Real World Haskell by Bryan O'Sullivan, Don Stewart, and John Goerzen

Prev Next

Chapter 7. I/O

Table of Contents

It should be obvious that most, if not all, programs are devoted to gathering data from outside, processing it, and providing results back to the outside world. That is, input and output are key. No comments

Haskell's I/O system is powerful and expressive. It is easy to work with and important to understand. Haskell strictly separates pure code from code that could cause things to occur in the world. That is, it provides a complete isolation from side-effects in pure code. Besides helping programmers to reason about the correctness of their code, it also permits compilers to automatically introduce optimizations and parallelism. No comments

We'll begin this chapter with simple, standard-looking I/O in Haskell. Then we'll discuss some of the more powerful options as well as provide more detail on how I/O fits into the pure, lazy, functional Haskell world. No comments

Classic I/O in Haskell

Let's get started with I/O in Haskell by looking at a program that looks surprisingly similar to I/O in other languages such as C or Perl. No comments

```
-- file: ch07/basicio.hs
main = do
       putStrLn "Greetings!  What is your name?"
       inpStr <- getLine
       putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!"
```

21 comments

You can compile this program to a standalone executable, run it with **runghc**, or invoke `main` from within **ghci**. Here's a sample session using **runghc**: No comments

```
$ runghc basicio.hs
Greetings!  What is your name?
John
Welcome to Haskell, John!
```

No comments

That's a fairly simple, obvious result. You can see that `putStrLn` writes out a `String`, followed by an end-of-line character. `getLine` reads a line from standard input. The `<-` syntax may be new to you. Put simply, that binds the result from executing an I/O action to a name. [15] We use the simple list concatenation operator `++` to join the input string with our own text. No comments

Let's take a look at the types of `putStrLn` and `getLine`. You can find that information in the library reference, or just ask **ghci**: No comments

```
ghci> :type putStrLn
putStrLn :: String -> IO ()
ghci> :type getLine
getLine :: IO String
```

No comments

Notice that both of these types have `IO` in their return value. That is your key to knowing that they may have side effects, or that they may return different values even when called with the same arguments, or both. The type of `putStrLn` looks like a function. It takes a parameter of type `String` and returns value of type `IO ()`. Just what is an `IO ()` though? No comments

Anything that is type `IO` *`something`* is an I/O *action*. You can store it and nothing will happen. I could say `writefoo = putStrLn "foo"` and nothing happens right then. But if I later use `writefoo` in the middle of another I/O action, the `writefoo` action will be executed when its parent action is executed -- I/O actions can be glued together to form bigger I/O actions. The `()` is an empty tuple (pronounced "unit"), indicating that there is no return value from `putStrLn`. This is similar to `void` in Java or C.[16]
No comments

Tip

Actions can be created, assigned, and passed anywhere. However, they may only be performed (executed) from within another I/O action. No comments

Let's look at this with **ghci**: No comments

```
ghci> let writefoo = putStrLn "foo"
ghci> writefoo
foo
```

No comments

In this example, the output `foo` is not a return value from `putStrLn`. Rather, it's the side effect of `putStrLn` actually writing `foo` to the terminal. No comments

Notice one other thing: **ghci** actually executed `writefoo`. This means that, when given an I/O action, **ghci** will perform it for you on the spot. No comments

What Is An I/O Action?

Actions: No comments

- Have the type `IO` *`t`* No comments
- Are first-class values in Haskell and fit seamlessly with Haskell's type system No comments
- Produce an effect when *performed*, but not when *evaluated*. That is, they only produce an effect when called by something else in an I/O context. No comments
- Any expression may produce an action as its value, but the action will not perform I/O until it is executed inside another I/O action (or it is `main`) No comments
- Performing (executing) an action of type `IO t` may perform I/O and will ultimately deliver a result of type `t` No comments

The type of `getLine` may look strange to you. It looks like a value, rather than a function. And in fact, that is one way to look at it: `getLine` is storing an I/O action. When that action is performed, you get a `String`. The `<-` operator is used to "pull out" the result from performing an I/O action and store it in a variable. No comments

`main` itself is an I/O action with type `IO ()`. You can only perform I/O actions from within other I/O actions. All I/O in Haskell programs is driven from the top at `main`, which is where execution of every Haskell program begins. This, then, is the mechanism that provides isolation from side effects in Haskell: you perform I/O in your `IO` actions, and call pure (non-I/O) functions from there. Most Haskell code is pure; the I/O actions perform I/O and call that pure code. No comments

`do` is a convenient way to define a sequence of actions. As you'll see later, there are other ways. When you use `do` in this way, indentation is significant; make sure you line up your actions properly.
No comments

You only need to use `do` if you have more than one action that you need to perform. The value of a `do` block is the value of the last action executed. For a complete description of `do` syntax, see the section called "Desugaring of do blocks". No comments

Let's consider an example of calling pure code from within an I/O action: No comments

```
-- file: ch07/callingpure.hs
name2reply :: String -> String
name2reply name =
    "Pleased to meet you, " ++ name ++ ".\n" ++
    "Your name contains " ++ charcount ++ " characters."
    where charcount = show (length name)

main :: IO ()
main = do
       putStrLn "Greetings once again.  What is your name?"
       inpStr <- getLine
       let outStr = name2reply inpStr
       putStrLn outStr
```

5 comments

Notice the `name2reply` function in this example. It is a regular Haskell function and obeys all the rules we've told you about: it always returns the same result when given the same input, it has no side effects, and it operates lazily. It uses other Haskell functions: `(++)`, `show`, and `length`. No comments

Down in `main`, we bind the result of `name2reply inpStr` to `outStr`. When you're working in a `do` block, you use `<-` to get results from `IO` actions and `let` to get results from pure code. When used in a `do` block, you should not put `in` after your `let` statement. No comments

You can see here how we read the person's name from the keyboard. Then, that data got passed to a pure function, and its result was printed. In fact, the last two lines of `main` could have been replaced with `putStrLn (name2reply inpStr)`. So, while `main` did have side effects—it caused things to appear on the terminal, for instance—`name2reply` did not and could not. That's because `name2reply` is a pure function, not an action. No comments

Let's examine this with **ghci**: No comments

```
ghci> :load callingpure.hs
[1 of 1] Compiling Main             ( callingpure.hs, interpreted )
Ok, modules loaded: Main.
ghci> name2reply "John"
"Pleased to meet you, John.\nYour name contains 4 characters."
ghci> putStrLn (name2reply "John")
Pleased to meet you, John.
Your name contains 4 characters.
```

