




The Joy
of Haskell



SOCKETS — AND — PIPES

CHRIS MARTIN
JULIE MORONUKI

Sockets and Pipes

by Chris Martin and Julie Moronuki

© 2022 Chris Martin and Julie Moronuki. All rights reserved.

2020-02-25: First draft

2020-05-14: Second draft

2020-09-21: Third draft

2021-02-19: Fourth draft

2021-05-01: Fifth draft

2022-09-01: Sixth draft

Contents

Preface	7
Prerequisites	8
What's inside	8
Setup	10
Proximity to reality	13
1 Handles	15
1.1 The necessity of indirection	15
1.2 Writing to a file	17
1.3 Diligent cleanup	21
1.4 MonadIO	27
1.5 Exercises	28
2 Chunks	31
2.1 Packed characters	31
2.2 Reading from a file, one chunk at a time	34
2.3 Exercises	37
3 Bytes	41
3.1 Packed octets	41
3.2 Copying a file	43
3.3 Character encodings	45

3.4	The Show and IsString classes	50
3.5	Avoiding system defaults	56
3.6	Exercises	59
4	Sockets	62
4.1	Open up and connect	63
4.2	Extra details	66
4.3	Names and addresses	68
4.4	Address information	71
4.5	Exercises	75
5	HTTP	78
5.1	The specification	79
5.2	HTTP requests	80
5.3	ASCII strings	82
5.4	HTTP responses	86
5.5	Serving others	88
5.6	Exercises	91
6	HTTP types	93
6.1	Request and response	94
6.2	Request line	95
6.3	Status line	97
6.4	Header fields	98
6.5	Message body	99
6.6	HTTP version	101
6.7	Exercises	102
7	Encoding	105
7.1	String builders	105
7.2	Measuring time	108
7.3	Request and response	112
7.4	Higher-order encodings	116

7.5	The start line	118
7.6	Exercises	122
8	Responding	124
8.1	A measure of success	124
8.2	Response-building utilities	127
8.3	Integers	130
8.4	Response transmission	132
8.5	Exercises	135
9	Content types	137
9.1	Some common types	138
9.2	UTF-8	139
9.3	HTML	141
9.4	JSON	145
9.5	Exercises	149
10	Change	152
10.1	STM	153
10.2	Increment	155
10.3	Atomically	156
10.4	The counting server	159
10.5	Other STM topics	160
10.6	Exercises	162
11	Streaming	164
11.1	Chunked hello	166
11.2	Chunk types	169
11.3	Encoding a chunk	171
11.4	Transfer-Encoding	173
11.5	Serving the file	174
11.6	Exercises	177
12	ListT IO	179

12.1	The new response type	179
12.2	What is ListT	181
12.3	Constructing a response	187
12.4	Encoding a response	189
12.5	Sending a response	193
12.6	ListT in other libraries	194
12.7	Exercises	195
13	Parsing	198
13.1	Encoding vs decoding	200
13.2	Attoparsec	204
13.3	Request line	209
13.4	Explaining what's wrong	216
13.5	Incremental parsing	218
13.6	Exercises	222
14	Errors	226
14.1	Status codes	227
14.2	Constructing responses	229
14.3	Visibility in two places	230
14.4	Thread-safe logging	236
14.5	Either	238
14.6	ExceptT	242
14.7	Exercises	247
	Coming up	249
15	Reading the head	250
16	Reading the body	251
17	Connection reuse	252
A	Solutions to exercises	253

Preface

The content that eventually grew into this book began with the question: What exactly *is* a web server? A satisfactory answer that does not assume substantial background knowledge requires spanning quite a few areas of computing. Fortunately, they all serve as fruitful motivations for simultaneously learning about how to use Haskell, which is the larger objective of the *Joy of Haskell* collection.

The language a web server speaks is the Hypertext Transfer Protocol (HTTP), which this book explores in great detail while walking through the creation of a server from “scratch”. We encourage readers to follow along in reading the official definition of HTTP (RFC 7230 published by the Internet Engineering Task Force) as we implement the specification in Haskell. While high-level libraries make it possible to create web applications without detailed knowledge of HTTP, we believe that a full understanding of the underlying layers we build upon helps us use a platform more effectively. By studying HTTP we also gain an appreciation for what it is and is not good for, and for what applications we might stand to benefit from choosing a different network protocol instead.

Prerequisites

This book is for Haskell learners who have some basic faculty with the language and are now ready to work up to a substantial project. We expect that you understand the basic syntax and can do things like:

- write a case expression to pattern match over a sum type
- sequence IO actions in a do block
- use qualified imports
- define datatypes
- use GHCi
- install Haskell libraries

From the base package, we assume some familiarity with:

- types `Maybe`, `Either`, and `[]`
- classes `Eq`, `Show`, `Monoid`, `Foldable`, `Functor`, and `Monad`

We do not assume prior knowledge of any additional libraries or GHC language extensions.

What's inside

Bytes and characters The first several chapters introduce the `bytestring` and `text` libraries and are largely dedicated to tearing apart a traditional *hello world* program, looking underneath the abstract notion of “printing text” to start greeting the world in terms of writing bytes to a file handle. After discussing bytes, we need only a short hop to *sockets*, our means of writing bytes across great distances using the `network` library.

Encoding and parsers First we encoding HTTP messages as byte strings. That's the easy part; next, we go in the opposite direction and learn how to interpret byte strings using the `attoparsec` library. This will acquaint us even more closely with the HTTP message format.

Monad transformers We introduce three Monad transformers that are especially applicable to our subject matter: `ResourceT`, `ListT`, and `ExceptT`. No prior experience with transformers is required. We do not linger on the general concept, preferring instead to focus on each of the three examples and to create familiarity with transformers and lifting chiefly by demonstration.

Resource safety Use of `ResourceT` begins in chapter 1, and we use it throughout the book. This makes it a breeze to deal with files and sockets without resource leaks.

Streaming To move past toy examples that fit easily into memory, we have to start writing *streaming* processes that can deal with large amounts of data by handling it in smaller pieces. All of the code within this book is written with memory usage in mind. `ListT`, the subject of chapter 12, provides an especially convenient facility for working with streams.

Error handling As the amount of functionality of our server builds up, the number of possible error conditions starts to rise. Chapter 14 introduces `ExceptT` to work with errors in a clean and well-typed manner.

This book has a companion Haskell library called `sockets-and-pipes`, available from the standard package repository.

<https://hackage.haskell.org/package/sockets-and-pipes>

The library re-exports all of the modules from other libraries that we use in the book; prospective readers are encouraged to browse the documentation at the web address given above, as it provides an overview of the libraries that you will learn to use from reading this book.

Setup

We strongly encourage you to follow along with the book and type the code as you read. The exercises at the end of each chapter make use of the code given in the chapter. Subsequent chapters will also refer back to definitions from earlier in the book, so it is important to keep everything as you progress.

You can organize the code however you like, but here we give a recommended setup for the convenience of less opinionated readers.

book.cabal

```
cabal-version: 3.0
name: book
version: 0
data-files: **/*.txt

library
  default-language:      Haskell2010
  default-extensions:    BlockArguments QuasiQuotes
                        TypeApplications ScopedTypeVariables
  ghc-options:           -Wall -fdefer-typed-holes
  build-depends:         sockets-and-pipes ^>= 0.3
  exposed-modules:       Book
```

`cabal-version`, `name`, and `version` are necessities in any Cabal package file.

The `default-extensions` field enables a few language extensions:

- *Block arguments* is a small adjustment to the Haskell syntax which allows a `do` block to be used as a function parameter, a task which has traditionally been accomplished using the `($\$)` operator.

- *Quasi-quotes* enables an alternate form of string literal syntax that we will use for writing ASCII strings beginning in chapter 5.
- *Type applications* is in use anywhere you see the `@` symbol. It is used to explicitly specify a type argument for a polymorphic function, generally used in situations where there is not enough surrounding context for the type to be inferred automatically.
- *Scoped type variables* allows type applications to refer to type variables rather than only to concrete types.

The `ghc-options` field specifies what GHC flags we use:

- `-Wall` enables all warnings, which we always do because this includes some particularly important ones such as detecting when a case expression is missing some cases.
- `-fdefer-typed-holes` allows us to write an underscore `_` in place of an expression, which we will use to write definitions that aren't quite finished yet. Such an underscore is called a "hole".

We list what libraries we need in the `build-depends` field. The only library we'll be using is `sockets-and-pipes`, which re-exports all of the modules used in the book. If you go off exploring on your own and want to use other libraries that we have not included in the `sockets-and-pipes` package, you can add additional entries to this `build-depends` list, separated by commas.

Book.hs Start this file as follows:

```

module Book where

import Prelude ()
import Relude

import qualified System.Directory as Dir
import System.FilePath ((</>))

getDataDir :: IO FilePath
getDataDir = do
    dir <- Dir.getXdgDirectory Dir.XdgData "sockets-and-pipes"
    Dir.createDirectoryIfMissing True dir
    return dir

```

This is where you will enter all of the code and exercise solutions. All of the definitions given in the book are named uniquely, so you should not need to remove anything as you go along; keep everything for future reference.

These first `import` statement removes from scope the implicitly-imported `Prelude` module that comes with the base package. The second imports `Relude`, an alternative prelude module which comes from the `re-lude` package. `Relude` is similar to the standard prelude, but it includes a few extra conveniences and will spare us some additional imports.

There are `import` declarations interspersed throughout the book; enter all of these into your code file as you encounter them. When we use an identifier that is re-exported by `Relude`, we do not give an import declaration since it would be unnecessary, but we may mention what module it originally comes from.

Once you have `book.cabal` and `Book.hs`, you can then use the command `cabal repl` to load your code into GHCi. We will use the REPL to query

for type information and to run example programs. Try it now:

```
λ> getDataDir  
"/home/chris/.local/share/sockets-and-pipes"
```

The file path you see here will vary based on your system. Make note of it; this is where you will be storing data files that we use as examples throughout the book.

Proximity to reality

There is always some tension between the code one writes for learning and the code one writes for a practical purpose.

Our recommendation of putting all the code into a single module is a departure from normal practice. Organization is a critical aspect of manageably developing software, but a single file keeps best with the linear progression of a book, and so we set aside our usual emphasis on designing small, cohesive modules.

The code that we give in this book is not annotated with any comments; the surrounding context of the book serves as the explanation that comments in code would normally provide. You are encouraged to insert your own comments in your `Book.hs` file, for the sake of note-taking and practice in writing API documentation.

Our code is wrapped into short lines, and some local variables have highly abbreviated names; these superficial choices follow from the constraints of print and are not necessarily intended as a recommended style in general.

In all other matters, the code in this book is intended as a presentation of good Haskell as written in practice. The words of an old drama teacher

echo in my mind – “What you do in rehearsal is what you’ll do in the performance.” – and so we have aimed here to avoid oversimplification and to make the same choices one might make on the job.

Chapter 1

Handles



Let us begin with some perspective on a process's position in the world.

1.1 The necessity of indirection

A typical running computer contains multitudes. Some processes correspond to things that you see on the screen; others work quietly behind the scenes. Your text editor, your terminal, each tab in your web browser, the synchronization of your system clock with other machines, that icon in the corner that shows you the signal strength of the wifi – each of these is controlled by a separate process. For the purposes of this book, we'll oversimplify a little and say that a software *process* is an instance of a program that is running, although the distinction is not always so simple or clear – for example, a single instance of a web browser may create a separate process for each tab.

You can start as many processes as you want (within reason). But while this multiplicity exists in software, the hardware is fixed. A machine might

have only one screen displaying graphics, one speaker playing sound, one chip storing all of the files, one cable or wireless transmitter linking the machine to the internet. It seems surprising, then, that computers can work at all – as if a hundred chefs may simultaneously cook a hundred soups sharing only a single pot among them, or as if a hundred orators may be heard delivering a hundred speeches at once at a single podium. Allowing many tenants to share the same facilities without interfering with one another is called *multiplexing*. This is the job of an operating system: to coordinate shared use of physical resources, weaving together the actions performed by many separate processes.

When we say that a program performs an action, we are usually talking about interacting with physical resources. Reading and writing files from the hard drive, downloading from and uploading to the internet, listening to and playing sound – inputs and outputs – *I/O*: all of these actions are mediated by the operating system. To say that a program *does* something ascribes more agency to it than it really has; the only thing its process can ever really do is issue requests for the operating system to do things on its behalf. These requests are called *system calls*.

Each operating system has its own set of system calls that programs running on it have to use to do I/O. You can run `man 2 syscalls` at the command prompt on a Linux machine to see the complete list of system calls supported by the Linux kernel. We will not give much attention to the differences between the operating systems because the Haskell libraries we use take care of these differences for us; the `base` and `network` libraries will automatically use whatever system calls are appropriate for the specific platform our programs are compiled for.

1.2 Writing to a file

If we consider I/O actions as lines of dialogue in a conversation between process and operating system, then we might think of a *handle* as an identifier for the conversation.

A handle for a file is referred to by Microsoft Windows as a *file handle* and by Linux as a *file descriptor*, often abbreviated as *fd*. In the Haskell libraries we use, you will see both terms used interchangeably. We prefer ‘handle’, but both are misleading. In a strained attempt to justify the metaphor: a handle on a resource is a temporary means of grabbing onto it, like the handhold provided by an ice climber’s axe.

Our first demonstration involving a file handle is a brief `IO ()` action that uses the basic operations of the `base` library to write two lines of text to a file:

```
import qualified System.IO as IO
```

```
writeGreetingFile = do
  dir <- getDataDir
  h <- IO.openFile (dir </> "greeting.txt") WriteMode
  IO.hPutStrLn h "hello"
  IO.hPutStrLn h "world"
  IO.hClose h
```

Enter this into your code file. Open `GHCi` and run `writeGreetingFile`, then look at the file that it created. (Remember that you can run `getDataDir` to see the path where files are stored.)

Getting a handle The first argument to the `openFile` function is the path of the file we want to write. The second argument, whose type is `IOMode`, is where we declare what we intend to do with the file: read its contents, write

new contents, or both.

```
-- defined in System.IO  
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Why do we have to specify upfront whether we're opening this file for reading or for writing? Remember that we aren't opening a file directly; we're asking the operating system to open a file for us. The answer lies in the OS's mediation and multiplexing responsibilities:

- ▶ The filesystem may have security restrictions that, for example, permit our process to read this file but not to write to it. The OS is responsible for enforcing this sort of access policy, and it does this permission check at the time a file is opened.
- ▶ Ours might not be the only process that accesses this file. Two processes simultaneously reading a file is fine, but two processes attempting to write to the same file at once could be trouble. The OS keeps track of all processes' file handles, and whether each is for reading or for writing, to prevent any conflicts.

The result of `openFile` will be a handle, which we've called `h` in this example. `hPutStrLn` has to know the handle in order to know where to put its strings.

The Haskell `openFile` function corresponds to the `open` system call in Linux. You can run `man 2 open` at the command prompt on a Linux machine to see the documentation for this system call. The return type of `open` is `int`, a number that the OS has assigned to you, like when you file an insurance claim and your insurance company gives you a claim number. *Please provide this number in all further communications regarding this claim*, they will tell you. That's what the file handle is. We have to provide this number as an argument to all subsequent system calls that pertain to this particular OS-mediated interaction with this file. The Haskell `openFile` function wraps

up this number into a value of type `Handle`, which is a bit more complicated but serves the same role as a conversation identifier.

Writing to a handle If you’ve seen a Haskell program, you’ve probably seen the `putStrLn` function in the `Prelude` module. We can use it like this:

```
helloWorld = IO.putStrLn "hello world!"
```

What you may not have known is that this program is using a handle. `putStrLn` is a specialization of a more general function called `hPutStrLn` in the `System.IO` module. (The ‘h’ stands for ‘handle’.)

```
IO.hPutStrLn :: Handle -> String -> IO ()
```

`putStrLn` is defined in terms of `hPutStrLn` and `stdout`, a handle we discuss shortly below.

```
IO.putStrLn :: String -> IO ()  
IO.putStrLn s = hPutStrLn stdout s
```

Even in the simplest *hello world* program, handles are there, just under the surface. The small `helloWorld` program above can be expressed equivalently as:

```
helloHandle = IO.hPutStrLn IO.stdout "hello world!"
```

The `Handle` parameter lets us write strings to destinations other than the standard output stream `stdout`. We saw an example of that earlier when we wrote to a file called `greeting.txt`.

What is `stdout`? When the OS starts a process, it creates by default a few “standard” places for the process to read and write. The *standard output*

stream is one of those, and `stdout` is the handle for it.

```
IO.stdout :: Handle
```

Each process has its own `stdout`. What happens when the process writes to its standard output stream? It depends on context. We often think of it as “how you print messages to the terminal,” because if we run a program at a command prompt, that’s what will happen.

```
module Main (main) where           -- file name: hello-world.hs

import qualified System.IO as IO

main = IO.putStrLn "hello world!"
```

```
$ runhaskell hello-world.hs
hello world!
```

But remember, a process never does anything directly! All I/O goes through the OS, and `stdout` is no exception. If we start the process in a context where its output is piped to a file, for example, then what it writes to `stdout` doesn’t display in the terminal. That same `putStrLn` action ends up writing to a file instead.

```
$ runhaskell hello-world.hs > greet.txt
```

```
$ cat greet.txt
hello world!
```

Background processes like servers often write their log output to `stdout` with the expectation that the OS will store all of the daemons’ logs.

Closing a handle Once we’re done writing, we use `hClose` to tell the OS that we’re done with the file handle. This isn’t really necessary in our little demo program, because when the process ends the OS will close all of its handles automatically anyway. But for long-running processes that may end up doing a lot of file operations, it can be important to make sure that handles get closed. The OS has to use memory to keep track of all of these handles, and if the number of handles associated with your process just keeps rising because you’re not closing them, eventually your operating system will become enraged and refuse to keep giving you more.

It is okay to run `hClose` on a handle more than once. After the first time, it has no effect.

1.3 Diligent cleanup

There is a subtle problem with `writeGreetingFile`. Although it does close the file handle, this is only assured if nothing goes wrong.

Once the handle has been opened, you might ask, what could possibly go wrong in a program as simple as this?

- *IO exceptions* – The `hPutStrLn` action can fail. For example, when the `Handle` represents a file in persistent storage, the operation fails when the storage medium is full.
- *Asynchronous exceptions* – Even when not engaged in any risky behavior, an action can *receive* an exception at any time, when someone (or something) decides to kill it. This happens, for example, when you send an interrupt from a terminal program (typically using the “ctrl+C” keyboard shortcut), or when one thread in the program throws an exception to another (see the `throwTo` function in the `Control.Exception` module).

In either case, if an exception is raised during the printing of the “hello

world” text, the `writeGreetingFile` action terminates without closing the handle. In a single-threaded program, this is not a big deal; when the exception is thrown, the process halts, and the operating system cleans up any open handles. However, a Haskell program that employs concurrency, such as the server we will be writing, has many IO actions running at once, and the process can stay alive even if an exception halts one of its threads.

To be sure that we deal with handle-closing reliably, we will use `ResourceT` from the `resourcet` library. Add this import statement:

```
import Control.Monad.Trans.Resource
      (ReleaseKey, ResourceT, allocate, runResourceT)
```

The new program, which has the type `IO ()`, is written as follows:

```
writeGreetingSafe = runResourceT @IO do
  dir <- liftIO getDataDir
  (_releaseKey, h) <- allocate
    (IO.openFile (dir </> "greeting.txt") WriteMode)
    IO.hClose
  liftIO (IO.hPutStrLn h "hello")
  liftIO (IO.hPutStrLn h "world")
```

We discuss each new bit of this in detail below.

ResourceT The type of the `do` expression above is `ResourceT IO ()`. The `T` in `ResourceT` stands for “transformer”, because `ResourceT` is a *monad transformer* – if `m` is a monad, then `ResourceT m` is a monad as well. `ResourceT IO` represents the concept of `IO` that has been transformed or modified by adding a certain safety feature. A `ResourceT IO` action is much like an `IO` action, but it is augmented with a register of resources that are guaranteed to be closed when the `ResourceT IO` action concludes.

Above we have used three new functions: `allocate`, `liftIO`, and `runResourceT`. Before we talk about what each means, we must first reckon with the many constraints in their type signatures.

```
liftIO :: MonadIO m => IO a -> m a    -- Control.Monad.IO.Class
```

```
allocate :: MonadResource m =>      -- Control.Monad.Trans.Resource
    IO a -> (a -> IO ()) -> m (ReleaseKey, a)
```

```
                                -- Control.Monad.Trans.Resource
runResourceT :: MonadUnliftIO m => ResourceT m a -> m a
```

We said a moment ago that a `ResourceT IO` action is much like an `IO` action. Whenever one type is like another, there are probably typeclasses involved. In the constraints above, we see three classes: `MonadResource`, `MonadUnliftIO`, and `MonadIO`. Wherever there are monad transformers, we find this sort of abundance of polymorphism, because it is often (but not always) the case that if some operation can be performed in the base context, then it may also be performed in the transformed context.

MonadIO This class describes any monad that an `IO` action can be lifted into. Its sole method is:

```
liftIO :: MonadIO m => IO a -> m a
```

`ResourceT IO` belongs to the `MonadIO` class, which tells us that `ResourceT IO` is in some sense more powerful than plain old `IO`; anything that you can do in `IO`, you can also do in `ResourceT IO` by *lifting* the ordinary `IO` into a resource-safety-augmented `ResourceT IO` context.

`IO` also belongs to the `MonadIO` class, which expresses the comically

trivial fact that an IO action is an IO action.

```
instance MonadIO IO where
    liftIO x = x
```

Such an instance may usually be found whenever a class's role is only to express that values of one type may be converted to another type. Although silly-looking, an instance like this does serve a purpose. When we encounter a function with a `MonadIO` constraint, it means we can use that function in any context that `IO` can be lifted into. But in situations where we do not need any augmentations, it also means that we can use that function with plain old unadorned `IO`, thanks to the humble `MonadIO IO` instance.

MonadUnliftIO We do not need to understand this class. Let it suffice to say that `IO` has an instance for it. You don't necessarily need to know what every constraint in a polymorphic type signature means. Sometimes all you need to do is look at the class's instance list to verify that the concrete type you're interested in using satisfies the constraint. This can be an important skill when reading Haskell API documentation.

MonadResource This class describes the special safety augmentations that the `resourcet` library provides. `ResourceT IO` belongs to this class, but regular old non-augmented `IO` does not. The `MonadResource` class exists because `ResourceT IO` can have further monad transformers applied to it, and the transformed monad in many cases also supports the polymorphic `allocate` function.

It is easy to get lost in a sea of typeclasses. If you begin to feel overloaded, focus on concretized type signatures. The three functions mentioned above, as we will use them in our hello-world program, are specialized as follows:


```
liftIO :: IO a -> ResourceT IO a
allocate :: IO a -> (a -> IO ()) -> ResourceT (ReleaseKey, a)
runResourceT :: ResourceT IO a -> IO a
```

Whenever you learn about a new type, pay attention to which functions introduce that type and which functions eliminate it. For example, with `ResourceT`:

- Introduction: `liftIO` and `allocate` create `ResourceT` values.
- Elimination: `runResourceT` takes an `ResourceT` as its argument and turns it into something else.

This tells us what the general structure of a program using the type will look like. To use `ResourceT`, we construct a `ResourceT` value using `liftIO` and `allocate`, and then we turn it into `IO` using `runResourceT`.

allocate The `allocate` function has two parameters:

1. `IO a` – An action that creates/opens/acquires a resource;
2. `a -> IO ()` – An action that destroys/closes/releases the resource.

When a `ResourceT` action performs an `allocate`, it runs the “open” action immediately and holds onto the “close” action for later. At the end of the `ResourceT` computation, it runs all of the “close” actions that have been registered. The closing phase is guaranteed to take place even if an exception interrupts the process.

release In circumstances where it is known that the `Handle` is no longer needed *before* the end of the `ResourceT` computation, you can use the `release` function to close it early. We give it the `ReleaseKey` that we obtained from the `allocate` function.

```
-- Control.Monad.Trans.Resource  
release :: MonadIO m => ReleaseKey -> m ()
```

Behind the scenes, the `ResourceT` context is maintaining a register of every cleanup action that needs to run once the action is over. If you are done using the file handle before the end of the action, it is tidier to use `release` instead of applying `IO.hClose` directly, because the `release` also scratches off the relevant cleanup action from the list.

Our small program does not need to make use of `release`, so we simply discard the `ReleaseKey` that `allocate` provides.

runResourceT One never constructs a `ResourceT` computation for its own sake; the ultimate purpose is to “run” the computation, which is a cute way to describe converting it to an `IO` action that can be run in `GHCi` or used as `main` to compile an executable.

We prefer to use `runResourceT` with a type application, writing `runResourceT @IO` to specify `IO` as the “base” monad. This type application is what allows the compiler to automatically infer the type signature of `writeGreetingSafe` that we have not given explicitly:

```
writeGreetingSafe :: IO ()
```

Throughout the rest of this book, whenever we open a file handle or other resource that needs to be closed, the `runResourceT` function will be involved.

1.4 MonadIO

We have above discussed `MonadIO` class and what `liftIO` means. We add here a few pragmatic notes about a programmer's relationship to it.

When introducing a new monad transformer like `ResourceT` into an `IO` sequence, every line under the `do` keyword whose type is not already of type `ResourceT IO` will have to be lifted into `ResourceT IO` using `liftIO`. (We saw this in the `writeGreetingSafe` program earlier.) This is a function that we often forget to apply, and one eventually learns to quickly recognize the type error:

Couldn't match type `IO` with `ResourceT IO`

An error message like this often indicates that a missing `liftIO` must be inserted.

A design question arises whenever a definition has an `IO` type, such as the `helloWorld` action.

```
helloWorld :: IO ()
helloWorld = IO.putStrLn "hello world!"
```

Such a definition can always be made polymorphic:

```
helloWorld :: MonadIO m => m ()
helloWorld = liftIO (IO.putStrLn "hello world!")
```

In the wild ecosystem of Haskell libraries, we find disagreement about whether this should always be done as a matter of course. For example, the `System.IO` and `Prelude` modules makes no use of the `MonadIO` class whatsoever, whereas `Relude` provides lifted `MonadIO` variants of many of the same functions.

Neither choice is obviously preferable in all circumstances. Polymor-

phic `MonadIO` type signatures are more convenient for users of monad transformers like `ResourceT`, since they remove the need to sprinkle `liftIO` throughout the code. Monomorphic IO types are sometimes easier on users who do not need lifted IO because polymorphism can weaken type inference, requiring type annotations or explicit type applications, such as `runResourceT @IO` as we have written here. The polymorphic choice may also induce more complicated error messages if type errors arise.



1.5 Exercises

Exercise 1 – File resource function Define a function with the following type signature:

```
fileResource ::  
    FilePath -> IOMode -> ResourceT IO (ReleaseKey, Handle)
```

Rewrite `writeGreetingSafe`, this time using the `fileResource` function we defined instead of using `allocate` directly.

Keep this `fileResource` function around, because we will continue to make use of it in later chapters.

Exercise 2 – Showing handles `Handle` has an instance of the `Show` type-class, which means you can use this function:

```
show :: Handle -> String
```

However, `show` doesn't give you much of the information you might want to see when you look at a handle. This is because a `Handle` is a messy real-world mutable sort of entity that requires I/O to ascertain its current

state, and `show` is a puny pure function that does not have `IO` in its type and cannot look up information about a mutable object. So in addition to the `Show` instance, we also have the `hShow` function:

```
IO.hShow :: Handle -> IO String
```

Write a program that uses `show` and `hShow` on a few handles to experiment and see the difference between these two functions' outputs.

```
handlePrintTest :: IO ()
handlePrintTest = _
```

Exercise 3 – Exhaustion How many file handles can you have open at once? Fill in the two holes below and run `howManyHandles` to find out.

```
import qualified Control.Exception.Safe as Ex
```

```
howManyHandles = runResourceT @IO do
  hs <- openManyHandles
  putStrLn ("Opened " <> show (length hs) <> " handles")
```

```
openManyHandles :: ResourceT IO [Handle]
openManyHandles = _
```

```

fileResourceMaybe :: ResourceT IO (Maybe Handle)
fileResourceMaybe = do
    dir <- liftIO getDataDir
    result <- Ex.tryIO do
        -
    case result of
        Right x -> return x
        Left e  -> do
            print (displayException e)
            return Nothing

```

The `openManyHandles` action should keep reopening a file until `IO.openFile` throws an exception, and it should return a list of all the handles that were opened. Recursion will be useful for making this action repeat in a loop.

This action will need to *catch* the exception that `openFile` will throw when the maximum number of file handles has been reached. For this, we have used the `tryIO` function.

```

tryIO :: MonadCatch m => m a -> m (Either IOException a)

```

In this context, its type is specialized as:

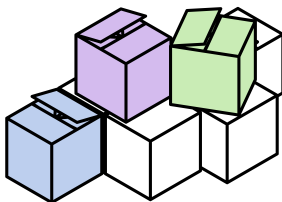
```

catchIO :: ResourceT IO (Maybe Handle)
    -> ResourceT IO (Either IOException (Maybe Handle))

```

Chapter 2

Chunks



When we move, we pack up our possessions because a box packed with many items is easier to store and move than a pile of loose items. Likewise in computer memory we prefer to move lists in bulk operations that can handle many elements at once.

Add the following imports for this chapter:

```
import qualified Data.Text as T
import qualified Data.Text.IO as T
```

We are also going to want the `Data.Char` module from `base`.

```
import qualified Data.Char as Char
```

2.1 Packed characters

The standard string type is a list of characters.

```
type String = [Char] -- defined in Data.String
```

A linked list is a bit like a pile of loose items – the elements are conceptually ordered, but not necessarily physically co-located in memory. For storing and moving in bulk, we often want a *packed* representation instead.

The `text` package provides one popular example of a packed type, `Text`. This type represents characters stored in one continuous block of memory. It is an *opaque* datatype, which means that as users of the library we do not use the constructor of `Text` directly. (For the curious: if you look at the source code, you can see that the data in a `Text` value is stored using `GHC.Exts.ByteArray#`, a *primitive type*.)

Packing and unpacking Instead of using a constructor, we obtain and use `Text`s by packing and unpacking `String`s,

```
T.pack  :: String -> Text
T.unpack :: Text  -> String
```

or via the many other handy functions provided by the `text` package.

Writing to a handle This package also provides a module named `Data.Text.IO`, which contains `Text`-based alternatives to many of the `String` functions in `System.IO`:

```
IO.hPutStrLn :: Handle -> String -> IO ()
T.hPutStrLn  :: Handle -> Text  -> IO ()
```

For a quick demonstration, we'll rewrite both of the example programs from chapter 1.

Writing to the standard output stream:

```
helloText = T.hPutStrLn stdout (T.pack "hello world!")
```

Writing to a file:

```
helloTextFile = runResourceT @IO do
  dir <- liftIO getDataDir
  (_, h) <- fileResource (dir </> "greeting.txt") WriteMode
  liftIO do
    T.hPutStrLn h (T.pack "hello")
    T.hPutStrLn h (T.pack "world")
```

Two things have changed:

1. We have applied `T.pack` to each of the strings that our programs write.
2. We have replaced `IO.hPutStrLn` with `T.hPutStrLn`.

Text is strict Packing a string into `Text` evaluates the entire string in one go, so our `Text` values have to fit in memory. For example, whereas we can inspect the beginning of a string even if the string is infinite,

```
λ> take 10 (cycle "abc")
"abcabcbabca"
```

a similar expression using `Text` cannot be evaluated.

```
λ> T.take 10 (T.pack (cycle "abc"))
-- (runs out of memory and crashes) --
```

We describe `Text` as a *strict* type because its contents are never partially

evaluated; either all of the characters are in memory, or none of them are. We describe `String` as *lazy* because it is evaluated only as needed.

2.2 Reading from a file, one chunk at a time

The *hello world* at the beginning of the chapter used this function to write a line of text to a handle:

```
T.hPutStrLn :: Handle -> Text -> IO ()
```

There is also a corresponding function to read a line of text from a handle:

```
T.hGetLine :: Handle -> IO Text
```

But, for our love of safety, we're not going to use it. Previously we demonstrated crashing GHCi by attempting to create a `Text` value that was too large; we want to avoid any possibility of doing that in a real program. Since we have no idea what we're going to get when we read from a handle, we're never going to write a program that is so thoughtless about the amount of data it attempts to pack into a `Text` value.

Reading from a handle Instead we'll read from handles using this function:

```
T.hGetChunk :: Handle -> IO Text
```

Chunking a stream of data is like taking reasonably sized bites out of food so we don't choke on a piece that's too big. Each bite is a *chunk*. This strategy offers a healthy compromise between working with text one character at a time – which doesn't take advantage of the machine's ability to

efficiently perform bulk operations on strings – and dealing with an entire file at a time.

`hGetChunk` reads a chunk – not guaranteed to be any particular size, but generally around a few thousand characters – from a handle. Once we’ve done whatever we need to do with that chunk, we can repeat the action to continue reading the rest of the handle. Once we’ve reached the end of the input and there is nothing left to read from the handle, `hGetChunk h` signals this by returning an empty chunk.

As a demonstration of dealing with text in a chunked manner, the follow program reads from the file `greeting.txt`, converts its contents to upper case, and prints the result.

```
printFileContentsUpperCase = runResourceT @IO do
  dir <- liftIO getDataDir
  (_, h) <- fileResource (dir </> "greeting.txt") ReadMode
  liftIO (printCapitalizedText h)
```

```
printCapitalizedText h = continue
  where
    continue = do -- 1
      chunk <- T.hGetChunk h -- 2
      case (T.null chunk) of
        True  -> return () -- 3
        False -> do
          T.putStr (T.toUpper chunk)
          continue -- 4
```

1. We’ve named the action that we’re performing repeatedly (`continue`) so that it can refer to itself to proceed in a loop.
2. `T.hGetChunk` reads some text from the handle.
3. If the chunk is empty, then we’ve reached the end of the file, so we

do nothing further.

4. After we've used the chunk, we continue on to the next repetition of this procedure.

Abstracting the loop We saw that the `Text` package provides us with:

1. `(Data.Text.IO.hGetChunk :: Handle -> IO Text)` to read a chunk;
2. `(Data.Text.null :: Text -> Bool)` to determine whether the result is actually a chunk or instead an indication that there is no further input.

We therefore needed to design a recursive loop that repeatedly gets chunks, exiting the loop when the end is reached. This isn't the last time we're going to encounter an API that looks like this; so let's write a more general function that describes only the looping pattern.

```
repeatUntilIO ::
  IO chunk          -- ^ Producer of chunks
-> (chunk -> Bool)   -- ^ Does chunk indicate end of file?
-> (chunk -> IO x)   -- ^ What to do with each chunk
-> IO ()
```

In this function, the `Text`-specific parts of `printCapitalizedText` have been relegated to parameters. The type `Text` is now a type variable called `chunk`, and the operations on the text are generalized as variables `getChunk`, `isEnd`, and `f`. What remains is only the concept of the loop.

```
repeatUntilIO getChunk isEnd f = continue
  where
    continue = do
      chunk <- getChunk
      case (isEnd chunk) of
        True  -> return ()
        False -> do{ _ <- f chunk; continue }
```

Our file capitalization program, rewritten using `repeatUntilIO`:

```
printFileContentsUpperCase2 = runResourceT @IO do
  dir <- liftIO getDataDir
  (_, h) <- -- 1
    fileResource (dir </> "greeting.txt") ReadMode
  liftIO $ repeatUntilIO (T.hGetChunk h) T.null \chunk -> -- 2
    T.putStr (T.toUpper chunk) -- 3
```

1. With a read handle open for `greeting.txt`;
2. For each chunk of text read from the handle;
3. Print the text converted to upper case.



2.3 Exercises

Exercise 4 – Pure text manipulation Use the utilities provided by the `Data.Text` module to define the following functions:

Get only the numeric characters from the text.

```
digitsOnly :: Text -> Text
```

```
λ> digitsOnly (Text.pack "ab c123 def4")
"1234"
```

Capitalize the last letter of the text.

```
capitalizeLast :: Text -> Text
```

```
λ> capitalizeLast (Text.pack "dog")
"doG"
```

Remove one layer of parentheses from the text, if possible, otherwise return Nothing.

```
unParen :: Text -> Maybe Text
```

```
λ> unParen (Text.pack "(cat)")
Just "cat"
```

```
λ> unParen (Text.pack "cat")
Nothing
```

Exercise 5 – Character count The following program prints the number of characters in a file.

