```
main
```

It's very similar to the previous version, only instead of making a function that takes a generator and then calls itself recursively with the new updated generator, we do all the work in `main`. After telling the user whether they were correct in their guess or not, we update the global generator and then call `main` again. Both approaches are valid but I like the first one more since it does less stuff in `main` and also provides us with a function that we can reuse easily.

Bytestrings

Lists are a cool and useful data structure. So far, we've used them pretty much everywhere. There are a multitude of functions that operate on them and Haskell's laziness allows us to exchange the for and while loops of other languages for filtering and mapping over lists, because evaluation will only happen once it really needs to, so things like infinite lists (and even infinite lists of infinite lists!) are no problem for us. That's why lists can also be used to represent streams, either when reading from the standard input or when reading from files. We can just open a file and read it as a string, even though it will only be accessed when the need arises.

However, processing files as strings has one drawback: it tends to be slow. As you know, `String` is a type synonym for `[Char]`. `Char`s 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.

The function `pack` has the type signature `pack :: [Word8] -> ByteString`. What that means is that it takes a list of bytes of type `Word8` and returns a `ByteString`. You can think of it as taking a list, which is lazy, and making it less lazy, so that it's lazy only at 64K intervals.

What's the deal with that `Word8` type? Well, it's like `Int`, only that it has a much smaller range, namely 0-255. It represents an 8-bit number. And just like `Int`, it's in the `Num` typeclass. For instance, we know that the value `5` is polymorphic in that it can act like any numeral type. Well, it can also take the type of `Word8`.

```
ghci> B.pack [99,97,110]
Chunk "can" Empty
ghci> B.pack [98..120]
Chunk "bcdefghijklmnopqrstuvwx" Empty
```

As you can see, you usually don't have to worry about the `Word8` too much, because the type system can makes the numbers choose that type. If you try to use a big number, like `336` as a `Word8`, it will just wrap around to `80`.

We packed only a handful of values into a `ByteString`, so they fit inside one chunk. The `Empty` is like the `[]` for lists.

`unpack` is the inverse function of `pack`. It takes a bytestring and turns it into a list of bytes.

`fromChunks` takes a list of strict bytestrings and converts it to a lazy bytestring. `toChunks` takes a lazy bytestring and converts it to a list of strict ones.

```
ghci> B.fromChunks [S.pack [40,41,42], S.pack [43,44,45], S.pack [46,47,48]]
```

```
Chunk "()*" (Chunk "+,-" (Chunk "./0" Empty))
```

This is good if you have a lot of small strict bytestrings and you want to process them efficiently without joining them into one big strict bytestring in memory first.

The bytestring version of `:` is called `cons` It takes a byte and a bytestring and puts the byte at the beginning. It's lazy though, so it will make a new chunk even if the first chunk in the bytestring isn't full. That's why it's better to use the strict version of **`cons`**, `cons'` if you're going to be inserting a lot of bytes at the beginning of a bytestring.

```
ghci> B.cons 85 $ B.pack [80,81,82,84]
Chunk "U" (Chunk "PQRT" Empty)
ghci> B.cons' 85 $ B.pack [80,81,82,84]
Chunk "UPQRT" Empty
ghci> foldr B.cons B.empty [50..60]
Chunk "2" (Chunk "3" (Chunk "4" (Chunk "5" (Chunk "6" (Chunk "7" (Chunk "8" (Chunk "9" (Chunk
<"
Empty))))))))))
ghci> foldr B.cons' B.empty [50..60]
Chunk "23456789:;<" Empty
```

As you can see `empty` makes an empty bytestring. See the difference between **`cons`** and **`cons'`**? With the **`foldr`**, we started with an empty bytestring and then went over the list of numbers from the right, adding each number to the beginning of the bytestring. When we used **`cons`**, we ended up with one chunk for every byte, which kind of defeats the purpose.

Otherwise, the bytestring modules have a load of functions that are analogous to those in **`Data.List`**, including, but not limited to, **`head`**, **`tail`**, **`init`**, **`null`**, **`length`**, **`map`**, **`reverse`**, **`foldl`**, **`foldr`**, **`concat`**, **`takeWhile`**, **`filter`**, etc.

It also has functions that have the same name and behave the same as some functions found in **`System.IO`**, only **`String`**s are replaced with **`ByteString`**s. For instance, the **`readFile`** function in **`System.IO`** has a type of **`readFile :: FilePath -> IO String`**, while the `readFile` from the bytestring modules has a type of **`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 `FilePath`s (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.

```
$ runhaskell bytestringcopy.hs something.txt ../../something.txt
```

Notice that a program that doesn't use bytestrings could look just like this, the only difference is that we used `B.readFile` and `B.writeFile` instead of `readFile` and `writeFile`. Many times, you can convert a program that uses normal strings to a program that uses bytestrings by just doing the necessary imports and then putting the qualified module names in front of some functions. Sometimes, you have to convert functions that you wrote to work on strings so that they work on bytestrings, but that's not hard.