No comments

The `\n` within the string is the end-of-line (newline) character, which causes the terminal to begin a new line in its output. Just calling `name2reply "John"` in **ghci** will show you the `\n` literally, because it is using `show` to display the return value. But using `putStrLn` sends it to the terminal, and the terminal interprets `\n` to start a new line. No comments

What do you think will happen if you simply type **`main`** at the **ghci** prompt? Give it a try. No comments

After looking at these example programs, you may be wondering if Haskell is really imperative rather than pure, lazy, and functional. Some of these examples look like a sequence of actions to be followed in order. There's more to it than that, though. We'll discuss that question later in this chapter in the section called "Is Haskell Really Imperative?" and the section called "Lazy I/O". No comments

Pure vs. I/O

As a way to help with understanding the differences between pure code and I/O, here's a comparison table. When we speak of pure code, we are talking about Haskell functions that always return the same result when given the same input and have no side effects. In Haskell, only the execution of I/O actions avoid these rules. No comments

Table 7.1. Pure vs. Impure

Pure	Impure
Always produces the same result when given the	May produce different results for the same

Pure	Impure
same parameters	parameters
Never has side effects	May have side effects
Never alters state	May alter the global state of the program, system, or world

Why Purity Matters

In this section, we've discussed how Haskell draws a clear distinction between pure code and I/O actions. Most languages don't draw this distinction. In languages such as C or Java, there is no such thing as a function that is guaranteed by the compiler to always return the same result for the same arguments, or a function that is guaranteed to never have side effects. The only way to know if a given function has side effects is to read its documentation and hope that it's accurate. No comments

Many bugs in programs are caused by unanticipated side effects. Still more are caused by misunderstanding circumstances in which functions may return different results for the same input. As multithreading and other forms of parallelism grow increasingly common, it becomes more difficult to manage global side effects. No comments

Haskell's method of isolating side effects into I/O actions provides a clear boundary. You can always know which parts of the system may alter state and which won't. You can always be sure that the pure parts of your program aren't having unanticipated results. This helps you to think about the program. It also helps the compiler to think about it. Recent versions of **ghc**, for instance, can provide a level of automatic parallelism for the pure parts of your code -- something of a holy grail for computing. No comments

For more discussion on this topic, refer to the section called “Side Effects with Lazy I/O”. No comments

Working With Files and Handles

So far, you've seen how to interact with the user at the computer's terminal. Of course, you'll often need to manipulate specific files. That's easy to do, too. No comments

Haskell defines quite a few basic functions for I/O, many of which are similar to functions seen in other programming languages. The library reference for `System.IO` provides a good summary of all the basic I/O functions, should you need one that we aren't touching upon here. No comments

You will generally begin by using `openFile`, which will give you a file `Handle`. That `Handle` is then used to perform specific operations on the file. Haskell provides functions such as `hPutStrLn` that work just like `putStrLn` but take an additional argument—a `Handle`—that specifies which file to operate upon. When you're done, you'll use `hClose` to close the `Handle`. These functions are all defined in `System.IO`, so you'll need to import that module when working with files. There are "h" functions corresponding to virtually all of the non-"h" functions; for instance, there is `print` for printing to the screen and `hPrint` for printing to a file. No comments

Let's start with an imperative way to read and write files. This should seem similar to a `while` loop that you may find in other languages. This isn't the best way to write it in Haskell; later, you'll see examples of more Haskellish approaches. No comments

```
-- file: ch07/toupper-imp.hs
import System.IO
import Data.Char(toUpper)

main :: IO ()
main = do
       inh <- openFile "input.txt" ReadMode
       outh <- openFile "output.txt" WriteMode
       mainloop inh outh
       hClose inh
       hClose outh

mainloop :: Handle -> Handle -> IO ()
mainloop inh outh =
    do ineof <- hIsEOF inh
```

```
            if ineof
                then return ()
                else do inpStr <- hGetLine inh
                        hPutStrLn outh (map toUpper inpStr)
                        mainloop inh outh
```

7 comments

Like every Haskell program, execution of this program begins with `main`. Two files are opened: `input.txt` is opened for reading, and `output.txt` is opened for writing. Then we call `mainloop` to process the file. No comments

`mainloop` begins by checking to see if we're at the end of file (EOF) for the input. If not, we read a line from the input. We write out the same line to the output, after first converting it to uppercase. Then we recursively call `mainloop` again to continue processing the file.[17] No comments

Notice that `return` call. This is not really the same as `return` in C or Python. In those languages, `return` is used to terminate execution of the current function immediately, and to return a value to the caller. In Haskell, `return` is the opposite of `<-`. That is, `return` takes a pure value and wraps it inside `IO`. Since every I/O action must return some `IO` type, if your result came from pure computation, you must use `return` to wrap it in `IO`. As an example, if `7` is an `Int`, then `return 7` would create an action stored in a value of type `IO Int`. When executed, that action would produce the result `7`. For more details on `return`, see the section called "The True Nature of Return". No comments

Let's try running the program. We've got a file named `input.txt` that looks like this: No comments

```
This is ch08/input.txt

Test Input
I like Haskell
Haskell is great
I/O is fun

123456789
```

No comments

Now, you can use `runghc toupper-imp.hs` and you'll find `output.txt` in your directory. It should look like this: No comments

```
THIS IS CH08/INPUT.TXT

TEST INPUT
I LIKE HASKELL
HASKELL IS GREAT
I/O IS FUN

123456789
```

No comments

More on openFile

Let's use **ghci** to check on the type of `openFile`: No comments

```
ghci> :module System.IO
ghci> :type openFile
openFile :: FilePath -> IOMode -> IO Handle
```

3 comments

`FilePath` is simply another name for `String`. It is used in the types of I/O functions to help clarify that the parameter is being used as a filename, and not as regular data. No comments

`IOMode` specifies how the file is to be managed. The possible values for `IOMode` are listed in Table 7.2, "Possible IOMode Values". No comments

FIXME: check formatting on this table for final book; openjade doesn't render it well

Table 7.2. Possible IOMode Values

IOMode	Can read?	Can write?	Starting position	Notes
ReadMode	Yes	No	Beginning of file	File must exist already
WriteMode	No	Yes	Beginning of file	File is truncated (completely emptied) if it already existed
ReadWriteMode	Yes	Yes	Beginning of file	File is created if it didn't exist; otherwise, existing data is left intact
AppendMode	No	Yes	End of file	File is created if it didn't exist; otherwise, existing data is left intact.

