all Node Values

- Assume given a list all of whose elements are numbers or sublists of numbers, nested arbitrarily
- This list encodes a tree all of whose nodes, including leaves, are labeled with a number
- We want to sum all numbers in the tree



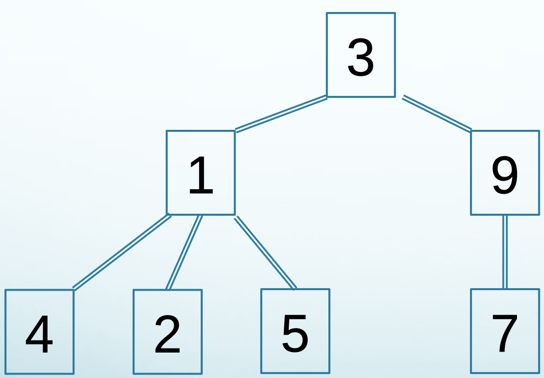
[3,[1,4,[2,3,1],5],[9,[7,2,5]]]

Code

```
def sumTree(L):
    if type(L) == int or type(L) == float:
        return L
    if type(L) != list:
        print("unknown tree node",L)
        return
    sum = 0
    for L1 in L:
        sum = sum + sumTree(L1)
    return sum
```

Summing all Node Values

1. Start with outer list. First element is evaluated to 3, so sum = 3
2. Second element is a list: create a new function copy to find its sum. That function copy returns 12; sum is now 15
3. Third element is a list: yet another function copy is created to sum its elements. That function copy returns 16; sum is now 31
4. The outer list is done, 31 is returned



[3,[1,4,2,5],[9,7]]

[3,[1,4,2,5],[9,7]]

[3,12,[9,7]]

[3,12,[9,7]]

[3,12,16]

31

Code

```
def sumTree(L):
    if type(L) == int or type(L) == float:
        return L
    if type(L) != list:
        print("unknown tree node",L)
        return
    sum = 0
    for L1 in L:
        sum = sum + sumTree(L1)
    return sum
```

Code

```
def sumTree(L):
    if type(L) == int or type(L) == float:
        return L
    if type(L) != list:
        print("unknown tree node",L)
        return
    sum = 0
    for L1 in L:
        sum = sum + sumTree(L1)
    return sum
```

Code

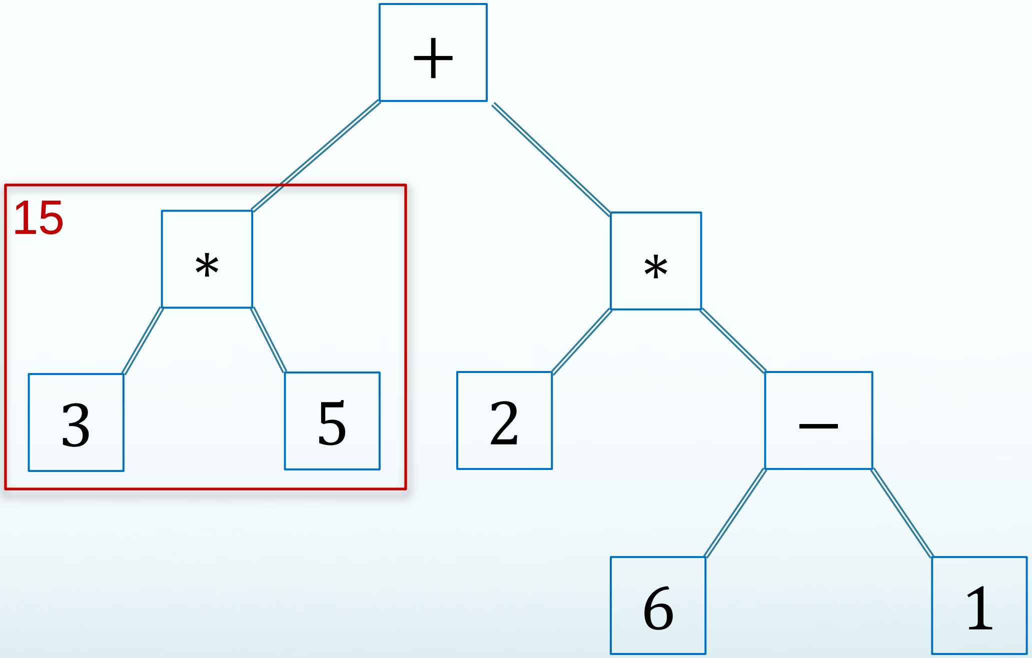
```
def sumTree(L):
    if type(L) == int or type(L) == float:
        return L
    if type(L) != list:
        print("unknown tree node",L)
        return
    sum = 0
    for L1 in L:
        sum = sum + sumTree(L1)
    return sum
```

