CS61A Lecture 11

2011-07-07

Colleen Lewis

Tree recursion

- A procedure in which each invocation makes more than one recursive call

Is this tree-recursion?

```
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

A) Yes B) No C)??

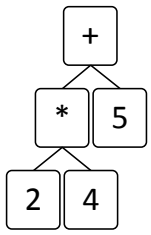
Is this tree-recursion?

```
(define (filter pred seq)
  (cond
    ((null? Seq) '())
    ((pred (car seq))
      (cons (car seq)
            (filter pred (cdr seq))))
    (else (filter pred (cdr seq)))))
```

A) Yes B) No C)??

calc-eval

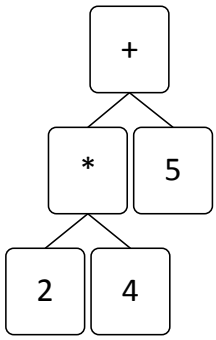
```
(define (calc-eval exp)
  (cond
     ((number? exp) exp)
     ((list? exp)
        (calc-apply
           (car exp)
           (map calc-eval (cdr exp))))
     (else (error "Calc: bad exp"))))
```



Is this tree-recursion?

A) Yes B) No C)??

Representing Math in Scheme

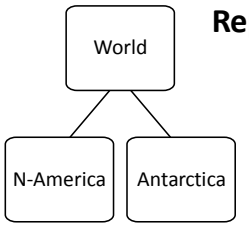



```
(+ (* 2 4) 5)
car: +
cdr: ((* 2 4) 5)
```

Capital-T Tree

- Abstract Data Type
- Not defined in the book
- The book calls any deep-list a "tree".
  - Lower-case-t tree (book) is different than Capital-T Tree

Representing Trees in Scheme



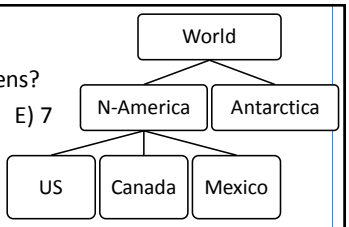
Extra set of parens

```
(World (N-America) (Antarctica))
car: World
cdr: ((N-America) (Antarctica))
```

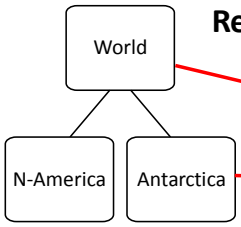
Try IT!

How many open parens?

A) 3 B) 4 C) 5 D) 6 E) 7



Representing Trees in Scheme



root node

leaf (no children)

```
(World (N-America) (Antarctica))
datum: World
children:((N-America) (Antarctica))
```

List of trees (forest)

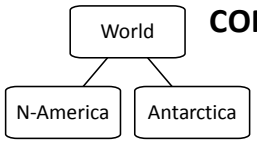
Tree constructors & selectors

```
(define (make-tree data children)
    (cons data children))

(define (datum Tree)
    (car Tree))

(define (children Tree)
    (cdr Tree))
```

children

- What type of thing is `(children tree)`?

A) List of Lists
B) List of Trees
C) List of trees
D) List
E) ??



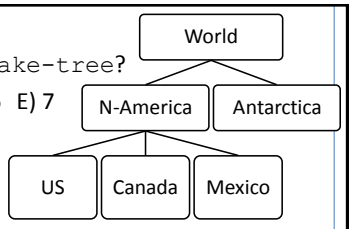
CONSTRUCTING Trees in Scheme

```
datum: World
children:((N-America) (Antarctica))
(make-tree
    'World
    (list
        (make-tree 'N-America '())
        (make-tree 'Antarctica '())))
```

Try IT!

How many calls to `make-tree`?

A) 3 B) 4 C) 5 D) 6 E) 7



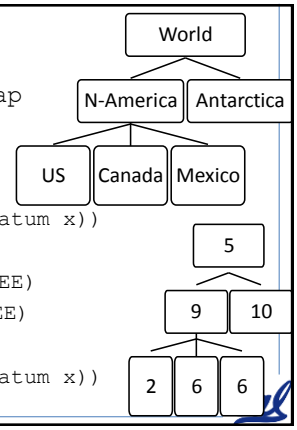
Treemap

```
STk>(define t1 (make-tree 3 '())
t1
STk>(datum t1)
3
STk>(children t1)
()
STk>(define t2 (treemap square t1))
t2
STk>(datum t2)
9
```

treemap

If `TREE` is the tree to the right, what call to `treemap` will create the below?