```
characterCount :: FilePath -> IO Int
characterCount fp = do
  dir <- getDataDir
  x <- T.readFile (dir </> fp)
  return (T.length x)
```

For example:

```
λ> characterCount "greeting.txt"
12
```

The problem is that `T.readFile` reads the entire file into memory. Fix this by rewriting `characterCount` as a recursive loop over smaller chunks of text read from a `Handle`.

Exercise 6 – When and unless In the `Control.Monad` module, you can find these two convenient functions:

```
when  :: Bool -> IO () -> IO ()
unless :: Bool -> IO () -> IO ()
```

We use these functions to perform an action conditionally.

- ▶ For `when`, the action is performed only *when* the `Bool` is `True`. When the `Bool` is `False`, nothing happens.
- ▶ `unless` is the opposite: The action is performed only when the `Bool` is `False`. In other words, the action is performed *unless* the `Bool` is `True`.

Modify the definition of the `repeatUntil` function to make use of either `when` or `unless`.

For extra practice, write your own definitions of `when` and `unless` instead of using the library definitions.

Exercise 7 – Beyond IO The `repeatUntilIO` function we gave in the chapter has the type:

```
repeatUntilIO ::  
  IO chunk -> (chunk -> Bool) -> (chunk -> IO x) -> IO ()
```

Although `IO` appears several times in this type, the function doesn't actually do anything `IO`-specific, and so it can be generalized to work with any monadic type constructor (for example, `ResourceT IO` instead of `IO`). Let's define a new function, called `repeatUntil`. The first thing we'll do is replace each occurrence of `IO` with a type variable, let's say `m`.

```
repeatUntil ::  
  m chunk -> (chunk -> Bool) -> (chunk -> m x) -> m ()
```

Finish defining the `repeatUntil` function. You will also need to add a constraint to the type signature, because this function cannot work with just *any* type constructor `m`.

Chapter 3



Bytes

There are not, of course, warm, friendly letters inside the computer – only cold, uncaring numbers. In this chapter we concern ourselves with how text is represented in this stark world of bits.

```
« 0110100001100101011011000110110001101111 »
```

3.1 Packed octets

Out of convenience, efficiency, and convention, we work with binary digits in groups of eight. We call these *bytes*, or sometimes *octets* (the prefix *oct-* meaning *eight*).

```
« 01101000, 01100101, 01101100, 01101100, 01101111 »
```

In Haskell we most often use the `Word8` type to represent a byte. The `Data.Word` module also provides types called `Word16`, `Word32`, and `Word64`. Given that a `Word8` is 8 bits, perhaps you can guess what the other types mean from their names.

Use of the `Word8` type implies the following correspondence between bytes and the numbers between 0 and 255:

- `00000000` → 0
- `00000001` → 1
- `00000010` → 2
- `00000011` → 3
- ...
- `11111100` → 252
- `11111101` → 253
- `11111110` → 254
- `11111111` → 255

So the bytes in our example bit string above correspond to the following decimal numbers:

« 104, 101, 108, 108, 111 »

These decimals have no more meaning than the groupings of eight bits for our purposes in this chapter, but we will prefer this representation only because it fits more compactly on the page.

Let's copy these bytes into our code file. We'll come back to them in a moment.

```
exampleBytes = [104, 101, 108, 108, 111] :: [Word8]
```

Packing and unpacking In chapter 2 we discussed why we prefer to deal with chunks of neatly-packed text rather than lists of individual characters. For the same reason, we don't often use `[Word8]` to store a list of bytes. Instead we prefer a *packed* representation. We can find one in the `bytesString` package.

```
BS.pack :: [Word8] -> ByteString
BS.unpack :: ByteString -> [Word8]
```

Just like `text`, the `bytestring` package provides an assortment of `ByteString`-based alternatives to the `String` functions in the base package. For example, we now have a third function called `null`. Like the others, it checks whether a string is empty, but this time the parameter is `ByteString`.

```
null      :: Foldable t => t a -> Bool
T.null    :: Text          -> Bool
BS.null   :: ByteString    -> Bool
```

3.2 Copying a file

We'll now turn again to I/O: reading and writing. We again find some functions for I/O with `ByteString` that resemble the ones we saw for `String` and `Text`.

Writing to a handle:

```
IO.hPutStr :: Handle -> String    -> IO ()
T.hPutStr  :: Handle -> Text      -> IO ()
BS.hPut    :: Handle -> ByteString -> IO ()
```

We'll read `ByteStrings` just like we read `Text`, one chunk at a time:

```
T.hGetChunk :: Handle -> IO Text
BS.hGetSome :: Handle -> Int -> IO ByteString
```

Somebody chose to call this one *get some* instead of *get chunk*, but the gist is the same. Each time we run the `hGetSome` action, it will give us some more bytes read from the handle. When `hGetSome` returns an empty byte string, we know that we have reached the end of whatever input the handle represents, and there are no more bytes to be gotten.

That `Int` parameter lets us specify the maximum chunk size. The `ByteString` that `hGetSome` produces is not guaranteed to have any particular size; although the string that we receive will never be *longer* than the size we requested, it may very well be *shorter*.

Recall that `T.hGetChunk` from the `text` package worked this way too, but it didn't have this `Int` parameter to let us request our own preferred size – it just has a built-in default. The `bytestring` package asks a little more of us. Any value you choose here will be fine, though; it should only affect performance. Once your program is written and working, if speed is critical, then perhaps go back and test to see whether you can get any measurable improvements by adjusting this number. Larger chunks require more memory, but smaller chunks require more loop iterations, so there is a tradeoff here that does not admit a satisfyingly decisive best answer.

For a small demonstration of I/O with `ByteStrings`, we'll again turn to files. In chapter 1 we wrote to a file; in chapter 2 we read from a file; now we'll do both at once. This program *copies* a file, reading from one handle while writing to another.

```
import qualified Data.ByteString as BS
```

```
copyGreetingFile = runResourceT @IO do
  dir <- liftIO getDataDir
  (_, h1) <-
    binaryFileResource (dir </> "greeting.txt") ReadMode
  (_, h2) <-
    binaryFileResource (dir </> "greeting2.txt") WriteMode
  liftIO $ repeatUntil (BS.hGetSome h1 1024) BS.null \chunk ->
    BS.hPutStr h2 chunk
```

```
binaryFileResource ::
  FilePath -> IOMode -> ResourceT IO (ReleaseKey, Handle)
binaryFileResource path mode =
  allocate (IO.openBinaryFile path mode) IO.hClose
```

Since the `Text` and `ByteString` handle-reading functions conveniently follow the same pattern, we have reused the `repeatUntil` function we defined in exercise 7.

Instead of `openFile`, this time we've used `openBinaryFile`. Both functions are defined in the `Pipes.Safe.Prelude` module and have the same type. To appreciate the difference between them, first we have to better understand the distinction and relationship between bytes and characters.

3.3 Character encodings

A `ByteString` value represents a list of bytes (the `Word8` type). A byte is an eight-bit number.

00000000 01100101 11100100 01001110

A `Text` value represents a list of characters (the `Char` type). *Characters* are meant to encompass all of the symbols used in all the world’s writing systems.

A

@

ß

犬

Unicode The meaning of “character” in Haskell (and nowadays most of the computing industry) is defined by the Unicode standard. The Unicode Consortium, the organization that publishes the standard, has assigned a number to (nearly) any symbol you might ever want to write. The “cent sign” (¢) is character number 162. “North east arrow” (↗) is character number 8,599. The “beamed eighth notes” symbol (♪) is character number 9,835.

Functions named `ord` and `chr` convert back and forth between a character and its Unicode-assigned number.

```
Char.ord :: Char -> Int
Char.chr :: Int -> Char
```

For example, we can confirm that the number for ♪ is 9,835:

```
λ> ord '♪'
9835
```

Since you probably do not have a ♪ character on your keyboard, Haskell also permits you to write any character by writing a backslash followed by the number of the character you want. So the Haskell expressions `'♪'` and `'\9835'` are equivalent.

```
λ> ord '\9835'  
9835
```

When GHCi is asked to print a character that is outside of the ASCII range, it shows it in this backslash format.

```
λ> chr 9835  
'\9835'
```

```
λ> "musical ♪ notes"  
"musical \9835 notes"
```

If you really want the characters themselves to be printed, you can put-Str them explicitly.

```
λ> putStrLn [chr 9835]  
♪
```

```
λ> putStrLn "musical ♪ notes"  
musical ♪ notes
```

Both bytes and characters are valuable notions, and often we only need to concern ourselves with one and ignore other. When we're writing programs to copy files or transmit messages over a network and we don't care about their content, we prefer to think in terms of bits and bytes. When we're writing a document in our natural human language, we want to use the characters in our writing system. But when we author text in a system where all things are represented as bytes, there must be some point at which these two models collide, and we need a mapping between characters and bytes.

There are a lot more characters than bytes. There are only 256 bytes, but Unicode contains over 100,000 characters; and the Unicode Consortium is continually expanding the list in its effort to meet the needs of every language on Earth. So there are a handful of ways to draw a correspondence between [Word8] and [Char], each with its own tradeoffs.

ASCII Let each byte represent one character. This approach has the advantage of being compact and simple, but it doesn't let us use most of the characters. The *A* in the acronym *ASCII* stands for *American*; this standard was designed with only American English in mind, and it is unsuitable for most other languages. It is not even entirely suitable for English, since various everyday symbols like degrees (°) or cents (¢) are not included in the ASCII character set.

UTF-32 Let each sequence of *four bytes* represent one character, so a character requires $(4 \text{ bytes}) \times (8 \text{ bits per byte}) = 32$ total bits. This gives us enough space to use the entire Unicode character set, but unfortunately we've quadrupled the amount of memory, disk space, and network bandwidth we need to store and transmit the text.

UTF-8 and UTF-16 Let the most commonly-used characters be represented by a single byte, and use more bytes (up to four) to represent the rest of the characters. (ASCII and UTF-32 are *fixed-width* encodings, whereas UTF-8 and UTF-16 are *variable-width encodings*.) These are like compressed versions of UTF-32. This has the advantage of not using as much space as UTF-32, but the disadvantage is that working with these encodings is more complicated. Fortunately, we have libraries that mostly hide this additional complexity from us.

There are two character encodings we will use in this book:

1. Much of the HTTP protocol requires use of ASCII, so we will use it out of necessity;

2. For the parts of HTTP messages that let us choose a character encoding, we use UTF-8 because at the current time of writing it is the most common choice for web servers.

We do not need to concern ourselves here with any further details about these character encoding schemes, but we mention them because there are some consequent facts of life that we must deal with whenever we store or transmit text:

To write text as bytes, you must choose an encoding. There is no single pair of functions to convert back and forth between `Text` and `ByteString`. The conversion always depends on what set of characters you care about and which encoding you're using to represent it.

To read bytes as text, you must know how it was encoded. A byte string by itself, bereft of any context, is meaningless. To interpret a byte string as character data and understand what the bytes signify, you need to know what character encoding was used to encode the string.

Encoding can fail. Encoding characters as bytes (converting from `Text` to `ByteString`) is an operation that can fail because (depending on the choice of encoding) not all characters have a byte representation.

Decoding can fail. Decoding bytes to interpret them as characters (converting from `ByteString` to `Text`) is an operation that can fail because (depending on the choice of encoding) not all byte strings represent characters.

For ASCII, encoding may fail (not all characters are ASCII characters), and decoding may fail (not all bytes represent an ASCII character, only bytes between 0 and 127).

For UTF-8, encoding never fails (UTF-8 can represent every character in the Unicode standard), but decoding may fail (not all byte strings are

valid UTF-8; parsing can fail if a byte string is improperly formatted, just like Haskell source code parsing can fail if you forget a closing paren). The types of a couple functions in the `text` package reflect this fact:

```
import qualified Data.Text.Encoding as T
```

```
T.encodeUtf8  :: Text -> ByteString
T.decodeUtf8' :: ByteString -> Either UnicodeException Text
```

3.4 The Show and IsString classes

We just spent a good while talking about why converting between character strings and byte strings is complicated. Now we have to look at some functions that, deceptively, make it look very simple. We do not like these functions very much, but we must address them.

Show The `Show` class, defined in the Haskell Report, specifies how a value appears as the result of evaluating an expression in GHCi.

```
class Show a                                -- Defined in Text.Show
  where
    show :: a -> String
```

When we evaluate an expression in the REPL like `4 + 5`, the expression is converted implicitly to `putStrLn (show (4 + 5))`, which is how the result 9 ends up appearing to us on the screen.

```
λ> 4 + 5
9
```

```
λ> putStrLn (show (4 + 5))
9
```

IsString The `IsString` class is related to a GHC extension called `OverloadedStrings`, which enables us to write expressions that look like `String` literals but evaluate to some type other than `String`.

```
class IsString a -- Defined in Data.String
  where
    fromString :: String -> a
```

When the extension is enabled and we write an expression in quotation marks like `"hello"`, this is converted implicitly to `fromString "hello"`.

instance IsString Text The overloaded strings feature is convenient when working with the `Text` type. Consider the following small example that packs three strings and concatenates the resulting text:

```
λ> import qualified Data.Text as T
```

```
λ> T.pack "a" <> T.pack "b" <> T.pack "c"
"abc"
```

This might be a good time to remind ourselves that the ways that GHCi displays things sometimes make the type opaque. In the REPL, `Word8` and `Int` values often look identical, and so do most string types, regardless of whether they are `String` or `Text`. You may want to consider setting the `+t` option in your REPL sessions to automatically print the type of the evaluated expression.

```
λ> :set +t
```

```
λ> x :: Word8; x = 223  
x :: Word8
```

```
λ> y :: Int; y = 223  
y :: Int
```

```
λ> x  
223  
it :: Word8
```

```
λ> y  
223  
it :: Int
```

The implementation of `fromString` for the `Text` type is defined as `Data.Text.pack`; in other words, the overloaded strings extension can automatically pack strings into `Text` for us, which can declutter code that involves many text constants.

```
λ> :set -XOverloadedStrings
```

```
λ> ("a" <> "b" <> "c") :: T.Text  
"abc"  
it :: T.Text
```

Implicit conversion from `String` to `Text` is reasonably safe. These two types essentially mean the same thing – they both represent sequences of

Unicode characters – and so there is really nothing left up to the imagination about what `fromString` will do, and it is difficult to imagine how any distinction between `String` and `Text` might adversely affect the meaning of our code.

instance Show ByteString Let's go back to that list of bytes we saved at the beginning of the chapter.

```
λ> exampleBytes
[104,101,108,108,111]
```

Now we'll pack these `Word8` values into a `ByteString`.

```
λ> BS.pack exampleBytes
"hello"
```

GHCi prints this `ByteString` value as the string `"hello"`, which, disconcertingly, seems to run contrary to everything we've said so far about the importance and subtle complexity of the distinction between bytes and characters.

What GHCi shows us is controlled by the `Show` instance, and the authors of the `ByteString` type have defined this instance as:

```
instance Show ByteString
where
    show = Data.ByteString.Char8.unpack
```

All of the functions in the `Data.ByteString.Char8` module use a very simple ASCII-like mapping between `Word8` and `Char` where each byte is taken as a number between 0 and 255 and translated with the `chr` function to its corresponding Unicode character. Below is a portion of the translation

table between decimal numbers and characters. 104 corresponds to 'h', 101 corresponds to 'e', and so on.

97	a	101	e	105	i	109	m
98	b	102	f	106	j	110	n
99	c	103	g	107	k	111	o
100	d	104	h	108	l	112	p

This is something you have to remember when you see `ByteString` values printed in the REPL; the `Show` instance treats each byte in the string as a character in a fixed-width 8-bit encoding, regardless of what the bytes actually signify in the context of your program. We recommend never allowing the behavior of your program to depend on the `show` function. The `Show` class exists for the sake of `GHCi` and for testing; not for real use in an application.

The `IsString` instance is a bigger trap to look out for.

instance IsString ByteString The `fromString` implementation for `ByteString` is roughly the inverse of `show`.

```
instance IsString ByteString
where
    fromString = Data.ByteString.Char8.pack
```

Notice that it's `Char8`: this specific encoding is only appropriate for 8-bit characters, and all others will be ruthlessly made to fit!

In the REPL example below, `fromString` is implicitly converting the string literal "hello" into a `ByteString` and then we use `BS.unpack` to print that `ByteString` as a list of `Word8` values.

```
λ> :set -XOverloadedStrings
```

```
λ> BS.unpack "hello"  
[104,101,108,108,111]
```

If you recall that there are hundreds of thousands of characters but only 256 bytes, you should find this fact alarming. The behavior of `fromString` on characters that correspond to numbers above 255 is surprising. Again, we'll convert a `String` to a `ByteString` and then unpack it.

```
λ> "𐀀" :: BS.ByteString  
"k"
```

```
λ> Data.ByteString.unpack ("𐀀" :: BS.ByteString)  
[107]
```

The number that the Unicode standard assigns to the “beamed eighth notes” character (𐀀) is 9,835. The `fromString` function for `ByteString` crudely truncates this number down to eight bits, interpreting it instead as character number 107, which happens to be the lower-case letter k.

```
λ> 9835 `mod` 256  
107
```

```
λ> chr 107  
'k'
```

None of the details here are particularly important to follow. The take-aways are:

- Libraries often include these sorts of lossy conversion functions without much warning, so be careful.

- You should only use `Data.ByteString.Char8.pack` on characters that you know for sure Unicode assigns to a number between 0 and 255. (This includes all of ASCII, which consists of the characters numbered between 0 and 127.)

For clarity's sake, we will never enable the `OverloadedStrings` extension in this book's code examples from here on.

```
λ> :set -XNoOverloadedStrings
```

Whenever we convert between characters and bytes, our code will make apparent what encoding is at play.

3.5 Avoiding system defaults

With everything we know about character encodings in mind, let's take another look at the *hello world* program from chapter 1.

```
helloHandle = IO.hPutStrLn IO.stdout "hello world!"
```

What ultimately gets written to the `stdout` handle is a string of bytes, not characters. Somewhere within the implementation of the `hPutStrLn` function, the characters of greeting that we wrote are getting converted to a byte string. But remember, there is no single way to convert characters to bytes, and at no point in this program did we specify which character encoding we want it to use. So how does the `hPutStrLn` function decide what character encoding to use by default?

Default system encoding To be honest, we don't know all the details. It uses what the `System.IO` module documentation refers to as “the default encoding on your system”, which has something to do with environment variables. The reason this concept exists is that for a lot of programs, you

don't care exactly what character encoding you're using – but what you *do* care about is making sure that all of the programs on your computer are using the *same* encoding, so that the output written from one program can be read as input by another program without losing any meaning in translation. So, in theory, each machine has a single default encoding, and as long as all of the programs running on that machine respect the system's default encoding, then they can interoperate.

But once we start writing network applications, the idea of a single “system” default encoding no longer suffices. In the next chapter, we will start to discuss the way we send bytes between two different machines. In a network context, we must take care to ensure that our software does *not* rely on our system's default encoding, because our bytes are going to end up on a different system that may prefer a different encoding. It is important that the encodings used in network communication depend not on our own local preferences, but on the rules established by a network protocol.

Line termination Another concept, separate from but handled similarly to character encodings, is line terminators. Most conventions dictate ending each line of text with a *line feed* character – Unicode character number 10, often written as `\n`. The Microsoft Windows convention, however, is to end lines of text with a two-character sequence: first a *carriage return* character – Unicode character number 13, often written as `\r` – followed by a line feed character.

Again here we see that the `hPutStrLn` function automatically does what is most appropriate for system that the code is running on. This is quite convenient in many cases, since it lets us write a program like `(hPutStrLn stdout "hello world! ")` which will produce the contextually-appropriate line termination characters without us having to think about it. But, again, in the context of network applications, we *do* need to think about it, because on the Internet, the Linux machines and the Windows machines need to be able to get along together, and so they must speak a common language.

We will end this chapter with two more implementations of *hello world*. This time, instead of using system defaults, we will write programs that always produce exactly the same bytes as output, no matter what system they are running on. We will print “hello world!” in a UTF-8 encoding with a Unix-style line terminator.

Binary mode By default, the functions in the `System.IO` module that deal with `Handles` assume that you’re dealing with text and try their best to treat it in context-appropriate ways as we have discussed above. They will use the system encoding, and they will automatically detect line terminators and convert them to match the platform’s preferred style. To disable these features when the data is not text – or when you have encoded it all yourself and don’t want the library to mess with it in any way – put the handle into *binary mode*.

```
IO.hSetBinaryMode :: Handle -> Bool -> IO ()
```

What the authors mean by “binary” is “not text.” A file is described as a *binary* if it has no meaningful interpretation as text. (For example, this is why compiled programs are referred to as “binaries”; in contrast with the source code, which is text, the result of compilation is not text.) Setting up a handle in *binary mode* means taking a more reductionist perspective on the data: bits are just bits and, and far as the read and write operations are concerned, they are only to be passed along blindly with no interpretation.

This is the reason that earlier in this chapter we switched from `openFile` to `openBinaryFile`.

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

The `openFile` function opens a file with the handle in text mode,

whereas `openBinaryFile` opens a file with the handle in binary mode.

We conclude this chapter with two more *hello world* programs. For the first, we explicitly list each byte:

```
helloByteString = do
  IO.hSetBinaryMode stdout True
  BS.hPut stdout (BS.pack helloBytes)
```

```
helloBytes = [
  104, 101, 108, 108, 111,      -- hello
  32,                            -- space
  119, 111, 114, 108, 100, 33, -- world!
  10 ]                          -- \n
```

In the second version, we produce these same bytes somewhat more conveniently by encoding a `Text` value:

```
helloUtf8 = do
  IO.hSetBinaryMode stdout True
  BS.hPutStr stdout (T.encodeUtf8 (T.pack "hello world!\n"))
```



3.6 Exercises

Exercise 8 – A character encoding bug The following function accepts the UTF-8 encoding of a person’s name and prints a greeting for that person.

```
greet :: BS.ByteString -> IO ()
greet nameBS = case T.decodeUtf8' nameBS of
    Left _ -> putStrLn "Invalid byte string"
    Right nameText -> T.putStrLn (T.pack "Hello, " <> nameText)
```

For example:

```
λ> greet (T.encodeUtf8 (T.pack "David Hilbert"))
Hello, David Hilbert
```

How might we accidentally use the `greet` function *incorrectly* if we did not know that it requires the UTF-8 encoding? Load this code into GHCi and try to apply it in a way that produces a funky result.

Hints:

- ▶ You may need to look through the `Data.Text.Encoding` module.
- ▶ People with non-ASCII characters in their names tend to notice encoding bugs more often than other people.

Exercise 9 – Byte manipulation If we study how a particular character encoding works, it can be fun (and sometimes advantageous to a program's speed) to write text manipulation functions that operate directly over the bytes in a `ByteString`.

Fill in the blank in the code below to write a function that converts any lower case characters in an ASCII-encoded string to upper case.

```
asciiUpper :: BS.ByteString -> BS.ByteString
asciiUpper = BS.map _
```

The `map` function in `Data.ByteString` is similar to the `map` function for

lists. It modifies a byte string by applying a function to each byte in the string.

```
map    :: (a -> a)          -> [a]          -> [a]
BS.map :: (Word8 -> Word8) -> ByteString -> ByteString
```

Bytes can be manipulated just like integers; the `Word8` type has instances of `Eq`, `Ord`, `Num`, etc.

You also need to know a little about the ASCII encoding:

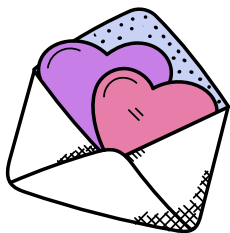
- ▶ The upper case letters A through Z correspond to the numbers 65 through 90.
- ▶ The lower case letters a through z correspond to the numbers 97 through 122.

So, for example, every time our function encounters byte 97 (lower case a), it should map it to byte 65 (upper case A).

A correctly implemented `asciiUpper` function should give the following result:

```
λ> asciiUpper (fromString "Hello!")
"HELLO!"
```

Chapter 4



Sockets

Operating systems provide us an interface for sending and receiving data on the internet that is surprisingly similar to writing and reading files. The network is made more difficult than the filesystem by a few differences:

- ▶ Whereas a file is located by a filesystem path (e.g. “/home/chris/notes.txt”), a socket is located by a *socket address*. Socket addresses are a bit more complicated than filesystem paths.
- ▶ Unlike filesystem interactions where our process is always the initiator of the conversation, on the internet we can make ourselves available and wait for people to talk to us. (Such a process is called a *server*.)
- ▶ Time is an unavoidable concept in networking, because if a remote computer unexpectedly ceases communication, our software must after some *timeout* period give up rather than wait indefinitely.

We’ll still be talking about handles. This time instead of file handles, they’ll be called *socket handles*. But the idea is the same: a socket handle is an identifier that the operating system assigns us to refer to a particular

conversation. A communication channel managed by the OS is a socket; the identifier that a process uses to refer to that channel is a socket handle.

Again, we have chosen to prefer the Windows nomenclature. The Linux term is *socket descriptors*, and the Linux literature emphasizes that a socket descriptor is actually just a kind of file descriptor; but this observation is of little consequence to us.

Using a socket handle is a lot like using a file handle. We can ask the OS to open a socket and give us a socket handle, we can read and write using the socket handle, and we should close the socket when we're done. But now instead of interacting with a local filesystem, we're interacting with a remote computer.

The core Haskell library that does these networking system calls for us is called `network`. You may notice some aspects of this package that seem unusual for Haskell; this is because `network` is a somewhat low-level library that does not abstract much over the underlying system calls it relies upon. Its `Network.Socket` module contains the `Socket` type which we'll be using throughout the book. If you look at the definition of `Socket` in the source code of the `network` package, you can see that it consists of little more than an integer; that is the socket handle assigned by the operating system. The actual mechanics of the socket are managed by the operating system.

```
import Network.Socket (Socket)
import qualified Network.Socket as S
import qualified Network.Socket.ByteString as S
```

4.1 Open up and connect

They say the internet is a place where you can talk to anyone in the world, so let's see if we can use it to meet new people. Here is a general outline of

how our program will proceed:

```
makeFriend :: S.SockAddr -> IO ()
makeFriend address = do
    s <- S.socket S.AF_INET S.Stream S.defaultProtocol -- 1
    S.connect s address -- 2
    S.sendAll s $ T.encodeUtf8 $ -- 3
        T.pack "Hello, will you be my friend?"
    repeatUntil (S.recv s 1024) BS.null BS.putStr -- 4
```

1. Open a socket;
2. Connect to some network address (not sure who we want to meet yet, though!);
3. Send them a welcoming message;
4. Listen politely to whatever they say in return and print it to *stdout* (diligent note-taking is an important part of friendship).

socket The `socket` function creates a socket. It corresponds directly to a C function of the same name, and you can run `man 2 socket` at a Linux command prompt to view the C manual pages for this function. It takes three arguments: address family, socket type, and protocol number.

Address family The `Family` type has a lot of constructors! Some are for specific kinds of hardware like Bluetooth. Some are historical relics. Only a few are really supported well by the `network` library:

Network protocol	Family name
Internet protocol, version 4	AF_INET
Internet protocol, version 6	AF_INET6
The Unix domain	AF_UNIX

AF_INET stands for *address family: internet*, which more specifically refers to IPv4 (internet protocol version 4). With the benefit of hindsight, we can say that it would be nice if AF_INET had been named AF_INET4; but AF_INET was, presumably, named at a time when no other other version of the Internet protocol was prevalent.

The Unix domain is not really a network protocol; it is for inter-process communication within a single machine. For example, a Unix domain socket can serve as the communication channel between a GUI application and a display management system like X or Wayland. As the name implies, Unix sockets are not available on Windows.

The phrase “address family” here refers to the format of network addresses – for example, an IPv4 address is a string of four bytes and an IPv6 address is a string of 16 bytes – but this phrase is a metonym. That is, specifying AF_INET signifies that we want the OS to use the IPv4 protocol, which includes a lot of details, including how packets are structured, not merely the address format. The address family is, however, the only aspect of the network protocol that will be visible to us.

Socket type The second argument to `S.socket` is the *socket type*, which gives us a handful of choices related to what degree of low-level control we want over the data transmission. For some particular needs, you may get better performance out of using a socket type that, for example, does not guarantee that all of the data arrives in the same order in which it was sent. But we will only be using `Stream` sockets, which means we want the OS to take care of *all* of the gritty networking concerns for us, and let us read a stream of bytes just as if we were reading them from a file.

Protocol number The *protocol number* is an elaboration upon the address family, allowing us to provide a specific version of a network protocol that has undergone revisions as people have improved it over the years. We simply use `defaultProtocol` here without thinking about this one too hard,

but it is worth taking a moment to observe this parameter because you will see this pattern over and over in networking: we always plan ahead and give version numbers to any interoperability standards, because any time two machines talk, they need to know they’re speaking exactly the same language.

recv The `recv` function in the `network` package follows the same pattern as the `Text` and `ByteString` handle-reading functions, again allowing us to reuse the `repeatUntil` function we defined in chapter 2. `recv` is an abbreviation for “receive”; it listens for a chunk of incoming message data, returning an empty (“null”) `ByteString` once all data has been read.

4.2 Extra details

We have a working start on the `makeFriend` function, but it isn’t quite complete without a few important adjustments.

Closing the socket Just like file handles, any socket that gets opened must eventually be closed. So we’ll run in a `ResourceT` context to ensure that an open is always paired with a close. The resource-safe version of the `friend`-ship function will then look like this:

```
makeFriendSafely address = runResourceT @IO do
  (_, s) <- allocate
    (S.socket S.AF_INET S.Stream S.defaultProtocol)
    S.close
  liftIO do
    {- ... connect, send, listen, etc. ... -}
    S.gracefulClose s 1000
```

There are two ways to close a socket: gracefully and abruptly. We make use of both.

- The graceful method (`gracefulClose`) is what should happen under normal circumstances. This sends a message across the network to inform the other machine that we intend to close the socket on our end. This helps out the peer, because it allows them to go ahead and close their socket as well, instead of having to wait and then eventually give up hope after some timeout period elapses. We do this because our machine has a responsibility to its peer to be a courteous user of the network.
- We employ the abrupt method (`close`) to quickly bail out when something goes wrong. If the `ResourceT` block halts abnormally in response to an asynchronous exception, our thread has a responsibility to its own machine to clean up and halt as quickly as possible. Therefore the resource release action should forego the social niceties and destroy the socket unceremoniously.

If the entire action completes without an exception, it will run both: first `gracefulClose`, then `close`. This is okay because `close` has no effect after the socket is already closed.

Timeout setting The default timeouts are fairly long. If you try to connect to an address that doesn't respond, it might take a minute or so before the connect action fails and throws an exception. This makes sense for some applications, but it will become tiresome when you're testing things out in GHCi. Here, we prefer to give up quickly if nobody seems to be responding.

Fortunately, the timing is configurable. It is one of many options that can be used to modify a socket's behavior using the `setSocketOption` function.

```
S.setSocketOption :: Socket -> SocketOption -> Int -> IO ()
```

We will insert the following line immediately after creating the socket:

```
S.setSocketOption s S.UserTimeout 1000
```

In both `gracefulClose` and `setSocketOption`, times are specified in milliseconds, so we write one second as `1000`.

With the `ResourceT` resource management and the timeout configuration inserted, the `friendship` function now looks like this:

```
makeFriendSafely address = runResourceT @IO do
  (_, s) <- allocate
    (S.socket S.AF_INET S.Stream S.defaultProtocol)
    S.close
  liftIO do
    S.setSocketOption s S.UserTimeout 1000
    S.connect s address
    S.sendAll s $ T.encodeUtf8 $
      T.pack "Hello, will you be my friend?"
    repeatUntil (S.recv s 1024) BS.null BS.putStr
    S.gracefulClose s 1000
```

4.3 Names and addresses

Let's get ourselves out there and find an address to talk to. So what is an address? If we look at the `SockAddr` type, we see that the network library supports three types of addresses:

```
data SockAddr =
  SockAddrInet PortNumber HostAddress
| SockAddrInet6 PortNumber FlowInfo HostAddress6 ScopeID
| SockAddrUnix String
```

1. `SocketAddrInet` is for addresses in the `AF_INET` (IP version 4) family.
2. `SocketAddrInet6` is for addresses in the `AF_INET6` (IP version 6) family.
3. `SocketAddrUnix` is for `AF_UNIX` (Unix-domain) sockets.

Your system and network may support IPv6, but we've chosen IPv4 for our demonstration because it is still the most widely supported and its socket addresses are simpler. The constructor we use has to match the address family we specified when creating the socket, so since we used `AF_INET` earlier, we have to use the `SocketAddrInet` constructor. It takes two arguments:

1. **A port number.** A computer contains multitudes, and so it needs more than one internet address. The port portion of a socket address is like a tiny little “attention:” line used to distinguish, within a particular machine, which of its many identities we want to talk to.
2. The address of the machine itself. This is called the *host address* because every specific field of computing has its own little quirks of terminology, and in the context of networking, we can't call a computer a “computer” anymore – we have to call it a “host”.

No sense in fretting about this any longer, let's just pick some random numbers and see what happens!

```
λ> makeFriendSafely (S.SocketAddrInet 874
    (S.tupleToHostAddress (28, 63, 3, 178)))
*** Exception: Network.Socket.connect: Connection timed out
```

Oh dear, not much happened at all. When you run this, you will see it pause for one second while waiting for anyone to respond. After a second, the timeout kicks in and throws an exception. At least no one was there to witness this faux pas.

Let's try again, but we'll prepare first this time. We happen to know

that there exists a lovely website named `www.haskell.org`, and a common command-line utility called `host` that can look up a name to find its corresponding IP address.

```
$ host www.haskell.org
www.haskell.org is an alias for www-combo-origin.haskell.org.
www-combo-origin.haskell.org has address 147.75.67.13
www-combo-origin.haskell.org has IPv6 address
    2604:1380:0:8900::b
```

There, on the second line, is an IP version 4 address! There's somebody we can talk to: `147.75.67.13`. (This address may change over time, so if you are following along, note that you may see something different.)

We also happen to know that the conventional port number for a web server is 80. So if we plug those numbers into the `makeFriend` function, we can finally –

```
λ> makeFriendSafely (S.SockAddrInet 80
    (S.tupleToHostAddress (147, 75, 67, 13)))
HTTP/1.1 400 Bad Request
Server: nginx/1.14.0 (Ubuntu)
Date: Wed, 12 Feb 2020 01:29:48 GMT
Content-Type: text/html
Content-Length: 182
Connection: close

<html>
<head><title>400 Bad Request</title></head>
<body bgcolor="white">
<center><h1>400 Bad Request</h1></center>
<hr><center>nginx/1.14.0 (Ubuntu)</center>
</body>
</html>
```

Well... We made a connection, at least. But the experience still leaves much to be desired in terms of social fulfillment. We've found someone out there and sent them a message, and they've sent us a response. All the response says, however, is that they don't understand our message.

The response began "HTTP/1.1" – This tells us what we need to study next. Before we go on to HTTP, let's make one more improvement to the program. Instead of using the `host` program to look up an IP address, we can use the `network` package.

4.4 Address information

There are two "address" types in the `network` library:

- ▶ `SockAddr`, which we just discussed, is useful only once you know ex-

actly what protocol and what type of socket you'll be using.

- `AddrInfo` consists of a `SockAddr` plus all the information about the protocol that you need to construct a socket and use the address. The address is just the raw numbers; the full *address information* puts those numbers in context.

getAddrInfo This function is the Haskell equivalent to the `host` command.

```
addrInfo :: Maybe AddrInfo    -- Hints
          -> Maybe HostName    -- e.g. "www.haskell.org"
          -> Maybe ServiceName -- e.g. "http"
          -> IO [AddrInfo]
```

Address resolution is more divination than science. There is no one correct `AddrInfo` for any given host and service; there may be a variety of ways to access the same service.

- A website might be available by both the IPv4 and IPv6 protocols. This allows the service to support a wider variety of clients, as some network users are limited to one protocol or the other.
- For each protocol, there may be more than one address, probably corresponding to different physical machines that all provide the same service. This sort of redundancy allows a service to remain available even if some of the machines crash.

The first argument to `addrInfo`, referred to as the “hints”, is a suggestion that we provide to the address resolution process that means “I am looking for an `AddrInfo` that resembles this one.” Start with the `defaultHints` constant and use record update syntax to specify the aspects you care about.

If you are only interested in IPv6 addresses, for instance, your address

hints argument would be:

```
Just S.defaultHints{ S.addrFamily = AF_INET6 }
```

Try running the following commands in the REPL:

```
λ> traverse_ print =<< S.getAddrInfo Nothing  
    (Just "www.haskell.org") (Just "http")
```

```
λ> traverse_ print =<< getAddrInfo  
    (Just S.defaultHints{ S.addrSocketType = S.Stream })  
    (Just "www.haskell.org") (Just "http")
```

Notice how the set of results diminishes as you specify more “hints”.

The result we obtain from `getAddrInfo` is a list in which all of the entries are supposedly different ways of reaching the same service. The list is ordered (to the best of the underlying library’s knowledge) with the most preferably addresses on top. If an application fails to make a connection using the first address, and it may continue on down the list, trying the alternatives one at a time.

We will take a simpler approach here and just grab the first `AddrInfo`, ignoring the others, and failing with an exception if the list is empty. To produce an exception, we use `fail` from the `MonadFail` class.

