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)??

Representing Math in Scheme
(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)??

(calc-eval (+ (* 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

**Try IT!**

How many open parens?
A) 3  B) 4  C) 5  D) 6  E) 7

**Tree constructors & selectors**

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

(define (datum Tree)
  (car Tree))

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

**Representing Trees in Scheme**

```
(World (N-America) (Antarctica))
```

**Try IT!**

- (World (N-America) (Antarctica))

**Extra set of parens**

- (World (N-America) (Antarctica))

- (World (N-America) (Antarctica))

- (World (N-America) (Antarctica))

- (World (N-America) (Antarctica))

- (World (N-America) (Antarctica))

- (World (N-America) (Antarctica))

- (World (N-America) (Antarctica))

**Representing Trees in Scheme**

```
(World (N-America) (Antarctica))
```

**Try IT!**

- (World (N-America) (Antarctica))

**root node**

- (World (N-America) (Antarctica))

**List of trees**

- (World (N-America) (Antarctica))

- (World (N-America) (Antarctica))

- (World (N-America) (Antarctica))

- (World (N-America) (Antarctica))

- (World (N-America) (Antarctica))

- (World (N-America) (Antarctica))

- (World (N-America) (Antarctica))

**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) ??

deepe-map

REVIEW

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

(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 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

```
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)))
```
How many open parens?
A) 3  B) 4  C) 5  D) 6  E) 7

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

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 '())))

SOLUTION

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

SOLUTION

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

SOLUTION

(define (treeadd tree)
 (+ (datum tree)
    (forest-add (children tree))))

Solution treeadd & forest-add

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

forest-map using map

SOLUTION

(define (treeadd tree)
 (+ (datum tree)
    (forest-add (children tree))))

SOLUTION

(define (treeadd tree)
 (+ (datum tree)
    (forest-add (children tree))))

SOLUTION

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

SOLUTION deep-add

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

SOLUTION

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

SOLUTION Draw the tree