### One-Slide Summary

- **Recursive transition networks** and Backus-Naur Form context-free grammars are equivalent formalisms for specifying formal languages.
- **find_closest** is quite powerful. Problem sets?
- L-system fractals are based on a rewriting system that is very similar to BNF grammars.
- We can practice our CS knowledge up to this point to solve problems by **writing recursive procedures**. (It won't take too long.)

---

### Outline

- Briefly: Recursive Transition Networks
  - vs. Backus-Naur Form Grammars
- Problem Sets
  - Revenge of find_closest
- PS3 L-System Fractals
- Solving Problems
  - Problem Representation
  - Important Functions

---

### Recursive Transition Networks

```
ORNATE NOUN ::= OPTARTICLE ADJECTIVES NOUN
ADJECTIVES ::= ADJECTIVE ADJECTIVES
ADJECTIVES ::= ε
OPTARTICLE ::= ARTICLE
OPTARTICLE ::= ε

Recall: the two notations are equivalent.
```

---

### Problem Sets

- **Not** just meant to review stuff you should already know
  - Get you to explore new ideas
  - Motivate what is coming up in the class
- The main point of the PSs is **learning**, not **evaluation**
  - Don’t give up if you can’t find the answer in the book (you won’t solve many problems this way)
  - Do discuss with other students
  - (This is why they are difficult.)

---

### find_closest live demo

- Let's code it together now in PyCharm.
  - **find_closest**(goal, lst, dist)
- Hint: let's make test cases first.
PS2: Question 1
1.i. \( \text{len}([1,2,3][0]) \)
    \[
    = \text{len}(1)
    = \text{error: 1 is not a list!}
    \]
1.q. \( \text{map}(\text{len},[[1,2,3],[4,5],[6]]) \)
    \[
    = [\text{len}([1,2,3]), \text{len}([4,5]), \text{len}([6])]
    = [3, 2, 1]
    \]

PS2: Question 4: Creativity
\[
def \text{seq_comp}\,(\text{lst}):
    \text{if not lst: return []}
    \text{return map(\text{nuc_comp}, \text{lst})}
\]
\[
def \text{seq_comp}\,(\text{lst}):
    \text{return set(\text{map}(\text{nuc_comp}, \text{lst}))}
\]
\[
def \text{seq_comp}\,(\text{lst}):
    \text{return map(\text{nuc_comp}, \text{lst})}
\]

Liberal Arts Trivia: Latin American Studies
• This important leader of Spanish America's successful struggle for independence is credited with decisively contributing to the independence of the present-day countries of Venezuela, Colombia, Ecuador, Peru, Panama, and Bolivia. He defeated the Spanish Monarchy and was in turn defeated by tuberculosis.

Liberal Arts Trivia: Media Studies
• This 1988 book by Herman and Chomsky presented the seminal “propaganda model”, arguing that as news media outlets are run by corporations, they are under competitive pressure. Consider the dependency of mass media news outlets upon major sources of news, particularly the government. If a particular outlet is in disfavor with a government, it can be subtly 'shut out', and other outlets given preferential treatment. Since this results in a loss in news leadership, it can also result in a loss of viewership. That can itself result in a loss of advertising revenue, which is the primary income for most of the mass media (newspapers, magazines, television). To minimize the possibilities of lost revenue, therefore, outlets will tend to report news in a tone more favorable to government and business, and giving unfavorable news about government and business less emphasis.

L-Systems
\[
\text{CommandSequence ::= [ CommandList ]}
\]
\[
\text{CommandList ::= Command CommandList}
\]
\[
\text{Command ::= F}
\]
\[
\text{Command ::= RAngle}
\]
\[
\text{Command ::= OCommandSequence}
\]
**L-System Rewriting**

**Start:** [F]

**Rewrite Rule:**

\[ F \rightarrow [F \text{ O}[R 30 F] F \text{ O}[R -60 F] F] \]

Work like BNF replacement rules, except replace all instances at once!

Why is this a better model for biological systems?

