Learn Haskell

by building a blog generator

A project-oriented approach to learning Haskell

Gil Mizrahi

Table of content

About this book

In this book, we will implement a simple static blog generator in Haskell, converting documents written in our own custom markup language to HTML.

We will:

1. Implement a tiny HTML printer library
2. Define and parse our own custom markup language
3. Read files and glue things together
4. Add command line arguments parsing
5. Write tests and documentation

In each chapter of the book, we will focus on a particular task we wish to achieve, and throughout the chapter, learn just enough Haskell to complete the task.

Why should you read this book?

There are many Haskell tutorials, guides, and books out there. Why read this one?

Pros

There are probably more, but here are a few possible pros:

- It is **relatively short** - most Haskell books are hundreds of pages long. This book (when exported to PDF) is roughly 200 pages long.
- It is **project oriented**. Many Haskell books teach Haskell by teaching the underlying concepts and features in a neat progression. In this book, we **build a Haskell program** and learn Haskell on the way. This will be a pro to some and a con to others.
- It touches on **important topics** such as design patterns, testing, and documentation.
- It has **many exercises** as well as **solutions** to those exercises.
- It's **online**, which means corrections are easy to make.
- It's **free**.

Cons

There are probably more, but here are a few possible cons:

- It **may lack depth** - many, much longer Haskell tutorials are long because they go deeper into the nuts and bolts of each feature, and I tried to keep this book relatively short.
- It **may not cover as many features or techniques** as other tutorials - we try to cover features as they pop up in our implementation, but we will probably miss features that aren't as important for our tasks, while other resources may try to cover many different use cases.
- It **does not have a technical editor**, though it has seen quite a bit of editing.

Other learning resources

The haskell.org/documentation page lists many tutorials, books, guides, and courses.

Who am I?

I'm  gilmi.

Discussions

Do you want to discuss the book? Maybe ask a question? Try the book's official discussion board!

Hello, world!

In this chapter, we will create a simple HTML "hello world" program and use the Haskell toolchain to compile and run it.

> If you haven't installed a Haskell toolchain yet, visit haskell.org/downloads for instructions on how to download and install a Haskell toolchain.

A Haskell source file

A Haskell source file is composed of definitions.

The most common type of definition has the following form:

```
<name> = <expression>
```

Note that:

1. Names must start with a lowercase letter
2. We cannot use the same name more than once in a file

A source file containing a definition of the name `main` can be treated as an executable, and the expression `main` is bound to is the entry point to the program.

Let's create a new Haskell source file called `hello.hs` and write the following line there:

```
main = putStrLn "<html><body>Hello, world!</body></html>"
```

We've defined a new name, `main`, and bound it to the expression `putStrLn "<html><body>Hello, world!</body></html>"`.

the body of `main` means calling the function `putStrLn` with the string `"<html><body>Hello, world!</body></html>"` as input. `putStrLn` takes a single string as input and prints that string to the standard output.

Note: we don't need parenthesis to pass arguments to functions in Haskell.

Running this program will result in the following text printed on the screen:

```
<html><body>Hello, world!</body></html>
```

Note that we cannot just write `putStrLn "<html><body>Hello, world!</body></html>"` without the `main =` part, because it is not a definition. This is something that is allowed in languages such as Python and OCaml, but not in Haskell or, for example, C.

Compiling programs

To run this little program, we can compile it using the command line program `ghc`:

```
> ghc hello.hs
[1 of 1] Compiling Main             ( hello.hs, hello.o )
Linking hello ...
```

Invoking `ghc` with `hello.hs` will create the following artifact files:

1. `hello.o` - Object file
2. `hello.hi` - Haskell interface file
3. `hello` - A native executable file

And after the compilation, we can run the `hello` executable:

```
> ./hello
<html><body>Hello, world!</body></html>
```

Interpreting programs

Alternatively, we can skip the compilation and creation of artifact files phase and run the source file directly using the command line program `runghc`:

```
> runghc hello.hs
<html><body>Hello, world!</body></html>
```

We can also redirect the output of the program to a file and then open it in Firefox.

```
> runghc hello.hs > hello.html
> firefox hello.html
```

This command should open Firefox and display a web page with `Hello, world!` written in it.

I recommend using `runghc` with this tutorial. While compiling produces significantly faster programs, intepreting programs provides us with faster feedback while we are developing and making frequent changes.

More bindings

We can define the HTML string passed to `putStrLn` in a new name instead of passing it directly to `putStrLn`. Change the content of file `hello.hs` we defined above to:

```
main = putStrLn myhtml

myhtml = "<html><body>Hello, world!</body></html>"
```

Note: the order in which we declare the bindings does not matter.

Building an HTML printer library

In this part, we'll explore a few basic building blocks in Haskell, including functions, types, and modules, while building a small HTML printer library with which we will later construct HTML pages from our markup blog posts.

If you're not familiar with HTML and would like a quick tutorial before diving in, MDN's Getting started with HTML is a good overview of the subject.

Flexible HTML content (functions)

We'd like to be able to write different HTML pages without having to write the whole structure of HTML and body tags over and over again. We can do that with functions.

To define a function, we create a definition as we saw previously and add the argument names after the name and before the equals sign (`=`). So a function definition has the following form:

```
<name> <arg1> <arg2> ... <argN> = <expression>
```

The argument names will be available in scope on the right side of the equals sign (in the `<expression>`), and the function name will be `<name>` .

We'll define a function that takes a string, which is the content of the page, and wraps it in the relevant `html` and `body` tags by concatenating them before and after the content. We use the operator `<>` to concatenate two strings.

```
wrapHtml content = "<html><body>" <> content <> "</body></html>"
```

This function, `wrapHtml` , takes one argument named `content` and returns a string that prefixes `<html><body>` before the content and appends `</body></html>` after it. Note that it is common to use camelCase in Haskell for names.

Now we can adjust our `myhtml` definition from the previous chapter:

```
myhtml = wrapHtml "Hello, world!"
```

Again, notice that we don't need parenthesis when calling functions. Function calls have the form:

```
<name> <arg1> <arg2> ... <argN>
```

However, if we wanted to substitute `myhtml` with the expression `myhtml` is bound to in `main = putStrLn myhtml` , we would have to wrap the expression in parenthesis:

```
main = putStrLn (wrapHtml "Hello, world!")
```

If we accidentally write this instead:

```
main = putStrLn wrapHtml "Hello, world!"
```

we'll get an error from GHC stating that `putStrLn` is applied to two arguments, but it only takes one. This is because the above is of the form `<name> <arg1> <arg2>` in which, as we defined earlier, `<arg1>` and `<arg2>` are arguments to `<name>` .

Using parenthesis, we can group the expressions together in the correct order.

An aside about operator precedence and fixity

operators (like `<>`) are infix functions that take two arguments - one from each side.

When there are multiple operators in the same expression without parenthesis, the operator *fixity* (left or right) and *precedence* (a number between 0 and 10) determine which operator binds more tightly.

In our case, `<>` has *right* fixity, so Haskell adds an invisible parenthesis on the right side of `<>`. So, for example:

```
"<html><body>" <> content <> "</body></html>"
```

is viewed by Haskell as:

```
"<html><body>" <> (content <> "</body></html>")
```

For an example of precedence, in the expression `1 + 2 * 3`, the operator `+` has precedence 6, and the operator `*` has precedence 7, so we give precedence to `*` over `+`. Haskell will view this expression as:

```
1 + (2 * 3)
```

You might run into errors when mixing different operators with the *same precedence* but *different fixity*, because Haskell won't understand how to group these expressions. In that case, we can solve the problem by adding parenthesis explicitly.

Exercises:

1. Separate the functionality of `wrapHtml` into two functions:

    1. One that wraps content in `html` tag
    2. one that wraps content in a `body` tag

    Name the new functions `html_` and `body_`.

2. Change `myhtml` to use these two functions.

3. Add another two similar functions for the tags `<head>` and `<title>` and name them `head_` and `title_`.

4. Create a new function, `makeHtml`, which takes two strings as input:

    1. One string for the title
    2. One string for the body content

    And construct an HTML string using the functions implemented in the previous exercises.

    The output for:

    ```
    makeHtml "My page title" "My page content"
    ```

should be:

```
<html><head><title>My page title</title></head><body>My page content</body></html>
```

5. Use `makeHtml` in `myhtml` instead of using `html_` and `body_` directly

Indentation

You might ask how does Haskell know a definition is complete? The answer is: Haskell uses indentation to know when things should be grouped together.

Indentation in Haskell can be a bit tricky, but in general: code that is supposed to be part of some expression should be indented further than the beginning of that expression.

We know two definitions are separate because the second one is not indented further than the first one.

Indentation tips

1. Choose a specific amount of spaces for indentation (2 spaces, 4 spaces, etc.) and stick to it. Always use spaces over tabs.
2. Do not indent more than once at any given time.
3. When in doubt, drop the line as needed and indent once.

Here are a few examples:

```
main =
    putStrLn "Hello, world!"
```

or:

```
main =
    putStrLn
        (wrapHtml "Hello, world!")
```

Avoid the following styles, which use more than one indentation step, or completely disregard indentation steps:

```
main = putStrLn
        (wrapHtml "Hello, world!")
```

```
main = putStrLn
                (wrapHtml "Hello, world!")
```

Adding type signatures

Haskell is a **statically typed** programming language. That means that every expression has a type, and we check that the types are valid with regards to each other before running the program. If we discover that they are not valid, an error message will be printed, and the program will not run.

An example of a type error would be if we'd pass 3 arguments to a function that takes only 2, or pass a number instead of a string.

Haskell is also **type inferred**, so we don't *need* to specify the type of expressions - Haskell can *infer* from the context of the expression what its type should be, and that's what we have done until now. However, **specifying types is useful** - it adds a layer of documentation for you or others that will look at the code later, and it helps verify to some degree that what was intended (with the type signature) is what was written (with the expression). It is generally recommended to annotate all *top-level* definitions with type signatures.

We use a double-colon (`::`) to specify the type of names. We usually write it right above the definition of the name itself.

Here are a few examples of types we can write:

- `Int` - The type of integer numbers
- `String` - The type of strings
- `Bool` - The type of booleans
- `()` - The type of the expression `()`, also called unit
- `a -> b` - The type of a function from an expression of type `a` to an expression of type `b`
- `IO ()` - The type of an expression that represents an IO subroutine that returns `()`

Let's specify the type of `title_`:

```
title_ :: String -> String
```

We can see in the code that the type of `title_` is a function that takes a `String` and returns a `String`.

Let's also specify the type of `makeHtml`:

```
makeHtml :: String -> String -> String
```

Previously, we thought about `makeHtml` as a function that takes two strings and returns a string.

But actually, all functions in Haskell take **exactly one argument** as input and return **exactly one value** as output. It's just convenient to refer to functions like `makeHtml` as functions with multiple inputs.

In our case, `makeHtml` is a function that takes **one** string argument and returns a **function**. *The function it returns* takes a string argument as well and finally returns a string.

The magic here is that `->` is right-associative. This means that when we write:

```
makeHtml :: String -> String -> String
```

Haskell parses it as:

```
makeHtml :: String -> (String -> String)
```

Consequently, the expression `makeHtml "My title"` is also a function! One that takes a string (the content, the second argument of `makeHtml`) and returns the expected HTML string with "My title" in the title.

This is called **partial application**.

To illustrate, let's define `html_` and `body_` in a different way by defining a new function, `el`.

```
el :: String -> String -> String
el tag content =
  "<" <> tag <> ">" <> content <> "</" <> tag <> ">"
```

el is a function that takes a tag and content, and wraps the content with the tag.

We can now implement `html_` and `body_` by partially applying `el` and only provide the tag.

```
html_ :: String -> String
html_ = el "html"

body_ :: String -> String
body_ = el "body"
```

Note that we didn't need to add the argument on the left side of equals sign because Haskell functions are "**first class**" - they behave exactly like values of primitive types like `Int` or `String`. We can name a function like any other value, put it in data structures, pass it to functions, and so on!

The way Haskell treats names is very similar to copy-paste. Anywhere you see `html_` in the code, you can replace it with `el "html"`. They are the same (this is what the equals signs say, right? That the two sides are the same). This property of being able to *substitute* the two sides of the equals sign with one another is called **referential transparency**. And it is pretty unique to Haskell (and a few similar languages such as PureScript and Elm)! We'll talk more about referential transparency in a later chapter.

Anonymous/lambda functions

To further drive the point that Haskell functions are first class and all functions take exactly one argument, I'll mention that the syntax we've been using up until now to define function is just syntactic sugar! We can also define **anonymous functions** - functions without a name, anywhere we'd like. Anonymous functions are also known as **lambda functions** as a tribute to the formal mathematical system which is at the heart of all functional programming languages - the lambda calculus.

We can create an anonymous function anywhere we'd expect an expression, such as `"hello"`, using the following syntax:

```
\<argument> -> <expression>
```

This little `\` (which bears some resemblance to the lowercase Greek letter lambda 'λ') marks the head of the lambda function, and the arrow (`->`) marks the beginning of the function's body. We can even chain lambda functions, making them "multiple argument functions" by defining another

lambda in the body of another, like this:

```
three = (\num1 -> \num2 -> num1 + num2) 1 2
```

As before, we evaluate functions by substituting the function argument with the applied value. In the example above, we substitute `num1` with `1` and get `(\num2 -> 1 + num2) 2`. Then substitute `num2` with `2` and get `1 + 2`. We'll talk more about substitution later.

So, when we write:

```
el :: String -> String -> String
el tag content =
  "<" <> tag <> ">" <> content <> "</" <> tag <> ">"
```

Haskell actually translates this under the hood to:

```
el :: String -> (String -> String)
el = \tag -> \content ->
  "<" <> tag <> ">" <> content <> "</" <> tag <> ">"
```

Hopefully, this form makes it a bit clearer why Haskell functions always take one argument, even when we have syntactic sugar that might suggest otherwise.

I'll mention one more syntactic sugar for anonymous functions: We don't actually have to write multiple argument anonymous functions this way, we can write:

```
\<arg1> <arg2> ... <argN> -> <expression>
```

to save us some trouble. For example:

```
three = (\num1 num2 -> num1 + num2) 1 2
```

But it's worth remembering what they are under the hood.

We won't be needing anonymous/lambda functions at this point, but we'll discuss them later and see where they can be useful.

Exercises:

1. Add types for all of the functions we created until now
2. Change the implementation of the HTML functions we built to use `el` instead
3. Add a couple more functions for defining paragraphs and headings:
    1. `p_` which uses the tag `<p>` for paragraphs
    2. `h1_` which uses the tag `<h1>` for headings
4. Replace our `Hello, world!` string with richer content, use `h1_` and `p_`. We can append HTML strings created by `h1_` and `p_` using the append operator `<>`.

Bonus: rewrite a couple of functions using lambda functions, just for fun!

Embedded Domain-Specific Languages

Right off the bat, we run into a common pattern in Haskell: creating Embedded Domain-Specific Languages (EDSLs for short).

Domain-specific languages (DSLs) are specialized programming languages that are tailored to specific domains, in contrast to general-purpose languages, which try to work well in many domains.

A few examples of DSLs are:

- make - for defining build systems
- DOT - for defining graphs
- Sed - for defining text transformations
- CSS - for defining styling
- HTML - for defining web pages

An *embedded* domain-specific language is a little language that is embedded inside another programming language, making a program written in the EDSL a valid program in the language it was written in.

The little HTML library we've been writing can be considered an EDSL. It is used specifically for building web pages (by returning HTML strings), and is valid Haskell code!

In Haskell, we frequently create and use EDSLs to express domain-specific logic. We have EDSLs for concurrency, command-line options parsing, JSON and HTML, creating build systems, writing tests, and many more.

Specialized languages are useful because they can solve specific problems in a concise (and often safe) way, and by embedding, we get to use the full power of the host language for our domain logic, including syntax highlighting and various tools available for the language.

The drawback of embedding domain-specific languages is that we have to adhere the rules of the programming language we embed in, such as syntactic and semantic rules.

Some languages alleviate this drawback by providing meta-programming capabilities in the form of macros or other features to extend the language. And while Haskell does provide such capabilities as well, it is also expressive and concise enough that many EDSLs do not need them.

Instead, many Haskell EDSLs use a pattern called *the combinator pattern*: They define *primitives* and *combinators* - primitives are basic building blocks of the language, and combinators are functions that combine primitives into more complex structures.

In our HTML EDSL, our primitives are functions such as `html_` and `title_` that can be used to create a single HTML node, and we pass other constructed nodes as input to these functions, and combine them into a more complex structure with the append function `<>`.

There are still a few tricks we can use to make our HTML EDSL better:

1. We can use Haskell's type system to make sure we only construct *valid* HTML, so for example, we don't create a `<title>` node without a `<head>` node or have user content that can include unescaped special characters, and throw a type error when the user tries to do something invalid.

2. Our HTML EDSL can move to its own module so it can be reused in multiple modules

In the next few sections, we'll take a look at how to define our own types and how to work with modules to make it harder to make errors, and a little bit about linked lists in Haskell.

Safer HTML construction with types

In this section, we'll learn how to create our own distinguished types for HTML, and how they can help us avoid the invalid construction of HTML strings.

There are a few ways of defining new types in Haskell; in this section, we are going to meet two ways: `newtype` and `type`.

newtype

A `newtype` declaration is a way to define a new, distinct type for an existing set of values. This is useful when we want to reuse existing values but give them a different meaning and ensure we can't mix the two. For example, we can represent seconds, minutes, grams, and yens using integer values, but we don't want to mix grams and seconds accidentally.

In our case, we want to represent structured HTML using textual values, but distinguish them from everyday strings that are not valid HTML.

A `newtype` declaration looks like this:

```
newtype <type-name> = <constructor> <existing-type>
```

For example, in our case, we can define a distinct type for `Html` like this:

```
newtype Html = Html String
```

The first `Html`, to the left of the equals sign, lives in the *types* name space, meaning that you will only see that name to the right of a double-colon sign (`::`).

The second `Html` lives in the *expressions* (or terms/values) namespace, meaning that you will see it where you expect expressions (we'll touch where exactly that can be in a moment).

The two names, `<type-name>` and `<constructor>`, do not have to be the same, but they often are. And note that both have to start with a capital letter.

The right-hand side of the newtype declaration describes the shape of a value of that type. In our case, we expect a value of type `Html` to have the constructor `Html` and then an expression of type string, for example: `Html "hello"` or `Html ("hello " <> "world")`.

You can think of the constructor as a function that takes the argument and returns something of our new type:

```
Html :: String -> Html
```

Note: We cannot use an expression of type `Html` the same way we'd use a `String`. So `"hello " <> Html "world"` would fail at type checking.

This is useful when we want *encapsulation*. We can define and use existing representation and functions for our underlying type, but not mix them with other unrelated (to our domain) types. Similar as meters and feet can both be numbers, but we don't want to accidentally add feet to

meters without any conversion.

For now, let's create a couple of types for our use case. We want two separate types to represent:

1. A complete Html document
2. A type for html structures such as headings and paragraphs that can go inside the tag

We want them to be distinct because we don't want to mix them.

Using newtypes

To use the underlying type that the newtype wraps, we first need to extract it out of the type. We do this using pattern matching.

Pattern matching can be used in two ways, in case-expressions and in function definitions.

1. case expressions are kind of beefed up switch expressions and look like this:

   ```
   case <expression> of
     <pattern> -> <expression>
     ...
     <pattern> -> <expression>
   ```

   The `<expression>` is the thing we want to unpack, and the `pattern` is its concrete shape. For example, if we wanted to extract the `String` out of the type `Structure` we defined in the exercise above, we do:

   ```
   getStructureString :: Structure -> String
   getStructureString struct =
     case struct of
       Structure str -> str
   ```

   This way, we can extract the `String` out of `Structure` and return it.

   > In later chapters we'll introduce `data` declarations (which are kind of a struct + enum chimera), where we can define multiple constructors to a type. Then the multiple patterns of a case expression will make more sense.

2. Alternatively, when declaring a function, we can also use pattern matching on the arguments:

   ```
   func <pattern> = <expression>
   ```

   For example:

   ```
   getStructureString :: Structure -> String
   getStructureString (Structure str) = str
   ```

> Using the types we created, we can change the HTML functions we've defined before, namely `html_`, `body_`, `p_`, etc., to operate on these types instead of `String`s.
>
> But first, let's meet another operator that will make our code more concise.

One very cool thing about `newtype` is that wrapping and extracting expressions doesn't actually have a performance cost! The compiler knows how to remove any wrapping and extraction of the `newtype` constructor and use the underlying type.

The new type and the constructor we defined are only there to help us *distinguish* between the type we created and the underlying type when *we write our code*, they are not needed *when the code is running*.

`newtype`s provide us with type safety with no performance penalty!

Chaining functions

Another interesting and extremely common operator (which is a regular library function in Haskell) is `.` (pronounced compose). This operator was made to look like the composition operator you may know from math (`∘`).

Let's look at its type and implementation:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

Compose takes 3 arguments: two functions (named `f` and `g` here) and a third argument named `x`. It then passes the argument `x` to the second function `g` and calls the first function `f` with the result of `g x`.

Note that `g` takes as input something of the type `a` and returns something of the type `b`, and `f` takes something of the type `b` and returns something of the type `c`.

Another important thing to note is that types that start with a *lowercase letter* are **type variables**. Think of them as similar to regular variables. Just like `content` could be any string, like `"hello"` or `"world"`, a type variable can be any type: `Bool`, `String`, `String -> String`, etc. This ability is called *parametric polymorphism* (other languages often call this generics).

The catch is that type variables must match in a signature, so if for example, we write a function with the type signature `a -> a`, the input type and the return type **must** match, but it could be any type - we cannot know what it is. So the only way to implement a function with that signature is:

```
id :: a -> a
id x = x
```

`id`, short for the identity function, returns the exact value it received. If we tried any other way, for example, returning some made-up value like `"hello"`, or trying to use `x` as a value of a type we know, like writing `x + x`, the type checker will complain.

Also, remember that `->` is right-associative? This signature is equivalent to:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Doesn't it look like a function that takes two functions and returns a third function that is the composition of the two?

We can now use this operator to change our HTML functions. Let's start with one example: `p_`.

Before, we had:

```
p_ :: String -> String
p_ = el "p"
```

And now, we can write:

```
p_ :: String -> Structure
p_ = Structure . el "p"
```

The function `p_` will take an arbitrary `String`, which is the content of the paragraph we wish to create, wrap it in `<p>` and `</p>` tags, and then wrap it in the `Structure` constructor to produce the output type `Structure` (remember: newtype constructors can be used as functions!).

Let's take a deeper look at the types:

- `Structure :: String -> Structure`
- `el "p" :: String -> String`
- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- `Structure . el "p" :: String -> Structure`

Let's see why the expression `Structure . el "p"` type checks, and why its type is `String -> Structure`.

Type checking with pen and paper

If we want to figure out if and how exactly an expression type-checks, we can do that rather systematically. Let's look at an example where we try and type-check this expression:

```
p_ = Structure . el "p"
```

First, we write down the type of the outer-most function. In our case, this is the operator `.` which has the type:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

After that, we can try to **match** the type of the arguments we apply to this function with the type of the arguments from the type signature.

In this case, we try to apply two arguments to `.`:

1. `Structure :: String -> Structure`
2. `el "p" :: String -> String`

And luckily, `.` expects two arguments with the types:

1. `b -> c`
2. `a -> b`

> Note: Applying a function with more arguments than it expects is a type error.

Since the `.` operator takes at least the number of arguments we supply, we continue to the next phase of type-checking: matching the types of the inputs with the types of the expected inputs (from the type signature of the operator).

When we match two types, we check for *equivalence* between them. There are a few possible scenarios here:

1. When the two types are **concrete** (as opposed to type variables) and **simple**, like `Int` and `Bool`, we check if they are the same. If they are, they type check, and we continue. If they aren't, they don't type check, and we throw an error.
2. When the two types we match are more **complex** (for example, both are functions), we try to match their inputs and outputs (in the case of functions). If the inputs and outputs match, then the two types match.
3. There is a special case when one of the types is a **type variable** - in this case, we treat the matching process like an equation and write it down somewhere. The next time we see this type variable, we *replace it with its match in the equation*. Think about this like *assigning* a type *variable* with a *value*.

In our case, we want to match (or check the equivalence of) these types:

1. `String -> Structure` with `b -> c`
2. `String -> String` with `a -> b`

Let's do this one by one, starting with (1) - matching `String -> Structure` and `b -> c`:

1. Because the two types are complex, we check that they are both functions, match their inputs and outputs: `String` with `b`, and `Structure` with `c`.
2. Because `b` is a *type variable*, we mark down somewhere that `b` should be equivalent to `String`. We write `b ~ String` (we use `~` to denote equivalence).
3. We match `Structure` and `c`, same as before, we write down that `c ~ Structure`.

No problem so far; let's try matching `String -> String` with `a -> b`:

1. The two types are complex; we see that both are functions, so we match their inputs and outputs.
2. Matching `String` with `a` - we write down that `a ~ String`.
3. Matching `String` with `b` - we remember that we have already written about `b` - looking back, we see that we already noted that `b ~ String`. We need to replace `b` with the type that we wrote down before and check it against this type, so we match `String` with `String` which, fortunately, type-check because they are the same.

So far, so good. We've type-checked the expression and discovered the following equivalences about the type variables in it:

1. `a ~ String`
2. `b ~ String`

3. `c ~ Structure`

Now, when asking what is the type of the expression:

```
p_ = Structure . el "p"
```

We say that it is the type of `.` after *replacing* the type variables using the equations, we found and *removing* the inputs we applied to it, so we started with:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Then we replaced the type variables:

```
(.) :: (String -> Structure) -> (String -> String) -> (String -> Structure)
```

And removed the two arguments when we applied the function:

```
Structure . el "p" :: String -> Structure
```

And we got the type of expression!

Fortunately, Haskell can do this process for us. But when Haskell complains that our types fail to type-check, and we don't understand exactly why, going through this process can help us understand where the types do not match, and then we can figure out how to solve it.

Note: If we use a *parametrically polymorphic* function more than once, or use different functions that have similar type variable names, the type variables don't have to match in all instances simply because they share a name. Each instance has its own unique set of type variables. For example:

```
id :: a -> a
ord :: Char -> Int
chr :: Int -> Char

incrementChar :: Char -> Char
incrementChar c = chr (ord (id c) + id 1)
```

In the snippet above, we use `id` twice (for no good reason other than for demonstration purposes). The first `id` takes a `Char` as argument, and its `a` is equivalent to `Char`. The second `id` takes an `Int` as argument, and its *distinct* `a` is equivalent to `Int`.

