Exercise 5:
Your Computer's File System

In this exercise, you'll learn to write programs that explore the files and directories (folders) your computer's file system. Specifically, you'll be writing:

- Procedures to **back up folders** by copying folders and files from one location to another, and
- Procedures to **search folders** for files of a certain name, extension, etc.

Note: This exercise needs the `file_operations.rkt` library, which is already required at the beginning of the starter code.

Introduction: Filesystem terminology

There are three concepts from file systems we are using here: **files**, **folders**, and **paths**. Filesystems are structured as trees, so this should feel familiar from past assignments. If you feel comfortable with filesystem data structures, feel free to skip this section.

- **Files** have names and contain data, like the .rkt file you're currently working in. (These are like leaf nodes on a tree.)
- **Folders**, also known as **directories**, have names and contain two types of data:
  - Files
  - Other folders

  Since they can contain other folders, they're a recursive structure, and form a tree.
- **Paths** represent where to find files in a file system. They consist of a sequence of folder names, and may or may not end in a file name. For example:
  - `/test/test_2/bar.txt` says there is a folder `test`, which contains a folder called `test_2`, which contains a file called `bar.txt` and we are referring to this file.
  - `/test/test_3/` says there is a folder `test`, which contains a folder called `test_3`, and we are referring to this folder.

  You may have seen paths written as strings, with folder names separated by a "/". In Racket, *paths are their own data type* – they are NOT strings.

  Note: in the last tutorial we represented paths as lists of strings (listof string). In this exercise we use a DIFFERENT representation with Racket's built-in path.

  To **create a new path**, use `build-path : string ... -> Path`:

```
(build-path "test" "test_2" "bar.txt")        ; /test/test_2/bar.txt
(build-path "test" "test_3")                  ; /test/test_3/
```

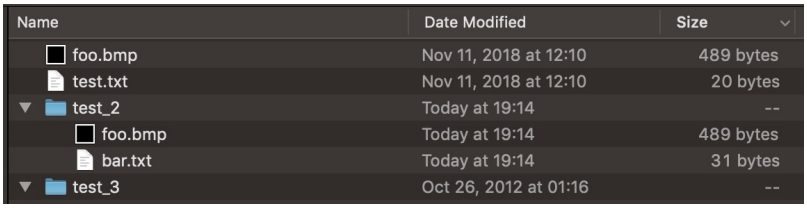
  You can also **convert between the two data types** using the procedures `string->path` and `path->string`:

```
(define p (build-path "test" "test_3"))
(path->string p) ; "test/test_3"
(string->path "test/test_2/bar.txt") ; #<path:test/test_2/bar.txt>
```

Note: On macOS 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 with "/" on macOS/Linux and "`C:\`" on windows, which mean the top-level directory of your hard drive. **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`" (note no / or `c:\` at the beginning) 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: We've modified the file operations that change the disk, like `delete-file!`, to only work on files and directories that are at in the same directory or sub directories as your assignment file. This is to prevent you from accidentally deleting files elsewhere on you computer. This means that they only work on *relative* paths. In this assignment, you will use the sample filesystem we have created for you to test with:



WARNING: Our safeguards for `delete-file!` do not prevent you from *accidentally deleting your homework*. BE CAREFUL NOT TO DO THIS! Make sure that whenever you close DrRacket, your homework file *still exists*. We will not grant extensions for accidentally erasing your work. *#imperativeprogramming*

Part 1: Backing up files

Before you write any new code, make sure you understand the `backup!` procedure. It's almost identical to the `copy-subtree!` procedure from the lectures. **Read through and understand** the code given in the starter file. If you're having difficulty making sense of it, ask a TA to help you before continuing.

Note: the `printf` function used in `backup!` has an appendix describing it at the end of this pdf. You can try using the procedure like this:

```
(backup! (string->path "test") (string->path "output"))
```

This command will take all the files in the test subdirectory of the assignment, and copy them into another directory called output. You should also see the following in the interactions window:

```
Copying file test\foo.bmp to output\foo.bmp
Copying file test\test.txt to output\test.txt
Copying file test\test_2\bar.txt to output\test_2\bar.txt
Copying file test\test_2\foo.bmp to output\test_2\foo.bmp
```

Question 1: Not copying existing files

In its current form,

```
(backup! (string->path "test") (string->path "output"))
```

will create a directory called `output` that holds identical copies of all the files and subdirectories of `test`. Unfortunately, if we run it a second time, it will re-copy all the files into `output`, even though the files already exist! So **modify `backup!` so that it only copies a file if it does not already exist in the destination directory**.

