# Lecture 8: Pairs and lists

CS 61A
Summer 2006
Instructor: Kevin Lin



## Administrative stuff

- HW 3A posted, due next Wednesday
- Project 2 posted, due on 7/20
- Solutions posted (on campus only): http://inst.eecs.berkeley.edu/~cs61a/su06/solutions
- Midterm exam #1 …
- THIS FRIDAY, 7/14, 4-6 PM
- Bring yourself, something to write with, and one standard-size page of notes (front and back)
- No additional notes, books, electronic devices, etc. allowed.



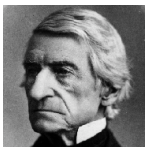
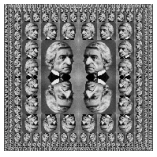
## Administrative stuff

- Ramesh and Gap will run a review session on Thursday, IN LECTURE. You'll do practice problems together.
- Anonymous feedback form: http://inst.eecs.berkeley.edu/~inst/anon
  Please use it to send anonymous feedback/comments/suggestions/insults for me and the TAs!
- The pace of the course is getting faster and faster and faster now … Start on assignments **early** so you don't fall behind. I mean it!!!



## Project 2

- You must do the reading in section 2.1 and 2.2 before you can do the project.



## Project 2

- **It is almost impossible to test most of your procedures in this project until you are completely finished with it.**
- **So the proper use of data abstraction is extremely important in this project. Data abstraction violations will be penalized.**
- **This project gives you yet more programming practice with functional programming, and it really emphasizes data abstraction, although it may not seem like it at first.**



## HOFs you can use on the exam

**We've seen every and keep countless times … So here are some more.**

```
> (keep (lambda (x) (or (even? x) (equal? x 1)))
        '(1 2 3 4))
(1 2 4)
> (keep (lambda (x) (member? x '(x y z)))
        'syzygy)
sg
> (every (lambda (x) (word x 's))
         '(tricky hobbits))
(trickys hobbitss)
> (every (lambda (x) (+ x 1)) '1234)
(2 3 4 5)
```

## HOFs you can use on the exam

There are two versions of accumulate you can use. There's accumulate from homework, exercise 1.32:

```
(accumulate combiner null-value term a next b)
```

This can be used as follows:

```
> (accumulate * 1 (lambda (x) x)
              1 (lambda (x) (+ x 1)) 5)
120
> (accumulate se '(foo) (lambda (x) x)
              1 (lambda (x) (+ x 1)) 5)
(1 2 3 4 5 foo)
```



## HOFs you can use on the exam

Here is another version:

```
(define (accumulate op null-val sent)
  (if (empty? sent)
      null-val
      (op (first sent)
          (accumulate op null-val (bf sent)))))
```

You can use this version of accumulate as follows:

```
> (accumulate + 0 '(2 5 3 4 1))
15
> (accumulate word 'd '(a b c))
abcd
```



## HOFs you can use on the exam

On the exam, when we say "use HOFs", we mean: "use every, keep, and/or accumulate".

It is unlikely that you will need to use accumulate on the exam, but you are allowed use it if you find it useful.

If you do use it, please specify which version you're using.



## More midterm stuff

The midterm will emphasize weeks 1 and 2. There will probably be one question based on the material in today's lecture and/or tomorrow's lecture.

The exam should take about 1 hour; but you have 2 hours to do it. It is worth 40 points, which is 13.33333% of your course grade.

Major topics: Scheme basics, programming skills, higher order functions (using them and also writing them), order of growth analysis, data abstraction.

The exam will not cover: invariants, induction, commenting, debugging.



## CONS pairs

We have seen sentences, which allow us to create groups of words. But how can we create groups of objects other than words (e.g. procedures)? And how can we represent sets of data that are not "flat"?

Answer: The only data structure you'll ever need in CS61A: pairs!

Pairs are an abstract data type, with constructor CONS and selectors CAR and CDR.

What is meant by this, exactly?



## CONS pairs

Note: cons is not a special form, so we always evaluate all of its arguments.

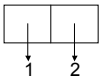
(CONS expA expB) creates a pair whose first element (the "CAR") is the value of expA, and whose second element (the "CDR") is the value of expB. The way that Scheme prints this pair is (A . B), where A is what's printed by expA and B is what's printed by expB.

```
> (cons 1 2)
(1 . 2)
> (cons (cons 1 2) 3)
((1 . 2) . 3)
> (cdr (car (cons (cons 1 2) 3)))
2
```

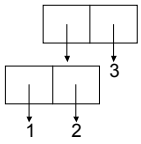
## CONS pairs

**Box-and-pointer diagrams:**

```
> (cons 1 2)
(1 . 2)
```



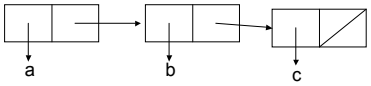

```
> (cons (cons 1 2) 3)
((1 . 2) . 3)
```






## CONS pairs

**Cons pairs can be used to create linear lists of objects.**

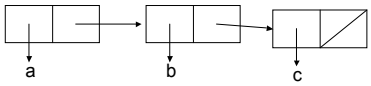



**But what's that funky thing on the last pair? It's the NULL ELEMENT (a.k.a. the EMPTY LIST), '(). We've seen this before as the EMPTY SENTENCE. However, you should not confuse the two; that would be a data abstraction violation. We use the term "empty sentence" when we're referring to '() in the context of sentences and "null element" or "empty list" when we're referring to '() in the context of pairs. It is merely a coincidence that we implemented them in the same way.**