Challenge Problem

- Modify the code so that only interior node values are summed…

Expression Evaluation

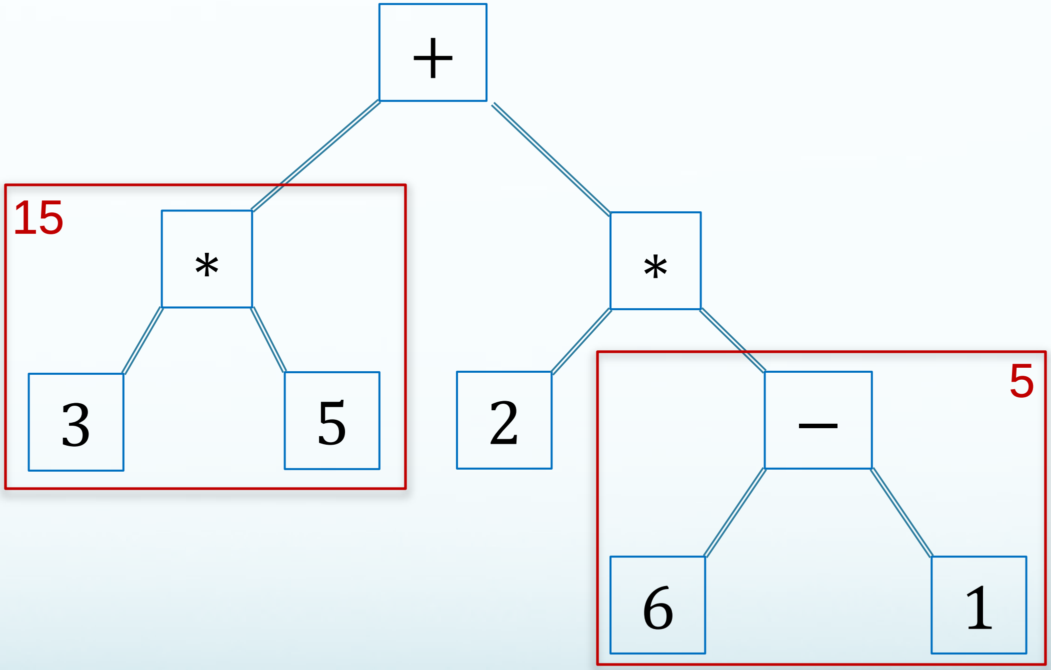
$$E = \boxed{3 * 5} + 2 * (6 - 1)$$



[+, [*, 3, 5], [*, 2, [-, 6, 1]]]

Expression Evaluation

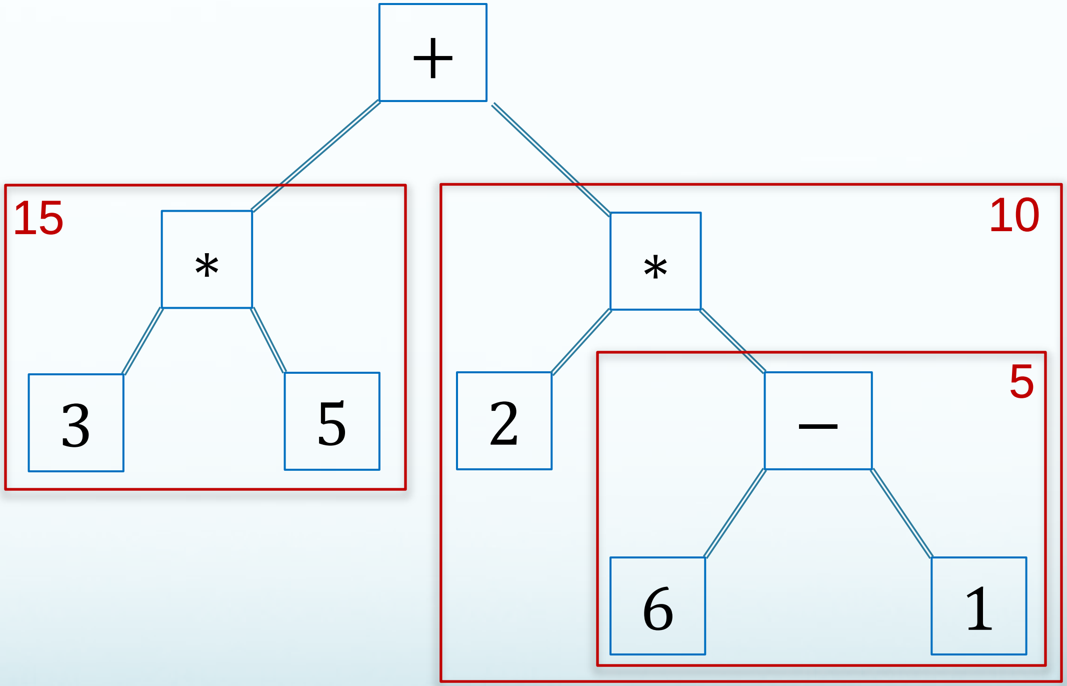
$$E = 3 * 5 + 2 * (6 - 1)$$



[+, [*, 3, 5], [*, 2, [-, 6, 1]]]

