; p6.scm

; A prototype for program 6 in scheme

; CS320

; 12/7/2019

; @author Jacob Byerline cssc0457

;

; Edit this file to add your documentation and function definitions.

; Leave the rest of this file unchanged.

; To run this file, you would start scheme at edoras command line prompt:

; scheme --load p6.scm, where the file is in the current directory

; and then in scheme type the load command (from the '%' prompt):

;(load "p6.scm")

;

; Defined LISTS for use with testing your functions.

(define list0 (list 'j 'k 'l 'm 'n 'o 'j) )

(define list1 (list 'a 'b 'c 'd 'e 'f 'g) )

(define list2 (list 's 't 'u 'v 'w 'x 'y 'z) )

(define list3 (list 'j 'k 'l 'm 'l 'k 'j) )

(define list4 (list 'n 'o 'p 'q 'q 'p 'o 'n) )

(define list5 '((a b) c (d e d) c (a b)) )

(define list6 '((h i) (j k) l (m n)) )

(define list7 '(f (a b) c (d e d) (b a) f) )

; My self-defined additional test cases

(define list8 '(f (a b) (d e d) (b a) f) )

(define list9 '(a))

(define list10 '())

;

; Here is a typical function definition from Sebesta Ch. 15

(define (adder lis)

(cond

((null? lis) 0)

(else (+ (car lis) (adder (cdr lis))))

))

; The above five lines are the sort of definition you would need to add to

; this file if asked to define an ADDER function.

; Uncomment and complete the following four definitions. At least have ODDS

; so the program can be tested.

; Self made reverse function

(define (rev myFirstList) ; Define reverse function

(if (null? myFirstList) ; Check if null

‘() ; If null, return empty

(append (rev (cdr myFirstList)) (cons (car myFirstList) ‘())))) ; Else, take the first element from front of list and add it to the back of a new list

; Function to find odd indices of list

(define (find\_odds myFirstList) ; Define odds function

(if (null? myFirstList) ; Check if null

'() ; If null, return empty list

(cons (car myFirstList) ; Else, construct a list of first elements

(if (null? (cdr myFirstList)) ‘() (find\_odds (cdr (cdr myFirstList))))))) ; As long as the end is not null, skip an index and recursively run again

; Error checking for odds

(define (odds myFirstList) ; Define error checking function

(if (list? myFirstList) ; Check if entered list is a list

(find\_odds myFirstList) ; If it is, proceed with finding odds

(error "USAGE: (odds {LIST})"))) ; Else, print error

; Function to find even reverse list

(define (find\_evenrev myFirstList) ; Define evenrev function

(cond ((null? myFirstList) '()) ; If the list is null, return empty list

((null? (cdr myFirstList)) '()) ; If the back elements of list are empty, return empty

(else (append (find\_evenrev (cddr myFirstList)) (list (cadr myFirstList)))))) ; Else, recursively append the first even element to the back of a new list

; Error checking for evenrev

(define (evenrev myFirstList) ; Define error checking function

(if (list? myFirstList) ; Check if entered list is a list

(find\_evenrev myFirstList) ; If it is, proceed with finding evenrev

(error "USAGE: (evenrev {LIST})"))) ; Else, print error

; Function to find penultimate

(define (find\_penultimate myFirstList) ; Define penultimate function

(cond ((null? myFirstList) '()) ; If the list is null, return empty list

((null? (cdr myFirstList)) '()) ; If the back elements of list are empty, return empty

((equal? (length myFirstList) 2) (cons (car myFirstList)'())) ; If length is 2, construct list using front

(else (find\_penultimate(cdr myFirstList))))) ; Else, recursively construct list using back

; Error checking for penultimate

(define (penultimate myFirstList) ; Define error checking function

(if (list? myFirstList) ; Check if entered list is a list

(find\_penultimate myFirstList) ; If it is, proceed with finding penultimate

(error "USAGE: (penultimate {LIST})"))) ; Else, print error

; Function to find palindrome, using my reverse function above

(define (find\_palindrome myFirstList) ; Define palindrome function

(cond ((null? myFirstList) #t) ; If list is null, return true

((equal? (rev myFirstList) myFirstList) #t) ; If the reverse of the list equals the list, return true

; (uses a RECURSIVE reverse function)

(else #f))) ; Else, return false

; Error checking for palindrome

(define (palindrome myFirstList) ; Define error checking function

(if (list? myFirstList) ; Check if entered list is a list

(find\_palindrome myFirstList) ; If it is, proceed with finding palindrome

(error "USAGE: (palindrome {LIST})"))) ; Else, print error