You can test this procedure by doing the following:

1. Run `(backup! (string->path "test") (string->path "output"))`, and verify the output directory is created
2. **It doesn't copy files unnecessarily:** Create a new file, such as `new-file.txt`, somewhere within the `test` directory. Rerun the command above, and verifying that `new-file.txt` gets copied into `output`, but none of the other files are re-copied.
3. **It still copies files when it needs to:** Delete one of the files from `output`, and re-run the command. The deleted file should be re-copied.

Useful functions

- `file-exists? : Path -> Boolean`
  Returns #true if the file at the given path exists, else #false.

Hints

- `when` and `unless` are simplified versions of `if` that are useful for imperative programming:

```
(when <test> <command>)
; If <test> returns #true, run <command>.
; Returns (void) either way.

(unless <test> <command>)
; If <test> returns #false, run <command>.
; Returns (void) either way.
```

  `when` is very similar to:

```
(if <test> <command> (void))
```

- The piece of code that handles copying files is this:

```
(begin
    (printf "Copying file ~A to ~A~n" file to)
    (copy-file! file
                (build-path to (path-filename file))
                #true))
```

  You'll want to *conditionally execute this begin statement,* using `when` or `unless`, depending on whether or not the destination file exists.

Question 2: Updating stale backups

Modify `backup!` to copy files that exist in the output directory, but have been modified in the original since they were last backed up.

In the previous question, we modified `backup!` to avoid making copies needlessly, but we made it too aggressive: now, if we make a backup and then change the original file, `backup!` won't copy the revised version into the output folder.

Update your code from Question 1 to copy files when:

- The file does **not** already exist in the backup directory (this was Question 1), or
- The file already exists in the backup directory, **but** the original file has been modified since the backup was created. In other words,

$$DateModified_{original} > DateModified_{backup}$$

.

To test this function:

1. **Check for regressions**: repeat all the tests from the previous question, to make sure no previously working code has been broken.
2. **It re-copies files that have been modified:** Modify one of the files in the origin test directory, for example, by editing `test.txt`. Now re-run the `backup!` command, and check to ensure the file was copied over.

Useful functions

- `file-or-directory-modify-seconds : Path -> Number`
  takes a path, and returns a number representing when the file was last changed. Greater numbers correspond to later times, meaning the file was modified more recently.

```
> (file-or-directory-modify-seconds (build-path "test" "test_2" "bar.txt"))
1479027433
```

Hints

- On Windows, copying a file gives it the same "Date Modified" as the original, so make sure you **do not copy the file** if the modification times are the same.
- If you are getting the following error:

```
file-or-directory-modify-seconds: error getting file/directory time ...
system error: No such file or directory; errno=2
```

  Remember that *order matters* in imperative code, and it's important to check whether the file exists BEFORE you check when it was last modified. (You can't ask the operating system for the date modified of a nonexistent file!)

  So you should structure your code like this:

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

  You won't have to worry about the error, because and will evaluate conditions in order and bail early as soon as one condition returns #false.

Part 2: Searching for files

Now that we have a functioning backup program, we'll write some procedures for searching through the filesystem to give you information about your files and directories.

Question 3: Counting files

Write a procedure, `count-files`, 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).

```
; count-files : path -> number
; Takes a path to a directory, and returns the number of files within the
; directory and all its descendants, recursively.
```

So:

```
(count-files (string->path "test"))
```

Would return 4, assuming you hadn't modified the test directory.

Important note: due to a hidden MacOS system file called `.DS_Store` you may see a result here that is one more than the number of visible files. Windows will sometimes also create a hidden file called `Thumbs.db`. So don't panic if you see a number of files that looks one or two files too large. It's probably giving the right answer; it's just including files you can't see. You can use `directory-files` to check the files that appear in a given directory and look to see if it's listing one of these hidden files. Alternatively, you can configure your operating system to display them in folders. To view hidden files on MacOS, use the keyboard shortcut **`Cmd+Shift+.`** while in the Finder (see here for more details). To view hidden files in Window check the "Hidden items" box in the "View" tab of File Explorer (see here for more details).