Expression Evaluation

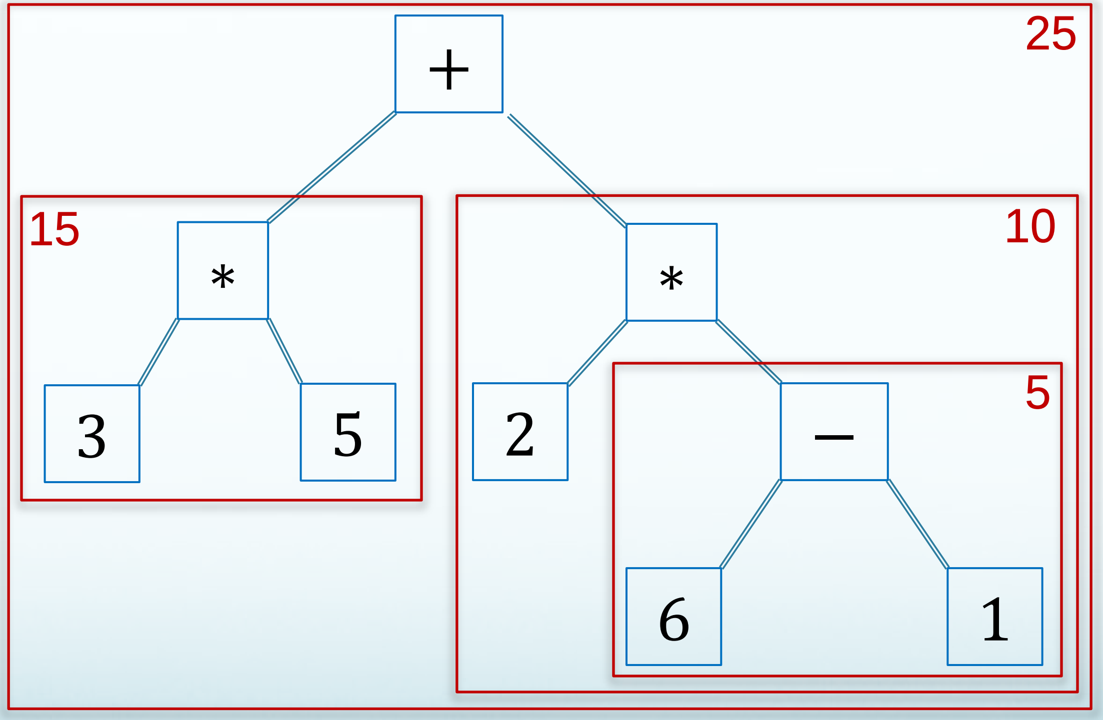
$$E = 3 * 5 + 2 * (6 - 1)$$



[+, [*, 3, 5], [*, 2, [-, 6, 1]]]

Expression Evaluation

$$E = 3 * 5 + 2 * (6 - 1)$$



[+, [*, 3, 5], [*, 2, [-, 6, 1]]]

Code

```
def evalTree(L):
    if type(L) == int or type(L) == float:
        return L
    if type(L) != list:
        print("unknown tree node",L)
        return
    x = evalTree(L[1])
    y = evalTree(L[2])
    if len(L)!= 3:
        print("too many operands", L)
        return
    if L[0] == '+': return x+y
    if L[0] == '-': return x-y
    if L[0] == '*': return x*y
    if L[0] == '/': return x/y
    print("unknown operation",L[0])
    return
```

Tree Leaf Case

```
def evalTree(L):
    if type(L) == int or type(L) == float:
        return L
    if type(L) != list:
        print("unknown tree node",L)
        return
    x = evalTree(L[1])
    y = evalTree(L[2])
    if len(L)!= 3:
        print("too many operands", L)
        return
    if L[0] == '+': return x+y
    if L[0] == '-': return x-y
    if L[0] == '*': return x*y
    if L[0] == '/': return x/y
    print("unknown operation",L[0])
    return
```

Node is either leaf or list

```
def evalTree(L):
    if type(L) == int or type(L) == float:
        return L
    if type(L) != list:
        print("unknown tree node",L)
        return
    x = evalTree(L[1])
    y = evalTree(L[2])
    if len(L)!= 3:
        print("too many operands", L)
        return
    if L[0] == '+': return x+y
    if L[0] == '-': return x-y
    if L[0] == '*': return x*y
    if L[0] == '/': return x/y
    print("unknown operation",L[0])
    return
```

Interior Node: [op,x,y]

```
def evalTree(L):
    if type(L) == int or type(L) == float:
        return L
    if type(L) != list:
        print("unknown tree node",L)
        return
    x = evalTree(L[1])
    y = evalTree(L[2])
    if len(L)!= 3:
        print("too many operands", L)
        return
    if L[0] == '+': return x+y
    if L[0] == '-': return x-y
    if L[0] == '*': return x*y
    if L[0] == '/': return x/y
    print("unknown operation",L[0])
    return
```

