Input and Output

Introduction

We've mentioned that Haskell is a purely functional language. Whereas in imperative languages you usually get things done by giving the computer a series of steps to execute, functional programming is more of defining what stuff is. In Haskell, a function can't change some state, like changing the contents of a variable (when a function changes state, we say that the function has side-effects).The only thing a function can do in Haskell is give us back some result based on the parameters we gave it. If a function is called two times with the same parameters, it has to return the same result. In an imperative language, you have no guarantee that a simple function that should just crunch some numbers won't burn down your house, kidnap your dog and scratch your car with a potato while crunching those numbers. For instance, when we were making a binary search tree, we didn't insert an element into a tree by modifying some tree in place. Our function for inserting into a binary search tree actually returns a new tree, because it can't change the old one.

While functions being unable to change state is good because it helps us reason about our programs, there's one problem with that. If a function can't change anything in the world, how is it supposed to tell us what it calculated? In order to tell us what it calculated, it has to change the state of an output device (usually the state of the screen).

It turns out that Haskell actually has a really clever system for dealing with functions that have side-effects that neatly separates the part of our program that is pure and the part of our program that is impure, which does all the dirty work like getting data to the keyboard and printing data on the screen. With those two parts separated, we can still reason about our pure program and take advantage of all the things that purity offers, like laziness, robustness and modularity while efficiently communicating with the outside world.

Pure vs Impure program is resumed in the table below.

Pure	Impure
Always produces the same result when given the same parameters	May produce different results for the same parameters
Never has side effects	May have side effects
Never alters state	May alter the global state of the program, system, or world

Input Output Action

In Haskell, Input/Output actions are handled by the parameterized type `IO a`

An I/O action is something that, when performed, will carry out an action with a side-effect (that's usually either reading from the input or printing stuff to the screen) and will also contain some kind of return value inside it. Printing a string to the terminal doesn't really have any kind of meaningful return value, so a dummy value of () is used.

Haskell separates pure functions from computations where side effects must be considered by encoding those side effects as values of a particular type. Specifically, a value of type (IO a) is an action, which if executed would produce a value of type a.

wada

Example:

```
getLine :: IO String
putStrLn :: String -> IO () -- note that the result value is an empty tuple.
randomRIO :: (Random a) => (a,a) -> IO a
```

The main action

Ordinary Haskell evaluation doesn't cause this execution to occur. A value of type (IO a) is almost completely inert. In fact, the only IO action which can really be said to run in a compiled Haskell program is main. Armed with this knowledge, we can write a "hello, world" program:

```
main :: IO ()
main = putStrLn "Hello, World!"
```

Note

What Is An I/O Action?

An I/O Action Actions:

- Have the type IO t
- Are first-class values in Haskell and fit seamlessly with Haskell's type system
- 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.
- 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)

- Performing (executing) an action of type IO t may perform I/O and will ultimately deliver a result of type t

Primitives for input/output actions

Haskell provides us with a few primitives for composing and chaining together IO actions.

(>>)

```
(>>) :: IO a -> IO b -> IO b
```

where if x and y are IO actions, then (x >> y) is the action that performs x, dropping the result, then performs y and returns its result. We can now write programs which do multiple things:
Example:

```
main = putStrLn "Hello" >> putStrLn "World"
```

will print "Hello" and "World" on separate lines.

(>>=)

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

This operation, called bind, gives us the possibilities to chain actions in which we are allowed to use the result of the first in order to affect what the second action will do.
x >>= f is the action that first performs the action x, and captures its result, passing it to f, which then computes a second action to be performed. That action is then carried out, and its result is the result of the overall computation.
Example:

```
main = putStrLn "Hello, what is your name?"
      >> getLine
      >>= \name -> putStrLn ("Hello, " ++ name ++ "!")
```

Return

In practice, it turns out to also be quite important to turn a value into an IO action which does nothing, and simply returns that value. This is quite handy near the end of a chain of actions, where we want to decide what to return ourselves, rather than leaving it up to the last action in the chain.

```
return :: a -> IO a
```

Do an variable binding

do primitive let us chain applications of (>>) and (>>=), and some lambdas when appropriate to capture the results of actions.
An action on its own on a line in a do-block will be executed, and a line of the form v <- x will cause the action x to be run, and the result bound to the variable v. To make a variable binding inside a do-block which doesn't involve running an action, then you can use a line of the form let a = b, which, like an ordinary let-expression will define a to be the same as b, but the definition scopes over the remainder of the do-block.

