```
;; CS 135 :: Fall 2017 :: Posted solution :: A07 :: trie.rkt

(require "a07lib.rkt")


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

(define c-tnode
  (make-tnode #\c false
              (list
               (make-tnode #\o false
                           (list (make-tnode #\o true empty)
                                 (make-tnode #\w true empty)))
               (make-tnode #\s false
                           (list (make-tnode #\1 false
                                             (list (make-tnode #\1 false
                                                               (list (make-tnode #\5 true empty)
                                                                     (make-tnode #\6 true
empty)))
                                                   (make-tnode #\3 false
                                                               (list (make-tnode #\5 true empty)
                                                                     (make-tnode #\6 true
empty))))))))))

(define a-tnode
  (make-tnode #\a false
              (list (make-tnode #\t true empty))))

(define a-c-trie
  (make-trie (list a-tnode c-tnode)))


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

;; trie-template: Trie -> Any
;(define (trie-template dict)
;  (... (list-tnode-template (trie-children dict)) ...))

;; list-tnode-template: (listof TNode) -> Any
;(define (list-tnode-template lonodes)
;  (cond
;    [(empty? lonodes) ...]
;    [else (... (tnode-template (first lonodes))
;           ... (list-tnode-template (rest lonodes)) ...)]))

;; tnode-template: TNode -> Any
;(define (tnode-template node)
;  (... (tnode-key node)
;   ... (tnode-ends-word? node)
;   ... (list-tnode-template (tnode-children node)) ...))


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

;; (in-trie-list? loc lonodes) determines if loc represents a word in lonodes
;; in-trie-list?: (listof Char) (listof TNode) -> Bool
;; Examples:
(check-expect (in-trie-list? (string->list "coo") empty) false)
```

```
(check-expect (in-trie-list? (string->list "coo")
                                     (trie-children a-c-trie)) true)

(define (in-trie-list? loc lonodes)
  (cond
    [(empty? loc) false]
    [(empty? lonodes) false]
    [else (or (in-trie-node? loc (first lonodes))
              (in-trie-list? loc (rest lonodes)))]))

;; Test:
(check-expect (in-trie-list? (string->list "error")
                                     (trie-children a-c-trie)) false)


;; (in-trie-node? loc node) determines if loc represents a word in node
;; in-trie-node?: (listof Char) TNode -> Bool
;; requires: loc is non-empty
;; Examples:
(check-expect (in-trie-node? (string->list "c") c-tnode) false)
(check-expect (in-trie-node? (string->list "coo") c-tnode) true)

(define (in-trie-node? loc node)
  (cond
    [(empty? (rest loc))
     (and (tnode-ends-word? node)
          (char=? (first loc)
                  (tnode-key node)))]
    [else (and (char=? (first loc)
                       (tnode-key node))
               (in-trie-list? (rest loc) (tnode-children node)))]))

;; Tests:
(check-expect (in-trie-node? (string->list "c") a-tnode) false)
(check-expect (in-trie-node? (string->list "coo") a-tnode) false)


;; (in-trie? target-string dict) determines if target-string
;;   is represented in dict
;; in-trie?: Str Trie -> Bool
;; Examples:
(check-expect (in-trie? "" c-d-trie) false)
(check-expect (in-trie? "donut" c-d-trie) true)
(check-expect (in-trie? "donut" blank-trie) false)

(define (in-trie? target-string dict)
  (in-trie-list? (string->list target-string) (trie-children dict)))

;; Tests:
(check-expect (in-trie? "don" c-d-trie) false)
(check-expect (in-trie? "dont" c-d-trie) false)
(check-expect (in-trie? "dog" c-d-trie) true)


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


;; (list-words-list lonodes letters-so-far) produces all words found in
;;   lonodes after already seeing letters-so-far
;; list-words-list: (listof TNode) (listof Char) -> (listof Str)
;; Examples:
(check-expect (list-words-list empty empty) empty)
(check-expect (list-words-list (trie-children h-u-trie) empty)
              (list "ha" "hat" "he" "hot" "use"))

(define (list-words-list lonodes letters-so-far)
  (cond
    [(empty? lonodes) empty]
    [else (append (list-words-node (first lonodes) letters-so-far)
                  (list-words-list (rest lonodes) letters-so-far))]))

;; Test:
(check-expect (list-words-list (list a-tnode) (list #\c)) (list "cat"))


;; (list-words-node node letters-so-far) produces all words found in
;;   node after already seeing letters-so-far
;; list-words-node: TNode (listof Char) -> (listof Str)
;; Examples:
(check-expect (list-words-node a-tnode empty) (list "at"))
(check-expect (list-words-node c-tnode empty) (list "coo" "cow" "cs115" "cs116" "cs135"
"cs136"))

(define (list-words-node node letters-so-far)
  (cond
    [(tnode-ends-word? node)
     (cons (list->string (reverse (cons (tnode-key node) letters-so-far)))
           (list-words-list (tnode-children node) (cons (tnode-key node) letters-so-far)))]
    [else (list-words-list (tnode-children node) (cons (tnode-key node) letters-so-far))]))

;; Tests:
(check-expect (list-words-node a-tnode (list #\c)) (list "cat"))


;; (list-words dict) produces a lexicographically ordered list of words
;;   represented in dict
;; list-words: Trie -> (listof Str)
;; Examples:
(check-expect (list-words blank-trie) empty)
(check-expect (list-words h-u-trie) (list "ha" "hat" "he" "hot" "use"))

(define (list-words dict)
  (list-words-list (trie-children dict) empty))

;; Tests:
(check-expect (list-words toydict-trie)
              (list "actress" "adopt" "awful" "billion" "blurt" "cold"
                    "concrete" "cook" "deprive" "describe" "dirt"
                    "effectively" "fault" "five" "petite"
                    "publication" "significantly" "smugly" "swapped"
                    "ugly" "wake"))
(check-expect (list-words c-d-trie)
              (list "cat" "catch" "cater" "catnip" "cattle" "dig"
                    "dog" "dogfish" "donald" "donut" "doze"))
```

