CS3:
Introduction to Symbolic Programming

Lecture 14:
Lists

Fall 2006

Nate Titterton
nate@berkeley.edu

Schedule

13	April 16-20	Lecture: CS3 Projects, Lists Lab: Begin work on CS3 Big Project Reading: Simply Scheme, chapter 20
14	April 23-27	Lecture: Non-functional programming, lists, project review Lab: Non-functional programming Work on projects
15	Apr 30-May 4	Lecture: CS at Berkeley (guest lecture) Lab: Finish projects (due end of week)
16	May 7	Lecture: Exam review *no more labs!*
	Thursday, May 17	*Final Exam, 5-8pm* *F295 Haas*

Midterm #2

Any questions?

4) tail-recursive `roman-sum`
5) price-is-right
6) `last-letter`
7) chips, drinks, and gum -- `snack3`

Project Check-offs

- **There are** 3 **checkoffs**

  You need **to do them on time in order to get credit for the project**

3. **Tell your TA which project you will do and who you will do it with**
4. **Show your TA that you have accomplished something. S/he will comment.**
5. **Show that you have most of the work done: your TA will run your code.**

Lists

Lists: review of new procedures

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

What goes in a list?

- Answer: anything!

- So,

```
        (word? x)

     (not (list? x))
```

are not the same thing!

A few other important topics re: lists

2. `map` can take multiple arguments

4. `apply`

6. Association lists

8. Generalized lists

`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
```

- Write `all-true?`, without using `cond`/`if`.

`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
```

CS3:
Introduction to Symbolic Programming

Lecture 14:
Lists

Fall 2006

Nate Titterton
nate@berkeley.edu

Schedule

13	April 16-20	Lecture: CS3 Projects, Lists Lab: Begin work on CS3 Big Project Reading: Simply Scheme, chapter 20
14	April 23-27	Lecture: Non-functional programming, lists, project review Lab: Non-functional programming Work on projects
15	Apr 30-May 4	Lecture: CS at Berkeley (guest lecture) Lab: Finish projects (due end of week)
16	May 7	Lecture: Exam review *no more labs!*
	Thursday, May 17	*Final Exam, 5-8pm* *F295 Haas*

Midterm #2

Any questions?

4) tail-recursive `roman-sum`

5) price-is-right

6) `last-letter`

7) chips, drinks, and gum -- `snack3`

Project Check-offs

- **There are** 3 **checkoffs**

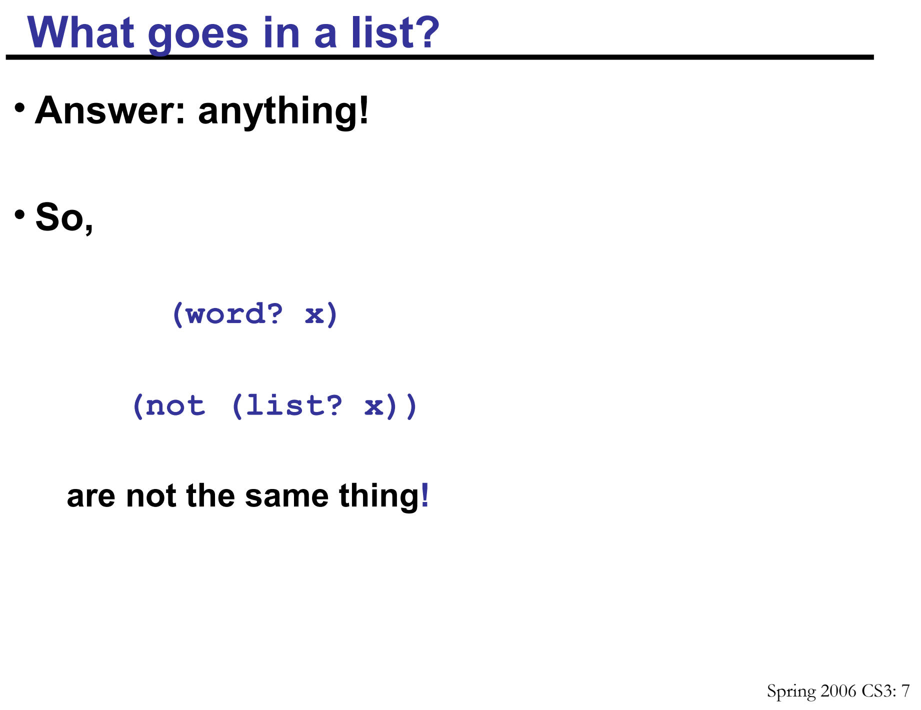
  You need **to do them on time in order to get credit for the project**

3. **Tell your TA which project you will do and who you will do it with**
4. **Show your TA that you have accomplished something. S/he will comment.**
5. **Show that you have most of the work done: your TA will run your code.**

Lists

Lists: review of new procedures

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



What goes in a list?