## CONS pairs

**Cons pairs can be used to create linear lists of objects.**



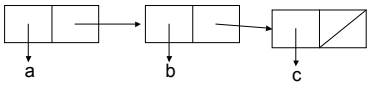

```
> (cons 'a (cons 'b (cons 'c '())))
(a b c)
```

**Shouldn't it print out (a . (b . (c . ()))) ?!?!?!**
**What's going on?**



## Lists

**This construction is so useful that we give it a name: list.**




```
> (cons 'a (cons 'b (cons 'c '())))
(a b c)
```

**Scheme does NOT print (a . (b . (c . ()))) because it would be annoying if we had to read such a huge expression every time we used this common construction.**

**And hey, doesn't that look like a sentence …?**



## Lists

**The Parenthesis Soap Opera:**

**How does Scheme "pretty-print" lists?**

**Actually … once upon a time there were some pairs.**

**(cons 'a (cons 'b 'c)) ---> (a . (b . c))**

**Each of the pairs of parentheses were a lovely couple. But the red period and the blue parenthesis were having an affair, and ran off with each other.**

**(a b . c))**

**Betrayed, the green parenthesis dies of loneliness.**

**(a b . c)**

**Ta-da!**



## Lists

**So lists are reduced to their pretty-printed form, as for instance below:**

```
(a . (b . (c . ()))) ->
(a . (b . (c))) ->
(a . (b c)) ->
(a b c)
```

## Lists and sentences

Hey, doesn't that look like a sentence??

```
(a b c)
```

Actually, the way that we have implemented the sentence abstract data type is using lists. That is to say, sentences are really flat LISTs of words. And now we finally have a rigorous definition for the word "list".

***However, you CANNOT use first, butfirst, etc. on lists and you CANNOT use car, cdr, etc. on sentences.***

Why not?
Because it would be a data abstraction violation. If you do this on an exam you will be penalized.



## Pairs and lists

Scheme provides the procedure (list e1 e2 …), which does the same thing as (cons e1 (cons e2 …)), creating a list of elements.

```
> (list 'a 'b 'c)
(a b c)
```



## Pairs and lists

Lists should not be viewed in the same way you view sentences. They provide much more functionality.

```
> (list 'a 'b 'c)
(a b c)
> (list (list 'a 'b 'c) (list 'd 'e 'f))
((a b c) (d e f))
```

If the first list above is X, and the second list is Y, then (using car and cdr) how can we extract the letter c from X? How can we extract the letter d from Y?

Lists can also be quoted, just like sentences:
e.g. '((a b c) (d e f))



## Pairs and lists

Scheme provides shortcuts so you don't have to type tedious expressions like (car (cdr (cdr x)).

```
(cdar x) = (cdar
(caddr x) = (car (cdr (cdr x)))
(caddar x) = (car (cdr (cdr (car x))))
Etc ... All combinations up to 4 letters.
```

They're fun to pronounce!



## Pairs and lists

Exercise: What does Scheme print in response to the following expressions? Also, draw the box-and-pointer diagrams corresponding to the return values.

(cons (list 1 2) 3)

(cons '() '())

(cons (cons 1 2) (cons 3 4))

'((1 2) 3 (4 (5) 6))

(list (list 1 2 (cons 3 4)))



## Pairs and lists

Exercise (harder): What does Scheme print in response to the following expressions? Also, draw the box-and-pointer diagrams corresponding to the return values.

(define x (cons 1 2))
x

(define y (cons x x))
y

(list y (car y) (cdr y))

Note: A pair is created if AND ONLY IF cons is called. (With the exception of quoted lists/pairs.)

## Pairs and lists

A list containing lists is called a “deep list”. (This was impossible on sentences)

```
> (list (list a b c) d (list e f g))
((a b c) d (e f g))
```

A deep list can be thought of as a tree (I will demonstrate thison the board) … furthermore, we can use deep lists to represent data that aren’t “flat” -- hierarchical data! Much more on this next time!

25

## Pairs and lists

Instead of keep and every, we use FILTER and MAP on lists. (It is a data abstraction violation if we used filter and map on sentences or keep and every on lists, even though it might not give us an error message.)

```
> (map (lambda (x) (if (list? x) x (+ x 1)))
       '((1 2 3) 4 5 (6 7) 8)
((1 2 3) 5 6 (6 7) 9)
> (filter pair? (list (cons 1 2) (list 3 4) 5))
((1 . 2) (3 4))
```

More details on this next time.

26

## Lists and sentences

```
Some analogies:

keep <-> filter
map <-> every
null? <-> empty?
car <-> first
cdr <-> butfirst
se <-> ?????
```

27

## Back to data abstraction

We want to create a bank account system. It stores the account holder's name, and their checking account balance, and their savings account balance. We will create our abstract data type using lists/pairs.

```
(define (make-bank-acct name checking savings)
  (list (cons 'name name)
        (cons 'checking checking)
        (cons 'savings savings)))
```

Now write the selectors for our A.D.T. (all three of them).

Now write a procedure, that takes a LIST of bank accounts, and returns the total money contained in all of the bank accounts.

28

## Back to … lambdas

Remember that I said that it is possible to write recursive procedures without using DEFINE and only using LAMBDA. Also remember that I said that anything you can do in Scheme, you can do using only LAMBDAs (in principle).

In Scheme cons, car, and cdr are primitive procedures. But they don’t have to be primitive; they can be defined in terms of lambdas -- how?

29