List length must be 3

```
def evalTree(L):
    if type(L) == int or type(L) == float:
        return L
    if type(L) != list:
        print("unknown tree node",L)
        return
    x = evalTree(L[1])
    y = evalTree(L[2])
    if len(L)!= 3:
        print("too many operands", L)
        return
    if L[0] == '+': return x+y
    if L[0] == '-': return x-y
    if L[0] == '*': return x*y
    if L[0] == '/': return x/y
    print("unknown operation",L[0])
    return
```

Unknown operation

```
def evalTree(L):
    if type(L) == int or type(L) == float:
        return L
    if type(L) != list:
        print("unknown tree node",L)
        return
    x = evalTree(L[1])
    y = evalTree(L[2])
    if len(L)!= 3:
        print("too many operands", L)
        return
    if L[0] == '+': return x+y
    if L[0] == '-': return x-y
    if L[0] == '*': return x*y
    if L[0] == '/': return x/y
    print("unknown operation",L[0])
    return
```

Bad Operand

```
def evalTree(L):
    if type(L) == int or type(L) == float:
        return L
    if type(L) != list:
        print("unknown tree node",L)
        return
    x = evalTree(L[1])
    y = evalTree(L[2])
    if len(L)!= 3:
        print("too many operands", L)
        return
    if L[0] == '+': return x+y
    if L[0] == '-': return x-y
    if L[0] == '*': return x*y
    if L[0] == '/': return x/y
    print("unknown operation",L[0])
    return
```

Modified Code

```
def evalTree(L):
    if type(L) == int or type(L) == float:
        return L
    if type(L) != list:
        print("unknown tree node",L)
        return
    x = evalTree(L[1])
    y = evalTree(L[2])
    if len(L)!= 3:
        print("too many operands", L)
        return
    if (type(x)!=int and type(x)!=float) or
       (type(y)!=int and type(y)!=float):
        return
    if L[0] == '+': return x+y
    if L[0] == '-': return x-y
    if L[0] == '*': return x*y
    if L[0] == '/': return x/y
    print("unknown operation",L[0])
    return
```

Summary

- Nested call to the same function is allowed.
  It is called ***recursion***.
- Think of it as multiple copies each with their own set of parameters and local variables.
- If there is no "base case" and you keep calling, then the program won't finish and will eventually die.
- To master recursion, you must:
  - Think on multiple levels (think *Inception*)
  - Visualize a calling tree
  - Understand a self-similar pattern

Challenge Problem

- Modify the expression evaluation code so as to allow that + has more than 2 operands.
- Example: $E = 1 * 2/(3 + 4 + 5) + 6 * 7 + 8 * 9$

  E = [+,[*,1,[/,2,[+,3,4,5]]],[*,6,7],[*,8,9]]