This, unfortunately, only applies to functions defined at the top-level. If we'd define a local function to be passed as an argument to `incrementChar` with the same type signature as `id`, the types must match in all uses. So this code:

```
incrementChar :: (a -> a) -> Char -> Char
incrementChar func c = chr (ord (func c) + func 1)
```

Will not type check. Try it!

Appending Structure

Before, when we wanted to create richer HTML content and appended nodes to one another, we used the append (`<>`) operator. Since we are now not using `String` anymore, we need another way to do it.

While it is possible to overload `<>` using a feature in Haskell called type classes, we will instead create a new function and call it `append_` , and cover type classes later.

`append_` should take two `Structure` s, and return a third `Structure` , appending the inner `String` in the first `Structure` to the second and wrapping the result back in `Structure` .

Try implementing `append_` .

Converting back Html to String

After constructing a valid `Html` value, we want to be able to print it to the output so we can display it in our browser. For that, we need a function that takes an `Html` and converts it to a `String` , which we can then pass to `putStrLn` .

Exercise: Implement the `render` function.

---

type

Let's look at one more way to give new names to types.

A `type` definition looks really similar to a `newtype` definition - the only difference is that we reference the type name directly without a constructor:

```
type <type-name> = <existing-type>
```

For example, in our case, we can write:

```
type Title = String
```

`type` , in contrast with `newtype` , is just a type name alias. When we declare `Title` as a *type alias* of `String` , we mean that `Title` and `String` are interchangeable, and we can use one or the other whenever we want:

```
"hello" :: Title

"hello" :: String
```

Both are valid in this case.

We can sometimes use `type` s to give a bit more clarity to our code, but they are much less useful

than `newtype`s which allow us to *distinguish* two types with the same type representation.

The rest of the owl

Exercise: Try changing the code we wrote in previous chapters to use the new types we created.

> **Tips**
>
> We can combine `makeHtml` and `html_`, and remove `body_` `head_` and `title_` by calling `el` directly in `html_`, which can now have the type `Title -> Structure -> Html`. This will make our HTML EDSL less flexible but more compact.
>
> Alternatively, we could create `newtype`s for `HtmlHead` and `HtmlBody` and pass those to `html_`, and we might do that in later chapters, but I've chosen to keep the API a bit simple for now, we can always refactor later!

Are we safe yet?

We have made some progress - now we can't write `"Hello"` where we'd expect either a paragraph or a heading, but we can still write `Structure "hello"` and get something that isn't a paragraph or a heading. So while we made it harder for the user to make mistakes by accident, we haven't really been able to **enforce the invariants** we wanted to enforce in our library.

Next, we'll see how we can make expressions such as `Structure "hello"` illegal as well using *modules* and *smart constructors*.

Preventing incorrect use with modules

In this section, we will move the HTML generation library to its own module.

Modules

Each Haskell source file is a module. The module name should have the same name as the source file and start with a capital letter. Sub-directories should also be part of the name, and we use `.` to denote a sub-directory. We'll see that in the next section.

The only exception to the rule are entry points to the program - modules with the name 'Main' that define `main` in them. Their source file names could have any name they want.

A module declaration looks like this:

```
module <module-name>
  ( <export-list>
  )
  where
```

The export list can be omitted if you want to export everything defined in the module, but we don't. We will list exactly the functions and types we want to export. This will give us control over how people can use our tiny library.

We will create a new source file named `Html.hs` and add the following module declaration code at the top of the file:

```
module Html
  ( Html
  , Title
  , Structure
  , html_
  , p_
  , h1_
  , append_
  , render
  )
  where
```

Note that we do not export:

1. The constructors for our new types, only the types themselves. If we wanted to export the constructors as well, we would've written `Html(Html)` or `Html(..)`. This way the user cannot create their own `Structure` by writing `Structure "Hello"`.
2. Internal functions used by the library, such as `el` and `getStructureString`.

And we will also move the HTML related functions from our `hello.hs` file to this new `Html.hs` file:

```
newtype Html
  = Html String

newtype Structure
  = Structure String

type Title
  = String

html_ :: Title -> Structure -> Html
html_ title content =
  Html
    ( el "html"
      ( el "head" (el "title" title)
        <> el "body" (getStructureString content)
      )
    )

p_ :: String -> Structure
p_ = Structure . el "p"

h1_ :: String -> Structure
h1_ = Structure . el "h1"

el :: String -> String -> String
el tag content =
  "<" <> tag <> ">" <> content <> "</" <> tag <> ">"

append_ :: Structure -> Structure -> Structure
append_ c1 c2 =
  Structure (getStructureString c1 <> getStructureString c2)

getStructureString :: Structure -> String
getStructureString content =
  case content of
    Structure str -> str

render :: Html -> String
render html =
  case html of
    Html str -> str
```

Now, anyone importing our module (using the `import` statement below module declarations but above any other declaration), will only be able to import what we export.

Add the following code at the top of the `hello.hs` file:

```
import Html
```

The `hello.hs` file should now look like this:

```
-- hello.hs

import Html

main :: IO ()
main = putStrLn (render myhtml)

myhtml :: Html
myhtml =
  html_
    "My title"
    ( append_
      (h1_ "Heading")
      ( append_
        (p_ "Paragraph #1")
        (p_ "Paragraph #2")
      )
    )
```

And the `Html.hs` file should look like this:

```haskell
-- Html.hs

module Html
  ( Html
  , Title
  , Structure
  , html_
  , p_
  , h1_
  , append_
  , render
  )
  where

newtype Html
  = Html String

newtype Structure
  = Structure String

type Title
  = String

html_ :: Title -> Structure -> Html
html_ title content =
  Html
    ( el "html"
      ( el "head" (el "title" title)
        <> el "body" (getStructureString content)
      )
    )

p_ :: String -> Structure
p_ = Structure . el "p"

h1_ :: String -> Structure
h1_ = Structure . el "h1"

el :: String -> String -> String
el tag content =
  "<" <> tag <> ">" <> content <> "</" <> tag <> ">"

append_ :: Structure -> Structure -> Structure
append_ c1 c2 =
  Structure (getStructureString c1 <> getStructureString c2)

getStructureString :: Structure -> String
getStructureString content =
  case content of
    Structure str -> str

render :: Html -> String
render html =
  case html of
    Html str -> str
```

As an aside, you might have noticed that I've decided to suffix the functions used to construct HTML values with an underscore (`_`). This is mostly an aesthetic decision which, in my opinion, makes the EDSL easier to recognize, but it is also useful to avoid name clashes with functions defined in the Haskell standard library, such as `head` . I took this idea from a Haskell HTML library named `lucid` !

Escaping characters

Now that `Html` has its own source file and module, and creating HTML code can be done only via the functions we exported, we can also handle user input that may contain characters that may conflict with our meta language, HTML, such as `<` and `>`, which are used for creating HTML tags.

We can convert these characters into different strings that HTML can handle.

See Stack overflow question for a list of characters, we need to escape.

Let's create a new function called `escape`:

```
escape :: String -> String
escape =
  let
    escapeChar c =
      case c of
        '<' -> "&lt;"
        '>' -> "&gt;"
        '&' -> "&amp;"
        '"' -> "&quot;"
        '\'' -> "&#39;"
        _ -> [c]
  in
    concat . map escapeChar
```

In `escape` we see a few new things:

1. Let expressions: we can define local names using this syntax:

   ```
   let
     <name> = <expression>
   in
     <expression>
   ```

   This will make `<name>` available as a variable `in` the second `<expression>`.

2. Pattern matching with multiple patterns: we match on different characters and convert them to a string. Note that `_` is a "catch all" pattern that will always succeed.

3. Two new functions: `map` and `concat`; we'll talk about these in more in-depth

Linked lists briefly

Linked lists are very common data structures in Haskell, so common that they have their own special syntax:

1. The list types are denoted with brackets, and inside them is the type of the element. For example:
   - `[Int]` - a list of integers
   - `[Char]` - a list of characters
   - `[String]` - a list of strings
   - `[[String]]` - a list of a list of strings
   - `[a]` - a list of any single type (all elements must be of the same type)
2. An empty list is written like this: `[]`
3. Prepending an element to a list is done with the operator `:` (pronounced cons), which is right-associative (like `->`). For example: `1 : []`, or `1 : 2 : 3 : []`.
4. The above lists can also be written like `[1]` and `[1, 2, 3]`.

Also, Strings are linked lists of characters - String is defined as: `type String = [Char]`, so we can use them the same way we use lists.

> Do note, however, that linked lists, despite their convenience, are often not the right tool for the job. They are not particularly space efficient and are slow for appending, random access, and more. That also makes `String` a lot less efficient than what it could be. And I generally recommend using a different string type, `Text`, instead, which is available in an external package. We will talk about lists, `Text`, and other data structures in the future!

We can implement our own operations on lists by using pattern matching and recursion. And we'll touch on this subject later when talking about ADTs.

For now, we will use the various functions found in the Data.List module. Specifically, map and concat.

map

Using `map`, we can apply a function to each element in a list. Its type signature is:

```
map :: (a -> b) -> [a] -> [b]
```

For example:

```
map not [False, True, False] == [True, False, True]
```

Or as can be seen in our `escape` function, this can help us escape each character:

```
map escapeChar ['<','h','1','>'] == ["&lt;","h","1","&gt;"]
```

However, note that the `escapeChar` has the type `Char -> String`, so the result type of `map escapeChar ['<','h','1','>']` is `[String]`, and what we really want is a `String` and not `[String]`.

This is where `concat` enters the picture to help us flatten the list.

concat

`concat` has the type:

```
concat :: [[a]] -> [a]
```

It flattens a list of list of something into a list of something. In our case it will flatten `[String]` into `String`, remember that `String` is a **type alias** for `[Char]`, so we actually have `[[Char]] -> [Char]`.

GHCi

One way we can quickly see our code in action is by using the interactive development environment **GHCi**. Running `ghci` will open an interactive prompt where Haskell expressions can be written and evaluated. This is called a "Read-Evaluate-Print Loop" (for short - REPL).

For example:

```
ghci> 1 + 1
2
ghci> putStrLn "Hello, world!"
Hello, world!
```

We can define new names:

```
ghci> double x = x + x
ghci> double 2
4
```

We can write multi-line code by surrounding it with `:{` and `:}`:

```
ghci> :{
| escape :: String -> String
| escape =
|   let
|     escapeChar c =
|       case c of
|         '<' -> "&lt;"
|         '>' -> "&gt;"
|         '&' -> "&amp;"
|         '"' -> "&quot;"
|         '\'' -> "&#39;"
|         _ -> [c]
|   in
|     concat . map escapeChar
| :}

ghci> escape "<html>"
"&lt;html&gt;"
```

We can import Haskell source files using the `:load` command (`:l` for short):

```
ghci> :load Html.hs
[1 of 1] Compiling Html    ( Html.hs, interpreted )
Ok, one module loaded.
ghci> render (html_ "<title>" (p_ "<body>"))
"<html><head><title>&lt;title&gt;</title></head><body><p>&lt;body&gt;</p></body>
</html>"
```

As well as import library modules:

```
ghci> import Data.Bits
ghci> shiftL 32 1
64
ghci> clearBit 33 0
32
```

We can even ask the type of an expression using the `:type` command (`:t` for short):

```
λ> :type escape
escape :: String -> String
```

To exit `ghci`, use the `:quit` command (or `:q` for short)

```
ghci> :quit
Leaving GHCi.
```

GHCi is a very useful tool for quick experiments and exploration. We've seen a couple of examples of that above - passing the string `"<html>"` to our `escape` function returns the string `"<html>"`, which can be rendered by a browser as `<html>` instead of an HTML tag.

If you are having a hard time figuring out what a particular function does, consider testing it in GHCi - pass it different inputs and see if it matches your expectations. Concrete examples of running code can aid a lot in understanding it!

> If you'd like to learn more about GHCi, you can find a more thorough introduction in the GHC user guide.

Escaping

The user of our library can currently only supply strings in a few places:

1. Page title
2. Paragraphs
3. Headings

We can apply our escape function at these places before doing anything else with it. That way, all HTML constructions are safe.

Try adding the escaping function in those places, and then try to construct an invalid HTML in `hello.hs` to see if this works or not!

Now we can use our tiny HTML library safely. But what if the user wants to use our library with a

valid use case we didn't think about, for example, adding unordered lists? We are completely blocking them from extending our library. We'll talk about this next.

Exposing internal functionality (Internal modules)

We have now built a very small but convenient and safe way to write HTML code in Haskell. This is something that we could (potentially) publish as a *library* and share with the world by uploading it to a package repository such as Hackage. Users interested in our library could use a package manager to include it in their project and build their own HTML pages.

It is important to note that users are building their projects against the API that we expose to them, and the package manager doesn't generally provide access to the source code, so they can't, for example, modify the `Html` module (that we expose) in their project directly without jumping through some hoops.

Because we wanted our `Html` EDSL to be safe, we **hid the internal implementation from the user**, and the only way to interact with the library is via the API we provide.

This provides the safety we wanted to provide, but in this case, it also *blocks* the user from extending our library *in their own project* with things we haven't implemented yet, such as lists or code blocks.

When a user runs into trouble with a library (such as missing features) the best course of action usually is to open an issue in the repository or submit a pull request, but sometimes the user needs things to work *now*.

We admit that we are not perfect and can't think of all use cases for our library. Sometimes the restrictions we add are too great and may limit the usage of advanced users who know how things work under the hood and need certain functionality to use our library.

Internal modules

For that, we can expose internal modules to provide some flexibility for advanced users. Internal modules are not a language concept but rather a (fairly common) design pattern (or idiom) in Haskell.

Internal modules are simply modules named `<something>.Internal`, which export all of the functionality and implementation details in that module.

Instead of writing the implementation in (for example) the `Html` module, we write it in the `Html.Internal` module, which will export everything. Then we will import that module in the `Html` module and write an explicit export list to only export the API we'd like to export (as before).

`Internal` modules are considered unstable and risky to use by convention. If you end up using one yourself when using an external Haskell library, make sure to open a ticket in the library's repository after the storm has passed!

Let's make the changes

We will create a new directory named `Html` and inside it a new file named `Internal.hs`. The name of this module should be `Html.Internal`.

This module will contain all of the code we previously had in the `Html` module, but **we will change the module declaration in `Html.Internal` and *omit* the export list**:

```
-- Html/Internal.hs

module Html.Internal where

...
```

And now in `Html.hs`, we will remove the code that we moved to `Html/Internal.hs` and in its stead we'll import the internal module:

```
-- Html.hs

module Html
  ( Html
  , Title
  , Structure
  , html_
  , p_
  , h1_
  , append_
  , render
  )
  where

import Html.Internal
```

Now, users of our library can still import `Html` and safely use our library, but if they run into trouble and have a dire need to implement unordered lists to work with our library, they could always work with `Html.Internal` instead.

Summary

For our particular project, `Internal` modules aren't necessary. Because our project and the source code for the HTML EDSL are part of the same project, and we have access to the `Html` module directly, we can always go and edit it if we want (and we are going to do that throughout the book).

However, if we were planning to release our HTML EDSL as a *library* for other developers to use, it would be nice to also expose the internal implementation as an `Internal` module. Just so we can save some trouble for potential users!

In a later chapter, we will see how to create a package from our source code.

Exercises

We need a few more features for our HTML library to be useful for our blog software. Add the following features to our `Html.Internal` module and expose them from `Html`.

1. Unordered lists

These lists have the form:

```
<ul>
  <li>item 1</li>
  <li>item 2</li>
  <li>...</li>
</ul>
```

We want in our library a new function:

```
ul_ :: [Structure] -> Structure
```

So that users can write this:

```
ul_
  [ p_ "item 1"
  , p_ "item 2"
  , p_ "item 3"
  ]
```

and get this:

```
<ul>
  <li><p>item 1</p></li>
  <li><p>item 2</p></li>
  <li><p>item 3</p></li>
</ul>
```

2. Ordered lists

Very similar to unordered lists, but instead of `<ul>` we use `<ol>`

3. Code blocks

Very similar to `<p>`, but use the `<pre>` tag. Call this function `code_`.

Summary

In this chapter, we built a very minimal HTML EDSL. We will later use this library to convert our custom markup formatted text to HTML.

We've also learned about:

- Defining and using functions
- Types and type signatures
- Embedded domain-specific languages
- Chaining functions using the `.` operator
- Preventing incorrect use with `newtype`s
- Defining modules and the `Internal` module pattern
- Encapsulation using `newtype`s and modules

Here's our complete program up to this point:

```
-- hello.hs

import Html

main :: IO ()
main = putStrLn (render myhtml)

myhtml :: Html
myhtml =
  html_
    "My title"
    ( append_
      (h1_ "Heading")
      ( append_
        (p_ "Paragraph #1")
        (p_ "Paragraph #2")
      )
    )
```

```
-- Html.hs

module Html
  ( Html
  , Title
  , Structure
  , html_
  , h1_
  , p_
  , ul_
  , ol_
  , code_
  , append_
  , render
  )
  where

import Html.Internal
```

```haskell
-- Html/Internal.hs

module Html.Internal where

-- * Types

newtype Html
  = Html String

newtype Structure
  = Structure String

type Title
  = String

-- * EDSL

html_ :: Title -> Structure -> Html
html_ title content =
  Html
    ( el "html"
      ( el "head" (el "title" (escape title))
        <> el "body" (getStructureString content)
      )
    )

p_ :: String -> Structure
p_ = Structure . el "p" . escape

h1_ :: String -> Structure
h1_ = Structure . el "h1" . escape

ul_ :: [Structure] -> Structure
ul_ =
  Structure . el "ul" . concat . map (el "li" . getStructureString)

ol_ :: [Structure] -> Structure
ol_ =
  Structure . el "ol" . concat . map (el "li" . getStructureString)

code_ :: String -> Structure
code_ = Structure . el "pre" . escape

append_ :: Structure -> Structure -> Structure
append_ c1 c2 =
  Structure (getStructureString c1 <> getStructureString c2)

-- * Render

render :: Html -> String
render html =
  case html of
    Html str -> str

-- * Utilities

el :: String -> String -> String
el tag content =
  "<" <> tag <> ">" <> content <> "</" <> tag <> ">"

getStructureString :: Structure -> String
getStructureString content =
  case content of
    Structure str -> str
```

```
escape :: String -> String
escape =
  let
    escapeChar c =
      case c of
        '<' -> "&lt;"
        '>' -> "&gt;"
        '&' -> "&amp;"
        '"' -> "&quot;"
        '\'' -> "&#39;"
        _ -> [c]
  in
    concat . map escapeChar
```

Custom markup language

In this chapter, we will define our own simple markup language and parse documents written in this language into Haskell data structures.

Our markup language will contain the following features:

- Headings: prefix by a number of `*` characters
- Paragraphs: a group of lines without empty lines in between
- Unordered lists: a group of lines, each prefixed with `-`
- Ordered lists: a group of lines, each prefixed with `#`
- Code blocks: a group of lines, each prefixed with `>`

Here's a sample document:

```
* Compiling programs with ghc

Running ghc invokes the Glasgow Haskell Compiler (GHC),
and can be used to compile Haskell modules and programs into native
executables and libraries.

Create a new Haskell source file named hello.hs, and write
the following code in it:

> main = putStrLn "Hello, Haskell!"

Now, we can compile the program by invoking ghc with the file name:

> ➜ ghc hello.hs
> [1 of 1] Compiling Main             ( hello.hs, hello.o )
> Linking hello ...

GHC created the following files:

- hello.hi - Haskell interface file
- hello.o - Object file, the output of the compiler before linking
- hello (or hello.exe on Microsoft Windows) - A native runnable executable.

GHC will produce an executable when the source file satisfies both conditions:

# Defines the main function in the source file
# Defines the module name to be Main or does not have a module declaration

Otherwise, it will only produce the .o and .hi files.
```

which we will eventually convert into this (modulo formatting) HTML:

```
<h1>Compiling programs with ghc</h1>

<p>Running ghc invokes the Glasgow Haskell Compiler (GHC),
and can be used to compile Haskell modules and programs into native
executables and libraries.
</p>

<p>Create a new Haskell source file named hello.hs, and write
the following code in it:
</p>

<pre>main = putStrLn "Hello, Haskell!"
</pre>

<p>Now, we can compile the program by invoking ghc with the file name:</p>

<pre>
→ ghc hello.hs
[1 of 1] Compiling Main             ( hello.hs, hello.o )
Linking hello ...
</pre>

<p>GHC created the following files:
</p>

<ul>
  <li>hello.hi - Haskell interface file</li>
  <li>hello.o - Object file, the output of the compiler before linking</li>
  <li>hello (or hello.exe on Microsoft Windows) - A native runnable executable.</li>
</ul>

<p>GHC will produce an executable when the source file satisfies both conditions:
</p>

<ol>
  <li>Defines the main function in the source file</li>
  <li>Defines the module name to be Main, or does not have a module declaration</li>
</ol>

<p>Otherwise, it will only produce the .o and .hi files.
</p>
```

Representing the markup language as a Haskell data type

One of the clear differentiators between Haskell (also other ML-family of languages) and most mainstream languages is the ability to represent data precisely and succinctly.

So how do we represent our markup language using Haskell?

Previously, in our HTML builder library, we used `newtype`s to differentiate between HTML documents, structures, and titles, but we didn't really need to differentiate between different kinds of structures, such as paragraphs and headings, not without parsing the data, at least.

In this case, we have a list of structures, and each structure could be one of a few specific options (a paragraph, a heading, a list, etc.), and we want to be able to know which structure is which so we can easily convert it into the equivalent HTML representation.

For that, we have `data` definitions. Using `data` we can create custom types by grouping multiple types together and having alternative structures. Think of them as a combination of both structs and enums.

`data` declarations look like this:

```
data <Type-name> <type-args>
  = <Data-constructor1> <types>
  | <Data-constructor2> <types>
  | ...
```

It looks really similar to `newtype`, but there are two important differences:

1. In the `<types>` part, we can write many types (Like `Int`, `String`, or `Bool`). For `newtype`s, we can only write one.
2. We can have alternative structures using `|`, `newtype`s have no alternatives.

This is because `newtype` is used to provide a type-safe **alias**, and `data` is used to build a new **composite** type that can potentially have *alternatives*.

Let's see a few examples of data types:

1. Bool

   ```
   data Bool
     = True
     | False
   ```

   We created a new data type named `Bool` with the possible values `True` or `False`. In this case, we only have *constructor* alternatives, and none of the constructors carry additional values. This is similar to enums in other languages.

2. Person

```
data Person
  = Person String Int -- where the first is the name and the second is
                      -- the age
```

We created a new data type named `Person`. Values of the type `Person` look like this:

```
Person <some-string> <some-int>
```

For example:

```
Person "Gil" 32
```

In this case, we create a *composite* of multiple types without alternatives. This is similar to structs in other languages, but structs give each field a name, and here we distinguish them by position.

Alternatively, Haskell has *syntactic sugar* for naming fields called **records**. The above definition can also be written like this:

```
data Person
  = Person
    { name :: String
    , age :: Int
    }
```

Values of this type can be written exactly as before,

```
Person "Gil" 32
```

Or with this syntax:

```
Person { name = "Gil", age = 32 }
```

Haskell will also generate functions that can be used to extract the fields from the composite type:

```
name :: Person -> String
age :: Person -> Int
```

Which can be used like this:

```
ghci> age (Person { name = "Gil", age = 32 })
32
```

We even have a special syntax for updating specific fields in a record. Of course, we do not update records in place - we generate a new value instead.

```
ghci> gil = Person { name = "Gil", age = 32 }
ghci> age (gil { age = 33 })
33
ghci> age gil
32
```

Unfortunately, having specialized functions for each field also means that if we defined a different data type with the field `age`, the functions which GHC needs to generate will clash.

The easiest way to solve this is to give fields unique names, for example by adding a prefix:

```
data Person
  = Person
    { pName :: String
    , pAge :: Int
    }
```

Another way is by using extensions to the Haskell language, which we will cover in later chapters.

3. Tuple

```
data Tuple a b
  = Tuple a b
```

This is pretty similar to `Person`, but we can plug any type we want for this definition. For example:

```
Tuple "Clicked" True :: Tuple String Bool

Tuple 'a' 'z' :: Tuple Char Char
```

This type has special syntax in Haskell:

```
("Clicked", True) :: (String, Bool)

('a', 'z') :: (Char, Char)
```

This `Tuple` definition is polymorphic; we define the structure but are able to plug different types into the structure to get concrete types. You can think of `Tuple` as a *template* for a data type waiting to be filled or as a **function** waiting for types as input in order to return a data type. We can even take a look at the "type" signature of `Tuple` in `ghci` using the `:kind` command.

```
ghci> data Tuple a b = Tuple a b
ghci> :kind Tuple
Tuple :: * -> * -> *
```

Quick detour: Kinds

The `:kind` command is called as such because the "type" of a type is called a **kind**. Kinds can be one of two things, either a `*`, which means a saturated (or concrete) type, such as `Int` or `Person`, or an `->` of two kinds, which is, as you might have guessed, a type function, taking kind and returning a kind.

Note that only types that have the kind `*` can have values. So, for example, while `Tuple Int` is a valid Haskell concept that has the *kind* `* -> *`, and we can write code that will work "generically" for all types that have a certain kind (e.g. `* -> *`), we cannot construct a value that has the kind `* -> *`. All values have types and all types that have values have the kind `*`.

We will talk more about kinds later; let's focus on types for now!

4. Either

```
data Either a b
  = Left a
  | Right b
```

Similar to Tuple, but instead of having only one constructor, we have two. This means that we can choose which side we want. Here are a couple of values of type `Either String Int`:

```
Left "Hello"

Right 17
```

This type is useful for modeling errors. Either we succeeded and got what we wanted (The `Right` constructor with the value), or we didn't and got an error instead (The `Left` constructor with a string or a custom error type).

In our program, we use `data` types to model the different kinds of content types in our markup language. We tag each structure using the data constructor and provide the rest of the information (the paragraph text, the list items, etc.) in the `<types>` section of the data declaration for each constructor:

```
type Document
  = [Structure]

data Structure
  = Heading Natural String
  | Paragraph String
  | UnorderedList [String]
  | OrderedList [String]
  | CodeBlock [String]
```

Note: `Natural` is defined in the `base` package but not exported from `Prelude`. Find out which module to import `Natural` by using Hoogle.

