Due: At the start of class on March 1, 2012 in person, or by noon that same day if you submit by email.

This document is designed to be read as a tutorial that includes six questions along the way. I strongly encourage you to read the text from start to finish and answer the questions as you go: you'll probably find useful context or suggestions in the text that will be missing if you just jump directly to the questions. The end of each question is marked by a horizontal line across the page, just like this:

Background: The `find` command that is provided on many Unix-based operating systems is a powerful and flexible tool for searching the contents of a directory and its subfolders. The first argument to find specifies a directory in which the search should begin; this is then followed by a sequence of filter commands, each of which can be used to narrow the selection of items or to perform some specified actions. For example:

```
find . -name '*hs' -print
```

begins a search in the current directory (indicated by `.`); looks for items with a name that end in `hs` (likely Haskell source files); and then prints out the final list. The find command has many more options than these; enter the command `man find` on your favorite Unix-based machine for more details.

The purpose of this exercise is to build a library for executing find-like commands in Haskell. Rather than building a sophisticated but monolithic program that recognizes only a predetermined set of filter commands, we'll aim for a more open design that can be extended by writing new filters as Haskell functions. The code that is given (or requested) in this assignment is not expected to be production quality, but might still help to demonstrate the flexibility that can be obtained using this approach.

[Aside: In some circles, the `find` command would be referred to as a "domain specific lannguage" (DSL); it is clearly not a general purpose programming language, but is sophisticated enough to let users construct a wide range of "programs" for searching directory contents. In the same circles, the approach that we use in this assignment might be referred to as an "embedded domain specific language" (EDSL) because of the way that it has been embedded inside another, so-called, "host" language. The host (in this case, Haskell) provides users not only with the specific features of the EDSL, but also with the full power of the language in which it is embedded. During the past decade (or more), functional languages have gained a strong reputation as ideal hosts for EDSL development and design.]

We'll start with the module header that we'll need for this library. Of course, if this were going to be used as a real library, then we'd probably want a better name than `HW7` :-)

```
module HW7 where
```

This code uses the `IOActions` library, which also exports all of the features of `Data.List`. That is handy, except that it includes a function called `find` that would clash with the find operation we're defining here. To fix that, we can use a `hiding` clause to prevent the import of the `Data.List` version of `find`:

```
import IOActions hiding (find)
import System.IO
```

QUESTION 1: Use online sources or inspect the files that are already installed on your computer to give brief documentation, in your own words, for the find function that is defined in `Data.List`. Target your comments at a reader who has basic familiarity with Haskell, but no previous experience with `Data.List`.

The main task that we need to handle is the process of traversing a directory to enumerate all of the files that it contains. The following function can be used for this task:

```
find      :: FilePath -> IO [FilePath]
find path  = doesDirectoryExist path >>= \isDir ->
             case isDir of
               True  -> getDirectoryContents path
                        >>= inIO (filter notDotDot)
                        >>= mapM (find . (path </>))
                        >>= inIO ((path :) . concat)
               False -> return [path]
 where notDotDot name = name /= "." && name /= ".."
```

As a simple example, you might try an expression like the following to should print a list of all the files and directories that are in the current directory or one of its subdirectories:

```
find "." >>= mapM_ putStrLn
```

QUESTION 2: Explain briefly how the code for `find` works, and comment on the need for the local definition of `notDotDot`. [Hint: if you're not sure why that is needed, you could always try commenting out the relevant section of the code and then see what happens. Hint 2: Remember that you can stop a long-running computation by hitting `^C` (control C) . . .]

Now that we have a way of producing a list of `FilePaths`, we can work on ways of filtering those lists down to contain only those names that meet specific criteria. In general, we'll use expressions of the following form to describe a complete search:

```
find dir -| filt1 -| filt2 -| ... -| filtn
```

where `filt1`, `filt2`, . . . , `filtn` are independent filters. The intuition here is that `find dir` produces an initial list of all the items in the specified directory, `dir`. After this the filters `filt1`, `filt2`, . . . `filtn` are run in turn to trim down the generated list of files. The key to making this work is the definition of the `-|` symbol as an infix operator using the following definition:

```
infixl -|

(-|)   :: IO [FilePath] -> (FilePath -> IO Bool) -> IO [FilePath]
g -| p  = g >>= filterIO p

filterIO         :: (a -> IO Bool) -> [a] -> IO [a]
filterIO p []     = return []
filterIO p (x:xs) = do b <- p x
                       if b then filterIO p xs >>= inIO (x:)
                            else filterIO p xs
```

As you can see from this code, the `-|` operator simply runs the IO action `g` to generate a list of `FilePath` values, then uses a function called `filterIO` to select the specific items that satisfy the predicate function `p`. As the name suggests, `filterIO` is much like the standard `filter` function on lists except that it takes a predicate of type `(a -> IO Bool)` instead of the `(a -> Bool)` type that is expected for the Prelude's `filter` function. This allows the predicate to perform IO actions (such as determining the size of a file) in the process of deciding whether to return either a `True` or `False` result to indicate whether each name should be kept or dropped from the list of file paths.

[Full disclosure: There is a more general version of the `filterIO` function called `filterM` that is defined in the `Control.Monad` library. We will get to that later; for now, it's only important that you understand how `filterIO` works.]

