Lecture 9: Pairs/lists and hierarchical data

CS 61A
Summer 2006
Instructor: Kevin Lin

Administrative stuff

- The midterm location has changed.
  - Now in 145 McCon.
  - Time and date still the same.
- You don’t have to do exercise 1.16 on homework 2A, because I gave the answer away in lecture. (But you should be done with hw2a by now anyway!)

Administrative stuff

- There are old midterms on the HKN website: http://hkn.eecs.berkeley.edu
  - It is highly recommended that you try out old midterms.
  - Solutions aren’t available for many of the old midterms, but you can always just check your answers on your computer yourself.
- Reminder: Thursday is review lecture.

Review

- Pairs are constructed using CONS.
  - The selectors are CAR and CDR.
  - A list is: either the empty list or a pair whose cdr is a list. (Recursive definition!)
  - Use NULL? to test whether something is the empty list (a.k.a. the null element, a.k.a. nil, a.k.a. ‘()).
  - Do not use EMPTY? for this; that would be a data abstraction violation.

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 1 2) ‘(cons 1 2)) ; note the apostrophe!
- ‘((1 2) 3 (4 (5) 6))
- (list (list 1 2 (cons 3 4)))

Review

- Lists can also be constructed using
  - (list e₁ e₂ e₃ … eₙ)
  - This is just a shortcut for
  - (cons e₁ (cons e₂ (cons … (cons eₙ nil) … )))
- Question: Why do we put the nil in the last pair?
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 (list x x))
y

(list y (car y) (cadr y))

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

Midterm

Everything after this point will not be on the midterm.

Sentences vs. Lists

(se 'a '(b c)) ==> (a b c)

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

(se '(a b) '(c d)) ==> (a b c d)

(cons '(a b) '(c d)) ==> ((a b) c d)

(append '(a b) '(c d)) ==> (a b c d)

Higher order functions

It would be useful to have versions of keep and every that work on lists.

Why can't we use keep and every? Well first of all it would be a data abstraction violation. Second of all ...

> (every (lambda (x) (lambda (y) (+ x y)) '(* 2 3 4))

ERROR!

Using map and every

> (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 6))

{(1 . 2) (3 4)}

> (filter (lambda (p) (= (p 1) 2))

(list (lambda (x) (x x))

(lambda (x) (* x 2))

(lambda (x) (+ x 5)))

[#[closure] #[closure]]
Using map and every

```
> (map (lambda (x) (lambda (y) (+ x y))) '(1 2 3 4))
(#[closure] #[closure] #[closure] #[closure])
```

More on pairs/lists

Define a procedure `list?` that checks whether its argument is a list. You may use the primitive predicate `pair?`.

Recall that a list is either the empty list or a pair whose CDR is a list.

```
(define (list? L)
  (cond ((null? L) #t)
        ((pair? L) (list? (cdr L)))
        (else #f)))
```

Trees

Big idea: Representing a hierarchy of information.

What are trees good for?

Hierarchy: country, state, city
Hierarchy: bosses and subordinates
Ordering: binary search trees (more on this later)
Composition: arithmetic operations at branches, numbers at leaves.

We call them “trees” because of the branching structure.

A node is a point on a tree.

In the picture I’ve drawn on the board, each node contains a datum (like California or 23), but also includes the entire structure under it. So the California node includes Berkeley, Los Angeles, etc.

Therefore, each node is itself a tree.

More basic terminology:

The root node is the node (or the tree) at the top. Every tree has a root node.

The children of a node are the nodes directly beneath it.

A branch node is a node with at least one child. A leaf node is a node with no children.
First we’ll consider a common type of tree, in which every tree has at least one node, every node has a datum, and nodes can have any number of children.

We’ll call this the Tree (with a capital T!) abstract data type.

**Constructor:**

(make-tree datum children)

; datum can be anything
; children is a list of Trees

**Selectors:**

(datum node)

; returns the datum of a node

(children node)

; returns the children of a node

[NOTE: SICP’s “tree” is different from our “Tree.”]

Now, how to implement this?

**Constructor:**

(define make-tree cons)

**Selectors:**

(define datum car)

(define children cdr)

How to construct the USA tree?

(define usa-tree
(make-tree 'USA
(list (make-tree 'california
(list (make-tree 'berkeley ()
(make-tree 'LA ()
(make-tree 'SF ())
(make-tree 'LA '())
(make-tree 'SF ()
(make-tree 'LA '())
(make-tree 'SF ()
(make-tree 'LA '()))))))
(list (make-tree 'newyork
(list (make-tree 'newyork ()
(make-tree 'buffalo ()
(make-tree 'ithaca ()
(make-tree 'newyork '())
(make-tree 'buffalo '())
(make-tree 'ithaca '()))))))))

What is the proper way to retrieve the word “berkeley” from this tree? DO NOT VIOLATE THE DATA ABSTRACTION.

Mapping over trees.

(define (treemap fn tree)
(make-tree (fn (datum tree))
(map (lambda (t) (treemap fn t))
(children tree))))

This is remarkably simple and elegant, and it is very versatile!

Pay attention to the recursion. Treemap does not call itself! Treemap calls map, which in turn calls treemap.

This pattern (X calls Y which calls X …) is called mutual recursion.

We can define treemap without using map, making the process of what’s going on more visible.

(define (treemap fn tree)
(make-tree (fn (datum tree))
(map (lambda (t) (treemap fn t))
(children tree))))

; a "forest" is a list of trees

(define (forestmap fn forest)
(if (null? forest)
'()
(cons (treemap fn (car forest))
(forestmap fn (cdr forest))))))
Deep lists

More on trees tomorrow.

Another type of hierarchical data is deep lists. These are lists that contain lists that contain lists that contain lists ...

Formal definition: A deep list is either a list or a list of deep lists. (Recursion again!)

'((paul mccartney) (john lennon) (george harrison) (ringo starr))

How can we map over a deep list?

(define (deep-map fn DL)
  (map (lambda (x) (if (list? x)
                       (deep-map fn x)
                       (fn x))) DL))

Beautiful!

(deep-map square '((2 2 ((3))) 2))

==> (4 4 ((9))) 4)

Deep lists

A more transparent definition:

(define (deep-map fn DL)
  (cond ((null? DL) '())
        ((list? (car DL)) (cons (deep-map fn (car DL))
                                (deep-map fn (cdr DL)))
        (else (cons (fn (car DL))
                    (deep-map fn (cdr DL)))))))