Exercises

Represent the following markup documents as values of `Document`:

1.
```
Hello, world!
```

2.
```
* Welcome

To this tutorial about Haskell.
```

3.
```
Remember that multiple lines with no separation
are grouped together into a single paragraph
but list items remain separate.

# Item 1 of a list
# Item 2 of the same list
```

4.
```
* Compiling programs with ghc

Running ghc invokes the Glasgow Haskell Compiler (GHC),
and can be used to compile Haskell modules and programs into native
executables and libraries.

Create a new Haskell source file named hello.hs, and write
the following code in it:

> main = putStrLn "Hello, Haskell!"

Now, we can compile the program by invoking ghc with the file name:

> ➜ ghc hello.hs
> [1 of 1] Compiling Main             ( hello.hs, hello.o )
> Linking hello ...

GHC created the following files:

- hello.hi - Haskell interface file
- hello.o - Object file, the output of the compiler before linking
- hello (or hello.exe on Microsoft Windows) - A native runnable executable.

GHC will produce an executable when the source file satisfies both conditions:

# Defines the main function in the source file
# Defines the module name to be Main or does not have a module declaration

Otherwise, it will only produce the .o and .hi files.
```

Translating directly?

You might ask, "Why do we even need to represent the markup as a type? Why don't we convert it into HTML as soon as we parse it instead?". That's a good question and a valid strategy. The reason we first represent it as a Haskell type is for flexibility and modularity.

If the parsing code is coupled with HTML generation, we lose the ability to pre-process the markup document. For example, we might want to take only a small part of the document (for a summary) and present it, or create a table of content from headings. Or maybe we'd like to add other targets and not just HTML - maybe markdown format or a GUI reader?

Parsing to an "abstract data type" (ADT) representation (one that does not contain the details of the language, for example, '#' for ordered lists) gives us the freedom to do so much more than just conversion to HTML that it's usually worth it, in my opinion, unless you really need to optimize the process.

Parsing markup part 01 (Recursion)

Let's have a look at how to parse a multi-lined string of markup text written by a user and convert it to the `Document` type we defined in the previous chapter.

Our strategy is to take the string of markup text and:

1. Split it into a list where each element represents a separate line, and
2. Go over the list line by line and process it, remembering information from previous lines if necessary

So the first thing we want to do is to process the string line by line. We can do that by converting the string to a list of string. Fortunately the Haskell `Prelude` module from the Haskell standard library `base` exposes the function `lines` that does exactly what we want. The `Prelude` module is exposed in every Haskell file by default, so we don't need to import it.

For the line processing part, let's start by ignoring all of the markup syntax and just group lines together into paragraphs (paragraphs are separated by an empty line), and iteratively add new features later in the chapter.

A common solution in imperative programs would be to iterate over the lines using some *loop* construct and accumulate lines that should be grouped together into some intermediate mutable variable. When we reach an empty line, we insert the content of that variable into another mutable variable that accumulates the results.

Our approach in Haskell isn't so different, except that we do not use loops or mutable variables. Instead, we use **recursion**.

Recursion and accumulating information

Instead of loops, in Haskell, we use recursion to model iteration.

Consider the following contrived example: let's say that we want to write an algorithm for adding two natural numbers together, and we don't have a standard operation to do that (+), but we do have two operations we could use on each number: `increment` and `decrement`.

A solution we could come up with is to slowly "pass" one number to the other number iteratively by incrementing one and decrementing the other. And we do that until the number we decrement reaches 0.

For example, for `3` and `2`:

- We start with `3` and `2`, and we increment `3` and decrement `2`
- In the next step, we now have `4` and `1`, we increment `4` and decrement `1`
- In the next step, we now have `5` and `0`, since the second number is `0` we declare `5` as the result.

This can be written imperatively using a loop:

```
function add(n, m) {
  while (m /= 0) {
    n = increment(n);
    m = decrement(m);
  }
  return n;
}
```

We can write the same algorithm in Haskell without mutation using recursion:

```
add n m =
  if m /= 0
    then add (increment n) (decrement m)
    else n
```

In Haskell, to *emulate iteration with a mutable state*, we call the function again with the values we want the variables to have in the next iteration.

Evaluation of recursion

Recursion commonly has a bad reputation for being slow and possibly unsafe compared to loops. This is because, in imperative languages, calling a function often requires creating a new call stack.

However, functional languages (and Haskell in particular) play by different rules and implement a feature called tail call elimination - when the result of a function call is the result of the function (this is called tail position), we can just drop the current stack frame and then allocate one for the function we call, so we don't require `N` stack frames for `N` iterations.

This is, of course, only one way to do tail call elimination and other strategies exist, such as translating code like our recursive `add` above to the iteration version.

Laziness

Haskell plays by slightly different rules because it uses a *lazy evaluation strategy* instead of the much more common strict evaluation strategy. An *evaluation strategy* refers to "when do we evaluate a computation". In a strict language, the answer is simple: *we evaluate the arguments of a function before entering a function*.

So, for example, the evaluation of `add (increment 3) (decrement 2)` using strict evaluation will look like this:

1. Evaluate `increment 3` to `4`
2. Evaluate `decrement 2` to `1`
3. Evaluate `add 4 1`

Or, Alternatively (depending on the language), we reverse (1) and (2) and evaluate the arguments from right-to-left instead of left-to-right.

On the other hand, with lazy evaluation, we *only evaluate computation when we need it*, which is when it is part of a computation that will have some effect on the outside world, for example, when writing a computation to standard output or sending it over the network.

So unless this computation is required, it won't be evaluated. For example:

```
main =
  if add (increment 2) (decrement 3) == 5
    then putStrLn "Yes."
    else putStrLn "No."
```

In the case above, we need the result of `add (increment 2) (decrement 3)` in order to know which message to write, so it will be evaluated. But:

```
main =
  let
    five = add (increment 2) (decrement 3)
  in
    putStrLn "Not required"
```

In the case above, we don't actually need `five`, so we don't evaluate it!

But then, if we know we need `add (increment 2) (decrement 3)`, do we use strict evaluation now? The answer is no - because we might not need to evaluate the arguments to complete the computation. For example, in this case:

```
const a b = a

main =
  if const (increment 2) (decrement 3) == 3
    then putStrLn "Yes."
    else putStrLn "No."
```

`const` ignores the second argument and returns the first, so we don't actually need to calculate `decrement 3` to provide an answer to the computation and in turn output an answer to the screen.

With the lazy evaluation strategy, we will evaluate expressions when we need to (when they are required in order to do something for the user), and we evaluate from the outside in - first we enter functions, and then we evaluate the arguments when we need to (usually when the thing we want to evaluate appears in some control flow such as the condition of an `if` expression or a pattern in pattern matching).

I've written a more in-depth blog post about how this works in Haskell: Substitution and Equational Reasoning.

Please read it and try to evaluate the following program by hand:

```haskell
import Prelude hiding (const) -- feel free to ignore this line

increment n = n + 1

decrement n = n - 1

const a b = a

add n m =
  if m /= 0
    then add (increment n) (decrement m)
    else n

main =
  if const (add 3 2) (decrement 3) == 5
    then putStrLn "Yes."
    else putStrLn "No."
```

Remember that evaluation always begins from `main`.

General recursion

In general, when trying to solve problems recursively, it is useful to think about the problem in three parts:

1. Finding the **base case** (the most simple cases - the ones we already know how to answer)
2. Figuring out how to **reduce** the problem to something simpler (so it gets closer to the base case)
3. **Mitigating the difference** between the reduced version and the solution we need to provide

The reduce and mitigate steps together are usually called the *recursive step*.

Let's take a look at another example problem: generating a list of a particular size with a specific value in place of every element.

In Haskell, this function would have the following signature:

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

Here are a few usage examples of `replicate`:

```
ghci> replicate 4 True
[True,True,True,True]
ghci> replicate 0 True
[]
ghci> replicate (-13) True
[]
```

How would we implement this function recursively? How would we describe it in the three steps above?

1. **Base case**: the cases we already know how to generate are the cases where the length of the list is zero (or less) - we just return an empty list.
2. **Reduce**: while we might not know how to generate a list of size `N` (where `N` is positive), if we knew the solution for `N-1`, we could:

3. **Mitigate**: Add another element to the solution for `N-1` using the `:` (cons) operator.

Exercise: Try to write this in Haskell!

Mutual recursion

When solving functions recursively, we usually call the same function again, but that doesn't have to be the case. It is possible to reduce our problem to something simpler that requires an answer from a different function. If, in turn, that function will (or another function in that call chain) call our function again; we have a **mutual recursive** solution.

For example, let's write two functions, one that checks whether a natural number is even or not, and one that checks whether a number is odd or not only by decrementing it.

```
even :: Int -> Bool

odd :: Int -> Bool
```

Let's start with `even`; how should we solve this recursively?

1. **Base case**: We know the answer for `0` - it is `True`.
2. **Reduction**: We might not know the answer for a general `N`, but we could check whether `N - 1` is odd,
3. **Mitigation**: if `N - 1` is odd, then `N` is even! if it isn't odd, then `N` isn't even.

What about `odd`?

1. **Base case**: We know the answer for `0` - it is `False`.
2. **Reduction**: We might not know the answer for a general `N`, but we could check whether `N - 1` is even,
3. **Mitigation**: if `N - 1` is even, then `N` is odd! if it isn't even, then `N` isn't odd.

Exercise: Try writing this in Haskell!

Partial functions

Because we didn't handle the negative numbers cases in the example above, our functions will loop forever when a negative value is passed as input. A function that does not return a result for some value (either by not terminating or by throwing an error) is called **a partial function** (because it only returns a result for a part of the possible inputs).

Partial functions are generally considered **bad practice** because they can have undesired behaviour at runtime (a runtime exception or an infinite loop), so we want to **avoid using** partial functions as well as **avoid writing** partial functions.

The best way to avoid writing partial functions is by covering all inputs! In the situation above, it is definitely possible to handle negative numbers as well, so we should do that! Or, instead, we could

require that our functions accept a `Natural` instead of an `Int`, and then the type system would've stopped us from using these functions with values we did not handle.

There are cases where we can't possibly cover all inputs; in these cases, it is important to re-examine the code and see if we could further restrict the inputs using types to mitigate these issues.

For example, the `head :: [a] -> a` function from `Prelude` promises to return the first element (the head) of a list, but we know that lists could possibly be empty, so how can this function deliver on its promise?

Unfortunately, it can't. But there exists a different function that can: `head :: NonEmpty a -> a` from the `Data.List.NonEmpty` module! The trick here is that this other `head` does not take a general list as input, it takes a different type entirely, one that promises to have at least one element and, therefore, can deliver on its promise!

We could also potentially use smart constructors with `newtype` and enforce some sort of restrictions in the type system, as we saw in earlier chapters, But this solution can sometimes be less ergonomic to use.

An alternative approach is to use `data` types to encode the absence of a proper result, for example, using `Maybe`, as we'll see in a future chapter.

Make sure the functions you write return a result for every input, either by constraining the input using types or by encoding the absence of a result using types.

Parsing markup?

Let's get back to the task at hand.

As stated previously, our strategy for parsing the markup text is:

1. Split the string into a list where each element is a separate line (which we can do with `lines`), and
2. Go over the list line by line and process it, remembering information from previous lines if necessary

Remember that we want to start by ignoring all of the markup syntax and just group lines together into paragraphs (paragraphs are separated by an empty line), and iteratively add new features later in the chapter:

```haskell
parse :: String -> Document
parse = parseLines [] . lines -- (1)

parseLines :: [String] -> [String] -> Document
parseLines currentParagraph txts =
  let
    paragraph = Paragraph (unlines (reverse currentParagraph)) -- (2), (3)
  in
    case txts of -- (4)
      [] -> [paragraph]
      currentLine : rest ->
        if trim currentLine == ""
          then
            paragraph : parseLines [] rest -- (5)
          else
            parseLines (currentLine : currentParagraph) rest -- (6)

trim :: String -> String
trim = unwords . words
```

Things to note:

1. We pass a list that contains the currently grouped paragraph (paragraphs are separated by an empty line)
2. Because of laziness, `paragraph` is not computed until it's needed, so we don't have to worry about the performance implications in the case that we are still grouping lines
3. Why do we reverse `currentParagraph`? (See point (6))
4. We saw case expressions used to deconstruct `newtype`s and `Char`s, but we can also pattern match on lists and other ADTs as well! In this case, we match against two patterns, an empty list (`[]`), and a "cons cell" - a list with at least one element (`currentLine : rest`). In the body of the "cons" pattern, we bind the first element to the name `currentLine`, and the rest of the elements to the name `rest`.

   We will talk about how all of this works really soon!
5. When we run into an empty line, we add the accumulated paragraph to the resulting list (A `Document` is a list of structures) and start the function again with the rest of the input.
6. We pass the new lines to be grouped in a paragraph **in reverse order** because of performance characteristics - because of the nature of singly-linked lists, prepending an element is fast, and appending is slow. Prepending only requires us to create a new cons (`:`) cell to hold a pointer to the value and a pointer to the list, but appending requires us to traverse the list to its end and rebuild the cons cells - the last one will contain the last value of the list and a pointer to the list to append, the next will contain the value before the last value of the list and a pointer to the list, which contains the last element and the appended list, and so on.

This code above will group together paragraphs in a structure, but how do we view our result? In the next chapter, we will take a short detour and talk about type classes, and how they can help us in this scenario.

Displaying the parsing results (type classes)

We want to be able to print a textual representation of values of our `Document` type. There are a few ways to do that:

1. Write our own function of type `Document -> String`, which we could then print, or
2. Have Haskell write one for us

Haskell provides us with a mechanism that can automatically generate the implementation of a *type class* function called `show`, that will convert our type to `String`.

The type of the function `show` looks like this:

```
show :: Show a => a -> String
```

This is something new we haven't seen before. Between `::` and `=>` you see what is called a **type class constraint** on the type `a`. What we say in this signature is that the function `show` can work on any type that is a member of the type class `Show`.

Type classes is a feature in Haskell that allows us to declare a common interface for different types. In our case, Haskell's standard library defines the type class `Show` in the following way (this is a simplified version but good enough for our purposes):

```
class Show a where
  show :: a -> String
```

A type class declaration describes a common interface for Haskell types. `show` is an overloaded function that will work for any type that is an *instance* of the type class `Show`. We can define an instance of a type class manually like this:

```
instance Show Bool where
  show x =
    case x of
      True -> "True"
      False -> "False"
```

Defining an instance means providing an implementation for the interface of a specific type. When we call the function `show` on a data type, the compiler will search the type's `Show` instance, and use the implementation provided in the instance declaration.

```
ghci> show True
"True"
ghci> show 187
"187"
ghci> show "Hello"
"\"Hello\""
```

As seen above, the `show` function converts a value to its textual representation. That is why `"Hello"` includes the quotes as well. The `Show` type class is usually used for debugging purposes.

Deriving instances

It is also possible to automatically generate implementations of a few selected type classes. Fortunately, `Show` is one of them.

If all the types in the definition of our data type already implement an instance of `Show`, we can *automatically derive* it by adding `deriving Show` at the end of the data definition.

```
data Structure
  = Heading Natural String
  | Paragraph String
  | UnorderedList [String]
  | OrderedList [String]
  | CodeBlock [String]
  deriving Show
```

Now we can use the function `show :: Show a => a -> String` for any type that implements an instance of the `Show` type class. For example, with `print`:

```
print :: Show a => a -> IO ()
print = putStrLn . show
```

We can first convert our type to `String` and then write it to the standard output.

And because lists also implement `Show` for any element type that has a `Show` instance, we can now print `Document`s, because they are just aliases for `[Structure]`. Try it!

There are many type classes Haskellers use everyday. A couple more are `Eq` for equality and `Ord` for ordering. These are also special type classes that can be derived automatically.

Laws

Type classes often come with "rules" or "laws" that instances should satisfy, the purpose of these laws is to provide *predictable behaviour* across instances, so that when we run into a new instance we can be confident that it will behave in an expected way, and we can write code that works generically for all instances of a type class while expecting them to adhere to these rules.

As an example, let's look at the `Semigroup` type class:

```
class Semigroup a where
  (<>) :: a -> a -> a
```

This type class provides a common interface for types with an operation `<>` that can combine two values into one in some way.

This type class also mentions that this `<>` operation should be associative, meaning that these two sides should evaluate to the same result:

```
x <> (y <> z) = (x <> y) <> z
```

An example of a lawful instance of `Semigroup` is lists with the append operation (`++`):

```
instance Semigroup [a] where
  (<>) = (++)
```

Unfortunately, the Haskell type system cannot "prove" that instances satisfy these laws, but as a community, we often shun unlawful instances.

Many data types (together with their respective operations) can form a `Semigroup`, and instances don't even have to look similar or have a common analogy/metaphor (and this is true for many other type classes as well).

Type classes are often just *interfaces* with *laws* (or expected behaviours if you will). Approaching them with this mindset can be very liberating!

To put it differently, **type classes can be used to create abstractions** - interfaces with laws/expected behaviours where we don't actually care about the concrete details of the underlying type, just that it *implements a certain API and behaves in a certain way*.

Regarding `Semigroup`, we have previously created a function that looks like `<>` for our `Html` EDSL! We can add a `Semigroup` instance for our `Structure` data type and have a nicer API!

Exercise: Please do this and remove the `append_` function from the API.

Parsing markup part 02 (Pattern matching)

Maybe

Previously on partial functions, we mentioned that one way to avoid writing partial functions is to encode the absence of a result using `Maybe`:

```
data Maybe a
  = Nothing
  | Just a
```

`Maybe` is a data type from the standard library (named base) for adding an additional value to a type: the absence of a value. For example, `Maybe Bool` has three values, two with the `Just` constructor to represent regular boolean values (`Just True` and `Just False`) and another value, `Nothing` to represent the absence of a boolean value.

We can use this to encode the result of `head`, a function that promises to return the first element of a list, without creating a partial function:

```
safeHead :: [a] -> Maybe a
```

This way, when the list is empty, we can return `Nothing`, and when it has at least one element, we can return `Just <first element>`. This function can be found in the Data.Maybe module under the name listToMaybe.

In order to *consume* values of type `Maybe <something>`, and other types created with `data`, we can use pattern matching.

Pattern Matching

We've already seen pattern matching a few times. It is an incredibly versatile feature of Haskell; we can use it to do two main things:

1. Deconstruct complex values
2. Control flow

As we've seen when discussing newtypes, we can use **case expressions** and **function definitions** to deconstruct a `newtype`. Same for `data` types as well:

```
import Data.Word (Word8) -- Word8 is an 8-bit unsigned integer type

-- | A data type representing colors
data Color
  = RGB Word8 Word8 Word8

getBluePart :: Color -> Word8
getBluePart color =
  case color of
    RGB _ _ blue -> blue
```

In `getBluePart` we deconstruct a composite value into its part and extract the third component representing the blue value in a color represented by red, green, and blue components (RGB).

Note that `blue` is the name we give to the third component, so it will be bound to the right of the arrow that comes after the pattern. This is similar to a function argument. Also note that `_` matches any value *without* binding it to a name.

We can also try to match a value with more than one pattern:

```
data Brightness
  = Dark
  | Bright

data EightColor
  = Black
  | Red
  | Green
  | Yellow
  | Blue
  | Magenta
  | Cyan
  | White

data AnsiColor
  = AnsiColor Brightness EightColor

ansiColorToVGA :: AnsiColor -> Color
ansiColorToVGA ansicolor =
  case ansicolor of
    AnsiColor Dark Black ->
      RGB 0 0 0
    AnsiColor Bright Black ->
      RGB 85 85 85
    AnsiColor Dark Red ->
      RGB 170 0 0
    AnsiColor Bright Red ->
      RGB 255 85 85
    -- and so on
```

It's important to notice a few things here:

1. Patterns can be nested; notice how we deconstructed `ansicolor` on multiple levels
2. We try to match patterns from the top down; it is possible for patterns to overlap with one another, and the top one will win
3. If the value we try to match does not match any of the patterns listed, an error will be thrown at runtime

We can ask GHC to notify us when we accidentally write overlapping patterns, or when we haven't listed enough patterns to match all possible values, by passing the flag `-Wall` to `ghc` or `runghc`.

My recommendation is to always use `-Wall`!

> As an aside, while it is possible to use pattern matching in function definitions by defining a function multiple times, I personally don't like that feature very much and I would encourage you to avoid it, but if you want to use it instead of case expressions, it is possible.

Pattern matching on linked lists

Because linked lists have their own special syntax, we also have special syntax for their pattern match. We can use the same special syntax for creating lists when we pattern match on lists, replacing the *elements* of the list with patterns. For example:

```
safeHead :: [a] -> Maybe a
safeHead list =
  case list of
    -- Empty list
    [] -> Nothing

    -- Cons cell pattern, will match any list with at least one element
    x : _ -> Just x
```

```
exactlyTwo :: [a] -> Maybe (a, a)
exactlyTwo list =
  case list of
    -- Will match a list with exactly two elements
    [x, y] -> Just (x, y)

    -- Will match any other pattern
    _ -> Nothing
```

```
-- This will also work
exactlyTwoVersion2 :: [a] -> Maybe (a, a)
exactlyTwoVersion2 list =
  case list of
    -- Will match a list with exactly two elements
    x : y : [] -> Just (x, y)

    -- Will match any other pattern
    _ -> Nothing
```

Exercises:

1. Create a function `isBright :: AnsiColor -> Bool` that checks whether a color is bright
2. Use this table to write `ansiToUbuntu`
3. Create a function `isEmpty :: [a] -> Bool` that uses `listToMaybe` to check whether a list is empty
4. Create a function `isEmpty :: [a] -> Bool` that *doesn't* use `listToMaybe` to check whether a list is empty

---

Parsing with rich context

Previously we wrote a parser that separates documents into different paragraphs. With new features under our belt, we can now remember the exact context we are in (whether it is a text paragraph, a list, or a code block) and act accordingly!

Let's look again at the parsing code we wrote previously:

```haskell
parse :: String -> Document
parse = parseLines [] . lines

parseLines :: [String] -> [String] -> Document
parseLines currentParagraph txts =
  let
    paragraph = Paragraph (unlines (reverse currentParagraph))
  in
    case txts of
      [] -> [paragraph]
      currentLine : rest ->
        if trim currentLine == ""
          then
            paragraph : parseLines [] rest
          else
            parseLines (currentLine : currentParagraph) rest

trim :: String -> String
trim = unwords . words
```

Previously our context, `currentParagraph`, was used to group adjacent lines in an accumulative list.

Next, instead of using a `[String]` type to denote adjacent lines, we can instead use a `Structure` to denote the context.

One issue we might have, though, with representing context with the `Structure` type, is that when we start parsing, we don't have any context. But we have learned of a way to represent the absence of a value with `Maybe`! So our new context type can be `Maybe Structure` instead.

Let's rewrite our code above with our new context type:

```haskell
parse :: String -> Document
parse = parseLines Nothing . lines -- (1)

parseLines :: Maybe Structure -> [String] -> Document
parseLines context txts =
  case txts of
    [] -> maybeToList context -- (2)
    -- Paragraph case
    currentLine : rest ->
      let
        line = trim currentLine
      in
        if line == ""
          then
            maybe id (:) context (parseLines Nothing rest) -- (3)
          else
            case context of
              Just (Paragraph paragraph) ->
                parseLines (Just (Paragraph (unwords [paragraph, line]))) rest -- (4)
              _ ->
                maybe id (:) context (parseLines (Just (Paragraph line)) rest)

trim :: String -> String
trim = unwords . words
```

1. We can now pass `Nothing` when we don't have a context
2. Unsure what `maybeToList` does? Hoogle it!
3. We can split this line into two important parts:

1. `maybe id (:) context` - prepending the context to the rest of the document
2. `parseLines Nothing rest` - parsing the rest of the document

Let's focus on the first part. We want to prepend `context` to the rest of the document, but we can't write `context : parseLines Nothing rest` because `context` has the type `Maybe Structure` and not `Structure`, meaning that we *might* have a `Structure` but maybe not. If we do have a `Structure` to prepend, we wish to prepend it. If not, we want to return the result of `parseLines Nothing rest` as is. Try writing this using pattern matching!

The maybe function lets us do the same thing more compactly. It is a function that works similarly to pattern matching on a `Maybe`: the third argument to `maybe` is the value on which we pattern match, the second argument is a function to apply to the value found in a `Just` case, and the first argument is the value to return in case the value we pattern match on is `Nothing`. A more faithful translation of `maybe id (:) context (parseLines Nothing rest)` to pattern matching would look like this:

```
( case context of
    Nothing -> id
    Just structure -> (:) structure
) (parseLines Nothing rest)
```

Note how the result of this case expression is a function of type `Document -> Document`, how we partially apply `(:)` with `structure` to create a function that prepends `structure`, and how we apply `parseLines Nothing rest` to the case expression.

This way of encoding pattern matching using functions is fairly common.

Check out the types of `id`, `(:)`, and `maybe id (:)` in GHCi!

4. Hey! Didn't we say that appending `String`s/lists is slow (which is what `unwords` does)? Yes, it is. Because in our `Structure` data type, a paragraph is defined as `Paragraph String` and not `Paragraph [String]`, we can't use our trick of building a list of lines and then reverse it in the end.

   So what do we do? There are many ways to handle that; one simple way is to create a different type with the right shape:

   ```
   data Context
     = CtxHeading Natural String
     | CtxParagraph [String]
     | CtxUnorderedList [String]
     | CtxOrderedList [String]
     | CtxCodeBlock [String]
   ```

   Since creating new types in Haskell is cheap, this is a very viable solution.

   In this case, I'm going with the approach of not worrying about it too much, because it's a very local piece of code that can easily be fixed later if needed.

Let's cover more parsing cases; we want to handle headings and lists as well. We can do that by examining the first characters of a line:

```haskell
parse :: String -> Document
parse = parseLines Nothing . lines

parseLines :: Maybe Structure -> [String] -> Document
parseLines context txts =
  case txts of
    -- done case
    [] -> maybeToList context

    -- Heading 1 case
    ('*' : ' ' : line) : rest ->
      maybe id (:) context (Heading 1 (trim line) : parseLines Nothing rest)

    -- Unordered list case
    ('-' : ' ' : line) : rest ->
      case context of
        Just (UnorderedList list) ->
          parseLines (Just (UnorderedList (list <> [trim line]))) rest

        _ ->
          maybe id (:) context (parseLines (Just (UnorderedList [trim line])) rest)

    -- Paragraph case
    currentLine : rest ->
      let
        line = trim currentLine
      in
        if line == ""
          then
            maybe id (:) context (parseLines Nothing rest)
          else
            case context of
              Just (Paragraph paragraph) ->
                parseLines (Just (Paragraph (unwords [paragraph, line]))) rest
              _ ->
                maybe id (:) context (parseLines (Just (Paragraph line)) rest)

trim :: String -> String
trim = unwords . words
```

Exercise: Add the `CodeBlock` and `OrderedList` cases.