---

**CommandSequence** ::= [ CommandList ]

**CommandList** ::= Command CommandList

**CommandList** ::= Command

**Command** ::= F

**Command** ::= R Angle

**Command** ::= O CommandSequence

---

**The Great Lambda Tree of Ultimate Knowledge and Infinite Power**

(Level 5 with color)

---

*Previous CS 1120 Students:*

- [Tie Dye](#) by Bill Ingram
- [Rose Bush](#) by Jacinta Henry and Rachel Kay
- [A Heart](#)
- [Tree Outside My Window](#)
PS3 - Fractals

- In addition to completing the problem set, each team will submit its prettiest fractal.
- The class will then vote for favorites, and the authors of the favorites will receive extra credit.
- No Photoshop, etc. All PS3.
  - You just change the rules:
    • $F \rightarrow [F O[R30 F] F O[R-60 F] F] \# \text{one fractal}$
    • $F \rightarrow [O[R60 F] F F O[R45 F]] \# \text{a new one!}$

Hints

- Remember our strategy!
- Be optimistic!
  - Assume that you can write “count_fives”
  - So the recursive case will work out
- Identify the smallest input you can solve
  - The base case
- How would you combine answers
  - From the current call (usually $\text{lst}[0]$)
  - And the result of the recursive call (on $\text{lst}[1:]$)
- Be creative! There are usually many solutions.

Procedure Practice

- For the rest of this class, we will be practicing writing recursive procedures together.
- Write a procedure count-fives that takes as input a list of numbers. It returns the number of fives contained in its input list.
  - $\text{count_fives}([1, 2, 3, 4, 5,]) \rightarrow 1$
  - $\text{count_fives}([5, -5, 5, 7]) \rightarrow 2$
  - $\text{count_fives}([ ]) \rightarrow 0$
  - $\text{count_fives}([8, 6, 7, 5, 3, 0, 9]) \rightarrow 1$

Three versions of count_fives

```python
def count_fives(lst):
    if not lst: return 0
    if lst[0] == 5: return 1 + count_fives(lst[1:])
    return count_fives(lst[1:])
```

```python
def count_fives(lst):
    if not lst: return 0
    return (1 if lst[0] == 5 else 0) + count_fives(lst[1:])
```

```python
def count_fives(lst):
    return len(filter(lambda x : x == 5, lst))
```

All work fine! How are they different?

Liberal Arts Trivia: Medicine

- This vector-borne infectious disease is caused by protozoan parasites. It is widespread in tropical regions, such as sub-Saharan African. Each year there are about 515 million cases of it, killing between one and three million people. No formal vaccine is available. Classic symptoms include sudden coldness followed by rigor and then fever and sweating.

Liberal Arts Trivia: Cognitive Psychology

- This American psychologist coined the term Cognitive Psychology in his late 1960's book of the same name. He was critical of linear programming models of psychology, felt that psychology should address everyday concerns, and respected the direct perception theories of J.J. And Eleanor Gibson. He headed the APA task force that reviewed The Bell Curve.
Liberal Arts Trivia: Accounting

- In this bookkeeping system, each transaction is recorded in at least two accounts. Each transaction results in one account being debited and another account being credited, with the total debits equal to the total credits. Luca Pacioli, a monk and collaborator of Leonardo da Vinci, is called the “father of accounting” because he published a usable, detailed description of this system.

contains

- Write a procedure `contains` that takes two arguments: an element and a list. It returns True if the list contains the given element, False otherwise.
  - `contains(5, [1, 2, 3, 4])` -> False
  - `contains(5, [2, 3, 4, 5])` -> True
  - `contains([], [1, 2, 3])` -> False
  - `contains([], [1, 2, []])` -> True
  - `contains(1, [2, [], 1])` -> True
  - `contains(3, [])` -> False

contains explained

```python
def contains(elt, lst):
    if not lst: return False
    elif lst[0] == elt: return True
    else: return contains(elt, lst[1:])
```

- All work fine! How are they different?

common_elt

- Write a procedure `common_elt` that takes two lists as arguments. It returns True if there is a common element contained in both lists, False otherwise.
  - `common_elt([1, 2, 3], [3, 4, 5])` -> True
  - `common_elt([1, 2, 3], [4, 5, 6])` -> False
  - `common_elt([1, 2], [0, 0, 0, 1])` -> True
  - `common_elt([1], [])` -> False
  - `common_elt([], [1,2,3])` -> False
  - `common_elt([], [])` -> False
  - Hint: You can use contains.

common_elt?

```python
def common_elt(lst1, lst2):
    if not lst1: return False
    elif contains(lst1[0], lst2): return True
    else: return common_elt(lst1[1:], lst2)
```

