CS3:
Introduction to Symbolic Programming

Lecture 12:
Lists

Spring 2008

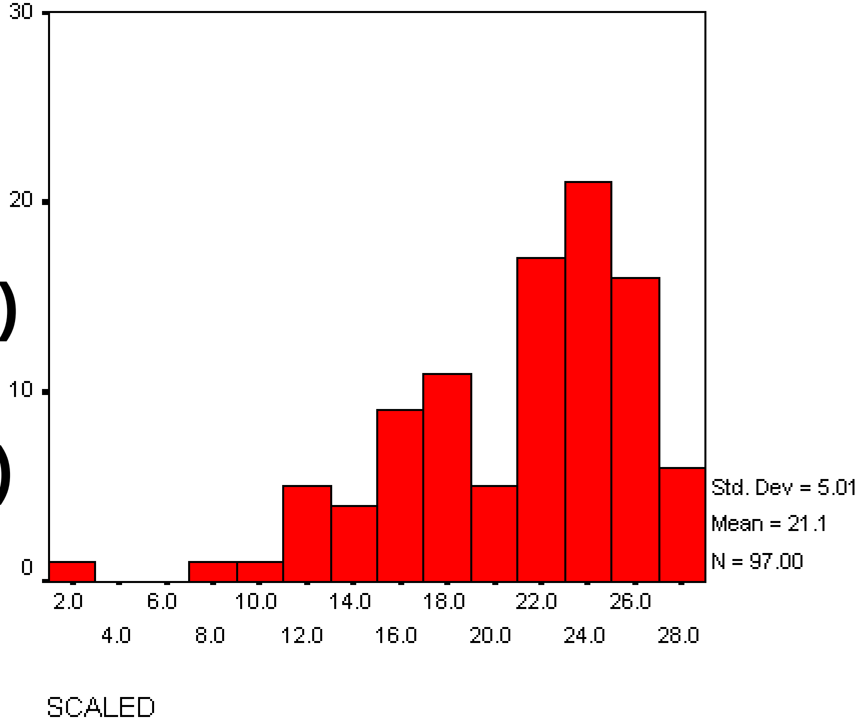
Nate Titterton
nate@berkeley.edu

Schedule

12	Apr 14-18	Lecture: Lists, lists, lists Lab: Generalized lists and trees (Tue/Wed) Sequential Programming (Thur/Fri) Reading: Simply Scheme chap. 20 (Thur/Fri) SS ch. 18 is not required, but maybe useful
13	April 21-25	Introduction to the big project Lab: Big project – introduction, and choose partners (checkoff #1)
14	April 28 – May 2	Lecture: Advanced lists, Scheme vs. other lang Lab: Big project (checkoff #2)
15	May 5 – 9	Lecture (guest):CS at Berkeley and outside… Lab: Big project (checkoff #3 and due at end)
16	May 12	Final Exam Review Lab: *no more labs!*

Any questions about midterm #2?

1. Bowling questions (small HOF questions)
2. `occurs-in-order-in?` (debugging recursion and tree recursion)
3. Write `every` using `accumulate` (debugging HOF)
4. `early-words` (two-stage recursion and HOF)
5. The game of `darts` (accumulating recursion)



Lists

Sentences(words) vs lists: constructors

cons Takes an element and a list Returns a list with the element at the front, and the list contents trailing	
append Takes two lists Returns a list with the elements of each list put together	
list Takes any number of elements Returns the list with those elements	**sentence** Takes a bunch of words and sentences and puts "them" in order in a new sentence.

Sentences(words) vs lists: selectors

car Returns the first element of the list	**First** Returns the first word (although, works on non-words)
cdr Returns a list of everything but the first element of the list	**butfirst** Returns a sentence of everything but the first word (but, works on lists)
	last
	butlast
list-ref Gets a particular item in the list, with a 0-based index (note, reversed arguments)	**item** Gets a particular item in the list, with a 1-based index

What is the point of `cons`? (2/2)

```
(define (square-all seq)
  (if (null? seq)
      '()
      (cons (square (car seq))
            (square-all (cdr seq)))))

(s-a '(1 2 3)) → (cons 1 (cons 4 (cons 9 '())))
```

Sentence (and word) do more, though

- Consider

```
(define (reverse lst)
  (if (null? lst)
      '()
      (cons (reverse (cdr lst))
            (car lst))
      ))
```

- What will the following return?
- What is the right construction?

Sentences(words) vs lists: HOF

map Returns a list where a func is applied to every element of the input list. Can take multiple input lists.	**every** Returns a sentence where a func is applied to every element of an input sentence or word.
filter Returns a list where every element satisfies a predicate. Takes a single list as input	**keep** Returns a sentence or word where every element satisfies a predicate
reduce Returns the value of applying a function to successive pairs of the (single) input list	**accumulate** Returns the value of applying a function to successive pairs of the input sentence or word
apply Takes a function and list of arguments, and calls that function with those arguments	…

Fashion matching…

- **Write a function `pair-up` that takes a list of tops and a list of bottoms, and returns matches:**

```
 (pair-up '(t-shirt sweatshirt tank-top)
          '(jeans skirt capris))
   →
((t-shirt jeans) (sweat-shirt skirt)
 (tank-top capris))
```

- **And, can you write `pair-all`, which returns all pairs of matches?**

A few other important topics re: lists

1. `map` can take multiple arguments

2. `apply`

3. Association lists

4. Generalized lists
   - And data structures they can represent

`map` can take multiple list arguments

```
(map + '(1 2 3) '(100 200 300))
➔(101 202 303)
```

The argument lists have to be the same length

```
(define (palindrome? lst)
  (all-true?
    (map equal? lst (reverse lst))))
```

```
(palindrome?
       '(a m a n a p l a n a c a n a l p a n a m a))
   ➔ #t
```

`apply` (not the same as `accumulate`!)

- apply takes a function and a list, and calls the function with the elements of the list as its arguments:

```
(apply + '(1 2 3))

(apply cons '(joe (bob)) )

(apply day-span
       '((january 1) (december 31)))
```

Association lists

- Used to associate *key-value* pairs

```
((i 1) (v 5) (x 10) (l 50) (c 100) (d 500) (m 1000))
```

- `assoc` looks up a key and returns a pair

```
(assoc 'c '((i 1) (v 5) (x 10) … ) )
➔ (c 100)
```

```
;; Write sale-price, which takes a list of items
;; and returns a total price
(define *price-list*
        '((bread 2.89) (milk 2.33)
          (cheese 5.21) (chocolate .50)
          (beer 6.99) (tofu 1.67) (pasta .69)))

(sale-price '(bread tofu))
```

Generalize lists and trees

Generalized lists

- Elements of a list can be anything
- A list that contains one or more lists (which contain…) we call a generalized list.
- e.g.,

```
()

(this (((list) contains) ((only three))
 things) (really) )
```

car-cdr recursion

- Tree recursion for generalized lists

- Write deep-add which returns the sum of all numbers in the list:

  - `(deep-add '(1 (2 3) (((4)) 5) 6))`
    ➔ `21`

Write deep-member?

```
(deep-member?  'b
   '((a b) (c d) (e f) (g h i)) )
➔ #t

(deep-member?  'x
   '((a b) (c d) (e f) (g h i)) )
➔ #f

(deep-member?  '(c d)
   '((a b) (c d) (e f) (g h i)) )
➔ #t
```

Trees…

- A tree is a special kind of generalized list, where each level has a name and a list of children (trees):

```
(define (name node) (car node))
(define (children node) (cdr node))
(define (leaf? tree)
   (null? (children tree)))
```