Final module

```haskell
-- Markup.hs

module Markup
  ( Document
  , Structure(..)
  , parse
  )
where

import Numeric.Natural
import Data.Maybe (maybeToList)

type Document
  = [Structure]

data Structure
  = Heading Natural String
  | Paragraph String
  | UnorderedList [String]
  | OrderedList [String]
  | CodeBlock [String]
  deriving (Eq, Show)    -- (1)


parse :: String -> Document
parse = parseLines Nothing . lines

parseLines :: Maybe Structure -> [String] -> Document
parseLines context txts =
  case txts of
    -- done case
    [] -> maybeToList context

    -- Heading 1 case
    ('*' : ' ' : line) : rest ->
      maybe id (:) context (Heading 1 (trim line) : parseLines Nothing rest)

    -- Unordered list case
    ('-' : ' ' : line) : rest ->
      case context of
        Just (UnorderedList list) ->
          parseLines (Just (UnorderedList (list <> [trim line]))) rest

        _ ->
          maybe id (:) context (parseLines (Just (UnorderedList [trim line])) rest)

    -- Ordered list case
    ('#' : ' ' : line) : rest ->
      case context of
        Just (OrderedList list) ->
          parseLines (Just (OrderedList (list <> [trim line]))) rest

        _ ->
          maybe id (:) context (parseLines (Just (OrderedList [trim line])) rest)

    -- Code block case
    ('>' : ' ' : line) : rest ->
      case context of
        Just (CodeBlock code) ->
          parseLines (Just (CodeBlock (code <> [line]))) rest

        _ ->
          maybe id (:) context (parseLines (Just (CodeBlock [line])) rest)

    -- Paragraph case
```

```
    currentLine : rest ->
      let
        line = trim currentLine
      in
        if line == ""
          then
            maybe id (:) context (parseLines Nothing rest)
          else
            case context of
              Just (Paragraph paragraph) ->
                parseLines (Just (Paragraph (unwords [paragraph, line]))) rest
              _ ->
                maybe id (:) context (parseLines (Just (Paragraph line)) rest)

trim :: String -> String
trim = unwords . words
```

How do we know our parser works correctly?

In an earlier chapter, we parsed a few examples of our markup language by hand. Now, we can try to test our parser by comparing our solutions to our parser. By deriving `Eq` for our `Structure` data type (marked with (1) in "final module" above), we can compare solutions with the `==` (equals) operator.

Try it in GHCi! You can read a text file in GHCi using the following syntax:

```
ghci> txt <- readFile "/tmp/sample.txt"
```

And then compare with the handwritten example values from the solutions (after adding them to the module and loading them in GHCi):

```
ghci> parse txt == example4
```

In a later chapter, we'll write automated tests for our parser using a testing framework. But before that, I'd like to glue things together so we'll be able to:

1. Read markup text from a file
2. Parse the text
3. Convert the result to our HTML EDSL
4. Generate HTML code

And also discuss how to work with IO in Haskell while we're at it.

Gluing things together

In this chapter, we are going to glue the pieces we built together and build an actual blog generator. We will:

1. Read markup text from a file
2. Parse the text to a `Document`
3. Convert the result to our `Html` EDSL
4. Generate HTML code
5. Write it to a file

While doing so, we will learn:

- How to work with IO
- How to import external libraries to process whole directories and create a simple command-line interface

Converting Markup to HTML

One key part is missing before we can glue everything together, and that is to convert our `Markup` data types to `Html`.

We'll start by creating a new module and importing both the `Markup` and the `Html` modules.

```
module Convert where

import qualified Markup
import qualified Html
```

Qualified Imports

This time, we've imported the modules qualified. Qualified imports mean that instead of exposing the names that we've defined in the imported module to the general module namespace, they now have to be prefixed with the module name.

For example, `parse` becomes `Markup.parse`. If we would've imported `Html.Internal` qualified, we'd have to write `Html.Internal.el`, which is a bit long.

We can also give the module a new name with the `as` keyword:

```
import qualified Html.Internal as HI
```

And write `HI.el` instead.

I like using qualified imports because readers do not have to guess where a name comes from. Some modules are even designed to be imported qualified. For example, the APIs of many container types, such as maps, sets, and vectors, are very similar. If we want to use multiple containers in a single module, we pretty much have to use qualified imports so that when we write a function such as `singleton`, which creates a container with a single value, GHC will know which `singleton` function we are referring to.

Some people prefer to use import lists instead of qualified imports, because qualified names can be a bit verbose and noisy. I will often prefer qualified imports to import lists, but feel free to try both solutions and see which fits you better. For more information about imports, see this wiki article.

Converting Markup.Structure to Html.Structure

Converting a markup structure to an HTML structure is mostly straightforward at this point, we need to pattern match on the markup structure and use the relevant HTML API.

```
convertStructure :: Markup.Structure -> Html.Structure
convertStructure structure =
  case structure of
    Markup.Heading 1 txt ->
      Html.h1_ txt

    Markup.Paragraph p ->
      Html.p_ p

    Markup.UnorderedList list ->
      Html.ul_ $ map Html.p_ list

    Markup.OrderedList list ->
      Html.ol_ $ map Html.p_ list

    Markup.CodeBlock list ->
      Html.code_ (unlines list)
```

Notice that running this code with `-Wall` will reveal that the pattern matching is *non-exhaustive*. This is because we don't currently have a way to build headings that are not `h1`. There are a few ways to handle this:

- Ignore the warning - this will likely fail at runtime one day, and the user will be sad
- Pattern match other cases and add a nice error with the `error` function - it has the same disadvantage above, but will also no longer notify of the unhandled cases at compile time
- Pattern match and do the wrong thing - user is still sad
- Encode errors in the type system using `Either`, we'll see how to do this in later chapters
- Restrict the input - change `Markup.Heading` to not include a number but rather specific supported headings. This is a reasonable approach
- Implement an HTML function supporting arbitrary headings. Should be straightforward to do

---

Exercises

1. Implement `h_ :: Natural -> String -> Structure` which we'll use to define arbitrary headings (such as `<h1>`, `<h2>`, and so on).
2. Fix `convertStructure` using `h_`.

Document -> Html

To create an `Html` document, we need to use the `html_` function. This function expects two things: a `Title` and a `Structure`.

For a title, we could just supply it from outside using the file name.

To convert our markup `Document` (which is a list of markup `Structure`) to an HTML `Structure`, we need to convert each markup `Structure` and then concatenate them together.

We already know how to convert each markup `Structure`; we can use the `convertStructure` function we wrote and `map`. This will provide us with the following function:

```
map convertStructure :: Markup.Document -> [Html.Structure]
```

To concatenate all of the `Html.Structure`, we could try to write a recursive function. However, we will quickly run into an issue with the base case: what to do when the list is empty?

We could just provide a dummy `Html.Structure` that represents an empty HTML structure.

Let's add this to `Html.Internal`:

```
empty_ :: Structure
empty_ = Structure ""
```

Now we can write our recursive function. Try it!

Remember the `<>` function we implemented as an instance of the `Semigroup` type class? We mentioned that `Semigroup` is an **abstraction** for things that implements `(<>) :: a -> a -> a`, where `<>` is associative (`a <> (b <> c) = (a <> b) <> c`).

It turns out that having an instance of `Semigroup` and also having a value that represents an "empty" value is a fairly common pattern. For example, a string can be concatenated, and the empty string can serve as an "empty" value. And this is actually a well known **abstraction** called **monoid**.

Monoids

Actually, "empty" isn't a very good description of what we want, and isn't very useful as an abstraction. Instead, we can describe it as an "identity" element that satisfies the following laws:

- `x <> <identity> = x`
- `<identity> <> x = x`

In other words, if we try to use this "empty" - this identity value, as one argument to `<>`, we will always get the other argument back.

For `String`, the empty string, `""`, satisfies this:

```
"" <> "world" = "world"
"hello" <> "" = "hello"
```

This is, of course, true for any value we'd write and not just "world" and "hello".

Actually, if we move out of the Haskell world for a second, even integers with `+` as the associative binary operations `+` (in place of `<>`) and `0` in place of the identity member form a monoid:

```
17 + 0 = 17
0 + 99 = 99
```

So integers together with the `+` operation form a semigroup, and together with `0` form a monoid.

We learn new things from this:

1. A monoid is a more specific abstraction over semigroup; it builds on it by adding a new

condition (the existence of an identity member)
2. This abstraction can be useful! We can write a general `concatStructure` that could work for any monoid

And indeed, there exists a type class in `base` called `Monoid`, which has `Semigroup` as a **super class**.

```
class Semigroup a => Monoid a where
  mempty :: a
```

> Note: this is a simplified version. The actual is a bit more complicated because of backward compatibility and performance reasons. `Semigroup` was actually introduced in Haskell after `Monoid`!

We could add an instance of `Monoid` for our HTML `Structure` data type:

```
instance Monoid Structure where
  mempty = empty_
```

And now, instead of using our own `concatStructure`, we can use the library function:

```
mconcat :: Monoid a => [a] -> a
```

Which could theoretically be implemented as:

```
mconcat :: Monoid a => [a] -> a
mconcat list =
  case list of
    [] -> mempty
    x : xs -> x <> mconcat xs
```

Notice that because `Semigroup` is a *super class* of `Monoid`, we can still use the `<>` function from the `Semigroup` class without adding the `Semigroup a` constraint to the left side of `=>`. By adding the `Monoid a` constraint, we implicitly add a `Semigroup a` constraint as well!

This `mconcat` function is very similar to the `concatStructure` function, but this one works for any `Monoid`, including `Structure`! Abstractions help us identify common patterns and **reuse** code!

> Side note: integers with `+` and `0` aren't actually an instance of `Monoid` in Haskell. This is because integers can also form a monoid with `*` and `1`! But **there can only be one instance per type**. Instead, two other `newtype`s exist that provide that functionality, Sum and Product. See how they can be used in `ghci`:
>
> ```
> ghci> import Data.Monoid
> ghci> Product 2 <> Product 3 -- note, Product is a data constructor
> Product {getProduct = 6}
> ghci> getProduct (Product 2 <> Product 3)
> 6
> ghci> getProduct $ mconcat $ map Product [1..5]
> 120
> ```

Another abstraction?

We've used `map` and then `mconcat` twice now. Surely there has to be a function that unifies this pattern. And indeed, it is called `foldMap`, and it works not only for lists but also for any data structure that can be "folded", or "reduced", into a summary value. This abstraction and type class is called **Foldable**.

For a simpler understanding of `Foldable`, we can look at `fold`:

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

-- compare with
mconcat :: Monoid m            => [m] -> m
```

`mconcat` is just a specialized version of `fold` for lists. And `fold` can be used for any pair of a data structure that implements `Foldable` and a payload type that implements `Monoid`. This could be `[]` with `Structure`, or `Maybe` with `Product Int`, or your new shiny binary tree with `String` as the payload type. But note that the `Foldable` type must be of *kind* `* -> *`. So, for example `Html` cannot be a `Foldable`.

`foldMap` is a function that allows us to apply a function to the payload type of the `Foldable` type right before combining them with the `<>` function.

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

-- compare to a specialized version with:
-- - t ~ []
-- - m ~ Html.Structure
-- - a ~ Markup.Structure
foldMap
  :: (Markup.Structure -> Html.Structure)
  -> [Markup.Structure]
  -> Html.Structure
```

True to its name, it really "maps" before it "folds". You might pause here and think, "this 'map' we are talking about isn't specific for lists; maybe that's another abstraction?" Yes. It is actually a very important and fundamental abstraction called `Functor`. But I think we had enough abstractions for this chapter. We'll cover it in a later chapter!

Finishing our conversion module

Let's finish our code by writing `convert`:

```
convert :: Html.Title -> Markup.Document -> Html.Html
convert title = Html.html_ title . foldMap convertStructure
```

Now we have a full implementation and can convert markup documents to HTML:

```haskell
-- Convert.hs
module Convert where

import qualified Markup
import qualified Html

convert :: Html.Title -> Markup.Document -> Html.Html
convert title = Html.html_ title . foldMap convertStructure

convertStructure :: Markup.Structure -> Html.Structure
convertStructure structure =
  case structure of
    Markup.Heading n txt ->
      Html.h_ n txt

    Markup.Paragraph p ->
      Html.p_ p

    Markup.UnorderedList list ->
      Html.ul_ $ map Html.p_ list

    Markup.OrderedList list ->
      Html.ol_ $ map Html.p_ list

    Markup.CodeBlock list ->
      Html.code_ (unlines list)
```

Summary

We learned about:

- Qualified imports
- Ways to handle errors
- The `Monoid` type class and abstraction
- The `Foldable` type class and abstraction

Next, we are going to glue our functionality together and learn about I/O in Haskell!

Working with IO

In previous chapters, we were able to build a parser from a text string to a Haskell representation of our markup language, and we built an EDSL for easy writing of HTML code. However, our program is still not useful to other users because we did not make this functionality accessible via some sort of user interface.

In our program, we'd like to take user input and then convert it to HTML. There are many ways to design this kind of interface, for example:

- Get text input via the *standard input* and output HTML via the *standard output*
- Receive two file names as *command-line arguments*, read the contents of the first one, and write the output to the second one
- Ask for fancier command-line arguments parsing and prefix the file names with flags indicating what they are
- Some fancy GUI interface
- Combination of all of the above

To make this interesting, we will start with the following interface:

1. If the user calls the program without arguments, we will read from the standard input, and write to the standard output
2. If the user calls the program with two arguments, the first one will be the input file name, and the second one will be the output file name
3. If the output file already exists, we'll ask the user if they want to overwrite the file
4. On any other kind of input, we'll print a generic message explaining the proper usage

In a later chapter, we will add a fancier command-line interface using a library, and also read whole directories in addition to single files.

But first, we need to learn about I/O in Haskell, what makes it special, and why it's different from other programming languages.

Purely functional

Originally, Haskell was designed as an *open standard* functional programming language with **non-strict semantics**, to serve as a unifying language for future research in functional language design.

In GHC Haskell, we use a *lazy evaluation strategy* to implement non-strict semantics (We've talked about laziness before).

The requirement for non-strict semantics raises interesting challenges: How do we design a language that can do more than just evaluate expressions, how do we model interaction with the outside world, how do we do I/O?

The challenge with doing I/O operations in a language with a lazy evaluation strategy is that as programs grow larger, the order of evaluation becomes less trivial to figure out. Consider this hypothetical code example (which won't actually type-check in Haskell, we'll see why soon):

```
addWithInput :: Int -> Int
addWithInput n = readIntFromStdin + n

main =
  let
    result1 = addWithInput 1
    result2 = addWithInput 2
  in
    print (result2 - result1)
```

This hypothetical program will read 2 integers from the standard input, and then will subtract the second one (+2) from the first one (+1), or so we would expect if this was a strict language. In a strict language, we expect the order of operations to happen from the top down.

But in a lazy language, we don't evaluate an expression until it is needed, and so neither `result1` nor `result2` are evaluated until we wish to print the result of subtracting one from the other. And then, when we try to evaluate `-`, it evaluates the two arguments from left to right, so we first evaluate `result2`.

Evaluating `result2` with substitution means replacing occurrences of `n` with the input `2`, and then evaluate the top-level function (`+`), which is a primitive function. We then evaluate its arguments, `readIntFromStdin` and then `n`; at this point *we are reading the first integer from the stdin*.

After calculating the result, we can move to evaluate `result1`, which *will read the second integer from stdin*. This is the complete opposite of what we wanted!

Issues like these make lazy evaluation hard to work within the presence of **side effects** - when the evaluation of an expression *can affect or be affected by the outside world*, this includes reading/writing from mutable memory or performing I/O operations.

We call functions that have side-effects, such as `addWithInput`, **impure functions**. And an unfortunate consequence of impure functions is that **they can return different results even when they take the same input**.

The presence of impure functions makes it harder for us to reason about lazy evaluation, and also messes up our ability to use **equational reasoning** to understand programs.

Therefore, in Haskell, it was decided to only allow **pure** functions and expressions - ones that have **no side effects**. Pure functions will *always* return the same output (given the same input) and **evaluating pure expressions is deterministic**.

But now, how can we do I/O operations? There are many possible solutions.

For Haskell, it was decided to design an interface with an accompanied type called `IO`. `IO`'s interface will force a distinction from non-I/O expressions and will also require that in order to *combine* multiple `IO` operations, we will have to **specify the order of the operations**.

IO

`IO` is an opaque type, like our `Html` type, in which we hide its internal representation from the user behind an interface. But in this case, `IO` is a built-in type and its internals are hidden by the Haskell language rather than a module.

Similar to `Maybe`, `IO` has a payload type that represents the result of an `IO` operation. When there isn't a meaningful result, we use the unit type, `()` (which only has one value: `()`) to represent that.

Here are a few `IO` operations and functions that return `IO` operations:

```
putStrLn :: String -> IO ()

getLine :: IO String

getArgs :: IO [String]

lookupEnv :: String -> IO (Maybe String)

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

Notice that each function returns an `IO <something>`, but what does that mean?

The meaning behind `IO a` is that it is a *description of a program (or subroutine) that, when executed, will produce some value of type* `a`*, and may do some I/O effects during execution*.

Executing an `IO a` is different from evaluating it. Evaluating an `IO a` expression is pure - the **evaluation** will always reduce to the same **description** of a program. This helps us keep purity and equational reasoning!

The Haskell runtime will *execute* the entry point to the program (the expression `main` that must have the type `IO ()`). For our IO operation to also run - it has to be *combined into* the `main` expression - let's see what that means.

Combining IO expressions

Like our `Html.Structure` type, the IO interface provides **combinators** for composing small `IO` operations into bigger ones. This interface also ensures that the order of operations is well-defined!

Note that, just like the `<>` we've defined for `Html.Structure`, the combinators for `IO` are implemented as **type-class instances** rather than specialized variants (for example our `append_` function was a specialized version of `<>` tailored only for `Structure`).

In this section, I will introduce specialized type signatures rather than generalized ones, because I think it'll be easier to digest, but we'll talk about the generalized versions later.

>>=

Our first combinator is `>>=` (pronounced bind), and is the most useful of the bunch:

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

This combinator takes two arguments, the first is an IO operation, and the second is a function that takes as input *the result of the first IO operation* and returns a new `IO b`, which is the final result.

Here are a few examples using the functions we described above:

1. Echo

```
getLine >>= (\line -> putStrLn line)
```

We are reading a line from the standard input on the left of `>>=`, we receive the input to the right of `>>=` as an argument to the lambda function, and then write it to the standard output in the body of the lambda function. `>>=`'s role here is to **pass the result of the IO operation on the left to the function returning an IO operation on the right**.

Notice how `>>=` *defines an order of operations - from left to right*.

The type of each sub-expression here is:

```
getLine :: IO String

putStrLn :: String -> IO ()

(>>=) :: IO String -> (String -> IO ()) -> IO ()

line :: String
```

- Question: what is the type of the whole expression?

Also, note that this example can be written in a more concise manner in point-free style `getLine >>= putStrLn`.

2. Appending two inputs

```
getLine >>= (\honorific -> getLine >>= (\name -> putStrLn ("Hello " ++ honorific
++ " " ++ name)))
```

This subroutine combines multiple operations together; it reads two lines from the standard input and prints a greeting. Note that:

- Using `>>=` defines the order of operation from left to right
- Because of the scoping rules in Haskell, `honorific` will be in scope in the body of the function for which it is its input, including the most inner function

This is a bit hard to read, but we can remove the parenthesis and add indentation to make it a bit easier to read:

```
getLine >>= \honorific ->
  getLine >>= \name ->
    putStrLn ("Hello " ++ honorific ++ " " ++ name)
```

Let's see a few more combinators!

*> and >>

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

`*>` and `>>` have the same type signature for `IO` and mean the same thing. In fact, `*>` is a slightly

more generalized version of `>>` and can always be used instead of `>>`, which only still exists to avoid breaking backward compatibility.

`*>` for `IO` means run the first IO operation, discard the result then run the second operation. It can be implemented using `>>=`:

```
a *> b = a >>= \_ -> b
```

This combinator is useful when we want to run several `IO` operations one after the other that might not return anything meaningful, such as `putStrLn`:

```
putStrLn "hello" *> putStrLn "world"
```

pure and return

```
pure :: a -> IO a
```

like `*>` and `>>`, `pure` is a more general version of `return`. `pure` also has the advantage of not having a resemblance to an unrelated keyword in other languages.

Remember that we said `IO a` is a description of a program that, when executed, will produce some value of type `a` and may do some I/O effects during execution?

With `pure`, we can build an `IO a` that does no I/O and will produce a specific value of type `a`, the one we supply to `pure`!

This function is useful when we want to do some uneffectful computation that depends on `IO`.

For example:

```
confirm :: IO Bool
confirm =
  putStrLn "Are you sure? (y/n)" *>
    getLine >>= \answer ->
      case answer of
        "y" -> pure True
        "n" -> pure False
        _ ->
          putStrLn "Invalid response. use y or n" *>
            confirm
```

Trying to return just `True` or `False` here wouldn't work because of the type of `>>=`:

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

The right side of `>>=` in our code example (`\answer -> case ...`) must be of type `String -> IO Bool`. This is because:

1. `getLine :: IO String`, so the `a` in the type signature of `>>=` should be the same as `String` in this instance, and
2. `confirm :: IO Bool`, so `b` should be `Bool`

fmap and <$>

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

`<$>` is the infix version of `fmap`. Use it at your discretion.

What if we want a function that reads a line from stdin and returns it with `!` at the end? We could use a combination of `>>=` and `pure`:

```
getLine >>= \line -> pure (line ++ "!")
```

The pattern is unified to the `fmap` function:

```
fmap (\line -> line ++ "!") getLine
```

`fmap` applies a function to the value to be returned from the `IO` operation, also known as "mapping" over it.

(By the way, Have you noticed the similarities between `fmap` and `map :: (a -> b) -> [a] -> [b]`?)

Summary

Here's a list of `IO` combinators we ran into:

```
-- chaining IO operations: passing the *result* of the left IO operation
-- as an argument to the function on the right.
-- Pronounced "bind".
(>>=) :: IO a -> (a -> IO b) -> IO b

-- sequence two IO operations, discarding the payload of the first.
(*>) :: IO a -> IO b -> IO b

-- "lift" a value into IO context, does not add any I/O effects.
pure :: a -> IO a

-- "map" (or apply a function) over the payload value of an IO operation.
fmap :: (a -> b) -> IO a -> IO b
```

IO is first class

The beauty of `IO` is that it's a completely first-class construct in the language, and is not really different from `Maybe`, `Either`, or `Structure`. We can pass it to functions, put it in a container, etc. Remember that it represents a description of a program, and without combining it into `main` in some way won't actually *do* anything. It is just a value!

Here's an example of a function that takes IO actions as input:

```
whenIO :: IO Bool -> IO () -> IO ()
whenIO cond action =
  cond >>= \result ->
    if result
      then action
      else pure ()
```

And how it can be used:

```
main :: IO ()
main =
  putStrLn "This program will tell you a secret" *>
    whenIO confirm (putStrLn "IO is actually pretty awesome") *>
      putStrLn "Bye"
```

Notice how `putStrLn "IO is actually pretty awesome"` isn't executed right away, but only if it is what `whenIO` returns, and in turn, is *combined* with `*>` as part of the `main` expression.

Getting out of IO?

What we've seen above has great consequences for the Haskell language. In our `Html` type, we had a function `render :: Html -> String` that could turn an `Html` into a string value.

In Haskell, **there is no way** to implement a function such as `execute :: IO a -> a` that preserves purity and equational reasoning!

Also, `IO` is *opaque*. It does not let us examine it. So we are really bound to what the Haskell API for `IO` allows us to do.

This means that **we need to think about using IO differently**!

In Haskell, once we get into `IO`, there is no getting out. The only thing we can do is to build bigger IO computations by *combining* it with more IO computations.

We also can't use `IO a` in place of an `a`. For example, we can't write `getLine ++ "!"` because `++` expects both sides to be `String`, but `getLine`'s type is `IO String`. The types do not match! We have to use `fmap`, and the return type must be `IO String`, as we've seen before.

In Haskell, we like to keep `IO` usage minimal, and we like to push it to the edges of the program. This pattern is often called *Functional core, imperative shell*.

Functional core, imperative shell

In our blog generator program, we want to read a file, parse it, convert it to HTML, and then print the result to the console.

In many programming languages, we might interleave reading from the file with parsing, and writing to the file with the HTML conversion. But we don't mix these here. Parsing operates on a `String` value rather than some file handle, and `Html` is being converted to a `String` rather than being written to the screen directly.

This approach of separating `IO` and pushing it to the edge of the program gives us a lot of flexibility. These functions without `IO` are easier to test and examine (because they are guaranteed to have deterministic evaluation!), and they are more modular and can work in many contexts (reading from stdin, reading from a network socket, writing to an HTTP connection, and more).

This pattern is often a good approach for building Haskell programs, especially batch programs.

Building a blog generator

We'd like to start building a blog generator, and we want to have the following interface:

1. If the user calls the program without arguments, we will read from the standard input, and write to the standard output
2. If the user calls the program with two arguments, the first one will be the input file name, and the second one will be the output file name
3. If the output file already exists, we'll ask the user if they want to overwrite the file
4. On any other kind of input, we'll print a generic message explaining the proper usage

We are going to need a few functions:

```
getArgs :: IO [String] -- Get the program arguments

getContents :: IO String -- Read all of the content from stdin

readFile :: FilePath -> IO String -- Read all of the content from a file

writeFile :: FilePath -> String -> IO () -- Write a string into a file

doesFileExist :: FilePath -> IO Bool -- Checks whether a file exists
```

And the following imports:

```
import System.Directory (doesFileExist)
import System.Environment (getArgs)
```

We don't need to add the following import because `Prelude` already imports these functions for us:

```
-- imported by Prelude
import System.IO (getContents, readFile, writeFile)
```

Exercises

1. Implement a function `process :: Title -> String -> String` which will parse a document to markup, convert it to HTML, and then render the HTML to a string.

2. Try implementing the "imperative shell" for our blog generator program. Start with `main`, pattern match on the result of `getArgs`, and decide what to do. Look back at the examples above for inspiration.

Do notation

While using `>>=` to chain `IO` actions is manageable, Haskell provides an even more convenient syntactic sugar called *do notation* which emulates imperative programming.

A *do block* starts with the `do` keyword and continues with one or more "statements" which can be one of the following:

1. An expression of type `IO ()`, such as:
   - `putStrLn "Hello"`
   - `if True then putStrLn "Yes" else putStrLn "No"`
2. A let block, such as
   - `let x = 1`
   - or multiple let declarations:

     ```
     let
       x = 1
       y = 2
     ```

     Note that we do not write the `in` here.
3. A binding `<variable> <- <expresion>`, such as

   ```
   line <- getLine
   ```

And the last "statement" must be an expression of type `IO <something>` - this will be the result type of the do block.

These constructs are desugared (translated) by the Haskell compiler to:

1. `<expression> *>`,
2. `let ... in` and
3. `<expression> >>= \<variable>`

respectively.

For example:

```
greeting :: IO ()
greeting = do
  putStrLn "Tell me your name."
  let greet name = "Hello, " ++ name ++ "!"
  name <- getLine
  putStrLn (greet name)
```

Is just syntactic sugar for:

```
greeting :: IO ()
greeting =
  putStrLn "Tell me your name." *>
    let
      greet name = "Hello, " ++ name ++ "!"
    in
      getLine >>= \name ->
        putStrLn (greet name)
```

It's important to note the difference between `let` and `<-` (bind). `let` is used to give a new name to an expression that will be in scope for subsequent lines, and `<-` is used to bind the result `a` in an `IO a` action to a new name which will be in scope for subsequent lines.

code	operator	type of the left side	type of the right side	comment
`let gretting = "hello"`	`=`	`String`	`String`	Both sides are interchangeable
`let mygetline = getLine`	`=`	`IO String`	`IO String`	We just create a new name for `getLine`
`name <- getLine`	`<-`	`String`	`IO String`	Technically `<-` is not an operator, but just a syntactic sugar for `>>= +` lambda, where we bind the result of the computation to a variable

Do notation is very very common and is often preferable to using `>>=` directly.

Exercises

1. Translate the examples in this chapter to *do notation*.
2. Translate our glue code for the blog generator to *do notation*.

Summary

In this chapter, we discussed what "purely functional" means, where the initial motivation for being purely functional comes from, and how Haskell's I/O interface allows us to create descriptions of programs without breaking purity.

We have also achieved a major milestone. With this chapter, we have implemented enough pieces to finally run our program on a single document and get an HTML-rendered result!

However, our command-line interface is still sub-par. We want to render a blog with multiple articles, create an index page, and more. We still have more to do to be able to call our program a blog generator.

Let's keep going!

Defining a project description

Up until now, we've only used `base` and the libraries included with GHC. Because of that, we didn't really need to do anything fancier than `runghc` to run our program. However, we want to start using external libraries not included with GHC in our programs.

External packages can be downloaded from Hackage - Haskell's central package archive, Stackage - a subset of Hackage packages known to work together or even from remote git repositories. Usually, Haskellers use a **package manager** to download and manage packages for different projects. The most popular package managers for Haskell are cabal and stack.

A major difference between the two is their philosophy. `cabal` tries to be a more minimalist tool that handles building Haskell projects, doing package management using the whole of Hackage, and uses complicated algorithms to make sure packages work together. `stack` tries to be a more maximalistic tool that handles installing the right GHC for each project, provides integration with external tools like hoogle, and lets the user choose which 'set' of packages (including their versions) they want to use.

If you've installed Haskell using GHCup, you most likely have `cabal` installed. If you've installed Haskell using stack, well, you have `stack` installed. Check the haskell.org downloads page if that's not the case.

Creating a project

Using external packages can be done in multiple ways. For quick experimentation, we can just ask stack or cabal to build or even run our program with external packages. But as programs get larger, use more dependencies, and require more functionality, it is better to **create a project description** for our programs and libraries.

The project description is done in a **cabal file**. We can ask cabal or stack to generate one for us using `cabal init --libandexe` or `stack new`, along with many other files, but we will likely need to edit the file by hand later. For now, let's just paste an initial example in `hs-blog.cabal` and edit it.

