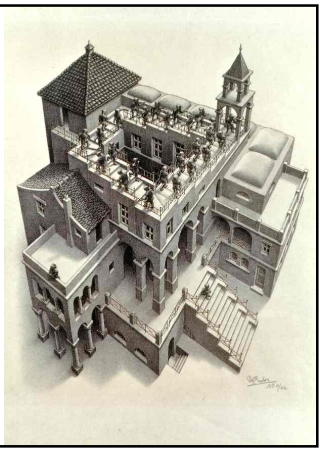
Lecture 8: Recursing Lists



CS150: Computer Science
University of Virginia
Computer Science

Defining Recursive Procedures

1. Be optimistic! Assume you can solve it.
2. Think of the simplest version of the problem, something you can already solve. (This is the **base case**.)
3. Consider how you would solve an instance of the problem using the result for a slightly smaller instance. (**recursive case**)
4. Combine them to solve the problem.

Defining Recursive Procedures on Lists

1. Be ***very*** optimistic!
2. Think of the simplest version of the problem (almost always null), something you can already solve. (**base case**)
3. Consider how you would solve an instance of the problem using the result for a slightly smaller instance (the cdr of the list). (**recursive case**)
4. Combine them to solve the problem.

---

list?

```
(define (list? p)
  (if (null? p)
      #t
      (if (pair? p)
          (list? (cdr p))
          #f)))
```

Tracing list?

To enable tracing: (require (lib "trace.ss"))

```
> (trace list?)
(list?)
> (list? null)
|(list? ())
|#t
#t
```

```
> (list? (list 1 2 3))
|(list? (1 2 3))
|(list? (2 3))
|(list? (3))
|(list? ())
|#t
#t
```

---

sumlist

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

Tracing sumlist

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

> (trace +)
> (sumlist (list 1 2 3 4))
|(+ 4 0)
|4
|(+ 3 4)
|7
|(+ 2 7)
|9
|(+ 1 9)
|10
10
```

map

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

Tracing map

```
> (map (lambda (x) (* x 2)) (list 1 2 3))
|(map #<procedure> (1 2 3))
| (map #<procedure> (2 3))
| |(map #<procedure> (3))
| | (map #<procedure> ())
| | ()
| |(6)
| (4 6)
|(2 4 6)
(2 4 6)
```

Similarities and Differences

```
(define (map f p)                   (define (sumlist p)
  (if (null? p)                       (if (null? p)
      null                                0
      (cons (f (car p))                   (+ (car p)
            (map f (cdr p)))))                (sumlist (cdr p)))))

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

Similarities and Differences

```
(define (map f p)                   (define (sumlist p)
  (if (null? p)                       (if (null? p)
      null                                0
      (cons (f (car p))                   (+ (car p)
            (map f (cdr p)))))                (sumlist (cdr p)))))

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

list-cruncher

```
(define (list-cruncher base proc combiner lst)
  (if (null? lst)
      base
      (combiner (proc (car lst))
                (list-cruncher base proc combiner
                               (cdr lst)))))

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

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

Can list-cruncher crunch length?

```
(define (list-cruncher base proc combiner lst)
  (if (null? lst)
      base
      (combiner (proc (car lst))
                (list-cruncher base proc combiner
                               (cdr lst)))))
```

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

Can list-cruncher crunch length?

```
(define (list-cruncher base proc combiner lst)
  (if (null? lst)
      base
      (combiner (proc (car lst))
                (list-cruncher base proc combiner
                               (cdr lst)))))
```

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

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

Can list-cruncher crunch list??

```
(define (list-cruncher base proc combiner lst)
  (if (null? lst)
      base
      (combiner (proc (car lst))
                (list-cruncher base proc combiner
                               (cdr lst)))))
```

```
(define (list? p)
  (if (null? p) #t
      (if (pair? p) (list? (cdr p))
          #f)))
```

list-or-not-cruncher

```
(define (list-or-not-cruncher base nonlist
                              proc combiner lst)
  (if (null? lst)
      base
      (if (not (pair? lst))
          nonlist
          (combiner (proc (car lst))
                    (list-or-not-cruncher base nonlist proc combiner
                                          (cdr lst))))))
```

```
(define (list? p)
  (list-or-not-cruncher
    #t #f (lambda (x) x) (lambda (f r) r) p))
```

This works, but is not an elegant or simple way of defining list?!

find-closest-number

Define find-closest-number that takes two parameters, a goal and a list of numbers, and produces the number in the list numbers list that is closest to goal:

```
> (find-closest-number 150 (list 101 110 120 157 340 588))
157
> (find-closest-number 12 (list 1 11 21))
11
> (find-closest-number 12 (list 95))
95
```

Find Closest Number

Be optimistic!

Assume you can define:

(find-closest-number goal numbers)

that finds the closest number to goal from the list of numbers.

What if there is one more number?

Can you write a function that finds the closest number to match from new-number and numbers?

Find Closest

Strategy:

If the new number is better, than the best match with the other number, use the new number. Otherwise, use the best match of the other numbers.

Optimistic Function

```
(define (find-closest goal-number numbers)
  ;; base case missing for now
  (if (< (abs (- goal (car numbers)))
         (abs (- goal
                 (find-closest-number
                  goal (cdr numbers)))))
      (car numbers)
      (find-closest-number goal (cdr numbers))))
```

Defining Recursive Procedures

2. Think of the simplest version of the problem (almost always null), something you can already solve. (**base case**)

Is null the base case for find-closest-number?

The Base Case

```
(define (find-closest-number goal numbers)
  (if (= 1 (length numbers))
      (car numbers)
      (if (< (abs (- goal (first numbers)))
             (abs (- goal
                     (find-closest-number
                      goal (cdr numbers)))))
          (car numbers)
          (find-closest-number (cdr numbers))))
```

```
(define (find-closest-number goal numbers)
  (if (= 1 (length numbers))
      (car numbers)
      (if (< (abs (- goal (car numbers)))
             (abs (- goal
                     (find-closest-number goal (cdr numbers)))))
         (car numbers)
         (find-closest-number goal (cdr numbers)))))
```

```
> (find-closest-number 150
    (list 101 110 120 157 340 588))
157
> (find-closest-number 0 (list 1))
1
> (find-closest-number 0 (list ))
first: expects argument of type <non-empty list>; given ()
```

Generalizing find-closest-number

- How would we implement find-closest-number-without-going-over?
- What about find-closest-word?
- ...

The "closeness" metric should be a procedure parameter

Charge

- Read GEB “Little Harmonic Labyrinth” and Chapter 5 before Monday’s class
- PS3 accepted until beginning of Monday’s class