- Answer: anything!

- So,

```
(word? x)

(not (list? x))
```

are not the same thing!

See the slide on flatten, and compare the code on the slide to the code on ucwise: in the slide, we use the proper "(not (list? thing))" rather than "(word? thing)", which won't be fooled by booleans and procedures (i.e., things that aren't words but aren't lists either).

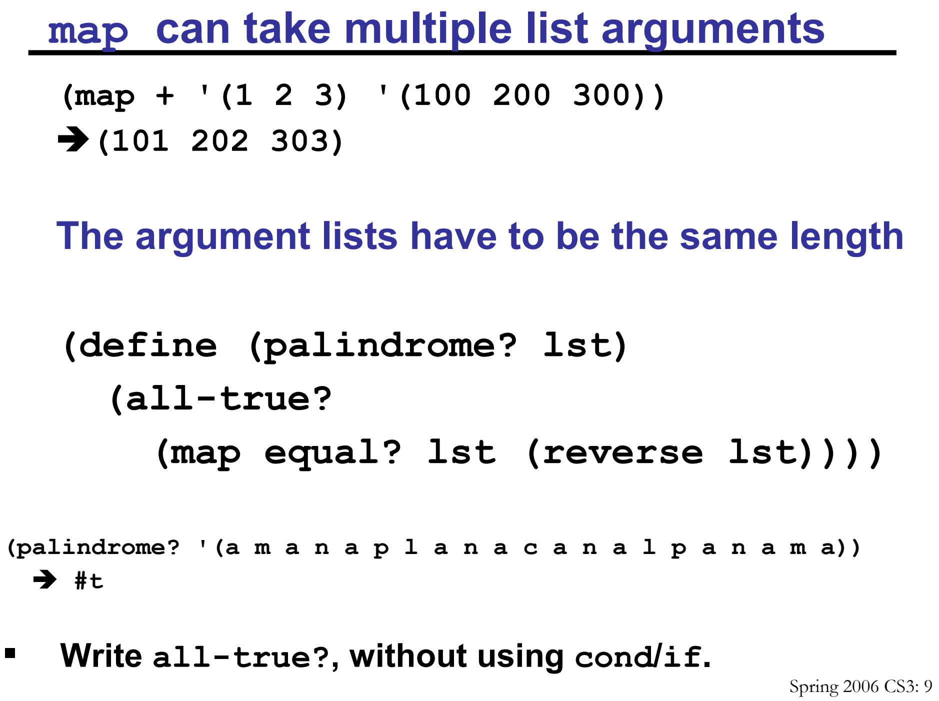
A few other important topics re: lists

2. `map` can take multiple arguments

4. `apply`

6. Association lists

8. Generalized lists



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
```

- Write `all-true?`, without using `cond`/`if`.

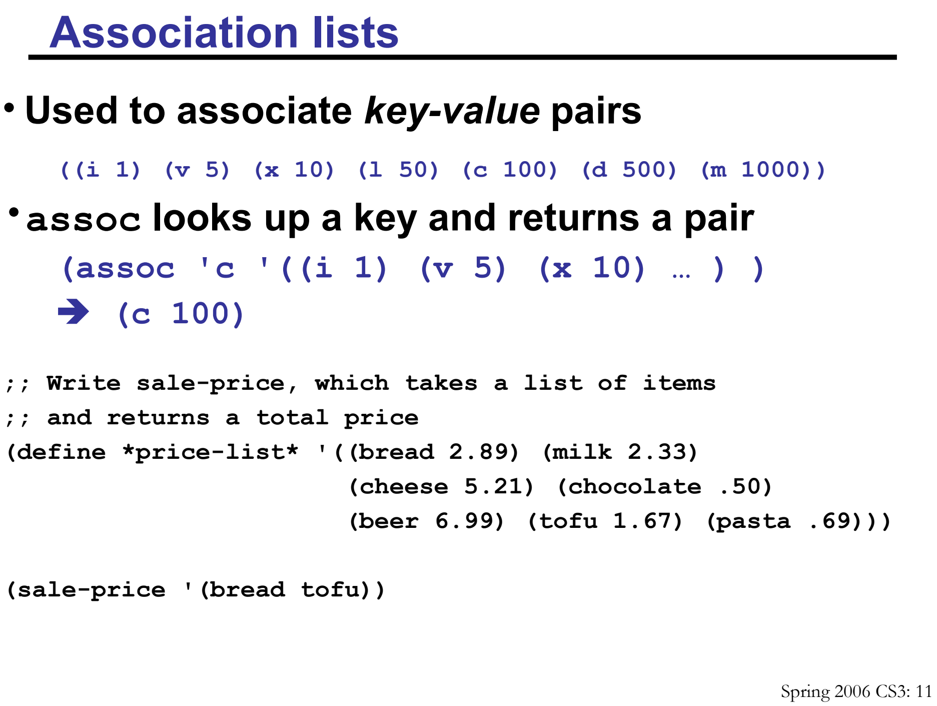
Spring 2006 CS3: 9