```
cabal-version:       2.4

name:                name should match with <name>.cabal
version:             version should use PvP
synopsis:            Synopsis will appear in the hackage package listing and search
description:         The description will appear at the top of a library
homepage:            Homepage url
bug-reports:         issue-tracker url
license:             License name
license-file:        License file
author:              Author name
maintainer:          Maintainer email
category:            Hackage categories, separated by commas
extra-doc-files:
  README.md

common common-settings
  default-language: Haskell2010
  ghc-options:
    -Wall

library
  import: common-settings
  hs-source-dirs: src
  build-depends:
      base
    , directory
  exposed-modules:
    HsBlog
      HsBlog.Convert
      HsBlog.Html
        HsBlog.Html.Internal
      HsBlog.Markup
  -- other-modules:

executable hs-blog-gen
  import: common-settings
  hs-source-dirs: app
  main-is: Main.hs
  build-depends:
      base
    , <package-name>
  ghc-options:
    -O
```

Let's break it down to a few parts, the package metadata, common settings, library and executable.

Package metadata

The first part should be fairly straightforward from the comments, maybe except for:

- `cabal-version` : Defines which cabal versions can build this project. We've specified 2.4 and above. More info on different versions.
- `name` : The name of your library and package. Must match with the `.cabal` filename. Usually starts with a lowercase. Check if your package name is already taken on Hackage.
- `version` : Some Haskell packages use semver, most use PvP.
- `license` : Most Haskell packages use BSD-3-Clause. Neil Mitchell blogged about this. You can find more licenses if you'd like at choosealicense.com.
- `extra-doc-files` : Include extra doc files here, such as `README` or `CHANGELOG` .

Let's fill this with the metadata of our project:

```
cabal-version:       2.4

name:                hs-blog
version:             0.1.0.0
synopsis:            A custom blog generator from markup files
description:         This package provides a static blog generator
                     from a custom markup format to HTML.
                     It defines a parser for this custom markup format
                     as well as an html pretty printer EDSL.

                     It is used as the example project in the online book
                     'Learn Haskell Blog Generator'. See the README for
                     more details.
homepage:            https://github.com/soupi/learn-haskell-blog-generator
bug-reports:         https://github.com/soupi/learn-haskell-blog-generator/issues
license:             BSD-3-Clause
license-file:        LICENSE.txt
author:              Gil Mizrahi
maintainer:          gilmi@posteo.net
category:            Learning, Web
extra-doc-files:
  README.md
```

Common settings

Cabal package descriptions can include multiple "targets": libraries, executables, and test suites. Since Cabal 2.2, we can use common stanzas to group settings to be shared between different targets, so we don't have to repeat them for each target.

In our case, we've created a new common stanza (or block) called `common-settings` and defined the default language (Haskell has two standards, 98 and 2010), and instructed GHC to compile with `-Wall`.

```
common common-settings
  default-language: Haskell2010
  ghc-options:
    -Wall
```

Later, in our targets' descriptions, we can add `import: common-settings`, and all of these settings will be automatically added.

Library

In a `library` target, we define:

- The settings with which to build the library (in this case, we just import `common-settings`)
- The directory in which the source files can be found
- The packages we require to build the library
- The modules exposed from the library and can be used by others
- The modules *not* exposed from the library and which *cannot* be used by others; these could be any module you don't wish to export, such as an internal utility functions module. In our case, we don't have anything like this, so we commented out the `other-modules` label.

Note that it is common to specify **version bounds** for packages. Version bounds specify *which package versions this library works with*. These can also be generated using cabal with the `cabal gen-bounds` command.

```
library
  import: common-settings
  hs-source-dirs: src
  build-depends:
      base
    , directory
  exposed-modules:
    HsBlog
      HsBlog.Convert
      HsBlog.Html
        HsBlog.Html.Internal
      HsBlog.Markup
  -- other-modules:
```

Also, note that we've added an additional *hierarchy* for our modules and defined a different source directory (`src`). This means we will need to move the files around a bit and change the `module` name in each file and the `import` statements. This is to avoid conflict with other packages that a user might import.

Exercise: Do this now.

Executable

We have separated our code into two sections: a library and an executable; why?

First, libraries can be used by others. If we publish our code and someone wants to use it and build upon it, they can. Executables can't be imported by other projects. Second, we can write unit tests for libraries. It is usually beneficial to write most, if not all, of our logic as a library, and provide a thin executable over it.

Executables' descriptions are very similar to libraries; here, we define:

- The name of the executable
- Where the source directory for this application is
- Which file is the 'Main' file
- Import our library, which is named `hs-blog`
- Additional flags for GHC, e.g., `-O` to compile with optimizations

```
executable hs-blog-gen
  import: common-settings
  hs-source-dirs: app
  main-is: Main.hs
  build-depends:
      base
    , hs-blog
  ghc-options:
    -O
```

We can write many executables descriptions. In this case, we only have one.

Exercise: Add a new file: `app/Main.hs` which imports `HsBlog` and runs `main`.

Test-suites

`test-suite` defines a target for running package tests. We will get back to it in a later chapter.

Our complete .cabal file

```
cabal-version:       2.4

name:                hs-blog
version:             0.1.0.0
synopsis:            A custom blog generator from markup files
description:         This package provides a static blog generator
                     from a custom markup format to HTML.
                     It defines a parser for this custom markup format
                     as well as an html pretty printer EDSL.

                     It is used as the example project in the online book
                     'Learn Haskell Blog Generator'. See the README for
                     more details.
homepage:            https://github.com/soupi/learn-haskell-blog-generator
bug-reports:         https://github.com/soupi/learn-haskell-blog-generator/issues
license:             BSD-3-Clause
license-file:        LICENSE.txt
author:              Gil Mizrahi
maintainer:          gilmi@posteo.net
category:            Learning, Web
extra-doc-files:
  README.md

common common-settings
  default-language: Haskell2010
  ghc-options:
    -Wall

library
  import: common-settings
  hs-source-dirs: src
  build-depends:
      base
    , directory
  exposed-modules:
    HsBlog
      HsBlog.Convert
      HsBlog.Html
        HsBlog.Html.Internal
      HsBlog.Markup
  -- other-modules:

executable hs-blog-gen
  import: common-settings
  hs-source-dirs: app
  main-is: Main.hs
  build-depends:
      base
    , hs-blog
  ghc-options:
    -O
```

We'll also add a `README.md` file and a `LICENSE.txt` file:

▸ README.md
▸ LICENSE.txt

cabal.project and stack.yaml

The cabal.project and stack.yaml files are used by `cabal` and `stack` respectively to add additional information on *how to build the package*. While `cabal.project` isn't necessary to use `cabal`, `stack.yaml` is necessary in order to use `stack`, so we will cover it briefly.

There are two important fields a `stack.yaml` file must have:

- `resolver` : Describes which snapshot to use for packages and ghc version. We will choose the latest (at time of writing) on the `lts` branch: `lts-18.22`. Visit this link to find out which packages this snapshot includes, what their versions are, and which GHC version is used with this snapshot
- `packages` : Describes the location of packages we plan to build. In our case we have only one, and it can be found in the current directory

We'll add `stack.yaml` to our project directory:

```
resolver: lts-18.22

packages:
- .
```

For additional options and configurations, please consult the relevant user guides.

Usage

Now, instead of manually running `runghc Main.hs`, we will use either `stack` or `cabal` to build and run our program and package (I mostly use stack, but it's up to you).

For cabal:

Building the project - on the first run, cabal will download the package dependencies and use the GHC on PATH to build the project.

Cabal caches packages between projects, so if a new project uses the same packages with the same versions (and the same flag settings), they will not need to be reinstalled.

> In older versions of cabal, packages could be installed globally or in sandboxes. In each sandbox (and globally), there could only be one version of a package installed, and users would usually create different sandboxes for different projects without caching packages between projects.
>
> With the new build system implementation, multiple versions of the same package can be installed globally, and for each project, cabal will (try to) choose a specific version for each package dependency such that they all work together without needing sandboxing. This change helps us increase sharing of built packages while avoiding conflicts and manual handling of sandboxes.

A few important commands we should be familiar with:

```
cabal update
```

`update` fetches information from remote package repositories (specifically Hackage unless specified otherwise) and updates the local package index, which includes various information about available packages, such as their names, versions, and dependencies.

`cabal update` is usually the first command to run before fetching package dependencies.

```
cabal build
```

`build` compiles the various targets (such as `library` and `executable`s). It will also fetch and install the package dependencies when they're not already installed.

When building executables, `cabal build` will report where the executable has been created, and it is also possible to find the path to the executable using `cabal exec -- which hs-blog-gen`.

```
cabal run hs-blog-gen -- <program arguments>
```

`run` Can be used to compile and then run a target (in our case our `executable` which we named `hs-blog-gen`). We separate arguments passed to `cabal` and arguments passed to our target program with `--`.

```
cabal repl hs-blog
```

`repl` runs `ghci` in the context of the target (in our case our `library` which we named `hs-blog`) - it will load the target's package dependencies and modules to be available in `ghci`.

```
cabal clean
```

`clean` Deletes the build artifacts that we built.

There are more interesting commands we could use, such as `cabal freeze`, to generate a file which records the packages versions and flags we used to build this project, and `cabal sdist` to bundle the project source to a package tarball which can be uploaded to Hackage. If you'd like to learn more, visit the Cabal user guide.

For stack:

Building the project - on the first run, stack will install the right GHC for this project which is specified by the `resolver` field in the `stack.yaml` file, download the package dependencies, and compile the project.

Stack caches these installations between projects that use the same resolver, so future projects with the same resolver and future runs of this project won't require reinstallation. This approach is kind of a middle ground between full packages sharing and sandboxes.

Let's look at the (somewhat) equivalent commands for Stack:

```
stack build
```

`build` will compile the project as described above - installing GHC and package dependencies if they are not installed.

When building executables, `stack build` will report where the executable has been created, and it is also possible to find the path to the executable using `stack exec -- which hs-blog-gen`.

```
stack exec hs-blog-gen -- <program arguments>
```

`exec` will run the executable passing the program arguments to our executable.

```
stack ghci hs-blog
```

`ghci` runs `ghci` in the context of our library `hs-blog` - loading the library modules and packages.

```
stack clean
```

`clean` cleans up build artifacts.

The Stack user guide contains more information about how stack works and how to use it effectively.

Build artifacts

Both stack and cabal create build artifacts that we will not want to track using our version control. These build artifacts are found in the `dist`, `dist-newstyle` and `.stack-work` directories. We can add these to a `.gitignore` file (or similar for other version control programs) to ignore them:

```
dist
dist-newstyle
.stack-work
```

Finding packages

Finding packages isn't a very straightforward process at the moment. People have written on how they choose packages, recommendation lists, books, and more.

My suggestion is:

- Search for a tutorial on something you'd like to do, and see which packages come up
- Use the download amount on Hackage as an indication of package popularity
- Use Stackage package synopses to locate a relevant package
- Check social network channels for recommendations, but know that sometimes people tend to recommend inappropriate solutions and packages that might be too complicated or still experimental

It's also important to note the amount of dependencies a package has. Adding many dependencies will affect compilation time and code size. And it can sometimes be a good thing to consider when comparing packages or considering whether a package is needed at all.

Summary

We've created a package description for our library and used `stack` and/or `cabal` to build our program. In future chapters, we'll start adding external packages, we'll only have to add them to the `build-depends` section in the cabal file, and our package manager will download and install the required package for us!

We've made some changes to our project directory, and it should now look like this:

```
.
├── app
│   └── Main.hs
├── hs-blog.cabal
├── LICENSE.txt
├── README.md
├── src
│   ├── HsBlog
│   │   ├── Convert.hs
│   │   ├── Html
│   │   │   └── Internal.hs
│   │   ├── Html.hs
│   │   └── Markup.hs
│   └── HsBlog.hs
└── stack.yaml

4 directories, 10 files
```

Note that this package format could be released on Hackage for other Haskell developers to use!

Fancy options parsing

We'd like to define a nicer interface for our program. While we could manage something ourselves with `getArgs` and pattern matching, it is easier to get good results using a library. We are going to use a package called optparse-applicative.

`optparse-applicative` provides us with an EDSL (yes, another one) to build command arguments parsers. Things like commands, switches, and flags can be built and composed together to make a parser for command-line arguments without actually writing operations on strings as we did when we wrote our Markup parser, and will provide other benefits such as automatic generation of usage lines, help screens, error reporting, and more.

While `optparse-applicative`'s dependency footprint isn't very large, it is likely that a user of our library wouldn't need command-line parsing in this particular case, so it makes sense to add this dependency to the `executable` section (rather than the `library` section) in the `.cabal` file:

```
 executable hs-blog-gen
   import: common-settings
   hs-source-dirs: app
   main-is: Main.hs
   build-depends:
       base
+    , optparse-applicative
     , hs-blog
   ghc-options:
     -O
```

Building a command-line parser

The optparse-applicative package has pretty decent documentation, but we will cover a few important things to pay attention to in this chapter.

In general, there are four important things we need to do:

1. Define our model - we want to define an ADT that describes the various options and commands for our program
2. Define a parser that will produce a value of our model type when run
3. Run the parser on our program arguments input
4. Pattern match on the model and call the right operations according to the options

Define a model

Let's envision our command-line interface for a second; what should it look like?

We want to be able to convert a single file or input stream to either a file or an output stream, or we want to process a whole directory and create a new directory. We can model it in an ADT like this:

```haskell
data Options
  = ConvertSingle SingleInput SingleOutput
  | ConvertDir FilePath FilePath
  deriving Show

data SingleInput
  = Stdin
  | InputFile FilePath
  deriving Show

data SingleOutput
  = Stdout
  | OutputFile FilePath
  deriving Show
```

> Note that we could technically also use `Maybe FilePath` to encode both `SingleInput` and `SingleOutput`, but then we would have to remember what `Nothing` means in each context. By creating a new type with properly named constructors for each option, we make it easier for readers of the code to understand the meaning of our code.

In terms of interface, we could decide that when a user would like to convert a single input source, they would use the `convert` command, and supply the optional flags `--input FILEPATH` and `--output FILEPATH` to read or write from a file. When the user does not supply one or both flags, we will read or write from the standard input/output accordingly.

If the user would like to convert a directory, they can use the `convert-dir` command and supply the two mandatory flags `--input FILEPATH` and `--output FILEPATH`.

Build a parser

This is the most interesting part of the process. How do we build a parser that fits our model?

The `optparse-applicative` library introduces a new type called `Parser`. `Parser`, similar to `Maybe` and `IO`, has the kind `* -> *` - when it is supplied with a saturated (or concrete) type such as `Int`, `Bool` or `Options`, it can become a saturated type (one that has values).

A `Parser a` represents a specification of a command-line options parser that produces a value of type `a` when the command-line arguments are successfully parsed. This is similar to how `IO a` represents a description of a program that can produce a value of type `a`. The main difference between these two types is that while we can't convert an `IO a` to an `a` (we just chain IO operations and have the Haskell runtime execute them), we *can* convert a `Parser a` to a function that takes a list of strings representing the program arguments and produces an `a` if it manages to parse the arguments.

As we've seen with the previous EDSLs, this library uses the *combinator pattern* as well. We need to consider the basic primitives for building a parser and the methods of composing small parsers into bigger parsers.

Let's see an example of a small parser:

```
inp :: Parser FilePath
inp =
  strOption
    ( long "input"
      <> short 'i'
      <> metavar "FILE"
      <> help "Input file"
    )

out :: Parser FilePath
out =
  strOption
    ( long "output"
      <> short 'o'
      <> metavar "FILE"
      <> help "Output file"
    )
```

`strOption` is a parser builder. It is a function that takes a combined *option modifiers* as an argument, and returns a parser that will parse a string. We can specify the type to be `FilePath` because `FilePath` is an alias to `String`. The parser builder describes how to parse the value, and the modifiers describe its properties, such as the flag name, the shorthand of the flag name, and how it would be described in the usage and help messages.

> Actually, `strOption` can return any string type that implements the interface `IsString`. There are a few such types, for example, `Text`, a much more efficient Unicode text type from the `text` package. It is more efficient than `String` because while `String` is implemented as a linked list of `Char`, `Text` is implemented as an array of bytes. `Text` is usually what we should use for text values instead of `String`. We haven't been using it up until now because it is slightly less ergonomic to use than `String`. But it is often the preferred type to use for text!

As you can see, modifiers can be composed using the `<>` function, which means modifiers implement an instance of the `Semigroup` type class!

With such an interface, we don't have to supply all the modifier options, only the relevant ones. So if we don't want to have a shortened flag name, we don't have to add it.

Functor

For the data type we've defined, having `Parser FilePath` takes us a good step in the right direction, but it is not exactly what we need for a `ConvertSingle`. We need a `Parser SingleInput` and a `Parser SingleOutput`. If we had a `FilePath`, we could convert it into `SingleInput` by using the `InputFile` constructor. Remember, `InputFile` is also a function:

```
InputFile :: FilePath -> SingleInput
OutputFile :: FilePath -> SingleOutput
```

However, to convert a parser, we need functions with these types:

```
f :: Parser FilePath -> Parser SingleInput
g :: Parser FilePath -> Parser SingleOutput
```

Fortunately, the `Parser` interface provides us with a function to "lift" a function like `FilePath ->`

`SingleInput` to work on parsers, making it a function with the type `Parser FilePath -> Parser SingleInput`. Of course, this function will work for any input and output, so if we have a function with the type `a -> b`, we can pass it to that function and get a new function of the type `Parser a -> Parser b`.

This function is called `fmap`:

```
fmap :: (a -> b) -> Parser a -> Parser b

-- Or with its infix version
(<$>)  :: (a -> b) -> Parser a -> Parser b
```

We've seen `fmap` before in the interface of other types:

```
fmap :: (a -> b) -> [a] -> [b]

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

`fmap` is a type class function like `<>` and `show`. It belongs to the type class `Functor`:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

And it has the following laws:

```
-- 1. Identity law:
--    if we don't change the values, nothing should change
fmap id = id

-- 2. Composition law:
--    Composing the lifted functions is the same a composing
--    them after fmap
fmap (f . g) == fmap f . fmap g
```

Any type `f` that can implement `fmap` and follow these laws can be a valid instance of `Functor`.

> Notice how `f` has a kind `* -> *`, we can infer the kind of `f` by looking at the other types in the type signature of `fmap`:
>
> 1. `a` and `b` have the kind `*` because they are used as arguments/return types of functions
> 2. `f a` has the kind `*` because it is used as an argument to a function, therefore
> 3. `f` has the kind `* -> *`

Let's choose a data type and see if we can implement a `Functor` instance. We need to choose a data type that has the kind `* -> *`. `Maybe` fits the bill. We need to implement a function `fmap :: (a -> b) -> Maybe a -> Maybe b`. Here's one very simple (and wrong) implementation:

```
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe func maybeX = Nothing
```

Check it out yourself! It compiles successfully! But unfortunately, it does not satisfy the first law. `fmap id = id` means that `mapMaybe id (Just x) == Just x`, however, from the definition, we can clearly see that `mapMaybe id (Just x) == Nothing`.

This is a good example of how Haskell doesn't help us ensure the laws are satisfied, and why they are important. Unlawful `Functor` instances will behave differently from how we'd expect a `Functor` to behave. Let's try again!

```
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe func maybeX =
  case maybeX of
    Nothing -> Nothing
    Just x -> Just (func x)
```

This `mapMaybe` will satisfy the functor laws. This can be proved by doing algebra - if we can do substitution and reach the other side of the equation in each law, then the law holds.

Functor is a very important type class, and many types implement this interface. As we know, `IO`, `Maybe`, `[]` and `Parser` all have the kind `* -> *`, and all allow us to map over their "payload" type.

> Often, people try to look for analogies and metaphors to what a type class means, but type classes with funny names like `Functor` don't usually have an analogy or a metaphor that fits them in all cases. It is easier to give up on the metaphor and think about it as it is - an interface with laws.

We can use `fmap` on `Parser` to make a parser that returns `FilePath` to return a `SingleInput` or `SingleOutput` instead:

```
pInputFile :: Parser SingleInput
pInputFile = fmap InputFile parser
  where
    parser =
      strOption
        ( long "input"
          <> short 'i'
          <> metavar "FILE"
          <> help "Input file"
        )

pOutputFile :: Parser SingleOutput
pOutputFile = OutputFile <$> parser -- fmap and <$> are the same
  where
    parser =
      strOption
        ( long "output"
          <> short 'o'
          <> metavar "FILE"
          <> help "Output file"
        )
```

Applicative

Now that we have two parsers, `pInputFile :: Parser SingleInput` and `pOutputFile :: Parser SingleOutput`, we want to *combine* them as `Options`. Again, if we only had `SingleInput` and `SingleOutput`, we could use the constructor `ConvertSingle`:

```
ConvertSingle :: SingleInput -> SingleOutput -> Options
```

Can we do a similar trick to the one we saw before with `fmap`? Does a function exist that can lift a binary function to work on `Parser`s instead? One with this type signature:

```
???
  :: (SingleInput -> SingleOutput -> Options)
  -> (Parser SingleInput -> Parser SingleOutput -> Parser Options)
```

Yes. This function is called `liftA2`, and it is from the `Applicative` type class. `Applicative` (also known as applicative functor) has three primary functions:

```
class Functor f => Applicative f where
  pure :: a -> f a
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  (<*>) :: f (a -> b) -> f a -> f b
```

`Applicative` is another very popular type class with many instances.

Just like any `Monoid` is a `Semigroup`, any `Applicative` is a `Functor`. This means that any type that wants to implement the `Applicative` interface should also implement the `Functor` interface.

Beyond what a regular functor can do, which is to lift a function over a certain `f`, applicative functors allow us to apply a function to *multiple instances* of a certain `f`, as well as "lift" any value of type `a` into an `f a`.

You should already be familiar with `pure`, we've seen it when we talked about `IO`. For `IO`, `pure` lets us create an `IO` action with a specific return value without doing IO. With `pure` for `Parser`, we can create a `Parser` that, when run, will return a specific value as output without doing any parsing.

`liftA2` and `<*>` are two functions that can be implemented in terms of one another. `<*>` is actually the more useful one between the two. Because when combined with `fmap` (or rather the infix version `<$>`), it can be used to apply a function with many arguments instead of just two.

To combine our two parsers into one, we can use either `liftA2` or a combination of `<$>` and `<*>`:

```
-- with liftA2
pConvertSingle :: Parser Options
pConvertSingle =
  liftA2 ConvertSingle pInputFile pOutputFile

-- with <$> and <*>
pConvertSingle :: Parser Options
pConvertSingle =
  ConvertSingle <$> pInputFile <*> pOutputFile
```

Note that both `<$>` and `<*>` associate to the left, so we have invisible parenthesis that look like this:

```
pConvertSingle :: Parser Options
pConvertSingle =
  (ConvertSingle <$> pInputFile) <*> pOutputFile
```

Let's take a deeper look at the types of the sub-expressions we have here to prove that this type-checks:

```haskell
pConvertSingle :: Parser Options

pInputFile :: Parser SingleInput
pOutputFile :: Parser SingleOutput

ConvertSingle :: SingleInput -> SingleOutput -> Options

(<$>) :: (a -> b) -> Parser a -> Parser b
  -- Specifically, here `a` is `SingleInput`
  -- and `b` is `SingleOutput -> Options`,

ConvertSingle <$> pInputFile :: Parser (SingleOutput -> Options)

(<*>) :: Parser (a -> b) -> Parser a -> Parser b
  -- Specifically, here `a -> b` is `SingleOutput -> Options`
  -- so `a` is `SingleOutput` and `b` is `Options`

-- So we get:
(ConvertSingle <$> pInputFile) <*> pOutputFile :: Parser Options
```

With `<$>` and `<*>` we can chain as many parsers (or any applicative, really) as we want. This is because of two things: currying and parametric polymorphism. Because functions in Haskell take exactly one argument and return exactly one, any multiple-argument function can be represented as `a -> b`.