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

(define e-tnode (make-tnode #\e false empty))
(define r-tnode (make-tnode #\r false empty))
(define e-r-list (list e-tnode r-tnode))


;; (new-branch loc) constructs a new branch of a trie to contain the word
;;   represented by loc
;; new-branch: (listof Char) -> TNode
;; requires:  loc is non-empty
;; Examples:
(check-expect (new-branch (string->list "a")) (make-tnode #\a true empty))
(check-expect (new-branch (string->list "at")) a-tnode)

(define (new-branch loc)
  (cond
    [(empty? (rest loc)) (make-tnode (first loc) true empty)]
    [else (make-tnode (first loc) false (list (new-branch (rest loc))))]))

;; Test:
(check-expect (new-branch (string->list "cat")) (make-tnode #\c false (list a-tnode)))


;; (insert-word-list loc lonodes) produces the result of inserting all characters
;;   in loc into lonodes
;; insert-word-list: (listof Char) (listof TNode) -> Any
;; requires: loc is non-empty
;; Examples:
(check-expect (insert-word-list (string->list "a") empty) (list (make-tnode #\a true empty)))
(check-expect (insert-word-list (string->list "aa") e-r-list)
              (list
               (make-tnode #\a false
                           (list (make-tnode #\a true empty)))
               e-tnode r-tnode))

(define (insert-word-list loc lonodes)
  (cond
    [(empty? lonodes) (list (new-branch loc))]
    [(char=? (first loc) (tnode-key (first lonodes)))
     (cons (insert-word-node loc (first lonodes)) (rest lonodes))]
    [(char>=? (first loc) (tnode-key (first lonodes)))
     (cons (first lonodes) (insert-word-list loc (rest lonodes)))]
    [else (cons (new-branch loc) lonodes)]))

;; Tests:
(check-expect (insert-word-list (string->list "fa") e-r-list)
              (list
               e-tnode
               (make-tnode #\f false
                           (list (make-tnode #\a true empty)))
               r-tnode))
(check-expect (insert-word-list (string->list "ea") e-r-list)
              (list
               (make-tnode #\e false
                           (list (make-tnode #\a true empty)))
               r-tnode))
(check-expect (insert-word-list (string->list "za") e-r-list)
              (list
               e-tnode
               r-tnode
               (make-tnode #\z false
                           (list (make-tnode #\a true empty)))))