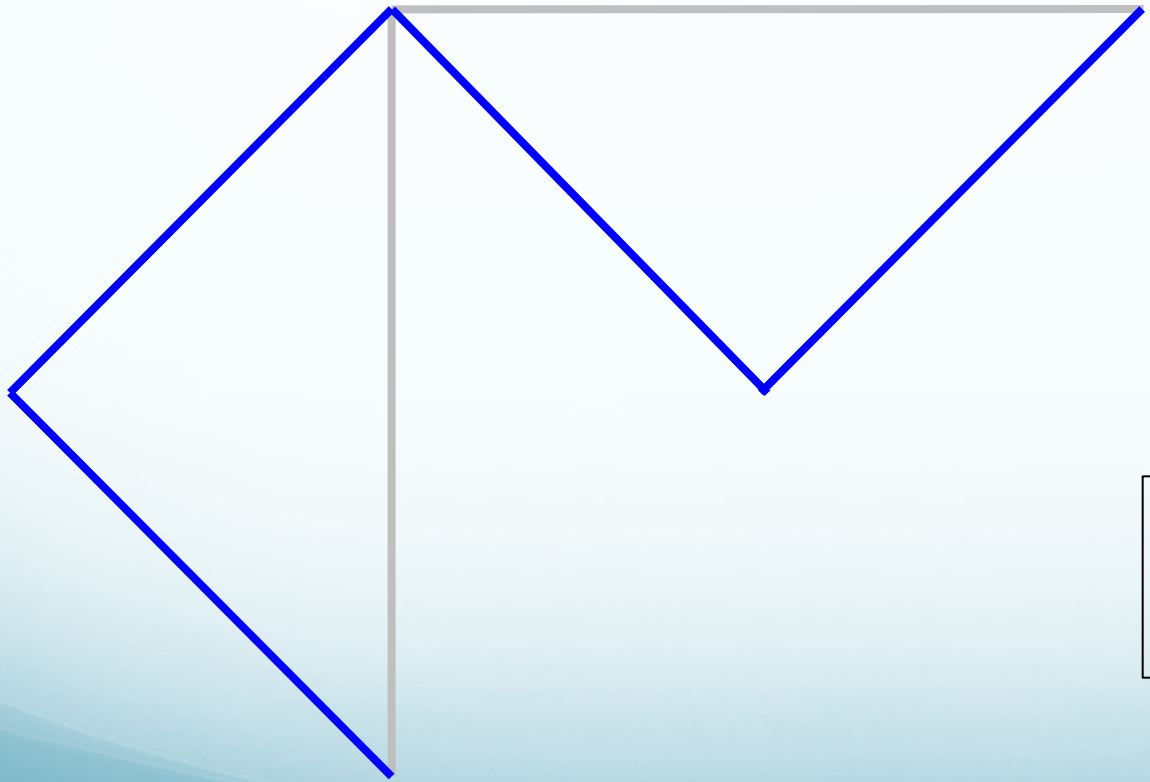
Dragon Curve

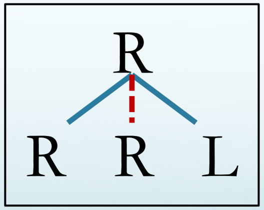
- Drawn in Project 4…
- How to generate the string of drawing commands?

- How does the dragon curve come about?

