```
;; CS 135 :: Fall 2017 :: Posted solution :: A08 :: intro.rkt


;; ------ Q2a ------

;;(keep-ints lst) produces a list of the integers in lst
;; keep-ints: (listof Any) -> (listof Int)
;; Examples:
(check-expect (keep-ints empty) empty)
(check-expect (keep-ints '(a 1 "b" 2)) '(1 2))

(define (keep-ints lst)
  (filter integer? lst))


;; ------ Q2b ------

;;(contains? elem lst) determines if elem is in lst
;; contains?: Any (listof Any) -> Bool
;; Examples:
(check-expect (contains? 7 empty) false)
(check-expect (contains? 'fun '(racket is fun)) true)
(check-expect (contains? 0 '(1 2 3)) false)

(define (contains? elem lst)
  (ormap (lambda (x) (equal? x elem)) lst))


;; ------ Q2c ------

;; An AL is a (listof (list Num Str))
;; requires: each key is unique

;;(lookup-al key alst) produces the value in alst associated with key
;;   or false is there is no such key
;; lookup-al: Num AL -> (anyof false Str)
;; Examples:
(check-expect (lookup-al 'a empty) false)
(check-expect (lookup-al 2 '((1 "a")(2 "b")(3 "c"))) "b")
(check-expect (lookup-al 4 '((1 "a")(2 "b")(3 "c"))) false)

(define (lookup-al key alst)
  (local [(define result (filter (lambda (x) (equal? key (first x))) alst))]
    (cond [(empty? result) false]
          [else (second (first result))])))


;; ------ Q2d ------

;;(extract-keys alst) produces a list of every key in alst
;; extract-keys: AL -> (listof Num)
;; Examples:
(check-expect (extract-keys empty) empty)
(check-expect (extract-keys '((1 "a")(2 "b")(3 "c"))) '(1 2 3))

(define (extract-keys alst)
  (map first alst))
```

```
;; ------ Q2e ------

;;(sum-positive lst) sums the positive numbers in lst
;; sum-positive (listof Int) -> Int
;; Examples:
(check-expect (sum-positive empty) 0)
(check-expect (sum-positive '(5 -3 4)) 9)

(define (sum-positive lst)
  (foldr + 0 (filter positive? lst)))


;; ------ Q2f ------

;;(countup-to n b) produces a list from n...b
;; countup-to: Int Int -> (listof Int)
;; requires: n <= b
;; Examples:
(check-expect (countup-to 6 8) '(6 7 8))
(check-expect (countup-to 6 6) '(6))
(check-expect (countup-to -3 2) '(-3 -2 -1 0 1 2))

(define (countup-to n b)
  (build-list (add1 (- b n)) (lambda (i) (+ n i))))


;; ------ Q2g ------

;;(shout los) produces a list of the upper case version of each
;;   string in los
;; shout: (listof Str) -> (listof Str)
;; Examples:
(check-expect (shout empty) empty)
(check-expect (shout '("ah")) '("AH"))
(check-expect (shout '("get" "off" "my" "lawn")) '("GET" "OFF" "MY" "LAWN"))

(define (shout los)
  (map (lambda (s) (list->string (map char-upcase (string->list s)))) los))


;; ------ Q2h ------

;;(make-validator lst) produces a function that determines if
;;   its argument is an element in lst
;; make-validator: (listof Any) -> (Any -> Bool)
;; Examples:
(check-expect ((make-validator empty) 'elem) false)
(check-expect ((make-validator '(1 2 3)) 'elem) false)
(check-expect ((make-validator '(1 2 3)) 2) true)

(define (make-validator lst)
  (lambda (x) (member? x lst)))



;; CS 135 :: Fall 2017 :: Posted solution :: A08 :: nestlist.rkt
```

```
(define sample '(1 (2 3) () ((4))))
(define slide77 '((1 (2 3)) 4 (5 (6 7 8) 9)))

;; A Nested-Listof-X is one of:
;; * empty
;; * (cons X Nested-Listof-X)
;; * (cons Nested-Listof-X Nested-Listof-X)


;; ------ Q3a ------

;;(nfoldr fsingle flist base nlst) abstracts structural recursion on a nested
;;   list, nlst, with a given base case by combining non-list elements with
;;   fsingle, and list elements with flist
;; nfoldr: (X Y -> Y) (Y Y -> Y) Y Nested-Listof-X -> Y
;; Examples:
(check-expect (nfoldr + + empty empty) empty)
(check-expect (nfoldr cons append empty sample) '(1 2 3 4))

(define (nfoldr fsingle flist base nlst)
  (cond [(empty? nlst) base]
        [(list? (first nlst)) (flist (nfoldr fsingle flist base (first nlst))
                                     (nfoldr fsingle flist base (rest nlst)))]
        [else (fsingle (first nlst) (nfoldr fsingle flist base (rest nlst)))]))

;; Tests:
(check-expect (nfoldr cons cons empty sample) sample)
(check-expect (nfoldr + + 0 sample) 10)


;; ------ Q3b ------

;;(nfilter pred? nlst) keeps all elements in nlst that satisfy pred?