```
fail :: MonadFail m => String -> m a -- from Control.Monad.Fail
```

```

findHaskellWebsite :: IO S.AddrInfo
findHaskellWebsite = do
    addrInfos <- S.getAddrInfo
        (Just S.defaultHints { S.addrSocketType = S.Stream })
        (Just "www.haskell.org")
        (Just "http")
    case addrInfos of
        [] -> fail "getAddrInfo returned []"
        x : _ -> return x

```

Recall that `makeFriendSafely` specified `AF_INET` as the address family, so it only works with IPv4. By changing the parameter type from `SockAddr` to `AddrInfo`, we can write a new version of this function that will be more flexible.

openSocket We previously used the `socket` function to open a socket. There is a second option, called `openSocket`, which we now employ. This does the same thing, but it is more convenient when we have an `AddrInfo`, whose information includes the `Family`, `SocketType`, and `ProtocolNumber` that are needed to construct a socket.

```

socket :: Family -> SocketType -> ProtocolNumber -> IO Socket
openSocket :: AddrInfo -> IO Socket

```

The final version of the friendship function is as follows:

```

makeFriendAddrInfo :: S.AddrInfo -> IO ()
makeFriendAddrInfo addressInfo = runResourceT @IO do
    (_, s) <- allocate (S.openSocket addressInfo) S.close    -- 1
    liftIO do
        S.setSocketOption s S.UserTimeout 1000
        S.connect s (S.addrAddress addressInfo)              -- 2
        S.sendAll s $ T.encodeUtf8 $
            T.pack "Hello, will you be my friend?"
        repeatUntil (S.recv s 1024) BS.null BS.putStr
        S.gracefulClose s 1000

```

1. Using `openSocket` instead of `socket`
2. Using the `addrAddress` function to get the `SockAddr` from the `AddrInfo`

Now our program will be able to automatically select either IPv4 or IPv6, depending on whichever the system deems the better choice.

Run the program in the REPL as follows:

```

λ> makeFriendAddrInfo =<< findHaskellWebsite

```



4.5 Exercises

Exercise 10 – Improper ResourceT allocation Since the ‘connect’ step is part of the initial socket setup that we have to get out of the way before we do anything else with the socket, we might want to combine `openSocket` and `connect` into a single function. Let’s call it `openAndConnect`:

```

openAndConnect ::
  S.AddrInfo -> ResourceT IO (ReleaseKey, Socket)
openAndConnect addressInfo = allocate setup S.close
  where
    setup = do
      s <- S.openSocket addressInfo
      S.setSocketOption s S.UserTimeout 1000
      S.connect s (S.addrAddress addressInfo)
      return s

```

There is a subtle defect in this `openAndConnect` function. What will happen if the connect step fails? Will `ResourceT` be able to do its job? How can we fix it?

Exercise 11 – Explore Gopherspace The Gopher protocol emerged in the early 1990s, around the same time as HTTP and with a similar purpose. Although its popularity today is severely diminished, some Gopher servers still exist – but they are not accessible from your typical web browser, which speaks HTTP, not Gopher.

We will not be learning about Gopher in this book. But, because it one of the simplest text-based protocols, it makes for a fun quick demonstration of what you can do using only the code we’ve written in this chapter.

Define a variant of the `findHaskellWebsite` action with the following modifications:

- Change `"http"` to `"gopher"`.
- Change `"www.haskell.org"` to one of the following (we have given multiple options in case some of these servers are no longer operational by the time you read this):
 - `"quux.org"`

- "gopher.floodgap.com"
- "gopher.metafilter.com"

Then send "\r\n" and see what the server sends back. This is the most basic Gopher request; it means “give me an overview of what’s on your server”. If all goes well, you should see some text that is the Gopher equivalent of a website’s home page.

Exercise 12 – Address resolution In networking parlance, looking up an address is described as “resolving” an address. Generalize the `findHaskellWebsite` we wrote in this chapter to implement the following address resolution function:

```
resolve :: S.ServiceName -> S.HostName -> IO S.AddrInfo
```

We will use this `resolve` function in the next chapter.

Chapter 5

HTTP



In the previous chapter, we opened a connection to `www.haskell.org` on the port conventionally designated for HTTP, sent a message to the machine on the other end, and read the response that it sent back. The response we got, however, just said “Bad Request” – because we have not yet learned to write proper messages that the remote HTTP service can understand.

First published in 1991, HTTP (Hypertext Transfer Protocol) along with HTML (Hypertext Markup Language) was devised as a system for transferring documents. The protocol went like this:

1. A document *server* listens for incoming connections.
2. A *client* who wants a document initiates a connection with the server (as we have done in the previous chapter).
3. The client sends a short message specifying which document it wants.
4. The server responds by transmitting the document (in HTML format) and then closing the connection.

This book follows a more recent version of HTTP published in 2014, which has some additional features:

- ▶ A client can send documents to a server, not only request them.
- ▶ The “documents” can be any kind of data, not only HTML pages.
- ▶ Both the requests and the responses may contain additional meta-data (for a variety of purposes; this book will discuss a few).
- ▶ Instead of always opening a new connection for each request, it is possible to make multiple requests over a single connection.

We will begin with simple examples that closely resemble the fledgling concept of HTTP from 1991.

5.1 The specification

The specification for exactly how HTTP works comes from the Internet Engineering Task Force, which is an organization that develops various standards for the internet. When the IETF proposes a new standard, they call the document a *Request for Comments*, abbreviated as ‘RFC’.

A brief history of HTTP specifications:

Year	HTTP version	IETF document
1991	HTTP/0.9	–
1996	HTTP/1.0	RFC 1945
1997	HTTP/1.1	RFC 2068
1999	HTTP/1.1	RFC 2616
2014	HTTP/1.1	RFC 7230

Notice that HTTP version “1.1” has been published three times, which can create a bit of confusion. You may see people still refer to RFC 2616, but it was made obsolete by RFC 7230 and can now be disregarded.

Don't be put off by the formal and somewhat old-fashioned IETF aesthetic; as you read this book, you should also read the official HTTP specification. We won't need the *entire* thing, but we are going to reference specific sections throughout the book.

<https://tools.ietf.org/html/rfc7230>

Begin by reading the abstract and the table of contents from RFC 7230. Never neglect the table of contents: it helps you understand the scope of the document and the context of the specific parts you'll be reading.

Then skip to section 2, which talks about the architecture and gives a general description of how HTTP is meant to be used. Read the following subsections:

- ▶ Section 2.1, *Client/Server Messaging*
- ▶ Section 2.2, *Implementation Diversity*

We reiterate here that HTTP is described as a “request/response protocol.” The program that makes the request is called the *client*, and the program that responds to the request is called the *server*. The words ‘client’ and ‘server’ describe the roles of the participants in a conversation; one program may function as a client in some exchanges and as a server in others. Much of the rest of the specification, as well as much of the rest of this book, deals more specifically with what goes into a *request* and a *response*.

5.2 HTTP requests

At the end of section 2.1, they start us off with a helpful example request. It is reproduced here, slightly modified for brevity:


```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3
Host: www.example.com
Accept-Language: en, mi
```

Request line The first line is called the *request line*, and always has three parts.

1. GET here is called the *request method*. The request method is usually either GET or POST, where GET signifies that we're requesting some information from the server, and POST signifies that we're sending some information to the server.
2. The next part, `/hello.txt`, is the path that we're GETting. We're asking the server to send us the contents of a file called `hello.txt`. (Because the content of the response doesn't necessarily have to be read from a file, to be more accurate we should say a "a *resource* named `hello.txt`". We use the word 'resource' abstractly to describe anything that an HTTP path might refer to.)
3. The last part of the request line, `HTTP/1.1`, specifies which version of the protocol we're using. The purpose of this is that you might have a server that supports multiple versions of the HTTP protocol as they change over time, and this lets the client specify which version of the protocol they support. In this book, we're not going to be concerned with switching versions of the protocol; we will always fix this as version 1.1.

Header fields The rest of the request consists of header fields. HTTP has a few dozen standard header fields; this small example has only three.

1. **User-Agent** tells the server what software is being used to make the request (e.g. a browser like Firefox, or in this case a command-line program called `curl`). The HTTP requests that we produce in this book will not include this field, because there is no software *agent* making requests on our behalf; we're doing all the work ourselves. If we fancifully wanted to include this field, it might look like something this:

User-Agent: Sockets and Pipes book

2. The **Host** field makes it possible to run websites for multiple domain names on a single server. For example, we have two websites, `typeclasses.com` and `joyofhaskell.com`, but they both run on the same server. When your browser requests the `"/` resource from one of these sites, it is the value of the **Host** field that determines whether the server responds with the Type Classes home page or the Joy of Haskell home page.
3. There may exist translations of a resource into different languages; the **Accept-Language** field specifies the requester's language preference. In this example, `"en, mi"` signifies a request for text written in either English or Māori.

One header field is given per line, and the end of the list is signalled by an empty line.

5.3 ASCII strings

All of the text in this example request must be transmitted in its ASCII representation. We might transcribe the example into Haskell as follows:

```
helloRequestString :: BS.ByteString
helloRequestString = fromString $
    "GET /hello.txt HTTP/1.1\r\nUser-Agent: curl/7.16.3\r\n" <>
    "Host: www.example.com\r\nAccept-Language: en, mi\r\n\r\n"
```

As the HTTP protocol specifies, each line is ended by `\r\n`, which stands for two ASCII characters: `\r` is a *carriage return*, and `\n` is a *line feed*. In RFC 7230, you'll see this sequence of characters referred to as "CRLF". You may recognize this from chapter 3 as the Microsoft Windows convention (whereas a Unix user might prefer to terminate each line with just a single `\n`), but when speaking HTTP you must use `\r\n` regardless of what operating system you are personally using. On the internet, we must all follow the same conventions to work together.

The definition of `helloRequestString` we've written above feels a bit tedious to read. We find that it becomes especially difficult to think about the correct number of line terminators when strings are written this way, so we'll introduce some concepts to help us write strings like these without drowning in a pool of backslashes.

```
line :: BS.ByteString -> BS.ByteString
line x = x <> fromString "\r\n"
```

```
helloRequestString =
    line (fromString "GET /hello.txt HTTP/1.1") <>
    line (fromString "User-Agent: curl/7.16.3") <>
    line (fromString "Accept-Language: en, mi") <>
    line (fromString "")
```

That seems better – now each line of the source code corresponds to one line of the request, and it's much easier to see that the request ends

with a blank line.

While we're at it, let's clear up one more vexation with how we write strings. Recall from chapter 3 that `fromString` is sort of a strange function for `ByteString` because, for example, `fromString "♫" = "k"`. We shouldn't be trying to include the `♫` character in an HTTP request header in the first place, because HTTP asks us to use only ASCII characters in this part of a message. But if we make a mistake, we would prefer to get a compilation error instead of an erroneous value.

The root of our problem is that we're trying to write ASCII strings, but string literals in Haskell are Unicode strings. The denotation of quotation marks `"..."` is fixed; quoted expressions always represent a list of Unicode characters. So when we use string literals, the compiler is not aware that we mean to restrict ourselves to a smaller character set. What we want is to be able to directly express our intent of writing a list of ASCII characters.

Fortunately, it is possible to define other kinds of string literals beyond the built-in one! We'll need two new tools:

- ▶ the `ascii` package;
- ▶ the `QuasiQuotes` language extension.

You can think of quasi-quotes as a generalization of normal quotes, with the 'quasi-' modifier signalling that there's something that distinguishes them from the ordinary. The `ascii` package provides a quasi-quoter called `string`. Once we import it and enable the language extension, we can replace each quotation of the form `"..."` with a quasi-quotation `[A.string|...|]`.

```
import qualified ASCII as A
import qualified ASCII.Char as A
```

```
helloRequestString =  
  line [A.string|GET /hello.txt HTTP/1.1|] <>  
  line [A.string|User-Agent: curl/7.16.3|] <>  
  line [A.string|Accept-Language: en, mi|] <>  
  line [A.string|]
```

```
crlf :: [A.Char]  
crlf = [A.CarriageReturn, A.LineFeed]
```

```
line :: ByteString -> ByteString  
line x = x <> A.lift crlf
```

In the definition above, the `lift` function from the `ASCII` module has the following type:

```
A.lift :: [ASCII.Char] -> BSB.Builder
```

Its more general type signature is:

```
A.lift :: Lift a b => a -> b
```

In chapter 7, we will see further use of the `lift` function.

Now that the compiler is aware of our intent to write ASCII strings, it can detect our mistakes. If we tried to request `␣` instead of `/hello.txt`, we would see this output from the compiler:

```

error:
  • Must be only ASCII characters.
  • In the quasi-quotation: [A.string|GET ␣ HTTP/1.1|]
|
14 |      line [A.string|GET ␣ HTTP/1.1|] <>
    |                               ^^^^^^^^^^^^^^^^^

```

5.4 HTTP responses

The example in RFC 7230 goes on to give a response to the request. We will use a slightly shorter example response here:

```

HTTP/1.1 200 OK
Content-Type: text/plain; charset=us-ascii
Content-Length: 6

Hello!

```

Status line The first line of an HTTP response is called the *status line*. It contains the protocol version, HTTP/1.1, along with a *status code* that summarizes the result of handling the request. When everything goes well, the status code is usually 200 OK.

Other common status codes you’ve probably seen before are 404 NOT FOUND, which would indicate that `hello.txt` is not something that the server knows about, or 500, which might indicate that the request couldn’t be serviced because something was wrong with the server.

Header fields The response also contains a number of header fields:

1. The Content-Type header specifies what kind of thing we’re going

to return in the response body; the client needs to know whether it is receiving HTML, an image, a video, etc. In this case, we're specifying that it is just plain text.

The first part of the `Content-Type` value here, `text/plain`, is an example of a *MIME type*. ('MIME' is an old acronym for *Multipurpose Internet Mail Extensions* because it was originally defined for the purpose of email. Since email was one of the first things on the internet, a lot of standards originated from it.)

The content type we're using isn't just `text/plain`. It is, more specifically, `text/plain; charset=us-ascii`. Whenever the MIME type is some kind of text, we need to specify the character set so that the recipient can know how to decode the bytes into text.

2. The second header field in our example response is `Content-Length`. This header means that, after the conclusion of the header portion of the message, we'll be sending 6 bytes.

We have to specify up front how long our body is going to be so that the client reading our response knows when to *stop* reading. A message does not arrive all at once; it comes in chunks. A recipient that has received some data must have some way of determining whether there is more data yet to come.

Body The very last part, after the header fields and the blank line, is called the *body* of the message. Since the request was `GET /hello.txt`, the response body here is presumably the contents of some file called `hello.txt`. The text "Hello!" is indeed 6 bytes; it consists of six characters, and each ASCII character is represented by a single byte.

Our expression of this response in Haskell looks much like that of the request:

```
helloResponseString =  
  line [A.string|HTTP/1.1 200 OK|] <>  
  line [A.string|Content-Type: text/plain; charset=us-ascii|] <>  
  line [A.string|Content-Length: 6|] <>  
  line [A.string||] <>  
  [A.string|Hello!|]
```

Notice that we did not apply the `line` function to the body. The orderly structure of an HTTP message – lines of text, each line followed a line terminator – ceases once we reach the end of the header fields.

5.5 Serving others

We went into full detail about how to establish a connection as a client using the low-level network library in chapter 4. Listening for connections as a server is a slightly more involved process; to skip over a bit of tedium, we’re going to rely on the `network-simple` library, which wraps up common network usage patterns into an interface that better suits our needs here.

```
import Network.Simple.TCP (serve, HostPreference (...))  
import qualified Network.Simple.TCP as Net
```

From here on, we will mostly be using `network-simple` instead of `network`. There are two functions we’ve seen already that we’ll replace with their `network-simple` variants:

- Instead of `S.sendAll`, we will use `Net.send`. The only difference between these is that `S.sendAll` is an IO action, whereas `Net.send` is polymorphic with a `MonadIO` constraint.
- Instead of `S.recv`, we will use `Net.recv`. Whereas `S.recv` returns

an empty `ByteString` to indicate the end of input, the `Net.recv version` has a return type of `m (Maybe ByteString)` and signals the end of input by returning `Nothing`.

There is also one important new function:

```
Network.Simple.TCP.serve :: MonadIO m =>
    HostPreference          -- 1
-> ServiceName             -- 2
-> ((Socket, SockAddr) -> IO ()) -- 3
-> m a                     -- 4
```

1. The `HostPreference` parameter asks you to specify who your server is. This may seem like a strange question to have to answer; is it not simply itself? Not quite. On the internet, just as you can have multiple separate personas, so can your machine. It probably has multiple network addresses, such as both an IPv4 and IPv6 address. If it matters to you which identity your server will assume, this is your change to specify. We do not, so we will always use `HostAny` for this argument.
2. `ServiceName` determines what port our server listens on. The standard port for HTTP is 80, so if we give this argument as `"http"`, then our server will listen on port 80. We will instead use port 8000; this number has no formal significance, but it is a conventional choice for HTTP servers that are for test or demonstration purposes only. We write this as the string `"8000"`.
3. The final parameter to `serve` is where we specify what to do each time a client connects to our server. You might sometimes see this sort of function described as a socket “handler” or “callback.” This `Socket` is how we talk to the client.
4. The result is an `IO` (or `IO-like`) action that await and responds to con-

nections initiated by others. It runs indefinitely (until terminated by an exception).

So without further ado, here is our first web server:

```
ourFirstServer = serve @IO HostAny "8000" \(s, a) -> do
  putStrLn ("New connection from " <> show a) -- 1
  Net.send s helloResponseString                -- 2
```

When a new client socket shows up, our server does two things:

1. Print a message that includes the client's network address;
2. Send the HTTP response that we wrote in the previous section.

We should make a few further remarks about aspects of the `serve` function that are not discernible from its type signature:

- ▶ We do not have to worry about closing the socket, because `serve` ensures that the socket is closed after our handler runs (regardless of whether our `IO` terminates normally or throws an exception).
- ▶ Connections are handled concurrently; each time a client socket appears, the handler is run in a new thread. We may never notice the effect of this while testing at a small scale, but out there on the wild internet where servers are rapidly receiving requests from many clients, concurrency helps ensure that a server can respond to each of them quickly.

This is, of course, a very strange web server. We don't read anything from the socket to find out what resource the client was asking for or what language they might prefer to receive it in; we just respond to every request with "Hello!". So we have a service which is perhaps affable but not very useful. To make a server that can do something more interesting than reply with a fixed response, we will need to move beyond strings.



5.6 Exercises

Exercise 13 – Repeat until nothing In exercise 7, we defined `repeatUntil`:

```
repeatUntil ::  
  m chunk -> (chunk -> Bool) -> (chunk -> m x) -> m ()
```

We used this to loop over chunks of input from a socket like so:

```
repeatUntil (S.recv s 1024) BS.null BS.putStr
```

Now we want to switch from `network` to `network-simple`, which has a slightly different interface for receiving from sockets.

```
S.recv :: Socket -> Int -> IO ByteString
```

```
Net.recv :: MonadIO m => Socket -> Int -> m (Maybe ByteString)
```

Instead of using `repeatUntil`, we should make another similar function using `Maybe` instead of a predicate to specify how far the repetition should go. Write a definition for the following type signature:

```
repeatUntilNothing :: Monad m =>  
  m (Maybe chunk) -> (chunk -> m x) -> m ()
```

`repeatUntilNothing getChunkMaybe f` should repeatedly apply `f` to the results successively obtained from `getChunkMaybe`, stopping when `getChunkMaybe` returns `Nothing`.

Can either `repeatUntil` or `repeatUntilNothing` be defined using the other?

Exercise 14 – Make an HTTP request Write a program to submit the following request to `http://haskell.org` and print the response.

```
GET / HTTP/1.1
Host: haskell.org
Connection: close
```

Use the functions that we have written previously: `repeatUntilNothing`, `openAndConnect`, `resolve`, and `line`.

First run your program, then try running `curl http://haskell.org` at the command line. How do the results compare?

Exercise 15 – Test the hello server Run `ourFirstServer` in `GHCi`.

1. While it is running, make requests to it using various HTTP clients. Try experimenting with:
 - a command-line program like `curl`;
 - a web browser like Firefox.
2. A typical internet-connected machine has four network addresses: an IPv4 address, an IPv6 address, the local IPv4 address `127.0.0.1`, and the local IPv6 address `::1`. Try experimenting with:
 - the IPv4 address `http://127.0.0.1:8000`;
 - the IPv6 address `http://[::1]:8000`.

Do you observe any differences in what happens?

Chapter 6

HTTP types



In the previous chapter, we started considering very basic web servers that only send one fixed response. This is the response we wrote for our example:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=us-ascii
Content-Length: 6

Hello!
```

Writing out messages in HTTP format is not a skill that a person needs to learn. Most HTTP requests and responses are produced by a computer and consumed by a computer, quietly and invisibly, without our direct involvement. So now let's more precisely examine the message format and codify in Haskell how to read and write it. This will allow us to write software that is literate in the language of HTTP.

Read the docs Please read the following sections from RFC 7230:

- Section 3, *Message Format*
- Section 3.1, *Start Line* (including the subsections)
- Section 3.2, *Header Fields* (just the beginning, not the subsections)
- Section 3.3, *Message Body* (just the beginning, not the subsections)

<https://tools.ietf.org/html/rfc7230>

We will be transcribing the content of these sections into Haskell. In this chapter, we begin by defining the relevant types.

6.1 Request and response

There are two kinds of message in HTTP: requests and responses. Section 3, *Message Format*, opens by telling us that a message consists of:

- The *start line*
- Zero or more *header fields*
- An empty line
- An optional *message body*

Since an empty line contains no information, that part of the message format will not appear anywhere in our data definitions.

Only the format of the start line differs between requests and responses. For requests, the start line is called the *request line*. For responses, the start line is called the *status line*.

```
data Request =  
    Request RequestLine [HeaderField] (Maybe MessageBody)
```

```
data Response =  
    Response StatusLine [HeaderField] (Maybe MessageBody)
```

So far we have defined `Request` and `Response`. Their fields include types whose names we have taken from RFC 7230 but whose definitions we have yet to write: `RequestLine`, `StatusLine`, `HeaderField`, and `MessageBody`. Next we'll continue to proceed through the document to see how these types should be defined.

6.2 Request line

The request line (the first line of a request) consists of:

- The *method*
- A space
- The *request target*
- A space
- The *protocol version*
- A line break

Again we're ignoring the spaces and line break for now, because they contain no information, so they aren't relevant to the datatype definitions. We're left, then, with three fields: the method, the target, and the version.

```
data RequestLine =  
    RequestLine Method RequestTarget HttpVersion
```

Method In chapter 5 we mentioned that the request method is usually either GET (when requesting information from the server) or POST (when sending information to the server). So we might consider defining it as follows:

```
data Method = Get | Post
```

Another document – RFC 7231, *HTTP/1.1 Semantics and Content* – lists

the names and meanings of some additional methods. After reading that document, we could expand the list from two to eight:

```
data Method = Get | Head | Post | Put | Delete
            | Connect | Options | Trace
```

Even that specification, however, notes that its list is incomplete and subject to future expansion. The HTTP protocol itself does not require the method to be among some fixed enumeration of possibilities. So we will define our type accordingly and permit this value to be any string.

```
data Method = Method BS.ByteString
```

Request target Now let's figure out how to define `RequestTarget`. We have previously seen an example with the following request line:

```
GET /hello.txt HTTP/1.1
```

In that example, the request target was `/hello.txt` – something that looks like a Unix file path, starting with a slash. Seems pretty simple, but let's look at the specification. If we skip down to RFC 7230 section 5.3, *Request Target*, it turns out that the definition of 'request target' is surprisingly involved!

This is a more detail than we care to go into right now, so we'll just gloss over this by again defining this type with `BS.ByteString`. Although the request target has more meaning and rules than this, we'll be fine just thinking of it as an opaque string and not considering it in any further detail.

```
data RequestTarget = RequestTarget BS.ByteString
```


6.3 Status line

The status line (the first line of a response) consists of:

- The *HTTP version*
- A space
- The *status code*
- A space
- The *reason phrase*
- A line break

```
data StatusLine =  
    StatusLine HttpVersion StatusCode ReasonPhrase
```

Status code The status code is “a 3-digit integer code describing the result of the server’s attempt to understand and satisfy the client’s corresponding request.” Two commonly-seen status codes are 200 (okay) and 404 (not found).

We might consider defining `StatusCode` as an `Integer`. 200 and 404 certainly look like integers. But this seems a bit wrong, because:

- A status code doesn’t have the semantics of a number. It would be nonsensical to perform operations like addition or multiplication on status codes.
- Integer is not precise enough; the type we choose should only admit three-digit values. Larger integers like ‘7129’ are not valid status codes, and thus ideally should not be representable using the `StatusCode` type.

RFC 7230 section 1.2, *Syntax Notation*, tells us that a digit is an integer between 0 and 9. The `ascii` library has a `Digit` type, representing the ASCII characters 0 through 9, that corresponds quite well to this definition. So we can transcribe into a type declaration very directly:

```
data StatusCode = StatusCode A.Digit A.Digit A.Digit
```

Reason phrase The last part of a status line is the *reason phrase*, which the specification describes as “a textual description associated with the numeric status code, mostly out of deference to earlier Internet application protocols that were more frequently used with interactive text clients.” In other words, it’s an explanation of the status code for the benefit of human readers who haven’t memorized all the status codes. As far as the machines are concerned, this piece of the message is superfluous.

Since the content of this field does not contain any structure that is meaningful to the protocol, we will just represent it as a `ByteString`.

```
data ReasonPhrase = ReasonPhrase BS.ByteString
```

6.4 Header fields

After the start line, an HTTP message contains a number of header fields. Each header field consists of:

- The *field name*
- A colon (:)
- The *field value*.

```
data HeaderField = HeaderField FieldName FieldValue
```

```
data FieldName = FieldName BS.ByteString
```

```
data FieldValue = FieldValue BS.ByteString
```

Colloquially, a “header field” is often referred to as a “header”. In this book we make some attempt to stick to the terminology that the specification uses, although the temptation toward brevity is difficult to resist.

The “Hello!” example response at the beginning of this chapter has two header fields:

1. The field name `Content-Type` with the field value `text/plain; charset=us-ascii`
2. The field name `Content-Length` with the field value `6`.

The header section serves an interesting hodgepodge of purposes. The meanings of some particular field names are defined as part of the HTTP protocol; we have already discussed the semantics of `Content-Type` and `Content-Length`. However, an HTTP message is also allowed to have header field names that aren’t defined in RFC 7230; you can make up your own headers that have whatever meaning you choose to assign to them.

6.5 Message body

The last part of an HTTP message is the *body*. Not all messages have a body, which is why our `Request` and `Response` types each have a field of type `Maybe MessageBody`.

- ▶ When a client requests a file *from* a server using the `GET` request method, the request does not have a body, and the response does have a body (the content of the requested file).
- ▶ When a client sends a file *to* a server using the `POST` request method, the request does have a body (the content of the file being uploaded), and the response may or may not have a body.

Like many of the types we have defined so far, a message body will be a byte string. But this one is going to be a *lazy* byte string.

The strict and lazy `ByteString` types come from different modules in the `bytestring` package, but the types have the same name. So we must use qualified imports to distinguish them.

```
import qualified Data.ByteString as BS
import qualified Data.ByteString.Lazy as LBS
```

```
data MessageBody = MessageBody LBS.ByteString
```

Why be lazy? Although the efficient memory usage of a packed data structure like `BS.ByteString` is appealing, we sometimes miss the non-strict evaluation that the standard list type enjoys.

You can think of `LBS.ByteString` as roughly equivalent to `[BS.ByteString]` – a lazy string is like a list of strict strings. The strict pieces that combine to constitute a larger lazy value are called its *chunks*. We can see this in the names of the conversion functions:

```
LBS.fromChunks :: [BS.ByteString] -> LBS.ByteString
LBS.toChunks  :: LBS.ByteString -> [BS.ByteString]
```

The most common reason to use a lazy type is when you’re dealing with strings that might be very long. You can take advantage of the “streaming” nature of lazy lists to avoid needing to hold an entire string in memory at once. Whereas the start line and header fields ought to be fairly short, a message body could reasonably be quite long. Imagine a client requests a video file; the HTTP response body will be large.

Using lazy byte strings can help allow for the possibility of transmitting messages that would not reasonably fit in a machine’s memory. Later in this book, we will use `ListT` to accomplish this. The chief virtue of the present lazy-string approach is that it is easy; simply choosing

`LBS.ByteString` instead of `BS.ByteString`, with no further changes, may suffice to improve memory usage.

We have some mundane functions to convert between the lazy and strict variants.

```
LBS.fromStrict :: BS.ByteString -> LBS.ByteString
LBS.toStrict  :: LBS.ByteString -> BS.ByteString
```

`fromStrict` lifts a strict `ByteString` into a lazy context by constructing a list that consists of a single chunk. To convert in the other direction, `toStrict` forces evaluation the entire list, repacking all of the chunks together into a single contiguous strict chunk.

6.6 HTTP version

For both requests and responses, the start line contains an *HTTP version*, which specifies which version of the HTTP protocol specification should be used to interpret the message. The details of the version are given in RFC 7230 section 2.6.

The version number takes the form “HTTP/x.y” where `x` and `y` are both a single digit. Therefore our `HttpVersion` constructor has two `Digit` fields.

```
data HttpVersion = HttpVersion A.Digit A.Digit
```

This is a bit more constrained than a typical software version numbering scheme. For example, “HTTP/3.14” is not a valid version number. In case this seems like a strange choice, the authors offer an explanation for it: In an appendix entitled *Changes from RFC 2616*, RFC 7230 notes that “version numbers have been restricted to single digits, due to the fact that implementations are known to handle multi-digit version numbers incor-

rectly.” What was once a popular mistake eventually became officially accepted into a later version of a protocol.



It may feel like we have accomplished very little so far, but type definitions are important work. The vocabulary we have established through the type definitions in this chapter will be our guide as we proceed to write functions involving these types.

The code in this book should be especially straightforward to write because we have the benefit of an existing specification document; other people have already designed the system for us, and our task here is only to transcribe the words into code. If we were designing the protocol ourselves, the task would be harder, but we would still begin by defining types.

6.7 Exercises

Exercise 16 – Construct some values Using the types that we have defined in this chapter, fill in the blanks in the code given below to define a Request and a Response representing the following two example messages.

1.

```
GET /hello.txt HTTP/1.1
Host: www.example.com
Accept-Language: en, mi
```

```
2. HTTP/1.1 200 OK
Content-Type: text/plain; charset=us-ascii
Content-Length: 6

Hello!
```

```
import ASCII.Decimal (Digit (..))
```

```
helloRequest :: Request
helloRequest = Request start [host, lang] Nothing
  where
    start = RequestLine (Method [A.string|GET|]) _ _
    host = HeaderField _ _
    lang = _
```

```
helloResponse :: Response
helloResponse = Response _ _ _
```

A.`Digit` is an alias for a type called `D10`, which is imported above. Browse the `ASCII.Decimal` module to find its constructors.

Exercise 17 – Infinite byte strings We demonstrated in chapter 2 that a `String` can go on forever. We can still make use of such a value, as long as we never ask for anything that would require the *entire* string.

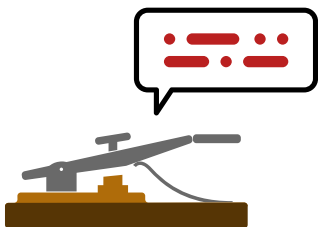
```
λ> take 10 (cycle "abc")
"abcabcbabca"
```

The lazy byte string type `LBS.ByteString` has this property as well. Browse the `Data.ByteString.Lazy` module, find a function for construct-

ing a lazy byte string that never stops, and try to use it in some way that doesn't crash GHCi.

Chapter 7

Encoding



In chapter 6, we defined types to represent HTTP requests and responses. Now we will write an *encoding* function to convert a `Response` into a `ByteString`. Once we have this power, we will no longer have to write HTTP responses by hand, and so it will be easier to make our web server do more interesting and dynamic things.

But first, yet another digression about strings.

7.1 String builders

So far we have seen quite a few types of strings:

- ▶ `String` (a list of characters);
- ▶ `Text` and `ByteString` (packed chunks of characters and bytes); and
- ▶ `LazyText` and `LazyByteString` (lists of chunks of characters and bytes).

Perhaps you thought – or hoped – that we were done with this subject. But there’s one more sort of string that we should know about: `string`

builders. These types are designed to support efficient concatenation, so they are useful when combining many small strings into one big string – such as when building an HTTP response by joining together all of the little parts we discussed in the last chapter.

Both the `Text` and `Text.Builder` packages have a type called `Builder`, in the following modules respectively:

- ▶ `Data.Text.Lazy.Builder`
- ▶ `Data.ByteString.Builder`

We'll start with a small example using the strict `Text` type we've already seen from chapter 2, and then rewrite it using a `Builder`.

```
sayHello :: T.Text -> T.Text
sayHello name = T.pack "Hello, " <> name <> T.pack "!"
```

```
λ> sayHello (T.pack "Alonzo")
"Hello, Alonzo!"
```

The `sayHello` function joins together three strict `Text` values, using two concatenations.

Next comes the `Builder` version. It has the same type signature, and it produces exactly the same result, but we've changed the implementation details.

```
import qualified Data.Text as T
import qualified Data.Text.Lazy as LT
import qualified Data.Text.Lazy.Builder as TB
```

```
sayHelloWithBuilder :: T.Text -> T.Text
sayHelloWithBuilder name = LT.toStrict $ TB.toLazyText $
    TB.fromString "Hello " <>
    TB.fromText name <>
    TB.fromString "!"
```

Every time we use a Builder type, it will follow this same pattern.

1. Each small piece of text gets wrapped up in the Builder type using one of these functions:

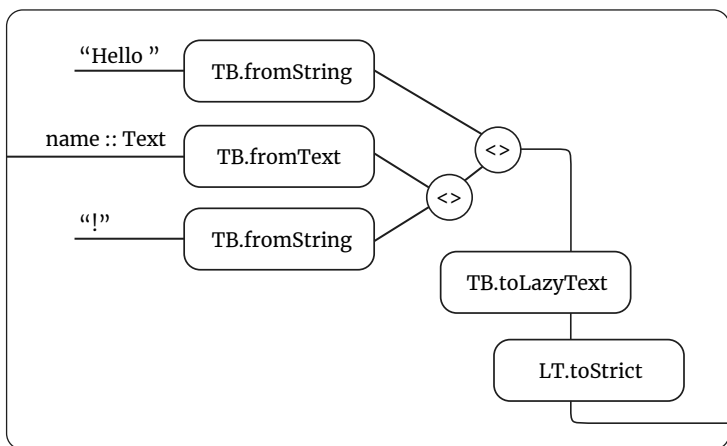
```
TB.fromText      :: T.Text -> TB.Builder
TB.fromLazyText  :: LT.Text -> TB.Builder
TB.fromString    :: String -> TB.Builder
```

2. We join all the little builders into one big builder using the semigroup operator (<>).
3. When we're done concatenating, we convert the builder into a lazy string.

```
TB.toLazyText :: TB.Builder -> LT.Text
```

4. Sometimes we will leave the result as a lazy string; in this case, we further convert the lazy string into a strict one.

```
LT.toStrict :: LT.Text -> T.Text
```



7.2 Measuring time

So what was wrong with just concatenating `Text` values directly?

Each time a chunk of text is incorporated into a larger chunk, a new copy of it is written into memory. In the first `sayHello` example, first “Alonzo” and “!” are joined to create a new chunk: “Alonzo!” – Then “Hello,” and “Alonzo!” will be joined together into the final result. By the time we are done, the letters of the name “Alonzo” appear in memory three times:

1. As original chunk – “Alonzo”
2. Within the intermediate string – “Alonzo!”
3. Within the final result – “Hello, Alonzo!”

This isn’t significant in such a small example, but as the number of chunks increases, the costs of all these copies can add up to a noticeable problem. A full explanation of what’s going on in memory and how the `Builder` type improves upon the situation is outside the scope of this book, but we do at least want to offer a demonstration so that you don’t have to

entirely just take our word for it.