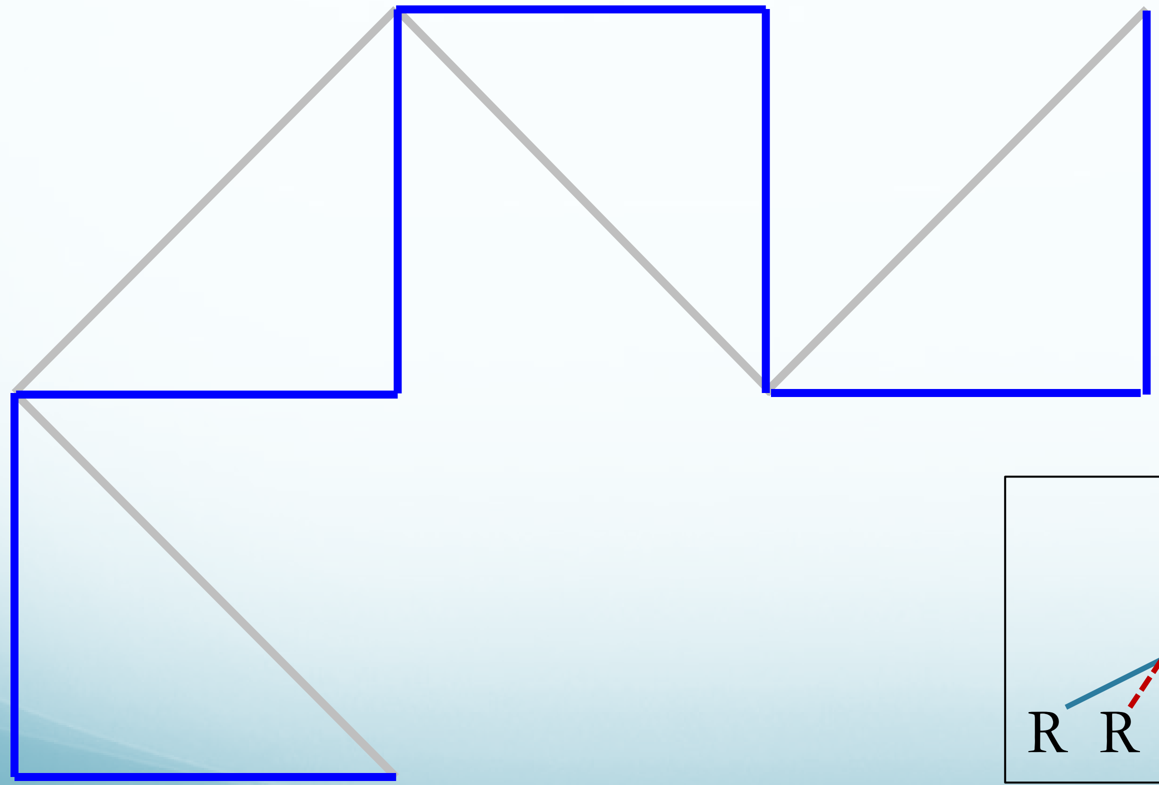
Startup: 1 fold

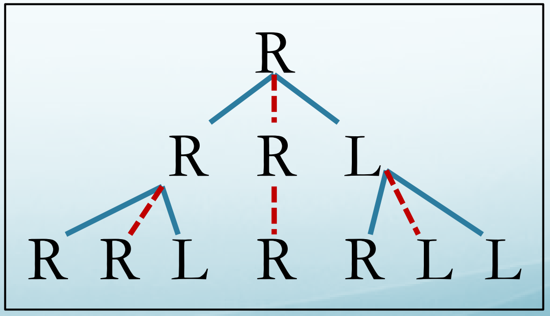
2^{nd} Fold



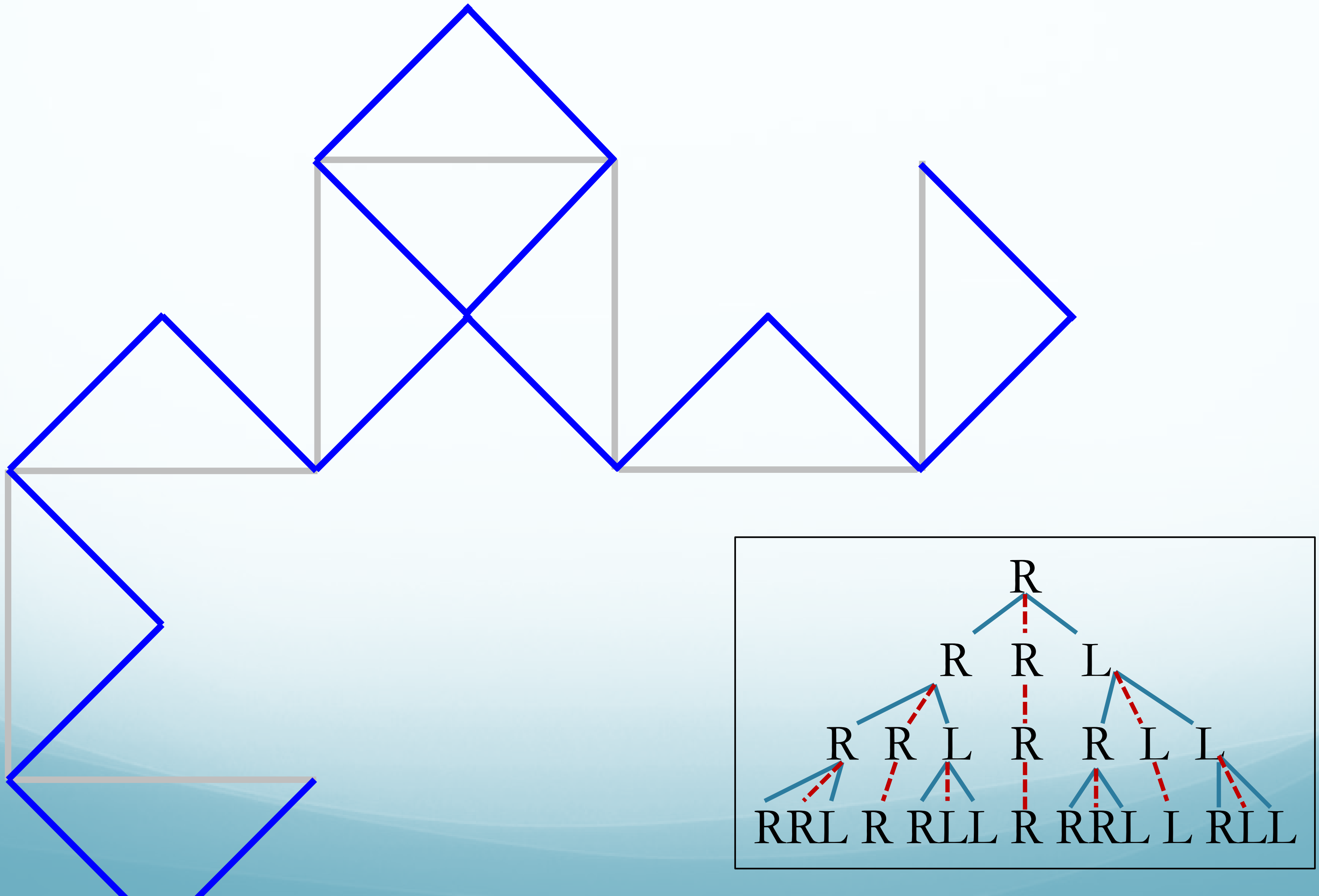


3rd Fold

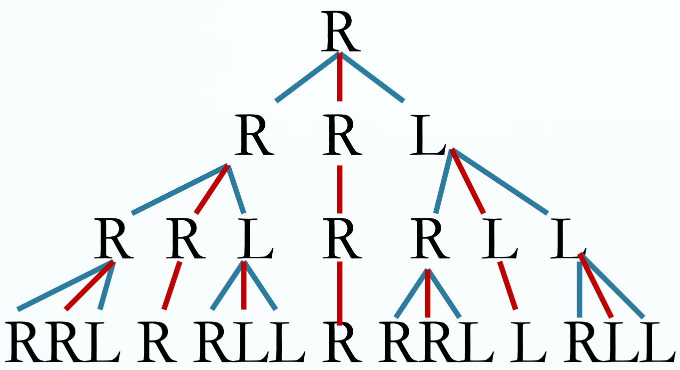


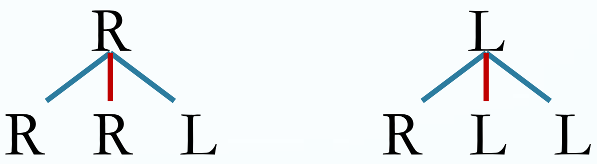


4^{th} Fold

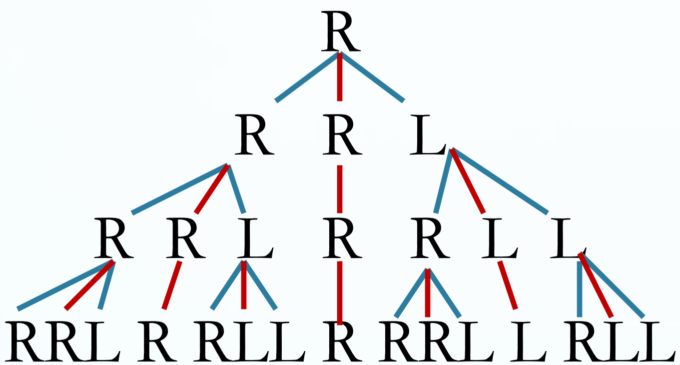


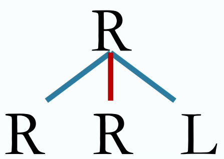
Patterns

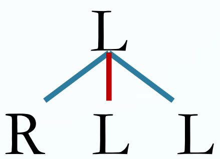


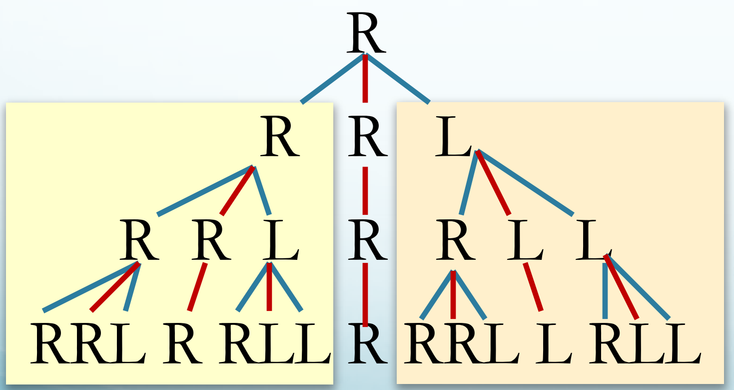


Patterns

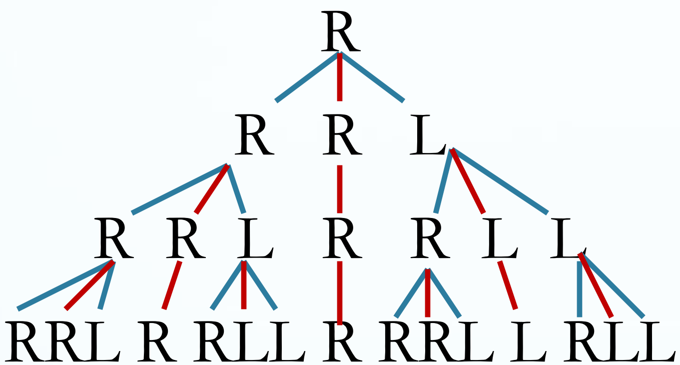

R
R R L
R R L R R L L
RRL R RLL R RRL L RLL

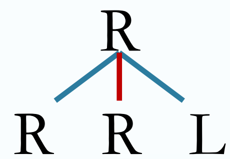

R
R R L

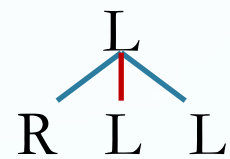

