CS3:
Introduction to Symbolic Programming

Lecture 14:
Lists

Fall 2007

Nate Titterton
nate@berkeley.edu

Schedule

13	**Nov 19–23**	**Lecture: Introduction to the Big Project** **Advanced Lists** **Lab: Work on the Big Project: checkoff #1**
14	**Nov 26–30**	**Lecture: Advanced Lists** **Scheme versus other languages** **Lab: Big Project: checkoff #2**
15	**Dec 3–7**	**Lecture (guest):CS at Berkeley and outside..** **Lab: Big Project: checkoff #3 and due**
16	**Dec 10**	**Lecture: Exam Review** **Labs: *No thank you***
	Dec 18 (Tuesday)	**Final Exam 8-11am (?)**

Any questions about the project?

Tues/Wed	Thur/Fri
(Nov 20/21) Introduction Checkoff #1	*(Nov 22/23)* ~~*Thanksgiving*~~
(Nov 27/28) Checkoff #2	*(Nov 29/30)*
(Dec 4/5) Checkoff #3	*(Dec 7th, Friday)* Due (at midnight)

Partnerships

- **If you want/need a partner for the big project, please come see me after lecture, or email.**

Lists

Lists: review of new procedures

- Constructors
  - `append`
  - `list`
  - `cons`
- Selectors
  - `car`
  - `cdr`
- HOF
  - `map`
  - `filter`
  - `reduce`
  - `apply`

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 element 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 …

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

```
(define (square-all seq)
  (if (empty? seq)
      '()
      ( se   (square (first seq))
             (square-all ( bf seq)))))

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

```
(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

```
(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 arguments, and applies that function to its 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

- `map` can take multiple arguments
- `apply`
- Association lists
- Generalized lists
  - And data structures they can represent
    - Like Trees

`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))
```

Generalized lists

- Elements of a list can be anything, including any list

- Lab materials discuss
  - `flatten` (3 ways)
  - `completely-reverse`
  - processing a tree-structured directory

How about this `flatten`?

```
(define (flatten thing)
   (if (list? thing)
      (reduce ________ (map flatten thing))
      (_______ thing)))
```

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)))
```

Schedule

13	Nov 19–23	**Lecture: Introduction to the Big Project** **Advanced Lists** **Lab: Work on the Big Project: checkoff #1**
14	Nov 26–30	**Lecture: Advanced Lists** **Scheme versus other languages** **Lab: Big Project: checkoff #2**
15	Dec 3–7	**Lecture (guest):CS at Berkeley and outside..** **Lab: Big Project: checkoff #3 and due**
16	Dec 10	**Lecture: Exam Review** **Labs: *No thank you***
	Dec 18 (Tuesday)	**Final Exam 8-11am (?)**

Any questions about the project?

Tues/Wed	Thur/Fri
(Nov 20/21) **Introduction** **Checkoff #1**	*(Nov 22/23)* ***Thanksgiving***
(Nov 27/28) **Checkoff #2**	*(Nov 29/30)*
(Dec 4/5) **Checkoff #3**	*(Dec 7th, Friday)* **Due (at midnight)**

Partnerships

- If you want/need a partner for the big project, please come see me after lecture, or email.

Lists

Lists: review of new procedures

- Constructors
  - `append`
  - `list`
  - `cons`
- Selectors
  - `car`
  - `cdr`
- HOF
  - `map`
  - `filter`
  - `reduce`
  - `apply`

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 element 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.

Some "tips":

With append, you erase the middle parentheses

```
  (append  '( a   b    c    ) (   d  (e)  f )      )
  ;;                              |   |
  ;;                              X X
         ->     ( a   b    c          d   (e)  f )
```

With list, you add parentheses around the arguments

```
   (list      '( a b c )   (d  (e)  f)   (g h i)        )
     ;;      |                                          |
     ;;      V                                         V
      ->   ( ( a b c )   (d  (e)  f)   (g h i)  )
```

With cons, the last argument is a list (almost always in the real world, and always in this class). cons stretches the opening paren for that second argument to include the first argument:

```
      (cons    'a    '( b )      )
;;                        |
;;              ---------
;;              V
    ->         (  a       b )
```

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 …

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

```
(define (square-all seq)
  (if (empty? seq)
      '()
      ( se   (square (first seq))
             (square-all ( bf seq)))))

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

```
(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

```
(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 arguments, and applies that function to its 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

- `map` can take multiple arguments
- `apply`
- Association lists
- Generalized lists
  - And data structures they can represent
    - Like Trees

`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
```

```
(define (all-true? lst)
  (or (null? lst)
      (and (car lst)
           (all-true? (cdr lst)))))
```

`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))
```

```
(define *price-list* '((bread 2.89) (milk 2.33) (cheese 5.21) (chocolate .50)
                       (beer 6.99) (tofu 1.67) (pasta .69)))


(define (sale-price items))
  (* 1.0825        ;; tax, why not…
    (apply +
      (map (lambda (i) (cadr (assoc i *price-list*)))
           items))))


#|
(sale-price '(cheese milk pasta tofu) *price-list*)  ;; 10.71675
(sale-price '(beer beer beer beer) *price-list*)  ;; 30.2667

|#
```

Generalized lists

- Elements of a list can be anything, including any list

- Lab materials discuss
  - `flatten` (3 ways)
  - `completely-reverse`
  - processing a tree-structured directory

How about this `flatten`?

```
(define (flatten thing)
   (if (list? thing)
      (reduce _______ (map flatten thing))
      (______ thing)))
```

;; The way to think about this is to "trust
;; the recursion". "flatten" has to return a flat list, right? So, both
;; cases in the if have to return properly flattened lists.

;; what is (map flatten thing) going to return?
;; well, it has to be something like this:
;; ((a b c) (d e f) (g h i))
;; or, a "list of flat lists". The full reduce has to return, when given
;; this,
;; (a b c d e f g h i)
;; or a properly flat list. With that, you should be able to fill
;; in the first blank.

;; The second blank is also easy, when you realize that the return value
;; must be a flat list. "thing" is a word (or, more properly, not a list).
;; So, turning it into a flat list is easy!

```
;; Here is the solution
(define (flatten thing)
  (if (list? thing)
    (reduce append (map flatten thing))
    (list thing)))
```

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
```

```
;; similar to solution for flatten
(define (deep-member? item gl)
  (cond ((null? gl)  #f)
       ((list? (car gl))
         (or (equal? item (car gl))
            (deep-member? item (car gl))
            (deep-member? item (cdr gl))
        ) )
       (else   ;; first element is a non-list
        (or (equal? item (car gl))
            (deep-member? item (cdr gl)))
        )))
```

```
;; another way
(define (deep-member? item gl)
  (cond ((null? gl)  #f)
       ((equal? item (car gl)) #t)    ; checks with either a list or non-list as
first element
       ((list? (car gl))
         (or (deep-member? item (car gl))
            (deep-member? item (cdr gl))
        ) )
       (else (deep-member? item (cdr gl)))
        ))
```

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)))
```