Example:

```
main = do putStrLn "Hello, what is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name ++ "!")
```

```haskell
main = do
    let a = "hell"
        b = "yeah"
    putStrLn $ a ++ " " ++ b
```

Standard I/O Functions and stream

Although Haskell provides fairly sophisticated I/O facilities, as defined in the IO library, it is possible to write many Haskell programs using only the few simple functions that are exported from the Prelude, and which are described in this section.
All I/O functions defined here are character oriented. The treatment of the newline character will vary on different systems.

Output Functions

These functions write to the standard output device (this is normally the user's terminal).

```haskell
putChar  :: Char -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()  -- adds a newline
print    :: Show a => a -> IO ()
```

The print function outputs a value of any printable type to the standard output device. Printable types are those that are instances of class Show; print converts values to strings for output using the show operation and adds a newline.

For example, a program to print the first 20 integers and their powers of 2 could be written as:

```haskell
main = print ([(n, 2^n) | n <- [0..19]])
```

```
main = do
   putChar 't'
   putChar 'e'
   putChar 'h'

$ runhaskell putchar_test.hs
teh
```

Input Functions

These functions read input from the standard input device (normally the user's keyboard).

```
  getChar      :: IO Char -- Read à single character
  getLine      :: IO String -- Read is press on keyboard
untill enter is pressed.
  getContents :: IO String -- is lazy.  read until EOF.
(that's usually done by pressing Ctrl-D)
  interact     :: (String -> String) -> IO ()
  readIO       :: Read a => String -> IO a
  readLn       :: Read a => IO a
```

The getChar operation raises an exception on end-of-file; a predicate isEOFError that identifies this exception is defined in the IO library. The getLine operation raises an exception under the same circumstances as hGetLine, defined in the IO library.

Example

```
import Data.Char
main = do
   contents <- getContents
```

```
    putStr (map toUpper contents)
```

The getContents operation returns all user input as a single string, which is read lazily as it is needed. Keep in mind that because strings are basically lists, which are lazy, and getContents is I/O lazy, it won't try to read the whole content at once and store it into memory before printing out the caps locked version. Rather, it will print out the caps locked version as it reads it, because it will only read a line from the input when it really needs to. When the result of getContents is bound to contents, it's not represented in memory as a real string, but more like a promise that it will produce the string eventually.

The interact function takes a function of type String->String as its argument. The entire input from the standard input device is passed to this function as its argument, and the resulting string is output on the standard output device.

Typically, the read operation from class Read is used to convert the string to a value. The readIO function is similar to read except that it signals parse failure to the I/O monad instead of terminating the program. The readLn function combines getLine and readIO.

By default, these input functions echo to standard output.

The following program simply removes all non-ASCII characters from its standard input and echoes the result on its standard output. (The isAscii function is defined in a library.)

```
main = interact (filter isAscii)
```

Application

Now we're going to make a program that continuously reads a line and prints out the same line with the words reversed. The program's execution will stop when we input a blank line. This is the program:

```
main = do
   line <- getLine
   if null line
      then return ()
      else do
        putStrLn $ reverseWords line
        main
reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

Files and Handles

So far, we've worked with I/O by printing out stuff to the terminal and reading from it. But what about reading and writing files? Well, in a way, we've already been doing that. One way to think about reading from the terminal is to imagine that it's like reading from a (somewhat special) file. Same goes for writing to the terminal, it's kind of like writing to a file. We can call these two files handle stdout and stdin, meaning standard output and standard input, respectively. Keeping that in mind, we'll see that writing to and reading from files is very much like writing to the standard output and reading from the standard input.

In most use cases, 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.

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.

To deal with file in haskell, we need to import the module System.IO

Open a File

To open a file in Haskell we use the function openFile. It type is:

```
openFile :: FilePath -> IOMode -> IO Handle
```

FilePath is simply another name (synonym) 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.