While we are mostly working with text examples in this chapter, binary files can also be used in Haskell. If you are working with a binary file, you should use `openBinaryFile` instead of `openFile`. Operating systems such as Windows process files differently if they are opened as binary instead of as text. On operating systems such as Linux, both `openFile` and `openBinaryFile` perform the same operation. Nevertheless, for portability, it is still wise to always use `openBinaryFile` if you will be dealing with binary data. No comments

Closing Handles

You've already seen that `hClose` is used to close file handles. Let's take a moment and think about why this is important. No comments

As you'll see in the section called "Buffering", Haskell maintains internal buffers for files. This provides an important performance boost. However, it means that until you call `hClose` on a file that is open for writing, your data may not be flushed out to the operating system. No comments

Another reason to make sure to `hClose` files is that open files take up resources on the system. If your program runs for a long time, and opens many files but fails to close them, it is conceivable that your program could even crash due to resource exhaustion. All of this is no different in Haskell than in other languages. No comments

When a program exits, Haskell will normally take care of closing any files that remain open. However, there are some circumstances in which this may not happen[18], so once again, it is best to be responsible and call `hClose` all the time. No comments

Haskell provides several tools for you to use to easily ensure this happens, regardless of whether errors are present. You can read about `finally` in the section called "Extended Example: Functional I/O and Temporary Files" and `bracket` in the section called "The acquire-use-release cycle". No comments

Seek and Tell

When reading and writing from a `Handle` that corresponds to a file on disk, the operating system maintains an internal record of the current position. Each time you do another read, the operating system returns the next chunk of data that begins at the current position, and increments the position to reflect the data that you read. No comments

You can use `hTell` to find out your current position in the file. When the file is initially created, it is empty and your position will be 0. After you write out 5 bytes, your position will be 5, and so on. `hTell` takes a `Handle` and returns an `IO Integer` with your position. No comments

The companion to `hTell` is `hSeek`. `hSeek` lets you change the file position. It takes three parameters: a `Handle`, a `SeekMode`, and a position. No comments

`SeekMode` can be one of three different values, which specify how the given position is to be interpreted. `AbsoluteSeek` means that the position is a precise location in the file. This is the same kind of information that `hTell` gives you. `RelativeSeek` means to seek from the current position. A positive number requests going forwards in the file, and a negative number means going backwards. Finally, `SeekFromEnd` will seek to the specified number of bytes before the end of the file. `hSeek handle SeekFromEnd 0` will take you to the end of the file. For an example of `hSeek`, refer to the section called "Extended Example: Functional I/O and Temporary Files". No comments

Not all `Handle`s are seekable. A `Handle` usually corresponds to a file, but it can also correspond to other things such as network connections, tape drives, or terminals. You can use `hIsSeekable` to see if a given `Handle` is seekable. No comments

Standard Input, Output, and Error

Earlier, we pointed out that for each non-"h" function, there is usually also a corresponding "h" function that works on any `Handle`. In fact, the non-"h" functions are nothing more than shortcuts for their "h" counterparts. No comments

There are three pre-defined `Handle`s in `System.IO`. These `Handle`s are always available for your use. They are `stdin`, which corresponds to standard input; `stdout` for standard output; and `stderr` for standard error. Standard input normally refers to the keyboard, standard output to the monitor, and standard error also normally goes to the monitor. No comments

Functions such as `getLine` can thus be trivially defined like this: No comments

```
getLine = hGetLine stdin
putStrLn = hPutStrLn stdout
print = hPrint stdout
```

No comments

Tip

We're using partial application here. If this isn't making sense, consult the section called "Partial function application and currying" for a refresher. No comments

Earlier, we told you what the three standard file handles "normally" correspond to. That's because some operating systems let you redirect the file handles to come from (or go to) different places —files, devices, or even other programs. This feature is used extensively in shell scripting on POSIX (Linux, BSD, Mac) operating systems, but can also be used on Windows. No comments

It often makes sense to use standard input and output instead of specific files. This lets you interact with a human at the terminal. But it also lets you work with input and output files—or even combine your code with other programs—if that's what's requested.[19] No comments

As an example, we can provide input to `callingpure.hs` in advance like this: No comments

```
$ echo John|runghc callingpure.hs
Greetings once again.  What is your name?
Pleased to meet you, John.
Your name contains 4 characters.
```

No comments

While `callingpure.hs` was running, it did not wait for input at the keyboard; instead it received `John` from the `echo` program. Notice also that the output didn't contain the word `John` on a separate line as it did when this program was run at the keyboard. The terminal normally echoes everything you type back to you, but that is technically input, and is not included in the output stream. No comments

Deleting and Renaming Files

So far in this chapter, we've discussed the contents of the files. Let's now talk a bit about the files themselves. No comments

`System.Directory` provides two functions you may find useful. `removeFile` takes a single argument, a filename, and deletes that file.[20] `renameFile` takes two filenames: the first is the old name and the second is the new name. If the new filename is in a different directory, you can also think of this as a move. The old filename must exist prior to the call to `renameFile`. If the new file already exists, it is removed before the rename takes place. No comments

Like many other functions that take a filename, if the "old" name doesn't exist, `renameFile` will raise an exception. More information on exception handling can be found in Chapter 19, *Error handling*. No comments

There are many other functions in `System.Directory` for doing things such as creating and removing directories, finding lists of files in directories, and testing for file existence. These are discussed in the section called "Directory and File Information". No comments

Temporary Files

Programmers frequently need temporary files. These files may be used to store large amounts of data needed for computations, data to be used by other programs, or any number of other uses. No comments

While you could craft a way to manually open files with unique names, the details of doing this in a secure way differ from platform to platform. Haskell provides a convenient function called `openTempFile` (and a corresponding `openBinaryTempFile`) to handle the difficult bits for you. No comments

`openTempFile` takes two parameters: the directory in which to create the file, and a "template" for naming the file. The directory could simply be `"."` for the current working directory. Or you could use `System.Directory.getTemporaryDirectory` to find the best place for temporary files on a given machine. The template is used as the basis for the file name; it will have some random characters added to it to ensure that the result is truly unique. It guarantees that it will be working on a unique filename, in fact. No comments

