```
;;
;; ***************************************************
;; Sample Solutions
;; CS 135 Fall 2019
;; Assignment 08, Problem 2
;; ***************************************************
;;


;;(unzip pairs) produces a pair of all the first elements
;;  of pairs, and all the second elements of pairs
;; unzip: (listof (list Any Any)) -> (list (listof Any) (listof Any))
;; Examples:
(check-expect (unzip empty) (list empty empty))
(check-expect (unzip '((a 1) (b 2) (c 3))) '((a b c) (1 2 3)))

(define (unzip pairs)
  (cond
    [(empty? pairs) (list empty empty)]
    [else (local
            [(define rec (unzip (rest pairs)))]
            (list (cons (first (first pairs)) (first rec))
                  (cons (second (first pairs)) (second rec))))]))

;; Test:
(check-expect (unzip '((a 1))) '((a) (1)))



;;
;; ***************************************************
;; Sample Solutions
;; CS 135 Fall 2019
;; Assignment 08, Problem 3
;; ***************************************************
;;


;; A Filesystem Object (FSObject) is one of:
;; * File
;; * Dir

(define-struct file (name size owner))
;; A File is a (make-file Str Nat Sym)

(define-struct dir (name owner contents))
```

```
;; A Dir is a (make-dir Str Sym (listof FSObject))


(define example-fs
  (make-dir "root" 'root
            (list
             (make-dir "Dan" 'dan
                       (list (make-file "log.txt" 768 'dan)
                             (make-file "profile.jpg" 60370 'dan)
                             (make-dir "music" 'dan
                                       (list
                                        (make-file "Thelonius Monk.mp3"
92227584 'dan)))))
             (make-dir "Slides" 'teaching
                       (list (make-dir "cs135" 'teaching
                                       (list (make-file "01-intro.pdf"
72244 'teaching)
                                             (make-file "11-trees.pdf"
123124 'teaching)
                                             (make-dir "system" 'root
                                                       (list (make-dir
"logs" 'teaching

empty)))))))
             (make-file "vmlinuz" 30 'root))))


;; 3a)

;; fsobject-template: FSObject -> Any
(define (fsobject-template fso)
  (cond
    [(file? fso) (... (file-name fso) ...
                      (file-size fso) ...
                      (file-owner fso) ...)]
    [(dir? fso) (... (dir-name fso) ...
                     (dir-owner fso) ...
                     (listof-fsobject-template (dir-contents fso)) ...)]))

;; listof-fsobject-template: (listof FSObject) -> Any
(define (listof-fsobject-template lofso)
  (cond
    [(empty? lofso) ...]
    [else (... (fsobject-template (first lofso)) ...
               (listof-fsobject-template (rest lofso)) ...)]))
```

```
;; 3b)

;;(fsobject-name fso) produces the name of fso
;; fsobject-name: FSObject -> Str
;; Examples:
(check-expect (fsobject-name (make-file "a" 1 'a)) "a")
(check-expect (fsobject-name (make-dir "b" 'b empty)) "b")

(define (fsobject-name fso)
  (cond [(file? fso) (file-name fso)]
        [else (dir-name fso)]))

;; Test:
(check-expect (fsobject-name example-fs) "root")


;; 3c)

;;(count-files fs) produces the number of files in fs
;; count-files: FSObject -> Nat
;; Examples:
(check-expect (count-files (make-file "a" 1 'a)) 1)
(check-expect (count-files (make-dir "b" 'b empty)) 0)
(check-expect (count-files (make-dir "c" 'c (list
                                             (make-file "d" 2 'd)
                                             (make-file "e" 3 'e)
                                             (make-dir "f" 'f empty)))) 2)

(define (count-files fso)
  (local [;;(count-files/lst lofs) produces the number of files in lofs
          ;; count-files: (listof FSObject) -> Nat
          (define (count-files/list  lofso)
            (cond [(empty? lofso) 0]
                  [else (+ (count-files (first lofso))
                           (count-files/list (rest lofso)))]))]
    (cond [(file? fso) 1]