```
type FilePath =  String
```

IOMode specifies how the file is to be managed. The possible values for IOMode are listed below.

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

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 created if it didn't exist; 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.

Read and Write in a file

These functions write to the file.

```haskell
hPutChar :: Handle -> Char -> IO ()
hPutStr :: Handle -> String -> IO ()
hPutStrLn :: Handle -> String -> IO ()  -- adds a newline
hPrint :: Show a => Handle -> a -> IO ()
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

These functions read input from the file.

```haskell
hGetChar :: Handle -> IO Char
hGetLine :: Handle -> IO String
hGetContents :: Handle -> IO String
readFile :: FilePath -> IO String
```

Lazy I/O

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.

hGetContents

The String hGetContents represents all of the data in the file given by the Handle.
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.
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.

```haskell
-- 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
```

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.

readFile, writeFile and appendFile

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, writeFile and appendFile 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.

Now, here's an example program that uses readFile and writeFile:

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

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

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.

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.

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.

We can convert our example program to operate on standard input and standard output by using interact. Here's one way to do that:

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

main = interact (map toUpper)
```

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.

Close a File

It is important to close the file when you have finished using it in order to free up the allocated resources. We use the function hClose to close a file handle.

```
hClose :: Handle -> IO ()
```

It is practice to alway close files. There are many reasons:

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.

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.

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, so once again, it is best to be responsible and call hClose all the time.

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.

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.

The companion to hTell is hSeek. hSeek lets you change the file position. It takes three parameters: a Handle, a SeekMode, and a position.

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.

Note:
Not all Handles 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.

Standard Input, Output, and Error handle

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.

There are three predefined Handles in System.IO. These Handles 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 terminal, and standard error also normally goes to the terminal.

Functions such as getLine can thus be trivially defined like this:

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

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.

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.

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.

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.

Command line arguments

Dealing with command line arguments is pretty much a necessity if you want to make a script or application that runs on a terminal. Luckily, Haskell's standard library has a nice way of getting command line arguments of a program.

Many command-line programs are interested in the parameters passed on the command line. System.Environment.getArgs returns IO [String] listing each argument. The program name is available from System.Environment.getProgName.

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.

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.

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.

Randomness

Many times while programming, you need to get some random data. Maybe you're making à game where a dice needs to be thrown or you need to generate some test data to test out your program. There are a lot of uses for random data when programming.In this section, we'll take a look at how to make Haskell generate seemingly random data.

Well, remember, Haskell is a pure functional language. What that means is that it has referential transparency. It means that a function, if given the same parameters twice, must produce the same result twice. That's

really cool because it allows us to reason differently about programs and it enables us to defer evaluation until we really need it. If I call a function, I can be sure that it won't do any funny stuff before giving me the results. However, this makes it a bit tricky for getting random numbers.
In Haskell, we can make a random number then if we make a function that takes as its parameter that randomness and based on that returns some number (or other data type). The System.Random module has all the functions that satisfy our need for randomness.
Let's just dive into one of the functions it exports then, namely random. Here's its type:

```
random:: (RandomGen g, Random a) => g -> (a, g)
```

Some new typeclasses in this type declaration up in here! The RandomGen typeclass is for types that can act as sources of randomness. The Random typeclass is for things that can take on random values. A boolean value can take on a random value, namely True or False. A number can also take up a plethora of different random values.
If we try to translate the type declaration of random to English, we get something like: it takes a random generator (that's our source of randomness) and returns a random value and a new random generator. Why does it also return a new generator as well as a random value? Well, we'll see in a moment.

To use our random function, we have to get our hands on one of those random generators. The System.Random module exports a cool type, namely StdGen that is an instance of the RandomGen typeclass. We can either make a StdGen manually or we can tell the system to give us one based on a multitude of sort of random stuff.

To manually make a random generator, use the mkStdGen function. It has a type of `mkStdGen :: Int -> StdGen`. It takes an integer and based on that, gives us a random generator. Okay then, let's try using random and mkStdGen in tandem to get a (hardly random) number.

```
ghci> random (mkStdGen 100)
(-3633736515773289454,693699796 2103410263)
```

The first component of the tuple is our number whereas the second component is a textual representation of our new random generator. What happens if we call random with the same random generator again?

