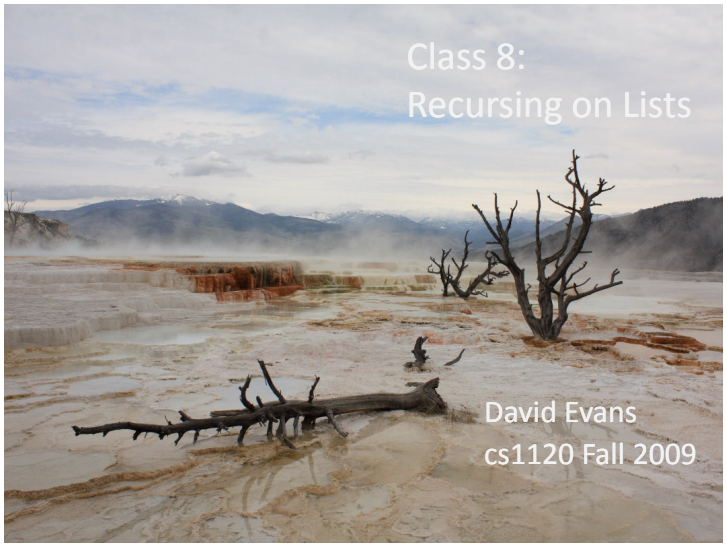


Menu

- Quiz Comments
- List Procedures
  - is-list? (Exercise 5.11)
  - list-sum ⇒ list-product, deep-list-sum
- Generalizing list procedures
- find-closest
- Monday: GEB Chapter 5

Quiz Comments

- About 2/3 of you have read the GEB reading
  - I really hope everyone reads this!
  - We'll talk about it in Monday's class
- Everyone should know the definition of a List (but only about ½ did)
- A List is either:
  - **null**
  - or, a **Pair** whose second part is a **List**

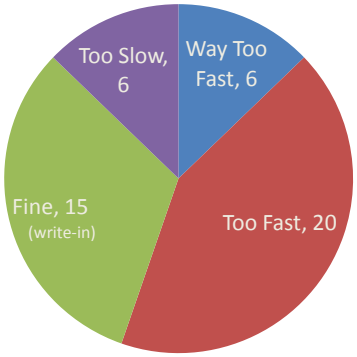
This definition is very important: all of our List procedures depend on it!

Common, Incorrect List Definition

A List is a Pair whose second part is either **null** or a List.

If this is our List definition, there is no way to make a list with no elements.

Class Pace



Book Comments

- Many people want answers to the exercises (13 people mentioned this)
  - We will do some in class
  - If you ask, I'm happy to provide hints/answers or comments on your answers
- More diagrams, examples
- Editing problems: please do send me anything specific you notice!

is-list?

Define a procedure **is-list?** that takes one input and outputs **true** if the input is a List, and **false** otherwise.

(is-list? (list 1 2 3)) → **true**
(is-list? (cons 1 (cons 2 null)))
→ **true**
(is-list? null) → **true**
(is-list? 3) → **false**
(is-list? (cons 1 (cons 2 3)))
→ **false**

is-list?: Easy Way

This is exactly what the built-in **list?** procedure does.

(define is-list? list?)

is-list? without using list?

null?: Value → Boolean
outputs true if input is **null**
pair?: Value → Boolean
outputs true if input is a Pair

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

```
(define (is-list? p)
  (or (null? p)
      (and (pair? p) (is-list? (cdr p)))))
```


list-sum

Define a procedure, **list-sum**, that takes a list of numbers as input and outputs the sum of the numbers in the input list.

(list-sum (list 1 2 3)) → **6**
(list-sum null) → **0**

list-sum

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

Okay, what about **list-product**?

deep-list-sum

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

Okay, what about **list-product**?

Tracing deep-list-sum

```
(require (lib "trace.ss"))
(trace deep-list-sum)
```

Use trace to see entrances and exits of procedure applications.

```
|(deep-list-sum ((1 2) 3))
| (deep-list-sum (1 2))
| |(deep-list-sum (2))
| | (deep-list-sum ())
| | 0
| |2
| 3
| (deep-list-sum (3))
| |(deep-list-sum ())
| |0
| 3
|6
6
```


list-product

```
(define (list-product p)
  (if (null? p)
    1
    (* (car p) (list-product (cdr p)))))
```

Okay, what about **list-length**?

list-length

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


Comparing List Procedures

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

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

```
(define (list-product p)
  (if (null? p)
    1
    (* (car p) (list-product (cdr p)))))
```

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


Base Cases

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

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

```
(define (list-product p)
  (if (null? p)
    1
    (* (car p) (list-product (cdr p)))))
```

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


Recursive Calls

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

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

```
(define (list-product p)
  (if (null? p)
    1
    (* (car p) (list-product (cdr p)))))
```

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

What does each do with the **car** of the list?

find-closest-number

Define a procedure **find-closest-number** that takes two inputs, a goal number, 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?

Finding the Closest

Strategy:

If the first number is closer than the closest number of the rest of the numbers, use the first number.

Otherwise, use the closet number of the rest of the numbers.

Optimistic Function

```
(define (find-closest goal numbers)
  (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, something you can already solve.

If there is only one number, that is the best match.

The Base Case

```
(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)))))
```

Same as before

Testing

```
(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 ()
```

Charge

- Problem Set 2: Due Monday
  - Help hours Sunday 6-8:30 in Olsson 001
- Monday: Even more Recursiveness!
  - GEB Chapter 5