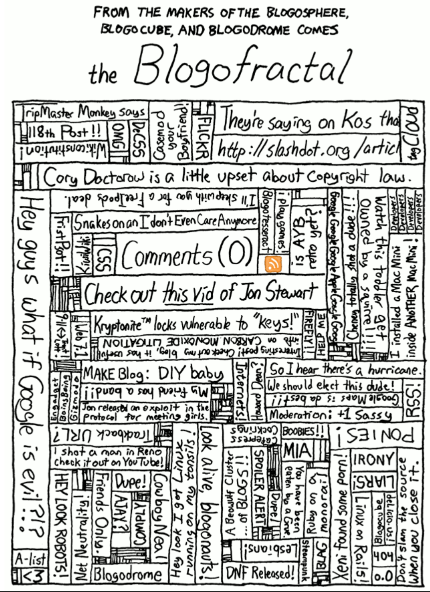
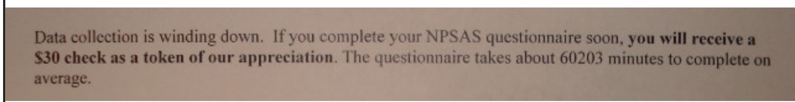
L-System Fractals & Procedure Practice



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.)

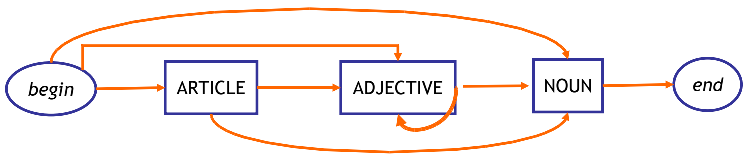


Outline

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

Recursive Transition Networks

ORNATE NOUN



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.)

PS2: Question 1

- 1.i. (list-length (list-append (list 1 2 3) 4))

```
> (list-append (list 1 2 3) 4)
(1 2 3 . 4) ;; this is not a list!

> (list-length (list-append (list 1 2 3) 4))
length: expects argument of type <proper list>;
  given (1 2 3 . 4)
```

PS2: Question 4

Why is

```
(define (higher-card? card1 card2)
  (> (card-rank card1) (card-rank card2)
```

better than

```
(define (higher-card? card1 card2)
  (> (car card1) (car card2))
```

?

PS2: Question 8

- 8. How long is **analyze-flop-situation**?

```
(let ((current-deck (remove-cards (append hole1
                          hole2 community) full-deck)))
  (map (lambda (turn-card)
    (map (lambda (outs) (cons turn-card outs))
      (analyze-turn-situation hole1 hole2 (cons turn-
    card community)))) current-deck)))
```

  - Current-deck is the initial deck of 52 cards with 7 already dealt, leaving 45.
  - We call analyze-turn 45 times (slow!), each time we putting the answer in a list (fast!), so 45x.

PS2: Question 9, 10

- Predict how long it will take
- Identify ways to make it faster

Most of next week and much of many later classes will be focused on how computer scientists **predict** how long programs will take, and on how to **make them faster**.

Can we do better?

This is what we used in PS2 for our Poker-Bot:

```
(define (find-best-hand hole-cards community-cards)
  (car (sort (possible-hands hole-cards
                             community-cards))
             higher-hand?))
```

But didn't we learn something in the last class for finding the "closest" or "best" element in a list?

Recall From Last Time

```
(define (find-closest goal lst closeness)
  (if (= 1 (length lst))
      (car lst)
      (pick-closest closeness goal (car lst)
                    (find-closest goal (cdr lst) closeness))))

(define (pick-closest closeness goal num1 num2)
  (if (< (closeness goal num1)
         (closeness goal num2))
      num1
      num2))
```

We could use these to find the best hand!

find-bestest

```
(define (find-bestiest lst bestiness)
  (if (= 1 (length lst))
      (car lst)
      (pick-bestier bestiness
                    (car lst)
                    (find-bestiest (cdr lst) bestiness))))

(define (pick-bestier bestiness num1 num2)
  (if (bestiness num1 num2)
      num1
      num2))
```