For people who are very serious about speed testing, there exist sophisticated measurement libraries. (One popular choice can be found in a package called *criterion*.) For our purposes here, we will take a more casual approach:

1. Get the starting time;
2. Run the action that we're testing;
3. Get the ending time;
4. Subtract the start time from the ending time to find out how long the action took.

```
import qualified Data.Time as Time
```

```
time :: IO () -> IO ()
time action = do
  a <- Time.getCurrentTime      -- 1
  action                       -- 2
  b <- Time.getCurrentTime      -- 3
  print (Time.diffUTCTime b a) -- 4
```

This will give us a very rough estimate, but we will choose examples with large enough time differences that even a coarse measurement will still get the point across. To do this, we need to write expressions that involve many thousands of concatenations.

```
T.pack "a" <> T.pack "a" <> T.pack "a" <> T.pack "a" <>
T.pack "a" <> T.pack "a" <> T.pack "a" <> T.pack "a" <> ...
```

```
LT.toStrict $ TB.toLazyText $
  TB.fromString "a" <> TB.fromString "a" <>
  TB.fromString "a" <> TB.fromString "a" <>
  TB.fromString "a" <> TB.fromString "a" <>
  TB.fromString "a" <> TB.fromString "a" <> ...
```

We don't want to actually write them all out like this, of course. Instead we will produce these long (<>) chains using two functions both available from the `Relude` module:

replicate The first is `replicate`, which produces a list of an item repeated some number of times:

```
replicate :: Int -> a -> [a]
```

```
λ> replicate 8 (T.pack "a")
["a","a","a","a","a","a","a","a"]
```

fold The second is `fold`, which joins together all the items of a list with successive applications of the (<>) operator.

```
fold :: (Foldable t, Monoid m) => t m -> m
```

```
λ> fold [T.pack "a", T.pack "b", T.pack "c"]
"abc"
```

```
λ> fold (replicate 8 (T.pack "a"))
"aaaaaaaa"
```

Both `replicate` and `fold` are exported by `Relude`, so no additional imports are required. Our two definitions can now look like this:

```
concatWithStrict :: Int -> T.Text
concatWithStrict numberOfTimes =
    fold $ replicate numberOfTimes $ T.pack "a"
```

```
concatWithBuilder :: Int -> T.Text
concatWithBuilder numberOfTimes = LT.toStrict $ TB.toLazyText $
    fold $ replicate numberOfTimes $ TB.fromString "a"
```

Haskell, being lazily evaluated, has a charming tendency to thwart performance measurements by not actually doing the work. So when we want to find out how long it takes to evaluate some expression, we should be in the habit of including in our test code some I/O that actually *uses* the result. We've chosen here to have each test action write the text to a file. Since this effect could not be completed without evaluating the entire `Text` value, we are able to reasonably convince ourselves that the evaluation is actually taking place during the test.

```
concatSpeedTest :: Int -> IO ()
concatSpeedTest n = do
    dir <- getDataDir
    time $ T.writeFile (dir </> "strict.txt")
                (concatWithStrict n)
    time $ T.writeFile (dir </> "builder.txt")
                (concatWithBuilder n)
```

We have selected 50,000 as the number of repetitions after a bit of experimentation, aiming to arrive at a number large enough to show a noticeable difference but not so large that the demonstration would take very long to run. (As with all timing tests, your results may vary.)

```
λ> concatSpeedTest 50000  
2.463692763s  
0.00499423s
```

The result shows us that the fold of many strict texts takes several seconds, whereas the `Builder` variant is nearly instantaneous.

Lazy Text concatenation performance suffers under more particular conditions than strict `Text`, but we won't belabor this digression with another demonstration. Suffice it to say that `Builder` is designed for concatenations, and so we will use it for that purpose.

7.3 Request and response

Our goal for the remainder of this chapter is to write functions that turn HTTP messages (the `Request` and `Response` types from chapter 6) into byte strings. This will require writing a lot of functions that encode the constituent parts of a message. The strings representing the smaller parts will be concatenated together into strings representing larger parts, and eventually into an encoding of a complete HTTP message.

All this concatenation is why we introduced the concept of a string *builder*. In the previous sections we demonstrated using the `Builder` type from the `text` package. Here, we will be using the `Builder` type from the `bytestring` package. The idea is the same, but instead of ending up with a `Text` value at the end, we will produce a `ByteString` suitable for transmission over a socket.

We will be using the `ascii` and `bytestring` packages, as well as all of the datatypes defined in chapter 6. Make sure you still have the `QuasiQuotes` language extension enabled for writing ASCII strings.


```
import qualified Data.ByteString.Builder as BSB
```

These will be the type signatures for encoding requests and responses:

```
encodeRequest :: Request -> BSB.Builder
encodeResponse :: Response -> BSB.Builder
```

To write these functions, we have to go back and re-read the same parts of RFC 7230 that we used when we defined the datatypes. But this time we have to pay more attention to some of the finer details that we skipped over on the first reading. To define the datatypes, all we cared about was what information a message conveys. Now we need to closely observe:

- ▶ the required ordering of the information in the message; and
- ▶ the spacing and punctuation that delineates the components.

The implementation of these functions follows from the beginning of RFC 7230 section 3, *Message Format*:

```
HTTP-message  = start-line
                *( header-field CRLF )
                CRLF
                [ message-body ]
```

This is the grammar for a message (which is either a request or a response). The notation used to describe the grammar is called “Augmented Backus-Naur Form” (ABNF). You don’t need to fully understand this notation; we’ll explain the relevant parts as we go.

HTTP-message is the part of the grammar being defined here. `start-line`, `header-field`, and `message-body` are references to other parts of the grammar; these are defined elsewhere, and we will look at them when

we define their corresponding encoding functions later.

Line terminator CRLF, as we mentioned in chapter 5, stands for a sequence of two characters: “carriage return” and “line feed”. We defined a `crlf` constant already in chapter 5; here we’ll just add a `Builder` version of it.

```
encodeLineEnd :: BSB.Builder
encodeLineEnd = A.lift crlf
```

The next two things are bits of punctuation that serve as modifiers:

Repetition An asterisk `*` before something means “repetition”. In Haskell we will represent repetitions with a list. The list is allowed to be empty. Here, the repeated pattern consists of two things: a header field, followed by a line terminator. There could be more than one header field, but it’s the whole pattern that repeats, so each one would have to be followed by a line terminator.

Optional Square brackets `[...]` means “optional”. In Haskell, we will represent optional values using `Maybe`. Here, the square brackets indicate that a message may, or may not, have a body.

There are four parts to the HTTP-message grammar:

1. `start-line`
2. `*(header-field CRLF)`
3. `CRLF`
4. `[message-body]`

And so we construct the request and response builders exactly like that, as the concatenation of those four parts:

```

encodeRequest (Request requestLine headerFields bodyMaybe) =
    encodeRequestLine requestLine                -- 1
    <> repeatedlyEncode
        (\x -> encodeHeaderField x <> encodeLineEnd) -- 2
        headerFields
    <> encodeLineEnd                                -- 3
    <> optionallyEncode encodeMessageBody bodyMaybe -- 4

```

```

encodeResponse (Response statusLine headerFields bodyMaybe) =
    encodeStatusLine statusLine                -- 1
    <> repeatedlyEncode
        (\x -> encodeHeaderField x <> encodeLineEnd) -- 2
        headerFields
    <> encodeLineEnd                                -- 3
    <> optionallyEncode encodeMessageBody bodyMaybe -- 4

```

Just like we did in chapter 6 for defining the types, again we're taking a top-down approach: We started with the functions we want (`encodeRequest` and `encodeResponse`) and defined them in terms of other pieces that we haven't written yet.

We are now left with the task of defining the following smaller components, which will have the following type signatures:

```

repeatedlyEncode :: (a -> BSB.Builder) -> [a] -> BSB.Builder
optionallyEncode :: (a -> BSB.Builder) -> Maybe a -> BSB.Builder
encodeRequestLine :: RequestLine -> BSB.Builder
encodeStatusLine :: StatusLine -> BSB.Builder
encodeHeaderField :: HeaderField -> BSB.Builder -- Exercise
encodeMessageBody :: MessageBody -> BSB.Builder -- Exercise

```

The last two will be left as exercises at the end of the chapter.

Stick to the spec There is a curious inconsistency in the way RFC 7230 has chosen to express the message grammar: the definition of `start-line` includes the CRLF at the end of the line, whereas the definition of `header-field` does not. This is the cause for the mildly cumbersome lambda expression above:

```
repeatedlyEncode (\x -> encodeHeaderField x <> crlf) headerFields
```

Since a header field will *always* be followed by a line terminator, we are tempted to eliminate this annoyance by making a decision to incorporate the `crlf` into the `encodeHeaderField` function. But rather than assume the liberty to make subtle terminology adjustments as we see fit, we believe it is more important for our code to directly mirror the specification.

7.4 Higher-order encodings

Among our remaining to-do list, there are two functions that stand out from the others:

```
repeatedlyEncode :: (a -> BSB.Builder) -> [a] -> BSB.Builder
optionallyEncode :: (a -> BSB.Builder) -> Maybe a -> BSB.Builder
```

These aren't functions that encode any particular thing; rather, they represent modifications to other encoding functions. `repetition` transforms a function that encodes a single thing into a function that encodes a list of things.

```
repeatedlyEncode :: (a -> BSB.Builder) -> [a] -> BSB.Builder
repeatedlyEncode enc xs = fold (map enc xs)
                        -- 2      1
```

1. First we map the function over the list. The result is a list of `Builders`.

2. We can then apply the `fold` function to join them all together.

`optionally` transforms an encoder of a single thing into an encoder that returns the empty string when the input is `Nothing`. It is perhaps easiest to write the two cases separately.

```
optionallyEncode :: (a -> BSB.Builder) -> Maybe a -> BSB.Builder
optionallyEncode _ Nothing = mempty      -- 1
optionallyEncode enc (Just x) = enc x    -- 2
```

1. When a value is present, we apply the encoding function to it.
2. When no value is present, we return the empty string.

We could have spared ourselves this effort using a handy function called `foldMap`, available in `Relude`.

foldMap This function is named for the two things that it does:

1. *map* – applies a function to each element of the collection;
2. *fold* – combines all of the results into a single value using (`<>`).

```
foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m
```

Specifically in our case, what this means that `foldMap`'s two steps are:

1. Encode each element of the collection as a `BSB.Builder`.
2. Concatenate all of the resulting `BSB.Builders` into one.

Using `foldMap`, we actually don't have to write any code at all. `BSB.Builder` is a `Monoid`, and `[]` is `Foldable`. So the `repeatedlyEncode` function is in fact a specialization of `foldMap`, and we may write it this way:

```
repeatedlyEncode :: (a -> BSB.Builder) -> [a] -> BSB.Builder
repeatedlyEncode = foldMap
```

Although it is perhaps more difficult to notice, our `optional` function can *also* be written this way!

```
optionallyEncode :: (a -> BSB.Builder) -> Maybe a -> BSB.Builder
optionallyEncode = foldMap
```

This is because `Maybe` is a `Foldable` too. To make sense of this, it helps to think of `Maybe` as an extremely limited sort of list – with the restriction that the list may only contain one item (`Just`) or zero items (`Nothing`).

At this point, it might be reasonable to remove the `repeatedlyEncode` and `optionallyEncode` functions altogether and simply use `foldMap` directly in the definitions of `encodeRequest` and `encodeResponse`. But we prefer to keep them, for two reasons:

- ▶ The words *repetition* and *optional* are used in RFC 7230, so using them in our code helps us more clearly mirror the specification document.
- ▶ When we use functions with more specialized types, instead of highly polymorphic functions like `foldMap`, our mistakes tend to yield more intelligible error messages.

7.5 The start line

Request line Section 3.1.1, *Request Line*, defines the request line (the start line for an HTTP request) as follows:

```
request-line = method SP request-target SP HTTP-version CRLF
```

We already have all the machinery we need to transcribe this into Haskell.

```
encodeRequestLine (RequestLine method target version) =  
    encodeMethod method <> A.lift [A.Space]  
    <> encodeRequestTarget target <> A.lift [A.Space]  
    <> encodeHttpVersion version <> encodeLineEnd
```

Rather than `A.lift [A.Space]`, we could have opted to write a quasi-quotation `[A.string| |]`, but we prefer to avoid situations where meaningful data is invisible.

The method and request target are trivial conversions, because we defined these as `ByteString`. The `bytestring` package provides us with a conversion function to turn a strict byte string value into a builder:

```
BSB.byteString :: BS.ByteString -> BSB.Builder
```

And so we can use this to implement `encodeMethod` and `encodeRequestTarget`.

```
encodeMethod :: Method -> BSB.Builder  
encodeMethod (Method x) = BSB.byteString x
```

```
encodeRequestTarget :: RequestTarget -> BSB.Builder  
encodeRequestTarget (RequestTarget x) = BSB.byteString x
```

Status line Section 3.1.2, *Status Line*, gives the grammar for the status line (the first line of an HTTP response).

```
status-line      = HTTP-version SP status-code
                  SP reason-phrase CRLF

status-code      = 3DIGIT

reason-phrase    = *( HTAB / SP / VCHAR / obs-text )
```

Again, we'll take our top-down approach and worry about encoding the separate parts of this afterwards.

```
encodeStatusLine :: StatusLine -> BSB.Builder
encodeStatusLine (StatusLine version code reason) =
    encodeHttpVersion version <> A.lift [A.Space]
    <> encodeStatusCode code <> A.lift [A.Space]
    <> encodeReasonPhrase reason <> encodeLineEnd
```

Specific repetition An ABNF grammar rule of the form `<n>element`, where `<n>` is a number, as in `3DIGIT`, signifies “exactly *n* occurrences of element”.

The status code is specified to be three digits, so to encode the status code, we encode each of its three digits.

```
encodeStatusCode :: StatusCode -> BSB.Builder
encodeStatusCode (StatusCode x y z) = A.lift [x, y, z]
```

Here `A.lift` takes the form of `[ASCII.Digit] -> BSB.Builder`. We have heretofore used `A.lift` only to build byte strings out of `[ASCII.Char]`; now we use it to lift from `[ASCII.Digit]` as well.

The definition of `reason-phrase` is a little more complicated, but the

important takeaway is that it's a list of characters. The details in the parentheses specify in detail *which* characters are allowed, but we will not go into such subtleties. We previously chose to represent the `ReasonPhrase` type as a byte string, and so all we do here is convert it to a `Builder` using the `BSB.ByteString` function.

```
encodeReasonPhrase :: ReasonPhrase -> BSB.Builder
encodeReasonPhrase (ReasonPhrase x) = BSB.ByteString x
```

HTTP version For the protocol version number, we need to scroll back up in RFC 7230 to section 2.6, *Protocol Versioning*.

```
HTTP-version = HTTP-name "/" DIGIT "." DIGIT
HTTP-name    = %x48.54.54.50 ; "HTTP", case-sensitive
```

For our convenience, the definition of `HTTP-name` is given to us in two forms.

- `%x48.54.54.50` tells us exactly what four bytes comprise the string. The `%x` at the beginning indicates that the numbers are given in hexadecimal notation.
- Then we are told that these four byte represent the ASCII string `"HTTP"`.

If you wanted to confirm that these two things are indeed equivalent, it is easy to do so in GHCi. Numbers can be written in hexadecimal notation in Haskell by prefixing each number with `0x`.

```
λ> map Char.chr [0x48, 0x54, 0x54, 0x50]
"HTTP"
```

We will continue to express ASCII constants using the quasi-quoter.

```
encodeHttpVersion :: HttpVersion -> BSB.Builder
encodeHttpVersion (HttpVersion x y) =
    [A.string|HTTP/|] <> A.lift [x]
    <> [A.string|.|] <> A.lift [y]
```



7.6 Exercises

Now it's time to write a couple encoding functions yourself for practice.

Exercise 18 – Header field encoding Read RFC 7239 section 3.2, *Header Fields*. Then implement the function for encoding a header field.

```
encodeHeaderField :: HeaderField -> BSB.Builder
```

Exercise 19 – Message body encoding The last thing we need to encode is `MessageBody`. Read RFC 7230 section 3.3, *Message Body*, and then implement this function.

```
encodeMessageBody :: MessageBody -> BSB.Builder
```

The answer to this exercise is short, but you may need to look at the `bytestring` package to find a function that we haven't introduced yet.

Exercise 20 – Encoding test Now that have all the necessary encoding functions written, let's do a quick test in GHCi to give ourselves an opportunity to spot any mistakes we may have made.

1. In exercise 16, you defined `Request` and `Response` values called `helloRequest` and `helloResponse`. Apply the appropriate encoding functions to each of these messages and view the results in the REPL.
2. Use `(==)` to compare the results you just viewed to the `helloRequestString` and `helloResponseString` values that we defined in chapter 5 to verify that they are the same.

Chapter 8

200

Responding

We will now begin to gradually make our HTTP server's responses more interesting. In this chapter and the next, we recreate a classic Web novelty: the hit counter.

8.1 A measure of success

In the 1990s, at the bottom of a web page you could often find a line of text like: "This page has been viewed 167 times." Such a web server was set up to keep track of how many times it had received a request for a particular resource, adding one to that number each time it served the page – thus demonstrating to the webmaster (and to everyone else) the website's fantastic popularity.

When we repeatedly make requests to our server on the command line, it will look like this:

```
$ curl http://localhost:8000
Hello!
This page has been viewed 1 time.
```

```
$ curl http://localhost:8000
Hello!
This page has been viewed 2 times.
```

```
$ curl http://localhost:8000
Hello!
This page has been viewed 3 times.
```

Just like when we wrote the encoding functions, we can use the `byte string Builder` type to assemble the text of these messages. (Soon we will switch to using the `text Builder` type, but we will stick to ASCII message bodies for a little while longer.) We define the response not as a constant anymore, but as a function of the hit count.

```
countHelloAscii :: Natural -> LBS.ByteString
countHelloAscii count = BSB.toLazyByteString $
  [A.string|Hello!|] <> encodeLineEnd <> case count of
    0 -> [A.string|This page has never been viewed.|]
    1 -> [A.string|This page has been viewed 1 time.|]
    _ -> [A.string|This page has been viewed |] <>
        A.showIntegralDecimal count <> [A.string| times.|]
```

A `Natural` is a non-negative integer (0, 1, 2, 3, ...). Mathematicians call these “the natural numbers”. (They sometimes argue about whether the definition ought to include zero. For the Haskell type, it does.) English grammar compels us to consider separately the cases 0 and 1 to construct

sensibly-sounding sentences.

showIntegralDecimal The `ascii` package conveniently provides us with a function called `showIntegralDecimal` which gives a number's decimal representation as an ASCII string. We make use of this to display the number of page views.

```
A.showIntegralDecimal ::  
    (Integral n, A.StringSuperset string) => n -> string
```

Integral The `Integral` class includes `Integer` and any integer subsets that can be converted to `Integer`, as we can see by its chief method:

```
toInteger :: Integral a => a -> Integer
```

This includes `Natural`. In `countHelloAscii`, the type of `showIntegralDecimal` is:

```
A.showIntegralDecimal :: Natural -> BSB.Builder
```

The `countHelloAscii` function will sit somewhere in the middle of our program. Now we must concern ourselves with two questions:

1. Where does the count come from?
2. Where does the byte string go?

We'll tackle the second question first: we want to package up this byte string as an HTTP Response.

8.2 Response-building utilities

If you felt that the task of constructing `Request` and `Response` values in exercise 16 was unsatisfyingly cumbersome, the utilities here may offer some remedy for your complaint.

Let's clump the `StatusCode` and `ReasonPhrase` constants together into a single `Status` datatype, because these two values will be used together.

```
data Status = Status StatusCode ReasonPhrase
```

For example, the status code 200 will always corresponds to the reason phrase "OK" (assuming we never care to change it up just for fun).

```
ok :: Status
ok = Status
    (StatusCode Digit2 Digit0 Digit0)
    (ReasonPhrase [A.string|OK|])
```

We'll also write a function to construct a status line that takes a `Status` as its argument. We fix the HTTP version at 1.1 because that's the only version we're concerned with in the book.

```
status :: Status -> StatusLine
status (Status code phrase) = StatusLine http_1_1 code phrase
```

```
http_1_1 :: HttpVersion
http_1_1 = HttpVersion Digit1 Digit1
```

The Haskell expression `status ok`, then, represents a status line that reads as "HTTP/1.1 200 OK" when encoded.

```
λ> BSB.toLazyByteString (encodeStatusLine (status ok))
"HTTP/1.1 200 OK\r\n"
```

We should pick out all of the constant values that we used in exercise 16 and will probably want to use again. This just gives us a shorthand notation that we can use in place of longer expressions. We’ve already started with `ok` and `http_1_1`. We can add a few more:

```
contentType = FieldName [A.string|Content-Type|]
```

```
plainAscii = FieldValue [A.string|text/plain; charset=us-ascii|]
```

```
contentLength = FieldName [A.string|Content-Length|]
```

There are many more header fields and status codes that we could define constants for, but for brevity’s sake we have presented only the ones we need for this particular response. For an idea of what a more complete library could look like, find the popular `http-types` package on Hackage and browse through its `Network.HTTP.Types` module.

Response We have named the response-constructing function `asciiOk` to convey the two important facts that distinguish it from other response-constructing functions that we may write in the future:

1. It constructs a response from a message body containing ASCII data;
2. The response status is “OK”.


```

asciiOk :: LBS.ByteString -> Response
asciiOk str = Response (status ok) [typ, len] (Just body)
  where
    typ = HeaderField contentType plainAscii
    len = HeaderField contentLength (bodyLengthValue body)
    body = MessageBody str

```

So far this is all stuff we’ve seen already, moved around and rewritten to make use of the new constants. The only truly new thing here is the part about the length of the body, and so we defer that to a separate function so we may consider it in isolation.

```

bodyLengthValue :: MessageBody -> FieldValue

```

Content length Whereas before we wrote `[A.string|6|]` to describe the length of the “Hello!” body, now we want this field value to be generated automatically as a function of the body. Producing the value for the Content-Length field involves a few steps:

```

bodyLengthValue :: MessageBody -> FieldValue
bodyLengthValue (MessageBody x) =
  FieldValue (A.showIntegralDecimal (LBS.length x))
--      3              2              1

```

1. Apply `LBS.length` to the byte string, which gives us the length as an `Int64` value.
2. Construct the integer’s decimal representation as an ASCII string. Again we may use `A.showIntegralDecimal`.
3. Wrap up the resulting `BS.ByteString` value in our `FieldValue` type.

This time, the type of `showIntegralDecimal` is:

```
A.showIntegralDecimal :: Int64 -> BS.ByteString
```

Both the argument and return types have changed from when we used this function before. The number being encoded is now `Int64` instead of `Natural`, and the resulting string is now `BS.ByteString` instead of `BSB.Builder` (both of which belong to the `A.StringSuperset` class because both types are supersets of ASCII text).

In case you're wondering why the `bytestring` library gives us the length of a lazy `ByteString` as a value of the type `Int64`, we will next take a quick side trip to discuss the integers.

8.3 Integers

If we browse the contents of the `Data.Int` module, we find five types: `Int`, `Int8`, `Int16`, `Int32`, and `Int64`. There is also the `Integer` type in `Prelude`. They all belong to the `Integral` class. Why so many varieties of integer?

Integer The one called `Integer` represents the classic mathematical definition of the integers: the whole numbers extending forever upward (1, 2, 3, ...) and forever downward (-1, -2, -3, ...) from 0. We describe this type as *unbounded* because there is no particular highest or lowest integer.

Int<n> The `Int` types with numbers in the name, on the other hand, are sort of abbreviated spinoffs of the integer idea that usually come into play when we're worried about performance. Unlimited possibilities are exciting, but often we're dealing with numbers that we know won't ever actually be very big – and by imposing some bounds, we can represent the numbers very compactly in memory. `Int8`, `Int16`, `Int32`, and `Int64` are integers represented in eight, sixteen, thirty-two, and sixty-four bits respectively.

Int The `Int` type exists for the sake of an even further optimization – one which might not make a lot of sense unless you’ve studied computer hardware. The upper and lower bounds of `Int` are *not precisely specified* – this allows the compiler to choose an `Int` size that is optimal for the particular machine the code is running on. The Haskell language specification says that `Int` must support values up to *at least* 536,870,911 ($2^{29} - 1$), but they may go higher if the hardware has special built-in support for some larger `int` size.

For situations in which you have to make a choice of which integer type to use, here are some general guidelines:

- ▶ Use `Integer` by default; this is safe from overflow because it has no bounds to exceed.
- ▶ Use `Int` when something needs to be fast and you’re confident that the number won’t ever exceed half a billion.
- ▶ Use `Int64` when something needs to be fast and be able to represent larger-than-normal numbers. Not astronomically large, but perhaps *terrestrially* large – an `Int64` can represent the surface area of Earth in square centimeters, barely.
- ▶ Use `Int8`, `Int16`, or `Int32` only when there is some sort of very specific byte manipulation going on that requires these particular sizes.

The `ByteString` library gives us the size of a strict byte string as an `Int` because strict byte strings are not expected to exceed 537 megabytes (2^{29} bytes) – that would be an awful lot of data to store in a single strict chunk. But, whereas a strict string is limited to what may fit in memory at once, a lazy string is not. A lazy byte string could reasonably be 2^{64} bytes long, so the library authors have chosen a type that can hold larger numbers.

```
BS.length  :: BS.ByteString -> Int
LBS.length :: LBS.ByteString -> Int64
```

In practice, `Int` and `Int64` might be the same size. On a typical 64-bit desktop machine, for example, the `maxBound` of each type is the same.

```
λ> maxBound :: Int
9223372036854775807

λ> import Data.Int

λ> maxBound :: Int64
9223372036854775807
```

However, with the `Int` type, this is due to the machine architecture rather than being a guarantee of the type itself. Since the type of `BS.length` is `Int`, the maximum size of a chunk that can be held in memory on a given machine may not actually be as high as `Int64` can handle.

The abundance of numeric types is why functions like `A.showIntegralDecimal` are polymorphic. All of these types of number can be converted to decimal strings, and having to select an appropriate monomorphic function depending on what type of number we have as the argument would be an unnecessary bother.

8.4 Response transmission

We now have two functions we'll be able to chain together to create responses:

1. `countHelloAscii :: Natural -> LBS.ByteString`
2. `asciiOk :: LBS.ByteString -> Response`

Once we have that response, we're going to want to send it over a `Socket`. So let's write that function.

```
sendResponse :: Socket -> Response -> IO ()
```

It's been a while since we've used a socket, so let's review. In chapter 4, we had the following import from the network package:

```
import qualified Network.Simple.TCP as Net
```

And we used send to send a chunk of bytes.

```
Net.send :: Socket -> BS.ByteString -> IO ()
```

Using that function again, we might write sendResponse like this:

```
sendResponse :: Socket -> Response -> IO ()
sendResponse s r = Net.send s $
    LBS.toStrict (BSB.toLazyByteString (encodeResponse r))
```

That works, but we can do a little better by switching to the sendLazy function. It is very similar, but it accepts a lazy byte string rather than a strict one.

```
Net.sendLazy :: Socket -> LBS.ByteString -> IO ()
```

This makes more sense for us here because byte string Builders produce lazy byte strings. The following two expressions are functionally equivalent:

- ▶ Net.send s (LBS.toStrict lbs)
- ▶ Net.sendLazy s lbs

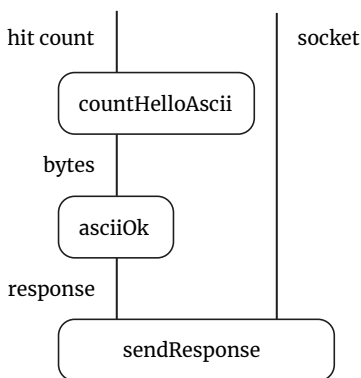
But the former requires the machine to copy the lazy list of chunks into

one big strict string before sending it – which is just unnecessary extra work for the poor mechanical beast. So instead we'll define `sendResponse` like this:

```
sendResponse :: Socket -> Response -> IO ()
sendResponse s r = Net.sendLazy s $
    BSB.toLazyByteString (encodeResponse r)
```

We now have three functions that will work together:

1. `countHelloAscii :: Natural -> LBS.ByteString`
2. `asciiOk :: LBS.ByteString -> Response`
3. `sendResponse :: Socket -> Response -> IO ()`



This picture sums up the “responding” part of the program. Here’s how we then fit it into a server that looks much like our `FirstServer`:

```
stuckCountingServer = serve @IO HostAny "8000" \(s, _) -> do
    let count = 0 -- to-do!
    sendResponse s (asciiOk (countHelloAscii count))
```

If we run `stuckCountingServer` in `GHCI`, we can switch to another terminal and try making a request to the server.

```
$ curl http://localhost:8000
Hello!
This page has never been viewed.
```

For now the count is stuck at 0, and it will remain that way for a while longer. But this is a good start.



8.5 Exercises

Exercise 21 – Read the header By default, `curl` shows us only the body of the response. Now that we have a server that is encoding our HTTP responses automatically, we don’t ever have to see the header fields anymore. It is produced by a server, consumed by a client, and never touched by human hands. This is what we wanted, but it’s a little sad, like we’ve lost touch with a friend.

Fortunately, `curl` has a lovely command-line option called “`--dump-header <filename>`” to write the message header to a file. If you give a dash “`-`” as the filename, it will display the header in the terminal instead of writing it to a file.

Run `countingServer` and use `curl` to view the complete response, header and all.

Exercise 22 – Overflow Bounded number types like `Int` and `Int64` can be tricky. For this exercise, we will be working with the `Word8` type, which represents the numbers between 0 and 255.

The following function calculates the midpoint between two numbers (rounding down if the difference between the numbers is odd).

```
mid :: Word8 -> Word8 -> Word8
mid x y = div (x + y) 2
```

For example, the number halfway between 10 and 30 is 20.

```
λ> mid 10 30
20
```

However, the `mid` function we've given you has a problem. Try it out in GHCi. What number is halfway between 210 and 230? Can you spot the mistake?

For extra credit, can you fix it?

You might be interested in the following pair of functions:

```
fromInteger :: Num a => Integer -> a
toInteger   :: Integral a => a -> Integer
```


Chapter 9

Content types



Not often do we actually want a response body that consists only of plain ASCII text – we only started off this way because it is simple, for two reasons:

1. The concept of “plain text” asks very little of us. It is a generic content type that requires no particular structure of the message body – it promises only text, not text in any particular format.
2. Much of the status line and header fields is required by the protocol to be ASCII, and so choosing ASCII for the message body as well means we were able to construct entire HTTP messages by writing text in a single encoding.

But now we ought to step into a larger world of document formats. Switching from ASCII to UTF-8 will let us use the full Unicode character set, and switching from plain text to HTML will let us serve web pages.

9.1 Some common types

Here are a handful of common values for the Content-Type field:

Content type	Structure	Text encoding
text/plain; charset=us-ascii	—	ASCII
text/plain; charset=utf-8	—	UTF-8
text/html; charset=utf-8	HTML	UTF-8
application/json	JSON	UTF-8

JSON is a late addition to this family, and its name looks a bit different from the others. By the time JSON showed up on the scene, the immense popularity of UTF-8 encoding made it the de facto encoding for all JSON message bodies. In 2017, RFC 8259 makes this official: “the vast majority of JSON-based software implementations have chosen to use the UTF-8 encoding, to the extent that it is the only encoding that achieves interoperability.” This is why we do not specify `charset=utf-8` in the `application/json` content type; the encoding of JSON is *always* UTF-8.

Let us expand upon our collection of constants:

```
plainUtf8 :: FieldValue
plainUtf8 = FieldValue [A.string|text/plain; charset=utf-8|]
```

```
htmlUtf8 :: FieldValue
htmlUtf8 = FieldValue [A.string|text/html; charset=utf-8|]
```

```
json :: FieldValue
json = FieldValue [A.string|application/json|]
```

9.2 UTF-8

The first thing we'll do is change the character set to slough off the claustrophobic limitations of ASCII.

Instead of constructing messages as byte strings, we'll be working with `Text`, the type that abstractly represents character sequences in no particular encoding – because we don't really want to be thinking about character encodings all the time! We'll do our high-level thinking in `Text`, and then only convert the text to bytes at the end when it's time to encode the response to send it.

The `text` package, like the `ascii` package, gives us some tools for writing numbers as strings. We have to look in another module to find this one:

```
import qualified Data.Text.Lazy.Builder.Int as TB
```

```
TB.decimal :: Integral a => a -> TB.Builder
```

Let's write some text!

```
countHelloText :: Natural -> LT.Text
countHelloText count = TB.toLazyText $
  TB.fromString "Hello! \9835\r\n" <>
  case count of
    0 -> TB.fromString "This page has never been viewed."
    1 -> TB.fromString "This page has been viewed 1 time."
    _ -> TB.fromString "This page has been viewed " <>
      TB.decimal count <> TB.fromString " times."
```

This definition looks like a hybrid of `sayHello` from chapter 7 and `countHelloAscii` from chapter 8. Since we're now working with Unicode – the same character set as Haskell's built-in string literals – we no

longer have reason to write these strings in quasi-quotes. We use the regular quote notation, and we convert from `String` to `TB.Builder` with the `TB.fromString` function.

We have brought back the musical notes character from chapter 3 just because it's exciting that we can.

```
import qualified Data.Text.Lazy.IO as LT
```

```
λ> LT.putStr (countHelloText 3)
Hello! ♪
This page has been viewed 3 times.
```

To turn a `Text` into `Response`, we'll need to write another function that looks very similar to `asciiOk`.

```
import qualified Data.Text.Lazy.Encoding as LT
```

```
textOk :: LT.Text -> Response
textOk str = Response (status ok) [typ, len] (Just body)
  where
    typ = HeaderField contentType plainUtf8           -- 1
    len = HeaderField contentLength (bodyLengthValue body)
    body = MessageBody (LT.encodeUtf8 str)           -- 2
```

Notice that two things that have changed:

1. We updated the content-type header field to inform the message recipient that they must now decode a UTF-8 message body.
2. Instead of a byte string, we have `Text`, so to construct a `MessageBody` we need to convert that `Text` to bytes using the character encoding that corresponds to the Content-Type header.

Only the body’s encoding has changed; the message’s header section is still ASCII. This is the role that the header section plays in the HTTP protocol: its job is to be simple and consistent, a single universal format for introducing the message body that follows it.

We may now write a version of the counting server that makes use of the more sophisticated text-based definitions.

```
stuckCountingServerText = serve @IO HostAny "8000" \(s, _) -> do
  let count = 0 -- to-do!
  sendResponse s (textOk (countHelloText count))
```

9.3 HTML

There is a great deal that may be said about HTML, CSS, and everything else that goes into the construction of websites. A sufficient treatment of this subject is outside the scope of this book, which will remain focused on the HTTP protocol. However, since the original purpose of the HTTP protocol was to serve HTML content (as reflected in the names: the “*hypertext* transfer protocol” was built to transfer documents written in the “*hypertext* markup language”), and because web pages are the most visible application of HTTP for most of us, the subject demands some cursory attention here.

blaze-html Just as we chose to rely on the `text` package to implement UTF-8 encoding instead of delving into how UTF-8 byte strings are formed, so too will we rely on the `blaze-html` package to encode HTML and ignore what the HTML looks like in its encoded form. If you happen to already be familiar with what the concrete syntax of HTML looks like, we ask that you set that knowledge aside; we will not need it here.

```
import Text.Blaze.Html (Html, toHtml)
import Text.Blaze.Html.Renderer.Utf8 (renderHtml)
import qualified Text.Blaze.Html5 as HTML
```

Our first import is the `Html` type, which is similar in concept to the `Builder` types that we used previously, but this type specifically represents HTML content.

Constructing `Html` values `toHtml` is a polymorphic function that lifts values from other types into the `Html` type.

```
toHtml :: ToMarkup a => a -> Html           -- Text.Blaze.Html
```

Members of the `ToMarkup` class include:

- Various types of strings (`String`, `Text`, etc.);
- Various types of integers (`Int`, `Int64`, etc.). Integers are converted to HTML that displays their base-10 representation.

For contrast, take a moment to scroll through the `Data.ByteString.Builder` module if you have not done so already. It contains over 60 functions that return `BSB.Builder`. Each time we want to represent some value as a `BSB.Builder` using this library, we have to look up the name of the appropriate conversion function. Here instead we have a whole menagerie of conversion functions collected together under a single name, `toHtml`. Instead of a module with a lot of functions, this library has a typeclass with a lot of instances. The polymorphic approach can be convenient because fewer named functions means fewer times we have to consult the documentation to find the name of a function we need.

The `Text.Blaze.Html5` module contains a definition corresponding to each kind of HTML tag. There are very many HTML tags, so we use a quali-

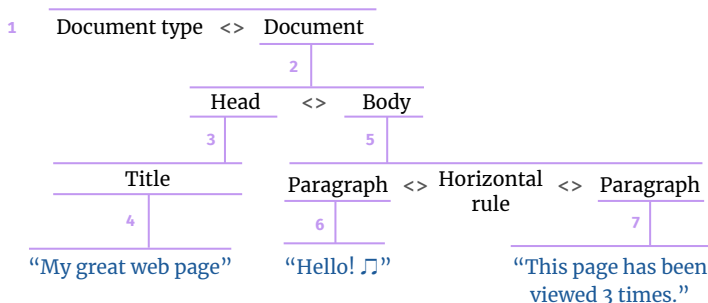
fied import for this module to avoid name conflicts. Every definition in this module is either an `Html -> Html` function or an `Html` constant.

```
html, head, body, title, p :: Html -> Html    -- Text.Blaze.Html5
docType, hr :: Html
```

We build up the tree structure of an HTML document using:

- The `(Html -> Html)` functions; and
- The `Html` type's `Monoid` instance.

