Brian Cullinan

Principles of Languages

Assignment 2 - Gentle as a newborn lamb(da)

Bjc76

Homework 2 - The objective of this assignment is to get familiar with the key fundamentals of scheme programming and the functional paradigm.

In Part 1, each of the 3 sections illustrates a different key concept of functional programming.

**ndelete** - a very simple function for deleting the nth character of a top level list. To test this function I used the example variable from the site.

**deep-member** - also a simple function traverses and entire multi-dimensional list and finds an input value. As you can see, my function erases the lines between data types, and a symbol d returns true even if the d in the list is a string.

**mergesort** - This illustrates schemes ability to minimize the amount of code, that would take many lines of code in other languages. Because scheme is function, this mergesort is very quick and easy to implement. Because we are using a sorting algorithm, I decided that is wasn’t crucial to preserve the order of symbols when splitting the array. The reason being is it is more efficient to just split them, even arbitrarily, because the objective of sorting the list can still be obtained, and processing time is saved (another benefit of short code).

**fn-maker** - this creates a function based on a specific input and displays the function name and verification that the function has been created.

**tic-tac-toe** - shows how manipulating state variables is not easy in scheme. I used 1 set command within a function called from 2 different places. Some functions could be more efficient or recursive like the ttstat or make-diags function, but it all works the way it currently is without violating schemes principles.

**ndelete:**

(define x '(1 2 3 4 5 6 7 8 9 10))

(display (ndelete x 2))

Output:

(1 3 5 7 9)

**deep-member:**

(define y '(((a) b) (c j ("d" (f g) h)) i))

(display (deep-member? 'd y))

Output:

#t

**mergesort:**

(define x3 '(I can "go" 4 "about" 123 sodas k?))

(display (mergesort x3))

Output:

(4 123 about can go i k? sodas)

**fn-maker:**

(define temp '((name: addtwo) (args: x) (body: (+ x 2))))

(fn-maker temp)

(display (addtwo 5))

Output:

addtwo ;yay the function wuz made

7

**tic-tac-toe:**

; main global definition of ttboard

(define ttboard '((\_ \_ \_) (\_ \_ \_) (\_ \_ \_) ))

; test functions

(ttnew-game)

(ttstat)

(ttplay 'x 3 3)

(ttplay 'o 2 3)

(ttplay 'x 2 3)

(ttplay 'x 1 1)

(ttplay 'o 33 -45)

(ttplay 'o 3 1)

(ttplay 'x 2 2)

Output:

(\_ \_ \_)

(\_ \_ \_)

(\_ \_ \_)

((\_ \_ \_) (\_ \_ \_) (\_ \_ \_))

(\_ \_ \_)

(\_ \_ \_)

(\_ \_ x)

((\_ \_ \_) (\_ \_ \_) (\_ \_ x))

(\_ \_ \_)

(\_ \_ o)

(\_ \_ x)

((\_ \_ \_) (\_ \_ o) (\_ \_ x))

Can't place a piece there! Try again!

(x \_ \_)

(\_ \_ o)

(\_ \_ x)

((x \_ \_) (\_ \_ o) (\_ \_ x))

Can't place a piece there! Try again!

(x \_ \_)

(\_ \_ o)

(o \_ x)

((x \_ \_) (\_ \_ o) (o \_ x))

(x \_ \_)

(\_ x o)

(o \_ x)

((x \_ \_) (\_ x o) (o \_ x))

The winnar is: x

**Code:**

**ndelete:**

; delete the nth symbol in a top level list

(define ndelete

(lambda (thelist nth)

; start the counter at the nth position and count down

(ndelete\_helper thelist nth nth)

)

)

; this function has an extra attribute just to control counting down

(define ndelete\_helper

(lambda (thelist count nth)

; return null when we get to the end of the list

(if (null? thelist)

thelist

; skip over car when we count down to 1

(if (= 1 count)

; only return the helper of the cdr because the car is being removed

(ndelete\_helper (cdr thelist) nth nth)

; combine the car with the helper of the cdr

(cons (car thelist) (ndelete\_helper (cdr thelist) (- count 1) nth))

)

)

)

)

; predefined variables from assignment

(define x '(1 2 3 4 5 6 7 8 9 10))

(define y '(((a) b) (c d (d (f g) h)) i))

; call ndelete and print out the output

(display (ndelete x 2))

**deep-member:**

; check a complex list and search for an item in it

(define deep-member?