> You can find the laws for the applicative functors in this article called Typeclassopedia, which talks about various useful type classes and their laws.

Applicative functor is a very important concept and will appear in various parser interfaces (not just for command-line arguments, but also JSON parsers and general parsers), I/O, concurrency, non-determinism, and more. The reason this library is called optparse-applicative is because it uses the `Applicative` interface as the main API for constructing parsers.

Exercise: create a similar interface for the `ConvertDir` constructor of `Options`.

Alternative

One thing we forgot about is that each input and output for `ConvertSingle` could also potentially use the standard input and output instead. Up until now, we only offered one option: reading from or writing to a file by specifying the flags `--input` and `--output`. However, we'd like to make these flags optional, and when they are not specified, use the alternative standard i/o. We can do that by using the function `optional` from `Control.Applicative`:

```haskell
optional :: Alternative f => f a -> f (Maybe a)
```

`optional` works on types which implement instances of the `Alternative` type class:

```haskell
class Applicative f => Alternative f where
  (<|>) :: f a -> f a -> f a
  empty :: f a
```

`Alternative` looks very similar to the `Monoid` type class, but it works on applicative functors. This

type class isn't very common and is mostly used for parsing libraries as far as I know. It provides us with an interface to combine two `Parser`s - if the first one fails to parse, try the other. It also provides other useful functions such as `optional`, which will help us with our case:

```
pSingleInput :: Parser SingleInput
pSingleInput =
  fromMaybe Stdin <$> optional pInputFile

pSingleOutput :: Parser SingleOutput
pSingleOutput =
  fromMaybe Stdout <$> optional pOutputFile
```

Note that with `fromMaybe :: a -> Maybe a -> a` we can extract the `a` out of the `Maybe` by supplying a value for the `Nothing` case.

Now we can use these more appropriate functions in `pConvertSingle` instead:

```
pConvertSingle :: Parser Options
pConvertSingle =
  ConvertSingle <$> pSingleInput <*> pSingleOutput
```

Commands and subparsers

We currently have two possible operations in our interface, convert a single source, or convert a directory. A nice interface for selecting the right operation would be via commands. If the user would like to convert a single source, they can use `convert`, for a directory, `convert-dir`.

We can create a parser with commands with the `subparser` and `command` functions:

```
subparser :: Mod CommandFields a -> Parser a

command :: String -> ParserInfo a -> Mod CommandFields a
```

`subparser` takes *command modifiers* (which can be constructed with the `command` function) as input and produces a `Parser`. `command` takes the command name (in our case, "convert" or "convert-dir") and a `ParserInfo a`, and produces a command modifier. As we've seen before these modifiers have a `Monoid` instance, and they can be composed, meaning that we can append multiple commands to serve as alternatives.

A `ParserInfo a` can be constructed with the `info` function:

```
info :: Parser a -> InfoMod a -> ParserInfo a
```

This function wraps a `Parser` with some additional information such as a helper message, description, and more, so that the program itself, and each sub-command can print some additional information.

Let's see how to construct a `ParserInfo`:

```
pConvertSingleInfo :: ParserInfo Options
pConvertSingleInfo =
  info
    (helper <*> pConvertSingle)
    (progDesc "Convert a single markup source to html")
```

Note that `helper` adds a helper output screen in case the parser fails.

Let's also build a command:

```
pConvertSingleCommand :: Mod CommandFields Options
pConvertSingleCommand =
  command "convert" pConvertSingleInfo
```

Try creating a `Parser Options` combining the two options with `subparser`.

ParserInfo

Since we finished building a parser, we should wrap it up in a `ParserInfo` and add some information to it to make it ready to run:

```
opts :: ParserInfo Options
opts =
  info (helper <*> pOptions)
    ( fullDesc
      <> header "hs-blog-gen - a static blog generator"
      <> progDesc "Convert markup files or directories to html"
    )
```

Running a parser

`optparse-applicative` provides a non-`IO` interface to parse arguments, but the most convenient way to use it is to let it take care of fetching program arguments, try to parse them, and throw errors and help messages in case it fails. This can be done with the function `execParser :: ParserInfo a -> IO a`.

We can place all these options parsing stuff in a new module and then import it from `app/Main.hs`. Let's do that.

Pattern matching on Options

After running the command-line arguments parser, we can pattern match on our model and call the right functions. Currently, our program does not expose this kind of API. So let's go to our `src/HsBlog.hs` module and change the API. We can delete `main` from that file and add two new functions instead:

```
convertSingle :: Html.Title -> Handle -> Handle -> IO ()

convertDirectory :: FilePath -> FilePath -> IO ()
```

`Handle` is an I/O abstraction over file system objects, including `stdin` and `stdout`. Before, we used `writeFile` and `getContents` - these functions either get a `FilePath` to open and work on, or they assume the `Handle` is the standard I/O. We can use the explicit versions that take a `Handle` from `System.IO` instead:

```
convertSingle :: Html.Title -> Handle -> Handle -> IO ()
convertSingle title input output = do
  content <- hGetContents input
  hPutStrLn output (process title content)
```

We will leave `convertDirectory` unimplemented for now and implement it in the next chapter.

In `app/Main.hs`, we will need to pattern match on the `Options` and prepare to call the right functions from `HsBlog`.

Let's look at our full `app/Main.hs` and `src/HsBlog.hs`:

- app/Main.hs

```haskell
-- | Entry point for the hs-blog-gen program

module Main where

import OptParse
import qualified HsBlog

import System.Exit (exitFailure)
import System.Directory (doesFileExist)
import System.IO

main :: IO ()
main = do
  options <- parse
  case options of
    ConvertDir input output ->
      HsBlog.convertDirectory input output

    ConvertSingle input output -> do
      (title, inputHandle) <-
        case input of
          Stdin ->
            pure ("", stdin)
          InputFile file ->
            (,) file <$> openFile file ReadMode

      outputHandle <-
        case output of
          Stdout -> pure stdout
          OutputFile file -> do
            exists <- doesFileExist file
            shouldOpenFile <-
              if exists
                then confirm
                else pure True
            if shouldOpenFile
              then
                openFile file WriteMode
              else
                exitFailure

      HsBlog.convertSingle title inputHandle outputHandle
      hClose inputHandle
      hClose outputHandle

-------------------------------------------------
-- * Utilities

-- | Confirm user action
confirm :: IO Bool
confirm =
  putStrLn "Are you sure? (y/n)" *>
    getLine >>= \answer ->
      case answer of
        "y" -> pure True
        "n" -> pure False
        _ -> putStrLn "Invalid response. use y or n" *>
          confirm
```

- src/HsBlog.hs

```haskell
-- HsBlog.hs
module HsBlog
  ( convertSingle
  , convertDirectory
  , process
  )
  where

import qualified HsBlog.Markup as Markup
import qualified HsBlog.Html as Html
import HsBlog.Convert (convert)

import System.IO

convertSingle :: Html.Title -> Handle -> Handle -> IO ()
convertSingle title input output = do
  content <- hGetContents input
  hPutStrLn output (process title content)

convertDirectory :: FilePath -> FilePath -> IO ()
convertDirectory = error "Not implemented"

process :: Html.Title -> String -> String
process title = Html.render . convert title . Markup.parse
```

We need to make a few small changes to the `.cabal` file.

First, we need to add the dependency `directory` to the `executable`, because we use the library `System.Directory` in `Main`.

Second, we need to list `OptParse` in the list of modules in the `executable`.

```
 executable hs-blog-gen
   import: common-settings
   hs-source-dirs: app
   main-is: Main.hs
+  other-modules:
+    OptParse
   build-depends:
       base
+    , directory
     , optparse-applicative
     , hs-blog
   ghc-options:
     -O
```

Summary

We've learned about a new fancy library called `optparse-applicative` and used it to create a fancier command-line interface in a declarative way. See the result of running `hs-blog-gen --help` (or the equivalent `cabal` / `stack` commands we discussed in the last chapter):

```
hs-blog-gen - a static blog generator

Usage: hs-blog-gen COMMAND
  Convert markup files or directories to html

Available options:
  -h,--help                Show this help text

Available commands:
  convert                  Convert a single markup source to html
  convert-dir              Convert a directory of markup files to html
```

Along the way, we've learned two powerful new abstractions, `Functor` and `Applicative`, as well as revisited an abstraction called `Monoid`. With this library, we've seen another example of the usefulness of these abstractions for constructing APIs and EDSLs.

We will continue to meet these abstractions in the rest of the book.

Bonus exercise: Add another flag named `--replace` to indicate that if the output file or directory already exists, replacing them is okay.

Handling errors and multiple files

We left an unimplemented function last chapter, and there are a few more things left for us to do to actually call our program a static blog generator. We still need to process multiple files in a directory and create an index landing page with links to other pages.

Links in HTML

Our HTML EDSL currently does not support links or other content modifiers such as bold and italics. We should add these so we can use them when creating an index.

Up until now, we've passed `String` to `Structure`, creating functions such as `p_` and `h_`. Instead, we can create and pass them a new type, `Content`, which can be regular text, links, images, and so on.

Exercise: implement what we've just discussed. Follow the compiler errors and refactor what needs refactoring.

Creating an index page

With our extended HTML EDSL, we can now create an index page with links to the other pages.

To create an index page, we need a list of files with their *target destinations*, as well as their `Markup` (so we can extract information to include in our index page, such as the first heading and paragraph). Our output should be an `Html` page.

Exercise: implement the following function:

```
buildIndex :: [(FilePath, Markup.Document)] -> Html.Html
```

Processing directories

Our general strategy for processing whole directories is going to be:

- Create the output directory
- Grab all file names in a directory
- Filter them according to their extension; we want to process the `txt` files and copy other files without modification
- We want to parse each text file, build an index of the result, convert the files to HTML, and write everything to the target directory

While our parsing function can't really fail, trying to read or write a file to the file-system can fail in

several ways. It would be nice if our static blog generator was robust enough that it wouldn't fail completely if one single file gave it some trouble. This is an excellent opportunity to learn about error handling in Haskell, both in uneffectful code and for I/O code.

In the next few chapters, we'll survey the landscape of error handling in Haskell before figuring out the right approach for our use case.

Handling errors with Either

There are quite a few ways to indicate and handle errors in Haskell. We are going to look at one solution: using the type Either. Either is defined like this:

```
data Either a b
  = Left a
  | Right b
```

Simply put, a value of type `Either a b` can contain either a value of type `a`, or a value of type `b`. We can tell them apart from the constructor used.

```
Left True :: Either Bool b
Right 'a' :: Either a Char
```

With this type, we can use the `Left` constructor to indicate failure with some error value attached, and the `Right` constructor with one type to represent success with the expected result.

Since `Either` is polymorphic, we can use any two types to represent failure and success. It is often useful to describe the failure modes using an ADT.

For example, let's say that we want to parse a `Char` as a decimal digit to an `Int`. This operation could fail if the Character is not a digit. We can represent this error as a data type:

```
data ParseDigitError
  = NotADigit Char
  deriving Show
```

And our parsing function can have the type:

```
parseDigit :: Char -> Either ParseDigitError Int
```

Now when we implement our parsing function, we can return `Left` on an error describing the problem, and `Right` with the parsed value on successful parsing:

```
parseDigit :: Char -> Either ParseDigitError Int
parseDigit c =
  case c of
    '0' -> Right 0
    '1' -> Right 1
    '2' -> Right 2
    '3' -> Right 3
    '4' -> Right 4
    '5' -> Right 5
    '6' -> Right 6
    '7' -> Right 7
    '8' -> Right 8
    '9' -> Right 9
    _ -> Left (NotADigit c)
```

`Either a` is also an instance of `Functor` and `Applicative`, so we have some combinators to work with if we want to combine these kind of computations.

For example, if we had three characters and we wanted to try and parse each of them and then find the maximum between them; we could use the applicative interface:

```
max3chars :: Char -> Char -> Char -> Either ParseDigitError Int
max3chars x y z =
  (\a b c -> max a (max b c))
    <$> parseDigit x
    <*> parseDigit y
    <*> parseDigit z
```

The `Functor` and `Applicative` interfaces of `Either a` allow us to apply functions to the payload values and **delay** the error handling to a later phase. Semantically, the first Either in order that returns a `Left` will be the return value. We can see how this works in the implementation of the applicative instance:

```
instance Applicative (Either e) where
    pure          = Right
    Left  e <*> _ = Left e
    Right f <*> r = fmap f r
```

At some point, someone will actually want to **inspect** the result and see if we get an error (with the `Left` constructor) or the expected value (with the `Right` constructor) and they can do that by pattern-matching the result.

Applicative + Traversable

The `Applicative` interface of `Either` is very powerful and can be combined with another abstraction called `Traversable` - for data structures that can be traversed from left to right, like a linked list or a binary tree. With these, we can combine an unspecified amount of values such as `Either ParseDigitError Int`, as long as they are all in a data structure that implements `Traversable`.

Let's see an example:

```
ghci> :t "1234567"
"1234567" :: String
-- remember, a String is an alias for a list of Char
ghci> :info String
type String :: *
type String = [Char]
      -- Defined in ‘GHC.Base’

ghci> :t map parseDigit "1234567"
map parseDigit mystring :: [Either ParseDigitError Int]
ghci> map parseDigit "1234567"
[Right 1,Right 2,Right 3,Right 4,Right 5,Right 6,Right 7]

ghci> :t sequenceA
sequenceA :: (Traversable t, Applicative f) => t (f a) -> f (t a)
-- Substitute `t` with `[]`, and `f` with `Either Error` for a specialized version

ghci> sequenceA (map parseDigit mystring)
Right [1,2,3,4,5,6,7]

ghci> map parseDigit "1a2"
[Right 1,Left (NotADigit 'a'),Right 2]
ghci> sequenceA (map parseDigit "1a2")
Left (NotADigit 'a')
```

The pattern of doing `map` and then `sequenceA` is another function called `traverse`:

```
ghci> :t traverse
traverse
  :: (Traversable t, Applicative f) => (a -> f b) -> t a -> f (t b)
ghci> traverse parseDigit "1234567"
Right [1,2,3,4,5,6,7]
ghci> traverse parseDigit "1a2"
Left (NotADigit 'a')
```

We can use `traverse` on any two types where one implements the `Applicative` interface, like `Either a` or `IO`, and the other implements the `Traversable` interface, like `[]` (linked lists) and `Map k` (also known as a dictionary in other languages - a mapping from keys to values). For example, using `IO` and `Map`. Note that we can construct a `Map` data structure from a list of tuples using the `fromList` function - the first value in the tuple is the key, and the second is the type.

```
ghci> import qualified Data.Map as M -- from the containers package

ghci> file1 = ("output/file1.html", "input/file1.txt")
ghci> file2 = ("output/file2.html", "input/file2.txt")
ghci> file3 = ("output/file3.html", "input/file3.txt")
ghci> files = M.fromList [file1, file2, file3]
ghci> :t files :: M.Map FilePath FilePath -- FilePath is an alias of String
files :: M.Map FilePath FilePath :: M.Map FilePath FilePath

ghci> readFiles = traverse readFile
ghci> :t readFiles
readFiles :: Traversable t => t FilePath -> IO (t String)

ghci> readFiles files
fromList [("output/file1.html","I'm the content of file1.txt\n"),
("output/file2.html","I'm the content of file2.txt\n"),("output/file3.html","I'm the
content of file3.txt\n")]
ghci> :t readFiles files
readFiles files :: IO (Map String String)
```

Above, we created a function `readFiles` that will take a mapping from *output file path* to *input file path* and returns an IO operation that, when run will read the input files and replace their contents right there in the map! Surely this will be useful later.

Multiple errors

Note, since `Either` has the kind `* -> * -> *` (it takes two type parameters) `Either` cannot be an instance of `Functor` or `Applicative`: instances of these type classes must have the kind `* -> *`. Remember that when we look at a type class function signature like:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

And we want to implement it for a specific type (in place of the `f`), we need to be able to *substitute* the `f` with the target type. If we'd try to do it with `Either` we would get:

```
fmap :: (a -> b) -> Either a -> Either b
```

And neither `Either a` or `Either b` are *saturated*, so this won't type check. For the same reason, if

we'll try to substitute `f` with, say, `Int`, we'll get:

```
fmap :: (a -> b) -> Int a -> Int b
```

Which also doesn't make sense.

While we can't use `Either`, we can use `Either e`, which has the kind `* -> *`. Now let's try substituting `f` with `Either e` in this signature:

```
liftA2 :: Applicative => (a -> b -> c) -> f a -> f b -> f c
```

And we'll get:

```
liftA2 :: (a -> b -> c) -> Either e a -> Either e b -> Either e c
```

What this teaches us is that we can only use the applicative interface to combine two *`Either`s with the same type for the `Left` constructor*.

So what can we do if we have two functions that can return different errors? There are a few approaches; the most prominent ones are:

1. Make them return the same error type. Write an ADT that holds all possible error descriptions. This can work in some cases but isn't always ideal. For example, a user calling `parseDigit` shouldn't be forced to handle a possible case that the input might be an empty string
2. Use a specialized error type for each type, and when they are composed together, map the error type of each function to a more general error type. This can be done with the function `first` from the `Bifunctor` type class

Monadic interface

The applicative interface allows us to lift a function to work on multiple `Either` values (or other applicative functor instances such as `IO` and `Parser`). But more often than not, we'd like to use a value from one computation that might return an error in another computation that might return an error.

For example, a compiler such as GHC operates in stages, such as lexical analysis, parsing, type-checking, and so on. Each stage depends on the output of the stage before it, and each stage might fail. We can write the types for these functions:

```
tokenize :: String -> Either Error [Token]

parse :: [Token] -> Either Error AST

typecheck :: AST -> Either Error TypedAST
```

We want to compose these functions so that they work in a chain. The output of `tokenize` goes to `parse`, and the output of `parse` goes to `typecheck`.

We know that we can lift a function over an `Either` (and other functors), we can also lift a function that returns an `Either`:

```
-- reminder the type of fmap
fmap :: Functor f => (a -> b) -> f a -> f b
-- specialized for `Either Error`
fmap :: (a -> b) -> Either Error a -> Either Error b

-- here, `a` is [Token] and `b` is `Either Error AST`:

> fmap parse (tokenize string) :: Either Error (Either Error AST)
```

While this code compiles, it isn't great, because we are building layers of `Either Error`, and we can't use this trick again with `typecheck`! `typecheck` expects an `AST`, but if we try to fmap it on `fmap parse (tokenize string)`, the `a` will be `Either Error AST` instead.

What we would really like is to flatten this structure instead of nesting it. If we look at the kind of values `Either Error (Either Error AST)` could have, it looks something like this:

- `Left <error>`
- `Right (Left error)`
- `Right (Right <ast>)`

Exercise: What if we just used pattern matching for this instead? What would this look like?

Flattening this structure for `Either` is very similar to that last part - the body of the `Right tokens` case:

```
flatten :: Either e (Either e a) -> Either e a
flatten e =
  case e of
    Left l -> Left l
    Right x -> x
```

Because we have this function, we can now use it on the output of `fmap parse (tokenize string) :: Either Error (Either Error AST)` from before:

```
> flatten (fmap parse (tokenize string)) :: Either Error AST
```

And now, we can use this function again to compose with `typecheck`:

```
> flatten (fmap typecheck (flatten (fmap parse (tokenize string)))) :: Either Error
TypedAST
```

This `flatten` + `fmap` combination looks like a recurring pattern which we can combine into a function:

```
flatMap :: (a -> Either e b) -> Either a -> Either b
flatMap func val = flatten (fmap func val)
```

And now, we can write the code this way:

```
> flatMap typecheck (flatMap parse (tokenize string)) :: Either Error TypedAST

-- Or using backticks syntax to convert the function to infix form:
> typecheck `flatMap` parse `flatMap` tokenize string

-- Or create a custom infix operator: (=<<) = flatMap
> typeCheck =<< parse =<< tokenize string
```

This function, `flatten` (and `flatMap` as well), have different names in Haskell. They are called `join` and `=<<` (pronounced "reverse bind"), and they are the essence of another incredibly useful abstraction in Haskell.

If we have a type that can implement:

1. The `Functor` interface, specifically the `fmap` function
2. The `Applicative` interface, most importantly the `pure` function
3. This `join` function

They can implement an instance of the `Monad` type class.

With functors, we were able to "lift" a function to work over the type implementing the functor type class:

```
fmap :: (a -> b) -> f a -> f b
```

With applicative functors we were able to "lift" a function of multiple arguments over multiple values of a type implementing the applicative functor type class, and also lift a value into that type:

```
pure :: a -> f a

liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

With monads we can now flatten (or "join" in Haskell terminology) types that implement the `Monad` interface:

```
join :: m (m a) -> m a

-- this is =<< with the arguments reversed, pronounced "bind"
(>>=) :: m a -> (a -> m b) -> m b
```

With `>>=`, we can write our compilation pipeline from before in a left-to-right manner, which seems to be more popular for monads:

```
> tokenize string >>= parse >>= typecheck
```

We had already met this function before when we talked about `IO`. Yes, `IO` also implements the `Monad` interface. The monadic interface for `IO` helped us with creating a proper ordering of effects.

The essence of the `Monad` interface is the `join` / `>>=` functions, and as we've seen we can implement `>>=` in terms of `join`, we can also implement `join` in terms of `>>=` (try it!).

The monadic interface can mean very different things for different types. For `IO` this is ordering of effects, for `Either` it is early cutoff, for `Logic` this means backtracking computation, etc.

Again, don't worry about analogies and metaphors; focus on the API and the laws.

> Hey, did you check the monad laws? left identity, right identity, and associativity? We've already discussed a type class with exactly these laws - the `Monoid` type class. Maybe this is related to the famous quote about monads being just monoids in something something...

Do notation?

Remember the do notation? It turns out it works for any type that is an instance of `Monad`. How cool is that? Instead of writing:

```
pipeline :: String -> Either Error TypedAST
pipeline string =
  tokenize string >>= \tokens ->
    parse tokens >>= \ast ->
      typecheck ast
```

We can write:

```
pipeline :: String -> Either Error TypedAST
pipeline string = do
  tokens <- tokenize string
  ast <- parse tokens
  typecheck ast
```

And it will work! Still, in this particular case, `tokenize string >>= parse >>= typecheck` is so concise it can only be beaten by using >=> or <=<:

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

-- compare with function composition:
(.) ::              (b ->   c) -> (a ->   b) -> a ->   c
```

```
pipeline  = tokenize >=> parse >=> typecheck
```

or

```
pipeline = typecheck <=< parse <=< tokenize
```

Haskell's ability to create very concise code using abstractions is great once one is familiar with the abstractions. Knowing the monad abstraction, we are now already familiar with the core composition API of many libraries - for example:

- Concurrent and asynchronous programming
- Web programming
- Testing
- Emulating stateful computation
- sharing environment between computations
- and many more.

Summary

Using `Either` for error handling is useful for two reasons:

1. We encode possible errors using types, and we **force users to acknowledge and handle** them, thus making our code more resilient to crashes and bad behaviours
2. The `Functor`, `Applicative`, and `Monad` interfaces provide us with mechanisms for **composing** functions that might fail (almost) effortlessly - reducing boilerplate while maintaining strong guarantees about our code and delaying the need to handle errors until it is appropriate

Either with IO?

When we create `IO` actions that may require I/O, we risk running into all kinds of errors. For example, when we use `writeFile`, we could run out of disk space in the middle of writing, or the file might be write-protected. While these scenarios aren't super common, they are definitely possible.

We could've potentially encoded Haskell functions like `readFile` and `writeFile` as `IO` operations that return `Either`, for example:

```
readFile :: FilePath -> IO (Either ReadFileError String)
writeFile :: FilePath -> String -> IO (Either WriteFileError ())
```

However, there are a couple of issues here; the first is that composing `IO` actions becomes more difficult. Previously we could write:

```
readFile "input.txt" >>= writeFile "output.html"
```

But now the types no longer match - `readFile` will return an `Either ReadFileError String` when executed, but `writeFile` wants to take a `String` as input. We are forced to handle the error before calling `writeFile`.

Composing IO + Either using ExceptT

One way to handle this is by using **monad transformers**. Monad transformers provide a way to stack monad capabilities on top of one another. They are called transformers because **they take a type that has an instance of monad as input and return a new type that implements the monad interface, stacking a new capability on top of it**.

