# Exercise 8: Your Computer's File System

EECS 111

Due Friday, March 23rd by Midnight

In this exercise, you'll learn to write programs that explore your computer's file system. Specifically, you'll be writing:

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

**Note:** This exercise needs the `cs111/file-operations` library, which is already required at the beginning of the starter code. Other than this library, you can only require libraries in `cs111/●` and `2htdp/●`.

## Introduction: Filesystem terminology

There are three concepts from file systems we are using here: **files**, **folders**, and **paths**. Filesystems are shaped like 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**, are recursive structures that have names and contain two types of data:
  - Files
  - Other folders

  Folders are like non-leaf nodes on trees.
- **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/Test2/bar.txt` says there is a folder `test`, which contains a folder called `Test2`, which contains a file called `bar.txt` and we are referring to this file.
  - `/test/Test3/` says there is a folder `test`, which contains a folder called `Test3`, 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" "Test2" "bar.txt")  ; /test/Test2/bar.txt
  (build-path "test" "Test3")  ; /test/Test3/
  ```

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

```
(define p (build-path "test" "Test3"))
(path->string p)   ; "test/Test3"
(string->path "test/Test2/bar.txt")   ; #<path:test/Test2/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 at the top of the file system (i.e. paths that start with "/" on macOS/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".

In this assignment, you will use the sample filesystem we have created for you to test with:

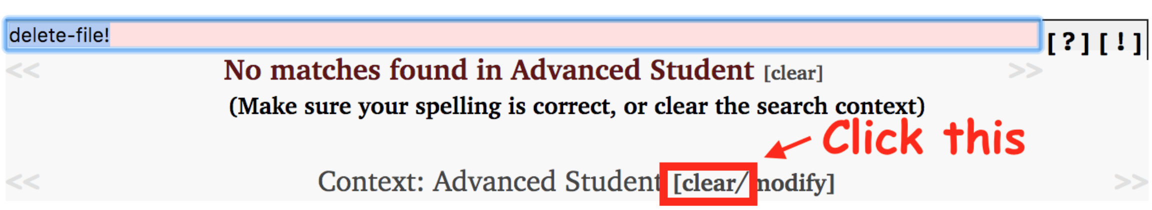
| Name | Date Modified | Size |
|---|---|---|
| foo.bmp | Nov 6, 2007, 10:56 PM | 489 bytes |
| Test.txt | Nov 13, 2016, 2:49 AM | 20 bytes |
| ▼ Test2 | Today, 8:02 PM | -- |
| bar.txt | Nov 13, 2016, 2:57 AM | 31 bytes |
| foo.bmp | Nov 6, 2007, 10:56 PM | 489 bytes |
| ▼ Test3 | Oct 26, 2012, 12:16 AM | -- |

The functions you'll be writing have *real side effects* (for instance, `delete-file!` will actually delete real files). As a safeguard, all of these operations will only work on files and directories that are *descendants of the folder where your assignment file is saved.*

However, this does 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

## Troubleshooting documentation

**All of the file operations you need for this assignment are documented under `cs111/file-operations`.** If, when you search, you see something like this:




then **click the "clear"** link and the documentation you're looking for should appear.

## Part 1: Backing up files

Before you write any new code, **read through and understand** the starter code for the `backup!` procedure.

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

### Question 1: Not copying existing files

Modify `backup!` so that it only copies a file if it does not already exist in the destination directory.

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!

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. Run the same command, and verifying that `new-file.txt` gets copied into `output`, but none of the other files are re-copied. (You can check the "Date Added" field in your file explorer to confirm this.)
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 the imperative counterparts to `if`:

  ```
  (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* 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 origin 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 all 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,

$$DateModified_{from} \geq DateModified_{to}$$

.

To test this function:

1. **Check for regressions:** Repeat all the tests from the previous question, to make sure no behavior regressed (used to work and now doesn't).
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" "Test2" "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 your 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 conditionals 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 -> void
; Takes a path to a directory, and returns the number of files within the directory
; and all its descendants, recursively.
; 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

(count-files (build-path "test"))  ; 4, assuming you didn't modify the test filesystem
```

### 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/Test2> #<path:test/Test3>)

  > (directory-files (build-path "test" "Test2"))
  (list
   #<path:test/Test2/bar.txt>
   #<path:test/Test2/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).