Useful functions

- `directory-files : Path -> List-of-Path`
  takes a directory path, and returns a list of paths to the **files** contained in that directory.

  ```
  > (directory-files (build-path "test"))
  (list #<path:test/test.txt> #<path:test/foo.bmp>)
  ```

- `directory-subdirectories : Path -> List-of-Path`
  Takes a directory path, and returns a list of paths to the sub-directories contained in that directory.

  ```
  > (directory-subdirectories (build-path "test"))
  (list #<path:test/test_2> #<path:test/test_3>)
  > (directory-files (build-path "test" "test_2"))
  (list #<path:test/test_2/bar.txt> #<path:test/test_2/foo.bmp>)
  ```

Hints

You can break this problem down as follows:

1. Write a simple procedure called `count-files` to count the number of files in a directory itself (i.e. not the subdirectories).
2. Modify it to call itself recursively for each subdirectory. You can do this by using `map` to recursively call `count-files` on each path in the list of subdirectories!

3. Finally, use `foldl` or `apply` to add up the number of files in the current directory, as well as the number of files in all subdirectories.

   Remember (`map count-files ...`) will return a *list* of results, and `foldl` and `apply` take a list argument!

   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. Since `map` has an `if` that checks for the empty list, that's enough to stop the recursion.

Question 4: Getting the size of a directory

Write a procedure called `directory-size`, which takes a path and returns the total size in bytes of all files in the directory and its subdirectories, recursively.

```
; directory-size : Path -> Number
; Returns the number of bytes of the given directory and all its contents,
; recursively
```

So:

```
(directory-size (string->path "test"))
```

Would return 1029 for the original version of the `test` directory. If you've changed the `test` directory, you may get a somewhat different number. Also, remember that both MacOS and Windows can create hidden files (`.DS_Store` and `Thumbs.db`, respectively). You can ask the operating system what it thinks the size of the directory is by right-clicking and choosing Get Info (MacOS) or Properties (Windows).

This will be very similar to the previous function, except instead of getting the number of files, you're getting the size of all the files.

Useful functions

- `file-size : Path -> Number`
  Takes a path to a file, and returns its size in bytes.

```
> (file-size (string->path "test/test.txt"))
20

; file-size only works on files, not directories!
> (file-size (string->path "test/test_2"))
file-size: cannot get size
path: /Users/sarah/exercise_6/test/test_2
system error: path refers to a directory; rktio_err=9
```

Question 5: Searching a directory

Write a procedure called `search-directory` that takes a search string and a directory path, and returns a list of paths of files in the directory whose names contains the given string. **Important:** it should only returns paths of files who *file name* contains the string, not whose paths happen to contain the string elsewhere.

As with previous functions, you need to search the given directory and all its subdirectories.

```
; search-directory : String, Path -> (listof Path)
; Returns a list of paths to files within the original directory, whose
; filenames contain the given string.

> (search-directory "foo" (string->path "test"))
(list #<path:test/foo.bmp> #<path:test/test_2/foo.bmp>)
; Only search filenames, NOT folder names!
> (search-directory "test_2" (string->path "test"))
'()
```

Useful functions

- `path-filename : Path -> Path`
  Takes a path to a file, and returns a shortened path containing only the filename portion.

  ```
  > (define file (string->path "test/test_2/foo.bmp"))
  > file
  #<path:test/test_2/foo.bmp>
  > (path-filename file)
  #<path:foo.bmp>
  ```

- `path->string : Path -> String`
  Takes a path, and returns a string version.

  ```
  > (path->string (string->path "some/path/to/file"))
  "some/path/to/file"
  ```

- `string-contains? : String, String -> Boolean`
  Takes a query string and a string to search, and returns #true if the second string contains the query:

  ```
  > (string-contains? "ack" "Racket")
  #true

  > (string-contains? "Racketttt" "Racket")
  #false
  ```

Hints

As before, you can follow this recipe to break down your function:

1. Write `search-directory` to only search the files immediately contained by the original directory. Ignore subdirectories for now.

   ```
   > (search-directory "foo" (string->path "test"))
   (list #<path:test/foo.bmp>) ; only one file
   ```

2. Now use `map` to recursively call `search-directory` on all of the subdirectories.

   Note: since `search-directory` itself returns a list of paths, calling `(map search-directory ...)` will return a list of lists of paths (woah):

   Note: `map` can't call `search-directory` directly, since `search-directory` takes two inputs and

`map` only calls the procedure you give it with one argument (a list element). So you should use lambda to create a new one-argument procedure that calls `search-directory`. (Sound familiar? Think back to the homework with `artist-is-versatile?`!)

