Exercise 7: Your computer's file system.

Due: Friday, March 3

In this exercise, you'll learn to write programs that explore your computer's file system. Updating Racket Before you begin, update your course package (go to File>Update package, then press Update, as before).

Now search the documentation for "cs111/file-operations". This documentation has all of the file operations you will need for this assignment. There aren't that many functions, so seriously, **read it.** It will save you time later.

Note: If, when you search you see "No matches found in Advanced Student [clear]", press the [clear] and the documentation you are looking for should appear. Make sure when searching you go to the documentation for "cs111/file-operations".

Part 0: Some terms

There are three concepts from file systems we are using here: files, folders, and paths.

Files are objects that have names and store data, like the `.rkt` file you're currently working in.

Folders, otherwise known as **directories**, are objects that have names and store files and other folders.

Paths are a way of representing where to find files in a file system. They contain a sequence of folder names, each inside the previous, and may or may not contain a final name which refers to a file inside the previous folder.

Traditionally, paths are written as strings. They contain the names as described above separated by "/". So, the path "/Users/First Last/test.txt" says you can find a file called "test.txt" inside of the "First Last" folder which is inside the "Users" folder, which can be found in the main folder of the system (that's the first "/").

When stored in Racket, paths are in a special format, and thus **paths are not strings.** However, there are two procedures, `string->path` and `path->string` that allow you to easily convert between the two.

Note: Mac (OSX) and Unix paths look like "/folder1/folder2/folder3/folder4", while Windows paths look like "c:\folder1\folder2\folder3". *However,* in programming, usually the Windows style backslash "\" is replaced with the Unix style forward slash "/". So, "c:/folder1/folder2/folder3" is a valid Windows path.

Paths can be either "absolute paths" or "relative paths". Absolute paths start at the top of the file system (i.e. paths that start with "/" on OSX/Linux and "C:\"

on windows). Relative paths do not start at the top of the file system. Instead, they start at what we call the current directory. So, a path "test-folder/file.txt" says "file.txt, which is inside of test-folder, which is inside of the directory where I currently am". The current directory is (while in DrRacket) typically the directory where the current open file is saved to.

NOTE: All of the file operations that have side effects (like `delete-file!`) will only work on files that are and directories that are at in the same directory or sub directories of the directory where your assignment file is saved. This is to prevent you from accidentally deleting files elsewhere on you computer. This means that they only work on *relative* paths that to do not contain "..", which, in a path, means "go up a directory".

WARNING: This does not prevent you from accidentally deleting your homework. Take extra care to not do this. Make sure whenever you close DrRacket that your homework file is still there. We will not grant extensions for accidentally erasing your work.

Part 1: Making a backup program

We've included the copy-tree procedure at the beginning of the starter Racket file:

```
;; path path -> void (copy-tree! from-directory  to-directory)
Copies all the files and subdirectories in from-directory to
to-directory. This is a hacked (i.e. modified) version of the
copy-tree procedure we discussed in class. It has been modified
to call printf every time it copies a file or directory.  This
will print the names of the files it copies in the
interaction window.
```

Copy-tree uses the `printf` procedure that prints text to the interaction window.

```
;; string any ... -> void (printf format-string  other-arguments ...)
Prints format-string to the screen.  If format-string contains the
magic code ~A, it replaces the ~A with the next argument to printf.
Multiple ~A's print subsequent arguments.  ~n prints a newline
character (i.e. it starts a new line of output).  It's called printf
and not printf! for historical reasons; printf is the name used in the
original C language from the early 1970s, and it stuck.
```

Now type:

(copy-tree! "test" "output")

This command will copy all the files in the `test` subdirectory of the problem set into another directory called `output`. You should also see the following in the interactions window:

```
(copy-tree! "test" "output")

Copying file test\foo.bmp to output\foo.bmp

Copying file test\Test.txt to output\Test.txt

Copying file test\Test2\bar.txt to output\Test2\bar.txt

Copying file test\Test2\foo.bmp to output\Test2\foo.bmp
```

(Note: you may also see one or two extra files called Thumbs.db or .DS_Store that are normally invisible; if so, ignore them).