;; (insert-word-node loc node) produces the result of inserting all characters
;;   in loc into node
;; insert-word-node: (listof Char) TNode -> TNode
;; requires: loc is non-empty
;;           (first loc) = (tnode-key node)
;; Examples:
(check-expect (insert-word-node (string->list "a") a-tnode)
              (make-tnode #\a true (list (make-tnode #\t true empty))))
(check-expect (insert-word-node (string->list "am") a-tnode)
              (make-tnode #\a false (list (make-tnode #\m true empty)
                                          (make-tnode #\t true empty))))

(define (insert-word-node loc node)
  (cond
    [(empty? (rest loc)) (make-tnode (first loc) true (tnode-children node))]
    [else (make-tnode (tnode-key node) (tnode-ends-word? node)
                      (insert-word-list (rest loc) (tnode-children node)))]))


;; (insert-word word dict) produces the result of insert target-string
;;   into dict
;; insert-word: Str Trie -> Trie
;; requires: word is not the empty string
;; Examples:
(check-expect (list-words (insert-word "cat" blank-trie))
              (list "cat"))
(check-expect (list-words (insert-word "cat" h-u-trie))
              (list "cat" "ha" "hat" "he" "hot" "use"))

(define (insert-word word dict)
  (make-trie (insert-word-list (string->list word) (trie-children dict))))

;; Tests:
(check-expect (list-words (insert-word "ho" h-u-trie))
              (list "ha" "hat" "he" "ho" "hot" "use"))
(check-expect (list-words (insert-word "him" h-u-trie))
              (list "ha" "hat" "he" "him" "hot" "use"))
(check-expect (list-words (insert-word "zzz" h-u-trie))
              (list "ha" "hat" "he" "hot" "use" "zzz"))
(check-expect (list-words (insert-word "useless" h-u-trie))
              (list "ha" "hat" "he" "hot" "use" "useless"))


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

;; (insert-some-words word-list dict) produces the result of inserting
;;   each word in word-list into dict
;; insert-some-words: (listof Str) Trie -> Trie
;; requires: none of the strings in word-list are the empty string
;; Examples:
(check-expect (insert-some-words empty a-c-trie) a-c-trie)
```

```
(check-expect (list-words (insert-some-words (list "cow" "cod" "corinthean")
                                   blank-trie))
              (list "cod" "corinthean" "cow"))

(define (insert-some-words word-list dict)
  (cond
    [(empty? word-list) dict]
    [else (insert-some-words (rest word-list)
                             (insert-word (first word-list) dict))]))

;; Tests:
(check-expect (list-words (insert-some-words (list "cow" "cod" "corinthean")
                                             a-c-trie))
              (list "at" "cod" "coo" "corinthean" "cow" "cs115" "cs116"
                    "cs135" "cs136"))
(check-expect (list-words (insert-some-words (list "cod") a-c-trie))
                          (list "at" "cod" "coo" "cow" "cs115" "cs116"
                                "cs135" "cs136"))
(check-expect (list-words (insert-some-words (list "cow" "cow" "cow") a-c-trie))
                          (list-words a-c-trie))


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

;; (list-completions-list loc lonodes letters-so-far) produces every word
;;   in lonodes that starts with loc after seeing letters-so-far
;; list-completions-list: (listof Char) (listof TNode) (listof Char) -> (listof Str)
;; requires: loc is non-empty
;; Examples:
(check-expect (list-completions-list (string->list "a") (list a-tnode) empty) (list "at"))
(check-expect (list-completions-list (string->list "cs") (list a-tnode c-tnode) empty)
              (list "cs115" "cs116" "cs135" "cs136"))

(define (list-completions-list loc lonodes letters-so-far)
  (cond
    [(empty? lonodes) empty]
    [(char=? (first loc) (tnode-key (first lonodes)))
     (list-completions-node loc (first lonodes) letters-so-far)]
    [else (list-completions-list loc (rest lonodes) letters-so-far)]))