The return type of `openTempFile` is `IO (FilePath, Handle)`. The first part of the tuple is the name of the file created, and the second is a `Handle` opened in `ReadWriteMode` over that file. When you're done with the file, you'll want to `hClose` it and then call `removeFile` to delete it. See the following example for a sample function to use. No comments

Extended Example: Functional I/O and Temporary Files

Here's a larger example that puts together some concepts from this chapter, from some earlier chapters, and a few you haven't seen yet. Take a look at the program and see if you can figure out what it does and how it works. No comments

```
-- file: ch07/tempfile.hs
import System.IO
import System.Directory(getTemporaryDirectory, removeFile)
import System.IO.Error(catch)
import Control.Exception(finally)

-- The main entry point.  Work with a temp file in myAction.
main :: IO ()
main = withTempFile "mytemp.txt" myAction

{- The guts of the program.  Called with the path and handle of a temporary
   file.  When this function exits, that file will be closed and deleted
   because myAction was called from withTempFile. -}
myAction :: FilePath -> Handle -> IO ()
myAction tempname temph =
    do -- Start by displaying a greeting on the terminal
       putStrLn "Welcome to tempfile.hs"
       putStrLn $ "I have a temporary file at " ++ tempname

       -- Let's see what the initial position is
       pos <- hTell temph
       putStrLn $ "My initial position is " ++ show pos

       -- Now, write some data to the temporary file
       let tempdata = show [1..10]
       putStrLn $ "Writing one line containing " ++
                  show (length tempdata) ++ " bytes: " ++
                  tempdata
       hPutStrLn temph tempdata

       -- Get our new position.  This doesn't actually modify pos
       -- in memory, but makes the name "pos" correspond to a different
       -- value for the remainder of the "do" block.
       pos <- hTell temph
       putStrLn $ "After writing, my new position is " ++ show pos
```

```
        -- Seek to the beginning of the file and display it
        putStrLn $ "The file content is: "
        hSeek temph AbsoluteSeek 0

        -- hGetContents performs a lazy read of the entire file
        c <- hGetContents temph

        -- Copy the file byte-for-byte to stdout, followed by \n
        putStrLn c

        -- Let's also display it as a Haskell literal
        putStrLn $ "Which could be expressed as this Haskell literal:"
        print c

{- This function takes two parameters: a filename pattern and another
   function.  It will create a temporary file, and pass the name and Handle
   of that file to the given function.

   The temporary file is created with openTempFile.  The directory is the one
   indicated by getTemporaryDirectory, or, if the system has no notion of
   a temporary directory, "." is used.  The given pattern is passed to
   openTempFile.

   After the given function terminates, even if it terminates due to an
   exception, the Handle is closed and the file is deleted. -}
withTempFile :: String -> (FilePath -> Handle -> IO a) -> IO a
withTempFile pattern func =
    do -- The library ref says that getTemporaryDirectory may raise on
       -- exception on systems that have no notion of a temporary directory.
       -- So, we run getTemporaryDirectory under catch.  catch takes
       -- two functions: one to run, and a different one to run if the
       -- first raised an exception.  If getTemporaryDirectory raised an
       -- exception, just use "." (the current working directory).
       tempdir <- catch (getTemporaryDirectory) (\_ -> return ".")
       (tempfile, temph) <- openTempFile tempdir pattern

       -- Call (func tempfile temph) to perform the action on the temporary
       -- file.  finally takes two actions.  The first is the action to run.
       -- The second is an action to run after the first, regardless of
       -- whether the first action raised an exception.  This way, we ensure
       -- the temporary file is always deleted.  The return value from finally
       -- is the first action's return value.
       finally (func tempfile temph)
               (do hClose temph
                   removeFile tempfile)
```

27 comments

Let's start looking at this program from the end. The `withTempFile` function demonstrates that Haskell doesn't forget its functional nature when I/O is introduced. This function takes a `String` and another function. The function passed to `withTempFile` is invoked with the name and `Handle` of a temporary file. When that function exits, the temporary file is closed and deleted. So even when dealing with I/O, we can still find the idiom of passing functions as parameters to be convenient. Lisp programmers might find our `withTempFile` function similar to Lisp's `with-open-file` function. No comments

There is some exception handling going on to make the program more robust in the face of errors. You normally want the temporary files to be deleted after processing completes, even if something went wrong. So we make sure that happens. For more on exception handling, see Chapter 19, *Error handling*. No comments

Let's return to the start of the program. `main` is defined simply as `withTempFile "mytemp.txt" myAction`. `myAction`, then, will be invoked with the name and `Handle` of the temporary file. No comments

`myAction` displays some information to the terminal, writes some data to the file, seeks to the beginning of the file, and reads the data back with `hGetContents`.[21] It then displays the contents of the file byte-for-byte, and also as a Haskell literal via `print c`. That's the same as `putStrLn (show c)`. No comments

Let's look at the output: No comments

```
$ runhaskell tempfile.hs
Welcome to tempfile.hs
I have a temporary file at /tmp/mytemp8572.txt
My initial position is 0
Writing one line containing 22 bytes: [1,2,3,4,5,6,7,8,9,10]
After writing, my new position is 23
The file content is:
[1,2,3,4,5,6,7,8,9,10]
```

```
Which could be expressed as this Haskell literal:
"[1,2,3,4,5,6,7,8,9,10]\n"
```

No comments

Every time you run this program, your temporary file name should be slightly different since it contains a randomly-generated component. Looking at this output, there are a few questions that might occur to you: No comments

1. Why is your position 23 after writing a line with 22 bytes? No comments
2. Why is there an empty line after the file content display? No comments
3. Why is there a `\n` at the end of the Haskell literal display? No comments

You might be able to guess that the answers to all three questions are related. See if you can work out the answers for a moment. If you need some help, here are the explanations: No comments

1. That's because we used `hPutStrLn` instead of `hPutStr` to write the data. `hPutStrLn` always terminates the line by writing a `\n` at the end, which didn't appear in `tempdata`. No comments
2. We used `putStrLn c` to display the file contents `c`. Because the data was written originally with `hPutStrLn`, `c` ends with the newline character, and `putStrLn` adds a second newline character. The result is a blank line. No comments
3. The `\n` is the newline character from the original `hPutStrLn`. No comments

As a final note, the byte counts may be different on some operating systems. Windows, for instance, uses the two-byte sequence `\r\n` as the end-of-line marker, so you may see differences on that platform. No comments