          [else (count-files/list (dir-contents fso))])))

;; Test:
(check-expect (count-files example-fs) 6)


;; 3d)
```

```
;;(file-exists? fso path) determines if path leads to a file in fso
;; file-exists?: FSObject (listof Str) -> Bool
;; Examples:
(check-expect (file-exists? (make-file "a" 1 'a) empty) false)
(check-expect (file-exists? (make-file "a" 1 'a) (list "a")) true)
(check-expect (file-exists? example-fs (list "root" "Slides" "cs135"
"11-trees.pdf")) true)

(define (file-exists? fso path)
  (cond
    [(empty? path) false]
    [(not (string=? (first path) (fsobject-name fso))) false]
    [(file? fso) (empty? (rest path))]
    [else (local
            [;;(file-exists?/lst lofso path) determines if path leads to
any file in lofso
             ;; file-exists?/lst: (listof FSObject) (listof Str) -> Bool
             (define (file-exists?/lst lofso path)
               (cond
                 [(empty? lofso) false]
                 [else (or (file-exists? (first lofso) path)
                           (file-exists?/lst (rest lofso) path))]))]
            (file-exists?/lst (dir-contents fso) (rest path)))]))

;; Tests:
(check-expect (file-exists? example-fs (list "root" "Dan")) false)
(check-expect (file-exists? example-fs (list "test.txt")) false)
(check-expect (file-exists? example-fs (list "boot" "Slides" "cs135"
"11-trees.pdf")) false)


;; 3e)

;;(remove-empty fso) removes all initially empty directories from fso
;; remove-empty: FSObject -> FSObject
;; Examples:
(check-expect (remove-empty (make-file "a" 1 'a)) (make-file "a" 1 'a))
(check-expect (remove-empty (make-dir "a" 'a empty)) (make-dir "a" 'a
empty))
(check-expect (remove-empty (make-dir "a" 'a
                                      (list (make-file "b" 2 'b)
                                            (make-dir "c" 'c empty)
                                            (make-dir "d" 'd (list
(make-file "e" 3 'e))))))
              (make-dir "a" 'a (list (make-file "b" 2 'b)
                                     (make-dir "d" 'd (list (make-file "e"
```

```
3 'e))))))

(define (remove-empty fso)
  (local [;;(remove-empty/list lofso) removes all initially empty
directories from lofso
          ;; remove-empty/list: (listof FSObject) -> (listof FSObject)
          (define (remove-empty/list lofso)
            (cond [(empty? lofso) empty]
                  [(file? (first lofso))
                   (cons (first lofso) (remove-empty/list (rest lofso)))]
                  [(empty? (dir-contents (first lofso)))
                   (remove-empty/list (rest lofso))]
                  [else
                   (cons (remove-empty (first lofso))
                         (remove-empty/list (rest lofso)))]))]
    (cond [(file? fso) fso]
          [else (make-dir (dir-name fso)
                          (dir-owner fso)
                          (remove-empty/list (dir-contents fso)))])))

;; Tests:
(check-expect (remove-empty (make-dir "a" 'a (list (make-dir "b" 'b
empty))))
              (make-dir "a" 'a empty))
(check-expect (remove-empty (make-dir "a" 'a (list (make-file "b" 0 'b))))
              (make-dir "a" 'a (list (make-file "b" 0 'b))))


;; 3f)

;;(disk-hog fos) produces the owner of the largest file in fso
;; disk-hog: FSObject -> (anyof Sym False)
;; Examples:
(check-expect (disk-hog (make-file "a" 1 'a)) 'a)
(check-expect (disk-hog (make-dir "a" 'a empty)) false)

(define (disk-hog fso)
  (local [;;(disk-hog/acc biggest-file fso) determines the largest file in
          ;;  fso given the biggest-file found so far
          ;; disk-hog/acc: (anyof File false) FSObject -> (anyof File
false)
          (define (disk-hog/acc biggest-file fso)
            (cond [(and (file? fso)
                        (or (false? biggest-file)
                            (> (file-size fso)
                               (file-size biggest-file)))) fso]
```

```
                [(file? fso) biggest-file]
                [else (disk-hog/list biggest-file (dir-contents fso))]))