;; (list-completions-node loc node letters-so-far) produces every word
;;   in node that start with loc after seeing letters-so-far
;; list-completions-node: (listof Char) TNode (listof Char) -> (listof Str)
;; requires: loc is non-empty
;;           (first loc) = (tnode-key node)
;; Examples:
(check-expect (list-completions-node (string->list "a") a-tnode empty) (list "at"))
(check-expect (list-completions-node (string->list "cs") c-tnode empty)
              (list "cs115" "cs116" "cs135" "cs136"))

(define (list-completions-node loc node letters-so-far)
  (cond
    [(empty? (rest loc)) (list-words-node node letters-so-far)]
    [else (list-completions-list (rest loc)
                                 (tnode-children node)
                                 (cons (first loc) letters-so-far))]))
```

```
;; (list-completions prefix dict) produces a lexicographically sorted list
;;   of all words in dict that start with prefix
;; list-completions: Str Trie -> (listof Str)
;; Examples:
(check-expect (list-completions "empty" blank-trie) empty)
(check-expect (list-completions "" c-d-trie) (list-words c-d-trie))
(check-expect (list-completions "don" c-d-trie)
              (list "donald" "donut"))

(define (list-completions prefix dict)
  (cond
    [(empty? (string->list prefix)) (list-words dict)]
    [else (list-completions-list (string->list prefix) (trie-children dict) empty)]))

;; Tests:
(check-expect (list-completions "do" c-d-trie)
              (list "dog" "dogfish" "donald" "donut" "doze"))
(check-expect (list-completions "bl" c-d-trie) empty)
(check-expect (list-completions "cat" c-d-trie)
              (list "cat" "catch" "cater" "catnip" "cattle"))
(check-expect (list-completions "cattle" c-d-trie)
              (list "cattle"))
(check-expect (list-completions "catches" c-d-trie) empty)



;; CS 135 :: Fall 2017 :: Posted solution :: A07 :: moretabular.rkt


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

;; (mirror table) reverses each row of table
;; mirror: Table -> Table
;; Examples:
(check-expect (mirror empty) empty)
(check-expect (mirror (list (list 1 2 3 4 5))) (list (list 5 4 3 2 1)))

(define (mirror table)
  (local
    [;; (reverse-row/acc r result) produces the reverse of row r prepended to result
     ;; reverse-row/acc: (listof Num) (listof Num) -> (listof Num)
     (define (reverse-row/acc r result)
       (cond
         [(empty? r) result]
         [else (reverse-row/acc (rest r) (cons (first r) result))]))]

    (cond
      [(empty? table) empty]
      [else (cons (reverse-row/acc (first table) empty)
                  (mirror (rest table)))])))

;; Tests:
(check-expect (mirror (list (list empty))) (list (list empty)))
(check-expect (mirror (list (list 3))) (list (list 3)))
(check-expect (mirror (list (list 1 2 3 4 5) (list 6 7 8 9 10)))
              (list (list 5 4 3 2 1) (list 10 9 8 7 6)))
```

```
;; ------ Q3b ------

;; A Table-Element-Function is (anyof (Num -> Num) (Num -> Int) (Num -> Nat))


;; (element-apply-many function-list base-table) produces a list of tables where
;;    the ith table consists of applying the ith function in function-list
;;    to each element of base-table
;; element-apply-many: (listof Table-Element-Function) Table -> (listof Table)
;; Examples:
(check-expect (element-apply-many empty empty) empty)
(check-expect (element-apply-many (list abs floor)
                                  '((7 4.5 -3.2)(-3 3 13)))
              (list '((7 4.5 3.2)(3 3 13)) '((7 4 -4) (-3 3 13))))

(define (element-apply-many function-list base-table)
  (local
    [;; (apply-function/row func row) produces a row that is the result of
     ;;   applying func to every element of row
     ;; apply-function/row: Table-Element-Function (listof Num) -> (listof Num)
     (define (apply-function/row func row)
       (cond
         [(empty? row) empty]
         [else (cons (func (first row))
                     (apply-function/row func (rest row)))]))

