Quiz: Box and Pointer fun!

- (cons (cons (cons ‘hey
                    (cons ‘there nil))
              nil)
        (cons ‘wow nil))

- (list ‘boo (append (list ‘hoo ‘hoo)
                     (cons ‘see ‘me)))

What we’re doing today…

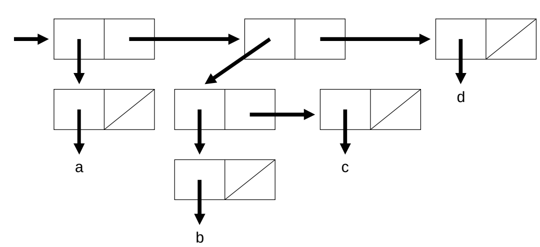
- Flat vs. Deep List Recursion
- Trees…what they are…and WHY I don’t like them…
- Tree Recursion
- Intro to DDP

Flat vs. Deep Recursion…

…ooh…that reaches all levels…

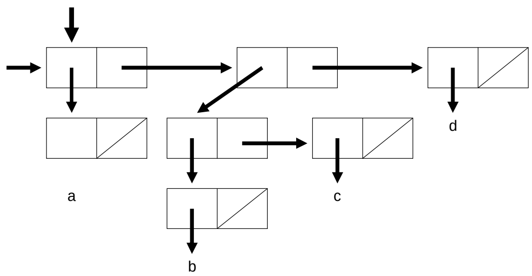
Flat vs. Deep Recursion

- Think about counting the elements in a list: ((a) ((b) c) d)



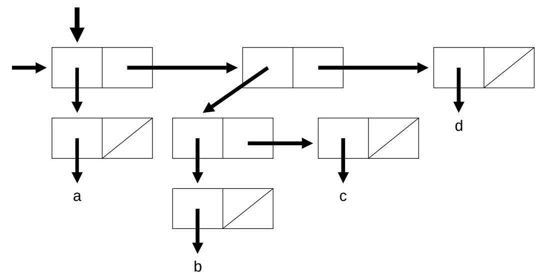
Flat vs. Deep Recursion

- How many elements are there in flat recursion?



Flat vs. Deep Recursion

- How many elements in deep recursion?



Flat vs. Deep Recursion

- So think about flat recursion as the top level of the list….so just go through the backbone of a box and pointer diagram.
- Deep recursion goes through EVERY level of the list.

Flat vs. Deep Recursion

- Last Discussion…
  - (define (deep-square L)

```
(cond ((null? L) nil)
      ((list? (car L)) (cons (deep-square (car L))
                             (deep-square (cdr L))))
      (else (cons (square (car L))
                  (deep-square (cdr L))))))
```

- You thought that was easy?...let's shorten it even more!

Flat vs. Deep Recursion

- Using pair? or list?
  - ```
    (define (deep-square L)
        (cond ((null? L) '())
              ((not (pair? L)) (square L))
              (else (cons (deep-square (car L))
                          (deep-square (cdr L))))))
    ```
- Wasn’t that easier?

Flat vs. Deep Recursion

- Templates!
  - Flat Recursion

```
(define (flat L)
    (if (null? L)
        <return value at the end of the list>
        <combine first & recurse on 'cdr' list>))
```

Flat vs. Deep Recursion

- Deep Recursion

```
(define (deep L)
    (cond ((null? L) <return value when end>)
          ((not (pair? L)) <do something to
                            element>)
          (else <combine recursive call to 'car'
                 list & recursive call to 'cdr'
                 list>)))
```

Deep Recursion Practice

- Write deep-accumulate.
  - (deep-accumulate + 0 ‘(1 (2 3) 4))
    → 10

  It should work like the 3 argument accumulate but on deep lists. No HOFs
```

Deep Recursion Practice Answers

- (define (deep-accumulate op init L)

```
(define (deep-accumulate op init L)
    (cond ((null? L) init)
          ((not (pair? L)) L)
          (else
            (op (deep-accumulate op init (car L))
                (deep-accumulate op init (cdr L))))))
```

Deep Recursion using HOFs

- It's AS easy as normal recursion.
- Let's take a closer look at what MAP does:
  - (map f (list x y z))
    → ((f x) (f y) (f z))
- What if x, y and z were lists?

Deep Recursion using HOFs

- Map DOESN'T care!
  - (map f (list '(x y z) '(a b c) '(d e f)))
    → ((f '(x y z)) (f '(a b c)) (f '(d e f)))
- Map just applies the function to all the car's of a list.
- So the question is, how can we use map on deep lists?

Deep Recursion using HOFs

- Well, look at the structure of deep-square.

```
(define (deep-square L)
    (cond ((null? L) '())
          ((not (pair? L)) (square L))
          (else (cons (deep-square (car L))
                      (deep-square (cdr L))))))
```

- Here is a new version using map:

```
(define (deep-square-map L) ;;assume L is a list
   (map (lambda (sublist) (cond ((null? sublist) sublist)
                                ((not (pair? sublist)) (square sublist))
                                (else (deep-square-map sublist))))
        L))
```

Deep Recursion Practice w/ HOF

- Write deep-appearances
  - (deep-appearances ‘a ‘(a (b c ((a))) d))
    → 2

  First version without HOFs.

  Second version with HOFs.

Deep Recursion Answer

- ```
  (define (deep-appearances x struct)
      (cond ((null? struct) 0)
            ((not (pair? struct))
             (if (equal? x struct) 1 0))
            (else (+ (deep-appearances x (car struct))
                     (deep-appearances x (cdr struct))))))
  ```

- Which condition isn’t needed in this case?