```
ghci> random (mkStdGen 100)
(-3633736515773289454,693699796 2103410263)
```

Of course. The same result for the same parameters. So let's try giving it a different random
generator as a parameter.

```
random (mkStdGen 10)
(-2774747785423059091,1925364037 2103410263)
```

Alright, cool, great, a different number. We can use the type annotation to get different types back from that function.

```
Prelude System.Random>  random (mkStdGen 10) :: (Float, StdGen)
(3.2916963e-2,432453652 1655838864)

Prelude System.Random>  random (mkStdGen 10) :: (Bool, StdGen)
(True,440154 40692)

Prelude System.Random>  random (mkStdGen 10) :: (Integer,
StdGen)
(-2774747785423059091,1925364037 2103410263)
```

Let's make a function that simulates tossing a coin three times. If random didn't return a new generator along with a random value, we'd have to make this function take three random generators as a parameter and then return coin tosses for each of them. But that sounds wrong because if one generator can make a random value of type Int (which can take on a load of different values), it should be able to make three coin tosses (which can take on precisely eight combinations). So this is where random returning a new generator along with a value really comes in handy. We'll represent a coin with a simple Bool. True is tails, False is heads.

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen = let (firstCoin, newGen) = random gen
                     (secondCoin, newGen') = random newGen
                     (thirdCoin, newGen'') = random newGen'
                 in  (firstCoin, secondCoin, thirdCoin)
```

We call random with the generator we got as a parameter to get a coin and a new generator. Then we call it again, only this time with our new generator, to get the second coin. We do the same for the third coin. Had we called it with the same generator every time, all the coins would have had the same value and we'd only be able to get (False, False, False) or (True, True, True) as a result.

```
ghci> threeCoins (mkStdGen 21)
(True,True,True)
ghci> threeCoins (mkStdGen 22)
(True,False,True)
ghci> threeCoins (mkStdGen 943)
(True,False,True)
ghci> threeCoins (mkStdGen 944)
(True,True,True)
```

So what if we want to flip four coins? Or five? Well, there's a function called randoms that takes a generator and returns an infinite sequence of values based on that generator.

```
ghci> take 5 $ randoms (mkStdGen 11) :: [Int]
[-1807975507,545074951,-1015194702,-1622477312,-502893664]
ghci> take 5 $ randoms (mkStdGen 11) :: [Bool]
[True,True,True,True,False]
ghci> take 5 $ randoms (mkStdGen 11) :: [Float]
[7.904789e-2,0.62691015,0.26363158,0.12223756,0.38291094]
```

Why doesn't randoms return a new generator as well as a list? We could implement the randoms function very easily like this:

```
randoms' :: (RandomGen g, Random a) => g -> [a]
randoms' gen = let (value, newGen) = random gen in
value:randoms' newGen
```

Because we have to be able to potentially generate an infinite amount of numbers, we can't give the new random generator back.
We could make a function that generates a finite stream of numbers and a new generator like this:

```
finiteRandoms :: (RandomGen g, Random a, Num n) => n -> g
-> ([a], g)
finiteRandoms 0 gen = ([], gen)
finiteRandoms n gen =
   let (value, newGen) = random gen
   (restOfList, finalGen) = finiteRandoms (n-1) newGen
in
(value:restOfList, finalGen)
```

What if we want a random value in some sort of range? All the random integers so far were outrageously big or small. What if we want to throw a dice? Well, we use randomR for that purpose. It has a type of randomR :: (RandomGen g, Random a) :: (a, a) -> g -> (a, g), meaning that it's kind of like random, only it takes as its first parameter a pair of values that set the lower and upper bounds and the final value produced will be within those bounds.

```
ghci> randomR (1,6) (mkStdGen 359353)
(6,1494289578 40692)
ghci> randomR (1,6) (mkStdGen 35935335)
(3,1250031057 40692)
```

There's also randomRs, which produces a stream of random values within our defined ranges.
Check this out:

```
ghci> take 10 $ randomRs ('a','z') (mkStdGen 3) :: [Char]
"ndkxbvmomg"
```

