# Exercise 5:
# Directories and subdirectories

**Out:** Friday, November 6
**Due:** Friday, November 13, **noon**

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

## Part 1: Making a backup program

We've included the copy-tree procedure at the beginning of Exercise 5.rkt:

- (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 print-line 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 a procedure you haven't seen before, called print-line that prints text to the interaction window. 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 interaction window:

```
 > (copy-tree "test" "output")
 Copying file test\foo.bmp to output\foo.bmp
 Copying file test\Test.txt to output\Test.txt
 Copying directory test\Test2 to output\Test2
 Copying file test\Test2\bar.txt to output\Test2\bar.txt
 Copying file test\Test2\foo.bmp to output\Test2\foo.bmp
 Copying directory test\Test3 to output\Test3
>
```

(Note: you may also see one or two extra files called Thumbs.db 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")
Copying directory test\Test2 to output\Test2
Copying directory test\Test3 to output\Test3
>
```

Since copy-tree already copied all the files over, then you should just see the "Copying directory" lines in the output, and not 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 it 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 verison 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 verison 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, and check the last write time second, since you can't ask the operating system for the last write time of a file that doesn't exist in the first place. It turns out that this kind of situation is common in programming. You have to check two things – *test1* and *test2* – but *test2* will break if *test1* is false. 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. Thus you don't have to worry about the time check generating an error if the file doesn't exist, so long as you make sure the the existence test comes first.

### 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:

- Go into the test folder, and modify one of the files (e.g. edit Test.txt).
- 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: Fun with recursion

The backup program you wrote used a higher-order procedure, for-each, to do the iteration. That's the standard programming style used in Racket. Now that you have that version of the code down, we want you to write equivalent procedures using a recursion. Write a new procedure, backup-without-for-each, that doesn't use for-each, but instead uses a recursion to loop through the list of files.

**Hints:**

- An easy way to do this is to write your own version of for-each using recursion; that will allow you to keep your existing code
- If you do reimplement for-each, however, make sure you either give it a different name than for-each, or you put it inside of backup-without-for-each (using local) because otherwise you will be redefining Racket's built-in for-each, and Racket will object to that.

**Extra hint:** A common mistake to make here would be to copy the original version of backup, rename it, and modify it, but forget to also change the name in the recursive call. Then backup-without-for-each doesn't call itself when it "recurses"; instead it calls the original backup. This will be counted as a bug during grading, so make sure you don't make this mistake.

## Part 3: 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 + 0 *list-of-sizes*) to add up the sizes of all the subdirectories, then add in the number of files in the directory itself.

2. Now 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.

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

   - First write a procedure that finds all the files in the directory whose name contains the *string*, ignoring the subdirectories. You can use the Racket procedure (string-contains? *search-string string*), that returns true 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 the 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 λ 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. Remember that 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 directory-path*), 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 (*directory-path*), 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 directory-path*), that takes a file *extension*, such as ".jpg" and a *directory-path*, 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 extension*).

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 5.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.