This used to be (< (dist num1 goal) (dist num2 goal))

find-best-hand

```
(define (find-bestiest lst bestiness)
  (if (= 1 (length lst)) (car lst)
      (pick-bestier bestiness
                    (car lst)
                     (find-bestiest (cdr lst) bestiness))))
(define (pick-bestier bestiness num1 num2)
  (if (bestiness num1 num2) num1 num2))
```

(define (find-best-hand lst)
(find-bestest lst higher-hand?))

Next week: how much better is this?
At home: convince yourself that they get the same answer.

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.

PS3: Lindenmayer System Fractals

L-Systems

CommandSequence ::= **(** *CommandList* **)**
CommandList ::= *Command CommandList*
CommandList ::=
Command ::= **F**
Command ::= **R***Angle*
Command ::= **O***CommandSequence*

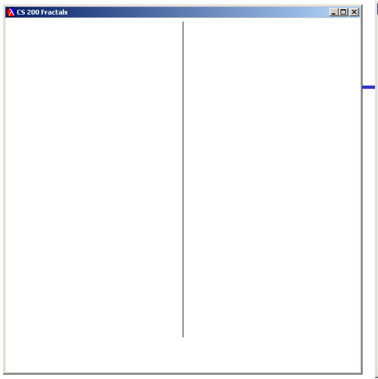
L-System Rewriting

CommandSequence ::= **(** *CommandList* **)**
CommandList ::= *Command CommandList*
CommandList ::=
Command ::= **F**
Command ::= **R***Angle*
Command ::= **O***CommandSequence*

Start: (F)
Rewrite Rule:
F → (F O(R30 F) F O(R-60 F) F)

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

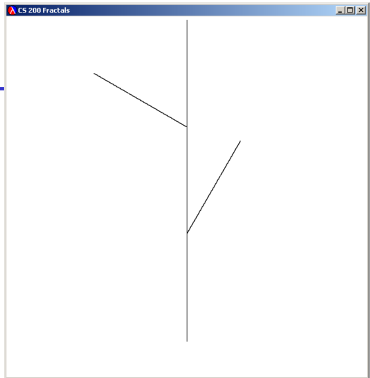
Why is this a better model for biological systems?



Level 0

Start: (F)

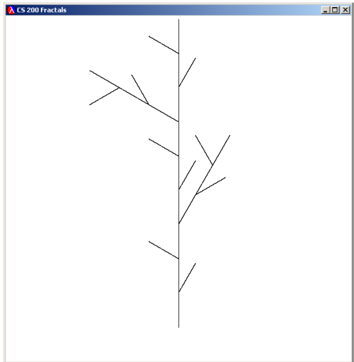
(F)



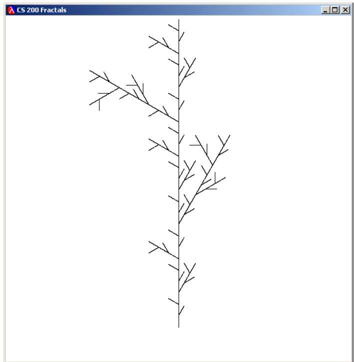
Level 1

F → (F O(R30 F) F O(R-60 F) F)

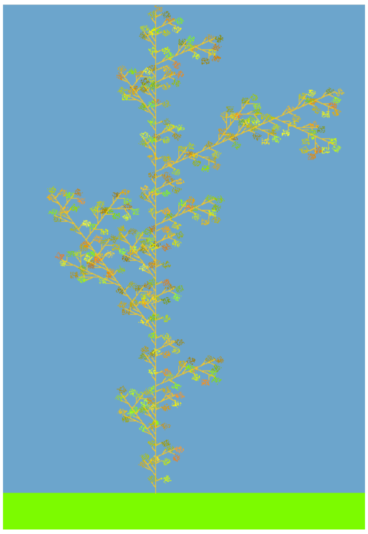
(F O(R30 F) F O(R-60 F) F)