;; nfilter: (X -> Bool) Nested-Listof-X -> Nested-Listof-X
;; Example:
(check-expect (nfilter even? empty) empty)
(check-expect (nfilter odd? slide77) '((1 (3)) (5 (7) 9)))

(define (nfilter pred? nlst)
  (nfoldr (lambda (frst rec) (cond [(pred? frst) (cons frst rec)]
                                   [else rec]))
          cons empty nlst))


;; ------ Q3c ------

;;(nmap f nlst) changes all elements x in nlst to be (f x)
;; nmap: (X -> Y) Nested-Listof-X -> Nested-Listof-Y
;; Example:
(check-expect (nmap - empty) empty)
(check-expect (nmap sqr slide77) '((1 (4 9)) 16 (25 (36 49 64) 81)))

(define (nmap f nlst)
  (nfoldr (lambda (frst rec) (cons (f frst) rec)) cons empty nlst))


;; ------ Q3d ------
```

```
;;(nreverse nlst) reverses every nested list in nlst
;; nreverse: Nested-Listof-X -> Nested-Listof-X
;; Example:
(check-expect (nreverse empty) empty)
(check-expect (nreverse slide77) '((9 (8 7 6) 5) 4 ((3 2) 1)))

(define (nreverse nlst)
  (nfoldr (lambda (frst rec) (append rec (list frst)))
          (lambda (frst rec) (append rec (list frst)))
          empty nlst))


;; ------ Q3e ------

;;(nheight nlst) determines the height of nlst
;; nheight: Nested-Listof-X -> Nat
;; Example:
(check-expect (nheight empty) 1)
(check-expect (nheight '(a b c)) 1)
(check-expect (nheight slide77) 3)

(define (nheight nlst)
  (nfoldr (lambda (frst rec) rec)
          (lambda (frst rec) (max (add1 frst) rec)) 1 nlst))


;; ------ Q3f ------

;;(prune nlst) removes all empty nested lists in nlst
;; prune: Nested-Listof-X -> Nested-Listof-X
;; Examples:
(check-expect (prune empty) empty)
(check-expect (prune '(1 (2 3 ()) ( (()) (4) ()((((((() '(1 (2 3) ((4))))
(check-expect (prune '((()(((())))))) '(()))

(define (prune nlst)
  (nfoldr cons (lambda (frst rec) (cond [(empty? frst) rec]
                                        [else (cons frst rec)]))
          empty nlst))


;; CS 135 :: Fall 2017 :: Posted solution :: A08 :: suggest.rkt


;; A Word is a Str
;; requires: only lowercase letters appear in the word
;;           (no spaces, punctuation, etc.)

