**Structures and Interpretation of Computer Program**

**Exercise Chapter 2.1 Name:** Wan Huzaifah bin Wan Azhar

**Exercise 2.2.2 Hierarchical Structures**



(define x (list (list 1 2) (list 3 4)))

(define k (list (list 1 2) (list 3 (list 4 5)) (list (list 2 3) 3)))

(define (deep-reverse x)

(define empty-list '())

(define (deep-reverse-iter inlist outlist)

(cond

((and (null? (cdr inlist)) (pair? (car inlist))) ; Final value before null is pair

(cons (deep-reverse-iter (car inlist) empty-list) outlist))

((null? (cdr inlist)) ; Final value before null is not pair

(cons (car inlist) outlist))

((pair? (car inlist)) ; Reverse inner list

(deep-reverse-iter (cdr inlist) (cons (deep-reverse-iter (car inlist) empty-list) outlist)))

(else ; Reverse list

(deep-reverse-iter (cdr inlist) (cons (car inlist) outlist)))))

(deep-reverse-iter x empty-list))

(display x)

(newline)

(display (deep-reverse x))

(newline)

(display k)

(newline)

(display (deep-reverse k))

Output:

((1 2) (3 4))

((4 3) (2 1))

((1 2) (3 (4 5)) ((2 3) 3))

((3 (3 2)) ((5 4) 3) (2 1))



(define (append x list)

(cons x list))

(define (fringe x)

(define empty-list '())

(define (fringe-iter item current-list)

(cond ((null? item) current-list)

((not (pair? item)) (fringe-iter '() (append item current-list)))

((pair? item) (fringe-iter (cdr item) (fringe-iter (car item) current-list)))

))

(reverse (fringe-iter x empty-list)))

(define x (list (list 1 2 3 4) (list 3 4 (list 5 7))))

(display (fringe x))

Output:

(1 2 3 4 3 4 5 7)

1. abcd

(define (make-mobile left right)

(list left right))

(define (make-branch length structure)

(list length structure))

(define (left-branch mobile)

(car mobile))

(define (right-branch mobile)

(cadr mobile))

(define (branch-length branch)

(car branch))

(define (branch-structure branch)

(cadr branch))

(define (total-weight mobile)

(let ((l-branch (left-branch mobile))

(r-branch (right-branch mobile)))

(cond ((and (pair? (branch-structure l-branch)) (pair? (branch-structure r-branch)))

(+ (total-weight (branch-structure l-branch)) (total-weight (branch-structure r-branch))))

((pair? (branch-structure l-branch))

(+ (total-weight (branch-structure l-branch)) (branch-structure r-branch)))

((pair? (branch-structure r-branch))

(+ (branch-structure l-branch) (total-weight (branch-structure r-branch))))

(else

(+ (branch-structure l-branch) (branch-structure r-branch)))

)))

(define m1 (make-mobile

(make-branch 4 6)

(make-branch 5

(make-mobile

(make-branch 3 7)

(make-branch 9 8)))))

(display (total-weight m1)) ; should be 21

Mapping over trees



(define (map proc items) ;SICP version of map

(if (null? items)

'()

(cons (proc (car items))

(map proc (cdr items)))))

(define (square-tree tree)

(map (lambda (sub-tree)

(if (pair? sub-tree)

(square-tree sub-tree)

(\* sub-tree sub-tree)))

tree))

(display (square-tree

(list 1

(list 2 (list 3 4) 5)

(list 6 7))))

Output:

(1 (4 (9 16) 25) (36 49))



(define (map proc items) ;SICP version of map

(if (null? items)

'()

(cons (proc (car items))

(map proc (cdr items)))))

(define (square x) (\* x x))

(define (tree-map proc tree)

(map (lambda (sub-tree)

(if (pair? sub-tree)

(tree-map proc sub-tree)

(proc sub-tree)))

tree))

(define (square-tree tree) (tree-map square tree))

(display (square-tree

(list 1

(list 2 (list 3 4) 5)

(list 6 7))))

Output:

(1 (4 (9 16) 25) (36 49))