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. I usually write programs by using normal strings and then convert them to use bytestrings if the performance is not satisfactory.

Exceptions

All languages have procedures, functions, and pieces of code that might fail in some way. That's just a fact of life. Different languages have different ways of handling those failures. In C, we usually use some abnormal return value (like `-1` or a null pointer) to indicate that what a function returned shouldn't be treated like a normal value. Java and C#, on the other hand, tend to use exceptions to handle failure. When an exception is thrown, the control flow jumps to some code that we've defined that does some cleanup and then maybe re-throws the exception so that some other error handling code can take care of some other stuff.

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. In C, returning, say, `-1` on failure is completely a matter of convention. It only has special meaning to humans. If we're not careful, we might treat these abnormal values as ordinary ones and then they can cause havoc and dismay in our code. Haskell's type system gives us some much-needed safety in that aspect. A function `a -> Maybe b` clearly indicates that it it may produce a `b` wrapped in `Just` or that it may return `Nothing`. The type is different from just plain `a -> b` and if we try to use those two functions interchangeably, the compiler will complain at us.

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.

Okay, 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
```

Pure code can throw exceptions, but it they can only be caught in the I/O part of our code (when we're inside a *do* block that goes into **`main`**). That's because you don't know when (or if) anything will be evaluated in pure code, because it is lazy and doesn't have a well-defined order of execution, whereas I/O code does.

Earlier, we talked about how we should spend as little time as possible in the I/O part of our program. The logic of our program should reside mostly within our pure functions, because their results are dependant only on the parameters that the functions are called with. When dealing with pure functions, you only have to think about what a function returns, because it can't do anything else. This makes your life easier. Even though doing some logic in I/O is necessary (like opening files and the like), it should preferably be kept to a minimum. Pure functions are lazy by default, which means that we don't know when they will be evaluated and that it really shouldn't matter. However, 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!"
```

A very simple program. We perform the `getArgs` I/O action and bind the first string in the list that it yields to `fileName`. Then we call the contents of the file with that name `contents`. Lastly, we apply `lines` to those contents to get a list of lines and then we get the length of that list and give it to `show` to get a string representation of that number. 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)
```

Aha, 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!"
```

We did `fileExists <- doesFileExist fileName` because `doesFileExist` has a type of `doesFileExist :: FilePath -> IO Bool`, which means that it returns an I/O action that has as its result a boolean value which tells us if the file exists. We can't just use `doesFileExist` in an *if* expression directly.

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.

If you're familiar with *try-catch* blocks in languages like Java or Python, the `catch` function is similar to them. The first

parameter is the thing to try, kind of like the stuff in the *try* block in other, imperative languages. The second parameter is the handler that takes an exception, just like most *catch* blocks take exceptions that you can then examine to see what happened. The handler is invoked if an exception is thrown.

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 **`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!"
```

First of all, you'll see that put backticks around it so that we can use it as an infix function, because it takes two parameters. Using it as an infix function makes it more readable. So **`toTry `catch` handler`** is the same as **`catch toTry handler`**, which fits well with its type. **`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.

Let's give this a go:

```
$ 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 **`IOError`**s, 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 re-throw 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. Pretty cool, huh?

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 re-throw 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
```

In the guard where **`isDoesNotExistError`** is **`True`**, we used a *case* expression to call **`ioeGetFileName`** with **`e`** and then pattern match against the **`Maybe`** value that it returned. Using *case* expressions is commonly used when you want to pattern match against something without bringing in a new function.

You don't have to use one handler to **`catch`** exceptions in your whole I/O part. You can just cover certain parts of your I/O code with **`catch`** or you can cover several of them with **`catch`** and use different handlers for them, like so:

```
main = do toTry `catch` handler1
          thenTryThis `catch` handler2
          launchRockets
```

Here, **`toTry`** uses **`handler1`** as the handler and **`thenTryThis`** uses **`handler2`**. **`launchRockets`** isn't a parameter to **`catch`**, so whichever exceptions it might throw will likely crash our program, unless **`launchRockets`** uses **`catch`** internally to handle its own exceptions. Of course **`toTry`**, **`thenTryThis`** and **`launchRockets`** are I/O actions that have been glued together using *do* syntax and hypothetically defined somewhere else. This is kind of similar to *try-catch* blocks of other languages, where you can surround your whole program in a single *try-catch* or you can use a more fine-grained approach and use different ones in different parts of your code to control what kind of error handling happens where.

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`**.

Making Our Own Types and Typeclasses | Table of contents | Functionally Solving Problems