(lambda (needle haystack)

; if the hackstack is empty we know for sure the answer is false

(if (null? haystack)

#f

(if (list? (car haystack))

; check the car list then check the cdr list either one can return true

(or (deep-member? needle (car haystack)) (deep-member? needle (cdr haystack)))

; convert the symbol to a string and compare it

;(if (eq? needle (car haystack))

(if (string=? (convert needle) (convert (car haystack)))

#t

; keep checking the cdr of the list

(deep-member? needle (cdr haystack))

)

)

)

)

)

;need help converting numbers to string and symbols to strings

(define convert

(lambda (x)

; already a string just return

(if (string? x)

x

;convert symbol

(if (symbol? x)

(symbol->string x)

;convert number

(if (number? x)

(number->string x)

;weird type, just return false

'"E"

)

)

)

)

)

; predefined variables from assignment

(define x '(1 2 3 4 5 6 7 8 9 10))

(define y '(((a) b) (c j ("d" (f g) h)) i))

; call deep-member and print out the output

(display (deep-member? 'd y))

**mergesort:**

; steamroller to flatten array from last years assignment (wrote it myself)

(define steamroller

(lambda (x)

(if (null? x)

x

(if (list? (car x))

(if (null? (cdar x))

(steamroller (cons (caar x) (steamroller (cdr x))))

(steamroller (cons (caar x) (steamroller (cons (steamroller (cdar x)) (steamroller (cdr x))))))

)

(cons (car x) (steamroller (cdr x)))

)

)

)

)

; merge sort function from assignment page

(define mergesort

(lambda (alist)

(if (null? (cdr alist)) alist

(let ((splits (splitter alist)))

(merge (mergesort (car splits)) (mergesort (cadr splits)))

)

)

)

)

; this calls a splitter helper function

(define splitter

(lambda (alist)

(splitter\_helper2 '() '() alist)

)

)

; this splitter does not preserve the arrangment of the atoms,

; since they are being merge sorted efficiency is more important

(define splitter\_helper2

(lambda (first-half second-half last)

(if (null? last)

(cons first-half (cons second-half '()))

(if (null? (cdr last))

(cons (cons (car last) first-half) (cons second-half '()))

(splitter\_helper2 (cons (car last) first-half) (cons (cadr last) second-half) (cddr last))

)

)

)

)

(define splitter\_helper

(lambda (alternate alist)

(if (null? alist)

alternate

(if (null? (cdr alist))

(cons (car alist) alternate)

(splitter\_helper (cons (cadr alist) (cons (car alist) alternate)) (cddr alist))

)

)

)

)

; merge two lists together

(define merge

(lambda (list1 list2)

(if (null? list1)

list2

(if (null? list2)

list1

; if is a number then do a number comparison

(if (and (number? (car list1)) (number? (car list2)))

(if (< (car list1) (car list2))

(cons (car list1) (merge (cdr list1) list2))

(cons (car list2) (merge list1 (cdr list2)))

)

; do a string comparison instead to sort atoms and strings

(if (string<? (convert (car list1)) (convert (car list2)))

(cons (car list1) (merge (cdr list1) list2))

(cons (car list2) (merge list1 (cdr list2)))

)

)

)

)

)

)

;need help converting numbers to string and symbols to strings

(define convert

(lambda (x)

; already a string just return

(if (string? x)

x

;convert symbol

(if (symbol? x)

(symbol->string x)

;convert number

(if (number? x)

(number->string x)

;weird type, just return false

'"E"

)

)

)

)

)

;; testing data:

(define x2 '(how does he do that cool stuff?))

(define x3 '(I can "go" 4 "about" 123 sodas k?))