3. Finally, use append to merge all the lists of pathnames together. You can use `(apply append ...)` to flatten a list of lists:

```
; `append` only merges lists one level deep, so passing it lists of
; lists will return lists of list:
> (append (list (list 1 2)
                (list 3 4))
          (list (list 4 5)
                (list 6 7)))
(list (list 1 2) (list 3 4) (list 4 5) (list 6 7))

; And calling it with just one list, returns it unchanged:
> (append (list (list 1 2)
                (list 3 4)))
(list (list 1 2) (list 3 4))

; But if we apply append to a list of lists, it calls append with the
; sublists as separate arguments.  So this:
> (apply append
         (list (list 1 2)
               (list 3 4)))
; really just runs (append (list 1 2) (list 3 4)), which gives us:
(list 1 2 3 4)
```

Question 6: Filtering directory contents

Write a variant of `search-directory` called `filter-directory`, which takes a predicate and a path to a directory, and returns all files within that directory (and its descendants) whose paths the predicate returns true for.

```
; filter-directory : (Path -> Boolean), Path -> List-of-Path
; Returns a list of paths to files within the original directory, which
; pass the given predicate.

; Example usage (results will vary from computer to computer):
> (filter-directory (lambda (file-path)
                      (< (file-size file-path)
                         10))
                    (string->path "test"))
```

Hints

- Recall that a **predicate** is simply a procedure that returns a Boolean.
- This is just a more general version of `search-directory`! In fact, you could have written `search-directory` to just call `filter-directory` with a predicate that checks whether the file name of the path contains the search string you're looking for.

Question 7: Finding certain filetypes

The extension of a file name is the part at the end that says "`.`*something*": `.jpg`, `.rkt`, `.txt`, etc. Most contemporary operating systems use the extension to designate what kind of data a file has in it.

Use `filter-directory` to write a procedure `find-file-type`, which takes a file extension such as "`.jpg`" and a path, and returns a list of paths to all files with that extension.

```
; find-file-type: String, Path -> List-of-Path
; Returns a list of paths to files within the original directory, which
; have the given extension.
; You should call `filter-directory` in your solution.

> (find-file-type ".bmp"
                  (string->path "test"))
(list #<path:test/foo.bmp> #<path:test/test_2/foo.bmp>)
```

Useful functions

- `path-has-extension? : Path, String -> Boolean`
  Tests whether a given path has a given extension.

```
> (path-has-extension? (string->path "test/test.txt") ".txt")
#true
> (path-has-extension? (string->path "test/test.txt") ".jpg")
#false
```

Question 8: Finding file type storage space

Now use `find-file-type` to write a procedure `file-type-disk-usage` that takes an extension and a directory path, and returns the number of bytes used by all files of that type within that directory and its descendants.

```
; file-type-disk-usage: String, Path -> Number
; Returns the number of bytes used by files within the original directory,
; which have the given extension.
; You should call `find-file-type` in your solution.

; Example usage (results will vary by computer)
> (file-type-disk-usage ".bmp" (string->path "test"))
```

Hints

- Although you previously wrote a procedure called `directory-size`, remember that you have to use `find-file-type` in your solution. Before you immediately jump to replicating your logic from `directory-size`, think about what `find-file-type` returns, and how you might use that to simplify your answer!
- You don't have to write a recursion for this.

Turning it in

1. Make sure your code is at the end of the `Exercise 6.rkt` file.

2. **Very important**: remove any calls in your file to the procedures you wrote, such as calls to `backup!`, etc. The grader will run your program in a directory with a different file structure (e.g. there won't be a subdirectory named test), so any calls you leave in will very likely throw exceptions and stop the grading process.

Appendix 1: Understanding `printf` in the starter code

The starter code includes a function called copy-tree!

```
; copy-tree! : Path, Path -> Void
```

which takes a *from* path and a *to* path, and recursively copies the contents of *from* into *to*. This is a modified version of the copy-tree procedure discussed in class. Namely, it includes the following line:

```
(printf "Copying file ~A to ~A~n" file to)
```

The `printf` function has the signature

```
; printf : string, any ... -> void
```

and prints the given string, called the "format string", to the screen (the REPL), with two twists:

- If the format string contains the magic code ~A, it replaces each ~A with the corresponding argument after the format string. For example:

  ```
  (printf "~A is ~A" "Racket" "great")
  ```

  will print "Racket is great".

- If the format string contains ~n, then `printf` starts a new line of output.

  Note: 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.