Level 2



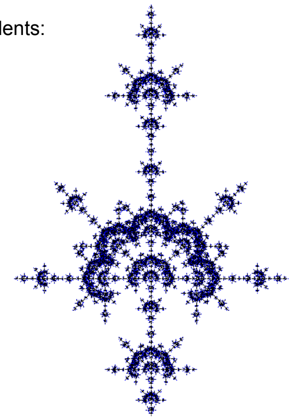
Level 3



The Great Lambda Tree of Ultimate Knowledge and Infinite Power

(Level 5 with color)

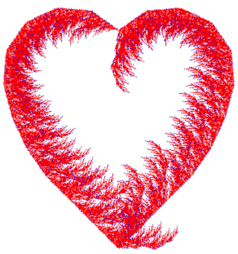
Previous CS 1120 Students:



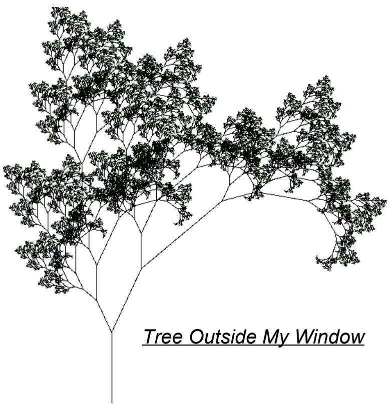
Tie Dye by Bill Ingram

Rose Bush by Jacintha Henry and Rachel Kay

Previous CS 1120 Students:



A Heart



Tree Outside My Window

PS3 - Fractals

- In addition to completing the problem set, each time will submit their 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 → (F O(R30 F) F O(R-60 F) F) ;; one fractal
    - F → (O(R60 F) F F O(R45 F)) ;; a new one!

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.
  - (count-fives (list 1 2 3 4 5)) -> 1
  - (count-fives (list 5 -5 5 7)) -> 2
  - (count-fives (list)) -> 0
  - (count-fives (list 8 6 7 5 3 0 9)) -> 1

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 the car of the list)
  - And the result of the recursive call
- Be creative! There are usually many solutions.

Two versions of count-fives

```
(define (count-fives lst)
  (if (null? lst)
      0
      (if (eq? (car lst) 5)
          (+ 1 (count-fives (cdr lst)))
          (count-fives (cdr lst)))))

(define (count-fives lst)
  (if (null? lst) 0
      (+    (if (eq? (car lst) 5) 1 0)
            (count-fives (cdr lst)))))
```

Both 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 #t if the list contains the given element, #f otherwise.
  - (contains? 5 (list 1 2 3 4)) -> #f
  - (contains? 5 (list 2 3 4 5)) -> #t
  - (contains? null (list 1 2 3)) -> #f
  - (contains? 1 (list 2 null 1)) -> #t
  - (contains? 3 (list)) -> #f

contains explained

```
(define (contains? elt lst)
  (if (null? lst)
    #f
    (if (eq? elt (car lst))
      #t
      (contains? elt (cdr lst)))))

(define (contains? elt lst)
  (if null? lst) #f
  (or     (eq? elt (car lst))
          (contains? elt (cdr lst))))
```

Both work fine! How are they different?

common-elt?

