Lecture 11: Generalizing List Procedures

cs1120 Fall 2009
David Evans
http://www.cs.virginia.edu/cs1120

Menu

- Using list-map
- PS3
- Generalizing List Procedures

I'll be away next week (no office hours, but I will have email).
Prof. Wes Weimer will lead the classes.

list-map

From last class:

```
(define (list-map f p)
  (if (null? p)
      null
      (cons (f (car p))
            (list-map f (cdr p)))))
```

Equivalent to the built-in procedure **map** (except **map** can work on more than one list).

Maps are Useful

From PS2:

```
(define full-deck
 (list-flatten
  (list-map
   (lambda (rank) (list-map
                   (lambda (suit) (make-card rank suit))
                   (list Hearts Diamonds Clubs Spades)))
   (list 2 3 4 5 6 7 8 9 10 Jack Queen King Ace))))
```

Image: http://www.codinghorror.com/blog/archives/001008.html

Maps are Useful

From PS2:

```
(define (choose-n n lst)
  ;; operands: a number n and a list (of at least n elements)
  ;; result: evaluates to a list of all possible was of choosing n elements from lst
  (if (= n 0)
      (list null)
      (if (= (list-length lst) n)
          (list lst) ; must use all elements
          (list-append
           (choose-n n (cdr lst)) ;; all possibilities not using first element
           (list-map (lambda (clst) (cons (car lst) clst))
                     (choose-n (- n 1) (cdr lst)))))))
```


Maps are Useful!

From PS1:

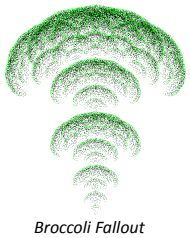
```
(define (select-mosaic-tiles samples tiles)
  (map2d find-best-match samples))
```

How can we define **map2d** that maps a list of lists with the input procedure?

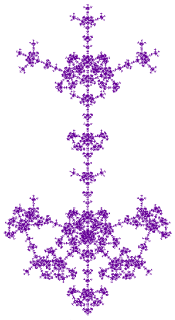
```
(define (map2d f p)
  (list-map
   (lambda (inner-list) (list-map f inner-list))
   p))
```

PS3: Lindenmayer System Fractals

Aristid Lindenmayer (1925-1989)



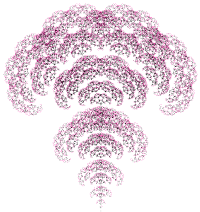
Broccoli Fallout
by Paul DiOrio, Rachel Phillips



Purple Arrow
by Rachel Lathbury and Andrea Yoon

L-Systems

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



Cherry Blossom
by Ji Hyun Lee, Wei Wang

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!

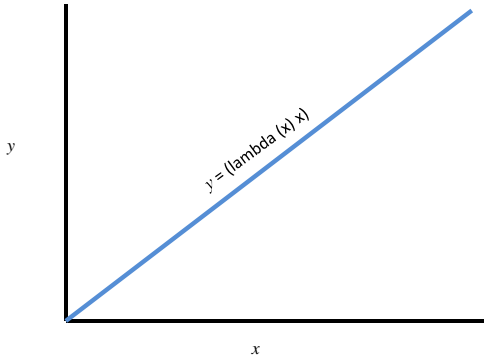
Why is this a better model for biological systems?

Maps are Useful!!

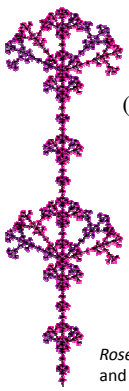
PS3 Question 5:

```
(define (rewrite-lcommands lcommands replacement)
  (flatten-commands
    (map
      ; Procedure to apply to each command
      lcommands)))
```


Drawing Functions

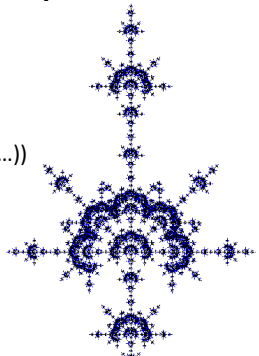


Drawing Arbitrary Curves



Rose Bush by Jacintha Henry and Rachel Kay

(*x*, *y*) = (lambda (t)
(make-point ...))