(define letters (string->list "abcdefghijklmnopqrstuvwxyz"))


;; ------ Q4a ------

;;(remove-dups slst) removes all duplicate elements from slst
;; remove-dups: (listof Word) -> (listof Word)
;; requires: slst is sorted in non-decreasing order
```

```
;; Examples:
(check-expect (remove-dups empty) empty)
(check-expect (remove-dups '("a" "a" "b" "c" "c" "c" "d"))
              '("a" "b" "c" "d"))

(define (remove-dups slst)
  (foldr (lambda (frst rec) (cond [(empty? rec) (list frst)]
                                  [(equal? frst (first rec)) rec]
                                  [else (cons frst rec)]))
         empty slst))


;; ------ Q4b ------

;;(ifoldr combine base lst) abstracts structural recursion
;;   on a lst given a base case, by combining each element
;;   with its position in the list
;; ifoldr: (Nat X Y -> Y) Y (listof X) -> Y
;; Examples:
(check-expect (ifoldr + empty empty) empty)
(check-expect (ifoldr (lambda (i x y) (cons (list i x) y)) empty '(a b c))
              '((0 a) (1 b) (2 c)))

(define (ifoldr combine base lst)
  (local [(define (ifoldr/count i rlst)
            (cond [(empty? rlst) base]
                  [else (combine i (first rlst)
                                 (ifoldr/count (add1 i) (rest rlst)))]))]
    (ifoldr/count 0 lst)))


;; ------ Q4c ------

;;(remove-at i lst) removes element with index i from lst
;; remove-at: Nat (listof Any) -> (listof Any)
;; Examples:
(check-expect (remove-at 0 '(a b c d)) '(b c d))
(check-expect (remove-at 3 '(a b c d)) '(a b c))
(check-expect (remove-at 0 '()) '())

(define (remove-at i lst)
  (ifoldr (lambda (k x y)
            (cond [(= i k) y]
                  [else (cons x y)]))
          empty lst))


;;(remove-letters s) produces a list of Words,
;;   each with one letter removed from s
;; remove-letters: Word -> (listof Word)
;; Examples:
(check-expect (remove-letters "abc") '("bc" "ac" "ab"))
(check-expect (remove-letters "") '())

(define (remove-letters s)
  (local [(define loc (string->list s))]
    (build-list (length loc) (lambda (i) (list->string (remove-at i loc))))))
```

```
;;(insert-single-i letter i loc) inserts letter before
;;  the ith character in loc
;; insert-single-i: Char Nat (listof Char) -> Word
;; Examples:
(check-expect (insert-single-i #\a 0 empty) "")
(check-expect (insert-single-i #\a 5 (string->list "bbb")) "bbb")
(check-expect (insert-single-i #\a 1 (string->list "bbb")) "babb")

(define (insert-single-i letter i loc)
  (list->string (ifoldr (lambda (k frst rec)
                          (cond
                            [(= i k) (cons letter (cons frst rec))]
                            [else (cons frst rec)]))
                        empty loc)))


;;(insert-single letter loc) produces each possible variation of loc
;;   with letter inserted before a letter in loc
;; insert-single: Char (listof Char) -> (listof Word)
;; Examples:
(check-expect (insert-single #\a empty) empty)
(check-expect (insert-single #\a (string->list "bbb"))
              (list "abbb" "babb" "bbab"))

(define (insert-single letter loc)
  (build-list (length loc)
              (lambda (i) (insert-single-i letter i loc))))


;;(insert-letters s) produces each variation of s with a letter inserted
;;   before an existing letter in s
;; insert-letters: Word -> (listof Word)
;; Examples:
(check-expect (insert-letters "") empty)
(check-expect (member? "azbc" (insert-letters "abc")) true)

(define (insert-letters s)
  (foldr (lambda (frst rec)
           (append (insert-single frst (string->list s)) rec))
         empty letters))


;;(trailing-letters s) produces each variation of s with a letter
;;   added to the end of s
;; trailing-letters: Word -> (listof Word)
;; Examples:
(check-expect (member? "a" (trailing-letters "")) true)
(check-expect (member? "abcd" (trailing-letters "abc")) true)

(define (trailing-letters s)
  (local
    [(define loc (string->list s))]