Lazy I/O

So far in this chapter, you've seen examples of fairly traditional I/O. Each line, or block of data, is requested individually and processed individually. No comments

Haskell has another approach available to you as well. Since Haskell is a lazy language, meaning that any given piece of data is only evaluated when its value must be known, there are some novel ways of approaching I/O. No comments

hGetContents

One novel way to approach I/O is the `hGetContents` function.[22] `hGetContents` has the type `Handle -> IO String`. The `String` it returns represents all of the data in the file given by the `Handle`.[23] No comments

In a strictly-evaluated language, using such a function is often a bad idea. It may be fine to read the entire contents of a 2KB file, but if you try to read the entire contents of a 500GB file, you are likely to crash due to lack of RAM to store all that data. In these languages, you would traditionally use mechanisms such as loops to process the file's entire data. No comments

But `hGetContents` is different. The `String` it returns is evaluated lazily. At the moment you call `hGetContents`, nothing is actually read. Data is only read from the `Handle` as the elements (characters) of the list are processed. As elements of the `String` are no longer used, Haskell's garbage collector automatically frees that memory. All of this happens completely transparently to you. And since you have what looks like—and, really, is—a pure `String`, you can pass it to pure (non-`IO`) code. No comments

Let's take a quick look at an example. Back in the section called "Working With Files and Handles", you saw an imperative program that converted the entire content of a file to uppercase. Its imperative algorithm was similar to what you'd see in many other languages. Here now is the much simpler algorithm that exploits lazy evaluation: No comments

```
-- file: ch07/toupper-lazy1.hs
import System.IO
import Data.Char(toUpper)
```

```
main :: IO ()
main = do
       inh <- openFile "input.txt" ReadMode
       outh <- openFile "output.txt" WriteMode
       inpStr <- hGetContents inh
       let result = processData inpStr
       hPutStr outh result
       hClose inh
       hClose outh

processData :: String -> String
processData = map toUpper
```

4 comments

Notice that `hGetContents` handled *all* of the reading for us. Also, take a look at `processData`. It's a pure function since it has no side effects and always returns the same result each time it is called. It has no need to know—and no way to tell—that its input is being read lazily from a file in this case. It can work perfectly well with a 20-character literal or a 500GB data dump on disk. No comments

You can even verify that with **ghci**: No comments

```
ghci> :load toupper-lazy1.hs
[1 of 1] Compiling Main             ( toupper-lazy1.hs, interpreted )
Ok, modules loaded: Main.
ghci> processData "Hello, there!  How are you?"
"HELLO, THERE!  HOW ARE YOU?"
ghci> :type processData
processData :: String -> String
ghci> :type processData "Hello!"
processData "Hello!" :: String
```

No comments

Warning

If we had tried to hang on to `inpStr` in the above example, past the one place where it was used (the call to `processData`), the program would have lost its memory efficiency. That's because the compiler would have been forced to keep `inpStr`'s value in memory for future use. Here it knows that `inpStr` will never be reused, and frees the memory as soon as it is done with it. Just remember: memory is only freed after its last use. No comments

This program was a bit verbose to make it clear that there was pure code in use. Here's a bit more concise version, which we will build on in the next examples: No comments

```
-- file: ch07/toupper-lazy2.hs
import System.IO
import Data.Char(toUpper)

main = do
       inh <- openFile "input.txt" ReadMode
       outh <- openFile "output.txt" WriteMode
       inpStr <- hGetContents inh
       hPutStr outh (map toUpper inpStr)
       hClose inh
       hClose outh
```

1 comment

You are not required to ever consume all the data from the input file when using `hGetContents`. Whenever the Haskell system determines that the entire string `hGetContents` returned can be garbage collected —which means it will never again be used—the file is closed for you automatically. The same principle applies to data read from the file. Whenever a given piece of data will never again be needed, the Haskell environment releases the memory it was stored within. Strictly speaking, we wouldn't have to call `hClose` at all in this example program. However, it is still a good practice to get into, as later changes to a program could make the call to `hClose` important. No comments

Warning

When using `hGetContents`, it is important to remember that even though you may never again explicitly reference `Handle` directly in the rest of the program, you must not close the `Handle` until you have finished consuming its results via `hGetContents`. Doing so would cause you to miss on some or all of the file's data. Since Haskell is lazy, you generally can assume that you have consumed input only after you have output the result of the computations involving the input. No comments

readFile and writeFile

Haskell programmers use `hGetContents` as a filter quite often. They read from one file, do something to the data, and write the result out elsewhere. This is so common that there are some shortcuts for doing it. `readFile` and `writeFile` are shortcuts for working with files as strings. They handle all the details of opening files, closing files, reading data, and writing data. `readFile` uses `hGetContents` internally. No comments

Can you guess the Haskell types of these functions? Let's check with **ghci**: No comments

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

3 comments

Now, here's an example program that uses `readFile` and `writeFile`: No comments

```
-- file: ch07/toupper-lazy3.hs
import Data.Char(toUpper)

main = do
       inpStr <- readFile "input.txt"
       writeFile "output.txt" (map toUpper inpStr)
```

2 comments

Look at that—the guts of the program take up only two lines! `readFile` returned a lazy `String`, which we stored in `inpStr`. We then took that, processed it, and passed it to `writeFile` for writing. No comments

Neither `readFile` nor `writeFile` ever provide a `Handle` for you to work with, so there is nothing to ever `hClose`. `readFile` uses `hGetContents` internally, and the underlying `Handle` will be closed when the returned `String` is garbage-collected or all the input has been consumed. `writeFile` will close its underlying `Handle` when the entire `String` supplied to it has been written. No comments

A Word On Lazy Output

By now, you should understand how lazy input works in Haskell. But what about laziness during output? No comments

As you know, nothing in Haskell is evaluated before its value is needed. Since functions such as `writeFile` and `putStr` write out the entire `String` passed to them, that entire `String` must be evaluated. So you are guaranteed that the argument to `putStr` will be evaluated in full.[24] No comments

But what does that mean for laziness of the input? In the examples above, will the call to `putStr` or `writeFile` force the entire input string to be loaded into memory at once, just to be written out? No comments

The answer is no. `putStr` (and all the similar output functions) write out data as it becomes available. They also have no need for keeping around data already written, so as long as nothing else in the program needs it, the memory can be freed immediately. In a sense, you can think of the `String` between `readFile` and `writeFile` as a pipe linking the two. Data goes in one end, is transformed some way, and flows back out the other. No comments