Looks like a super secret password or something. You may be asking yourself, what does this section have to do with I/O anyway? We haven't done anything concerning I/O so far. Well, so far we've always made our random number generator manually by making it with some arbitrary integer. The problem is, if we do that in our real programs, they will always return the same random numbers, which is no good for us. That's why System.Random offers the getStdGen I/O action, which has a type of IO StdGen. When your program starts, it asks the system for a good random number generator and stores that in a so called global generator. getStdGen fetches you that global random generator when you bind it to something. Here's a simple program that generates a random string.

```
import System.Random
main = do
```

```
  gen <- getStdGen
  putStr $ take 20 (randomRs ('a','z') gen)
$ runhaskell random_string.hs
pybphhzzhuepknbykxhe
$ runhaskell random_string.hs
eiqgcxykivpudlsvvjpg
$ runhaskell random_string.hs
nzdceoconysdgcyqjruo
$ runhaskell random_string.hs
bakzhnnuzrkgvesqplrx
```

Be careful though, just performing getStdGen twice will ask the system for the same global generator twice. If you do this:

```
import System.Random
main = do
  gen <- getStdGen
  putStrLn $ take 20 (randomRs ('a','z') gen)
  gen2 <- getStdGen
  putStr $ take 20 (randomRs ('a','z') gen2)
```

you will get the same string printed out twice! One way to get two different strings of length 20 is to set up an infinite stream and then take the first 20 characters and print them out in one line and then take the second set of 20 characters and print them out in the second line. For this, we can use the splitAt function from Data.List, which splits a list at some index and returns a tuple that has the first part as the first component and the second part as the second component.

```
import System.Random
import Data.List
main = do
```

```
  gen <- getStdGen
  let randomChars = randomRs ('a','z') gen
    (first20, rest) = splitAt 20 randomChars
    (second20, _) = splitAt 20 rest
  putStrLn first20
  putStr second20
```

Another way is to use the newStdGen action, which splits our current random generator into two generators. It updates the global random generator with one of them and encapsulates the other as its result.

```
import System.Random
main = do
  gen <- getStdGen
  putStrLn $ take 20 (randomRs ('a','z') gen)
  gen' <- newStdGen
  putStr $ take 20 (randomRs ('a','z') gen')
```

Not only do we get a new random generator when we bind newStdGen to something, the global one gets updated as well, so if we do getStdGen again and bind it to something, we'll get a generator that's not the same as gen.

Here's a little program that will make the user guess which number it's thinking of.

```
import System.Random
import Control.Monad(when)
main = do
  gen <- getStdGen
  askForNumber gen

askForNumber :: StdGen -> IO ()
```

```
askForNumber gen = do
  let (randNumber, newGen) = randomR (1,10) gen :: (Int, StdGen)
  putStr "Which number in the range from 1 to 10 am I thinking of? "
  numberString <- getLine
  when (not $ null numberString) $ do
    let number = read numberString
    if randNumber == number
      then putStrLn "You are correct!"
      else putStrLn $ "Sorry, it was " ++ show randNumber
    askForNumber newGen
```

Bytestrings

processing files as strings has one drawback: it tends to be slow. As you know, String is a type synonym for [Char]. Chars don't have a fixed size, because it takes several bytes to represent a character from, say, Unicode. Furthemore, lists are really lazy. If you have a list like [1,2,3,4], it will be evaluated only when completely necessary. So the whole list is sort of a promise of a list. Remember that [1,2,3,4] is syntactic sugar for 1:2:3:4:[]. When the first element of the list is forcibly evaluated (say by printing it), the rest of the list 2:3:4:[] is still just a promise of a list, and so on. So you can think of lists as promises that the next element will be delivered once it really has to and along with it, the promise of the element after it. It doesn't take a big mental leap to conclude that processing a simple list of numbers as a series of promises might not be the most efficient thing in the world.

That overhead doesn't bother us so much most of the time, but it turns out to be a liability when reading big files and manipulating them. That's why Haskell has bytestrings. Bytestrings are sort of like lists, only each element is one byte (or 8 bits) in size. The way they handle laziness is also different.