Tie Dye by Bill Ingram

Curves

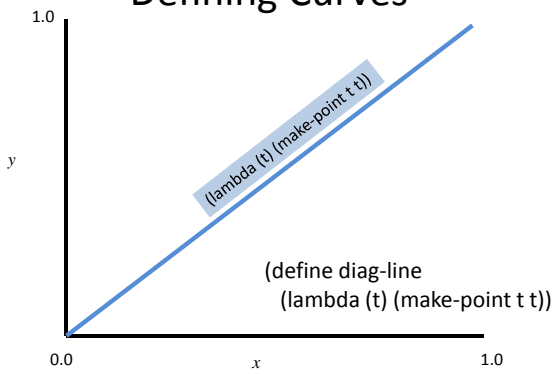
- A *curve* c is the set of points that results from applying *c* to every real number *t* value between 0.0 and 1.0.

How many points?

Infinitely many!

We can draw an approximate representation of the curve by sampling some of the points.

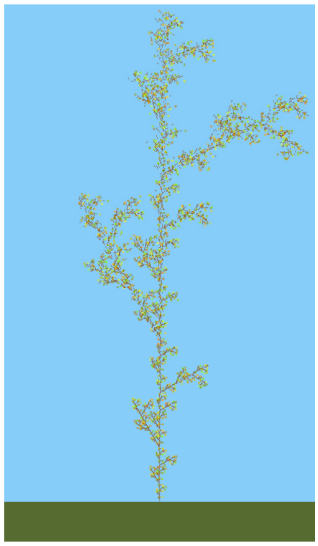
Defining Curves



Drawing Curves

```
(define (draw-curve-points curve n)
  (define (draw-curve-worker curve t step)
    (if (<= t 1.0)
        (begin
          (window-draw-point (curve t))
          (draw-curve-worker curve (+ t step) step)))
  (draw-curve-worker curve 0.0 (/ 1 n)))
```

Does this recursive definition have a base case?



PS3 Questions?

The Great Lambda Tree of Ultimate Knowledge and Infinite Power

(Level 5 with color + randomness)

Generalizing List Procedures

```
(define (list-map f p)
  (if (null? p)
      null
      (cons (f (car p))
            (list-map f (cdr p)))))
```

```
(define (list-sum p)
  (if (null? p)
      0
      (+ (car p) (list-sum (cdr p)))))
```

```
(define (is-list? p)
  (if (null? p)
      true
      (if (pair? p) (is-list? (cdr p)) false)))
```

```
(define (list-length p)
  (if (null? p)
      0
      (+ 1 (list-length (cdr p)))))
```


Similarities and Differences

```
(define (list-map f p)
  (if (null? p)
      null
      (cons (f (car p))
            (map f (cdr p)))))
```

```
(define (list-sum p)
  (if (null? p)
      0
      (+ (car p)
         (sumlist (cdr p)))))
```

```
(define (list-cruncher ? ... ? p)
  (if (null? p)
      base result
      (combiner (car p)
                (recursive-call ... (cdr p)))))
```

list-cruncher

```
(define (list-cruncher baseres carproc combiner p)
   (if (null? p) baseres
      (combiner
       (carproc (car p))
       (list-cruncher baseres carproc combiner (cdr p)))))

(define (list-sum p)
  (list-cruncher 0 (lambda (x) x) + p))

(define (list-map f p)
  (list-cruncher null f cons p))
```

Can list-cruncher crunch length?

```
(define (list-cruncher baseres carproc combiner p)
   (if (null? p) baseres
      (combiner
       (carproc (car p))
       (list-cruncher baseres carproc combiner (cdr p)))))
```

```
(define (list-length p)
  (if (null? p) 0
      (+ 1 (list-length (cdr p)))))
```