You can verify this yourself by generating a large `input.txt` for `toupper-lazy3.hs`. It may take a bit to process, but you should see a constant—and low—memory usage while it is being processed. No comments

interact

You learned that `readFile` and `writeFile` address the common situation of reading from one file, making a conversion, and writing to a different file. There's a situation that's even more common than that: reading from standard input, making a conversion, and writing the result to standard output. For that situation, there is a function called `interact`. The type of `interact` is `(String -> String) -> IO ()`. That is, it takes one argument: a function of type `String -> String`. That function is passed the result of `getContents`—that is, standard input read lazily. The result of that function is sent to standard output. No comments

We can convert our example program to operate on standard input and standard output by using `interact`. Here's one way to do that: No comments

```
-- file: ch07/toupper-lazy4.hs
import Data.Char(toUpper)

main = interact (map toUpper)
```

4 comments

Look at that—*one* line of code to achieve our transformation! To achieve the same effect as with the previous examples, you could run this one like this: No comments

```
$ runghc toupper-lazy4.hs < input.txt > output.txt
```

No comments

Or, if you'd like to see the output printed to the screen, you could type: No comments

```
$ runghc toupper-lazy4.hs < input.txt
```

No comments

If you want to see that Haskell output truly does write out chunks of data as soon as they are received, run `runghc toupper-lazy4.hs` without any other command-line parameters. You should see each character echoed back out as soon as you type it, but in uppercase. Buffering may change this behavior; see the section called “Buffering” later in this chapter for more on buffering. If you see each line echoed as soon as you type it, or even nothing at all for awhile, buffering is causing this behavior. No comments

You can also write simple interactive programs using `interact`. Let's start with a simple example: adding a line of text before the uppercase output. No comments

```
-- file: ch07/toupper-lazy5.hs
import Data.Char(toUpper)

main = interact (map toUpper . (++) "Your data, in uppercase, is:\n\n")
```

6 comments

> **Tip**
>
> If the use of the `.` operator is confusing, you might wish to refer to the section called “Code reuse through composition”. No comments

Here we add a string at the beginning of the output. Can you spot the problem, though? No comments

Since we're calling `map` on the *result* of `(++)`, that header itself will appear in uppercase. We can fix that in this way: No comments

```
-- file: ch07/toupper-lazy6.hs
```

```
import Data.Char(toUpper)

main = interact ((++) "Your data, in uppercase, is:\n\n" .
                 map toUpper)
```

2 comments

This moved the header outside of the `map`. No comments

Filters with interact

Another common use of `interact` is filtering. Let's say that you want to write a program that reads a file and prints out every line that contains the character "a". Here's how you might do that with `interact`: No comments

```
-- file: ch07/filter.hs
main = interact (unlines . filter (elem 'a') . lines)
```

1 comment

This may have introduced three functions that you aren't familiar with yet. Let's inspect their types with **ghci**: No comments

```
ghci> :type lines
lines :: String -> [String]
ghci> :type unlines
unlines :: [String] -> String
ghci> :type elem
elem :: (Eq a) => a -> [a] -> Bool
```

No comments

Can you guess what these functions do just by looking at their types? If not, you can find them explained in the section called “Warming up: portably splitting lines of text” and the section called “Special string-handling functions”. You'll frequently see `lines` and `unlines` used with I/O. Finally, `elem` takes a element and a list and returns `True` if that element occurs anywhere in the list. No comments

Try running this over our standard example input: No comments

```
$ runghc filter.hs < input.txt
I like Haskell
Haskell is great
```

No comments

Sure enough, you got back the two lines that contain an "a". Lazy filters are a powerful way to use Haskell. When you think about it, a filter—such as the standard Unix program **grep**—sounds a lot like a function. It takes some input, applies some computation, and generates a predictable output. No comments

The IO Monad

You've seen a number of examples of I/O in Haskell by this point. Let's take a moment to step back and think about how I/O relates to the broader Haskell language. No comments

Since Haskell is a pure language, if you give a certain function a specific argument, the function will return the same result every time you give it that argument. Moreover, the function will not change anything about the program's overall state. No comments

You may be wondering, then, how I/O fits into this picture. Surely if you want to read a line of input from the keyboard, the function to read input can't possibly return the same result every time it is run, right? Moreover, I/O is all about changing state. I/O could cause pixels on a terminal to light up, to cause paper to start coming out of a printer, or even to cause a package to be shipped from a warehouse on a different continent. I/O doesn't just change the state of a program. You can think of I/O as changing the state of the world. No comments

Actions

Most languages do not make a distinction between a pure function and an impure one. Haskell has functions in the mathematical sense: they are purely computations which cannot be altered by anything external. Moreover, the computation can be performed at any time—or even never, if its result is never needed. No comments

Clearly, then, we need some other tool to work with I/O. That tool in Haskell is called *actions*. Actions resemble functions. They do nothing when they are defined, but perform some task when they are invoked. I/O actions are defined within the `IO` monad. Monads are a powerful way of chaining functions together purely and are covered in [Chapter 14, *Monads*](#). It's not necessary to understand monads in order to understand I/O. Just understand that the result type of actions is "tagged" with `IO`. Let's take a look at some types: No comments

```
ghci> :type putStrLn
putStrLn :: String -> IO ()
ghci> :type getLine
getLine :: IO String
```

No comments

The type of `putStrLn` is just like any other function. The function takes one parameter and returns an `IO ()`. This `IO ()` is the action. You can store and pass actions in pure code if you wish, though this isn't frequently done. An action doesn't do anything until it is invoked. Let's look at an example of this: No comments

```
-- file: ch07/actions.hs
str2action :: String -> IO ()
str2action input = putStrLn ("Data: " ++ input)

list2actions :: [String] -> [IO ()]
list2actions = map str2action

numbers :: [Int]
numbers = [1..10]

strings :: [String]
strings = map show numbers

actions :: [IO ()]
actions = list2actions strings

printitall :: IO ()
printitall = runall actions

-- Take a list of actions, and execute each of them in turn.
runall :: [IO ()] -> IO ()
runall [] = return ()
runall (firstelem:remainingelems) =
    do firstelem
       runall remainingelems

main = do str2action "Start of the program"
          printitall
          str2action "Done!"
```