Bytestrings come in two flavors: strict and lazy ones. Strict bytestrings reside in Data.ByteString and they do away with the laziness completely. There are no promises involved; a strict bytestring represents a series of bytes in an array. You can't have things like infinite strict bytestrings. If you evaluate the first byte of a strict bytestring, you have to evaluate it whole. The upside is that there's less overhead because there are no thunks (the technical term for promise) involved. The downside is that they're likely to fill your memory up faster because they're read into memory at once.

The other variety of bytestrings resides in Data.ByteString.Lazy. They're lazy, but not quite as lazy as lists. Like we said before, there are as many thunks in a list as there are elements. That's what makes them kind of slow for some purposes. Lazy bytestrings take a different approach — they are stored in chunks (not to be confused with thunks!) , each chunk has a size of 64K. So if you evaluate a byte in a lazy bytestring (by printing it or something) , the first 64K will be evaluated. After that, it's just a promise for the rest of the chunks. Lazy bytestrings are kind of like lists of strict bytestrings with a size of 64K. When you process a file with lazy bytestrings, it will be read chunk by chunk. This is cool because it won't cause the memory usage to skyrocket and the 64K probably fits neatly into your CPU's L2 cache.

If you look through the documentation for Data.ByteString.Lazy, you'll see that it has a lot of functions that have the same names as the ones from Data.List, only the type signatures have ByteString instead of [a] and Word8 instead of a in them. The functions with the same names mostly act the same as the ones that work on lists. Because the names are the same, we're going to do a qualified import in a script and then load that script into GHCI to play with bytestrings.

```
import qualified Data.ByteString.Lazy as B
import qualified Data.ByteString as S
```

B has lazy bytestring types and functions, whereas S has strict ones. We'll mostly be using the lazy version.

Using ByteString to read files

The ByteString module has a function of type `readFile : FilePath -> IO ByteString`.
Watch out, if you're using strict bytestrings and you attempt to read a file, it will read it into memory at once! With lazy bytestrings, it will read it into neat chunks.

Let's make a simple program that takes two filenames as command-line arguments and copies the first file into the second file. Note that System.Directory already has a function called copyFile, but we're going to implement our own file copying function and program anyway.

```
import System.Environment
import qualified Data.ByteString.Lazy as B
main = do
   (fileName1:fileName2:_) <- getArgs
   copyFile fileName1 fileName2
copyFile :: FilePath -> FilePath -> IO ()
copyFile source dest = do
   contents <- B.readFile source
   B.writeFile dest contents
```

We make our own function that takes two FilePaths (remember, FilePath is just a synonym for String) and returns an I/O action that will copy one file into another using bytestring. In the main function, we just get the arguments and call our function with them to get the I/O action, which is then performed.

Note

Whenever you need better performance in a program that reads a lot of data into strings, give bytestrings a try, chances are you'll get some good performance boosts with very little effort on your part. We usually write programs by using normal strings and then convert them to use bytestrings if the performance is not satisfactory.

You can run the command :browse Data.ByteString.Lazy to see all functions available in the module Data.ByteString.Lazy.

Exceptions

Haskell has a very good type system. Algebraic data types allow for types like Maybe and Either and we can use values of those types to represent results that may be there or not.
Haskell's type system gives us some much-needed safety in that aspect. A function a -> Maybe b clearly indicates that it may produce a b wrapped in Just or that it may return Nothing.

Despite having expressive types that support failed computations, Haskell still has support for exceptions, because they make more sense in I/O contexts. A lot of things can go wrong when dealing with the outside world because it is so unreliable. For instance, when opening a file, a bunch of things can go wrong. The file might be locked, it might not be there at all or the hard disk drive or something might not be there at all. So it's good to be able to jump to some error handling part of our code when such an error occurs.

so I/O code (i.e. impure code) can throw exceptions. It makes sense. But what about pure
code? Well, it can throw exceptions too. Think about the div and head functions. They have types of (Integral a) => a -> a -> a and [a] -> a, respectively. No Maybe or Either in their return type and yet they can both fail! div explodes in your face if you try to divide by zero and head throws a tantrum when you give it an empty list.

```
ghci> 4 `div` 0
*** Exception: divide by zero
ghci> head []
*** Exception: Prelude.head: empty list
```

Once pure functions start throwing exceptions, it matters when they are evaluated. That's why we can only catch exceptions thrown from pure functions in the I/O part of our code. And that's bad, because we want to keep the I/O part as small as possible. However, if we don't catch them in the I/O part of our code, our program crashes. The solution? Don't mix exceptions and pure code. Take advantage of Haskell's powerful type system and use types like Either and Maybe to represent results that may have failed.