- This version is super slow!

```python
def common_elt(lst1, lst2):
    if (not lst1) or (not lst2): return False
    return (lst1[0] == lst2[0]) or common_elt(lst1, lst2[1:]) or \
       common_elt(lst1[1:], lst2)
```

- All work! How are they different?

zero2hero

- Write a procedure `zero2hero` that takes as input a list of strings. It returns the same list in the same order, but every element that used to be “zero” is now “hero”.
  - `zero2hero(["a", "zero", "b", "jercules"])` -> ["a", "hero", "b", "jercules"]
  - `zero2hero(["zorro"])` -> ["zorro"]
  - `zero2hero(["zero", "zero", "one", "zero"])` -> ["hero", "hero", "one", "hero"]

```python
def zero2hero(lst):
    return filter(lambda e: e != "zero", lst)
```

- All work! How are they different?
def zero2hero(lst):
    if not lst: return []
    if lst[0] == "zero":
        return ["hero"] + zero2hero(lst[1:])
    return [lst[0]] + zero2hero(lst[1:]), zero2hero

def zero2hero(lst):
    if not lst: return []
    return ["hero" if lst[0] == "zero" else lst[0]] +
    zero2hero(lst[1:])

def zero2hero(lst):
    return ["hero" if x == "zero" else x for x in lst]

tiny_squares
• Write a procedure tiny_squares that takes as input a list of numbers. It returns a list of the squares of those numbers (in the same order), but any square above 100 is not included in the output.
  - tiny_squares([8, 9, 10, 11, 12])
    • [64, 81, 100]
  - tiny_squares([-2, 12, 4, 77, 5])
    • [4, 16, 25]
  - tiny_squares([3, 2, 1, 100])
    • [9, 4, 1]

def tiny_squares(lst):
    if not lst: return []
    if abs(lst[0]) <= 10:
        return [lst[0] * lst[0]] + tiny_squares(lst[1:])
    return tiny_squares(lst[1:]), tiny_squares

def tiny_squares(lst):
    return [x*x for x in lst if x*x <= 100] # so awesome!
def tiny_square(lst):
    return map(lambda x : x * x, 
        filter(lambda y : abs(y) <= 10, lst))

every
• Write a procedure every that takes two elements, a predicate and a list. (A predicate is a function that takes an element and returns True or False.) The procedure every returns True if the predicate returns True on each one of its elements. It returns False if even one element does not pass the test. On the empty list, every returns True.
  - every(lambda (x) :(x >3), [4,5,6]) -> True
  - every(lambda (x) :(x > 3), [9,1,1]) -> False
  - every(lambda (x) :(x == 3), [3,3]) -> True
  - every(lambda (x) :(x < y), [ ]) -> True

def every(pred, lst):
    if not lst: return True
    if pred(lst[0]): return every(pred, lst[1:])
    return False
def every(pred, lst):
    if not lst: return True
    return pred(lst[0]) and every(pred, lst[1:])

count_false
• Write a function count_false that takes two arguments: a predicate and a list. It returns the number of elements in the list for which the predicate returns False.
  - count_false(lambda x: x > 3, [1,2,3,4,5,6,7,8,9,10])
    • 3
  - count_false(lambda x: x > 3, [-1, -2, -3, -4])
    • 4
  - count_false(lambda x: x == "a", ["a", "b", "b", "a"])
    • 2
  - count_false(lambda x: x == "a", [ ])
    • 0

def every(pred, lst):
    if not lst: return False
    if pred(lst[0]): return every(pred, lst[1:])
    return False
def every(pred, lst):
    if not lst: return True
    return pred(lst[0]) and every(pred, lst[1:])
def count_false(pred, lst):
    if not lst: return 0
    if pred(lst[0]): return count_false(pred, lst[1:])
    return 1 + count_false(pred, lst[1:])

def count_false(pred, lst):
    return len(filter(lambda x: not pred(x), lst))
    # return len([x for x in lst if not pred(x)])

def count_false(pred, lst):
    return len(lst) - len(filter(pred, lst))

Questions

• We're more or less done with procedure practice in class.
• Test questions may look a lot like this.
• If you're still having trouble with these, come see Wes or a TA. We'll make up problems for you to practice and go over writing recursive procedures with you.
• Time permitting, ask me anything now.

Homework

• Problem Set 3
  - Reading!