   ```
   (count-files (build-path "test"))          ; 2
   (count-files (build-path "test" "Test2"))  ; 2
   (count-files (build-path "test" "Test3"))  ; 0
   ```

2. Modify that procedure to call itself recursively for each subdirectory.

   ```
   (count-files (build-path "test"))          ; now returns 4
   (count-files (build-path "test" "Test2"))  ; still 2
   (count-files (build-path "test" "Test3"))  ; still 0
   ```

   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.

## 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
; Note: the size here should match the number when you use the system viewer
; right-click -> Get Info on macOS, right-click -> Properties on Windows
; If it matches that number but it isn't 1029 like the comment below don't worry
; your code is correct

(directory-size (build-path "test"))  ; 1029, assuming no modifications to the filesystem
```

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 (build-path "test" "Test.txt"))
  20

  ; file-size only works on files, not directories!
  > (file-size (build-path "test" "Test2"))
  file-size: cannot get size
    path: /Users/sarah/exercise_8/test/Test2
    system error: path refers to a directory; rktio_err=9
  ```

## Question 5: Searching a directory

Write a procedure called `search-directory`, which takes a search string and a directory path. Return a list of paths, where each path points to a file whose name contains the given string.

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

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

> (search-directory "foo" (build-path "test"))
(list
    #<path:test/foo.bmp>
    #<path:test/Test2/foo.bmp>)

; Only search filenames, NOT folder names!
> (search-directory "Test2" (build-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 (build-path "test" "Test2" "foo.bmp"))
  > file
  #<path:test/Test2/foo.bmp>
  > (path-filename file)
  #<path:foo.bmp>
  ```

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

  ```
  > (path->string (build-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" (build-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 that since `search-directory` itself returns a `List-of-Path`, calling `(map search-directory ...)` will return a `List-of-List-of-Path` (woah):

```
; (map search-directory ...)
;   |     |__________|
;   |      Listof Path
;   |____________________|
;      Listof Listof Path
```

Note that `map` can't call `search-directory` directly, since `search-directory` takes two inputs and `map` only iterates one list. 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 a single list of lists
; will just return the same thing:
> (append (list
           (list 1 2)
           (list 3 4)))
(list (list 1 2) (list 3 4))

; Using `apply` calls `append`, but "spreads" the list of arguments as the inputs
; to `append`. We can use this to flatten a single list of lists:
> (apply append
         (list
                  (list 1 2)
                  (list 3 4)))
        (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) that pass the predicate.

```
; 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))
    (build-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`! You should be able to pass in a predicate that will make `filter-directory` behave exactly like `search-directory`.

## Question 7: Finding certain filetypes

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"
    (build-path "test"))
(list
 #<path:test/foo.bmp>
 #<path:test/Test2/foo.bmp>)
```

#### Useful functions

- `path-has-extension? : Path, String -> Boolean` tests whether a given path has a given extension (anything that starts with a `.`).

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

### Question 8: Finding file type storage space

Use `find-file-type` to write a procedure `file-type-disk-usage`, which takes an extension and a directory path, and returns the number of bytes used by all files 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" (build-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!

## Turning it in

1. Make sure your code is at the end of the `exercise_8.rkt` file.
2. **Very important:** Remove any function calls at the top level of your file (i.e., not inside a `check-expect`). If your file includes testing code (calls to `backup!`, etc.) it will CRASH in the grader, because it will be run on a different set of files than you have on your hard disk.
3. Upload to the handin server as usual. Congrats!

## 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 (deeply) 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 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.