L
R L L

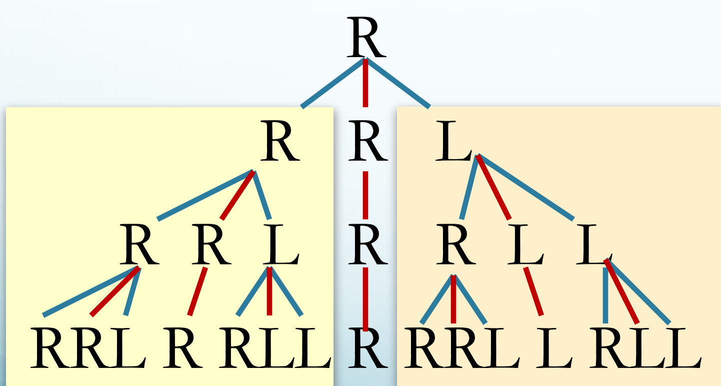

R
R R L
R R L R R L L
RRL R RLL R RRL L RLL

Patterns










```
T(f+1,R) = T(f,R)+R+T(f,L)

T(f+1,L) = T(f,R)+L+T(f,L)
```

Resulting Code

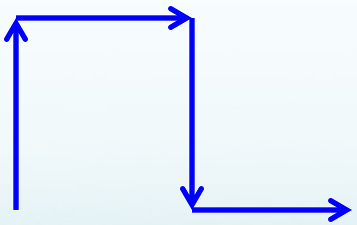
```
T(f+1,'R') = T(f,'R')+'R'+T(f,'L')
T(f+1,'L') = T(f,'R')+'L'+T(f,'L')
```

```
def dragon(fold, root):
    if fold == 1:
        return root
    return dragon(fold-1,'R')+root+dragon(fold-1,'L')
```

RL → NSEW

- Done to simplify project 4
- Conversion:
  - Head north by one length
  - Then execute the turn instructions writing the resulting heading
- Example: RRL
  1. N
  2. E
  3. S
  4. E