    (map (lambda (letter) (list->string (append loc (list letter))))
         letters)))


;;(replace-single-i letter i loc) replaces the ith element of
```

```
;;   loc with letter
;; replace-single-i: Char Nat (listof Char) -> Word
;; Examples:
(check-expect (replace-single-i #\a 0 empty) "")
(check-expect (replace-single-i #\a 1 (string->list "bbb")) "bab")

(define (replace-single-i letter i loc)
  (list->string (ifoldr (lambda (k frst rec)
                          (cond
                            [(= k i) (cons letter rec)]
                            [else (cons frst rec)]))
                        empty loc)))


;;(replace-single letter loc) produces each possible variation of loc
;;   with letter replacing an element of loc
;; replace-single: Char (listof Char) -> (listof Word)
;; Examples:
(check-expect (replace-single #\a empty) empty)
(check-expect (member? "bba" (replace-single #\a (string->list "bbb"))) true)

(define (replace-single letter loc)
  (build-list (length loc)
              (lambda (i) (replace-single-i letter i loc))))


;;(replace-letters s) produces each variation of s with a letter replacing
;;   an existing letter in s
;; replace-letters: Word -> (listof Word)
;; Examples:
(check-expect (replace-letters "") empty)
(check-expect (member? "bzb" (replace-letters "bbb")) true)

(define (replace-letters s)
  (foldr (lambda (frst rec)
           (append (replace-single frst (string->list s)) rec))
         empty letters))


;;(swap-at-i i loc) swaps the ith and i+1th characters
;;   of loc (if they exist)
;; swap-at-i: Nat (listof Char) -> Word
;; Examples:
(check-expect (swap-at-i 0 empty) "")
(check-expect (swap-at-i 0 (string->list "a")) "a")
(check-expect (swap-at-i 1 (string->list "abcd")) "acbd")

(define (swap-at-i i loc)
  (list->string (ifoldr (lambda (k frst rec)
                          (cond
                            [(and (= k i) (not (empty? rec)))
                             (cons (first rec) (cons frst (rest rec)))]
                            [else (cons frst rec)]))
                        empty loc)))


;;(swap-letters s) produces each variation of s where two
;;   adjacent letters are swapped
```

```
;; swap-letters: Word -> (listof Word)
;; Examples:
(check-expect (swap-letters "") empty)
(check-expect (swap-letters "a") (list "a"))
(check-expect (swap-letters "abc") (list "bac" "acb"))

(define (swap-letters s)
  (local
    [(define loc (string->list s))]
    (cond
      [(empty? loc) empty]
      [(empty? (rest loc)) (list s)]
      [else (build-list (sub1 (length loc))
                        (lambda (i) (swap-at-i i loc)))])))


(define accept-any (lambda (x) true))

;;(members? needles haystack) determines if ls of the needles appear in the haystack
;; members? (listof Any) (listof Any) -> Bool
;; Examples:
(check-expect (members? '(2 4 5) '(1 2 3 4 5)) true)
(check-expect (members? '(2 4 6) '(1 2 3 4 5)) false)

(define (members? needles haystack)
  (andmap (lambda (x) (member? x haystack)) needles))


;;(suggest s valid?) suggests words similar to s that are valid?
;; suggest: Word (Word -> Bool) -> (listof Word)
;; Examples:
(check-expect (member? "c" (suggest "" accept-any)) true)
(check-expect (suggest "abc" (lambda (x) false)) empty)
(check-expect (member? "azc" (suggest "abc" accept-any)) true)

(define (suggest s valid?)
  (local [(define words (append (remove-letters s)
                                (insert-letters s)
                                (trailing-letters s)
                                (replace-letters s)
                                (swap-letters s)))

          (define valid-words (filter valid? words))

          (define legal-words (filter (lambda (x) (and (not (string=? s x))
                                                       (not (string=? x ""))))
                                      valid-words))

          (define clean-words (remove-dups (sort legal-words string<=?)))]