Example Here is a depiction of the structure we will be assembling:



And here is the code that constructs it:

```

countHelloHtml :: Natural -> Html
countHelloHtml count = HTML.docType <> htmlDocument      -- 1
  where
    htmlDocument = HTML.html $                             -- 2
      documentMetadata <> documentBody

    documentMetadata = HTML.head titleHtml                 -- 3
    titleHtml = HTML.title (toHtml "My great web page")    -- 4

    documentBody = HTML.body $                             -- 5
      greetingHtml <> HTML.hr <> hitCounterHtml
    greetingHtml = HTML.p (toHtml "Hello! \9835")          -- 6
    hitCounterHtml = HTML.p $ case count of                -- 7
      0 -> toHtml "This page has never been viewed."
      1 -> toHtml "This page has been viewed 1 time."
      _ -> toHtml "This page has been viewed " <>
        toHtml @Natural count <> toHtml " times."

```

In most of the uses of `toHtml` above, its type is `String -> Html`. In one place, however, it is `Natural -> Html`, and in that instance we have opted to use a type application. This makes explicit that the type being converted to HTML here must be `Natural`, which is important because the sentence wouldn't make sense if the value being lifted into HTML here weren't a number. There are times when polymorphism can deprive us of the surety that static typing provides. If you feel like there is some possibility of making a type error that would not be caught by the compiler, an explicit type application may help you regain confidence.

This isn't a very fancy web page, but it will work. In exercise 23, you'll see what it looks like in a web browser.

Encoding to a byte string Once we have an `Html` value for the entire web page, we can use the `renderHtml` function to convert the final result to the UTF-8 byte string that will constitute the message body.

```
-- Text.Blaze.Html.Renderer.Utf8
renderHtml :: Html -> LBS.ByteString
```

9.4 JSON

JSON stands for ‘JavaScript Object Notation’, a name which it earned because its encoded form looks very similar to code written in the JavaScript programming language. You do not, however, need any knowledge of JavaScript to use JSON.

aeson The popular library for working with JSON is named `aeson`. We import it as follows:

```
import qualified Data.Aeson as J
import qualified Data.Aeson.Key as J.Key
import qualified Data.Aeson.KeyMap as J.KeyMap
import Data.Aeson (ToJSON (toJSON))
```

The crown jewel of the `Aeson` library is a sum type called `Value` which nicely sums up what kinds of data the JSON format can represent. Every JSON value is one of the following:

- A string, a number, `true`, `false`, or `null`;
- A list of JSON values; or
- A mapping from text keys to JSON values.

In the standard parlance of JSON, a list is called an “array” and a mapping is called an “object”. The latter is represented by the `KeyMap` type in

the `aeson` library. (Behind the scenes, it's a `Map` from the `containers` package.)

Constructing `JSON.Value` values Similar to the `ToMarkup` class that we used earlier for HTML, types that can be converted to JSON belong to a class called `ToJSON`.

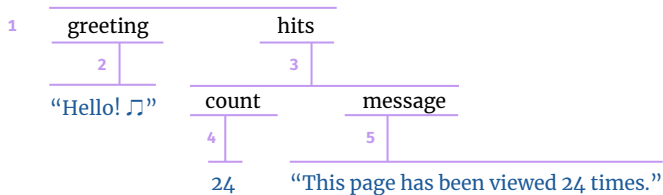
```
toJSON :: ToJSON a => a -> JSON.Value           -- Data.Aeson
```

Notable members of the `ToJSON` class include:

- Various types of strings and integers
- Various types of lists:
 - `[JSON.Value]`
 - `Seq JSON.Value`
 - `Vector JSON.Value` (also known as `JSON.Array`)
- Various types of mappings:
 - `Map Text JSON.Value`
 - `HashMap Text JSON.Value`
 - `KeyMap JSON.Value` (also known as `JSON.Object`)

We build up the tree structure of a JSON value by using the `toJSON` function to lift arrays, key maps, and other various bits like strings and integers into the `JSON.Value` type.

Example The diagram below depicts the JSON structure we will be assembling. It is an object with an entry for each of the two components of our response: the greeting message and the hit counter. The hit counter is an object which presents the page count information in two formats: once as the unadorned number, and again in the form of a sentence.



Here is the Haskell code to construct this JSON value:

```

countHelloJson :: Natural -> J.Value
countHelloJson count =
    toJSON (J.KeyMap.fromList [greetingJson, hitsJson]) -- 1
  where
    greetingJson = (J.Key.fromString "greeting",      -- 2
                    toJSON "Hello! \9835")

    hitsJson      = (J.Key.fromString "hits",          -- 3
                    toJSON (J.KeyMap.fromList
                        [numberJson, messageJson]))

    numberJson    = (J.Key.fromString "count",        -- 4
                    toJSON count)

    messageJson   = (J.Key.fromString "message",      -- 5
                    toJSON (countHelloText count))

```

The Aeson library provides several typeclass-based shortcuts to help write these sorts of functions more concisely.

- The `Key` type belongs to the `IsString` class, so we can avoid writing `Key.fromString` by using the polymorphic `fromString` method or by enabling `OverloadedStrings`.
- `toJSON (KeyMap.fromList (...))` is a common pattern, and so an equivalent function called `object` is provided.

- Expressions of the form `("...", toJSON (...))` appear four times in our code above; we can use the Aeson's `(.=)` function instead.

When we desire terseness (an impulse that strengthens as our JSON objects grow in size), we employ these shortcuts to write something like this:

```
import Data.Aeson ((.=))
```

```
countHelloJson count = J.object [
    fromString "greeting" .=
        fromString @T.Text "Hello! \9835",
    fromString "hits" .= J.object [
        fromString "count" .= count,
        fromString "message" .= countHelloText count ] ]
```

Encoding to a byte string The typical reason to construct a JSON value is to convert it to a byte string and send it across a network – so, as we might expect, the library provides us with an encoding function.

```
J.encode :: ToJSON a => a -> LBS.ByteString
```

The polymorphism may be surprising. Fortunately, `JSON.Value` is a member of the `ToJSON` class, because a JSON value can be converted to a JSON value. If that statement sounds a little silly to you, then the instance definition probably also look like some sort of joke:

```
instance ToJSON Value where           -- Data.Aeson.Types.ToJSON
    toJSON a = a
```

Any time there is a “to-something” or “from-something” typeclass, it will have an instance like this to express the triviality that, whatever the *something* type is, we can convert it to or from itself by doing nothing. This

is the same observation we made in chapter 1 when we noted that `IO` belongs to the `MonadIO` class.

So `encode` can specialize to:

```
J.encode :: JSON.Value -> LBS.ByteString
```

This observation tells us that `JSON.encode` suffices as our means to construct a `MessageBody` in the JSON version of our `Response`-constructing function.

```
jsonOk :: J.Value -> Response
jsonOk str = Response (status ok) [typ, len] (Just body)
  where
    typ = HeaderField contentType json
    len = HeaderField contentLength (bodyLengthValue body)
    body = MessageBody (J.encode str)
```



9.5 Exercises

Exercise 23 – HTML in the browser Write the HTML equivalents of `textOk` and `stuckCountingServer`.

```
htmlOk :: Html -> Response
```

```
stuckCountingServerHtml :: IO ()
```

Then run `stuckCountingServerHtml`, and open `http://localhost:8000` in a web browser.

What happens if there is a mistake in the Content-Type header field?
(Try it and see!)

Exercise 24 – Type ambiguity Try removing the type annotation (`:: T.Text`) that we included in the second definition of `countHelloJson`. Read the error messages carefully – Can you explain what has gone wrong?

Exercise 25 – Encoding with class We have now seen two libraries that use a typeclass-based approach for their type conversions, and two libraries that do not.

Package	Type	Class	Function
blaze-html	Html	ToMarkup	toHtml
aeson	JSON.Value	ToJSON	toJSON
bytestring	BSB.Builder	–	–
text	TB.Builder	–	–

What if `bytestring` had a class similar to `ToMarkup` and `ToJSON`? Nobody's here to stop us, so let's try writing one.

```
class Encode a where
    encode :: a -> BSB.Builder
```

Write `Encode` instances for the following types:

1. `Request`
2. `Response`
3. `Integer`
4. `Int64`
5. `T.Text`
6. `LT.Text`

- 7. `BS.ByteString`
- 8. `LBS.ByteString`
- 9. `BSB.Builder`

What advantages does this class offer? Does it present any problems?

Chapter 10

Change



So far we still have a hit counter that is stuck uselessly at 1. In this chapter, the number will go up.

Returning from the previous chapter’s diversions into HTML and JSON, we will pick up from `stuckCountingServerText`, which sends its response as plain text.

```
stuckCountingServerText = serve @IO HostAny "8000" \(s, _) -> do
  let count = 0 -- to-do!
  sendResponse s (textOk (countHelloText count))
```

It’s time to address that “to-do”. The new version will have two alterations (written below as holes to be filled in soon):


```

countingServer :: IO ()
countingServer = do
    hitCounter <- ___ -- 1
    serve @IO HostAny "8000" \(s, _) -> do
        count <- ___ -- 2
        sendResponse s (textOk (countHelloText count))

```

1. At the beginning of the program, we create the hit counter.
2. Upon each request, we increment the hit counter and get the new count.

Notice that there are two different variables in play here, `hitCounter` and `count`. The `hitCounter` variable represents the mutable entity holding a number that increases over time. The `count` variable, on the other hand, stands for some particular number, a snapshot of the hit counter's value at a moment in time.

In our notes about the `serve` function at the end of chapter 5, we mentioned that it handles requests concurrently by starting a separate thread for each connection. We have not needed to care about this fact so far. But now it means that multiple threads may be incrementing the hit counter at the same time, which might have interesting implications.

10.1 STM

STM requires no additional imports because everything we need from the `stm` package is already provided by `Relude`. If we were not using `Relude`, we would instead import from the `Control.Concurrent.STM` module. The two central types of the `stm` package are `TVar` and `STM`.

TVar A `TVar` holds a value that can be changed.

Like a file or socket handle, a TVar is something that we can create, read from, and write to.

```
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Sockets connect machines; files connect processes; TVars connect threads. Because our HTTP server handles requests concurrently, we are going to use a TVar as our hit counter so that all of the threads can share the information it contains.

STM The STM monad is the context in which we manipulate TVars.

Reading from a file and reading from a socket are both IO actions; reading from a TVar is an STM action. One of these things is not like the others:

```
IO.hGetChunk :: Handle          -> IO Text
S.recv       :: Socket -> Int -> IO ByteString
readTVar     :: TVar a        -> STM a
```

atomically An STM action can be converted into an IO action using the `atomically` function. In the `stm` library, its type is:

```
atomically :: STM a -> IO a
```

In Relude, the `atomically` function has a more general type:

```
atomically :: MonadIO m => STM a -> m a
```

It is otherwise the same. Some authors prefer type signatures with `MonadIO`, and others do not.

With this, we have introduced enough to get into a basic demonstration. Then we'll get around to why the STM context exists and why the function is named 'atomically'.

Using a TVar The preliminary example will be rather unexciting. Say we create a TVar String named `x` that initially contains the value "Constantinople".

```
λ> x <- atomically (newTVar "Constantinople")
```

When we read from `x`, we'll see the "Constantinople".

```
λ> atomically (readTVar x)
"Constantinople"
```

Suppose we change the value to "Istanbul".

```
λ> atomically (writeTVar x "Istanbul")
```

Now when we read from it again, this time we see the new value.

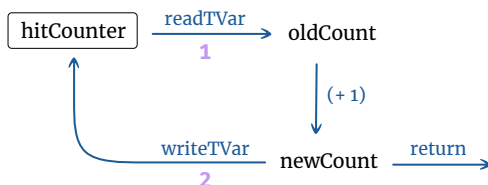
```
λ> atomically (readTVar x)
"Istanbul"
```

The previous state, "Constantinople", is now gone. When we look at a TVar, we see only what was written to it most recently.

10.2 Increment

Each time our server handles a request, it will execute a sequence of two actions involving the TVar:

1. Read its value to find out how many people have been here so far.
2. Write a new number (the old number plus one), to record the fact that we've now had another visitor.



STM is in the `Monad` class, like `IO`, so we can use a `do` block to tell a story that involves a sequence of multiple STM actions.

```

increment :: TVar Natural -> STM Natural
increment hitCounter = do
    oldCount <- readTVar hitCounter -- 1
    let newCount = oldCount + 1
    writeTVar hitCounter newCount -- 2
    return newCount
  
```

10.3 Atomically

So what's the difference between STM and IO?

1. TVar interactions are the only sort of I/O that an STM action can include.
2. STM has `retry` and `orElse` operations, which facilitate some marvellous techniques that this book does not cover.
3. Whereas the steps of an IO action running in different threads can be interleaved, the steps of an STM action happen all at once as if it were a single action.

We will focus on the third aspect. When an action happens all at once, we call it “atomic”. We sometimes also refer to a collection of actions happening all at once without interruption as a “transaction”, which is what the letter T stands for in both ‘TVar’ and ‘STM’.

Atomicity matters because if multiple threads are allowed to fiddle with the state at the same time, an action that makes sense in a single-threaded program often doesn’t mean the same thing anymore. Imagine we *didn’t* write `increment` as an atomic transaction, and suppose two threads were to increment the counter nearly simultaneously:

- Let’s say the value of `hitCounter` is 9.
- In thread 1:
 - `atomically (readTVar hitCounter)`
 - `oldCount` is 9.
- In thread 2:
 - `atomically (readTVar hitCounter)`
 - `oldCount` is 9.
- In thread 1:
 - `atomically (writeTVar hitCounter (9 + 1))`
- In thread 2:
 - `atomically (writeTVar hitCounter (9 + 1))`

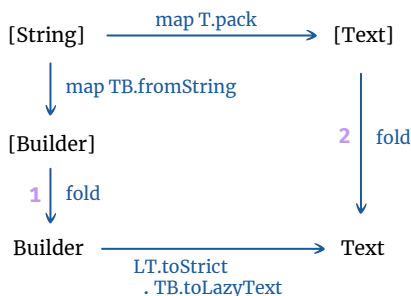
When events transpire like this, we describe the actions as “interleaved”; their steps are shuffled together like two halves of a deck of cards. This interleaving gives the wrong result! At the end, the hit counter only reads ten, when it ought to have gone to eleven.

The distinction between atomic and interleaved operations is subtle and requires careful thought. It is reminiscent of another subtlety we’ve looked at: the difference between `Text` and `TextBuilder`.

Text / text builder Recall from chapter 7 that although the `Text` and `TextBuilder` types both represent text, they have important operational dif-

ferences.

The diagram below presents two ways to concatenate a list of strings, starting from a `[String]` input and ending up with a `Text` output. We get the same result via either route.



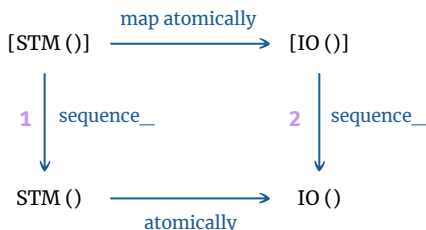
The `fold` function in `Data.Foldable` concatenates a list of strings into a single string. It works on any `Monoid`, so we can either:

1. concatenate a list of `TB.Builders`; or
2. concatenate a list of `Texts`.

Although these two paths are semantically the same, there is reason to prefer one over the other: the path via `fold 2` is potentially much slower. So we would usually rather concatenate our texts as `Builders`, even if the `Text` type is ultimately what we want to end up with.

IO / STM `STM` is like a *builder* type for `IO` actions; even when `IO` is what we want to end up with, it may be important to combine steps in an `STM` context first.

Suppose we have a list of `STM` actions, and we want to produce an `IO` action that executes them all. We can draw another diagram that is similar to the previous one:



The `sequence_` function that combines a list of actions into a single action that does them all, one after another. It works on any `Monad`, so we have two choices:

1. combine a list of STMs together into one STM; or
2. combine a list of IOs together into one IO.

Which path is better? Unlike the text-concatenation situation, in which going through `TB.Builder` is generally preferable, the question in this case has no single answer. It depends upon what the STM actions are, and upon what guarantees we want out of the resulting IO.

- If the list of STM actions could potentially be rendered nonsensical by interference from other threads, then we should combine them all into a single transaction via path 1.
- If the STM actions in the list are unrelated to one another, then we might safely take path 2, in which each runs in a separate transaction.

10.4 The counting server

At the start of the chapter, we left some holes in the server. Let's get those filled in.

```
countingServer = do
  hitCounter <- atomically (newTVar @Natural 0)           -- 1
  serve @IO HostAny "8000" \(s, _) -> do
    count <- atomically (increment hitCounter)           -- 2
    sendResponse s (textOk (countHelloText count))
```

1. `newTVar 0` creates the counter, initialized at zero. The type application is not mandatory, but a reader of the code may find it helpful to see what type the `TVar` will contain.
2. `increment hitCounter` increases the counter and returns the information that will appear in the response.

Run the server and try it out!

```
λ> countingServer
```

```
$ curl localhost:8000
Hello! 🎵
This page has been viewed 1 time.
```

```
$ curl localhost:8000
Hello! 🎵
This page has been viewed 2 times.
```

10.5 Other STM topics

`TVar` and `STM` together constitute the foundation of `STM`. The `stm` package also includes some bonus features to demonstrate how larger structures can be built out of `TVars`. One delightful example is the *queue*.


```

data TQueue a                                -- Control.Concurrent.STM.TQueue
newTQueue :: STM (TQueue a)
readTQueue :: TQueue a -> STM a
writeTQueue :: TQueue a -> a -> STM ()

```

We see here the same basic operations that TVar has: create, read, and write. The difference is that whereas a TVar holds only a single item, a queue holds a list. Writing to a queue adds an item to the list, and reading from a queue removes an item from the list. If the queue is empty, the readTQueue operation waits patiently for something to arrive.

If you wish to delve further into fancy STM techniques, you may also be interested in the `stm-containers` package, which provides a TVar-based hash map.

```

data Map key value                            -- StmContainers.Map
new :: STM (Map key value)
insert :: (Eq key, Hashable key) => value -> key
                                             -> Map key value -> STM ()
lookup :: (Eq key, Hashable key) => key -> Map key value
                                             -> STM (Maybe value)
delete :: (Eq key, Hashable key) => key -> Map key value
                                             -> STM ()

```

But our exploration of STM ends here, because we have achieved our goal: a TVar holds the state of the hit counter. Getting more sophisticated with concurrent data structures is usually about optimizing performance. Simply keeping mutable state in a TVar is often all that is needed.



10.6 Exercises

Exercise 26 – Interleaving Create a non-atomic version of `increment` in which the `readTVar` and `writeTVar` steps do not run within a single transaction.

```
incrementNotAtomic :: TVar Integer -> IO Integer
```

Add the following import from the `async` package:

```
import qualified Control.Concurrent.Async as Async
```

Add this definition:

```
testIncrement :: (TVar Natural -> IO a) -> IO Natural
testIncrement inc = do
  x <- atomically (newTVar @Natural 0)
  Async.replicateConcurrently_ 10 (replicateM 1000 (inc x))
  atomically (readTVar x)
```

The `testIncrement` function creates a `TVar` and starts ten concurrent threads. Each thread increments the `TVar` 1,000 times.

Then try the following two experiments in `GHCi`:

```
λ> testIncrement (\x -> atomically (increment x))
```

```
λ> testIncrement incrementNotAtomic
```

The first test should print “10000”. The second test likely will not.

Exercise 27 – Times gone by Write an HTTP server that responds with the amount of time that has elapsed since the previous request. An outline is given, with some blanks, below.

```
timingServer :: IO ()
timingServer = do
    ----
    serve @IO HostAny "8000" \(s, _) -> do
        ---
        sendResponse s $ textOk $ LT.pack $
            show (___ :: Maybe Time.NominalDiffTime)
```

The `NominalDiffTime` type from the `time` library was involved indirectly in the timing measurement we did in chapter 7, although we never mentioned it by name. It represents a duration between two moments in time, produced by subtracting one time from another.

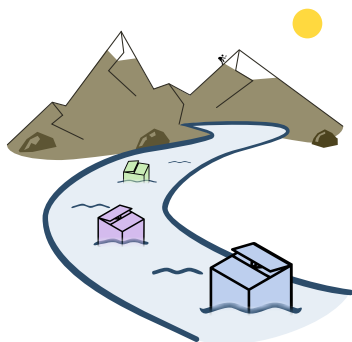
```
Time.diffUTCTime :: UTCTime -> UTCTime -> NominalDiffTime
```

You will need `sendResponse` from chapter 8 and `textOk` from chapter 9.

The solution requires a `TVar`; the first thing to decide is what type of state it should store.

Chapter 11

Streaming



One typical sort of HTTP server – perhaps more common in the earlier days of the Web – is a *file server*. The host machine has a collection of files. When a client requests one, the server reads the file and sends its contents over a socket to satisfy the happy recipient.

Traditionally, the request would specify *which* file the client wants. But we aren’t reading requests yet. We’ll get to that soon! For now, the goal in this chapter is just to build a simple server that distributes a single file. Create a file called “stream.txt” in your data directory (run `getDataDir` in GHCi if you’ve forgotten where this is), and enter into it your favorite Heraclitus quotation:

No one ever steps in the same stream twice, for it is not the same stream and they are not the same person.

The most convenient way to get this server working would be to use a library function that reads the contents of the file.

```
LBS.hGetContents :: Handle -> IO LBS.ByteString
```

Once we have the file content, we would then use that to construct an `HTTPMessageBody` and a `Response`.

```
hContentsResponse :: Handle -> IO Response
hContentsResponse h = do
    fileContent <- liftIO (LBS.hGetContents h)
    let body = Just (MessageBody fileContent)
    return (Response (status ok) [] body)
```

And we can use that to make a file server. Like `countingServer` from chapter 10, this server action has a setup step; before the serving begins, we use `getDataDir` to find out where the files are stored.

```
fileStrict = do
    dir <- getDataDir
    serve @IO HostAny "8000" \(s, _) -> runResourceT @IO do
        (_, h) <-
            binaryFileResource (dir </> "stream.txt") ReadMode
        r <- liftIO (hContentsResponse h)
        liftIO (sendResponse s r)
```

That works, but what if the file were very large? Even though this uses the lazy `ByteString` type (and even despite `LBS.hGetContents` employing “lazy I/O”, a topic which we have not discussed), this server reads the file’s entire content into memory at once. This has nothing to do with what library function we chose; it was forced by the requirement that we know the file’s size upfront to generate the `Content-Length` header. We will solve this problem using a different kind of HTTP message that does not require `Content-Length`.

11.1 Chunked hello

The Content-Length header field is one of two ways to let the client know when it's time to stop waiting for more of the response to read. This chapter is about the second way: segmenting the message body in multiple smaller parts ("chunks").

Read the docs Please read the following parts of RFC 7230:

- Section 3.3.1, *Transfer-Encoding*
- Section 3.3.2, *Content-Length*
- Section 4.1, *Chunked Transfer Coding*

Hello! We need to think again about what encoded HTTP messages look like, so let's revisit our first manually-written *hello* response from chapter 5:

```
helloResponseString =  
  line [A.string|HTTP/1.1 200 OK|] <>  
  line [A.string|Content-Type: text/plain; charset=us-ascii|] <>  
  line [A.string|Content-Length: 6|] <>  
  line [A.string||] <>  
  [A.string|Hello!|]
```

Now we will write that message again, this time in a chunked format.

```

helloResponseStringChunked =
  line [A.string|HTTP/1.1 200 OK|] <>
  line [A.string|Content-Type: text/plain; charset=us-ascii|] <>
  line [A.string|Transfer-Encoding: chunked|] <>
  line [A.string||] <>
  line [A.string|2|] <> line [A.string|He|] <>
  line [A.string|4|] <> line [A.string|llo!|] <>
  line [A.string|0|] <>
  line [A.string||]

```

helloResponseStringChunked is semantically the same message, but it is encoded differently: The body is divided into two segments.

Status line	HTTP/1.1 200 OK
Header part	Content-Type: text/plain; charset=us-ascii
	Transfer-Encoding: chunked
First chunk	2 He
Second chunk	4 llo!
Last chunk	0
Trailer part	

As always, the first line is the status line, then we have the headers, and then a blank line. What comes after the blank line is the body. When we're using the chunked transfer encoding, the body consists of a list of *chunks*.

Each chunk consists of:

1. One line containing the chunk length
2. The chunk content

A server is free to divide a message into as many chunks of whatever size

it prefers. Our demonstration divides a 6-byte message into two comically tiny chunks. Typically a chunk in an HTTP message body will be several kilobytes in size, but here we must design toy-sized examples to fit on a page.

Chunk one Our first chunk in this example was:

```
line [A.string|2|] <> line [A.string|He|]
```

The content of this chunk is “He”, which is two ASCII characters. Each character is represented by one byte, so this chunk length is 2.

Chunk two The second chunk contains four characters.

```
line [A.string|4|] <> line [A.string|llo!|]
```

The last chunk The protocol then demands that we conclude the body part of a chunked message with a line that just says “0”.

```
line [A.string|0|]
```

Trailers We’re almost done, but there is one more little bit at the end. After the chunks, an HTTP message is allowed to include more header fields! This is in the same format as the header part of the message, but since it comes at the end, it’s called the *trailer* part.

Like the headers, the trailers are concluded by a blank line to signify the end of the list. Since our example message has no trailers, the trailer part consists solely of this blank line.

```
line [A.string||]
```


Trailers are not a commonly used feature, and we will not make use of them. However, we need to be aware that this part of the grammar exists, only because it obliges us to add that extra final CRLF at the end of each chunked message body.

11.2 Chunk types

We ought to augment our collection of HTTP types from chapter 6 with a Chunk type. A chunk consists of two things: the size of the data, and the data itself.

```
data Chunk = Chunk ChunkSize ChunkData
```

```
data ChunkData = ChunkData BS.ByteString
```

```
data ChunkSize = ChunkSize Natural
```

If you're thinking that it seems superfluous to define yet more types – and such simple ones! – think of it this way: HTTP has already given us the benefit of a coherent set of terminology, and it would be a shame to waste this gift only to make our own code less lucid by rejecting the specification's words.

Chunk construction Next we could benefit from some additional functions that help construct chunks. Typically, we will want to start by obtaining a `ChunkData` value, and then use that to determine the size.

```
dataChunk :: ChunkData -> Chunk  
dataChunk chunkData = Chunk (chunkDataSize chunkData) chunkData
```

```

chunkDataSize :: ChunkData -> ChunkSize
chunkDataSize (ChunkData bs) =
    case toIntegralSized @Int @Natural (BS.length bs) of
        Just n -> ChunkSize n
        Nothing -> error (T.pack "BS.length is always Natural")

```

One does not wish to spend one's time thinking about numbers, but this fate will continually befall a programmer, it seems. It is sort of funny that `length` functions so often use `Int` as their result type, since lengths are never negative! The `Word` type would be more appropriate, since an `Int` can be negative and a `Word` cannot. Rather than either `Int` or `Word`, though, we prefer to describe lengths and counts using `Natural` so that we may remove the possibility of arithmetic overflow from our minds as much as possible.

toIntegralSized Above we have used the `toIntegralSized` function, which is generally the best way to convert between two integer-like types. If the input number cannot be represented by the output type, it returns `Nothing` rather than produce an erroneous result.

```

toIntegralSized ::                                     -- from Data.Bits
    (Integral a, Integral b, Bits a, Bits b) =>
    a -> Maybe b

```

In the context above, this type is:

```

toIntegralSized :: Int -> Maybe Natural

```

Since `Natural` has no upper bound and `BS.length` never produces as negative values, this conversion should always return `Just`. Since the `Nothing` case is unreachable, we use `error` to fill it in vacuously. For its `Text` argument, we write a brief explanation of why we believe this situa-

tion cannot occur. If we are wrong, this text will print as our program comes to a crashing halt. One should avoid making such judgments when possible, and otherwise strive to be certain about them. To bolster our certainty, we use `toIntegralSized` with type applications because these particular type parameters are critically relevant to our unproven assertion that the conversion must succeed.

11.3 Encoding a chunk

Remember in chapter 7 when we wrote a whole bunch of encoding functions that return `BSB.Builder`? Here we go again. In RFC 7230 section 4.1, *Chunked Transfer Coding*, we find the relevant part of the grammar. It is reproduced in abridged form below (for brevity's sake, we will ignore the parts about *chunk extensions*).

```
chunk           = chunk-size CRLF
                  chunk-data CRLF
chunk-size      = 1*HEXDIG
last-chunk      = 1*("0") CRLF
chunk-data      = 1*OCTET
```

We will write one Haskell definition for each definition given to us in the ABNF grammar. This helps keep the code tidy and conformant to the specification.

```
encodeChunk     :: Chunk      -> BSB.Builder
encodeChunkSize :: ChunkSize -> BSB.Builder
encodeLastChunk ::             BSB.Builder
encodeChunkData :: ChunkData -> BSB.Builder
```

The encoders for `chunk` and `last-chunk` can mirror the grammar quite closely.

```
encodeChunk (Chunk chunkSize chunkData) =  
    encodeChunkSize chunkSize <> encodeLineEnd <>  
    encodeChunkData chunkData <> encodeLineEnd
```

```
encodeLastChunk :: BSB.Builder  
encodeLastChunk = encodeChunkSize (ChunkSize 0) <> encodeLineEnd
```

Encoding the chunk-data is only a matter of extracting the ByteString from the ChunkData constructor and lifting it into the Builder type.

```
encodeChunkData :: ChunkData -> BSB.Builder  
encodeChunkData (ChunkData x) = BSB.byteString x
```

Hexadecimal chunk size Unlike the Content-Length header, which is encoded in decimal (base 10, using the digits 0 through 9), chunk lengths are encoded in hexadecimal (base 16, using the digits 0 through 9 and the letters 'a' through 'f'). So instead of showIntegralDecimal, we use showIntegralHexadecimal.

```
A.showIntegralDecimal ::  
    (Integral n, StringSuperset string) => n -> string  
A.showIntegralHexadecimal ::  
    (Integral n, StringSuperset string) => Case -> n -> string
```

The hexadecimal function has an additional Case parameter because the letters that represent digits 10 through 15 may be given in either upper or lower case. HTTP does not specify which; either choice is okay.

If you need a reminder of what hexadecimal numbers look like, add this

definition to your code file, and print it in GHCi:

```
exampleHexNumbers = map hex [0 .. 32]
  where
    hex :: Natural -> Text
    hex = A.showIntegralHexadecimal A.LowerCase
```

The `showIntegralHexadecimal` function is all it takes to encode a chunk size.

```
encodeChunkSize (ChunkSize x) =
  A.showIntegralHexadecimal A.LowerCase x
```

11.4 Transfer-Encoding

We'll need a few more constants to help construct the appropriate header field for a message with a chunked body.

```
transferEncoding = FieldName [A.string|Transfer-Encoding|]
```

```
chunked = FieldValue [A.string|chunked|]
```

```
transferEncodingChunked = HeaderField transferEncoding chunked
```

Remember that not all HTTP messages have a body. If a message *does* have a body, then it must have either (not both):

- Content-Length: ...; or
- Transfer-Encoding: chunked.

If a message *does not* have a body, then it *must not* have either of these headers.

11.5 Serving the file

Chapter 2 was about why it's often good to work with text not one character at a time, and not all at once, but rather in reasonably-sized *chunks* of text somewhere in between. We had this method of consuming streams:

```
repeatUntil :: IO a    -> (a -> Bool) -> (a -> IO x) -> IO ()
              -- Get one -> Is the last? -> Some action
```

Then in chapter 3, we wrote a file-copying procedure that worked like this:

```
copyGreetingFile = runResourceT @IO do
  (_, h1) <- binaryFileResource "... ReadMode -- 1
  (_, h2) <- binaryFileResource "... WriteMode -- 2
  liftIO $ repeatUntil (BS.hGetSome h1 1024) BS.null $ -- 3
    \chunk -> BS.hPutStr h2 chunk
```

1. Open the source (h1 :: Handle)
2. Open the destination (h2 :: Handle)
3. Repeatedly read from the source and write to the destination

The file server will have a similar structure, because copying a file is still essentially what it does. The difference is that the destination is now a `Socket`, not another file `Handle`.

```

fileStreaming :: IO ()
fileStreaming = do
  dir <- getDataDir
  serve @IO HostAny "8000" \(s, _) -> runResourceT @IO do -- 1
    (_, h) <- -- 2
      binaryFileResource (dir </> "stream.txt") ReadMode
    liftIO do
      -
      repeatUntil (BS.hGetSome h 1024) BS.null -- 3
      \chunk -> _
      -

```

1. `serve` gives us the destination (`s :: Socket`)
2. Open the source (`h :: Handle`)
3. Repeatedly read from the source and write to the destination

The three holes in the code above stand in for the additional effort we have yet to exert to wrap the data up as an HTTP message.

Whereas previous incarnations of our HTTP server concatenated all the Builders together to then send a single lazy `ByteString` over the socket, this time we apply `Net.sendLazy` to the various parts of the message separately; this permits this implementation's "streaming" nature. Let us introduce a `sendBSB` function because we'll now be performing this particular action a lot:

```

sendBSB :: Socket -> BSB.Builder -> IO ()
sendBSB s bs = Net.sendLazy s (BSB.toLazyByteString bs)

```

Each of the holes is a place where we need to write some pieces of the response using `sendBSB`. The file server is completed as follows:

```

fileStreaming = do
  dir <- getDataDir
  serve @IO HostAny "8000" \(s, _) -> runResourceT @IO do
    (_, h) <-
      binaryFileResource (dir </> "stream.txt") ReadMode
    liftIO do
      sendBSB s (encodeStatusLine (status ok))
      sendBSB s (encodeHeaders [transferEncodingChunked])
      repeatUntil (BS.hGetSome h 1024) BS.null \c ->
        sendBSB s (encodeChunk (dataChunk (ChunkData c)))
      sendBSB s encodeLastChunk
      sendBSB s (encodeTrailers [])

```

```

encodeHeaders :: [HeaderField] -> BSB.Builder
encodeHeaders xs =
  repeatedlyEncode
    (\x -> encodeHeaderField x <> encodeLineEnd) xs
  <> encodeLineEnd

```

```

encodeTrailers :: [HeaderField] -> BSB.Builder
encodeTrailers = encodeHeaders

```

The headers and trailers have exactly the same format: they are both a sequence of lines, followed by a blank line. Since we are not including any trailers and have no intention to ever do so, the server's final action ...

```

sendBSB s (encodeTrailers [])

```

... could have been written without the `encodeTrailers` function as:


```
sendBSB s crlf
```

Instead, we chose to introduce the `encodeTrailers` function to help make evident *why* that final line break appears. Otherwise it's easy to get into a bad situation where you're counting line breaks trying to remember what each one means to work out whether you have the right number. This hard-earned wisdom inspired exercise 28.



11.6 Exercises

Exercise 28 – Tripping at the finish line In this exercise, we would like you to intentionally introduce some mistakes and see how to detect them.

- What happens when you include too few line breaks at the end of the message? (test with `curl`)
- What happens when you include too many? (test with `curl --verbose`)

The first version of the code we ever published contained such an error. Thanks to Alain O'Dea for pointing it out!

Exercise 29 – Infinite response One curious consequence of streaming responses is that the party doesn't *ever* have to stop. Write a server that just keeps sending HTTP chunks indefinitely.

Here are two fun suggestions for how to write an infinite loop:

```
forever do
  -- ...
```

```
for_ [1 ..] \n ->  
  -- ...
```

If you don't just want to see a blur of text fly past as quickly as your machine can spit it out, it might be a good idea to introduce some pauses. There is a function for pausing in the base package:

```
import Control.Concurrent (threadDelay)
```

```
threadDelay :: Int -> IO ()
```

The `Int` parameter specifies how long you want to pause, given as a number of microseconds. In case you don't remember science class, *micro-* is the metric prefix for 10^{-6} .

- 1,000,000 microseconds = 1 second
- 1,000 microseconds = 1 millisecond

Chapter 12

ListT IO



The code we wrote in chapter 11 sacrificed some intangible quality – some might say *modularity* or *composability*; we’ll say that the code does not feel quite in harmony with our thoughts. HTTP is all about requests and responses, but that sentiment is missing from the `fileStreaming` function because although we wrote a program that responds, at no point did we actually define *a response*. When we focus on operational aspects, like making sure a server deals with large data in a streaming way, it is easy to lose sight of semantic aspects: what the code means to us.

12.1 The new response type

In this chapter, we will define a new type of response called `StreamingResponse`. The task of responding will then be expressed as a composition of two functions. First we will construct a response;

```
hStreamingResponse :: Handle -> MaxChunkSize -> StreamingResponse
```

```
data MaxChunkSize = MaxChunkSize Int
```

then we will send it.