Note that the functions `doesFileExist` and `doesDirectoryExist` have exactly the type that we need for the second argument of `-|`, so we can immediately use expressions like the following to list all of the files (resp. directories) under the current directory:

```
find "." -| doesDirectoryExist >>= mapM_ putStrLn

find "." -| doesFileExist >>= mapM_ putStrLn
```

QUESTION 3: The declaration `infixl -|` specifies that the `-|` operator should be treated as a function that associates (or groups) to the left. Explain why this is necessary here. [Hint: you might like to consider expressions of the form: `find dir -| filt1 -| filt2`.]

As we've seen above, the right argument of the `-|` operator must be a function of type `(FilePath -> IO Bool)`. How can we construct useful functions of that type?

One detail that we can use in deciding whether or not to keep a `FilePath` in the list is to look at its name. This doesn't require any IO, so it can be handled by combining the `return` function with a (pure) predicate of type `(FilePath -> Bool)`:

```
name       :: (FilePath -> Bool) -> FilePath -> IO Bool
name p f    = return (p f)
```

For example, the find command that we gave at the start of this file for listing all the Haskell source files in the current folder can now be implemented using:

```
find "."  -|  name ("hs" `isSuffixOf`) >>= mapM_ putStrLn
```

If we thought we might use this particular example a lot, then it might even be worth giving a name to a function like this:

```
haskellFiles = name ("hs" `isSuffixOf`)
```

And then we can just type:

```
find "." -| haskellFiles >>= mapM_ putStrLn
```

Note that the argument to `name` can be any Haskell function of type `(FilePath -> Bool)`. So, if we wanted to, we could even find the list of all files that have an even number of characters in their path name by doing something like:

```
find "." -| name (even . length) >>= mapM_ putStrLn
```

Now I'm not going to claim that's particularly useful, but it isn't something you can immediately do with the standard Unix `find` command because that wasn't something that its designers imagined you might want to do when they designed their program. With our Haskell approach, on the other hand, we have free access to all of the features and libraries that are provided by the language.

QUESTION 4: Show how the expression above can be modified to verify that all of the `FilePath`s in the final list do indeed, have an even number of characters in their name.

We might also want to be able to trim a list of `FilePath` values by considering the sizes of the files that they refer to. For example, suppose I want a list of all of the Haskell files in the current directory that are less that 1000 bytes long:

```
find "." -| name ("hs" `isSuffixOf`) -| size (<1000) >>= mapM_ putStrLn
```

This time, we're using a function called `size` that uses an arbitrary predicate on file sizes to determine whether or not a file should be kept in the list. In the example above, the predicate is `(<1000)` because we want files that are less than 1000 bytes long, but again any `Integer`-valued predicate could be used.

The `size` function can be defined as follows:

```
size    :: (Integer -> Bool) -> FilePath -> IO Bool
size p f = do b <- doesFileExist f
              if b then do h <- openFile f ReadMode
                           l <- hFileSize h
                           hClose h
                           return (p l)
                   else return False
```

Note that I've started this definition by testing to ensure that the given `FilePath` refers to a file, and returning `False` if it refers instead to a directory. (This is necessary because we want size to produce the list of all *files* in the input that satisfy the given predicate.)

This code uses a few features that we've only glossed over in class. In principle, it would have been possible to have found the size of a file by reading in its contents as a list and then calculating its length. In practice, it is much more efficient to have the underlying operating system tell us how big a file is (using the `hFileSize` function from the `IO` library) because it can typically determine that information without having to read in the whole file.

There are times, however, when it might be useful to filter a list of `FilePath`s by looking at the contents if each file ...

QUESTION 5: Define a function of the following type:

```
contents    :: (String -> Bool) -> FilePath -> IO Bool
contents p f = undefined --- replace with your own code
```

The idea here is that, for an appropriately typed argument `p`, the expression `contents p` should produce a filter that keeps only those files whose contents satisfies the predicate `p`. For example, a command like:

```
find "." -| contents (even . length . lines) >>= mapM_ putStrLn
```

would find all of the files under the current directory that have an even number of lines in them.

[Hint: `readFile` might be a useful function for finding the contents of an arbitrary `FilePath` value.]

There are many other kinds of filter commands that we could define in this way. One simple example that would have saved from having to write `mapM_ putStrLn` in the previous code samples is:

```
display  :: FilePath -> IO Bool
display f = do putStrLn f
               return True
```

This "filter" command doesn't actually filter anything out—it returns `True` for every file—but it does at least display each name on the terminal, so now I can write things like the following:

```
find "." -| haskellFiles -| size (<100) -| display

find "." -| doesDirectoryExist -| display
```

At last, the syntax of these examples is starting to look a little bit prettier!

As another example, we might want to give the user an opportunity to select which of the files in a list should be included ...

QUESTION 6: Define a filter command of the following type that allows for user interation in a find command:

```
queryUser  :: String -> IO Bool
queryUser s = undefined --- replace with your own code!
```

More specifically, `queryUser` should display the `FilePath` that it is given, then a prompt `" (y/n)?"`, and then wait for the user to decide whether or not to keep that item by testing to see whether the line that is entered begins with a `y`. The following example shows how this function works on my machine.

```
HW7> find "." -| haskellFiles -| size (<100) -| queryUser -| display
./brain.hs (y/n)? y
./catalans.hs (y/n)? y
./Client.hs (y/n)? n
./brain.hs
./catalans.hs

HW7>
```