```
(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))
```

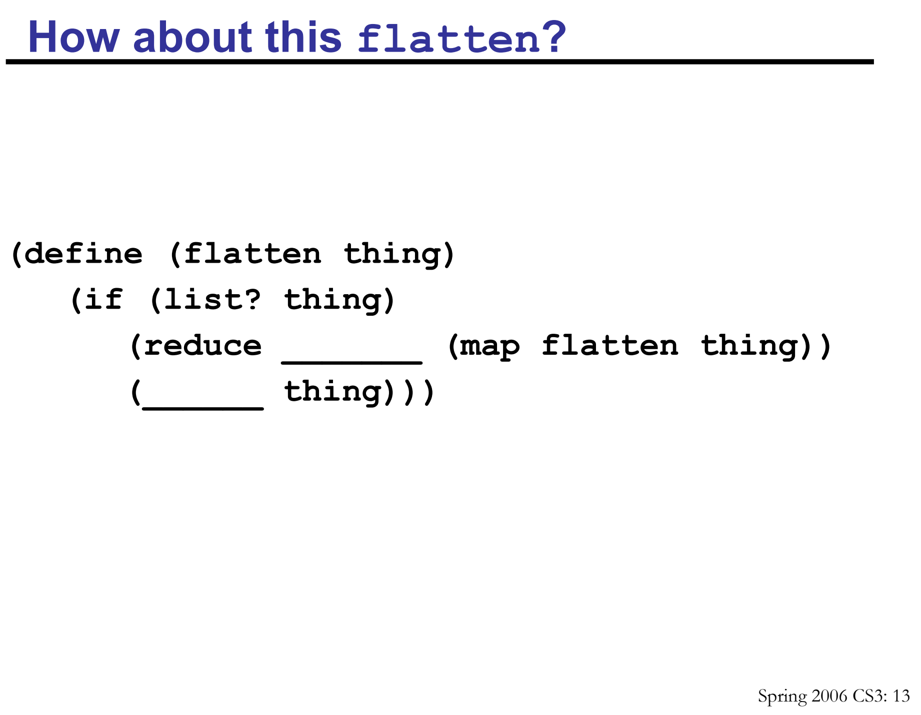
Spring 2006 CS3: 11

```
(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



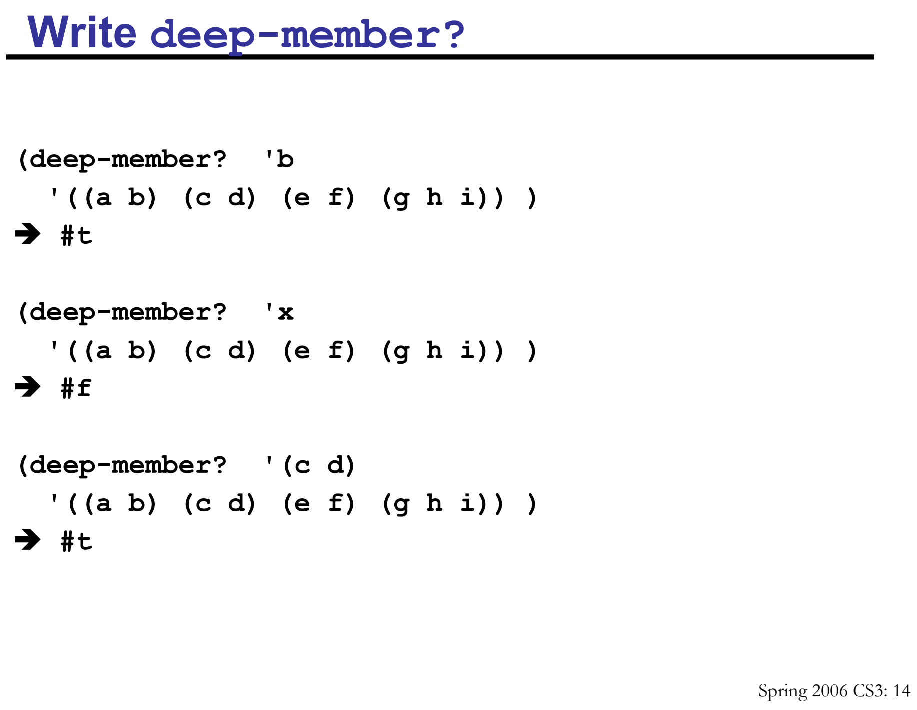
How about this flatten?

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

Spring 2006 CS3: 13

```
;; 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
```

Spring 2006 CS3: 14

```
;; 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)))
              ))
```