```
sendStreamingResponse :: Socket -> StreamingResponse -> IO ()
```

With the beloved concept of *responses* returned to the forefront, the file server can then be redefined as follows:

```
fileStreaming2 = do
  dir <- getDataDir
  let fp = dir </> "stream.txt"
  serve @IO HostAny "8000" \(s, _) -> runResourceT @IO do
    (_, h) <-
      binaryFileResource (dir </> "stream.txt") ReadMode
    let r = hStreamingResponse h (MaxChunkSize 1024)
    liftIO (sendStreamingResponse s r)
```

Compare this to the definition of `fileStrict` from the beginning of the previous chapter; we hope it feels like an improvement.

The trick to preserving the streaming behavior will be to choose appropriate field types when we define the `StreamingResponse` datatype. What we need is going to come from a package called `list-transformer`.

New imports for this chapter:

```
import List.Transformer (ListT, runListT)
import qualified List.Transformer as ListT
```

Recall the original response type from chapter 6:

```
data Response = Response
    StatusLine [HeaderField] (Maybe MessageBody)
```

```
data MessageBody = MessageBody LBS.ByteString
```

We defined a message body as a `ByteString`, and the message body appears as the third field of the `Response` constructor – in a `Maybe` context, because not every response has a body.

We’re going to define a new type of response that uses a new type to represent the body, using the `Chunk` type from chapter 11 and `ListT` from the `List.Transformer` module.

```
data StreamingResponse = StreamingResponse
    StatusLine [HeaderField] (Maybe ChunkedBody)
```

```
data ChunkedBody = ChunkedBody (ListT IO Chunk)
```

Notice that since we have chosen to define a new `StreamingResponse` type instead of attempting to “upgrade” the existing `Response` type, we now have separate types that both represent an HTTP response. This is okay! We should not be bound by an assumption that each concept deserves exactly one corresponding type. Often one abstract idea like “an HTTP message” has several possible representations that are each useful in different situations.

12.2 What is ListT

Repetition, repetition, repetition.

The type constructor `ListT IO` is a sort of amalgamation of `[]` and `IO`. We sometimes refer to a `ListT IO` of chunks as a *stream* of chunks. There are two good ways to approximate what it means:

1. **It is a list that can also do I/O.** A `ListT IO Chunk` is like a `[Chunk]` that can perform side effects (like reading from a file handle) while the chunk list is being enumerated. An `IO` effect, as part of larger a `ListT IO` operation, will likely be performed repeatedly, not just once.
2. **It is an I/O action that contains multitudes.** A `ListT IO Chunk` is like an `IO Chunk`, but instead of returning a final result, the action repeatedly emits chunks as it runs.

Both are useful paradigms; switch between them to suit whichever frame of mind fits best at the moment.

The `T` in `ListT` stands for “transformer”, because `ListT` is a *monad transformer* – if `m` is a monad, then `ListT m` is a monad as well. `ListT IO` represents the concept of `IO` that has been *transformed* or modified by adding a certain essence of listiness.

Overview of the functions we will be using to build, compose, and modify streams:

return A stream that emits one item.

```
return :: a -> ListT IO a                                -- from the Monad class
```

select Lifts an ordinary list into a stream with no I/O.

```
ListT.select :: Foldable list => list a -> ListT IO a
```

In this chapter, we will use it in two specializations:

```
ListT.select :: [a]      -> ListT IO a
ListT.select :: Maybe a -> ListT IO a
```

Use of `Maybe` as a `Foldable` requires us to think of `Nothing` as an empty list and `Just` as a list with one item.

liftIO Used to incorporate I/O actions into the stream.

```
liftIO :: IO a -> ListT IO a           -- from the MonadIO class
```

We will use `liftIO` twice in this chapter, because we need to run a stream that reads from a file handle and writes to a socket.

takeWhile Used to stop a stream early. The streaming will continue as long as the items satisfy the predicate.

```
ListT.takeWhile :: (a -> Bool) -> ListT IO a -> ListT IO a
```

runListT To use a stream we've made, we use `runListT` to turn it into an `IO ()` action.

```
runListT :: ListT IO a -> IO ()
```

Notice that the type parameter `a` appears in the function's parameter, but not in its result. The `a` disappears because the stream's values are discarded. Before we apply `runListT`, anything we want to do using streamed values needs to already have been incorporated into the stream using `>>=` and `liftIO`.

We don't convert a stream into an ordinary list that contains all the items.

```
_ :: ListT IO a -> IO [a] -- not like this
```

Such a thing is possible, but accumulating all the items would generally defeat the purpose of streaming, which is to work with a series of values without every having them all together in one place.

Much of the essence of `ListT` lies within its instances of `Alternative` and `Monad`. These instances behave the same for `ListT` as they do for ordinary lists. Since many of us don't often think about the list `Alternative` or the list `Monad`, will take a moment to discuss them next.

Alternative The two central methods of the `Alternative` class are:

```
empty :: Alternative f => f a
(<|>) :: Alternative f => f a -> f a -> f a
```

`empty` is as plainly-named a thing as a body could hope for. The empty list is the empty list, and the empty `ListT` is an empty stream; it emits no list items and performs no I/O.

The `Alternative` operator `(<|>)` concatenates two lists.

```
(<|>) @[] :: [a] -> [a] -> [a]
```

Likewise, `(<|>)` concatenates two streams.

```
(<|>) @ (ListT IO) :: ListT IO a -> ListT IO a -> ListT IO a
```

The resulting stream yields all items from the first followed by all items from the second, and it performs the I/O actions of both.

(Don't attempt to use `<>` or `fold` to concatenate streams! The `ListT Monoid` is something else entirely.)

The pipe operator `<|>` is meant to be read as “or”, even though we would more commonly say that a concatenation holds the contents of the first list *and* those of the second list. The methods of the `Alternative` class are simple enough, but it requires some effort to accept the metaphor. In what sense is concatenation disjunctive? We will return to this question after we discuss the `Monad`.

Monad The monadic `return` function lets us construct a singleton stream that emits one item.

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

For lists and streams, the meaning of the `(>=)` operator (and consequently the meaning of a `do` block) is novel and requires some contemplation.

Consider, for example, how we might generate a list of meals that one might order from a small restaurant menu.

```
haskellCafe :: [(String, String)]
haskellCafe = do
  entree <- ["chicken", "fish"]
  side   <- ["soup", "salad"]
  return (entree, side)
```

The result is a list of all possible combinations of one entree and one side:

```
[ ("chicken", "soup"),  
  ("chicken", "salad"),  
  ("fish", "soup"),  
  ("fish", "salad") ]
```

We read `inputStream >>= something` as “for each item from the input stream, do something with it”. So when you see an expression like:

```
outputStream = do  
  x <- inputStream  
  something x
```

Think “for each item from the input stream ...” and then consider what kind of behavior that `something` might be – performing some I/O, or emitting some values that will appear in the output stream. For example, we can use the `ListT` monad instead of `traverse_` to perform an action for each item in a list.

```
printMenu :: [(String, String)] -> IO ()  
printMenu meals = runListT @IO do  
  (entree, side) <- ListT.select @[] meals  
  liftIO (IO.putStrLn (entree <|> " and " <|> side))
```

```
λ> printMenu haskellCafe  
chicken and soup  
chicken and salad  
fish and soup  
fish and salad
```

Returning to the question of why we should comprehend concatenation as an “or” operation, consider the above expression `["soup", "salad"]`,

which can also be written as `["soup"] <|> ["salad"]`, and which signifies that one may order either the soup *or* the salad. For a `ListT IO` stream, picture the items on a conveyor belt rolling past one at a time as you perform some repetitive task. Each item in the stream represents one possibility for which item might be in hand at any given moment.

Infinity To induce unbounded repetition, select from an unbounded list.

```
metronome = runListT @IO do
  ListT.select @[] (repeat ()) -- For each of an infinite list
  liftIO (IO.putStrLn "tick")   -- tick the metronome
  liftIO (threadDelay 1000000)  -- and pause for a second.
```

The list monad is very different from the `IO` monad, so pay careful attention to the type context before you start reading a `do` block.

- If the context is `IO` or `ResourceT IO`, you're looking at a linear procedure where each `<-` binds the result of a single I/O action.
- If the context is `[]` or `ListT IO`, then `do` represents a stream transformation where each `<-` signifies a loop, and the subsequent lines of the `do` block will be repeated for each item obtained from the input stream.

12.3 Constructing a response

Now let's start putting some of this `ListT` stuff to serious use. Recall that we wanted to write a function that will build a `StreamingResponse` where the body comes from reading a file handle.

```
hStreamingResponse :: Handle -> Int -> StreamingResponse
```

The `Int` parameter specifies the maximum size of each chunk read from the handle. Since there is no particular reason that `hStreamingResponse` itself should choose one number or another, it leaves this as a parameter to push the decision off onto someone else.

It begins in a straightforward way, similar to other responses we have made in the past.

```
hStreamingResponse h maxChunkSize =
    StreamingResponse statusLine headers (Just body)
  where
    statusLine = status ok -- 1
    headers = [transferEncodingChunked] -- 2
    body = chunkedBody (hChunks h maxChunkSize) -- 3
    --      ^^^^^^^^^^^^^      ^^^^^^^
```

1. Nothing interesting in the status line, same as always.
2. As in the previous chapter, the only header field we need is `Transfer-Encoding: chunked`. A content type header might be a nice addition to this list, but we have not included that here because it depends on what type of file you have chosen to serve.
3. The fun part will be turning the handle contents into an HTTP message body, so we write the steps of that task in separate functions so that we may savor them fully.

`hChunks` converts a `Handle` into an `ListT IO` action that reads from the handle and emits the chunks of its contents as list items.

```
hChunks :: Handle -> MaxChunkSize -> ListT IO BS.ByteString
hChunks h maxChunkSize =
    ListT.takeWhile (not . BS.null)
        (hInfiniteChunks h maxChunkSize)
```

```

hInfiniteChunks ::
  Handle -> MaxChunkSize -> ListT IO BS.ByteString
hInfiniteChunks h (MaxChunkSize mcs) = do
  ListT.select @[] (repeat ())
  liftIO (BS.hGetSome h mcs)

```

`hInfiniteChunks` is an never-ending stream of byte strings, written in the same style as `metronome` from earlier. We turn it into a finite stream by applying `ListT.takeWhile`, specifying that we are only interested in non-null chunks. Once the end of the file is reached, `hGetSome` will return an empty chunk, the `hChunks` action will halt, just as we have always done.

A stream of byte strings can be turned into a `ChunkedBody` using the functions we wrote earlier for turning byte strings into `ChunkData` values.

```

chunkedBody :: ListT IO BS.ByteString -> ChunkedBody
chunkedBody xs = ChunkedBody do
  bs <- xs
  return (dataChunk (ChunkData bs))

```

If you're really into the whole brevity thing, you might notice that what we're doing to this `ListT` is merely an `fmap` operation. It might be written like this, if one were so inclined:

```

chunkedBody = ChunkedBody . fmap (dataChunk . ChunkData)

```

12.4 Encoding a response

In chapter 7, we wrote a function to encode a `Response` as a byte string builder.

```
encodeResponse :: Response -> BSB.Builder
```

Now we must write its streaming analog. Since the message body is now given as a stream, so too will the output of the encoding be a stream – a stream of bytes destined for a socket.

```
encodeStreamingResponse ::  
    StreamingResponse -> ListT IO BS.ByteString
```

The variety of strings Once again, maybe you thought we were done introducing new types of byte strings, but we’ve concocted yet another one. We’re now up to five! Each with its proper place in the world.

Type	Notes	Introduced
[Word8]	List of bytes	Chapter 3
BS.ByteString	Strict packed list of bytes	Chapter 3
LBS.ByteString	List of strings	Chapter 6
BSB.Builder	... with fast concatenation	Chapter 7
ListT IO BS.ByteString	Stream of strings	Right now

Of the string types we’ve used previously, `Builder` is the most akin to this new *byte stream* concept we’re working with now. They are both sorts of intermediate types that we use to concatenate shorter strings to form longer strings without necessarily accumulating the entire longer string into memory at once. The difference is that `Builder` didn’t have any notion of interleaving I/O, like reading from a file, throughout the production of the shorter pieces.

We could go through a passionate fit of rewriting and modify all of the

encoding functions from chapter 7 to produce streams instead of builders. But we may as well leave them as they are, and instead introduce a means of turning the existing builders into streams.

```
selectChunk :: BSB.Builder -> ListT IO BS.ByteString
```

First we turn the `Builder` into an ordinary list of its strict constituent chunks using two functions from the `bytestring` library. Then we use `select` to lift the list into `ListT IO`, producing a stream that emits each chunk.

```
selectChunk b =  
    ListT.select @[] (LBS.toChunks (BSB.toLazyByteString b))
```

Now, using `selectChunk`, we can reuse the previously-written encoding functions within a streaming context.

encodeStreamingResponse The definition for the encoding function itself ought to serve as a nice outline of the main parts of the HTTP message grammar and our strategy for producing them.

```

encodeStreamingResponse
  (StreamingResponse statusLine headers bodyMaybe) =
  asum
    [ selectChunk (encodeStatusLine statusLine)
    , selectChunk (encodeHeaders headers)
    , do
      ChunkedBody chunks <- ListT.select @Maybe bodyMaybe
      asum
        [ do
            chunk <- chunks
            selectChunk (encodeChunk chunk)
          , selectChunk encodeLastChunk
          , selectChunk (encodeTrailers [])
        ]
    ]

```

Much of the result is built up by concatenating various smaller streams. We do not write `<|>` here, but it is there, just under the surface. Infix operators can be awkward when you have to wrap them and their arguments across multiple lines, especially when some of the arguments are `do` blocks. We have therefore opted to do the stream concatenation using `asum` instead of `<|>`.

asum In the same way that the `fold` function joins together all the elements of a list using `<>`, the `asum` function joins together all the elements of a list using `<|>`.

```

fold :: (Foldable list, Monoid m)      => list m    -> m
asum :: (Foldable list, Alternative f) => list (f a) -> f a

```

```
asum [a, b, c] = a <|> b <|> c
```


The beginning of the message, consisting of the status line and header field list, does not involve any streaming I/O. We encode these parts as before, and then throw the pieces into the stream using `selectChunk`. Then, if a body is present, we encode it in a streaming way, finishing it off by encoding the last chunk and the empty trailer part.

When we were encoding with `Builder`, we used `foldMap` to express optional and repeated parts. This time we're using the `ListT` monad, and each optional or repetitive aspect of the message corresponds to a `do` block in `encodeStreamingResponse`. Recall that the lines of a `do` block mean “for each” in a `ListT IO` context. For each HTTP chunk, we encode the chunk and toss the encoded bytes into the river to meet whatever destiny awaits them downstream.

12.5 Sending a response

We're almost done revising the file server! All that remains is to write the function that directs the bytes of an encoded `StreamingResponse` into a `Socket` to send them along to the HTTP client.

```
sendStreamingResponse :: Socket -> StreamingResponse -> IO ()
```

Encoding the response gives us a stream of byte strings, and we want to write each one to the socket. “Each” is the key word here: it means we're using the monad again.

```
sendStreamingResponse s r = runListT @IO do
  bs <- encodeStreamingResponse r
  Net.send s bs
```

Notice that we have now made use of the `MonadIO` instance twice, because there are two kind of I/O involved in the stream: the input, and the

output.

1. In `hInfiniteChunks`, we applied `liftIO` to lift the file-reading `IO` action into the `ListT IO` context.
2. `Net.send` has a `MonadIO` constraint. Behind the scenes, it is using `liftIO` to lift the socket-writing action into the `ListT IO` context.

This final stream, containing both the input and output actions, is now a complete thought, describing the entire flow from start to finish. We can now take a step back from the river metaphor and view “send the file to the client” as a single action. This shift in perspective is put forth in code by the application of `runListT`, which produces a result of the type `IO ()`.

12.6 ListT in other libraries

There are several things in the world of Haskell that share the name ‘ListT’, and we wish to acknowledge a few in attempt to avoid confusing them.

list-t The `list-t` package is quite similar to `list-transformer`, and it offers an equivalent `ListT` type.

pipes The `pipes` library defines a type called a `Proxy` which is a more general notion of a stream. A proxy can be a `Producer`, a `Consumer`, or a `Pipe` that inserts, removes, and transforms elements along their way from a producer to a consumer. This library offers a `ListT` type, which is a type wrapper around `Producer`. It is functionally equivalent to the `ListT` type from the `list-transformer` library that we have presented in this chapter.

If you switch between `list-transformer`, `list-t`, and `pipes`, pay attention to the meanings of the typeclass instances, because they do not all behave in the same way. In particular, there is more than one way to define a monoid for `ListT`, and among these packages there is disagreement on the `Monoid` instance.

transformers The `transformers` and `mtl` libraries once had a type called `ListT`. It looked like this:

```
-- Control.Monad.Trans.List
newtype ListT m a = ListT { runListT :: m [a] }
```

This type, regarded as a mistake, has been removed from these libraries because its `Monad` instance was not actually valid. This definition is unhelpful for us anyway because it does not provide streaming behavior.



12.7 Exercises

Exercise 30 – List and stream operations In the definitions of `haskell-Cafe` and `printMenu`, identify each place where we:

1. Lifted `[]` into `ListT`
2. Lifted `IO` into `ListT IO`
3. Turned the `ListT IO` into an `IO` action to run
4. Used `[]` as a monad
5. Used `[]` as an alternative
6. Used `ListT` as a monad (and what is the type of the implicit `(>=)` operation?)

Exercise 31 – Repeat until In chapter 2, we parameterized some of the specifics of `printCapitalizedText` to produce more general-purpose `repeatUntilIO` function.

```
repeatUntilIO ::
  IO chunk          -- ^ Producer of chunks
-> (chunk -> Bool) -- ^ Does chunk indicate end of file?
-> (chunk -> IO x) -- ^ What to do with each chunk
-> IO ()
```

A similar gem can be extracted from this chapter. We will call it `listUntilIO`, and it will have the following type:

```
listUntilIO ::
  IO chunk          -- ^ Producer of chunks
-> (chunk -> Bool) -- ^ Does chunk indicate end of file?
-> ListT IO chunk
```

Define the `listUntilIO` function. Then rewrite `hChunks` using `listUntilIO`.

Exercise 32 – File copying In chapter 3, we gave the following file-copying program:

```
copyGreetingFile = runResourceT @IO do
  dir <- liftIO getDataDir
  (_, h1) <-
    binaryFileResource (dir </> "greeting.txt") ReadMode
  (_, h2) <-
    binaryFileResource (dir </> "greeting2.txt") WriteMode
  liftIO $ repeatUntil (BS.hGetSome h1 1024) BS.null \chunk ->
    BS.hPutStr h2 chunk
```

Let us rewrite this program as follows:

```

copyGreetingStream = runResourceT @IO do
  dir <- liftIO getDataDir
  (_, h1) <-
    binaryFileResource (dir </> "greeting.txt") ReadMode
  (_, h2) <-
    binaryFileResource (dir </> "greeting2.txt") WriteMode
  liftIO (hCopy h1 h2)

```

```

hCopy source destination = runListT @IO do
  chunk <- _
  -

```

Fill in the blanks to implement hCopy.

Exercise 33 – Copying to multiple destinations Suppose we want to write a function that makes multiple copies of a file.

```

fileCopyMany :: FilePath -> [FilePath] -> IO ()
fileCopyMany source destinations = runResourceT @IO do
  (_, hSource) <- binaryFileResource source ReadMode
  hDestinations <- forM destinations \fp -> do
    (_, h) <- binaryFileResource fp WriteMode
    return h
  liftIO (hCopyMany hSource hDestinations)

```

```

hCopyMany :: Handle -> [Handle] -> IO ()
hCopyMany source destinations = _

```

Fill in the blank to implement hCopyMany.

Chapter 13

Parsing



The HTTP servers we have written so far can construct messages but cannot read them! Good communication isn't just about expression; it requires listening and understanding. If we want our server to do something useful, we're going to have to figure out how to also decode HTTP-encoded messages so that we can figure out what the client is asking for.

In chapter 11, we wrote `fileStreaming`, which responds to HTTP requests by sending content read from some fixed file path. In this chapter, instead of using a fixed file path, we will construct a `Map` that relates resource names to corresponding file paths. When a client requests a resource from us, we will look up the request target in the map to find the path of the file to send in response.

Map The `Map` type itself is re-exported by `ReLude`, but we'll need to import the functions for working with maps:

```
import qualified Data.Map.Strict as Map
```

There are a lot of great functions in the `Data.Map.Strict` module, and you should read through it some time. But the two most basic `Map` functions are:

```
Map.fromList :: Ord k => [(k, a)] -> Map k a
```

```
Map.lookup :: Ord k => k -> Map k a -> Maybe a
```

`fromList` constructs a `Map` from a list of key-value pairs, and `lookup` tells you what value, if any, is mapped at a particular key.

For our purposes here, these types are:

```
Map.fromList :: [(T.Text, FilePath)] -> Map T.Text FilePath
```

```
Map.lookup :: T.Text -> Map T.Text FilePath -> Maybe FilePath
--      Request target -> Resource collection -> What file to send
```

Add a second text file alongside `stream.txt`. Call this one `read.txt`, and for its content we'll borrow from Frederick Douglass this time:

Once you learn to read, you will be forever free.

This is what our response will be when a client requests the `/read` resource. But when a client requests the `/stream` resource, we will give them Heraclitus.

```
data ResourceMap = ResourceMap (Map T.Text FilePath)
```

```
resourceMap :: FilePath -> ResourceMap
resourceMap dir = ResourceMap $ Map.fromList
  [ (T.pack "/stream", dir </> "stream.txt"),
    (T.pack "/read", dir </> "read.txt") ]
```

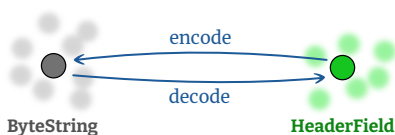
But before we look up the request target in the Map, the server will have to read the incoming HTTP message to figure out what the request target is.

13.1 Encoding vs decoding

Each encoding function we wrote in chapter 7 maps values of some type, like `HeaderField`, to strings that can then be shot through a socket.

```
encodeHeaderField :: HeaderField -> BSB.Builder
```

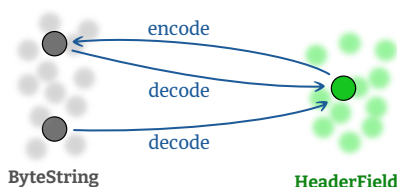
Decoding involves the same correspondence between values and strings, but in the other direction.



Given a `HeaderField` that has been encoded to a string, the decoding should convert it back to recover the original `HeaderField`. Working backwards isn't so simple as it may seem; reading a message string is more complicated than constructing one.

Synonyms The set of valid possibilities for what our decoder might *receive* is larger than the set of strings that our encoding functions *produce*; some—

times there's more than one valid way to encode a value.



For example, an HTTP header field may be written with any number of spaces after the colon.

- Content-Length: 7
- Content-Length:7
- Content-Length: 7

These are three equally valid encodings of the same `HeaderField` value; they all mean the same thing.

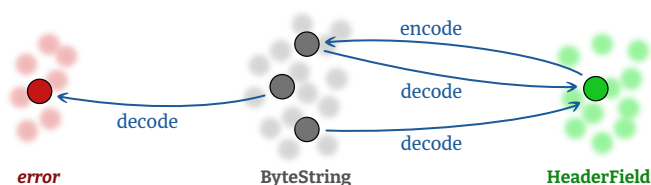
Our `encodeHeaderField` function produces a single space after the colon. But when we decode a message that we have received from someone else, we will have to accept whatever number of spaces we find, because the protocol allows others the freedom to make their own choice. Compared to encoding, therefore, decoding requires a more complete understanding of the message format and a more careful reading of the specification document.

Errors If decoding were the *inverse* of encoding, we might define a function like this:

```
decodeHeaderField :: LBS.ByteString -> HeaderField
```

This type signature is appealing, but we can't make it work; not every string can be meaningfully interpreted. If a string doesn't contain a colon,

for example, then it isn't an HTTP header field. So decoding can *fail*.



The codomain for a decoding function therefore must include some additional possibilities. For example, when we talked about UTF-8 in chapter 3, we saw that the decoding function produces an `Either` result, where invalid inputs correspond to `Left` outputs.

```
T.encodeUtf8 :: Text -> ByteString
T.decodeUtf8' :: ByteString -> Either UnicodeException Text
```

A more realistic type for `decodeHeaderField` might take this form:

```
decodeHeaderField :: LBS.ByteString -> Either _ HeaderField
```

What type should fill in that hole? It's a tricky question. When decoding fails, we usually want to know *why* so that we may figure out what is wrong (a mistake in either the input string or in the decoding process), and herein lies the devil. Decoding a valid string is not as difficult as constructing an error that sensibly explains *why* an invalid string is invalid.

But there is yet one more concern.

Parsing As always, we want to work with manageably small units of code; this means defining the decoders for messages in terms of decoders of their constituent parts. In chapter 7 we wrote encoding functions for all the small parts of an HTTP message, and we combined them to build up to functions

that encode entire requests and responses. To form combinations of encoding functions, all we had to do was write a new function that encodes the smaller parts and then concatenates the resulting strings.

- For byte strings and text, we concatenated with `<>`, `fold`, and `foldMap`.
- For `ListT` streams, we concatenated with `<|>` and `asum`, thinking of each item as an alternative possibility for what chunk of data a looped action might be working with during any one iteration. We also used `>>=`, via `do` notation, to build up increasingly long streams as nested loops: “For each item in the first stream, ...”

However, going backwards introduces a new requirement: How do we figure out where to partition the string to feed the right parts of it into the appropriate smaller decoders? This sort of structural analysis is called *parsing*. If parsers are to be composable – that is, if multiple parsers may be combined to construct one larger parser – then each parser must be responsible not only for decoding some bytes into a more usable form, but also for determining how much of the input to decode and how much to leave behind. Most parsers stop before the end, leaving behind some untouched remainder.

- Our chief tool for combining parsers is `>>=`, and most of the parsers will be written as `do` blocks. Parsing is a procedure, a sequence of actions taken as we move across the bytes of input from beginning to end. It is in this way analogous to IO.
- Parsing is also analogous to `ListT`, as we again picture ourselves in front of an assembly line, a stream of bytes passing by. We will at times consider what alternative (`<|>`) situations we may be in any one moment.

13.2 Attoparsec

There are a lot of parsing libraries. We've chosen Attoparsec because it can handle *incremental input*. This means we can use Attoparsec to interpret a stream of chunks from the socket with I/O interspersed, beginning as soon as the first chunk comes in, instead of first having to first receive the entire HTTP message. Not all parsing libraries can do this; some are built exclusively for reading strings in memory.

Add the following imports:

```
import qualified Data.Attoparsec.ByteString as P
import Data.Attoparsec.ByteString (Parser, (<?>))
```

Attoparsec's interface comes in two flavors. These modules are quite similar, differing chiefly in whether the parsers consume `ByteString` or `Text` input:

- ▶ `Data.Attoparsec.ByteString` deals with `Word8` and `ByteString`.
- ▶ `Data.Attoparsec.Text` deals with `Char` and `Text`.

Since HTTP is a text-based protocol, it is tempting to consider parsing with Attoparsec's `Text` interface. But only the leading portion of an HTTP message – the start line and header fields – are constrained to be text. All that can be said in general of message bodies is that they should be of whatever format is specified by the `Content-Type` header, which is not necessarily text.

Therefore we cannot honestly claim an HTTP message as a whole to be text, and we must deal with it as a byte string. If the leading text of a message were represented in a variable-width encoding, this would be cause for some consternation. Fortunately, ASCII uses a fixed-width encoding – each character is one byte – and is thus quite amenable to treatment with tools designed for `ByteStrings`.

parseOnly Before we get to incremental parsing or HTTP, let's see some small Parser examples. We'll use `parseOnly`, the non-incremental (not suitable for streaming) way to run a parser, for REPL demonstrations.

```
P.parseOnly :: Parser a -> ByteString -> Either String a
```

The Parser `a` specifies how to read a byte string to get a value of type `a`. If all goes well, `parseOnly` returns `Right` with the interpreted value. Otherwise, it returns `Left` and an error message. This function is called “parse only” as in “parse only this one string” because it provides no mechanism for subsequently feeding additional chunks into the parser; `parseOnly` only gets one shot, and once it's done, it's done.

The HTTP parsers we define in this chapter will need three library functions: `takeWhile`, `string`, and `anyWord8`.

```
P.takeWhile :: (Word8 -> Bool) -> Parser ByteString
```

```
P.string :: ByteString -> Parser ByteString
```

```
P.anyWord8 :: Parser Word8
```

We will use the following string as an example input to introduce these functions:

```
countString :: BS.ByteString  
countString = [A.string|one-two-three-four|]
```

takeWhile This is analogous to several common functions of the same name in other modules.

```
takeWhile :: (a -> Bool) -> [a] -> [a]           -- from Data.List
```

```
ListT.takeWhile :: (a -> Bool) -> ListT IO a -> ListT IO a
```

```
BS.takeWhile :: (Word8 -> Bool) -> ByteString -> ByteString
```

```
P.takeWhile :: (Word8 -> Bool) -> Parser ByteString
```

For example, `(BS.takeWhile A.isLetter)` returns the longest prefix of the string consisting of bytes that represent ASCII letters.

```
λ> BS.takeWhile A.isLetter countString  
"one"
```

The parser `(P.takeWhile A.isLetter)` does the same thing.

```
λ> P.parseOnly (P.takeWhile A.isLetter) countString  
Right "one"
```

The result is enclosed in the `Right` constructor, indicating that parsing succeeded. With `P.takeWhile`, this is always the case; this is an example of a parser that cannot fail. Even if there are no characters to take, `P.takeWhile` will “take” zero matching characters and be satisfied with having done nothing at all.

```
λ> P.parseOnly (P.takeWhile A.isDigit) countString  
Right ""
```

Our next function introduces a failure possibility.

P.string We use this function when we know exactly what to expect to find next if the input is valid. There are a number of particular strings that we expect to find within an HTTP message, such as “HTTP” and the [carriage return, line feed] pairs that demarcate lines.

```
P.string :: ByteString -> Parser ByteString
```

The first argument is the string that we expect to find. If the input starts with that, then the parsing succeeds.

```
λ> P.parseOnly (P.string [A.string|one-two|]) countString
Right "one-two"
```

If the string does not begin with what we expect, we get a `Left` result indicating that the input is rejected.

```
λ> P.parseOnly (P.string [A.string|three|]) countString
Left "string"
```

The rather minimalist error message here, “string”, indicates that it was the `P.string` parser that was running when the failure occurred. This is not especially helpful information! We will get around to fixing that later in the chapter.

anyWord8 This parser grabs the next byte of input.

```
P.anyWord8 :: Parser Word8
```

```
λ> P.parseOnly P.anyWord8 countString
Right 111
```

(Byte 111 represents ASCII character 'o'.)

This parser doesn't care what byte it finds, but it does fail if there is no input remaining at all.

```
λ> P.parseOnly P.anyWord8 BS.empty
Left "not enough input"
```

The `anyWord8` parser is analogous to the `BS.uncons` function, which breaks a `ByteString` into its first byte and the remainder.

```
BS.uncons :: ByteString -> Maybe (Word8, ByteString)
```

```
λ> BS.uncons countString
Just (111, "ne-two-three-four")
```

```
λ> BS.uncons BS.empty
Nothing
```

In addition to the functions mentioned above, we will also take advantage of the fact that `Parser` belongs to the typeclasses `Applicative`, `Monad`, and `MonadFail`.

Monadic sequencing What we have shown so far doesn't by itself help you accomplish anything that you couldn't have already done with functions from the `bytestring` library. What's special about parsers, though, is that we can combine them sequentially in a `do` block. They run one after another, each picking up where the last parser left off.


```
takeTwoWords :: Parser (BS.ByteString, BS.ByteString)
takeTwoWords = do
  a <- P.takeWhile A.isLetter           -- 1
  _ <- P.string (A.lift [A.HyphenMinus]) -- 2
  b <- P.takeWhile A.isLetter           -- 3
  return (a, b)
```

1. Takes “one”, leaving “-two-three-four” as the remainder.
2. Takes the hyphen character, leaving “two-three-four”.
3. Takes “two”.

```
λ> P.parseOnly takeTwoWords countString
Right ("one", "two")
```

The remainder of the input, “-three-four”, is disregarded.

Now let’s get back to our project.

13.3 Request line

The first thing we encounter in an HTTP request is the *request line*. Recall the format of a request line as given in RFC 7230 section 3.1.1, *Request Line*:

```
request-line = method SP request-target SP HTTP-version CRLF
```

SP and CRLF are defined early on in section 1.2, *Syntax Notation*. SP is an abbreviation for ‘space’. The request line is not a place where any amount of space is acceptable; when it says SP, it means there must be exactly one space character.

Since these are exact strings that we expect to find in the message, we

can use `P.string` for them.

```
spaceParser :: Parser BS.ByteString
spaceParser = P.string (A.lift [A.Space])
```

```
lineEndParser :: Parser BS.ByteString
lineEndParser = P.string (A.lift crlf)
```

There are six components of the request line; each will have a corresponding line within our parser's `do` block.

```
requestLineParser :: Parser RequestLine
requestLineParser = do
  method <- methodParser           -- 1
  _ <- spaceParser                 -- 2
  requestTarget <- requestTargetParser -- 3
  _ <- spaceParser                 -- 4
  httpVersion <- httpVersionParser -- 5
  _ <- lineEndParser               -- 6
  return (RequestLine method requestTarget httpVersion)
```

Remaining to be defined are the smaller pieces:

```
methodParser :: Parser Method
```

```
requestTargetParser :: Parser RequestTarget
```

```
httpVersionParser :: Parser HttpVersion
```

Method The same section of the specification also describes the request method. It is defined as:

```
method = token
```

So we'll parse a "token", and then wrap it up in the `Method` constructor.

```
methodParser :: Parser Method
methodParser = do
    x <- tokenParser
    return (Method x)
```

We are left to wonder: What is a token?

```
tokenParser :: Parser BS.ByteString
```

'Token' is a generic-sounding word that appears often in grammars. In the context of RFC 7230, a token is a sequence of letters, numbers, and some but not all of the characters like `$&*!` that are used to censor profanities in cartoon speech bubbles.

The definition of token is somewhat oddly tucked into section 3.2.6, *Field Value Components*, despite the fact that it is also referenced in various other places throughout the document.

```
token = 1*tchar
```

```
tchar = "!" / "#" / "$" / "%" / "&" /
        "'" / "*" / "+" / "-" / "." /
        "^" / "_" / "`" / "|" / "~" /
        DIGIT / ALPHA
```

Let's start with `tchar`. This defines exactly which bytes are allowed in a token. We'll want a predicate that can check whether a byte of input is of this sort.

```
isTChar :: Word8 -> Bool
```

The definition of `tchar` begins with an enumeration of 15 specific characters. First let's get those written down.

```
tcharSymbols :: [Word8]
tcharSymbols = A.lift
  [ A.ExclamationMark, A.NumberSign, A.DollarSign,
    A.PercentSign, A.Ampersand, A.Apostrophe, A.Asterisk,
    A.PlusSign, A.HyphenMinus, A.FullStop, A.Caret,
    A.Underscore, A.GraveAccent, A.VerticalLine, A.Tilde ]
```

With that tedium out of the way, then return to `isTChar`. A byte satisfies the definition of a `tchar` if it can be found on the list of acceptable symbols, if it is a digit, or if it is a letter.

```
isTChar c = elem c tcharSymbols || A.isDigit c || A.isLetter c
```

We have discussed previously that an asterisk `*` signified repetition. So we might try:

```
tokenParser = P.takeWhile isTChar
```

This is almost right, but look closer: The grammar actually says `1*`, which means a repetition of at least one. So if the parser comes back to us with an empty result, we need to reject it. This is where `fail` comes in.

```
tokenParser = do
  token <- P.takeWhile isTchar
  case BS.null token of
    True  -> fail "tchar expected"
    False -> return token
```

Alternatively, using when:

```
tokenParser = do
  token <- P.takeWhile isTchar
  when (BS.null token) (fail "tchar expected")
  return token
```

For Parser, the fail function specializes to:

```
fail :: String -> Parser a
```

fail returns a trivial parser that consumes no input and never succeeds.

Target The second part of the request line is the *target*. RFC 7230 section 5.3, *Request Target* enumerates a handful of forms that the request target can take. It also references another document, *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax*, which contains the full specification for the syntax of a URI. This is way more detail than we want to go into for this book.

Allowing our HTTP implementation to be somewhat incorrect, we will consider a request target to simply be any non-empty string consisting of visible characters (letters, numbers, and symbols, but not spaces). The specification refers to this character class as ‘VCHAR’.

```
requestTargetParser :: Parser RequestTarget
requestTargetParser = do
  x <- P.takeWhile A.isVisible
  when (BS.null x) (fail "vchar expected")
  return (RequestTarget x)
```

Again, we have included a check to ensure that at least one character was read. The empty string is not a valid request target.

HTTP version The third interesting part of the request line is the HTTP version, e.g. 'HTTP/1.1'. This is discussed in section 2.6, *Protocol Versioning*. This string is little, yet still our parser shall subdivide it, enumerating its pieces one by one, so that we may:

1. Skip past the fixed parts that should always be there;
2. Pick out the variable parts that hold useful information.