For example, if we want to compose values of a type that is equivalent to `IO (Either Error a)`, using the monadic interface (the function `>>=`), we can use a monad transformer called `ExceptT` and stack it over `IO`. Let's see how `ExceptT` is defined:

```
newtype ExceptT e m a = ExceptT (m (Either e a))
```

Remember, a `newtype` is a new name for an existing type. And if we substitute `e` with `Error` and `m` with `IO`, we get exactly `IO (Either Error a)` as we wanted. And we can convert an `ExceptT Error IO a` into `IO (Either Error a)` using the function `runExceptT`:

```
runExceptT :: ExceptT e m a -> m (Either e a)
```

`ExceptT` implements the monadic interface in a way that combines the capabilities of `Either`, and whatever `m` it takes. Because `ExceptT e m` has a `Monad` instance, a specialized version of `>>=` would look like this:

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

-- Specialized version, replace the `m` above with `ExceptT e m`.
(>>=) :: Monad m => ExceptT e m a -> (a -> ExceptT e m b) -> ExceptT e m b
```

Note that the `m` in the specialized version still needs to be an instance of `Monad`.

Unsure how this works? Try to implement `>>=` for `IO (Either Error a)`:

```
bindExceptT :: IO (Either Error a) -> (a -> IO (Either Error b)) -> IO (Either Error b)
```

Solution

```
bindExceptT :: IO (Either Error a) -> (a -> IO (Either Error b)) -> IO (Either Error b)
bindExceptT mx f = do
  x <- mx -- `x` has the type `Either Error a`
  case x of
    Left err -> pure (Left err)
    Right y -> f y
```

Note that we didn't actually use the implementation details of `Error` or `IO`, `Error` isn't mentioned at all, and for `IO`, we only used the monadic interface with the do notation. We could write the same function with a more generalized type signature:

```
bindExceptT :: Monad m => m (Either e a) -> (a -> m (Either e b)) -> m (Either e b)
bindExceptT mx f = do
  x <- mx -- `x` has the type `Either e a`
  case x of
    Left err -> pure (Left err)
    Right y -> f y
```

And because `newtype ExceptT e m a = ExceptT (m (Either e a))`, we can just pack and unpack that `ExceptT` constructor and get:

```
bindExceptT :: Monad m => ExceptT e m a -> (a -> ExceptT e m b) -> ExceptT e m b
bindExceptT mx f = ExceptT $ do
  -- `runExceptT mx` has the type `m (Either e a)`
  -- `x` has the type `Either e a`
  x <- runExceptT mx
  case x of
    Left err -> pure (Left err)
    Right y -> runExceptT (f y)
```

> Note that when stacking monad transformers, the order in which we stack them matters. With `ExceptT Error IO a`, we have an `IO` operation that, when run will return `Either` an error or a value.

`ExceptT` can enjoy both worlds - we can return error values using the function `throwError`:

```
throwError :: e -> ExceptT e m a
```

and we can "lift" functions that return a value of the underlying monadic type `m` to return a value of `ExceptT e m a` instead:

```
lift :: m a -> ExceptT e m a
```

for example:

```
getLine :: IO String

lift getLine :: ExceptT e IO String
```

> Actually, `lift` is also a type class function from `MonadTrans`, the type class of monad transformers. So technically, `lift getLine :: MonadTrans t => t IO String`, but we are specializing for concreteness.

Now, if we had:

```
readFile :: FilePath -> ExceptT IOError IO String

writeFile :: FilePath -> String -> ExceptT IOError IO ()
```

We could compose them again without issue:

```
readFile "input.txt" >>= writeFile "ouptut.html"
```

But remember - the error type `e` (in both the case `Either` and `Except`) must be the same between composed functions! This means that the type representing errors for both `readFile` and `writeFile` must be the same - that would also force anyone using these functions to handle these errors - should a user who called `writeFile` be required to handle a "file not found" error? Should a user who called `readFile` be required to handle an "out of disk space" error? There are many, many more possible IO errors! "network unreachable", "out of memory", "cancelled thread", we cannot require a user to handle all these errors, or even cover them all in a data type.

So what do we do?

We give up on this approach **for IO code**, and use a different one: Exceptions, as we'll see in the next chapter.

> Note - when we stack `ExceptT` on top of a different type called `Identity` that also implements the `Monad` interface, we get a type that is exactly like `Either` called `Except` (without the `T` at the end). You might sometimes want to use `Except` instead of `Either` because it has a more appropriate name and better API for error handling than `Either`.

Exceptions

The Control.Exception module provides us with the ability to throw exceptions from `IO` code, `catch` Haskell exceptions in `IO` code, and even convert them to `IO (Either ...)` with the function `try`:

```
throwIO :: Exception e => e -> IO a

catch
  :: Exception e
  => IO a         -- The computation to run
  -> (e -> IO a)  -- Handler to invoke if an exception is raised
  -> IO a

try :: Exception e => IO a -> IO (Either e a)
```

The important part of these type signatures is the `Exception` type class. By making a type an instance of the `Exception` type class, we can throw it and catch it in `IO` code:

```
{-# language LambdaCase #-}

import Control.Exception
import System.IO

data MyException
  = ErrZero
  | ErrOdd Int
  deriving Show

instance Exception MyException

sayDiv2 :: Int -> IO ()
sayDiv2 n
  | n == 0 = throwIO ErrZero
  | n `mod` 2 /= 0 = throwIO (ErrOdd n)
  | otherwise = print (n `div` 2)

main :: IO ()
main =
  catch
    ( do
      putStrLn "Going to print a number now."
      sayDiv2 7
      putStrLn "Did you like it?"
    )
    ( \case
      ErrZero ->
        hPutStrLn stderr "Error: we don't support dividing zeroes for some reason"
      ErrOdd n ->
        hPutStrLn stderr ("Error: " <> show n <> " is odd and cannot be divided by 2")
    )
```

Note: we are using two new things here: guards and the `LambdaCase` language extension.

1. Guards, as seen in `sayDiv2` are just a nicer syntax around `if-then-else` expressions. Using guards, we can have multiple `if` branches and finally use the `else` branch by using `otherwise`. After each guard (`|`), there's a condition; after the condition, there's a `=` and then the expression (the part after `then` in an `if` expression)

2. LambdaCase, as seen in `catch`, is just a syntactic sugar to save a few characters, instead of writing `\e -> case e of`, we can write `\case`. It requires enabling the `LambdaCase` extension

   Language extensions

   Haskell is a standardized language. However, GHC provides *extensions* to the language - additional features that aren't covered in the 98 or 2010 standards of Haskell. Features such as syntactic extensions (like LambdaCase above), extensions to the type checker, and more.

   These extensions can be added by adding `{-# language <extension-name> #-}` (the `language` part is case insensitive) to the top of a Haskell source file, or they can be set globally for an entire project by specifying them in the default-extensions section in the `.cabal file`.

   The list of language extensions can be found in the GHC manual, feel free to browse it, but don't worry about trying to memorize all the extensions.

Of course, this example would work much better using `Either` and separating the division and printing à la 'functional core, imperative shell'. But as an example, it works. We created a custom exception and handled it specifically outside an `IO` block. However, we have not handled exceptions that might be raised by `putStrLn`. What if, for example, for some reason, we close the `stdout` handle before this block:

```
main :: IO ()
main = do
  hClose stdout
  catch
    ( do
      putStrLn "Going to print a number now."
      sayDiv2 7
      putStrLn "Did you like it?"
    )
    ( \case
      ErrZero ->
        hPutStrLn stderr "Error: we don't support dividing zeroes for some reason"
      ErrOdd n ->
        hPutStrLn stderr ("Error: " <> show n <> " is odd and cannot be divided by 2")
    )
```

Our program will crash with an error:

```
ghc: <stdout>: hFlush: illegal operation (handle is closed)
```

First, how do we know which exception we should handle? Some functions' documentation include

this, but unfortunately, `putStrLn`'s does not. We could guess from the list of instances the `Exception` type class has; I think `IOException` fits. Now, how can we handle this case as well? We can chain catches:

```
-- need to add these at the top

{-# language ScopedTypeVariables #-}

import GHC.IO.Exception (IOException(..))

main :: IO ()
main = do
  hClose stdout
  catch
    ( catch
      ( do
        putStrLn "Going to print a number now."
        sayDiv2 7
        putStrLn "Did you like it?"
      )
      ( \case
        ErrZero ->
          hPutStrLn stderr "Error: we don't support dividing zeroes for some reason"
        ErrOdd n ->
          hPutStrLn stderr ("Error: " <> show n <> " is odd and cannot be divided by
2")
      )
    )
    ( \(e :: IOException) ->
      -- we can check if the error was an illegal operation on the stderr handle
      if ioe_handle e /= Just stderr && ioe_type e /= IllegalOperation
        then pure () -- we can't write to stderr because it is closed
        else hPutStrLn stderr (displayException e)
    )
```

> We use the `ScopedTypeVariables` to be able to specify types inside let expressions, lambdas, pattern matching, and more.

Or we could use the convenient function `catches` to pass a list of exception handlers:

```
main :: IO ()
main = do
  hClose stdout
  catches
    ( do
      putStrLn "Going to print a number now."
      sayDiv2 7
      putStrLn "Did you like it?"
    )
    [ Handler $ \case
      ErrZero ->
        hPutStrLn stderr "Error: we don't support dividing zeroes for some reason"
      ErrOdd n ->
        hPutStrLn stderr ("Error: " <> show n <> " is odd and cannot be divided by 2")

    , Handler $ \(e :: IOException) ->
      -- we can check if the error was an illegal operation on the stderr handle
      if ioe_handle e /= Just stderr && ioe_type e /= IllegalOperation
        then pure () -- we can't write to stderr because it is closed
        else hPutStrLn stderr (displayException e)
    ]
```

As an aside, `Handler` uses a concept called existentially quantified types to hide inside it a function that takes an arbitrary type that implements `Exception`. This is why we can encode a seemingly heterogeneous list of functions that handle exceptions for `catches` to take as input. This pattern is rarely useful, but I've included it here to avoid confusion.

And if we wanted to catch any exception, we'd catch `SomeException`:

```
main :: IO ()
main = do
  hClose stdout
  catch
    ( do
      putStrLn "Going to print a number now."
      sayDiv2 7
      putStrLn "Did you like it?"
    )
    ( \(SomeException e) ->
      hPutStrLn stderr (show e)
    )
```

This could also go in `catches` as the last element in the list if we wanted specialized handling for other scenarios.

A couple more functions worth knowing are `bracket` and `finally`. These functions can help us handle resource acquisition more safely when errors are present.

In our `main` in the `app/Main.hs` file, we do a small ritual of opening and closing handles. Are there scenarios where we would clean-up after ourselves (meaning, close handles we've opened)? Which parts of the code could throw an exception? Which handles won't get closed?

- Try to use `bracket` to make sure we always close a handle afterward, even if an exception is thrown, and avoid closing the handle for the `stdin` and `stdout` cases.
- How can we avoid duplicating the `outputHandle` code for the `Stdin` and `InputFile` branches?

There's action a custom function that does a similar thing to `bracket (openFile file <mode>) hClose`, it's called withFile. Keep an eye out for functions that start with the prefix `with`; they are probably using the same pattern of continuation-passing style.

Summary

Exceptions are useful and often necessary when we work with `IO` and want to make sure our program is handling errors gracefully. They have an advantage over `Either` in that we can easily compose functions that may throw errors of different types, but also have a disadvantage of not encoding types as return values, and therefore does not force us to handle them.

For Haskell, the language designers have made a choice for us by designing `IO` to use exceptions instead of `Either`. And this is what I would recommend for handling your own effectful computations. However, I think that `Either` is more appropriate for uneffectful code, because it forces us to acknowledge and handle errors (eventually), thus making our programs more robust. And also because we can only catch exceptions in `IO` code.

Let's code already!

This was a long info dump. Let's practice what we've learned. We want to:

- Create the output directory
- Grab all file names in a directory
- Filter them according to their extension
- Process .txt files
- Copy other files without modification
- Parse each text file, build an index of the result, convert the files to HTML, and write everything to the target directory

> Note: I did not write this code immediately in the final form it was presented. It was an iterative process of writing code, refactoring, splitting functions, changing type signatures, and more. When solving a coding problem, start small and simple, do the thing that works, and refactor it when it makes sense and makes the code clearer and more modular. In Haskell, we pride ourselves in our ability to refactor code and improve it over time, and that principle holds when writing new software as well!

New module

Let's create a new module, `HsBlog.Directory`, which will be responsible for handling directories and multiple files. From this module, we will export the `convertDirectory` and `buildIndex` functions we've defined before:

```
-- | Process multiple files and convert directories

module HsBlog.Directory
  ( convertDirectory
  , buildIndex
  )
  where
```

In this module, we are going to use the directory and filepath libraries to manipulate directories, files, and filepaths. We'll use the new abstractions we've learned, `Traversable` and `Monad`, and the concepts and types we've learned about: `Either`, `IO`, and exceptions.

For all of that, we need quite a few imports:

```
import qualified HsBlog.Markup as Markup
import qualified HsBlog.Html as Html
import HsBlog.Convert (convert, convertStructure)

import Data.List (partition)
import Data.Traversable (for)
import Control.Monad (void, when)

import System.IO (hPutStrLn, stderr)
import Control.Exception (catch, displayException, SomeException(..))
import System.Exit (exitFailure)
import System.FilePath
  ( takeExtension
  , takeBaseName
  , (<.>)
  , (</>)
  , takeFileName
  )
import System.Directory
  ( createDirectory
  , removeDirectoryRecursive
  , listDirectory
  , doesDirectoryExist
  , copyFile
  )
```

If you are unsure what a specific function we're using does, look it up at Hoogle, read the type signature and the documentation, and play around with it in `ghci`.

Converting a directory

We can start by describing the high-level function `convertDirectory`, which encapsulates many smaller functions, each responsible for doing a specific thing. `convertDirectory` is quite imperative looking, and looks like a different way to describe the steps of completing our task:

```
-- | Copy files from one directory to another, converting '.txt' files to
--   '.html' files in the process. Recording unsuccessful reads and writes to stderr.
--
-- May throw an exception on output directory creation.
convertDirectory :: FilePath -> FilePath -> IO ()
convertDirectory inputDir outputDir = do
  DirContents filesToProcess filesToCopy <- getDirFilesAndContent inputDir
  createOutputDirectoryOrExit outputDir
  let
    outputHtmls = txtsToRenderedHtml filesToProcess
  copyFiles outputDir filesToCopy
  writeFiles outputDir outputHtmls
  putStrLn "Done."
```

Here we trust that each `IO` function handles errors responsibly, and terminates the project when necessary.

Let's examine the steps in order.

getDirFilesAndContent

```haskell
-- | The relevant directory content for our application
data DirContents
  = DirContents
    { dcFilesToProcess :: [(FilePath, String)]
      -- ^ File paths and their content
    , dcFilesToCopy :: [FilePath]
      -- ^ Other file paths, to be copied directly
    }

-- | Returns the directory content
getDirFilesAndContent :: FilePath -> IO DirContents
```

`getDirFilesAndContent` is responsible for providing the relevant files for processing -- both the ones we need to convert to markup (and their textual content) and other files we might want to copy as-is (such as images and style-sheets):

```haskell
-- | Returns the directory content
getDirFilesAndContent :: FilePath -> IO DirContents
getDirFilesAndContent inputDir = do
  files <- map (inputDir </>) <$> listDirectory inputDir
  let
    (txtFiles, otherFiles) =
      partition ((== ".txt") . takeExtension) files
  txtFilesAndContent <-
    applyIoOnList readFile txtFiles >>= filterAndReportFailures
  pure $ DirContents
    { dcFilesToProcess = txtFilesAndContent
    , dcFilesToCopy = otherFiles
    }
```

This function does 4 important things:

1. Lists all the files in the directory
2. Splits the files into 2 groups according to their file extension
3. Reads the contents of the .txt files and reports when files fail to be read
4. Returns the results. We've defined a data type to make the result content more obvious

Part (3) is a little bit more involved than the rest; let's explore it.

applyIoOnList

`applyIoOnList` has the following type signature:

```haskell
applyIoOnList :: (a -> IO b) -> [a] -> IO [(a, Either String b)]
```

It tries to apply an `IO` function on a list of values and document successes and failures.

Try to implement it! If you need a hint for which functions to use, see the import list we wrote earlier.

`applyIoOnList` is a higher-order function that applies a particular `IO` function (in our case, `readFile`) on a list of things (in our case, `FilePath`s). For each thing, it returns the thing itself along with the result of applying the `IO` function as an `Either`, where the `Left` side is a `String` representation of an error if one occurred.

Notice how much the type of this function tells us about what it might do. Because the types are polymorphic, there is nothing else to do with the `a`s other than apply them to the function, and nowhere to generate `b` from other than the result of the function.

> Note: when I first wrote this function, it was specialized to work only on `readFile`, take specifically `[FilePath]` and return `IO [(FilePath, Either String String)]`. But after running into other use cases where I could use it (`writeFiles` and `copyFiles`) I refactored out the `action`, the input type, and the return type.

This function uses exceptions to catch any error that might be thrown and encodes both the failure and success cases in the type system using `Either`, delaying the handling of exceptions to the function caller while making sure it won't be forgotten!

Next, let's look at the function that handles the errors by reporting and then filtering out all the cases that failed.

filterAndReportFailures

`filterAndReportFailures` has the following type signature:

```
filterAndReportFailures :: [(a, Either String b)] -> IO [(a, b)]
```

It filters out unsuccessful operations on files and reports errors to the stderr.

Try to implement it!

Solution

```
-- | Filter out unsuccessful operations on files and report errors to stderr.
filterAndReportFailures :: [(a, Either String b)] -> IO [(a, b)]
filterAndReportFailures =
  foldMap $ \(file, contentOrErr) ->
    case contentOrErr of
      Left err -> do
        hPutStrLn stderr err
        pure []
      Right content ->
        pure [(file, content)]
```

This code may seem a bit surprising - how come we can use `foldMap` here? Reminder, the type of `foldMap` is:

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

If we specialize this function for our use case, substituting the general type with the types we are using, we learn that `IO [(a, b)]` is a monoid. And indeed - `[a]` is a monoid for any `a` with `[]` (the empty list) as `mempty` and `++` as `<>`, but also `IO a` is a monoid for any `a` that is itself a monoid with `pure mempty` as `mempty` and `liftA2 (<>)` as `<>`!

Using these instances, we can `map` over the content, handle errors, and return an empty list to filter

out a failed case, or a singleton list to keep the result. And the `fold` in `foldMap` will concatenate the resulting list where we return all of the successful cases!

If you've written this in a different way that does the same thing, that's fine too! It's just nice to see how sometimes abstractions can be used to write concise code.

These functions are responsible for fetching the right information. Next, let's look at the code for creating a new directory.

createOutputDirectoryOrExit

```
-- | Creates an output directory or terminates the program
createOutputDirectoryOrExit :: FilePath -> IO ()
createOutputDirectoryOrExit outputDir =
  whenIO
    (not <$> createOutputDirectory outputDir)
    (hPutStrLn stderr "Cancelled." *> exitFailure)

-- | Creates the output directory.
--   Returns whether the directory was created or not.
createOutputDirectory :: FilePath -> IO Bool
createOutputDirectory dir = do
  dirExists <- doesDirectoryExist dir
  create <-
    if dirExists
      then do
        override <- confirm "Output directory exists. Override?"
        when override (removeDirectoryRecursive dir)
        pure override
      else
        pure True
  when create (createDirectory dir)
  pure create
```

`createOutputDirectoryOrExit` itself is not terribly exciting; it does what it is named -- it tries to create the output directory and exits the program in case it didn't succeed.

`createOutputDirectory` is the function that actually does the heavy lifting. It checks if the directory already exists, and checks if the user would like to override it. If they do, we remove it and create the new directory; if they don't, we do nothing and report their decision.

txtsToRenderedHtml

```
let
  outputHtmls = txtsToRenderedHtml filesToProcess
```

In this part of the code, we convert files to markup and change the input file paths to their respective output file paths (`.txt` -> `.html`). We then build the index page and convert everything to HTML.

Implement `txtsToRenderedHtml`, which has the following type signature:

```
txtsToRenderedHtml :: [(FilePath, String)] -> [(FilePath, String)]
```

Hint

I implemented this by defining three functions:

```
txtsToRenderedHtml :: [(FilePath, String)] -> [(FilePath, String)]

toOutputMarkupFile :: (FilePath, String) -> (FilePath, Markup.Document)

convertFile :: (FilePath, Markup.Document) -> (FilePath, Html.Html)
```

---

copyFiles and writeFiles

The only thing left to do is to write the directory content, after the processing is completed, to the newly created directory:

```
-- | Copy files to a directory, recording errors to stderr.
copyFiles :: FilePath -> [FilePath] -> IO ()
copyFiles outputDir files = do
  let
    copyFromTo file = copyFile file (outputDir </> takeFileName file)
  void $ applyIoOnList copyFromTo files >>= filterAndReportFailures
```

Here we use `applyIoOnList` again to do something a bit more complicated, instead of reading from a file, it copies from the input path to a newly generated output path. Then we pass the result (which has the type `[(FilePath, Either String ())]`) to `filterAndReportFailures` to print the errors and filter out the unsuccessful copies. Because we are not really interested in the output of `filterAndReportFailures`, we discard it with `void`, returning `()` as a result instead:

```
-- | Write files to a directory, recording errors to stderr.
writeFiles :: FilePath -> [(FilePath, String)] -> IO ()
writeFiles outputDir files = do
  let
    writeFileContent (file, content) =
      writeFile (outputDir </> file) content
  void $ applyIoOnList writeFileContent files >>= filterAndReportFailures
```

Once again, this code looks almost exactly like `copyFiles`, but the types are different. Haskell's combination of parametric polymorphism + type class for abstractions is really powerful, and has helped us reduce quite a bit of code.

This pattern of using `applyIoOnList` and then `filterAndReportFailures` happens more than once. It might be a good candidate for refactoring. Try it! What do you think about the resulting code? Is it easier or more difficult to understand? Is it more modular or less? What are the pros and cons?

Summary

With that, we have completed our `HsBlog.Directory` module, which is responsible for converting a directory safely. Note that the code could probably be simplified quite a bit if we were fine with errors crashing the entire program altogether, but sometimes this is the price we pay for

robustness. It is up to you to choose what you can live with and what not, but I hope this saga has taught you how to approach error handling in Haskell in case you need to.

Summary

This was quite a section. Let's recount the things we've learned.

We discussed several ways to handle errors in Haskell:

1. Encoding errors as a data type and using the `Either` type to encode "a value or an error". Useful approach for uneffectful code
2. Using `ExceptT` when we want to combine the approach in (1) on top of an existing type with monadic capabilities
3. Using exceptions for IO code

We've also learned a few new abstractions and techniques:

1. The `Traversable` type class, for data structures that can be traversed from left to right, such as linked lists, binary trees and `Map`s. Pretty useful when combined with another applicative functor type like `Either` or `IO`
2. The `Monad` type class extends the `Applicative` type class with the `join :: m (m a) -> m a` function. We learned that `Either` implements this type class interface, and so does `IO`
3. The `MonadTrans` type class for *monad transformers* for types that take other monads as inputs and provide a monadic interface (`>>=`, do notation, etc.) while combining both their capabilities. We saw how to stack an `Either`-like monad transformer, `ExceptT`, on top of `IO`

We are almost done – only a couple more things left to do with this project. Let's go!

Passing environment variables

We'd like to add some sort of an environment to keep general information on the blog for various processings, such as the blog name, stylesheet location, and so on.

Environment

We can represent our environment as a record data type and build it from user input. The user input can be from command-line arguments, a configuration file, or something else:

```
module HsBlog.Env where

data Env
  = Env
    { eBlogName :: String
    , eStylesheetPath :: FilePath
    }
  deriving Show

defaultEnv :: Env
defaultEnv = Env "My Blog" "style.css"
```

After filling this record with the requested information, we can pass it as input to any function that might need it. This is a simple approach that can definitely work for small projects. But sometimes, when the project gets bigger, and many nested functions need the same information, threading the environment can get tedious.

There is an alternative solution to threading the environment as input to functions, and that is using the `ReaderT` type from the `mtl` (or `transformers`) package.

ReaderT

```
newtype ReaderT r m a = ReaderT (r -> m a)
```

`ReaderT` is another *monad transformer* like `ExceptT`, which means that it also has an instance of `Functor`, `Applicative`, `Monad`, and `MonadTrans`.

As we can see in the definition, `ReaderT` is *a newtype* over a function that takes some value of type `r`, and returns a value of type `m a`. The `r` usually represents the environment we want to share between functions we want to compose, and the `m a` represents the underlying result we return. The `m` could be any type that implements `Monad` that we are familiar with. Usually, it goes well with `IO` or `Identity`, depending on if we want to share an environment between effectful or uneffectful computations.

`ReaderT` *carries* a value of type `r` and passes it around to other functions when we use the `Applicative` and `Monad` interfaces so that we don't have to pass the value around manually. And when we want to grab the `r` and use it, all we have to do is `ask`.

For our case, this means that instead of passing around `Env`, we can instead convert our functions to use `ReaderT` - those that are uneffectful and don't use `IO` can return `ReaderT Env Identity a`

instead of `a` (or the simplified version, `Reader Env a`), and those that are effectful can return `ReaderT Env IO a` instead of `IO a`.

Note, as we've said before, `Functor`, `Applicative`, and `Monad` all expect the type that implements their interfaces to have the kind `* -> *`. This means that it is `ReaderT r m` which implements these interfaces, and when we compose functions with `<*>` or `>>=` we replace the `f` or `m` in their type signature with `ReaderT r m`.

This means that as with `Either e` when we had composed functions with the same error type, so it is with `ReaderT r m` - we have to compose functions with the same `r` type and the same `m` type, we can't mix different environment types or different underlying `m` types.

We're going to use a specialized version of `ReaderT` that uses a specific `m` = `Identity` called `Reader`. The `Control.Monad.Reader` provides an alias: `Reader r a = ReaderT r Identity a`.

If the idea behind `ReaderT` is still a bit fuzzy to you and you want to get a better understanding of how `ReaderT` works, try doing the following exercise:

1. Choose an `Applicative` or `Monad` interface function; I recommend `liftA2`, and specialize its type signature by replacing `f` (or `m`) with a concrete `ReaderT` type such as `ReaderT Int IO`
2. Unpack the `ReaderT` newtype, replacing `ReaderT Int IO t` with `Int -> IO t`
3. Implement this specialized version of the function you've chosen

Solutions

- Solution for liftA2

```
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
```

- Solution for (1)

```
-- Specialize: replace `f` with `ReaderT Env IO`
liftA2 :: (a -> b -> c) -> ReaderT Env IO a -> ReaderT Env IO b -> ReaderT Env IO
c
```

- Solution for (2)

```
-- Unpack the newtype, replacing `ReaderT Env IO a` with `Env -> IO a`
liftA2 :: (a -> b -> c) -> (Env -> IO a) -> (Env -> IO b) -> (Env -> IO c)
```

- Solution for (3)

```
specialLiftA2 :: (a -> b -> c) -> (Env -> IO a) -> (Env -> IO b) -> (Env -> IO c)
specialLiftA2 combine funcA funcB env =
  liftA2 combine (funcA env) (funcB env)
```

Notice how the job of our special `liftA2` for `ReaderT` is to supply the two functions with `env` and then use the `liftA2` implementation of the underlying `m` type (in our case, `IO`) to do the rest of the work. Does it look like we're adding a capability on top of a different `m`? That's the idea behind monad transformers.

How to use Reader

Defining a function

Instead of defining a function like this:

```
txtsToRenderedHtml :: Env -> [(FilePath, String)] -> [(FilePath, String)]
```

We define it like this:

```
txtsToRenderedHtml :: [(FilePath, String)] -> Reader Env [(FilePath, String)]
```

Now that our code uses `Reader`, we have to accommodate that in the way we write our functions.

Before:

```
txtsToRenderedHtml :: Env -> [(FilePath, String)] -> [(FilePath, String)]
txtsToRenderedHtml env txtFiles =
 let
   txtOutputFiles = map toOutputMarkupFile txtFiles
   index = ("index.html", buildIndex env txtOutputFiles)
   htmlPages = map (convertFile env) txtOutputFiles
 in
   map (fmap Html.render) (index : htmlPages)
```

Note how we needed to thread the `env` to the other functions that use it.

After:

```
txtsToRenderedHtml :: [(FilePath, String)] -> Reader Env [(FilePath, String)]
txtsToRenderedHtml txtFiles = do
  let
    txtOutputFiles = map toOutputMarkupFile txtFiles
  index <- (,) "index.html" <$> buildIndex txtOutputFiles
  htmlPages <- traverse convertFile txtOutputFiles
  pure $ map (fmap Html.render) (index : htmlPages)
```

Note how we use *do notation* now, and *instead of threading* `env` around we *compose* the relevant functions, `buildIndex` and `convertFile`, we use the type classes interfaces to compose the functions. Note how we needed to `fmap` over `buildIndex` to add the output file we needed with the tuple and how we needed to use `traverse` instead of `map` to compose the various `Reader` values `convertFile` will produce.

Extracting Env

When we want to use our `Env`, we need to *extract* it from the `Reader`. We can do it with:

```
ask :: ReaderT r m r
```

Which yanks the `r` from the `Reader` - we can extract with `>>=` or `<-` in do notation. See the comparison:

Before:

```
convertFile :: Env -> (FilePath, Markup.Document) -> (FilePath, Html.Html)
convertFile env (file, doc) =
  (file, convert env (takeBaseName file) doc)
```

After:

```
convertFile :: (FilePath, Markup.Document) -> Reader Env (FilePath, Html.Html)
convertFile (file, doc) = do
  env <- ask
  pure (file, convert env (takeBaseName file) doc)
```

> Note: we didn't change `convert` to use `Reader` because it is a user-facing API for our library.

> By providing a simpler interface, we allow more users to use our library - even those that aren't yet familiar with monad transformers.
>
> Providing a simple function argument passing interface is preferred in this case.

Run a Reader

Similar to handling the errors with `Either`, at some point, we need to supply the environment to a computation that uses `Reader` and extract the result from the computation. We can do that with the functions `runReader` and `runReaderT`:

```
runReader :: Reader r a -> (r -> a)

runReaderT :: ReaderT r m a -> (r -> m a)
```

These functions convert a `Reader` or `ReaderT` to a function that takes `r`. Then we can pass the initial environment to that function:

```
convertDirectory :: Env -> FilePath -> FilePath -> IO ()
convertDirectory env inputDir outputDir = do
  DirContents filesToProcess filesToCopy <- getDirFilesAndContent inputDir
  createOutputDirectoryOrExit outputDir
  let
    outputHtmls = runReader (txtsToRenderedHtml filesToProcess) env
  copyFiles outputDir filesToCopy
  writeFiles outputDir outputHtmls
  putStrLn "Done."
```

See the `let outputHtmls` part.

Extra: Transforming Env for a particular call

Sometimes we may want to modify the `Env` we pass to a particular function call. For example, we may have a general `Env` type that contains a lot of information and functions that only need a part of that information.

If the functions we are calling are like `convert` and take the environment as an argument instead of a `Reader`, we can just extract the environment with `ask`, apply a function to the extracted environment, and pass the result to the function like this:

```
outer :: Reader BigEnv MyResult
outer = do
  env <- ask
  pure (inner (extractSmallEnv env))

inner :: SmallEnv -> MyResult
inner = ...

extractSmallEnv :: BigEnv -> SmallEnv
extractSmallEnv = ...
```

But if `inner` uses a `Reader SmallEnv` instead of argument passing, we can use `runReader` to *convert `inner` to a normal function*, and use the same idea as above!

