Haskell/Libraries/IO

The IO Library

Here, we'll explore the most commonly used elements of the `System.IO` module.

```haskell
data IOMode = ReadMode | WriteMode
             | AppendMode | ReadWriteMode

openFile :: FilePath -> IOMode -> IO Handle
hClose :: Handle -> IO ()

hIsEOF :: Handle -> IO Bool

hGetChar :: Handle -> IO Char
hGetLine :: Handle -> IO String
hGetContents :: Handle -> IO String

getChar :: IO Char
getLine :: IO String
getContents :: IO String

hPutChar :: Handle -> Char -> IO ()
hPutStr :: Handle -> String -> IO ()
hPutStrLn :: Handle -> String -> IO ()

putChar :: Char -> IO ()
putStr :: String -> IO ()
putStrLn :: String -> IO ()

readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

Note

`FilePath` is a *type synonym* for `String`. So, for instance, the `readFile` function takes a `String` (the file to read) and returns an action that, when run, produces the contents of that file. See the Type declarations chapter for more about type synonyms.

Most of the IO functions are self-explanatory. The `openFile` and `hClose` functions open and close a file, respectively. The `IOMode` argument determines the mode for opening the file. `hIsEOF` tests for end-of file. `hGetChar` and `hGetLine` read a character or line (respectively) from a file. `hGetContents` reads the entire file. The `getChar`, `getLine`, and `getContents` variants read from standard input. `hPutChar` prints a character to a file; `hPutStr` prints a string; and `hPutStrLn` prints a string with a newline character at the end. The variants without the `h` prefix work on standard output. The `readFile` and `writeFile` functions read and write an entire file without having to open it first.

Bracket

The `bracket` function comes from the `Control.Exception` module. It helps perform actions safely.

```haskell
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

Consider a function that opens a file, writes a character to it, and then closes the file. When writing such a function, one needs to be careful to ensure that, if there were an error at some point, the file is still successfully closed. The `bracket` function makes this easy. It takes three arguments: The first is the action to perform at the beginning. The second is the action to perform at the end, regardless of whether there's an error or not. The third is the action to perform in the middle, which might result in an error. For instance, our character-writing function might look like:

```
writeChar :: FilePath -> Char -> IO ()
writeChar fp c =
    bracket
      (openFile fp WriteMode)
      hClose
      (\h -> hPutChar h c)
```

This will open the file, write the character, and then close the file. However, if writing the character fails, `hClose` will still be executed, and the exception will be reraised afterwards. That way, you don't need to worry too much about catching the exceptions and about closing all of your handles.

A File Reading Program

We can write a simple program that allows a user to read and write files. The interface is admittedly poor, and it does not catch all errors (such as reading a non-existent file). Nevertheless, it should give a fairly complete example of how to use IO. Enter the following code into "FileRead.hs," and compile/run:

```
import System.IO
import Control.Exception

main = doLoop

doLoop = do
  putStrLn "Enter a command rFN wFN or q to quit:"
  command <- getLine
  case command of
    'q':_ -> return ()
    'r':filename -> do putStrLn ("Reading " ++ filename)
                       doRead filename
                       doLoop
    'w':filename -> do putStrLn ("Writing " ++ filename)
                       doWrite filename
                       doLoop
    _            -> doLoop

doRead filename =
    bracket (openFile filename ReadMode) hClose
            (\h -> do contents <- hGetContents h
                      putStrLn "The first 100 chars:"
                      putStrLn (take 100 contents))

doWrite filename = do
  putStrLn "Enter text to go into the file:"
  contents <- getLine
  bracket (openFile filename WriteMode) hClose
          (\h -> hPutStrLn h contents)
```

What does this program do? First, it issues a short string of instructions and reads a command. It then performs a **`case`** switch on the command and checks first to see if the first character is a `q.' If it is, it returns a value of unit type.

Note

The `return` function is a function that takes a value of type `a` and returns an action of type `IO a`. Thus, the type of `return ()` is `IO ()`.

If the first character of the command wasn't a `q,' the program checks to see if it was an 'r' followed by some string that is bound to the variable `filename`. It then tells you that it's reading the file, does the read and runs `doLoop` again. The check for `w' is nearly identical. Otherwise, it matches _, the wildcard character, and loops to `doLoop`.

The `doRead` function uses the `bracket` function to make sure there are no problems reading the file. It opens a file in `ReadMode`, reads its contents and prints the first 100 characters (the `take` function takes an integer n and a list and returns the first n elements of the list).

The `doWrite` function asks for some text, reads it from the keyboard, and then writes it to the specified file.

Note

Both `doRead` and `doWrite` could have been made simpler by using `readFile` and `writeFile`, but they were written in the extended fashion to show how the more complex functions are used.

The program has one major problem: it will die if you try to read a file that doesn't already exist or if you specify some bad filename like `*\bs^#_@`. You may think that the calls to `bracket` in `doRead` and `doWrite` should take care of this, but they don't. They only catch exceptions within the main body, not within the startup or shutdown functions (`openFile` and `hClose`, in these cases). To make this completely reliable, we would need a way to catch exceptions raised by `openFile`.

Exercises

Write a variation of our program so that it first asks whether the user wants to read from a file, write to a file, or quit. If the user responds with "quit", the program should exit. If they respond with "read", the program should ask them for a file name and then print that file to the screen (if the file doesn't exist, the program may crash). If they respond with "write", it should ask them for a file name and then ask them for text to write to the file, with "." signaling completion. All but the "." should be written to the file.

For example, running this program might produce:

```
Do you want to [read] a file, [write] a file, or [quit]?
read
Enter a file name to read:
foo
...contents of foo...
Do you want to [read] a file, [write] a file, or [quit]?
write
Enter a file name to write:
foo
Enter text (dot on a line by itself to end):
this is some
text for
```

```
foo
.
Do you want to [read] a file, [write] a file, or [quit]?
read
Enter a file name to read:
foo
this is some
text for
foo
Do you want to [read] a file, [write] a file, or [quit]?
blech
I don't understand the command blech.
Do you want to [read] a file, [write] a file, or [quit]?
quit
Goodbye!
```