```
httpVersionParser :: Parser HttpVersion
httpVersionParser = do
  _ <- P.string [A.string|HTTP|]      -- 1
  _ <- P.string (A.lift [A.Slash])    -- 1
  x <- digitParser                    -- 2
  _ <- P.string (A.lift [A.FullStop]) -- 1
  y <- digitParser                    -- 2
  return (HttpVersion x y)
```

This leaves one more component to write a parser for:

```
digitParser :: Parser A.Digit
```

Digit There is an `isDigit` predicate in the `ASCII` module.

```
A.isDigit :: Word8 -> Maybe Digit
```

The Digit parser can start by creating one character, and then failing if it is not a digit.

```
digitParser :: Parser A.Digit
digitParser = do
  x <- P.anyWord8
  unless (A.isDigit x) (fail "0-9 expected")
  return _
```

This then leave us with the question of what to return, because x is a Word8 and we want a Digit. Perhaps this conversion function would be appropriate:

```
A.word8ToDigitUnsafe :: Word8 -> Digit
```

```
digitParser = do
  x <- P.anyWord8
  unless (A.isDigit x) (fail "0-9 expected")
  return (A.word8ToDigitUnsafe x)
```

This function has the “unsafe” label in its name because it is partial. On any byte that does not correspond to an ASCII digit character, word8ToDigitUnsafe is undefined. This should be fine, though, because we have already checked to ensure that it is a digit. (If this approach feels wrong, don’t worry; we will return to it in exercise 35)

13.4 Explaining what's wrong

As we mentioned earlier, when parsing fails, we usually want the output to include some explanation of why. The parser that we've written so far fails to provide much explanation.

Suppose a client constructs a response in which there is no slash after the word "HTTP" in the version portion of a request line.

```
λ> P.parseOnly requestLineParser [A.string|GET /hello HTTP1.1|]
Left "string"
```

The error message "string" tells us nearly nothing. We can do much better. When parsing fails, the error information consists of two things:

- *Context* – A [String] list of context names that describes where within the grammar the problem is situated
- *Description* – A String description of the problem

When we're done making revisions here, the new result is going to be:

```
λ> P.parseOnly requestLineParser [A.string|GET /hello HTTP1.1|]
Left "Version: Failed reading: HTTP should be followed by a slash"
```

The context is ["Version"], and the description is "HTTP should be followed by a slash".

Adding context We can add contextual information by naming sub-parsers using (<?>). This operator has the following type:

```
(<?>) :: Parser a -> String -> Parser a
```


In parsing as well as in other walks of life, accumulating appropriate contextual information is the key to avoiding inscrutable error messages. If you try to make a request to an HTTP server and it responds with “HTTP should be followed by a slash”, that does not contain enough information to be a meaningful explanation. That statement only makes sense if you know that this problem was encountered while attempting to read the HTTP version from a request line.

We should, therefore, use `<?>` to annotate the interesting constituent parts of a request line: The method, the target, and the version.

```
requestLineParser = do
  method <- methodParser <?> "Method"
  _ <- spaceParser
  requestTarget <- requestTargetParser <?> "Target"
  _ <- spaceParser
  httpVersion <- httpVersionParser <?> "Version"
  _ <- lineEndParser
  return (RequestLine method requestTarget httpVersion)
```

In any failure originating from the version portion of a request line, then, the name "Version" will appear within the context stack of the resulting error message.

Changing the description We can use `<|>` and `fail` to change a parser’s error description. This is demonstrated three times below in our final revision to `requestLineParser`:

```

requestLineParser = do
  method <- methodParser <?> "Method"
  _ <- spaceParser
    <|> fail "Method should be followed by a space"
  requestTarget <- requestTargetParser <?> "Target"
  _ <- spaceParser
    <|> fail "Target should be followed by a space"
  httpVersion <- httpVersionParser <?> "Version"
  _ <- lineEndParser
    <|> fail "Version should be followed by end of line"
  return (RequestLine method requestTarget httpVersion)

```

This is the first time we are using `<|>` with parsers. The meaning of (a `<|>` b) is: first try parser a, and then if it fails, use parser b instead. In the above example, parser b always fails as well – but it fails with a better error message. We could have written something terse like “space expected”, but we prefer to write a longer message that also mentions the preceding part of the request line. If a space is missing, this extra text helps give the reader some idea of *where* a space is missing.

Whenever we define a parser that is made up of multiple smaller parts, we should give each of the parts either a name or a custom error message. We have just done this to `requestLineParser`. The other example in this chapter is `httpVersionParser`; we leave it as exercise 36.

13.5 Incremental parsing

The chapter threatens to end soon, and we must deliver a server. The request parser is not yet complete – we can only read the first line – but the request line will suffice to make good on what was promised.

The easiest mechanism for executing our `requestLineParser` on

chunks of input from a socket is to Attoparsec's `parseWith` function.

```
parseWith ::  
    IO ByteString -> Parser a -> ByteString -> IO (Result a)  
-- ----- 1 -----      -- 2 --      --- 3 ---
```

Its parameters are:

1. An action which, like several we have seen already in this book, fetches the next chunk of input to be parsed, returning an empty string when there is nothing remaining.
2. The parser that we want to run (`requestLineParser`).
3. Some initial input to feed into the parser preceding the chunks obtained from the action. In a later chapter we will see why this parameter exists; for now, we use `BS.empty`.

```
readRequestLine :: MaxChunkSize -> Socket -> IO RequestLine  
readRequestLine (MaxChunkSize mcs) s = do  
    result <- P.parseWith (S.recv s mcs)  
                        requestLineParser BS.empty  
    -
```

Attoparsec requires the end of input to be signified by an empty string, so for the first parameter, we have gone back to using the `S.recv` function from the network library.

We do not need to go into detail about the `Result` type, except to note that there is an `eitherResult` function to convert it to an error message `String` or a successfully-parsed value.

```
P.eitherResult :: Result a -> Either String a
```

What shall we do with a `Left String` value? Let us do our best to ignore

it. For the Left case, use fail to throw an exception that contains the error string.

```
readRequestLine (MaxChunkSize mcs) s = do
    result <- P.parseWith (S.recv s mcs)
                        requestLineParser BS.empty
    case P.eitherResult result of
        Left errorMessage -> fail errorMessage
        Right requestLine  -> return requestLine
```

Server time! We've split it into two definitions because the main action is getting larger now and one always ought to present code in bite-sized portions.

resourceServer contains the one-time setup part:

```
resourceServer :: IO ()
resourceServer = do
    dir <- getDataDir
    let resources = resourceMap dir -- 1
    let maxChunkSize = MaxChunkSize 1024
    serve @IO HostAny "8000" \(s, _) ->
        serveResourceOnce resources maxChunkSize s
```

And the serveResourceOnce function specifies what we do on each incoming connection:

```

serveResourceOnce :: ResourceMap -> Socket -> IO ()
serveResourceOnce resources maxChunkSize s = runResourceT @IO do
    RequestLine _ target _ <-                                     -- 2
        liftIO (readRequestLine maxChunkSize s)
    filePath <- liftIO (getTargetFilePath resources target) -- 3
    (_, h) <- binaryFileResource filePath ReadMode
    let r = hStreamingResponse h maxChunkSize                    -- 4
    liftIO (sendStreamingResponse s r)

```

1. In a preliminary setup step, construct the resource Map.
2. The first thing to do when a client connects is read the beginning portion of the message they send. We only care about the request target, so the other two fields are discarded.
3. We map the RequestTarget to the appropriate FilePath using getTargetFilePath (not yet written).
4. The response is constructed and sent, as before.

```

getTargetFilePath :: ResourceMap -> RequestTarget -> IO FilePath

```

This function does not actually involve I/O, but we have placed it in an IO context because again here we have a failure possibility – two, actually.

1. In our oversimplified view of a request target, we’re interpreting the byte string as ASCII text. If the string contains non-ASCII bytes, we must fail here.
2. If the request target is not one of the keys in the Map, we deal with that by throwing an exception.

```
targetFilePathMaybe ::
  ResourceMap -> RequestTarget -> Maybe FilePath
targetFilePathMaybe (ResourceMap rs) (RequestTarget target) = do
  resource <- A.convertStringMaybe target -- 1
  Map.lookup resource rs                  -- 2
```

```
getTargetFilePath rs target =
  case targetFilePathMaybe rs target of
    Nothing -> fail "not found"
    Just fp  -> return fp
```

An observant reader may notice that the first error case is actually unreachable in the `resourceServer` program, because the `P.takeWhile A.isVisible` line in `requestTargetParser` ensures that only visible ASCII characters ever appear in a `RequestTarget`. Since here we find ourselves performing the same check a second time, this suggests a mistake in the data model that we established way back in chapter 6. Instead of defining `RequestTarget` as a `BS.ByteString`, perhaps we could have been more precise.

Now run `resourceServer` and try using `curl` to fetch these URLs:

- `http://localhost:8000/stream`
- `http://localhost:8000/read`
- `http://localhost:8000/whatever`



13.6 Exercises

Exercise 34 – Parsing parentheses In exercise 4, you wrote a function that removes a set of parentheses from a string.

```
unParen :: Text -> Maybe Text
```

```
λ> unParen (Text.pack "(cat)")  
Just "cat"
```

Now write an equivalent parser.

```
parenParser :: Parser BS.ByteString
```

It should behave as follows:

```
λ> P.parseOnly parenParser [A.string|(cat)|]  
Right "cat"
```

Exercise 35 – To digit, maybe Can you write `digitParser` without using any partial functions? Browse the `ASCII` module for ideas.

Exercise 36 – Better parse errors Revise `httpVersionParser` in the same way that we revised `requestLineParser`. For each of the five sub-parsers, do one of the following:

- ▶ If the sub-parser already gives a reasonably useful error message, use `<?>` to contextualize it.
- ▶ Otherwise, replace the sub-parser's error message with something useful, using `<|>` and `fail`.

As mentioned in the chapter, we want the following behavior to result:

```
λ> P.parseOnly requestLineParser [A.string|GET /hello HTTP1.1|]
Left "Version: Failed reading: HTTP should be followed by a slash"
```

Exercise 37 – Status line In this book we focus on writing an HTTP server, so we will only be parsing requests. If you were writing an HTTP client, you would need to be able to parse responses.

Write a parser for `StatusLine`.

```
statusLineParser :: Parser StatusLine
```

Then write a function to test it:

```
statusLineParseTest :: StatusLine -> Maybe String
```

This function should encode the `StatusLine` and then feed the result into `statusLineParser`. If everything is correct, this function should return `Nothing`.

```
λ> statusLineParseTest (status ok)
Nothing
```

The test function should return `Just` if it detects a problem.

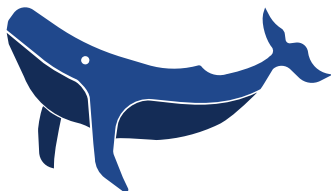
- ▶ If the parsing fails, return the error message from `Attoparsec`.
- ▶ If parsing succeeds but does not produce the correct value, return a string that shows the erroneous value.

You will need to add some `deriving` clauses to the datatypes from chapter 6.

Exercise 38 – P.string Suppose Attoparsec did not provide the `P.string` function. Can you define it yourself using `P.anyWord8` and `fail`?

Chapter 14

Errors



Our `resourceServer` program from the previous chapter has some shameful omissions that we will correct now.

```
$ curl 'http://localhost:8000/whatever'  
curl: (52) Empty reply from server
```

“Empty reply” means our server sent nothing at all in response to the HTTP request; it just disconnected from the client with no justification given. This is not how a proper web server is supposed to behave! If something goes wrong, we’re still supposed to reply to offer some explanation for what happened. We should also output some information elsewhere for the benefit of the server operator.

In this chapter we will introduce the `ExceptT` monad transformer, but not for a while yet. Before we jump into changing the definition of `resourceServer`, we need to make some decisions about how our server should react to each possible problem.

14.1 Status codes

The status codes are listed in RFC 7231.

<https://tools.ietf.org/html/rfc7231>

Please read the following from RFC 7231:

- ▶ The first few pages of section 6, *Response Status Codes*
- ▶ Skim the remainder of the section for a general idea of what kind of information status codes are used to convey.

There are two places in the code where we explicitly bailed out with `fail` and should return a response instead:

400 Bad request If parsing fails, we should respond with status code 400 to indicate that we were unable to make any sense out of the request message.

```
badRequest = Status
  (StatusCode Digit4 Digit0 Digit0)
  (ReasonPhrase [A.string|Bad request|])
```

404 Not found If the request target cannot be mapped to any of the resources we server, we should respond with status code 404.

```
notFound = Status
  (StatusCode Digit4 Digit0 Digit4)
  (ReasonPhrase [A.string|Not found|])
```

There is also a less visible source of failure:

500 Server error Any I/O operation might throw an exception. We should handle exceptions by responding with status code 500.

```
serverError = Status
  (StatusCode Digit5 Digit0 Digit0)
  (ReasonPhrase [A.string|Server error|])
```

Finally, there are two reasons why, even if we are able to give a successful response, we should refrain from doing so:

405 Method not allowed The only type of request that our server is currently able to service is when the client asks to “get” a resource. Clients are not allowed to ask our server to, for example, modify or delete files. So we should be looking at the request method to verify that it is “GET”. If not, we will reject the request with status code 405 to indicate that we do not support whatever other operation the client has requested.

```
methodNotAllowed = Status
  (StatusCode Digit4 Digit0 Digit5)
  (ReasonPhrase [A.string|Method not allowed|])
```

505 HTTP version not supported We’ll also look at the HTTP version. If it is not the version supported by our server, we should reject the request with error code 505.

```
versionNotSupported = Status
  (StatusCode Digit5 Digit0 Digit5)
  (ReasonPhrase [A.string|HTTP version not supported|])
```

When testing with `curl`, you may use the `--http1.1` flag to specify what protocol version `curl` should use when generating HTTP requests.

14.2 Constructing responses

A response that has an error status code will be brief; there is not much to say other than that a problem as occurred. But sometimes we try to include some helpful information in the response body to informally explain what went wrong.

Recall in chapter 9 we wrote the `textOk` function that constructs a simple text response:

```
textOk :: LT.Text -> Response
```

We will need a variant of that function with a parameter for the `Status`. Also, a few error codes require us to provide additional information in header fields, so our new function will need a `[HeaderField]` parameter.

```
textResponse :: Status -> [HeaderField] -> LT.Text -> Response
textResponse s additionalHeaders bodyText =
    Response
        (status s)                                -- 1
        ([typ, len] <> additionalHeaders)         -- 2
        (Just body)
    where
        typ = HeaderField contentType plainUtf8
        len = HeaderField contentLength (bodyLengthValue body)
        body = MessageBody (LT.encodeUtf8 bodyText)
```

1. The status now comes from a parameter rather than being fixed to `ok`.
2. Any additional header fields are appended after `content-type` and `content-length` (the order of the header fields does not matter).

14.3 Visibility in two places

Sending error responses makes more information about problems visible to the HTTP client. To make problems visible to ourselves, we will introduce *logging*. All this means is that our server will print messages to `stdout`, but when it's a server we call that "logging" to make it sound more sophisticated.

There is all manner of information that we might want to include in a logged event: when it happened, whether it is a problem (and if so, how severe), and any other context that might help in tracking down the source of a problem or in aggregating events into a wholistic picture of how well the system is doing. For demonstration, as well as for many small-scale applications, all an event needs to be is the string of text that we will see in the terminal.

```
data LogEvent = LogEvent LT.Text
```

```
printLogEvent :: LogEvent -> IO ()  
printLogEvent (LogEvent x) = LT.putStrLn x
```

We will find it useful to have an `Error` type that represents what happens when a particular erroneous situation arises. It consists of two pieces of information: what response (if any) to write to the socket, and what events (if any) to write to the log. The former is for the benefit of the client; the latter is for ourselves.

```
data Error = Error (Maybe Response) [LogEvent]
```

For each error condition, we should ask ourselves: When this happens, what information does the client need, and what information would I like to collect?

If the request is malformed Let's start with the fail in `readRequestLine`.

```
readRequestLine (MaxChunkSize mcs) s = do
    result <- P.parseWith (S.recv s mcs)
        requestLineParser BS.empty
    case P.eitherResult result of
        Left errorMessage -> fail errorMessage -- What happened?
        Right requestLine  -> return requestLine
```

If the `RequestLine` parser is failing, that indicates one of two things: either there is a client mistake in the software that encoded the request message, or there is a server mistake in our own `requestLineParser`. So let's make sure that the error message from the parser is visible both to the client and in the server log.

```
requestParseError :: String -> Error
requestParseError parseError = Error (Just response) [event]
    where
        response = textResponse badRequest [] message
        event = LogEvent message
        message = TB.toLazyText (
            TB.fromString "Malformed request: " <>
            TB.fromString parseError)
```

The `parserError` string will be the error message that we get from `Attoparsec`. This is why we bothered to worry about the quality of the error messages using `<?>` and `fail` in the previous chapter; here that information becomes a visible part of the server's behavior.

If the requested resource does not exist Now we move on to the fail in `getTargetFilePath`.

```
getTargetFilePath rs target =  
  case targetFilePathMaybe rs target of  
    Nothing -> fail "not found" -- What happened?  
    Just fp  -> return fp
```

This is the situation wherein a client has requested a resource that the server does not have. In real servers online, this happens all the time. Hackers request resources at random to search for servers that accidentally expose information they shouldn't, people try to access resources that no longer exist, and sometimes people simply make typographical errors. Whether you would want to see this information in a log is up to you – it can be amusing to see what antics the hackers get up to – but we will assume that you don't need this noise, so we will not log any events.

There is not much to do here other than send the appropriate status code.

```
notFoundError :: Error  
notFoundError = Error (Just response) []  
where  
  response = textResponse notFound [] message  
  message = LT.pack "It just isn't there!"
```

We included a playful message body, but it serves no real purpose and might as well be an empty string.

Depending on the server's purpose, a not-found response body might have a content type that matches the type of the resource that the client is expecting to receive. For example, if we know that the request is for an HTML document, then our error response might be an HTML document that contains some links to other resources that do exist. This content would then be visible in the web browser of a disappointed user.

If the method is not GET The only operation we presently support is *get*, and so our server should reject any other kind of request method.

RFC 7231 section 6.5.5, *405 Method Not Allowed*, tells us that whenever we reject a request for having an unsupported method, we must include the *Allow* header which tells the client what methods we *do* support. Section 7.4.1, *Allow*, tells us that the *Allow* field value must be a comma-separated list of methods such as “Allow: GET, HEAD, PUT”.

Our header will only say “Allow: GET”, but we’ll go ahead and write a more general error-constructing function with a `[Method]` parameter in anticipation of supporting more methods later on.

```
methodError :: [Method] -> Error
methodError supportedMethods = Error (Just response) []
  where
    response = textResponse methodNotAllowed [allow] LT.empty
    allow = HeaderField
      (FieldName [A.string|Allow|])
      (FieldValue (BS.intercalate (A.lift [A.Comma, A.Space])
        (map (\(Method m) -> m) supportedMethods)))
```

Intercalate joins a list of strings separated by delimiter, in this case a comma and a space.

```
BS.intercalate :: ByteString -> [ByteString] -> ByteString
```

Similar functions of the same name may also be found in `Data.Text` and `Data.List`.

If the file doesn’t open In `serveResourceOnce`, we have an implicit source of potential error at the point where we open the requested file.

```

serveResourceOnce resources maxChunkSize s = runResourceT @IO do
  RequestLine _ target _ <-
    liftIO (readRequestLine maxChunkSize s)
  filePath <- liftIO (getTargetFilePath resources target)

  -- What might go wrong here?
  (_, h) <- binaryFileResource filePath ReadMode

  let r = hStreamingResponse h maxChunkSize
  liftIO (sendStreamingResponse s r)

```

There are several reasons why the `binaryFileResource` action might throw an exception instead of obtaining a `Handle` – for example, if the file has been deleted. We will consider this situation to be a *server error*. This means that the client did nothing wrong, and they requested something that we normally expect to be able to provide, but we currently cannot provide it due to some mistake that we take responsibility for.

```

fileOpenError :: FilePath -> Ex.IOException -> Error
fileOpenError filePath ex = Error (Just response) [event]
  where
    response = textResponse serverError []
      (LT.pack "Something went wrong.")
    event = LogEvent $ TB.toLazyText $
      TB.fromString "Failed to open file " <>
      TB.fromString (show filePath) <> TB.fromString ": " <>
      TB.fromString (displayException ex)

```

The response body for a server error is usually terse and obscure – in this case, “Something went wrong.” This is because the client has no context for understanding the problem, and there is nothing they could do about it anyway. Furthermore, the internal workings and tribulations

of our server sometimes contain sensitive information. It is not typically worth carefully considering the security ramifications of one's error messages just to provide the client with information they have no use for anyway. Our internal error logging is the much more important consideration here.

Too late to handle With HTTP, the status line is a point of no return; once the status line has been sent, there is no way to change the status code of the response. This means that if anything goes wrong after that point, it is impossible to convey the cause of failure to the client.

```
serveResourceOnce resources maxChunkSize s = runResourceT @IO do
  RequestLine _ target _ <-
    liftIO (readRequestLine maxChunkSize s)
  filePath <- liftIO (getTargetFilePath resources target)
  (_, h) <- binaryFileResource filePath ReadMode
  let r = hStreamingResponse h maxChunkSize

  -- What might go wrong within this part?
  liftIO (sendStreamingResponse s r)
```

`sendStreamingResponse` first sends the status line and header fields, and then copies data from the file handle to the socket by repeatedly applying `BS.hGetSome` and `Net.send`.

What if `Net.send` throws an exception? This is truly nothing we could do here, because it means that the connection to the client has been lost. Perhaps a physical cable has been severed. Abrupt closures are an inherent possibility in networking.

What if `BS.hGetSome` throws an exception? We would wish at this point to communicate to the client that the server has experienced some internal error. However, the server has already sent a status line with code 200;

there is no way to retroactively change it to 500. The only thing that can happen at this point is to close the connection. The client, having received an incomplete message, will be left without knowing the cause of failure.

We cannot send an error response in these circumstances, but we can print something to the log.

```
lateError :: Ex.IOException -> Error
lateError e = Error Nothing [event]
  where
    event = LogEvent (LT.pack (displayException e))
```

This is similar to how we employed both `close` and `gracefulClose` when managing sockets, each used in the appropriate circumstance. We follow protocols wherever possible, but some conditions are unrecoverable. We design software that behaves only as nicely as circumstance permits.

14.4 Thread-safe logging

We must remember that we are writing multi-threaded programs; our servers here handle requests concurrently. It is easy to forget, because we have barely had to deal with the concurrency at all; the `serve` function takes care of forking the threads, and the only time we had to reckon with thread safety was when we introduced mutable state in chapter 10. There we discussed STM and how it allows multiple threads to perform sequences of actions on the same mutable state concurrently without any risk of interleaving the steps in an order that might not make sense.

When writing to a `Handle` as our `printLogEvent` function does, another unfortunate interleaving potential arises. Suppose two threads concurrently print “hello” and “world” to the same handle. We might expect to see either “hello world” or “world hello”, depending on which thread starts first. What we would never want to see is “hweolrllod”. However,

such a thing is possible if multiple concurrent writers are stepping on each other's toes.

The `unfork` package offers a ready remedy.

```
import Unfork (unforkAsyncIO_)
```

The function we're using has an unpleasant sort of type signature.

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

It will be more informative to look at the specialization of this type in the context where we will be using it:

```
unforkAsyncIO_ ::  
    (LogEvent -> IO ())           -- 1  
-> ( (LogEvent -> IO ()) -> IO () ) -- 2  
    -> IO ()
```

1. The first argument is a `LogEvent -> IO ()` action that is not thread-safe – in our case, `printLogEvent` – that we want to be able to use safely from multiple threads.
2. The second argument is where we give the main body of the program. It receives a different `LogEvent -> IO ()`, a modified version of `printLogEvent` that is safe to use concurrently.

In the remainder of this chapter we will be writing a new version of `resourceServer`. The new functions will be suffixed with an 'X' to distinguish them from the definitions of the previous chapter. The new server opens as follows:

```

resourceServerX = do
  dir <- getDataDir
  let resources = resourceMap dir
  let maxChunkSize = MaxChunkSize 1024
  unforkAsyncIO_ printLogEvent \log ->
    serve @IO HostAny "8000" \(s, _) -> do
    -

```

14.5 Either

The previous chapter involved a number of actions:

```

serveResourceOnce      :: ... -> IO ()
readRequestLine        :: ... -> IO RequestLine
getTargetFilePath      :: ... -> IO FilePath
binaryFileResource     :: ... -> ResourceT IO (ReleaseKey, Handle)
sendStreamingResponse  :: ... -> StreamingResponse -> IO ()

```

We applied `liftIO` to the `readRequestLine`, `getTargetFilePath`, and `sendStreamingResponse` steps to convert their types from `IO` to `ResourceT IO` to match the type of the `binaryFileResource` step so that all four actions could be performed in one `do` block together. We then applied `runResourceT` to convert the whole sequence back to `IO` to form the definition of `serveResourceOnce`.

`serveResourceOnceX` will involve a similar sequence of actions, but now each one has an `Error` possibility in its result. The new action types will look like this:

```

serveResourceOnceX      :: ... -> IO (Either Error ())
readRequestLineX        :: ... -> IO (Either Error RequestLine)
getTargetFilePathX      :: ... -> Either Error FilePath
binaryFileResourceX     :: ... -> ResourceT IO (Either Error
                                (ReleaseKey, Handle))
sendStreamingResponseX :: ... -> IO (Either Error ())

```

Notice that in addition to adding `Either Error` to all five types, we will also remove the `IO` from `getTargetPath`. This function never actually did any real I/O; it was only in the `IO` type so that we could use `fail :: (String -> IO a)` to throw an exception. Since `getTargetFilePathX` conveys failure via `Either Error` instead of an exception, it no longer needs the `IO`. Instead of `fail` and `return`, we now use `Left` and `Right`:

```

getTargetFilePathX ::
    ResourceMap -> RequestTarget -> Either Error FilePath
getTargetFilePathX rs t =
    case targetFilePathMaybe rs t of
        Nothing -> Left notFoundError -- was: fail "not found"
        Just fp  -> Right fp          -- was: return fp

```

In `readRequestLine`, likewise the only thing that changes is how we return the results. Instead of `fail ...`, we use `return (Left ...)`. Instead of `return ...`, we say `return (Right ...)`.

```
readRequestLineX ::
    MaxChunkSize -> Socket -> IO (Either Error RequestLine)
readRequestLineX (MaxChunkSize mcs) s = do
    result <- P.parseWith (S.recv s mcs)
        requestLineParser BS.empty
    case P.eitherResult result of
        Left errorMessage ->    -- was: fail errorMessage
            return (Left (requestParseError errorMessage))
        Right requestLine ->    -- was: return requestLine
            return (Right requestLine)
```

Recall that we defined `binaryFileResource` in chapter 3 as:

```
binaryFileResource ::
    FilePath -> IOMode -> ResourceT IO (ReleaseKey, Handle)
binaryFileResource path mode =
    allocate (IO.openBinaryFile path mode) IO.hClose
```

If the file cannot be opened and `IO.openBinaryFile` throws an exception, we want to return a `Left` value. Our means of turning exceptions into `Left`s is the “try” family of functions from the `Control.Exception.Safe` module.

```
import qualified Control.Exception.Safe as Ex
```

```
Ex.tryIO :: MonadCatch m => m a -> m (Either IOException a)
```

`MonadCatch` most frequently specializes to `IO`.

```
Ex.tryIO @IO :: IO a -> IO (Either IOException a)
```


Fortunately, `ResourceT IO` also belongs to the `MonadCatch` class. This is how we will use it:

```
Ex.tryIO @(ResourceT IO) ::  
  ResourceT IO (ReleaseKey, Handle)  
  -> ResourceT IO (Either IOException (ReleaseKey, Handle))
```

Whereas `tryIO @IO` catches any `IOException` that arises, `tryIO @(ResourceT IO)` more specifically catches an `IOException` that arises in either the resource acquisition step of an `allocate` or in any `IO` action incorporated into `ResourceT` with `liftIO`. It does not catch any exception that arises in the resource release step of an `allocate`.

So, the `tryIO` function is perfect here, and it does most of the work for us; its return type is already very nearly what we want. The only thing we have to do is change the `Left` type. We need to replace the `IOException` that it gives us with an `Error` – our specification of what to log and what response to send. We need to turn that `Either IOException ...` into `Either Error ...`.

```
binaryFileResourceX :: FilePath -> IOMode  
  -> ResourceT IO (Either Error (ReleaseKey, Handle))  
binaryFileResourceX fp mode = do  
  result <- Ex.tryIO (binaryFileResource fp mode)  
  case result of  
    Left e -> return (Left (fileOpenError fp e))  
    Right x -> return (Right x)
```

Then there is the matter of checking the request method, something we didn't concern ourselves at all with in the previous chapter. In this case, the `Right` type is `()` because there is no information that needs to be returned in the success case. The only thing that this function does is potentially generate an `Error`.

```
requireMethodX :: [Method] -> Method -> Either Error ()
requireMethodX supportedMethods x =
    case (elem x supportedMethods) of
        False -> Left (methodError supportedMethods)
        True -> Right ()
```

If you have not already done so, go ahead and add `deriving (Eq, Show)` to each of the datatype definitions from chapter 6. The `Method` type needs an instance of `Eq` to support this use of the `elem` function.

14.6 ExceptT

We must now combine the steps together into `serveResourceOnceX`.

```
serveResourceOnceX :: ResourceMap -> MaxChunkSize
-> Socket -> IO (Either Error ())
```

First we will present a straightforward but cumbersome approach.

1. Attempt to read the request line to obtain an `Either Error RequestLine`.
 - a. If the parse result is an `Error`, just return the error.
 - b. If the parse result is a `RequestLine`, then continue.
2. Attempt to look up the target in the `ResourceMap`.
 - a. If the lookup resulted in an `Error`, just return the error.
 - b. If we got a `FilePath`, then continue.
3. Attempt to open a file handle.
 - a. If the open resulted in an `Error`, just return the error.
 - b. If we got a `Handle`, then continue.
4. Send the response.

```

serveResourceOnce resources maxChunkSize s =
  runResourceT @IO do
    requestLineEither <- -- 1
      liftIO (readRequestLineX maxChunkSize s)
    case requestLineEither of
      Left e -> return (Left e) -- 1 a
      Right (RequestLine _ target _) -> -- 1 b
        case getTargetFilePathX resources target of -- 2
          Left e -> return (Left e) -- 2 a
          Right fp -> do -- 2 b
            handleEither <-
              binaryFileResourceX fp ReadMode -- 3
            case handleEither of
              Left e -> return (Left e) -- 3 a
              Right (_, h) -> do -- 3 b
                let r = hStreamingResponse h maxChunkSize
                liftIO (sendStreamingResponseX s r) -- 4

```

There are several bothersome aspects of this code. We dislike that the line `Left e -> return (Left e)` is repeated three times. It is also frustrating that we had to add a new level of indentation for each step. The `do` notation was meant to save us from that sort of awkwardness!

What we are doing here is strongly reminiscent of what the `Either` monad does: Stop at the first `Left` value encountered, otherwise keep going. But the three steps we're sequencing aren't of type `(Either Error a)`; their types are of the form `ResourceT IO (Either Error a)`. The Monad involved in the `do` sequencing is `ResourceT IO`, and the `Eithers` are merely values passed around within the `do` block, not involved in the `do` composition itself.

Here is where `ExceptT` comes into play. The 'T', again, stands for 'transformer', because `ExceptT Error` is another monad transformer. It

is the third one that we have seen:

1. We used `ResourceT` to slightly augment `IO`, adding the ability to register deallocation actions that are guaranteed to run at the end.
2. We used `ListT` to substantially alter `IO`, allowing it to return multiple values instead of just one. `(ListT IO)` is an amalgamation of `IO` and `[]`, combining aspects of both.
3. `(ExceptT e)` alters a monad by imbuing it with the option to not return normally at all, but rather to produce an `e` instead. `(ExceptT e IO)` is an amalgamation of `IO` and `(Either e)`, combining aspects of both.

This type originally comes from the `transformers` package, but everything we're using is exported by `Relude`, so no new imports are needed.

All we need is the definition of the `ExceptT` datatype...

```
newtype ExceptT e m a = ExceptT (m (Either e a))
```

... and the function that deconstructs it, which is just the inverse of the `ExceptT` constructor.

```
runExceptT :: ExceptT e m a -> m (Either e a)
```

These definitions are worth committing to memory. Unlike `ResourceT` and `ListT`, which we used opaquely, with `ExceptT` we are choosing to acknowledge that it is merely a `newtype` and can be interchanged at will with its underlying representation.

- When you see the type `(ExceptT Error IO a)`, you should recognize that this is equivalent to `IO (Either Error a)` and recall that you can apply `runExceptT` to convert to that type.
- When you have a bunch of `IO (Either Error a)` actions that you want to run in sequence, stopping at the first `Error`, the first thing

that should come to mind is “I can wrap these up in the `ExceptT` constructor to convert to `(ExceptT Error IO a)`.”

In the previous chapter, the monadic context of the `do` block in `serveResourceOnce` was `ResourceT IO`. This means that the monad that we’re augmenting isn’t `IO`; it’s `(ResourceT IO)`. For example, the return type of `readRequestLineX` is:

```
IO (Either Error RequestLine)
```

After it is lifted into the `ResourceT` context using `liftIO`, this type will be:

```
ResourceT IO (Either Error RequestLine)
```

Then we will apply the `ExceptT` constructor, and the action’s type ends up as:

```
ExceptT Error (ResourceT IO) RequestLine
```

Applying the `ExceptT` constructor is the only thing we need to do to make use of the `ExceptT` monad. The `do` block in `serveResourceOnceX` is going to look almost exactly like the previous chapter, with the only difference being that each of these three fallible actions is now wrapped in `ExceptT`.

```

do
  RequestLine _ target _ <- ExceptT $ -- 1
    liftIO (readRequestLineX maxChunkSize s)
  fp <- ExceptT $ -- 2
    return (getTargetFilePathX resources target)
  (_, h) <- ExceptT $ binaryFileResourceX fp ReadMode -- 3
  let r = hStreamingResponse h maxChunkSize
  liftIO (sendStreamingResponseX s r)

```

We only want the `ExceptT` context for how its monadic sequencing works, and we do not need it outside of this one function. Once we're done, immediately outside of the `do` block we then use `runExceptT` on the combined action sequence to turn it back into a more ordinary `ResourceT IO (Either Error ())`.

```

serveResourceOnceX :: ResourceMap -> MaxChunkSize
-> Socket -> IO (Either Error ())
serveResourceOnceX resources maxChunkSize s =
  runResourceT @IO $ runExceptT @Error @ (ResourceT IO) do
    RequestLine method target version <- ExceptT $
      liftIO (readRequestLineX maxChunkSize s)
    fp <- ExceptT $
      return (getTargetFilePathX resources target)
    (_, h) <- ExceptT $ binaryFileResourceX fp ReadMode
    let r = hStreamingResponse h maxChunkSize
    ExceptT $ liftIO $ sendStreamingResponseX s r

```

This does not replace `runResourceT`, which is still there as the final step to give us the desired `IO (Either Error ())` result.

Monad stacks We made use of two monad transformers at once while working with the type constructor `(ExceptT Error (ResourceT IO))`. A

type like this is often described as a “monad stack”. IO is the “bottom” or “underlying” monad, this is augmented with ResourceT atop it, and finally (ExceptT Error) forms the outermost layer. One can continue to stack transformers higher and higher, though when taken to excess this approach can take on a cumbersome quality of its own.



14.7 Exercises

Exercise 39 – Resource server X Finish the definition of resourceServerX that we started writing in this chapter.

```
resourceServerX = do
  dir <- getDataDir
  let resources = resourceMap dir
  let maxChunkSize = MaxChunkSize 1024
  unforkAsyncIO_ printLogEvent \log ->
    serve @IO HostAny "8000" \(s, _) -> do
      -
```

In the blank, use serveResourceOnceX and consider what should happen if it returns a Left value.

Exercise 40 – Check the method and version Fill in the blank below to write the function that asserts that the HTTP version specified in the request line is the version supported by this server.

```
requireVersionX :: HttpVersion -> HttpVersion -> Either Error ()
requireVersionX supportedVersion requestVersion = _
```

Then make two modifications to serveResourceOnceX:

- Use `requireVersionX` to assert that the request protocol version is HTTP/1.1.
- Use `requireMethodX` to assert that the request method is GET.

Exercise 41 – A sinister request We like to use `curl` to test HTTP servers because we trust that it works correctly. But to test some of the code we have written in this chapter, we need an HTTP client that does *not* work correctly! If you want something done wrong properly, you have to do it yourself.