4 comments

`str2action` is a function that takes one parameter and returns an `IO ()`. As you can see at the end of `main`, you could use this directly in another action and it will print out a line right away. Or, you can store—but not execute—the action from pure code. You can see an example of that in `list2actions`—we use `map` over `str2action` and return a list of actions, just like we would with other pure data. You can see that everything up through `printitall` is built up with pure tools. No comments

Although we define `printitall`, it doesn't get executed until its action is evaluated somewhere else. Notice in `main` how we use `str2action` as an I/O action to be executed, but earlier we used it outside of the I/O monad and assembled results into a list. No comments

You could think of it this way: every statement, except `let`, in a `do` block must yield an I/O action which will be executed. No comments

The call to `printitall` finally executes all those actions. Actually, since Haskell is lazy, the actions aren't generated until here either. No comments

When you run the program, your output will look like this: No comments

```
Data: Start of the program
Data: 1
Data: 2
Data: 3
Data: 4
Data: 5
Data: 6
Data: 7
Data: 8
Data: 9
Data: 10
Data: Done!
```

No comments

We can actually write this in a much more compact way. Consider this revision of the example: No comments

```
-- file: ch07/actions2.hs
str2message :: String -> String
str2message input = "Data: " ++ input

str2action :: String -> IO ()
str2action = putStrLn . str2message

numbers :: [Int]
numbers = [1..10]

main = do str2action "Start of the program"
          mapM_ (str2action . show) numbers
          str2action "Done!"
```

4 comments

Notice in `str2action` the use of the standard function composition operator. In `main`, there's a call to `mapM_`. This function is similar to `map`. It takes a function and a list. The function supplied to `mapM_` is an I/O action that is executed for every item in the list. `mapM_` throws out the result of the function, though you can use `mapM` to return a list of I/O results if you want them. Take a look at their types: No comments

```
ghci> :type mapM
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
ghci> :type mapM_
mapM_ :: (Monad m) => (a -> m b) -> [a] -> m ()
```

No comments

> **Tip**
>
> These functions actually work for more than just I/O; they work for any `Monad`. For now, wherever you see "M", just think "IO". Also, functions that end with an underscore typically discard their result. No comments

Why a `mapM` when we already have `map`? Because `map` is a pure function that returns a list. It doesn't—and can't—actually execute actions directly. `mapM` is a utility that lives in the `IO` monad and thus can actually execute the actions.[25] No comments

Going back to `main`, `mapM_` applies `(str2action . show)` to every element in `numbers`. `show` converts each number to a `String` and `str2action` converts each `String` to an action. `mapM_` combines these individual actions into one big action that prints out lines. No comments

Sequencing

`do` blocks are actually shortcut notations for joining together actions. There are two operators that you can use instead of `do` blocks: `>>` and `>>=`. Let's look at their types in **ghci**: No comments

```
ghci> :type (>>)
(>>) :: (Monad m) => m a -> m b -> m b
ghci> :type (>>=)
```

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

No comments

The `>>` operator sequences two actions together: the first action is performed, then the second. The result of the computation is the result of the second action. The result of the first action is thrown away. This is similar to simply having a line in a `do` block. You might write `putStrLn "line 1" >> putStrLn "line 2"` to test this out. It will print out two lines, discard the result from the first `putStrLn`, and provide the result from the second. No comments

The `>>=` operator runs an action, then passes its result to a function that returns an action. That second action is run as well, and the result of the entire expression is the result of that second action. As an example, you could write `getLine >>= putStrLn`, which would read a line from the keyboard and then display it back out. No comments

Let's re-write one of our examples to avoid `do` blocks. Remember this example from the start of the chapter? No comments

```
-- file: ch07/basicio.hs
main = do
       putStrLn "Greetings!  What is your name?"
       inpStr <- getLine
       putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!"
```

No comments

Let's write that without a `do` block: No comments

```
-- file: ch07/basicio-nodo.hs
main =
    putStrLn "Greetings!  What is your name?" >>
    getLine >>=
    (\inpStr -> putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!")
```

4 comments

The Haskell compiler internally performans a translation just like this when you define a `do` block. No comments

Tip

Forgetting how to use `\` (lambda expressions)? See the section called “Anonymous (lambda) functions”. No comments

The True Nature of Return

Earlier in this chapter, we mentioned that `return` is probably not what it looks like. Many languages have a keyword named `return` that aborts execution of a function immediately and returns a value to the caller. No comments

The Haskell `return` function is quite different. In Haskell, `return` is used to wrap data in a monad. When speaking about I/O, `return` is used to take pure data and bring it into the IO monad. No comments

Now, why would we want to do that? Remember that anything whose result depends on I/O must be within the IO monad. So if we are writing a function that performs I/O, then a pure computation, we will need to use `return` to make this pure computation the proper return value of the function. Otherwise, a type error would occur. Here's an example: No comments

```
-- file: ch07/return1.hs
import Data.Char(toUpper)

isGreen :: IO Bool
isGreen =
    do putStrLn "Is green your favorite color?"
       inpStr <- getLine
       return ((toUpper . head $ inpStr) == 'Y')
```

8 comments

We have a pure computation that yields a `Bool`. That computation is passed to `return`, which puts it into the `IO` monad. Since it is the last value in the `do` block, it becomes the return value of `isGreen`, but this is not because we used the `return` function. No comments

Here's a version of the same program with the pure computation broken out into a separate function. This helps keep the pure code separate, and can also make the intent more clear. No comments

```
-- file: ch07/return2.hs
import Data.Char(toUpper)

isYes :: String -> Bool
isYes inpStr = (toUpper . head $ inpStr) == 'Y'

isGreen :: IO Bool
isGreen =
    do putStrLn "Is green your favorite color?"
       inpStr <- getLine
       return (isYes inpStr)
```

2 comments

Finally, here's a contrived example to show that `return` truly does not have to occur at the end of a `do` block. In practice, it usually is, but it need not be so. No comments

```
-- file: ch07/return3.hs
returnTest :: IO ()
returnTest =
    do one <- return 1
       let two = 2
       putStrLn $ show (one + two)
```

No comments

Notice that we used `<-` in combination with `return`, but `let` in combination with the simple literal. That's because we needed both values to be pure in order to add them, and `<-` pulls things out of monads, effectively reversing the effect of `return`. Run this in **ghci** and you'll see `3` displayed, as expected. No comments

Is Haskell Really Imperative?