          ;;(disk-hog/lst biggest-file lofso) determines the largest file
in
          ;;  lofos given the biggest-file found so far
          ;; disk-hog/acc/lst: (anyof File false) (listof FSObject) ->
(anyof File false)
          (define (disk-hog/list biggest-file lofso)
            (cond [(empty? lofso) biggest-file]
                  [else (disk-hog/list (disk-hog/acc biggest-file (first
lofso))
                                       (rest lofso))]))

          (define answer (disk-hog/acc false fso))]
    (cond [(file? answer) (file-owner answer)]
          [else answer])))

;; Tests:
(check-expect (disk-hog (make-dir "a" 'a (list (make-file "b" 1 'b)
                                               (make-file "c" 2 'c)))) 'c)
(check-expect (disk-hog (make-dir "a" 'a (list (make-file "b" 2 'b)
                                               (make-file "c" 1 'c)))) 'b)
(check-expect (disk-hog example-fs) 'dan)
(check-expect (disk-hog (make-dir "a" 'a (list (make-dir "b" 'b empty)
                                               (make-dir "c" 'c empty)
                                               (make-dir "d" 'd empty))))
false)


;; 3g)

;;(owned-by fso owner) produces a list of paths to all fsobjects
;;  in fso owned by owner
;; owned-by: FSObject Sym -> (listof (listof Str))
;; Examples:
(check-expect (owned-by (make-file "a" 1 'a) 'a) (list (list "a")))
(check-expect (owned-by example-fs 'root)
              (list (list "root")
                    (list "root" "Slides" "cs135" "system")
                    (list "root" "vmlinuz")))

(define (owned-by fso owner)
  (local [;;(owned-by/path path cur-fso) produces a list of paths to all
fsobjects in
          ;;  fso owned by owner given the path to the cur-fso
```

```
          ;; owned-by/path: (listof Str) FSObject -> (listof (listof Str))
          (define (owned-by/path path cur-fso)
            (cond [(and (file? cur-fso) (symbol=? owner (file-owner
cur-fso)))
                   (list (append path (list (file-name cur-fso))))]
                  [(file? cur-fso) empty]
                  [(symbol=? owner (dir-owner cur-fso))
                   (cons (append path (list (dir-name cur-fso)))
                         (owned-by/dir (append path (list (dir-name
cur-fso)))
                                       (dir-contents cur-fso)))]
                  [else (owned-by/dir (append path (list (dir-name
cur-fso)))
                                       (dir-contents cur-fso))]))

          ;;(owned-by/dir path lofso) produces a list of paths to all
fsobjects in
          ;;  fso owned by owner given the path to lofso
          ;; owned-by/dir: (listof Str) (listof FSObject) -> (listof
(listof Str))
          (define (owned-by/dir path lofso)
            (cond [(empty? lofso) empty]
                  [else (append (owned-by/path path (first lofso))
                                (owned-by/dir path (rest lofso)))]))]
    (owned-by/path empty fso)))

;; Tests:
(check-expect (owned-by (make-dir "a" 'a empty) 'a) (list (list "a")))
(check-expect (owned-by (make-dir "a" 'a empty) 'b) empty)
(check-expect (owned-by (make-file "a" 1 'a) 'b) empty)
(check-expect (owned-by (make-dir "a" 'a
                                  (list (make-dir "b" 'b empty)
                                        (make-dir "c" 'c (list (make-file
"C" 2 'c)))
                                        (make-file "cc" 2 'c))) 'c)
              (list (list "a" "c")
                    (list "a" "c" "C")
                    (list "a" "cc")))



;;
;; *************************************************
;; Sample Solutions
;; CS 135 Fall 2019
;; Assignment 08, Problem 4
```

```
;; ***************************************************
;;


;; An HTML-Item (HI) is one of
;; * Str
;; * Tag

;; A Tag is (cons Sym (listof HI))


(define just-text "Hello, world!")
(define short-example '(p (h1 "Heading") "Text"))
(define html-example '(html (head (title "CS135"))
                            (body (h1 "Welcome")
                                  "More text...")))