```
A.(treemap
      (λ(x) (length(datum x))
       TREE)
B.(treemap length TREE)
C. (treemap count TREE)
D. (treemap
      (λ(x) (count (datum x))
       TREE)
```



forest-map

```
(define (forest-map fn forest)
```

- What type of thing is a forest?

A) List of Lists
B) List of Trees
C) List of trees
D) List
E) ??

Write it without calling `map`!

It should call `treemap` on each tree

forest-map

Is using cons, car & cdr a data abstraction violation (DAV)?

A) Yes B) No C) It depends

forest-map

```
(define (forest-map fn forest)
```

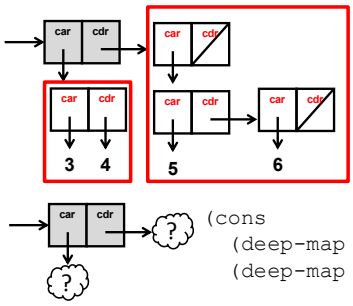
- What type of thing is a forest?

A) List of Lists
B) List of Trees
C) List of trees
D) List
E) ??

deep-map

REVIEW

(deep-map sq '((3 . 4) (5 6))




```
(cons
  (deep-map fn (car arg))
  (deep-map fn (cdr arg)))
```

deep-map solution

```
(define (deep-map fn arg)
  (cond
    ((null? arg) '())
    ((pair? arg)
       (cons
         (deep-map fn (car arg))
         (deep-map fn (cdr arg))))
    (else (fn arg))))
```

treemap

```
(define (treemap fn tree)
  (make-tree
    (fn (datum tree))
    (forest-map fn (children tree))))
```

treemap and forest-map

```
(define (treemap fn tree)
  (make-tree
    (fn (datum tree))
    (forest-map fn (children tree))))

(define (forest-map fn forest)
  (if (null? forest)
      '()
      (cons
          (treemap fn (car forest))
          (forest-map fn (cdr forest)))))
```

Mutual recursion

Why don't we need a base case in `treemap`?

forest-map using map

treeadd

```
STk>(define kid1 (make-tree 3 '())
STk>(define kid2 (make-tree 4 '())
STk>(define parent
        (make-tree '5
                   (list kid1 kid2))

STk>(treeadd parent)
12
```

treeadd and forest-add

```
(define (treeadd tree)
  (make-tree
    (datum tree)
    (forest-add (children tree))))

(define (forest-add forest)
  (if (null? forest)
    '()
    (cons
        (treeadd (car forest))
        (forest-add (cdr forest)))))
```

deep-add

```
STk> (deep-add 3)
3
STk>(deep-add '())
0
STk>(deep-add '( 1 2 3))
6
STk>(deep-add '((1 2) (3 . 2) 1))
9
```

Modify to become deep-add

```
(define (deep-add fn arg)
  (cond
    ((null? arg) '())
    ((pair? arg)
       (cons
         (deep-add fn (car arg))
         (deep-add fn (cdr arg))))
    (else (fn arg))))
```

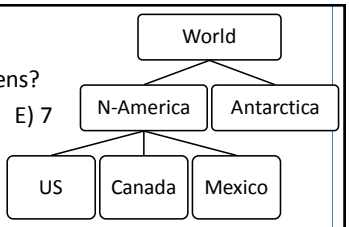
Draw the tree

```
(define a (make-tree 2 '()))
(define b (make-tree 3 '()))
(define c (make-tree 5 (list a b)))
(define d (make-tree 1 (list c)))
```

SOLUTION

How many open parens?

A) 3 B) 4 C) 5 D) 6 E) 7

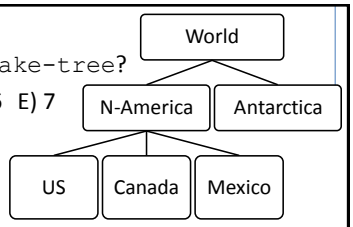



```
(World
    (N-America (US) (Canada) (Mexico))
    (Antarctica))
car: World
cdr: ((N-America (US) (Canada) (Mexico))
      (Antarctica))
```

SOLUTION

How many calls to `make-tree`?

A) 3 B) 4 C) 5 D) 6 E) 7




```
(make-tree
  'World
  (list
    (make-tree
        'N-America
        (list (make-tree 'US '())
          (make-tree 'Canada '())
          (make-tree 'Mexico '()))))
    (make-tree 'Antarctica '())))
```

forest-map using map

```
(define (forest-map fn forest)
  (map
        (lambda (one-tree)
            (tree-map fn one-tree))
        forest))
```

Solution treeadd & forest-add

```
(define (treeadd tree)
  (+ (datum tree)
     (forest-add (children tree))))
(define (forest-add forest)
  (if (null? forest)
      0
      (+ (treeadd (car forest))
     (forest-add (cdr forest)))))
```

SOLUTION deep-add

```
(define (deep-add arg)
  (cond
   ((null? arg) 0)
   ((pair? arg)
    (+
     (deep-add (car arg))
     (deep-add (cdr arg))))
   (else arg)))
```

SOLUTION Draw the tree

```
(define a (make-tree 2 '()))
(define b (make-tree 3 '()
(define c (make-tree 5 (list a b)))
(define d (make-tree 1 (list c)))
```