(load "chez-init.ss") ; Section 01

(define-datatype kontinuation kontinuation?

[init-k]

[flatten-cdr-k

(ls list?)

(k kontinuation?)]

[flatten-car-k

(flattened-cdr list? )

(k kontinuation?)]

[append-k

(first symbol?)

(k kontinuation?)])]

)

(define (apply-k k v)

(cases kontinuation? k

[init-k ()

(pretty-print v)

(read-flatten-print)]

[flatten-cdr-k (ls k)

(if (list? (car ls))

(flatten-cps (car ls)

(flatten-car-k v k))

(apply-k k (cons (car ls) v)))]

[flatten-car-k (flattened-cdr k)

(append-cps v flattened-cdr k)]

[append-k (first k)

)

(define read-flatten-print

(lambda ()

(display "enter slist to flatten: ")

(let ([slist (read)])

(unless (eq? slist 'exit)

(flatten-cps slist (init-k))))))

(define flatten-cps

(lambda (ls k)

(if (null? ls)

(apply-k k ls)

(flatten-cps (cdr ls)(flatten-cdr-k ls k)))))

(define append-cps

(lambda (l1 l2 k)

(if (null? l1)

(apply-k k l2)

(append-cps (cdr l1)

l2

(append-k (car l1) k)))))

(lambda (appended-cdr)

(apply-k k (cons (car l1)

appended-cdr)))))))

(load "chez-init.ss") ; Section 02

(define read-flatten-print

(lambda ()

(display "enter slist to flatten: ")

(let ([slist (read)])

(unless (eq? slist 'exit)

(flatten-cps slist (init-k))))))

(define-datatype kontinuation kontinuation?

[init-k]

[flatten-cdr-k

(ls list?)

(k kontinuation?)]

[flatten-car-k

(flattened-cdr list? (k kontinuation?))

(apply-k k (cons (car ls) v))))))))

))

(define (apply-k k v)

(cases kontinuation? k

[init-k ()

(pretty-print v)

(read-flatten-print)]

[flatten-cdr-k (ls k)

(if (list? (car ls))

(flatten-cps (car ls) (flatten-car-k v k))

(apply-k k (cons (car ls) v)))]

[flatten-car-k (flattened-cdr k)

(lambda (u) (append-cps u v k)))

(define flatten-cps

(lambda (ls k)

(if (null? ls)

(apply-k k ls)

(flatten-cps (cdr ls) (flatten-cdr-k ls k)))))

(lambda (v) (if (list? (car ls))

(flatten-cps (car ls)

(lambda (u) (append-cps u v k)))

(apply-k k (cons (car ls) v))))))))

(define append-cps

(lambda (l1 l2 k)

(if (null? l1)

(apply-k k l2)

(append-cps (cdr l1)

l2

(lambda (appended-cdr)

(apply-k k (cons (car l1)

appended-cdr)))))))

(load "chez-int.ss") ; section 03

(define read-flatten-print

(lambda ()

(display "enter slist to flatten: ")

(let ([slist (read)])

(unless (eq? slist 'exit)

(flatten-cps slist (init-k))))))

(define-datatype kontinuation kontinuation?

[init-k]

[flatten-cdr-k (ls list?) (k kontinuation?)]

(define apply-k

(lambda (k v)

(cases kontinuation k

[init-k ()

(pretty-print val)

(read-flatten-print)]

[flatten-cdr-k (ls k)

(if (list? (car ls))

(flatten-cps (car ls)

(lambda (u) (append-cps u v k)))

(apply-k k (cons (car ls) v)))]

[flatten-car-k (v k)

(define flatten-cps

(lambda (ls k)

(if (null? ls)

(apply-k k ls)

(flatten-cps (cdr ls) (flatten-cdr-k ls k)))))

(define append-cps

(lambda (l1 l2 k)

(if (null? l1)

(apply-k k l2)

(append-cps (cdr l1)

l2

(lambda (appended-cdr)

(apply-k k (cons (car l1)

appended-cdr)))))))

(load "chez-init.ss") ; Section 04

(define read-flatten-print

(lambda ()

(display "enter slist to flatten: ")

(let ([slist (read)])

(unless (eq? slist 'exit)

(flatten-cps slist (rfp-k))))))

(define-datatype kontinuation kontinuation?

[rfp-k]

[append-k (ls list?) (k kontinuation?)]

)

(define (apply-k k v)

(cases kontinuation k

[rfp-k ()

(pretty-print v)

(read-flatten-print)]

))

; code to add to apply-k

(lambda (appended-cdr)

(apply-k k (cons (car l1)

appended-cdr)))))))

(define flatten-cps

(lambda (ls k)

(if (null? ls)

(apply-k k ls)

(flatten-cps (cdr ls)

(lambda (v) (if (list? (car ls))

(flatten-cps (car ls)

(lambda (u) (append-cps u v k)))

(apply-k k (cons (car ls) v))))))))

(define append-cps

(lambda (l1 l2 k)

(if (null? l1)

(apply-k k l2)

(append-cps (cdr l1)

l2

(append-k ls k))))