CS 135 Fall 2019
Becker, Clarke, Giesbrecht, Holtby, Lanctot, Reetz

Assignment 8

Due:	Tuesday, November 19th, 9:00 pm
Language level:	Intermediate Student
Files to submit:	`unzip.rkt`, `filesystem.rkt`, `html.rkt`
Warmup exercises:	HtDP *Without using explicit recursion:* 9.5.2, 9.5.4
Practice exercises:	HtDP 20.2.4, 24.3.1, 24.3.2

Notes:

- **Make sure you read the OFFICIAL A08 post on Piazza** for the answers to frequently asked questions.
- **All helper functions must be encapsluated using `local`.**
- Unless stated otherwise, all policies from Assignment 07 carry forward.
- This assignment covers material up to the end of module 12.
- You may use any list or string function covered in class, as well as any mathematical function or type-checking predicate.
- Remember that basic tests are meant as sanity checks only; by design, passing them should not be taken as any indication that your code is correct, only that it has the right form.
- Unless the question specifically says otherwise, you are always permitted to write helper functions (however, note that they must be **local**, as stated above). You may use any constants or functions from any part of a question in any other part.
- Solutions will be marked for both correctness **[80%]** and style **[20%]**. Follow the guidelines in the Style Guide, pages 1–15.

Here are the assignment questions you need to submit.

1. (10%) The stepping problems for Module 12 at `https://www.student.cs.uwaterloo.ca/~cs135/assign/stepping/`.

2. (10%) Write the function `unzip` that consumes a list of pairs (two-element lists), and produces a list of two lists. The first list contains the first element from each pair, and the second list contains the second element from each pair, in the original order. The pairs can contain any type.

   Examples:

   ```
   (check-expect (unzip '((1 "a") (2 b) ("3" a))) '((1 2 "3") ("a" b a)))
   (check-expect (unzip empty) '(()()))
   ```

Place your solutions in `unzip.rkt`.

3. (40%)

   A general tree (a tree where internal nodes can have any number of children) can be used to represent the file system used by a computer's operating system to keep track of files.

   A `FSObject` is a tree. `File`s are the leaves, and `Dir`s are the internal nodes (Dir stands for "directory", also known as a "folder").

   Data definitions:

```
;; 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))
```

   Example tree:

```
(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))))
```

   Similar to how we represented a path in a binary tree as a `(listof (anyof 'left 'right))`, we can represent a path in our file system as a `(listof Str)`, where each `Str` is the name of an `FSObject`. Since `File`s are leaves, only the last `Str` in a path can be a `file-name` (though it is also possible that all `Str` in the path are `dir-name`s).

   For example, the path to the "11-trees.pdf" file in the sample tree would be:

```
(list "root" "Slides" "cs135" "11-trees.pdf")
```

   Note: In addition to the starter file, the course webpage also contains `filesystem-lib.rkt`, a program written in full Racket that can display `FSObject` trees in a prettier way than the default constructor format.

To use the function you need to place this file in the same folder as your `filesystem.rkt` file, and include the line `(require "filesystem-lib.rkt")` at the top of your file. Then, you can use the function `display-fsobject`.

```
;; (display-fsobject fso) prints fso in a pretty manner
;; display-fsobject: FSObject -> None
;; side-effect: string is printed to the screen
```

You must not use this function in your code, including your tests and examples! Only use the function in the interaction window.

(a) Write two templates, one for functions that consume an `FSObject`, and one for functions that consume a `(listof FSObject)`

Call the templates `fsobject-template` and `listof-fsobject-template` respectively.

(b) Write the function `fsobject-name` that consumes an `FSObject` and produces its name.

```
(check-expect (fsobject-name (make-file "hello.txt" 32 'Dan))
              "hello.txt")
(check-expect (fsobject-name (make-dir "My Music" 'nobody empty))
              "My Music")
```

(c) Write the function `count-files` that consumes an `FSObject` and produces the total number of `File`s in the `FSObject`.