These `do` blocks may look a lot like an imperative language. After all, you're giving commands to run in sequence most of the time. No comments

But Haskell remains a lazy language at its core. While it is necessary to sequence actions for I/O at times, this is done using tools that are part of Haskell already. Haskell achieves a nice separation of I/O from the rest of the language through the `IO` monad as well. No comments

Side Effects with Lazy I/O

Earlier in this chapter, you read about `hGetContents`. We explained that the `String` it returns can be used in pure code. No comments

We need to get a bit more specific about what side effects are. When we say Haskell has no side-effects, what exactly does that mean? No comments

At a certain level, side-effects are always possible. A poorly-written loop, even if written in pure code, could cause the system's RAM to be exhausted and the machine to crash. Or it could cause data to be swapped to disk. No comments

When we speak of no side effects, we mean that pure code in Haskell can't run commands that trigger side effects. Pure functions can't modify a global variable, request I/O, or run a command to take down a system. No comments

When you have a `String` from `hGetContents` that is passed to a pure function, the function has no idea that this `String` is backed by a disk file. It will behave just as it always would, but processing that `String` may cause the environment to issue I/O commands. The pure function isn't issuing them;

they are happening as a result of the processing the pure function is doing, just as with the example of swapping RAM to disk. No comments

In some cases, you may need more control over exactly when your I/O occurs. Perhaps you are reading data interactively from the user, or via a pipe from another program, and need to communicate directly with the user. In those cases, `hGetContents` will probably not be appropriate. No comments

Buffering

The I/O subsystem is one of the slowest parts of a modern computer. Completing a write to disk can take thousands of times as long as a write to memory. A write over the network can be hundreds or thousands of times slower yet. Even if your operation doesn't directly communicate with the disk—perhaps because the data is cached—I/O still involves a system call, which slows things down by itself. No comments

For this reason, modern operating systems and programming languages both provide tools to help programs perform better where I/O is concerned. The operating system typically performs caching —storing frequently-used pieces of data in memory for faster access. No comments

Programming languages typically perform buffering. This means that they may request one large chunk of data from the operating system, even if the code underneath is processing data one character at a time. By doing this, they can achieve remarkable performance gains because each request for I/O to the operating system carries a processing cost. Buffering allows us to read the same amount of data with far fewer I/O requests. No comments

Haskell, too, provides buffering in its I/O system. In many cases, it is even on by default. Up till now, we have pretended it isn't there. Haskell usually is good about picking a good default buffering mode. But this default is rarely the fastest. If you have speed-critical I/O code, changing buffering could make a significant impact on your program. No comments

Buffering Modes

There are three different buffering modes in Haskell. They are defined as the `BufferMode` type: `NoBuffering`, `LineBuffering`, and `BlockBuffering`. No comments

`NoBuffering` does just what it sounds like—no buffering. Data read via functions like `hGetLine` will be read from the OS one character at a time. Data written will be written immediately, and also often will be written one character at a time. For this reason, `NoBuffering` is usually a very poor performer and not suitable for general-purpose use. No comments

`LineBuffering` causes the output buffer to be written whenever the newline character is output, or whenever it gets too large. On input, it will usually attempt to read whatever data is available in chunks until it first sees the newline character. When reading from the terminal, it should return data immediately after each press of Enter. It is often a reasonable default. No comments

`BlockBuffering` causes Haskell to read or write data in fixed-size chunks when possible. This is the best performer when processing large amounts of data in batch, even if that data is line-oriented. However, it is unusable for interactive programs because it will block input until a full block is read. `BlockBuffering` accepts one parameter of type `Maybe`: if `Nothing`, it will use an implementation-defined buffer size. Or, you can use a setting such as `Just 4096` to set the buffer to 4096 bytes. No comments

The default buffering mode is dependent upon the operating system and Haskell implementation. You can ask the system for the current buffering mode by calling `hGetBuffering`. The current mode can be set with `hSetBuffering`, which accepts a `Handle` and `BufferMode`. As an example, you can say `hSetBuffering stdin (BlockBuffering Nothing)`. No comments

Flushing The Buffer

For any type of buffering, you may sometimes want to force Haskell to write out any data that has been saved up in the buffer. There are a few times when this will happen automatically: a call to `hClose`, for instance. Sometimes you may want to instead call `hFlush`, which will force any pending data to be written immediately. This could be useful when the `Handle` is a network socket and you want the data to be transmitted immediately, or when you want to make the data on disk available to other programs that might be reading it concurrently. No comments

Reading Command-Line Arguments

Many command-line programs are interested in the parameters passed on the command line. `System.Environment.getArgs` returns `IO [String]` listing each argument. This is the same as `argv` in C, starting with `argv[1]`. The program name (`argv[0]` in C) is available from `System.Environment.getProgName`. No comments

The `System.Console.GetOpt` module provides some tools for parsing command-line options. If you have a program with complex options, you may find it useful. You can find an example of its use in the section called “Command line parsing”. No comments

Environment Variables

If you need to read environment variables, you can use one of two functions in `System.Environment`: `getEnv` or `getEnvironment`. `getEnv` looks for a specific variable and raises an exception if it doesn't exist. `getEnvironment` returns the whole environment as a `[(String, String)]`, and then you can use functions such as `lookup` to find the environment entry you want. No comments

Setting environment variables is not defined in a cross-platform way in Haskell. If you are on a POSIX platform such as Linux, you can use `putEnv` or `setEnv` from the `System.Posix.Env` module. Environment setting is not defined for Windows. No comments

---

[15] You will later see that it has a more broad application, but it is sufficient to think of it in these terms for now.

[16] The type of the value `()` is also `()`.

[17] Imperative programmers might be concerned that such a recursive call would consume large amounts of stack space. In Haskell, recursion is a common idiom, and the compiler is smart enough to avoid consuming much stack by optimizing tail-recursive functions.

[18] If there was a bug in the C part of a hybrid program, for instance

[19] For more information on interoperating with other programs with pipes, see the section called “Extended Example: Piping”.

[20] POSIX programmers may be interested to know that this corresponds to `unlink()` in C.

[21] `hGetContents` will be discussed in the section called “Lazy I/O”

[22] There is also a shortcut function `getContents` that operates on standard input.

[23] More precisely, it is the entire data from the current position of the file pointer to the end of the file.

[24] Excepting I/O errors such as a full disk, of course.

[25] Technically speaking, `mapM` combines a bunch of separate I/O actions into one big action. The separate actions are executed when the big action is.

Chapter 6. Using Typeclasses
Home
Chapter 8. Efficient file processing, regular expressions, and file name matching