  So the result is NESE



Lindenmayer Systems

- How to model biological tree growth and plant architecture?
  - Our trees are constructed node-by-node, serially
  - Nature’s trees grow in parallel
- Parallel rewrite systems

Lindenmayer Systems

- Textually – the dragon curve does this:
  - R => R**R**L
  - **R** => **R**
  - L => R**L**L
  - **L** => **L**
- This is a parallel rewriting system

R**R**L**R**R**L**L => R**R**L**R**R**LL**L**R**R**R**L**L**L**R**L**L

Simple Rewriting Loop

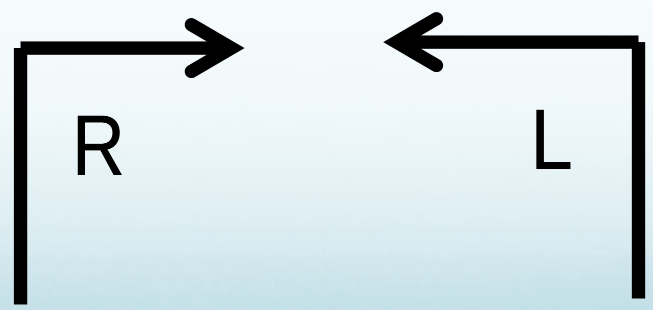
1. Start with a string **w**
2. Replace each character **c** in **w** with a string **s(c)** according to the rules stipulated
3. Repeat

R => R**R**L
L => R**L**L
R => **R**
L => **L**

R => R**R**L
R**R**L => R**R**L**R**RL**L**L
R**R**L**R**R**L**L => R**R**L**R**RL**L**L**R**R**R**L**L**RL**L**L

Drawing the string

- Interpret each character in the string as doing some drawing operation, exactly as in Project 4
- For the L-R string of the dragon curve:
  - Make turn, draw a single line (fixed length)



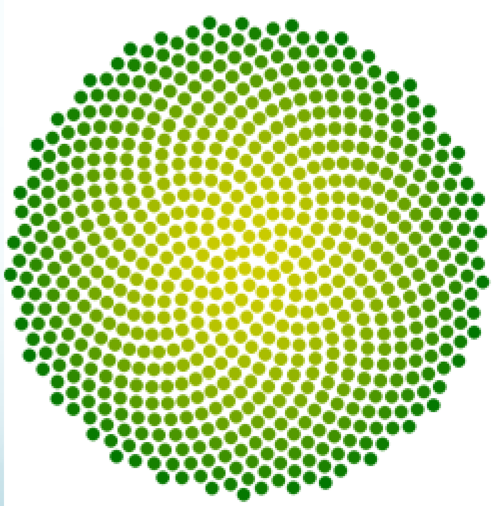
2 Parts

- Part 1: string rewriting system defined
  - Need start string w
  - Need substitution rules
- Part 2: string mapped to a drawing
  - Some characters used to draw simple shape, perhaps a line
  - Some characters used to change direction etc
  - Some characters used to save and restore state (recursively)

L-Systems

- Rewrite + drawing rules yield models of biological shapes and growth
- Some examples from the web that mention Prusinkievicz



Worked Example

- Characters: F + - [ ]
- Initial string: F
- Rewrite rule:
  $$\mathbf{F} \rightarrow \mathbf{F} [- \mathbf{F}] \mathbf{F} [+ \mathbf{F}] [\mathbf{F}]$$
- See http://www.biologie.uni-hamburg.de/b-online/e28_3/lsys.html

Generations 1 and 2

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

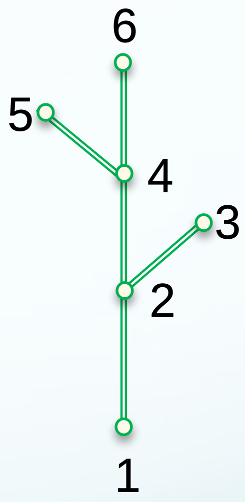
F[-F]F[+F][F][-F[-F]F[+F][F]]F[-F]F[+F][F][+F[-F]F[+F][F]][F[-F]F[+F][F]]

How to draw

- Let's use the turtle drawing program, but instead of only drawing NSEW allow lines at angle α
- Turtle state is (x, y, α): the turtle stands at point (x, y) and looks in direction α, where direction $\alpha = 0$ is North.
- F means the turtle moves forward a fixed distance d
- + means the turtle turns right by a fixed angle β
- - means the turtle turns left by a fixed angle β
- [means the turtle makes a note of its current state
-] means the turtle goes to the most recently noted state (and the note of that state is then deleted)

How to draw

- Turtle state is (x, y, α): the turtle stands at point (x, y) and looks in direction α, where direction $\alpha = 0$ is North.
- F means the turtle moves forward a fixed distance d
- + means the turtle turns right by a fixed angle β
- - means the turtle turns left by a fixed angle β
- [means the turtle makes a note of its current state
-] means the turtle goes to the most recently noted state (and the note of that state is then deleted)



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

Running this System

Generations 2 to 5