That's why we'll just be looking at how to use I/O exceptions for now. I/O exceptions are exceptions that are caused when something goes wrong while we are communicating with the outside world in an I/O action that's part of main. For example, we can try opening a file and then it turns out that the file has been deleted or something. Take a look at this program that opens a file whose name is given to it as a command line argument and tells us how many lines the file has.

```
import System.Environment
import System.IO
main = do (fileName:_) <- getArgs
   contents <- readFile fileName
```

```
   putStrLn $ "The file has " ++ show (length (lines contents)) ++ "
lines!"
```

It works as expected, but what happens when we give it the name of a file that doesn't exist?

```
$ runhaskell linecount.hs i_dont_exist.txt
```

linecount.hs: i_dont_exist.txt: openFile: does not exist (No such file or directory), we get an
error from GHC, telling us that the file does not exist. Our program crashes. What if we wanted to print out a nicer message if the file doesn't exist? One way to do that is to check if the file exists before trying to open it by using the doesFileExist function from System.Directory.

```
import System.Environment
import System.IO
import System.Directory
main = do (fileName:_) <- getArgs
    fileExists <- doesFileExist fileName
   if fileExists
      then do contents <- readFile fileName
      putStrLn $ "The file has " ++ show (length (lines contents)) ++ "
lines!"
      else do putStrLn "The file doesn't exist!"
```

Another solution here would be to use exceptions. It's perfectly acceptable to use them in this context. A file not existing is an exception that arises from I/O, so catching it in I/O is fine and dandy.

To deal with this by using exceptions, we're going to take advantage of the catch function from System.IO.Error. Its type is `catch : : IO a ->`

`(IOError -> IO a) -> IO a`. It takes two parameters. The first one is an I/O action. For instance, it could be an I/O action that tries to open a file. The second one is the so-called handler. If the first I/O action passed to catch throws an I/O exception, that exception gets passed to the handler, which then decides what to do. So the final result is an I/O action that will either act the same as the first parameter or it will do what the handler tells it if the first I/O action throws an exception.

The handler takes a value of type IOError, which is a value that signifies that an I/O exception occurred. It also carries information regarding the type of the exception that was thrown. How this type is implemented depends on the implementation of the language itself, which means that we can't inspect values of the type IOError by pattern matching against them, just like we can't pattern match against values of type IO something. We can use a bunch of useful predicates to find out stuff about values of type IOError as we'll learn in a second.
So let's put our new friend's catch to use!

```
import System.Environment
import System.IO
import System.IO.Error
main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_) <- getArgs
    contents <- readFile fileName
    putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"
handler :: IOError -> IO ()
handler e = putStrLn "Whoops, had some trouble!"
```

toTry is the I/O action that we try to carry out and handler is the function that takes an IOError and returns an action to be carried out in case of an exception.

```
$ runhaskell count_lines.hs i_exist.txt
The file has 3 lines!
$ runhaskell count_lines.hs i_dont_exist.txt
Whoops, had some trouble!
```

In the handler, we didn't check to see what kind of IOError we got. We just say "Whoops, had
some trouble!" for any kind of error. Just catching all types of exceptions in one handler is bad practice in Haskell just like it is in most other languages. What if some other exception happens that we don't want to catch, like us interrupting the program or something? That's why we're going to do the same thing that's usually done in other languages as well: we'll check to see what kind of exception we got. If it's the kind of exception we're waiting to catch, we do our stuff. If it's not, we throw that exception back into the wild. Let's modify our program to catch only the exceptions caused by a file not existing.

```
import System.Environment
import System.IO
import System.IO.Error
main = toTry `catch` handler
toTry :: IO ()
toTry = do (fileName:_) <- getArgs
   contents <- readFile fileName
   putStrLn $ "The file has " ++ show (length (lines contents)) ++ "
lines!"

handler :: IOError -> IO ()
handler e
```

```
| isDoesNotExistError e = putStrLn "The file doesn't exist!"
| otherwise = ioError e
```