After running copy-tree, there should be a directory called output that holds identical copies of all the files and subdirectories of test. So we already have 90% of a backup program. Unfortunately, if we run it again, it will recopy all the files into output, even though there's no need to. So now we want to make a version that doesn't recopy files needlessly.

Make a copy of the procedure at the end of the file and rename it backup; remember to change it so that when it recurses, it calls backup, and not copy-tree. For the rest of this problem, you will modify the code for backup so that it works as a proper backup program.

Not copying existing files

Start by modifying the procedure so that it only copies a file if it doesn't already appear in the destination directory. You can do this by using the unless command:

```
(unless test expression)
```

Which will run the `expression` (like, um, say, file copying?) only if `test` is `#false` (so it's basically just a version of if that lets you run multiple expressions in sequence if the test is false). You can test whether a file already exists by using `(file-exists? file)`. It will return `#true` if there's already a file by that name, otherwise #false.

Testing

Now you need to make sure your new backup program works. So try running it:

```
(backup! "test" "output")
```

Since copy-tree already copied all the files over, you shouldn't see any of the "Copying file" lines. If you do see it copying files, that probably means there's a bug in your code. Check it out and try to see what's wrong.

Once you have it so that your `backup` function doesn't recopy needlessly, you need to make sure that it really does copy things when it needs to. So go into the output folder and delete one of the files. Now run backup again. You should see it just copying over the one file that you deleted.

Updating old files

We're almost there, but we have a problem in that our backup program only copies new files. If we change a file that we've already backed up, it won't copy the revised version into the archive. So we need to change the code so that it copies the file unless:

- It already appears in the to-directory, and
- The version in the to-directory is at least as recent as the version in the from directory

You can change the unless to whether two tests are true just by saying (and test1 test2). So now all we have to do is figure out how old the file is. You can do that by saying:

```
(file-or-directory-modify-seconds file)
```

This will return a number representing when the file was last changed, with greater numbers indicating the file was modified more recently. So change the test in the unless to first check whether the backed up version of the file exists, and then if it does, whether it's at least as new as the original (i.e. the modification time is at least as large). Note that when you copy the file on Windows, it's given the same write time as the original, so you want to make sure you don't copy the file if the two versions have the same modification time.

Subtlety: (if this is confusing, don't worry about it for now) It's important to have it check whether the file exists first, before checking the modification time, since you can't ask the operating system for the last modification time of a file that doesn't exist in the first place. It turns out that this kind of situation is common in programming. For that reason, the and expression is designed so that it runs the tests in order and stops as soon as it finds one that's false. So if you say:

```
(and (file-exists? ... whatever ...) ... check last modification time ...)
```

you don't have to worry about the time check generating an error if the file doesn't exists, because it won't run the second part if the file doesn't exist.

Retesting

Now rerun the program. Since we haven't modified anything in the test folder, it shouldn't copy any files. Again, if it does, figure out why and fix it.

Assuming it isn't copying any files at this point, we need to check that it properly copies updated files by updating one of the files and rerunning it:

1. Go into the test folder, and modify one of the files (e.g. edit Test.txt).
2. Then rerun your backup program. It should copy just the file you updated. If not, then look at the test in the unless and try to understand why.

Part 2: Searching

Now we're going to write procedures for searching through the file system to give you information about your files and directories.

1. Write a procedure, `(count-files path)`, that takes the pathname of a directory as input and returns the number of files within the directory and all its subdirectories (and their subdirectories, recursively) as its output.

- First write a procedure to count the number of files in the directory itself (i.e. not the subdirectories). You can get a list of the files in a directory using `directory-files`, which takes the pathname of the directory and returns a list of the pathnames of all the files in the directory.
- Then modify the procedure to call itself recursively for each of the directory's subdirectories. You can get a list of all the subdirectories of a directory using the `directory-subdirectories` procedure, which takes the pathname of the directory as input and returns a list of pathnames of the subdirectories as output.