;; 4a)

;;(html->string hi) produces the html text generated from hi
;; html->string: HI -> Str
;; Examples:
(check-expect (html->string "text") "text")
(check-expect (html->string short-example)
              "<p><h1>Heading</h1>Text</p>")

;; 4a)

(define (html->string hi)
  (local [;;(html->string/lst lohi) produces the text generated from all
html items in lohi
          ;; html->string/lst: (listof HI) -> Str
          (define (html->string/list lohi)
            (cond [(empty? lohi) ""]
                  [else (string-append
                         (html->string (first lohi))
                         (html->string/list (rest lohi)))]))]
    (cond [(string? hi) hi]
          [else (string-append
                 "<" (symbol->string (first hi)) ">"
                 (html->string/list (rest hi))
                 "</" (symbol->string (first hi)) ">")])))

;; Tests:
(check-expect (html->string "Hello") "Hello")
```

```
(check-expect (html->string '(p "Hello")) "<p>Hello</p>")
(check-expect (html->string '(hr)) "<hr></hr>")
(check-expect (html->string html-example)
"<html><head><title>CS135</title></head><body><h1>Welcome</h1>More
text...</body></html>")


;; 4b)


;;(remove-tag hi tag) removes all occurrences of tag from hi
;; remove-tag: HI Sym -> (anyof HI (listof HI))
;; Examples:
(check-expect (remove-tag "Text" 'p) "Text")
(check-expect (remove-tag '(p "Text") 'li) '(p "Text"))
(check-expect (remove-tag '(p (b (b "Text") "More")) 'b) '(p "Text"
"More"))

(define (remove-tag hi tag)
  (local
    [;;(remove-tag/lst lohi) removes all occurrences of tag from lohi
     ;; remove-tag/lst: (listof HI) -> (listof HI)
     (define (remove-tag/lst lohi)
       (cond
         [(empty? lohi) empty]
         [(string? (first lohi)) (cons (first lohi)
                                       (remove-tag/lst (rest lohi)))]
         [(symbol=? tag (first (first lohi)))
          (append (remove-tag (first lohi) tag)
                  (remove-tag/lst (rest lohi)))]
         [else (cons (remove-tag (first lohi) tag)
                     (remove-tag/lst (rest lohi)))]))]
    (cond
      [(string? hi) hi]
      [(symbol=? tag (first hi)) (remove-tag/lst (rest hi))]
      [else (cons (first hi) (remove-tag/lst (rest hi)))])))

(check-expect (remove-tag "Hello" 'p) "Hello")
(check-expect (remove-tag '(p "Hello, " (b "World") "!") 'p)
              '("Hello, " (b "World") "!"))
(check-expect (remove-tag '(p "Hello, " (b "World") "!") 'b)
              '(p "Hello, " "World" "!"))
(check-expect (remove-tag '(p (hr) "foo") 'hr)
              '(p "foo"))
(check-expect (remove-tag html-example 'head)
              '(html (title "CS135")
```

```
                    (body (h1 "Welcome")
                          "More text..."))


;; 4c)

;;(bad-tags? hi) determines if there are any tag violations in hi
;; bad-tags?: HI -> Bool
;; Examples:
(check-expect (bad-tags? "Text") false)
(check-expect (bad-tags? '(ul (li "Text"))) false)
(check-expect (bad-tags? '(p (li "Text"))) true)

(define (bad-tags? hi)
  (local [;;(bad-tags?/acc in-list? hi) determines if there are any tag
violations
          ;;  in hi depending on if hi is in-list?
          ;; bad-tags?/acc: Bool HI -> Bool
          (define (bad-tags?/acc in-list? hi)
            (cond [(string? hi) false]
                  [(symbol=? 'hr (first hi))
                   (cons? (rest hi))]
                  [(and (not in-list?) (symbol=? 'li (first hi))) true]
                  [(or (symbol=? 'ol (first hi))
                       (symbol=? 'ul (first hi)))
                   (bad-tags?/list true (rest hi))]
                  [else (bad-tags?/list false (rest hi))]))

          ;;(bad-tags?/list in-list? lohi) determines if there are any tag
violations
          ;;  in lohi depending on if lohi is in-list?
          ;; bad-tags?/list: Bool (listof HI) -> Bool
          (define (bad-tags?/list in-list? lohi)
            (and (cons? lohi)
                 (or
                  (bad-tags?/acc in-list? (first lohi))
                  (bad-tags?/list in-list? (rest lohi)))))]
    (bad-tags?/acc false hi)))

;; Tests:
(check-expect (bad-tags? '(hr)) false)
(check-expect (bad-tags? '(hr "Text")) true)
(check-expect (bad-tags? '(li)) true)
(check-expect (bad-tags? '(ol (li "Text"))) false)
(check-expect (bad-tags? '(ul (p (li "Text")))) true)
(check-expect (bad-tags? '(p (ul (ol (li "Text"))))) false)
```

```
(check-expect (bad-tags? '(p (hr))) false)
(check-expect (bad-tags? '(p (hr (p)))) true)
(check-expect (bad-tags? '(p "Text")) false)
```