     ;; (apply-function/table func base-table) produces a table that is the result of
     ;;   applying func to each element in base-table
     ;; apply-function/table: Table-Element-Function Table -> Table
     (define (apply-function/table func t)
       (cond
         [(empty? t) empty]
         [else (cons (apply-function/row func (first t))
                     (apply-function/table func (rest t)))]))]

    (cond
      [(empty? function-list) empty]
      [else (cons (apply-function/table (first function-list) base-table)
                  (element-apply-many (rest function-list) base-table))])))

;; Tests:
(check-expect (element-apply-many empty '((7 4.5 -3.2)(-3 3 13))) empty)
(check-expect (element-apply-many (list abs floor) empty) (list empty empty))


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

;; (apply-function f arg) produces the result of f with the given argument arg.
;; apply-function: (X -> Y) X -> Y
;; Example:
(check-expect (apply-function abs -3) 3)

(define (apply-function f arg)
  (f arg))


;; (scale-smallest table offset) produces a function that multiplies
;;    the smallest element of table by its input, and adds the offset
;; scale-smallest: Table Num -> (Num -> Num)
;; requires: table has at least one element
;; Examples:
(check-expect (apply-function (scale-smallest '((-3 18 9 -10 -1 17 4 32.39)
                                                (-1 1.3 -10 72 5 6 7 -10)) 0) 1) -10)

(define (scale-smallest table offset)
  (local
    [;; (min-given-row row smallest-so-far) produces the smallest element
     ;;   in row, or smallest-so-far if that is smaller
     ;; min-given-row: (listof Num) Num -> Num
     (define (min-given-row row smallest-so-far)
       (cond
         [(empty? row) smallest-so-far]
         [(< (first row) smallest-so-far)
          (min-given-row (rest row) (first row))]
         [else (min-given-row (rest row) smallest-so-far)]))

     ;; (min-table remaining-table smallest-so-far) produces the smallest
     ;;   element in remaining-table, or smallest-so-far if that is smaller
     ;; min-table: Table Num -> Num
     (define (min-table remaining-table smallest-so-far)
       (cond
         [(empty? remaining-table) smallest-so-far]
         [else (min-table (rest remaining-table)
                          (min-given-row (first remaining-table) smallest-so-far))]))

     (define table-multiplier (min-table table (first (first table))))

     ;; (scale-function x) multiplies the smallest element of the table by
     ;;   x and adds the offset
     ;; scale-function: Num -> Num
     (define (scale-function x)
       (+ (* table-multiplier x) offset))]

    scale-function))

;; Tests:
(check-expect (apply-function (scale-smallest '((10)) 2) 3) 32)
(check-expect (apply-function (scale-smallest '((-3 18 9 0.32 -10 17 4 32.39)
                                                (12 1.3 9 72 5 6 7 8)) -0.5) -5) 49.5)
(check-expect (apply-function (scale-smallest '((-10 18 9 0.32  -3 17 4 32.39)
                                                (12 1.3 9 72 5 6 7 8)) -0.5) -5) 49.5)
(check-expect (apply-function (scale-smallest '((-3 18 9 0.32 17 4 32.39 -10)
                                                (12 1.3 9 72 5 6 7 8)) -0.5) -5) 49.5)
(check-expect (apply-function (scale-smallest '((-3 18 9 0.32 -10 17 4 32.39)
                                                (12 1.3 9 72 5 6 7 8)) -0.5) -5) 49.5)
(check-expect (apply-function (scale-smallest '((-3 18 9 0.32 -1 17 4 32.39)
                                                (-10 1.3 9 72 5 6 7 8)) -0.5) -5) 49.5)
(check-expect (apply-function (scale-smallest '((-3 18 9 0.32 -1 17 4 32.39)
                                                (-1 1.3 9 72 5 6 7 -10)) -0.5) -5) 49.5)
(check-expect (apply-function (scale-smallest '((-3 18 9 0.32 -1 17 4 32.39)
                                                (-1 1.3 9 72 -10 6 7 -7)) -0.5) -5) 49.5)
(check-expect (apply-function (scale-smallest '((-3 18 9 -10 -1 17 4 32.39)
                                                (-1 1.3 -10 72 5 6 7 -10)) -0.5) -5) 49.5)
```