```
sendSinisterRequest :: IO ()
sendSinisterRequest = _
```

Write an action that connects to your server and elicits a “malformed request” error (status code 400) from it.

Start by using `(resolve "8000" "localhost")` to get the `AddrInfo` for your server. You may wish to review some of the exercises from chapters 4 and 5.

Coming up

Thank you for supporting this book! <♥> Chris and Julie are hard at work on completing the remaining chapters.

Chapter 15

Reading the head

Chapter 16

Reading the body

Chapter 17

Connection reuse

Appendix A

Solutions to exercises

Exercise 1 – File resource function

```
fileResource path mode =  
    allocate (IO.openFile path mode) IO.hClose
```

```
writeGreetingSafe = runResourceT @IO do  
    dir <- liftIO getDataDir  
    (_, h) <- fileResource (dir </> "greeting.txt") WriteMode  
    liftIO (IO.hPutStrLn h "hello")  
    liftIO (IO.hPutStrLn h "world")
```

Exercise 2 – Showing handles There is no single right answer here, but this is what we did:

```

handlePrintTest = runResourceT @IO do
  (_, fh1) <- fileResource "/tmp/show-test-1" WriteMode
  (_, fh2) <- fileResource "/tmp/show-test-2" ReadWriteMode
  liftIO $ for_ [stdin, stdout, stderr, fh1, fh2] \h -> do
    IO.putStrLn (show h)
    info <- IO.hShow h
    IO.putStrLn info

```

```

λ> handlePrintTest
{handle: <stdin>}
  {loc=<stdin>,type=readable,buffering=none}
{handle: <stdout>}
  {loc=<stdout>,type=writable,buffering=none}
{handle: <stderr>}
  {loc=<stderr>,type=writable,buffering=none}
{handle:
 /home/chris/.local/share/sockets-and-pipes/greeting.txt}
  {loc=/home/chris/.local/share/sockets-and-pipes/greeting.txt,
   type=read-writable,buffering=block (2048)}
{handle:
 /home/chris/.local/share/sockets-and-pipes/greeting2.txt}
  {loc=/home/chris/.local/share/sockets-and-pipes/greeting2.txt,
   type=writable,buffering=block (2048)}

```

Exercise 3 – Exhaustion

```
openManyHandles = go []  
  
where  
  go hs = do  
    r <- fileResourceMaybe  
    case r of  
      Nothing -> return hs  
      Just h -> go (h : hs)
```

```
fileResourceMaybe = do  
  dir <- liftIO getDataDir  
  result <- Ex.tryIO do  
    (_, h) <- fileResource (dir </> "greeting.txt") ReadMode  
    return (Just h)  
  case result of  
    Right x -> return x  
    Left e -> do  
      print (displayException e)  
      return Nothing
```

The definition of `open` is very similar to what we have seen in the chapter, the only difference being that the `Just` constructor must be applied to the result.

It is important to choose `ReadMode` as the IO mode. If you tried one of the writing modes, the exception would say “/tmp/ex.txt: openFile: resource busy (file is locked)” and the program would fail after opening only one handle.

The file path could be anything, but a file must exist at that path. Otherwise, the exception would say “/tmp/ex.txt: openFile: does not exist (No such file or directory)”.

Before printing the number, the output should say “/tmp/ex.txt: open-File: resource exhausted (Too many open files)”.

The number of handles will vary according to your operating system, but will probably be at least a few thousand. If the limit is significantly higher, this program may take a little while to run.

Exercise 4 – Pure text manipulation `digitsOnly` can be expressed quite directly using the `filter` method, which retains characters that match a predicate.

```
digitsOnly = Text.filter Char.isDigit
```

We need `unsnoc` to pull the last character off of the end of `Text`, and `snoc` to attach the upper case character back onto the end. (“`snoc`” is the backwards spelling of “`cons`”, the function which appends a character to the beginning of the text.) The return type of `unsnoc` is `Maybe (Text, Char)`, which forces us to deal with the `Nothing` case in which the input text is empty and thus cannot be subdivided in this way.

```
capitalizeLast t = case T.unsnoc t of
  Just (t', c) -> T.snoc t' (Char.toUpper c)
  Nothing -> T.empty
```

We can use `stripPrefix` to remove an opening paren character from the beginning of the text, and then `stripSuffix` to remove a closing paren from the end.

```
unParen t = case T.stripPrefix (T.pack "(") t of
  Just t' -> T.stripSuffix (T.pack ")") t'
  Nothing -> Nothing
```


If you are very familiar with the `Maybe` monad, you may have equivalently written something like this:

```
import Control.Monad ((>=>))

unParen :: Text -> Maybe Text
unParen =
    Text.stripPrefix (Text.pack "(") >=>
    Text.stripSuffix (Text.pack ")")
```

Exercise 5 – Character count

```
characterCount :: FilePath -> IO Int
characterCount fp = runResourceT @IO do
    dir <- liftIO getDataDir
    (_, h) <- fileResource (dir </> fp) ReadMode
    liftIO (hCharacterCount h)
```

```
hCharacterCount :: Handle -> IO Int
hCharacterCount h = continue 0
    where
        continue n = do
            x <- T.hGetChunk h
            case T.null x of
                True -> return n
                False -> continue (n + T.length x)
```

Exercise 6 – When and unless `unless` is the appropriate function here, because we want to proceed only when the result of `getChunk` is *not* the value that signifies the end of the input.

```
repeatUntil getChunk isEnd f = continue
  where
    continue = do
      chunk <- getChunk
      unless (isEnd chunk) do{ _ <- f chunk; continue }
```

Haskell gives us plenty of great ways to express a function over Bool:

```
when True action = action
when False _ = return ()
```

```
when condition action =
  case condition of
    True -> action
    False -> return ()
```

```
when condition action = if condition then action else return ()
```

Since `unless` should have exactly the opposite behavior of `when`, we can copy any of the `when` definitions and swap the `True` and `False` cases.

```
unless False action = action
unless True _ = return ()
```

Often when we have two functions that are so similar, we want to define one in terms of the other:

```
unless condition action = when (not condition) action
```

This approach can spare us from writing duplicate code, although it

does not make much difference in this particular situation.

Exercise 8 – A character encoding bug To introduce a very typical sort of mistake, we can switch to a different encoding function. For example, if we encode in UTF-16, then the `greet` function will not necessarily read the same characters back when it attempts to decode UTF-8.

Unfortunately, this mistake does not always produce noticeable consequences.

```
λ> greet (T.encodeUtf16BE (T.pack "David Hilbert"))
Hello, David Hilbert
```

This is because most common character encodings are similar enough that, even if we encode via one encoding and decode via a different encoding, in some cases we will still recover the original uncorrupted text!

In particular, most encodings have a great deal of commonality in how they represent the 128 characters of the ASCII character set. To see encoding bugs, we need to venture outside this range and use letters not found on an English keyboard.

Consider the Hungarian mathematician Paul Erdős. The Hungarian alphabet includes a diacritic mark called the “double acute accent,” which we see over the `o` in “Erdős”. If you cannot type this letter on your keyboard, you can write the Unicode character named “letter `O` with double acute” as `\337` in a Haskell string.

```
λ> putStrLn "Paul Erd\337s"
Paul Erdős
```

Here is Paul’s name used with the appropriate encoding:

```
λ> greet (T.encodeUtf8 (T.pack "Paul Erdős"))
Hello, Paul Erdős
```

When we use this name in conjunction with a mismatched encoding, we can now see a problem that ensues:

```
λ> greet (T.encodeUtf16BE (T.pack "Paul Erdős"))
Hello, Paul ErdQs
```

Exercise 9 – Byte manipulation

```
asciiUpper = BS.map f
  where
    f x | x >= 97, x <= 122 = x - 32
    f x = x
```

The guard ($| x \geq 97, x \leq 122$) ensures that the only characters we affect are the lower case letters 97 through 122. We modify these characters by subtracting 32, which is the difference between each lower case character and its upper case counterpart ($97 - 65 = 32$).

Exercise 10 – Improper ResourceT allocation The first argument to `allocate` should either create a socket or fail. In the flawed `openAndConnect` function, our `open` action has a third possibility: If the `connect` action throws an exception, `open` will create a socket *and* fail. With all references to the socket gone, there is then no way for the socket to ever be closed. What we have here is a *resource leak*.

We can still have this nice function; we just need to fiddle with the definition a bit. Instead of running the `connect` as part of the `allocate` setup action, it should be a subsequent step within the `ResourceT` context.

```

openAndConnect addrInfo =
  do
    (rk, s) <- allocate (S.openSocket addrInfo) S.close
    liftIO do
      S.setSocketOption s S.UserTimeout 1000
      S.connect s (S.addrAddress addrInfo)
    return (rk, s)

```

Exercise 11 – Explore Gopherspace

```

findGopher = do
  addrInfos <- S.getAddrInfo
    (Just S.defaultHints { S.addrSocketType = S.Stream })
    (Just "quux.org")
    (Just "gopher")
  case addrInfos of
    [] -> fail "getAddrInfo returned []"
    x : _ -> return x

```

```

viewGopherHomePage = runResourceT @IO do
  addrInfo <- liftIO findGopher
  (_, s) <- allocate (S.openSocket addrInfo) S.close
  liftIO do
    S.setSocketOption s S.UserTimeout 1000
    S.connect s (S.addrAddress addrInfo)
    S.sendAll s (T.encodeUtf8 (T.pack "\r\n"))
    repeatUntil (S.recv s 1024) BS.null BS.putStr

```

You may also want to try exploring the server a bit beyond the front page. Some lines of the output will be “links” to other pages. If you see something in the server’s response that starts with a slash like /About This

Server.txt, try using it as your request message:

```
"/About This Server.txt\r\n"
```

Exercise 12 – Address resolution Starting from `findHaskellWebsite`, all we need to do is introduce parameters in place of the string literals for the host name and service name.

```
resolve serviceName hostName =  
  do  
    addrInfos <- S.getAddrInfo  
      (Just S.defaultHints { S.addrSocketType = S.Stream })  
      (Just hostName)  
      (Just serviceName)  
  case addrInfos of  
    [] -> fail "getAddrInfo returned []"  
    x : _ -> return x
```

Exercise 13 – Repeat until nothing `repeatUntilNothing` could be written using `repeatUntil`:

```
repeatUntilNothing getChunk f =  
  repeatUntil getChunk isNothing \xMaybe ->  
    case xMaybe of  
      Nothing -> return ()  
      Just chunk -> f chunk
```

However, this is a bit awkward because `xMaybe` is always a `Just` value; the `Nothing` case is unreachable. This code inspects the `Maybe` constructor redundantly: first with `isNothing`, and then again with the pattern match. This doesn't put us into too much trouble in this case, but in general it hints that there is some room for improvement.

Things work out a bit better if do things the other way around; put the recursion in `repeatUntilNothing`, and use that to redefine `repeatUntil`.

```
repeatUntilNothing getChunk f = continue
  where
    continue :: IO ()
    continue = do
      chunkMaybe <- getChunk
      case chunkMaybe of
        Nothing -> return ()
        Just chunk -> do{ _ <- f chunk; continue }
```

```
repeatUntil getChunk isEnd = repeatUntilNothing do
  x <- getChunk
  return (if isEnd x then Nothing else Just x)
```

Exercise 14 – Make an HTTP request The program then proceeds in the same manner as the examples from chapter 4: Look up the address, connect, send, and receive. We can write the request just like we did in the chapter. Don't forget the blank line at the end that indicates the end of the header field list.

```
haskellOrgHttpExchange = runResourceT @IO do
  addressInfo <- liftIO (resolve "http" "haskell.org")
  (_, s) <- openAndConnect addressInfo
  liftIO $ Net.send s $
    line [A.string|GET / HTTP/1.1|] <>
    line [A.string|Host: haskell.org|] <>
    line [A.string|Connection: close|] <>
    line [A.string|]
  liftIO $ repeatUntilNothing (Net.recv s 1024) BS.putStr
```

The first thing we see is the status line.

```
HTTP/1.1 301 Moved Permanently
```

This is followed by response headers:

```
Date: Fri, 08 May 2020 21:23:18 GMT
Server: Apache/2.2.22 (Debian)
Location: https://haskell.org/
...
```

The “moved permanently” status, along with the `Location` header, is telling us that the response does not actually contain the home page that we asked for. Rather, the server wants us to retry the request using the `https` protocol. HTTPS stands for “HTTP Secure”; it is the same protocol, plus a layer of encryption. This book will not deal with encryption, so our experiments with attempting to communicate with `haskell.org` will end here.

After a blank line, we then see the response body, which is an HTML document (as the `Content-Type` header field indicated it would be).

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
...
```

When we make the request with `curl`, it prints only the HTML, not the headers, because that is typically the only part of the response that we care to see.

Exercise 15 – Test the hello server No matter which address or client you use, you should see the same “Hello!” response.


```
$ curl http://127.0.0.1:8000
Hello!
```

```
$ curl http://[::1]:8000
Hello!
```

If you look at the output printed in GHCi, you can see that the client's socket address reflects whether the request was made via IPv4 or IPv6.

```
New connection from [::ffff:1.0.0.127]:44772
New connection from [::1]:58706
```

The number at the end (in the above examples, 44772 and 58706) will change each time. This is a number you don't ordinarily see – it is a *temporary* port assigned to each client connection. We mention this only for curiosity's sake.

When you use a web browser, you might (depending on the browser) see that the browser actually opened more than one connection. For example, Firefox makes an additional request for `/favicon.ico` in hopes of finding an icon that it can display in the browser tab. Therefore each time we load

the page in Firefox, our server prints two lines instead of one.

Exercise 16 – Construct some values

```
helloRequest = Request start [host, lang] Nothing
  where
    start = RequestLine
      (Method [A.string|GET|])
      (RequestTarget [A.string|/hello.txt|])
      (HttpVersion Digit1 Digit1)
    host = HeaderField
      (FieldName [A.string|Host|])
      (FieldValue [A.string|www.example.com|])
    lang = HeaderField
      (FieldName [A.string|Accept-Language|])
      (FieldValue [A.string|en, mi|])
```

```
helloResponse = Response start [typ, len] (Just body)
  where
    start = StatusLine
      (HttpVersion Digit1 Digit1)
      (StatusCode Digit2 Digit0 Digit0)
      (ReasonPhrase [A.string|OK|])
    typ = HeaderField
      (FieldName [A.string|Content-Type|])
      (FieldValue [A.string|text/plain; charset=us-ascii|])
    len = HeaderField
      (FieldName [A.string|Content-Length|])
      (FieldValue [A.string|6|])
    body = MessageBody [A.string|Hello!|]
```

Exercise 17 – Infinite byte strings Since lazy byte strings are much like lists, there are many similarities between the `Data.ByteString.Lazy` and

`Data.List` modules. Both of the list functions in the given example, `cycle` and `take`, have equivalents of the same name for lazy byte strings.

```
LBS.cycle :: LBS.ByteString -> LBS.ByteString
LBS.take  :: Int64 -> LBS.ByteString -> LBS.ByteString
```

There are many ways to construct the initial byte string to use as the argument to `cycle`. Here we will use the ASCII string quasi-quoter.

```
infiniteLaughs = LBS.cycle [A.string|ha|]
```

We don't want to try to print `infiniteLaughs` (since it will go on printing forever), but we can peek at the beginning.

```
laugh = LBS.take 12 infiniteLaughs
```

```
λ> laugh
"hahahahahaha"
```

Exercise 18 – Header field encoding

```
encodeHeaderField (HeaderField (FieldName x) (FieldValue y)) =
  BSB.byteString x
  <> A.lift [A.Colon, A.Space]
  <> BSB.byteString y
```

The colon and space may also be written as a quasi-quotation:

```
encodeHeaderField (HeaderField (FieldName x) (FieldValue y)) =
  BSB.byteString x <> [A.string|: |] <> BSB.byteString y
```

Alternatively, you may have chosen to break the solution down into more definitions.

```
encodeHeaderField (HeaderField x y) =  
    encodeFieldName x <> [A.Colon, A.Space] <> encodeFieldValue y
```

```
encodeFieldName :: FieldName -> BSB.Builder  
encodeFieldName (FieldName x) = BSB.byteString x
```

```
encodeFieldValue :: FieldValue -> BSB.Builder  
encodeFieldValue (FieldValue x) = BSB.byteString x
```

Note that the result should *not* include a line terminator at the end.

Exercise 19 – Message body encoding Because we defined `MessageBody` as a lazy byte string, we have to use the `BSB.lazyByteString` instead of `BSB.byteString`.

```
encodeMessageBody (MessageBody x) = BSB.lazyByteString x
```

The function should *not* be written like this:

```
encodeMessageBody (MessageBody x) =  
    BSB.byteString (LBS.toStrict x)    -- No!
```

This compiles, but conversion to a strict byte string forces evaluation of the entire body at once, eliminating laziness and generally defeating the purpose of `LBS.ByteString` and `BSB.Builder`.

Exercise 20 – Encoding test We can apply `encodeRequest` to `hello-Request`. However, because the `Builder` type does not belong to the `Show`

class, GHCi cannot print it.

```
λ> encodeRequest helloRequest
```

error:

- No instance for (Show BSB.Builder) arising from a use of 'print'

To actually see the contents, we have to convert the builder to a (lazy) byte string.

```
λ> BSB.toLazyByteString (encodeRequest helloRequest)
"GET /hello.txt HTTP/1.1\r\nHost: www.example.com\r\n
Accept-Language: en, mi\r\n\r\n"
```

And we can do the same with `encodeResponse` for `helloResponse`.

```
λ> BSB.toLazyByteString (encodeResponse helloResponse)
"HTTP/1.1 200 OK\r\nContent-Type: text/plain; charset=us-ascii
\r\nContent-Length: 6\r\n\r\nHello!"
```

To compare the encoding result to the hand-written `helloResponseString` string from chapter 5, we need another conversion; `BSB.toLazyByteString` produces a lazy string, whereas we defined `helloResponseString` using the strict string type.

```
λ> helloResponseString ==
    BSB.toLazyByteString (encodeResponse helloResponse)
```

error:

- Couldn't match expected type 'ByteString'
with actual type 'LBS.ByteString'

To get the two strings to be the same type, we must convert one of them. We might force evaluation of the lazy string to get a strict string:

```
λ> helloResponseString == LBS.toStrict (  
    BSB.toLazyByteString (encodeResponse helloResponse))  
True
```

A slightly more sensible choice would be to make the strict string lazy.

```
λ> LBS.fromStrict helloResponseString ==  
    BSB.toLazyByteString (encodeResponse helloResponse)  
True
```

In either case, the result of `True` tells us that `encodeResponse` did indeed produce the desired value for this message.

Exercise 21 – Read the header The `curl` command and output ought to look like this:

```
$ curl localhost:8000 --dump-header -  
HTTP/1.1 200 OK  
Content-Type: text/plain; charset=us-ascii  
Content-Length: 40  
  
Hello!  
This page has never been viewed.
```

Exercise 22 – Overflow The midpoint between 210 and 230 should be 220.

```
λ> mid 210 230  
92
```

When x is 210 and y is 230, $(x + y)$ is 440 – rather, it would be, if the `Word8` type could go that high. But it cannot, so the result we actually get is... something else.

```
λ> 210 + 230 :: Word8
184
```

When we do arithmetic with bounded number types, we have to stay within the bounds at all times!

With some extra-careful thought, you may have produced a solution like this one:

```
mid x y | x <= y    = x + ((y - x) `div` 2)
        | otherwise = y + ((x - y) `div` 2)
```

This works, but it may be difficult to see why. We can instead spare ourselves this mental taxation by doing all of the arithmetic with an unbounded type. Convert each argument to `Integer`, perform the addition and division, then convert the result back from `Integer`.

```
mid x y = fromInteger ((toInteger x + toInteger y) `div` 2)
```

Arithmetic may be faster with the bounded types than with `Integer`, but getting the right answer should come first.

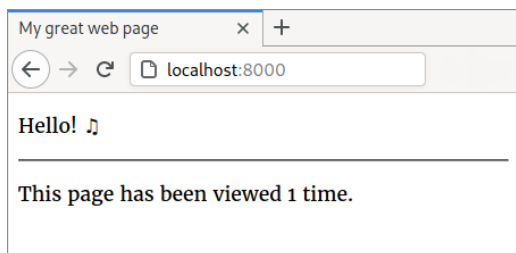
Exercise 23 – HTML in the browser To write the `htmlOk` function, again we have to change two things about the construction of the response:

1. The value of the `Content-Type` header should now be `htmlUtf8`;
2. We now use `renderHtml` to convert the `Html` value to a byte string.

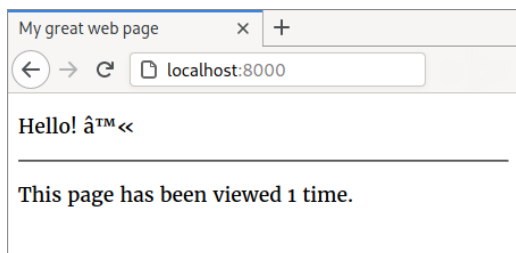
```
htmlOk str = Response (status ok) [typ, len] (Just body)
  where
    typ = HeaderField contentType htmlUtf8
    len = HeaderField contentLength (bodyLengthValue body)
    body = MessageBody (renderHtml str)
```

The server action looks just like `countingServer`; we just only to swap in our two new `Html`-based functions in place of the `Text` ones.

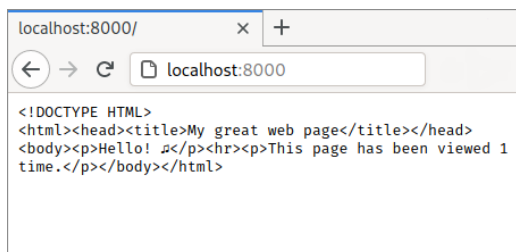
```
stuckCountingServerHtml = serve @IO HostAny "8000" \(s, _) -> do
  let count = 0 -- to-do!
  sendResponse s (htmlOk (countHelloHtml count))
```



There are a few interesting mistakes we could make with the content type. If we change the value of the `Content-Type` header to `"text/html; charset=ascii"`, then the '♪' character which is represented as three bytes in its UTF-8 encoded form is presented as three ASCII characters.



If we change the content type to “text/plain; charset=utf-8”, the web browser no longer knows that it is receiving an HTML page. What it displays, then, is the HTML itself – not rendered into a web page but simply shown to us still in its encoded form.



Exercise 24 – Type ambiguity There are two error messages:

- Ambiguous type variable `v0` arising from a use of `. =` prevents the constraint `ToJSON v0` from being solved.
- Ambiguous type variable `v0` arising from the literal `"Hello! \9835"` prevents the constraint `IsString v0` from being solved.

When the compiler says “ambiguous type variable”, this usually means you’ve applied a polymorphic function to a polymorphic argument. That’s what has happened here.

- `(. =)` has the polymorphic type `(KeyValue kv, ToJSON v) => Key ->`

v -> kv.

- Because we enabled `OverloadedStrings`, the literal `"Hello! \9835"` has the polymorphic type `IsString a => a`.

We (and the compiler) can infer that `v0`, the name the compiler assigned to represent the type of the expression `"Hello! \9835"`, must belong to both the `ToJSON` class and the `IsString` class. This certainly narrows down the possibilities about what `v0` might be, but not enough. The compiler always needs to come up with one specific type for every type variable.

Both messages also say: “Probable fix: use a type annotation to specify what `v0` should be.” This is indeed a good suggestion.

Exercise 25 – Encoding with class Here is one possible solution:

```
instance Encode Request where
    encode = encodeRequest
```

```
instance Encode Response where
    encode = encodeResponse
```

```
instance Encode Integer where
    encode = BSB.integerDec
```

```
instance Encode Int64 where
    encode = BSB.int64Dec
```

```
instance Encode T.Text where
    encode x = BSB.byteString (T.encodeUtf8 x)
```

```
instance Encode LT.Text where
    encode x = BSB.lazyByteString (LT.encodeUtf8 x)
```

```
instance Encode BS.ByteString where
    encode = BSB.byteString
```

```
instance Encode LBS.ByteString where
    encode = BSB.lazyByteString
```

```
instance Encode BSB.Builder where
    encode x = x
```

Some of these have a few other options that are equally valid. For example, UTF-8 is not the only character encoding that we may have chosen.

```
instance Encode T.Text where
    encode = BSB.byteString . T.encodeUtf16LE
```

And integers can be represented with number systems other than decimal. The `Data.ByteString.Builder` module showcases a wide assortment of ways to encode numbers. We will use the hexadecimal (base 16) number format in chapter 11.

```
instance Encode Int64 where
    encode = BSB.int64HexFixed
```

The `Encode` class affords some of the same convenience as `ToMarkup` and `ToJSON`. It reduces the number of function names to we have to keep track of. It also allows us to generalize other definitions. For example, con-

sider the `sendResponse` function from chapter 8:

```
sendResponse :: Socket -> Response -> IO ()
sendResponse s = SocketLBS.sendAll s .
    BSB.toLazyByteString . encodeResponse
```

If we replace `Response` with a type variable and use the polymorphic `encode` function instead of `encodeResponse`, then this function is no longer response-specific – it can now be used to send requests as well.

```
send :: Encode a => Socket -> a -> IO ()
send s = SocketLBS.sendAll s . BSB.toLazyByteString . encode
```

Difficulties with typeclasses arise when there is more than one reasonable way to define an instance for a type, such as the questions about text and integers discussed above. This is likely why the `bytestring` library does not offer a class-based approach to string building.

We could keep the advantages and sidestep the problems by more tightly focusing the purpose of the class:

- Add comments clarifying that the `Encode` class is for types that represent components of the HTTP grammar;
- Keep only the HTTP-relevant instances and discard the ambiguous text and integer instances.

Exercise 26 – Interleaving Since we want the two STM actions to run in two separate transactions, we will need two applications of the `atomically` function: one for `readTVar` and one for `writeTVar`. The monadic context for the `do` block is now `IO`, not `STM`.

```
incrementNotAtomic hitCounter = do
  oldCount <- atomically (readTVar hitCounter)
  let newCount = oldCount + 1
  atomically (writeTVar hitCounter newCount)
  return newCount
```

The first test should show 10,000.

```
λ> testIncrement (\x -> atomically (increment x))
10000
```

The second test should show some number less than 10,000, due to the problems with interleaving we discussed in the chapter. If you try this test repeatedly, you may get a different result each time.

```
λ> testIncrement incrementNotAtomic
2000
```

```
λ> testIncrement incrementNotAtomic
3976
```

Exercise 27 – Times gone by The TVar should store the UTCTime of the previous request.

There are a few twists here that differ from the example in the chapter.

```

timingServer = do
  timeVar <- atomically $
    newTVar @(Maybe Time.UTCTime) Nothing           -- 1
  serve @IO HostAny "8000" \(s, _) -> do
    now <- Time.getCurrentTime                       -- 2
    diff <- atomically $ updateTime timeVar now
    sendResponse s (textOk (show diff))

```

1. Whereas hit counter started at zero, there is no initial `UTCTime` to store in the `TVar` when there have been no requests yet. So its type actually has to be `TVar (Maybe UTCTime)`, so that it can be initialized as `Nothing`.
2. In addition to the previous time, we also need to know the current time, which comes from an IO action. The result from `getCurrentTime` will be used as the parameter to the `updateTime` function (given below) which creates the STM action.

```

updateTime ::
  TVar (Maybe Time.UTCTime) -- The last access time
-> Time.UTCTime             -- The current time
-> STM (Maybe Time.NominalDiffTime)
updateTime timeVar now = do
  previousTimeMaybe <- readTVar timeVar
  writeTVar timeVar (Just now)
  return $ case previousTimeMaybe of
    Nothing -> Nothing           -- 3
    Just past -> Just (Time.diffUTCTime now past) -- 4

```

`updateTime` sets the `TVar` to the present time and returns how long it has been since the last request. Like `increment`, it contains a read followed by a write.

Whether we return a `NominalDiffTime` or `Nothing` depends on what the `TVar` previously held.

3. For the first request, there is no previous time, so there is no elapsed time, and the result is `Nothing`.
4. For all subsequent requests, there is a `Just` value in the `TVar`, and we can calculate the amount of time that has passed.

A savvy reader may recognize the last few lines as `Maybe`'s `Functor`, and we might choose to express this more concisely using `fmap`:

```
updateTime timeVar now = do
  previousTimeMaybe <- readTVar timeVar
  writeTVar timeVar (Just now)
  return (Time.diffUTCTime now <$> previousTimeMaybe)
```

Exercise 28 – Tripping at the finish line If you include too few line breaks at the end, you'll see this warning from `curl`: "transfer closed with outstanding read data remaining"

If you run `curl` with the `--verbose` flag, then if you've included too many line breaks at the end, you'll see: "Leftovers after chunking: ..."

Exercise 29 – Infinite response This was a somewhat open-ended exercise, so the details of the response are up to you. We wrote a server that counts upward from 1.

```
infiniteServer = serve @IO HostAny "8000" \(s, _) -> do
  sendBSB s (encodeStatusLine (status ok))
  sendBSB s (encodeHeaders [transferEncodingChunked])
  for_ [1..] \n -> do
    sendOneOfInfinity s n
    threadDelay 500000
```

```

sendOneOfInfinity :: Socket -> Natural -> IO ()
sendOneOfInfinity s n =
    Net.send s $ encodeChunk $ dataChunk $ ChunkData $
        A.showIntegralDecimal n <> A.lift [A.LineFeed]

```

We didn't bother to include the last-chunk and the trailer part, because the server would never get around to it anyway!

Exercise 30 – List and stream operations

1. In `printMenu`, `ListT.select` turns a list of meals into a stream of meals.
2. In `printMenu`, `liftIO` turns the `print` action into a stream.
3. In `printMenu`, `runListT` turns a stream into an I/O action.
4. In `haskellCafe`, the `do` context is `[]`.
5. In `printMenu`, `<|>` is used to concatenate strings.
6. In `printMenu`, the `do` context is `ListT IO`.

In `printMenu`, the type of the implicit `(>>=)` is:

```

ListT IO (String, String)
-> ((String, String) -> ListT IO ())
-> ListT IO ()

```

Exercise 31 – Repeat until

```

listUntilIO action isEnd =
    ListT.takeWhile (not . isEnd) (listForeverIO action)

```



```
listForeverIO :: IO a -> ListT IO a
listForeverIO action = do
    ListT.select @[] (repeat ())
    liftIO action
```

```
hChunks h (MaxChunkSize mcs) =
    listUntil (BS.hGetSome h mcs) BS.null
```

Exercise 32 – File copying

```
hCopy source destination = runListT @IO do
    chunk <- hChunks source (MaxChunkSize 1024)
    liftIO (BS.hPutStr destination chunk)
```

Exercise 33 – Copying to multiple destinations

```
hCopyMany source destinations = runListT @IO do
    chunk <- hChunks source (MaxChunkSize 1024) -- 1
    destination <- ListT.select @[] destinations -- 2
    liftIO $ BS.hPutStr destination chunk
```

1. The top level of `hCopyMany` is a loop over the chunks of input, just like `hCopy`.
2. For each chunk, we write to every file before moving on to the next chunk.

The order of the lines in the `do` block is significant. Suppose we were to rearrange the steps as follows:

```
hCopyMany source destinations = runListT @IO do
  destination <- ListT.select @[] destinations -- 2
  chunk <- hChunks source 1024 -- 1
  liftIO $ BS.hPutStr destination chunk
```

This version does not work; it only copies into the first destination. After the first file is written, the source handle is exhausted, and there is nothing remaining to copy to the other files.

Exercise 34 – Parsing parentheses

```
parenParser = do
  _ <- P.string (A.lift [A.LeftParenthesis])
  x <- P.takeWhile (/= A.lift A.RightParenthesis)
  _ <- P.string (A.lift [A.RightParenthesis])
  return x
```

Exercise 35 – To digit, maybe Instead of checking the `Word8` with a predicate and then converting `Word8` to `Digit`, use a `Maybe` function that checks and converts at the same time.

```
A.word8ToDigitMaybe :: Word8 -> Maybe Digit
```

Or its polymorphic variant:

```
A.toDigitMaybe :: DigitSuperset char => char -> Maybe Digit
```

Nothing means we should fail, and `Just` gives us the digit to return.

```

digitParser = do
  x <- P.anyWord8
  case (A.toDigitMaybe x) of
    Nothing -> fail "0-9 expected"
    Just d   -> return d

```

Exercise 36 – Better parse errors

```

httpVersionParser = do
  _ <- P.string [A.string|HTTP|]
  _ <|> fail "Should begin with HTTP"
  _ <- P.string (A.lift [A.Slash])
  _ <|> fail "HTTP should be followed by a slash"
  x <- digitParser <?> "First digit"
  _ <- P.string (A.lift [A.FullStop])
  _ <|> fail "First digit should be followed by a dot"
  y <- digitParser <?> "Second digit"
  return (HttpVersion x y)

```

Exercise 37 – Status line

```

statusLineParser :: Parser StatusLine
statusLineParser = do
  version <- httpVersionParser
  _ <- spaceParser
  statusCode <- statusCodeParser
  _ <- spaceParser
  reasonPhrase <- reasonPhraseParser
  _ <- lineEndParser
  return (StatusLine version statusCode reasonPhrase)

```

```

statusCodeParser = do
  d1 <- digitParser
  d2 <- digitParser
  d3 <- digitParser
  return (StatusCode d1 d2 d3)

```

```

reasonPhraseParser = do
  phrase <- P.takeWhile \x ->
    elem x (A.lift [A.HorizontalTab, A.Space] :: [Word8])
    || A.isVisible x || x >= 0x80
  return (ReasonPhrase phrase)

```

```

statusLineParseTest :: StatusLine -> Maybe String
statusLineParseTest x =
  case result of
    Left e -> Just e
    Right x' | x /= x' -> Just (show x')
    Right _ -> Nothing
  where
    bs = LBS.toStrict (BSB.toLazyByteString (encodeStatusLine x))
    result = P.parseOnly (statusLineParser <* P.endOfInput) bs

```

The `StatusLine` type, and various others, need `Eq` and `Show` instances.

```

data StatusLine =
  StatusLine HttpVersion StatusCode ReasonPhrase
  deriving (Eq, Show)

```

Exercise 38 – P.string Below we give two approaches. The first is explicitly recursive, reading and checking one character at a time. The second

reads all of the characters upfront, then compares the result with the expected string.

```
stringParser :: BS.ByteString -> Parser BS.ByteString
stringParser bs = case BS.uncons bs of
    Nothing -> return bs
    Just (x, bs') -> do
        y <- P.anyWord8
        guard (x == y)
        stringParser bs'
```

```
stringParser :: BS.ByteString -> Parser BS.ByteString
stringParser bs = do
    xs <- replicateM (BS.length bs) P.anyWord8
    when (BS.pack xs /= bs) (fail "string")
    return bs
```

Exercise 39 – Resource server X

```
resourceServerX = do
  dir <- getDataDir
  let resources = resourceMap dir
  let maxChunkSize = MaxChunkSize 1024
  unforkAsyncIO_ printLogEvent \log ->
    serve @IO HostAny "8000" \(s, _) -> do
      result <-
        serveResourceOnceX resources maxChunkSize s
      case result of
        Right () -> return ()
        Left (Error responseMaybe events) -> liftIO do
          traverse_ log events
          traverse_ (sendResponse s) responseMaybe
```

Make sure you use `log`, not `printLogEvent`.

Exercise 40 – Check the method and version First define an `Error` that represents what we should do when the protocol version is wrong. Below, we have chosen not to log any event, but you may if you wish.

```
badRequestVersion :: Error
badRequestVersion = Error response []
  where
    response = textResponse versionNotSupported [] message
    message = LT.pack "Sorry, I only support HTTP version 1.1"
```

The check should compare the two versions using `(==)`. Add an `Eq` instance to `HttpVersion` if you have not done so already. Produce the `Error` if the two are not equal.

```

requireVersionX :: HttpVersion -> HttpVersion -> Either Error ()
requireVersionX supportedVersion requestVersion =
    case (supportedVersion == requestVersion) of
        False -> Left badRequestVersion
        True -> Right ()

```

The checks can be inserted into `serveResourceOnceX` anywhere after reading the request line and before sending the response, but it makes the most sense to perform them immediately after reading the request line.

```

serveResourceOnceX resources maxChunkSize s =
    runResourceT @IO $ runExceptT @Error @ (ResourceT IO) do
        RequestLine method target version <- ExceptT $
            liftIO (readRequestLineX maxChunkSize s)
        ExceptT $ return (requireVersionX http_1_1 version)
        ExceptT $ return (requireMethodX
            [Method [A.string|GET|]] method)
        ...

```

Exercise 41 – A sinister request

```

sendSinisterRequest = runResourceT @IO do
    a <- liftIO (resolve "8000" "localhost")
    (_, s) <- openAndConnect a
    liftIO (Net.send s [A.string|GET /hello|])
    liftIO (repeatUntilNothing (Net.recv s 1024) BS.putStr)

```

Above we have used an example from the previous chapter as the erroneous message. Your choice may vary, but it must contain enough input to make `requestLineParser` fail. For example, if we were to send `[A.string|GET /hello|]`, the server would not send an error response; it

would just wait indefinitely for more input.