Examples:

```
(check-expect (count-files (make-file "README" 16 'me)) 1)
(check-expect (count-files example-fs)  6)
```

(d) Write the function `file-exists?` that consumes an `FSObject` and a `(listof Str)`, and produces true if the list of strings represents a path to a `File` in the tree.

```
(check-expect
  (file-exists? example-fs
                (list "root" "Slides" "cs135" "11-trees.pdf"))
  true)
;; This next example is false because it's a path to a Dir
(check-expect (file-exists? example-fs (list "root" "Dan")) false)
;; And this because it's not a valid path
(check-expect (file-exists? example-fs (list "readme.txt")) false)
```

(e) Write the function `remove-empty` that consumes an `FSObject`, and produces a new tree with all empty directories removed. (A directory `d` is empty if and only if `(dir-contents d)` is empty). Note that you should not cascade. That is, only remove directories that are empty in the original tree. Do not remove directories that are only empty after removing empty directories.

For example, in `example-fs` the "logs" directory would be removed, leaving the "system" directory empty. The "system" directory would not be removed, however, since it

was not originally empty.

Additionally, if the root of the tree is an empty `Dir`, it should not be removed (in fact, it cannot be removed since there's no such thing as an empty `FSObject`).

(f) Write the function `disk-hog` that consumes an `FSObject`, and produces the `Sym` that owns the largest `File`. If the tree contains no files, the function produces `false`. If there are multiple files with the same size, you may break the tie in any way that you want.

```
(check-expect (disk-hog example-fs) 'dan)
(check-expect (disk-hog (make-dir "secrets" 'cia empty)) false)
```

(g) Write the function `owned-by` that consumes an `FSObject` and a `Sym` and produces a list of paths to all `FSObject` owned by that user. The list of paths must be in the same order as the `FSObject`s are in the original tree.

```
(check-expect (owned-by example-fs 'root)
              (list (list "root")
                    (list "root" "Slides" "cs135" "system")
                    (list "root" "vmlinuz")))
(check-expect (owned-by example-fs 'cia) empty)
```

(h)

*bonus*5%

Write the function `remove-empty/cascade` that consumes an `FSObject`, and produces a new tree with all empty directories removed. Unlike the previous `remove-empty`, you must cascade. For example, in `example-fs` "logs" will be removed. Since that was the only content in "system", "system" is now empty and should also be removed. Additionally, the function *can* remove the root directory, in which case it returns `false`.

```
(check-expect (remove-empty/cascade (make-dir "root" 'root empty))
   false)
(check-expect
  (remove-empty/cascade
    (make-dir "root" 'root
              (list (make-dir "a" 'root
                              (list (make-dir "b" 'root empty)))
                                    (make-file "c" 120 'root))))
    (make-dir "root" 'root (list (make-file "c" 120 'root))))
```

You must not visit any nodes more than once (e.g. you cannot just call `remove-empty` until no more changes occur).

Place your solutions in `filesystem.rkt`.

4. (20%)

As the slides claim, a web page can be represented as a general tree using nested lists.

Pages are written in HTML, which is text with annotated with *tags*.

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

;; A Tag is (cons Sym (listof HI))
;; (In other words, a Tag is a list where the first element is a symbol,
;;   and all other elements are HI values)
```

Examples:

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

(a) Write the function `html->string` that consumes an HI and produces the equivalent HTML text.

Examples:

```
(check-expect (html->string "text") "text")
(check-expect (html->string short-example)
              "<p><h1>Heading</h1>Text</p>")
(check-expect (html->string '(hr)) "<hr></hr>")
```

You can (and should) use `symbol->string` and `string-append`.

Things to note: Each tags begins with `<tag-name>` and ends with `</tag-name>`.

(b) Write the function `remove-tag` that consumes an `HI` and a `Sym` and removes all occurrences of that tag. When a tag is removed, its contents should be moved to the parent of the removed tag, not removed entirely. If the root is removed, the function should return a list of its children.

Examples

```
(check-expect (remove-tag html-example 'b) html-example)
(check-expect (remove-tag '(p "Hello, " (b "World") "!") 'b)
              '(p "Hello, " "World" "!"))
(check-expect (remove-tag '(p "Hello, " (b "World") "!") 'p)
              '("Hello, " (b "World") "!"))
```

(c) HTML has rules for where tags can appear, and what children they can have. Here are two of the rules:

- A list item (`'li`) can only occur when its parent is an ordered list (`'ol`) or an unordered list (`'ul`).
- A horizontal rule (`'hr`) cannot have any children.

Write the function `bad-tags?` that consumes an `HI` and produces true if it has broken at least one of the two rules above. Examples:

```
(check-expect (bad-tags? html-example) false)
(check-expect (bad-tags? '(body (hr "hello"))) true)
(check-expect (bad-tags? '(body (li "Q1") "text")) true)
```

Note: The list tags are allowed to have children that aren't list items.

Place your solutions in `html.rkt`.

This concludes the list of questions for which you need to submit solutions. Do not forget to always check your email for the basic test results after making a submission.

Enhancements: *Reminder—enhancements are for your interest and are not to be handed in.*

Professor Temple does not trust the built-in functions in Racket. In fact, Professor Temple does not trust constants, either. Here is the grammar for the programs Professor Temple trusts.

⟨exp⟩ = ⟨var⟩|(**lambda** (⟨var⟩) ⟨exp⟩) | (⟨exp⟩⟨exp⟩)

Although Professor Temple does not trust **define**, we can use it ourselves as a shorthand for describing particular expressions constructed using this grammar.

It doesn't look as if Professor Temple believes in functions with more than one argument, but in fact Professor Temple is fine with this concept; it's just expressed in a different way. We can create a function with two arguments in the above grammar by creating a function which consumes the first argument and returns a function which, when applied to the second argument, returns the answer we want (this should be familiar from the `make-adder` example from class, slide 10-47). This generalizes to multiple arguments.

But what can Professor Temple do without constants? Quite a lot, actually. To start with, here is Professor Temple's definition of zero. It is the function which ignores its argument and returns the identity function.

```
(define my-zero (lambda (f) (lambda (x) x)))
```

Another way of describing this representation of zero is that it is the function which takes a function f as its argument and returns a function which applies f to its argument zero times. Then "one" would be the function which takes a function f as its argument and returns a function which applies f to its argument once.

```
(define my-one (lambda (f) (lambda (x) (f x))))
```

Work out the definition of "two". How might Professor Temple define the function `add1`? Show that your definition of `add1` applied to the above representation of zero yields one, and applied to one

yields two. Can you give a definition of the function which performs addition on its two arguments in this representation? What about multiplication?

Now we see that Professor Temple's representation can handle natural numbers. Can Professor Temple handle Boolean values? Sure. Here are Professor Temple's definitions of true and false.

```
(define my-true (lambda (x) (lambda (y) x)))
(define my-false (lambda (x) (lambda (y) y)))
```

Show that the expression `((c a) b)`, where `c` is one of the values `my-true` or `my-false` defined above, evaluates to `a` and `b`, respectively. Use this idea to define the functions `my-and`, `my-or`, and `my-not`.

What about `my-cons`, `my-first`, and `my-rest`? We can define the value of `my-cons` to be the function which, when applied to `my-true`, returns the first argument `my-cons` was called with, and when applied to the argument `my-false`, returns the second. Give precise definitions of `my-cons`, `my-first`, and `my-rest`, and verify that they satisfy the algebraic equations that the regular Scheme versions do. What should `my-empty` be?

The function `my-sub1` is quite tricky. What we need to do is create the pair $(0,0)$ by using `my-cons`. Then we consider the operation on such a pair of taking the "rest" and making it the "first", and making the "rest" be the old "rest" plus one (which we know how to do). So the tuple $(0,0)$ becomes $(0,1)$, then $(1,2)$, and so on. If we repeat this operation n times, we get $(n-1,n)$. We can then pick out the "first" of this tuple to be $n-1$. Since our representation of n has something to do with repeating things n times, this gives us a way of defining `my-sub1`. Make this more precise, and then figure out `my-zero?`.

If we don't have **`define`**, how can we do recursion, which we use in just about every function involving lists and many involving natural numbers? It is still possible, but this is beyond even the scope of this challenge; it involves a very ingenious (and difficult to understand) construction called the Y combinator. You can read more about it at the following URL (PostScript document):

`http://www.ccs.neu.edu/home/matthias/BTLS/tls-sample.ps`

Be warned that this is truly mindbending.

Professor Temple has been possessed by the spirit of Alonzo Church (1903–1995), who used this idea to define a model of computation based on the definition of functions and nothing else. This is called the lambda calculus, and he used it in 1936 to show a function which was definable but not computable (whether two lambda calculus expressions define the same function). Alan Turing later gave a simpler proof which we discussed in the enhancement to Assignment 7. The lambda calculus was the inspiration for LISP, a predecessor of Racket, and is the reason that the teaching languages retain the keyword **`lambda`** for use in defining anonymous functions.