Can list-cruncher crunch length?

```
(define (list-cruncher baseres carproc combiner p)
   (if (null? p) baseres
      (combiner
       (carproc (car p))
       (list-cruncher baseres carproc combiner (cdr p)))))
```

```
(define (list-length p)
  (if (null? p) 0
      (+ 1 (list-length (cdr p)))))
```

```
(define (list-length p)
  (list-cruncher 0 (lambda (x) 1) + p))
```

Can list-cruncher crunch is-list??

```
(define (list-cruncher baseres carproc combiner p)
   (if (null? p) baseres
      (combiner
       (carproc (car p))
       (list-cruncher baseres carproc combiner (cdr p)))))
```

```
(define (is-list? p)
   (if (null? p)
      true
      (if (pair? p) (is-list? (cdr p)) false)))
```

No! If p is non-null, (car p) is always evaluated by list-cruncher so it produces an error if p is not a list.

Crunchers vs. Accumulators

```
(define (list-cruncher baseres carproc combiner p)
   (if (null? p) baseres
      (combiner
       (carproc (car p))
       (list-cruncher baseres carproc combiner (cdr p)))))

(define (list-accumulate f base p)
   (if (null? p) base
      (f (car p) (list-accumulate f base (cdr p)))))
```

```
(define (list-sum p)
  (list-cruncher 0 (lambda (x) x) + p))
```

```
(define (list-sum p)
  (list-accumulate + 0 p))
```


Crunchers vs. Accumulators

```
(define (list-cruncher baseres carproc combiner p)
   (if (null? p) baseres
      (combiner
       (carproc (car p))
       (list-cruncher baseres carproc combiner (cdr p)))))

(define (list-accumulate f base p)
   (if (null? p) base
      (f (car p) (list-accumulate f base (cdr p)))))
```

Is there any procedure that can be defined using list-accumulate that can't be defined using list-cruncher?

Crunchers vs. Accumulators

```
(define (list-cruncher baseres carproc combiner p)
  (if (null? p) baseres
      (combiner
       (carproc (car p))
       (list-cruncher baseres carproc combiner (cdr p)))))

(define (list-accumulate f base p)
  (if (null? p) base
      (f (car p) (list-accumulate f base (cdr p)))))
```

No! Proof-by-construction:

```
(define (list-accumulate f base p)
  (list-cruncher base (lambda (x) x) f p))
```


Crunchers vs. Accumulators

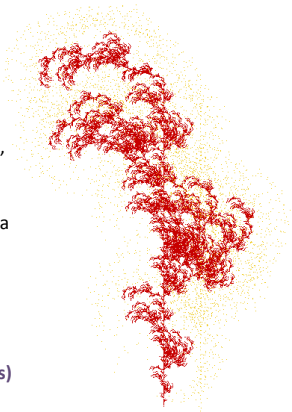
```
(define (list-cruncher baseres carproc combiner p)
  (if (null? p) baseres
      (combiner
       (carproc (car p))
       (list-cruncher baseres carproc combiner (cdr p)))))

(define (list-accumulate f base p)
  (if (null? p) base
      (f (car p) (list-accumulate f base (cdr p)))))
```

Gold star bonus: Is there any procedure that can be defined using list-cruncher that can't be defined using list-accumulate?

Charge

- You should be able to:
  - Define, understand, and use recursive procedures on lists
  - Including: list-map, list-length, list-sum, list-append, list-filter
  - Understand and define procedures using list-cruncher, list-accumulate, or a similar procedure
- PS3 due Wednesday
- Upcoming help hours:
  - Sunday 6-8:30pm (Olsson 001)
  - **Monday noon-1:30pm (Thorton Stacks)**
  - Monday 4-6:30pm (Small Hall)



where the red fern grows
by Jessica Geist, Ellen Clarke