    clean-words))

;; Tests:
(check-expect (members? '("bc" "ac" "ab") (suggest "abc" accept-any)) true)
(check-expect (members? '("aabc" "zabc" "aabc" "azbc" "abac" "abzc")
                        (suggest "abc" accept-any)) true)
(check-expect (members? '("abca" "abcz") (suggest "abc" accept-any)) true)
(check-expect (members? '("bbc" "zbc" "aac" "azc" "aba" "abz")
```

```
                        (suggest "abc" accept-any)) true)
(check-expect (members? '("bac" "acb") (suggest "abc" accept-any)) true)



;; ************************ Alternate Suggest Solution ************************


;; (suggest/alt s valid?) suggests words similar to s that are valid?
;; suggest/alt: Word (Word -> Bool) -> (listof Word)
;; Examples:
(check-expect (member? "c" (suggest/alt "" accept-any)) true)
(check-expect (suggest/alt "abc" (lambda (x) false)) empty)
(check-expect (member? "azc" (suggest/alt "abc" accept-any)) true)

(define (suggest/alt s valid?)
  (local [(define loc (string->list s))
          (define len (length loc))

          ;; (remove-at i lst) removes element with index i from lst
          ;; remove-at: Nat (listof Any) -> (listof Any)
          (define (remove-at i lst)
            (ifoldr (lambda (k x y) (cond [(= i k) y]
                                          [else (cons x y)]))
                    empty lst))

          ;; (insert-at i item lst) inserts item at index i in lst
          ;; insert-at: Nat Any (listof Any) -> (listof Any)
          (define (insert-at i item lst)
            (ifoldr (lambda (k x y) (cond [(= i k) (cons item (cons x y))]
                                          [else (cons x y)]))
                    empty lst))

          ;; (replace-at i item lst) replaces index i of lst with item
          ;; replace-at Nat Any (listof Any) -> (listof Any)
          (define (replace-at i item lst)
            (ifoldr (lambda (k x y) (cond [(= i k) (cons item y)]
                                          [else (cons x y)]))
                    empty lst))

          ;; (swap-at i lst) swaps the items at i and i + 1 in list
          ;; swap-at: Nat (listof Any) -> (listof Any)
          (define (swap-at i lst)
            (ifoldr (lambda (k x y) (cond [(and (= i k) (not (empty? y)))
                                           (cons (first y) (cons x (rest y)))]
                                          [else (cons x y)]))
                    empty lst))

          ;; (every-pos f) applies (f i loc) for every position i in loc
          ;; every-pos: (Nat (listof Char) -> (listof Char)) -> (listof Str)
          (define (every-pos f)
            (map list->string (build-list len (lambda (i) (f i loc)))))

          ;; (every-pos-char f) applies (f i c loc) for every position i
          ;;   in loc and every c in letters
          ;; every-pos-char: (Nat Char (listof Char)) -> (listof Str)
          (define (every-pos-char f)
            (foldr append empty
```

```
                (map (lambda (c) (map list->string
                                      (build-list len (lambda (i) (f i c loc)))))
                     letters)))

          (define trailing
            (map list->string (map (lambda (c) (foldr cons (list c) loc)) letters)))

          (define words (append (every-pos remove-at)
                                (every-pos swap-at)
                                (every-pos-char insert-at)
                                (every-pos-char replace-at)
                                trailing))

          (define valid-words (filter valid? words))

          (define legal-words (filter (lambda (x) (and (not (string=? s x))
                                                       (not (string=? x ""))))
                                      valid-words))

          (define clean-words (remove-dups (sort legal-words string<=?)))]

    clean-words))

;; Tests:
(check-expect (members? '("bc" "ac" "ab") (suggest/alt "abc" accept-any)) true)
(check-expect (members? '("aabc" "zabc" "aabc" "azbc" "abac" "abzc")
                        (suggest/alt "abc" accept-any)) true)
(check-expect (members? '("abca" "abcz") (suggest/alt "abc" accept-any)) true)
(check-expect (members? '("bbc" "zbc" "aac" "azc" "aba" "abz")
                        (suggest/alt "abc" accept-any)) true)
(check-expect (members? '("bac" "acb") (suggest/alt "abc" accept-any)) true)
```