(define x1 '(12 23 2 5 64 23 6756 234 2 42 535))

; merge some stuff

(display (mergesort x3))

**fn-maker:**

; create a function based on input

(define fn-maker

(lambda (fn-spec)

(eval (list 'define (cadar fn-spec) (cons 'lambda (cons (cdadr fn-spec) (cdaddr fn-spec)))))

; display verification that the function was created

(display (cadar fn-spec))

(display " ;yay the function was made!\n")

)

)

(define temp '((name: addtwo) (args: x) (body: (+ x 2))))

(fn-maker temp)

(display (addtwo 5))

**tic-tac-toe:**

; resets the ttboard variable

(define ttnew-game

(lambda ()

(set-helper '((\_ \_ \_) (\_ \_ \_) (\_ \_ \_) ))

)

)

; displays the ttboard

(define ttstat

(lambda ()

(display (car ttboard))

(display "\n")

(display (cadr ttboard))

(display "\n")

(display (caddr ttboard))

(display "\n")

(display ttboard)

(display "\n")

)

)

; allows recursive calling on the ttstat

(define ttstat-helper

(lambda (null)

(ttstat)

)

)

; the main action function passes variables to ttplay-helper, and determines what to print out

(define ttplay

(lambda (symbol row col)

(let ((error (ttplay-helper symbol row col))

(winnar (checkwin))

)

(if (eq? winnar #f)

(display "\n")

(if (eq? winnar 'x)

(display "The winnar is: x")

(display "The winnar is: o")

)

)

)

)

)

; calls the checkposition and set-helper functions

(define ttplay-helper

(lambda (symbol row col)

(let ((isvalid (checkposition col row))

)

(if (eq? isvalid #t)

(ttstat-helper (set-helper (nth-replace (nth-replace symbol (nth ttboard row) col) ttboard row)))

(display "Can't place a piece there! Try again!")

)

)

)

)

; simply sets the new board based on the alist input

(define set-helper

(lambda (alist)

(set! ttboard alist)

)

)

;; Simply checks to make sure a proposed placement position is on the board and not occupied. Returns false if

;; the proposed position won't work.

(define checkposition

(lambda (xpos ypos)

(not (or (invalid? xpos) (invalid? ypos)

(not (eq? (nth (nth ttboard ypos) xpos) '\_))))

)

)

;; Just a little helper function to check to see if an x,y position is even on the board.

(define invalid?

(lambda (pos)

(if (or (< pos 1) (> pos 3))

#t

#f)))

;; Helper function. Returns nth element in an input list.

(define nth

(lambda (alist num)

(if (eq? num 1)

(car alist)

(nth (cdr alist) (- num 1))

)

)

)

; replaces the nth (num) in alist element with the new element

(define nth-replace

(lambda (new alist num)

(if (eq? num 1)

(cons new (cdr alist))

(cons (car alist) (nth-replace new (cdr alist) (- num 1)))

)

)

)

; calls checkwin

(define checkwin

(lambda ()

(checkwin-helper '())

)

)

; allows for recursive calling via the null attribute

(define checkwin-helper

(lambda (null)

(let ((rows ttboard)

(cols (make-cols ttboard))

(diags (make-diags)))

(or (findwinners rows) (findwinners cols) (findwinners diags)))))

; create colums recursively

(define make-cols

(lambda (board)

(if (null? (cdar board))

(cons (list (caar board) (caadr board) (caaddr board)) '())

(cons (list (caar board) (caadr board) (caaddr board)) (make-cols (list (cdar board) (cdadr board) (cdaddr board))))

)

)

)

; make dialogs stupidly

(define make-diags

(lambda ()

(list (list (caar ttboard) (cadadr ttboard) (car (cdr (cdr (caddr ttboard))))) (list (caddar ttboard) (cadadr ttboard) (caaddr ttboard)))

)

)

; checks if any of the list elements match

(define findwinners

(lambda (alist)

(if (null? alist)

#f

(or (findwinners (cdr alist)) (all-match (car alist)))

)

)

)

; checks if all the elements in a list are the same

; not recursive

(define all-match

(lambda (alist)

(if (not (eq? (car alist) '\_))

(if (eq? (car alist) (cadr alist))

(if (eq? (cadr alist) (caddr alist))

(car alist)

#f

)

#f

)

#f

)

)

)

; main global definition of ttboard

(define ttboard '((\_ \_ \_) (\_ \_ \_) (\_ \_ \_) ))

; test functions

(ttnew-game)

(ttstat)

(ttplay 'x 3 3)

(ttplay 'o 2 3)

(ttplay 'x 2 3)

(ttplay 'x 1 1)

(ttplay 'o 33 -45)

(ttplay 'o 3 1)

(ttplay 'x 2 2)