- Write a procedure **common-elt?** that takes two lists as arguments. It returns #t if there is a common element contained in both lists, #f otherwise.
  - (common-elt? (list 1 2 3) (list 3 4 5)) -> #t
  - (common-elt? (list 1 2 3) (list 4 5 6)) -> #f
  - (common-elt? (list 1 2) (list 0 0 0 1)) -> #t
  - (common-elt? (list 1) null) -> #f
  - (common-elt? null (list 1 2 3)) -> #f
  - (common-elt? (list 1) (list 1 2 3)) -> #t
- Hint: contains?

common-elt?

```
(define (common-elt? lst1 lst2)
  (if (null? lst1) #f
    (if (contains? (car lst1) lst2)
      #t
      (common-elt? (cdr lst1) lst2))))

(define (common-elt? lst1 lst2)
  (if (or (null? lst1) (null? lst2)) #f
  (or (eq? (car lst1) (car lst2))
    (common-elt? lst1 (cdr lst2))
    (common-elt? (cdr lst1) lst2)))))
    ;; this version is super slow!
```

Both work! How are they different?

zero-to-hero

- Write a procedure **zero-to-hero** 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”.
  - (zero-to-hero (list “a” “zero” “b” “jercules”))
    - (“a” “hero” “b” “jercules”)
  - (zero-to-hero (list “zorro”))
    - (“zorro”)
  - (zero-to-hero (list “zero” “zero” “one” “zero”))
    - (“hero” “hero” “one” “hero”)

zero-to-hero

```
(define (zero-to-hero lst)
  (if (null? lst) null
    (if (eq? (car lst) “zero”))
      (cons “hero” (zero-to-hero (cdr lst)))
      (cons (car lst) (zero-to-hero (cdr lst))))))

(define (zero-to-hero lst)
  (map (lambda (x)
          (if (eq? x “zero”) “hero” x))
    lst))   ;; learn map if you haven't yet!
```

Both work! How are they different?

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 (list 8 9 10 11 12))
    - (64 81 100)
  - (tiny-squares (list -2 12 4 77 5))
    - (4 16 25)
  - (tiny-squares (list 3 2 1 100))
    - (9 4 1)

tiny-squares

```
(define (tiny-squares lst)
  (if (null? lst) null
    (if (<= (car lst) 10)
      (cons  (* (car lst) (car lst))
             (tiny-squares (cdr lst)))
      (tiny-squares (cdr lst)))))
```

Both work! How are they different?

```
(define (tiny-squares lst)
  (filter (lambda (squared) (<= squared 100))
    (map (lambda (x) (* x x)) lst)))
    ;; this ordering: map first, then filter!
```


every

- Write a procedure **every** that takes two elements, a predicate and a list. (Recall that a predicate is a function that takes an element and returns #t or #f.) The procedure **every** returns #t if the predicate returns #t on each one of its elements. It returns #f if even one element does not pass the test. On the empty list, every returns #t.
  - (every (lambda (x) (> x 3)) (list 4 5 6)) -> #t
  - (every (lambda (x) (> x 3)) (list 9 1 1)) -> #f
  - (every (lambda (x) (eq? x 3)) (list 3 3)) -> #t
  - (every (lambda (x) (< x y)) (list)) -> #t

every heartbeat belongs to you!

```
(define (every pred lst)
  (if (null? lst) #t
    (if (pred (car lst))
      (every pred (cdr lst))
      #f)))
```

Both work! How are they different?

```
(define (every pred lst)
  (eq?  (list-length (filter pred lst))
        (list-length lst)))
```


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 #f.
  - (count-false (lambda (x) (> x 3)) (list 1 2 3 4 5 6 7 8 9 10))
    - 3
  - (count-false (lambda (x) (> x 3)) (list -1 -2 -3 -4))
    - 4
  - (count-false (lambda (x) (eq? x “a”)) (list “a” “b” “a”))
    - 1
  - (count-false (lambda (x) (eq? x “a”)) (list))
    - 0

count-false

```
(define (count-false pred lst)
  (if (null? lst) 0
    (if (pred (car lst))
      (count-false pred (cdr lst))
      (+ 1 (count-false pred (cdr lst)))
```

All work! How are they different?

```
(define (count-false pred lst)
  (list-length (filter (lambda (x) (not (pred x))) lst))
```

```
(define (count-false pred lst)
  (-  (list-length lst)
    (list-length (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

- Read Course Book Chapter 6 before Monday
- Start in on PS3 (due Wednesday)
- Start on reading Chapter 7