Deep Recursion w/ HOF Answer

- ```
  (define (deep-appearances x struct)
      (accumulate + 0
           (map (lambda (sublist)
                  (if (not (pair? sublist))
                      (if (equal? x sublist) 1 0)
                      (deep-appearances x sublist)))
                struct)))
  ```

Hierarchical Data…

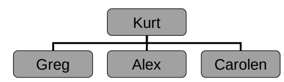
…trees…my nemesis…

Hierarchical Data

- Examples:
  - Animal Classification: Kingdom, Phylum…
  - Government: President, VP…etc.
  - CS Staff: Lecturer, TAs, Readers, Lab Assitants
  - Family **Trees**

Trees…*shudder*

- The reason as to why I don't like them…



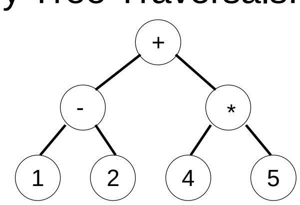
- But they're cool ☺ and they're a great way to represent the hierarchical data.

Binary Tree Traversals

- How you visit each node in a tree

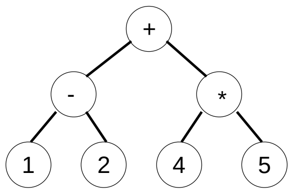
- Three ways:
  - Prefix: visit the node, left child, right child
  - Infix: visit left child, node, right child
  - Postfix: visit left child, right child, node

Binary Tree Traversals: Prefix



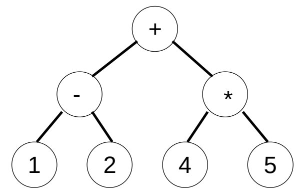
+ - 1 2 * 4 5

Binary Tree Traversals: Infix



1 – 2 + 4 * 5

Binary Tree Traversals: Postfix



1 2 – 4 5 * +

Trees? Those things outside?

- Trees are a data structure.
- They can be implemented in many ways.
  - Nodes have or don’t have data
  - Extra information can be held in each node or branch
  - We talked about this in lecture today

Trees…what do you need?

- To implement trees you need most of the following:
  - Constructor: make-tree
  - Selectors: datum, children
  - Operations: apply function on each of the datum, add/delete a child, count children, count all datum.

Tree Abstraction

- Constructor:
  (make-tree datum children)
  - returns a tree where the datum is an element and children is a list of trees
- Implementation:
  - (define (make-tree datum children)
    (cons datum children))
    OR
  - (define make-tree cons)

Tree Abstraction

- Selectors:
  (datum tree)
  - returns the element in the node of the tree
  (children tree)
  - returns a list of trees (a forest)
- Implementation:
  - (define (datum tree)
    (car tree))
    (define (children tree)
    (cdr tree))
    OR
  - (define datum car)
    (define children cdr)

Tree Abstraction

- Procedures:
  (leaf? tree)
  - returns #t if the tree has no children, otherwise #f
  (map-tree funct tree)
  - Returns a tree where each datum is (funct datum)
- Implementation:
  - (define (leaf? tree)
    (null? (children tree)))
  - We'll leave map-tree for an exercise.

Tree Abstraction Practice

- (define a-t
  '(4 (7 (8) (5 (2) (4))) (5 (7)) (3 (4 (9)))))
  - Draw a-t
    - Root: □
    - Leaves: ○
    - Underline Data
  - Use tree abstraction to construct a-t

Tree Recursion

- So how to write operations on trees…
  - So you can think of it like car/cdr recursion, but with using the tree abstraction.
  - You don't need to check for the null? tree.
  - Otherwise, you basically do something to the datum and recurse through the children.

Tree Recursion

- How would you go about counting the leaves in a tree. *<What are leaves?>*
  - Steps for count-leaves:
    - If the tree is a leaf return 1
    - Otherwise it has children, so go through the list of children by calling count-leaves on all of the children
    - Add everything up.

Tree Recursion

```
(define (count-leaves tree)
    (if (leaf? tree)
       1
       (count-leaves-in-forest (children tree))))

(define (count-leaves-in-forest list-of-trees)
    (if (null? forest)
       0
       (+ (count-leaves (car list-of-trees))
          (count-leaves-in-forest (cdr list-of-trees)))))
```

This is what we call **mutual recursion**! The two functions depend on each other

Tree Recursion

- Wait…count-list-in-forest kinda looks like…

```
(define (accumulate op init lst)
    (if (null? lst)
       init
       (op (car lst)
            (accumulate op init (cdr lst)))))
```

- And we're calling count-leaves with each child…it's like MAPPING!
- Why not use HOFs instead of creating a new procedure!

Tree Recursion w/ HOFs

```
(define (count-leaves tree)
  (cond ((null? tree) 0)
        ((leaf? tree) 1)
        (else (accumulate + 0
                          (map count-leaves (children tree))))))
```

Doesn't that look better ☺

Tree Recursion Practice

- Write tree-search
  - Takes an element and a tree
  - Returns #t if the element is found, otherwise #f
  - Use no Helper Procedures

Tree Recursion Answers

```
(define (tree-search data tree)
  (if (equal? (datum tree) data)
      #t
      (accumulate (lambda (x y) (or x y))
                  #f
                  (map (lambda (child)
                         (tree-search data child))
                       (children tree)))))
```

Tree Operation Practice

- Write map-tree (We did this in class ☹)
  - Takes a function and a tree
  - Returns a new tree where the function is applied to each of the datum
- Write update-nodes
  - Returns you a new tree where all the nodes are the sum of it's children

Tree Operation Answers

```
(define (update-nodes tree)
  (if (leaf? tree)
      tree
      (let ((new-children
              (map update-nodes (children tree))))
        (make-tree (accumulate + 0
                     (map datum new-children))
                   new-children))))
```

Next Time: DDP & Midterm Review…

…get your mind ready!