Hint: use `map` to call `count-files` on every element of the list of subdirectories. Note: this is a little weird in that (like `copy-tree` and `backup`) it's a recursion that doesn't require you to use an if to keep it from recursing infinitely. If you call `directory-subdirectories` on a directory with no subdirectories, it will return the empty list and so `map` won't attempt to recurse any farther.

- Now use `foldl` to add up the sizes of all the subdirectories and the number of files in the directory itself.

2. Now, copy and then modify this procedure to make a new procedure, `(directory-size path)`, that gives the total size in bytes of all the files in the directory and its subdirectories. We have provided you with a procedure, `file-size`, that returns the size of a file (in bytes) given its pathname:

```
;; path -> number (file-size path)
Returns the number of bytes in the file specified by path.
```

3. Now write a procedure, `(search-directory string directory-path)`, to return a list of all the *pathnames* of all the files in the directory specified by directorypath whose *filename* contains a given string.

- First write a procedure that finds all the files in the directory whose filename contains string, ignoring the subdirectories. You can use the Racket procedure string-contains? to test if a filename contains a given string:

```
;; string string -> Boolean  (string-contains? search-string  string)
Returns true if and only if string contains search-string within it.
```

Important: The procedure should not return files unless their *filenames* contain the search string. Do not return a file just because it's in a directory whose name contains the search string. You can extract the file's name from its pathname using `path-filename`. That is, `(path-filename "C:\\a\\b\\c.txt")` will return "c.txt".

- Now use map to recursively call search-directory on all the directory's subdirectories. This will return a list of lists of pathnames, one list of pathnames per subdirectory.

Hint: `map` can't call `search-directory` directly, since it takes two inputs. So use `lambda` to construct a new procedure that takes one input and calls `search-directory` with both inputs, then pass that new procedure to `map`.

- Now use `append` to merge all the lists of pathnames together.

Note: `(append '(1 2) '(3 4) '(5 6))` will return one big list with all the elements of all of the lists, in this case: `(1 2 3 4 5 6)`. However, if you call `append` on a list of lists, as in: `(append '((1 2) (3 4)))`, you will just get back the original answer (a list of lists). To merge a list of lists into one list, use `apply` with `append`. `apply` calls a procedure and takes the arguments to pass to the procedure from a list. So if you do `(apply append '((1 2) (3 4)))`, it will call append with two arguments, the `(1 2)` list, and the `(3 4)` list, and so `append` will return the single list: `(1 2 3 4)`.

4. Now make a variant of `search-directory`, `(filter-directory predicate directorypath)`, that uses predicate to decide which files to return rather than always searching for a specific string. Remember that a predicate is a procedure that returns true or false. So filter-directory should take a procedure of one argument (the predicate) and a pathname for a directory (directorypath), and call the predicate on every pathname of every file in the directory (and its subdirectories, etc., recursively), and return a list of all the pathnames for which predicate returned true.

5. Use `filter-directory` to write a procedure, `(find-file-type extension directorypath)`, that takes a file extension, such as ".jpg" and a directorypath, and returns the paths of all the files inside the directory with the specified extension. You can test whether a path has a given extension using the procedure: `path-has-extension?`:

```
;; path string -> Boolean (path-has-extension? path extension)
Returns true if the filename in path has the specified extension.
An extension is something like ".jpg" or ".txt"
```

6. Use `find-file-type` to write a procedure, `(file-type-disk-usage extension directory-path)`, that reports the number of bytes used by files with the specified extension in the specified directory and its subdirectories.

Turning it in

Make sure your code is at the end of the Exercise 7.rkt file. Remove any calls to testing code you've included in your file. If your file includes testing code (calls to backup, etc.) it may crash in the grader because it will be run on a different set of files than you have on your hard disk. Upload to the handin server as usual.