CS3:
Introduction to Symbolic Programming

Lecture 14:

Lists

Scheme vs. other programming languages

Fall 2006

Nate Titterton

nate@berkeley.edu

Schedule

13	Nov 20-24	Lecture: Lists, and introduce the big project Lab: Lists; start on the project
14	Nov 27–Dec 1	Lecture: Lists, other languages Lab: Big Project CHECKOFF #1 – Tue/Wed CHECKOFF #2 – Thur/Fri
15	Dec 4 – Dec 8	Lecture: Guest Lecture: what is CS at UCB? Lab: Finish up the Project CHECKOFF #3 – Tue/Wed Project Due on Fri (at midnight)
	Dec 16th	Final Exam (Saturday, 9am-11am) 100 Lewis (Lewis?)

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

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.

`cons` is closely tied to recursion

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

(ssa '(1 2 3)) → (se 1 (se 4 (se 9 '())))



(define (list-square-all lst)
  (if (null? lst)
      '()
      (cons (square (car lst))
            (list-square-all (cdr lst)))))

(lsa '(1 2 3)) → (cons 1 (cons 4 (cons 9 '())))
```

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 …

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

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
```

`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 a
;; table of item-price pairs, 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) *price-list*)
```

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)))
```

Scheme versus other languages

Functional Programming

- In CS3, we have focused on programming without *side-effects*.
  - All that can matter with a procedure is what it returns
  - In other languages, you typically:
    - Perform several actions in a sequence
    - Set the value of a variable – and it stays that way
  - All of this is possible in Scheme; Chapter 20 is a good place to start

The language Scheme

- **Scheme allows you to ignore tedium and focus on core concepts**
  - The core concepts are what we are teaching!
- **Other languages:**
  - Generally imperative, sequential
  - Lots and lots of syntactic structure (built in commands)
  - Object-oriented is very "popular" now

CS3 concepts out in the world

- Scheme/lisp does show up: scripting languages inside applications (emacs, autocad, Flash, etc.)
- Scheme/Lisp is used as a "prototyping" language
  - to quickly create working solutions for brainstorming, testing, to fine tune in other languages, etc.
- Recursion isn't used directly (often), but recursive ideas show up everywhere

Java

- **Java is a very popular programming language**
  - **Designed for LARGE programs**
  - **Very nice tools for development**
  - **Gobs of libraries (previous solutions) to help solve problems that you might want solved**

PHP

- **PHP**
  - Popular language for web development (combined with a web-server and database)
  - Lots of features, but little overall "sense"
  - Because programs in PHP execute behind a web-server and create, on the fly, programs in other languages, debugging can be onerous.

SQL resembles HOFs

- **SQL if for database retrieval**

- **query: "Tell me the names of all the lecturers who have been at UCB longer than I have."**

```
select name from lecturers
 where date_of_hire <
   (select date_of_hire from lecturers where name =
    'titterton');
```

- **query: "Tell me the names of all the faculty who are older than the faculty member who has been here the longest."**

```
select L1.name from lecturers as L1 where
L1.age >
   (select L2.age from lecturers as L2
    where L2.date_of_hire =
       (select min(date_of_hire) from lecturers) );
```

Problems

CS3:
Introduction to Symbolic Programming

Lecture 14:
Lists
Scheme vs. other programming languages

Fall 2006

Nate Titterton
nate@berkeley.edu

Schedule

13	Nov 20-24	Lecture: Lists, and introduce the big project Lab: Lists; start on the project
14	Nov 27–Dec 1	Lecture: Lists, other languages Lab: Big Project CHECKOFF #1 – Tue/Wed CHECKOFF #2 – Thur/Fri
15	Dec 4 – Dec 8	Lecture: Guest Lecture: what is CS at UCB? Lab: Finish up the Project CHECKOFF #3 – Tue/Wed Project Due on Fri (at midnight)
	Dec 16th	Final Exam (Saturday, 9am-11am) 100 Lewis (Lewis?)

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

Click to add text

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.

`cons` is closely tied to recursion

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

(ssa '(1 2 3)) → (se 1 (se 4 (se 9 '())))



(define (list-square-all lst)
  (if (null? lst)
      '()
      (cons (square (car lst))
            (list-square-all (cdr lst)))))

(lsa '(1 2 3)) → (cons 1 (cons 4 (cons 9 '())))
```

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 …

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

What goes in a list?

- **Answer: anything!**

- **So,**

```
(word? x)

(not (list? x))
```

are not the same thing!

Spring 2006 CS3: 9

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
```



```
(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 a
;; table of item-price pairs, 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) *price-list*)
```



```
(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 price-list)
  (* 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)))
```

Scheme versus other languages

Click to add text

Functional Programming

- In CS3, we have focused on programming without *side-effects*.
  - All that can matter with a procedure is what it returns
  - In other languages, you typically:
    - Perform several actions in a sequence
    - Set the value of a variable – and it stays that way
  - All of this is possible in Scheme; Chapter 20 is a good place to start

The language Scheme

- Scheme allows you to ignore tedium and focus on core concepts
  - The core concepts are what we are teaching!
- Other languages:
  - Generally imperative, sequential
  - Lots and lots of syntactic structure (built in commands)
  - Object-oriented is very "popular" now

CS3 concepts out in the world

- Scheme/lisp does show up: scripting languages inside applications (emacs, autocad, Flash, etc.)
- Scheme/Lisp is used as a "prototyping" language
  - to quickly create working solutions for brainstorming, testing, to fine tune in other languages, etc.
- Recursion isn't used directly (often), but recursive ideas show up everywhere

Java

- Java is a very popular programming language
  - Designed for LARGE programs
  - Very nice tools for development
  - Gobs of libraries (previous solutions) to help solve problems that you might want solved

PHP

- PHP
  - Popular language for web development (combined with a web-server and database)
  - Lots of features, but little overall "sense"
  - Because programs in PHP execute behind a web-server and create, on the fly, programs in other languages, debugging can be onerous.

SQL resembles HOFs

- **SQL if for database retrieval**

- **query: "Tell me the names of all the lecturers who have been at UCB longer than I have."**

```
select name from lecturers
 where date_of_hire <
   (select date_of_hire from lecturers where name =
    'titterton');
```

- **query: "Tell me the names of all the faculty who are older than the faculty member who has been here the longest."**

```
select L1.name from lecturers as L1 where
L1.age >
   (select L2.age from lecturers as L2
    where L2.date_of_hire =
       (select min(date_of_hire) from lecturers) );
```

Problems

Click to add text

A set of 4 problems was handed out. See the ucwise announcements.