Everything stays the same except the handler, which we modified to only catch a certain group of I/O exceptions. Here we used two new functions from System.IO.Error — isDoesNotExistError and ioError. isDoesNotExistError is a predicate over IOErrors, which means that it's a function that takes an IOError and returns a True or False, meaning it has a type of isDoesNotExistError : IOError -> Bool. We use it on the exception that gets passed to our handler to see if it's an error caused by a file not existing. We use guard syntax here, but we could have also used an if else. If it's not caused by a file not existing, we rethrow the exception that was passed by the handler with the ioError function. It has a type of ioError :: IOException -> IO a, so it takes an IOError and produces an I/O action that will throw it. The I/O action has a type of IO a, because it never actually yields a result, so it can act as IO anything. So the exception thrown in the toTry I/O action that we glued together with a do block isn't caused by a file existing, toTry `catch` handler will catch that and then re-throw it. There are several predicates that act on IOError and if a guard doesn't evaluate to True, evaluation falls through to the next guard. The predicates that act on IOError are:

- isAlreadyExistsError
- isDoesNotExistError
- isAlreadyInUseError
- isFullError
- isEOFError
- isIllegalOperation
- isPermissionError
- isUserError

Most of these are pretty self-explanatory. isUserError evaluates to True when we use the function userError to make the exception, which is

used for making exceptions from our code and equipping them with a string. For instance, you can do ioError $ userError "remote computer unplugged!", although It's prefered you use types like Either and Maybe to express possible failure instead of throwing exceptions yourself with userError.
So you could have a handler that looks something like this:

```
handler :: IOError -> IO ()
handler e
   | isDoesNotExistError e = putStrLn "The file doesn't exist!"
   | isFullError e = freeSomeSpace
   | isIllegalOperation e = notifyCops
   | otherwise = ioError e
```

Where notifyCops and freeSomeSpace are some I/O actions that you define. Be sure to rethrow exceptions if they don't match any of your criteria, otherwise you're causing your program to fail silently in some cases where it shouldn't.

System.IO.Error also exports functions that enable us to ask our exceptions for some attributes, like what the handle of the file that caused the error is, or what the filename is. These start with ioe and you can see a full list of them in the documentation. Say we want to print the filename that caused our error. We can't print the fileName that we got from getArgs, because only the IOError is passed to the handler and the handler doesn't know about anything else. A function depends only on the parameters it was called with. That's why we can use the ioeGetFileName function, which has a type of ioeGetFileName :: IOError -> Maybe FilePath. It takes an IOError as a parameter and maybe returns a FilePath (which is just a type synonym for String, remember, so it's kind of the same thing). Basically, what it does is it extracts the file path from the IOError, if it can. Let's modify our program to print out the file path that's responsible for the exception occurring.

```
import System.Environment
```

```
import System.IO
import System.IO.Error
main =  toTry `catch` handler
toTry :: IO ()
toTry = do (fileName:_) <- getArgs
   contents <- readFile fileName
   putStrLn $ "The file has " ++ show (length (lines contents)) ++ "
lines!"

handler :: IOError -> IO ()
handler e
   | isDoesNotExistError e =
     case ioeGetFileName e of Just path -> putStrLn $ "Whoops! File
does not exist at: " ++ path
     Nothing -> putStrLn "Whoops! File does not exist at unknown
location!"
   | otherwise = ioError e
```

Now you know how to deal with I/O exceptions! Throwing exceptions from pure code and dealing with them hasn't been covered here, mainly because, like we said, Haskell offers much better ways to indicate errors than reverting to I/O to catch them. Even when glueing together I/O actions that might fail, I prefer to have their type be something like IO (Either a b), meaning that they're normal I/O actions but the result that they yield when performed is of type Either a b, meaning it's either Left a or Right b.

Ref:

- https://wiki.haskell.org/Introduction_to_IO
- http://book.realworldhaskell.org/read/io.html
- http://www2.informatik.uni-freiburg.de/~thiemann/haskell/haskell98-report-html/io-13.html