```haskell
outer :: Reader BigEnv MyResult
outer = do
  env <- ask
  -- Here the type of `runReader inner` is `SmallEnv -> MyResult`
  pure (runReader inner (extractSmallEnv env))

inner :: Reader SmallEnv MyResult
inner = ...

extractSmallEnv :: BigEnv -> SmallEnv
extractSmallEnv = ...
```

This pattern is generalized and captured by a function called withReaderT, and works even for `ReaderT`:

```haskell
withReaderT :: (env2 -> env1) -> ReaderT env1 m a -> ReaderT env2 m a
```

`withReaderT` takes a function that modifies the environment, and converts a `ReaderT env1 m a` computation to a `ReaderT env2 m a` computation using this function.

Let's see it concretely with our example:

```haskell
outer :: Reader BigEnv MyResult
outer = withReaderT extractSmallEnv inner
```

Question: what is the type of `withReaderT` when specialized in our case?

Note the order of the environments! We use a function from a `BigEnv` to a `SmallEnv`, to convert a `Reader` of `SmallEnv` to a `Reader` of `BigEnv`!

This is because we are mapping over the *input* of a function rather than the *output*, and is related to topics like variance and covariance, but it isn't terribly important for us at the moment.

Using Env in our logic code

One thing we haven't talked about yet is using our environment in the `convert` function to generate the pages we want. And actually, we don't even have the ability to add stylesheets to our HTML EDSL at the moment! We need to go back and extend it. Let's do all that now:

Since stylesheets go in the `head` element, perhaps it's a good idea to create an additional `newtype` like `Structure` for `head` information? Things like title, stylesheet, and even meta elements can be composed together just like we did for `Structure` to build the `head`!

1. Do it now: extend our HTML library to include headings and add 3 functions: `title_` for titles, `stylesheet_` for stylesheets, and `meta_` for meta elements like twitter cards.
2. Fix `convert` and `buildIndex` to use the new API. Note: `buildIndex` should return `Reader`!
3. Create a command-line parser for `Env`, attach it to the `convert-dir` command, and pass the result to the `convertDirectory` function.

Summary

Which version do you like better? Manually passing arguments, or using `Reader`?

To me, it is not clear that the second version with `Reader` is better than the first with explicit argument passing in our particular case.

Using `Reader` and `ReaderT` makes our code a little less friendly toward beginners that are not yet familiar with these concepts and techniques, and we don't see (in this case) many benefits.

As programs grow larger, techniques like `Reader` become more attractive. For our relatively small example, using `Reader` might not be appropriate. I've included it in this book because it is an important technique to have in our arsenal, and I wanted to demonstrate it.

It is important to weigh the benefits and costs of using advanced techniques, and it's often better to try and get away with simpler techniques if possible.

Testing

We want to add some tests to our blog generator. At the very least a few regression tests to make sure that if we extend or change our markup parsing code, HTML generation code, or translation from markup to HTML code, and make a mistake, we'll have a safety net alerting us of issues.

We will use the Hspec testing framework to write our tests. There are other testing frameworks in Haskell, for example, tasty, but I like Hspec's documentation, so we'll use that.

Initial setup

Cabal file additions

We're going to define a new section in our `hs-blog-gen.cabal` file for our new test suite. This section is called `test-suite` and is fairly similar to the `library` and `executable` sections.

The interfaces for how to define a test suite are described in the Cabal documentation. We are going to use the `exitcode-stdio-1.0` interface. Let's go over the different settings and options:

```
test-suite hs-blog-gen-test
  import: common-settings
  type: exitcode-stdio-1.0
  hs-source-dirs: test
  main-is: Spec.hs

  -- other-modules:
  build-depends:
      base
    , hspec
    , hspec-discover
    , raw-strings-qq
    , hs-blog
  ghc-options:
    -O -threaded -rtsopts -with-rtsopts=-N
  build-tool-depends:
    hspec-discover:hspec-discover
```

- `hs-source-dirs: test` - The directory of the source files for the test suite
- `main-is: Spec.hs` - The entry point to the test suite
- `other-modules` - The modules in our test suite. Currently commented out because we haven't added any yet
- `build-depends` - The packages we are going to use:
  - `base` - The standard library for Haskell, as we've used before
  - `hspec` - The test framework we are going to use
  - `hspec-discover` - Automatic discovery of Hspec tests
  - `raw-strings-qq` - Additional syntax for writing raw string literals
  - `hs-blog` - Our library
- `ghc-options` - Extra options and flags for GHC:
  - `-O` - Compile with optimizations
  - `-threaded` - Use the multi-core runtime instead of the single-core runtime. The multi-

core runtime is generally a bit slower in my experience, but when writing code that actually uses multiple cores (such as a test framework that runs tests in parallel), it can give a good performance boost
    - `-rtsopts` - Let us configure the Haskell runtime system by passing command-line arguments to our application
    - `-with-rtsopts=-N` - Set specific default options for the program at link-time. Specifically, `-N` Sets the number of cores to use in our program
- `build-tool-depends` - Use a specific executable from a package dependency in aid of building the package. In this case, we are using the `hspec-discover` executable from the `hspec-discover` package, which goes over the source directory for the tests, finds all of the `Spec` files and creates an entry point for the program that will run all the tests it discovered

Hspec discovery

For `hspec-discover` to work, we need to add the following to the "main" file of the test suite, for us, this is `test/Spec.hs`:

```
{-# OPTIONS_GHC -F -pgmF hspec-discover #-}
```

That's it! `hspec-discover` will automatically define a `main` for us. Now we can run the tests using `stack test` or `cabal test` (your choice). Because we haven't defined any tests, our output is:

```
Finished in 0.0000 seconds
0 examples, 0 failures
```

When we add new Hspec tests, `hspec-discover` will find and run them automatically (though we will still need to add them to the `other-modules` section in the cabal file).

For `hspec-discover` to identify modules as test modules, the modules must follow a convention:

1. Their module names must end with `Spec`
2. They must define a value `spec :: Spec` (which describes the test) and export it outside of the module (by adding it to the export list of the module, for example)

Writing tests

Let's write our first test. We'll create a new module to test markup parsing. We'll call it `MarkupParsingSpec.hs`. We'll need the following imports as well:

```
module MarkupParsingSpec where

import Test.Hspec
import HsBlog.Markup
```

`Hspec` provides us with a monadic interface for describing, composing and nesting test specifications (`Spec`s).

Using the `describe` function, we can describe a group of tests; using the `it` function, we can add a new test, and using a function like `shouldBe`, we can compare two values and make sure they are

equal by using their `Eq` instance. If they are, the test will pass; if not, it will fail with a descriptive error.

Let's try it and write a test that obviously fails!

```
spec :: Spec
spec = do
  describe "Markup parsing tests" $ do
    it "empty" $
      shouldBe
        (parse "")
        [Heading 1 "bug"]
```

After adding the module to the `other-modules` list in the cabal file:

```
  other-modules:
    MarkupParsingSpec
```

And running the tests, we get this output:

```
MarkupParsing
  Markup parsing tests
    empty FAILED [1]

Failures:

  test/MarkupParsingSpec.hs:10:7:
  1) MarkupParsing, Markup parsing tests, empty
       expected: [Heading 1 "bug"]
        but got: []

  To rerun use: --match "/MarkupParsing/Markup parsing tests/empty/"

Randomized with seed 763489823

Finished in 0.0004 seconds
1 example, 1 failure
```

The output describes which tests are running in a hierarchy tree (module, group, and test), whether the tests pass or fail, and if they fail, the output and the expected output.

We can fix our test by matching the expected output:

```
      shouldBe
        (parse "")
        []
```

Now, running the tests will produce:

```
MarkupParsing
  Markup parsing tests
    empty

Finished in 0.0001 seconds
1 example, 0 failures
```

We can add a few more tests:

```
    it "paragraph" $
      shouldBe
        (parse "hello world")
        [Paragraph "hello world"]

    it "heading 1" $
      shouldBe
        (parse "* Heading 1")
        [Heading 1 "Heading 1"]

    it "code" $
      shouldBe
        (parse "> main = putStrLn \"hello world!\"")
        [CodeBlock ["main = putStrLn \"hello world!\""]]
```

And run the tests again:

```
MarkupParsing
  Markup parsing tests
    Test empty
    paragraph
    heading 1
    code

Finished in 0.0003 seconds
4 examples, 0 failures
```

This is the gist of writing unit tests with Hspec. It's important to note that we can nest `Spec` s that are declared with `describe` to create trees, and, of course, refactor and move things to different functions and modules to make our test suite better organized.

For example, we can write our tests like this:

```
spec :: Spec
spec = do
  describe "Markup parsing tests" $ do
    simple

simple :: Spec
simple = do
  describe "simple" $ do
    it "empty" $
      shouldBe
        (parse "")
        []

    it "paragraph" $
      shouldBe
        (parse "hello world")
        [Paragraph "hello world"]

    it "heading 1" $
      shouldBe
        (parse "* Heading 1")
        [Heading 1 "Heading 1"]

    it "code" $
      shouldBe
        (parse "> main = putStrLn \"hello world!\"")
        [CodeBlock ["main = putStrLn \"hello world!\""]]
```

Also, there are other "expectations" like `shouldBe` that we can use when writing tests. They are described in the Hspec tutorial and can be found in the haddock documentation as well.

Raw strings

If we want to write multi-line strings or avoid escaping strings as we did in the "code" test, we can use a library called raw-strings-qq which uses a language extension called `QuasiQuotes`. `QuasiQuotes` is a meta-programming extension that provides a mechanism for extending the syntax of Haskell.

A quasi-quote has the form `[quoter| string |]`, where the quoter is the name of the function providing the syntax we wish to use, and the string is our input.

In our case, we use the quoter `r`, which is defined in raw-strings-qq, and write any string we want, with multi-lines and unescaped characters! We could use this to write the tests we previously wrote:

```
{-# language QuasiQuotes #-}

...

import Text.RawString.QQ

...

example3 :: String
example3 = [r|
Remember that multiple lines with no separation
are grouped together to a single paragraph
but list items remain separate.

# Item 1 of a list
# Item 2 of the same list
|]
```

And add multi-line tests:

```
spec :: Spec
spec = do
  describe "Markup parsing tests" $ do
    simple
    multiline


multiline :: Spec
multiline = do
  describe "Multi-line tests" $ do
    it "example3" $
      shouldBe
        (parse example3)
        example3Result


example3 :: String
example3 = [r|
Remember that multiple lines with no separation
are grouped together to a single paragraph
but list items remain separate.

# Item 1 of a list
# Item 2 of the same list
|]

example3Result :: Document
example3Result =
  [ Paragraph "Remember that multiple lines with no separation are grouped together to
a single paragraph but list items remain separate."
  , OrderedList
    [ "Item 1 of a list"
    , "Item 2 of the same list"
    ]
  ]
```

Running the tests:

```
MarkupParsing
  Markup parsing tests
    simple
      Test empty
      paragraph
      heading 1
      code
    Multi-line tests
      example3

Finished in 0.0004 seconds
5 examples, 0 failures
```

Exercise: Add a test for the fourth example described in the previous exercises.

Parallel test execution

Without further configuration, Hspec will run all of our tests on the main thread sequentially.

There are a couple of ways to configure tests to run in parallel. One is to manually mark a `Spec` as parallel by passing it to the `parallel` function, and another is by creating a /hook/ that will apply `parallel` to each `Spec` automatically with `hspec-discover`.

Consult the Hspec manual on this topic and try both methods. Remember that we already enabled the threaded runtime and set it to use multiple cores in the cabal file.

Summary

This chapter has been just the tip of the iceberg of the Haskell testing landscape. We haven't talked about property testing or golden testing, testing expected failures, testing IO code, inspection testing, benchmarking, and more. There's just too much to cover!

I hope this chapter provided you with the basics of how to start writing tests for your projects. Please consult the tutorial for your chosen testing framework, and read more about this very important subject on your own.

Generating documentation

There are many ways to help others to get started with our projects and libraries. For example, we can write tutorials, provide runnable examples, describe the system's internals, and create an API reference.

In this chapter, we will focus on generating API reference pages (the kind that can be seen on Hackage) from annotated Haskell source code using Haddock.

Running Haddock

We can generate API reference pages (a.k.a. haddocks in the Haskell world) for our project using our favorite package manager:

Cabal

We can run `cabal haddock` to generate haddocks:

```
➜ cabal haddock
Resolving dependencies...
Build profile: -w ghc-9.0.1 -O1
In order, the following will be built (use -v for more details):
 - hs-blog-0.1.0.0 (lib) (first run)
Configuring library for hs-blog-0.1.0.0..
Preprocessing library for hs-blog-0.1.0.0..
Running Haddock on library for hs-blog-0.1.0.0..
Haddock coverage:
   0% (  0 /  3) in 'HsBlog.Env'
  Missing documentation for:
    Module header
    Env (src/HsBlog/Env.hs:3)
    defaultEnv (src/HsBlog/Env.hs:10)
  21% (  7 / 33) in 'HsBlog.Html.Internal'
  Missing documentation for:
    Module header
    Html (src/HsBlog/Html/Internal.hs:8)
...
Documentation created:
/tmp/learn-haskell-blog-generator/dist-newstyle/build/x86_64-linux/ghc-9.0.1/hs-
blog-0.1.0.0/doc/html/hs-blog/index.html
```

Cabal and Haddock will build our project and generate HTML pages for us at:

```
./dist-newstyle/build/<platform>/<compiler>/<package>-<version>/doc/html/<package>/
```

We can then open the `index.html` file from that directory in a web browser and view our package documentation.

Stack

We can run `stack haddock` to generate haddocks:

```
➜ stack haddock
...
hs-blog> build (lib + exe)
Preprocessing library for hs-blog-0.1.0.0..
Building library for hs-blog-0.1.0.0..
[1 of 7] Compiling HsBlog.Env
[2 of 7] Compiling HsBlog.Html.Internal
...
hs-blog> haddock
Preprocessing library for hs-blog-0.1.0.0..
Running Haddock on library for hs-blog-0.1.0.0..
Haddock coverage:
   0% (  0 /  3) in 'HsBlog.Env'
  Missing documentation for:
    Module header
    Env (src/HsBlog/Env.hs:3)
    defaultEnv (src/HsBlog/Env.hs:10)
  21% (  7 / 33) in 'HsBlog.Html.Internal'
  Missing documentation for:
    Module header
    Html (src/HsBlog/Html/Internal.hs:8)
...
Documentation created:
.stack-work/dist/x86_64-linux-tinfo6/Cabal-3.2.1.0/doc/html/hs-blog/index.html,
.stack-work/dist/x86_64-linux-tinfo6/Cabal-3.2.1.0/doc/html/hs-blog/hs-blog.txt
Preprocessing executable 'hs-blog-gen' for hs-blog-0.1.0.0..
...
```

Stack and Haddock will build our project and generate HTML pages for us at:

```
./.stack-work/dist/<platform>/Cabal-<version>/doc/html/<package>/
```

We can then open the `index.html` file from that directory in a web browser and view our package documentation.

Haddock coverage

Haddock will also output a coverage report when run and mention user-exposed constructs that are missing documentation. These constructs could be module headers, types, data constructors, type classes, functions, values, etc.

For example:

```
Haddock coverage:
...
   0% (  0 /  3) in 'HsBlog.Convert'
  Missing documentation for:
    Module header
    convert (src/HsBlog/Convert.hs:8)
    convertStructure (src/HsBlog/Convert.hs:23)
  67% (  2 /  3) in 'HsBlog.Directory'
  Missing documentation for:
    buildIndex (src/HsBlog/Directory.hs:80)
...
```

We can see that we did not document the `HsBlog.Convert` at all, and we are missing documentation for the module header, the `convert` function, and the `convertStructure` function.

On the other hand, it seems that we do currently have some documentation written for the `HsBlog.Directory` module! We'll see why, but first - try to generate haddocks, visit the module hierarchy, browse around the different modules, follow the links of the types, imagine what this API reference could look like, and let's see how we can improve it.

Haddock markup

Haddock builds the API reference pages by building our project, examining the exported modules and their exported definitions, and grabbing source code comments written in special markup format.

Let's take a quick look at this markup format. We will go over a few important bits, but if you'd like to learn more, a complete guide for Haddock markup can be found in the Haddock documentation.

Documenting definitions

All haddock annotations appear as part of regular Haskell comments. They can be used with both single-line form (`--`) and multi-line form (`{-` and `-}`). The placements of a comment block and the haddock marker determine which Haskell definition the haddock string is attached to.

We can annotate a Haskell definition by writing a comment block prefixed with `|` *before* the definition, or by writing a comment block prefixed with `^` *after* the definition.

For example:

```
-- | Construct an HTML page from a `Head`
--   and a `Structure`.
html_
  :: Head -- ^ Represents the @\<head\>@ section in an HTML file
  -> Structure -- ^ Represents the @\<body\>@ section in an HTML file
  -> Html
html_ = ...
...
```

Here's another example:

```
{- | Represents a single markup structure. Such as:

- A paragraph
- An unordered list
- A code block
-}
data Structure
  = Heading Natural String
  -- ^ A section heading with a level
  | Paragraph String
  -- ^ A paragraph
  | UnorderedList [String]
  -- ^ An unordered list of strings
  | OrderedList [String]
  -- ^ An ordered list of strings
  | CodeBlock [String]
  -- ^ A code block
```

And another:

```
{- | Markup to HTML conversion module.

This module handles converting documents written in our custom
Markup language into HTML pages.
-}
module HsBlog.Convert where
```

As you can see, `|` and `^` can be used to document functions, function arguments, types, data constructors, modules, and more. They are probably the most important Haddock annotations to remember (and even then, `|` alone will suffice).

> **Tip**: Annotate the modules, types, and the top-level definitions which are exported from your project with some high-level description of what they are used for (at the very least).
>
> Your users and collaborators will thank you!

Section headings

We can separate our module into sections by adding headings. Headings are comments prefixed with a number of `*` (just like in our markup language).

For example:

```
-- * HTML EDSL

html_ :: Head -> Structure -> Html
html_ = ...

-- ** Structure

p_ :: Content -> Structure
p_ = ..

h_ :: Content -> Structure
h_ = ..

...

-- ** Content

txt_ :: String -> Content
txt_ = ...

link_ :: FilePath -> Content -> Content
link_ = ...
```

It is also possible to add headings to the export list instead:

```
module HsBlog.Html
  ( -- * HTML EDSL
    Html
  , html_

    -- ** Combinators used to construct the @\<head\>@ section
  , Head
  , title_
  , stylesheet_
  , meta_

    -- ** Combinators used to construct the @\<body\>@ section
  , Structure
  , p_
  , h_
  , ul_
  , ol_
  , code_

    -- ** Combinators used to construct content inside structures
  , Content
  , txt_
  , img_
  , link_
  , b_
  , i_

    -- ** Render HTML to String
  , render
  )
  where
```

Separating parts of the module into sections helps keeping the important things together and Haddock will create a table-of-contents at the top of a module page for us as well.

Sometimes it's also easier to figure out whether a module should be split into multiple modules or not after splitting it into sections using headings.

Exercise: Try to re-arrange the modules in our project to your liking and add headings to sections.

Formatting

As we saw earlier, we can also add formatting in the content of our comments. For example, we can:

- Hyperlink identifiers by surrounding them with `` ` ``

  For example: `` `Heading` ``

- Write `monospaced text` by surrounding it with `@`

  For example: `@Paragraph "Hello"@`

- Add *emphasis* to text by surrounding it with `/`

  For example: `/this is emphasised/`

- Add **bold** to text by surrounding it with `__`

For example: `__this is bold__`

More

In this chapter, we've covered the basics of the Haddock markup language. If you'd like to know more, the Haddock markup guide contains information on creating even more interesting documentation structures, such as code blocks, grid tables, images, and examples.

Summary

We've briefly covered one aspect of documenting Haskell programs: using Haddock to generate informative API reference pages created from source code comments which are annotated with Haddock markup.

While API references are incredibly valuable, remember that there are other forms of documentation that can help your users get started quickly, such as examples and tutorials.

Exercise: Add haddock annotation to the top-level definitions in our project and test your understanding of the program and the various parts - sometimes, the best way to learn something is to try explaining it!

Recap

In this book, we've implemented a very simple static blog generator while learning Haskell as we go.

- We've learned about basic Haskell building blocks, such as definitions, functions, types, modules, recursion, pattern matching, type classes, IO, and exceptions.
- We've learned about EDSLs and used the *combinator pattern* to implement a composable html generation library.
- We've learned how to leverage types, modules, and smart constructors to make invalid states unrepresentable.
- We've learned how to represent complex data using ADTs.
- We've learned how to use pattern matching to transform ADTs, and how to use recursion to solve problems.
- We've used the *functional core, imperative shell* approach to build a program that handles IO and applies our domain logic to user inputs.
- We've learned about abstractions such as monoids, functors and monads, and how they can help us reuse code and convey information about shared interfaces.
- We've learned how to create fancy command-line interfaces, write tests, and generate documentation.

While Haskell is a very big and complex language, and there's always more to be learned, I think we've reached an important milestone where you can start building your own Haskell projects and be productive with Haskell!

This is a good time to celebrate and pat yourself on the back for getting this far! Great job, you!

If you'd like to learn even more about Haskell and continue your Haskell journey beyond this book, check out the appendix sections Where to go next and the FAQ.

Thank you!

Thank you for reading this book. I hope you enjoyed it and found Haskell interesting.

I would very much like to hear your feedback. If you'd like, you could leave your feedback on this book's discussion board, or you could reach me directly on mastodon or via email. You can find my contact information on my website.

If you liked this book, do let me know - your kind words mean a lot.

> Finally, if you *really* liked this book and would like to support future passion projects like it, you can support me directly via Ko-fi.

Thank you, and good luck with your next Haskell project!

Where to go next

Haskell is an incredibly rich and deep programming language. New cutting-edge techniques, concepts, and features are still being discovered and sometimes integrated into GHC. This sometimes makes it seemingly impossible to catch up to.

This phenomena is sometimes dubbed The Haskell pyramid. I hope that by reading this book and following the exercises, you readers have reached the bar of productivity, and you can now go and start working on your own projects with Haskell. I highly encourage you to do so. In my opinion, writing useful Haskell projects is the best method to solidify what you currently know and identify what you still need to learn.

Extending this project

If you'd like to extend this project, here are a few ideas for you:

1. **Serve over HTTP** - You can use a web library such as wai or twain to serve this blog over HTTP instead of generating it statically
2. **Rewrite it with libraries** - you could rewrite it and use a real-world HTML package and markdown parser
3. **Add features**
    1. You could add a metadata block at the top of each article which would include the title, publish date, and tags of a blog post, and use them when generating HTML, index page, and even tags pages
    2. You could add HTML pages templating using mustache or similar, and use that to generate a nice and customizable structure to the page
    3. You could add support for links and images in our markup language parser
    4. You could add support for configuration files which would include things like the blog title, description, or other meta information for things like twitter cards

Or anything else you can think of, consider this project your playground and do whatever you like with it!

Other resources

At some point, you are likely to run into new concepts, techniques, or even just a feeling of "I feel like this could be done better". I'd like to point you in the right direction so you can find additional information and learn new Haskell things when you need to or want to.

I've compiled a list of resources for learning Haskell called Haskell study plan, which includes links to very useful articles, community hubs, and news aggregators, project suggestions, and even cool open-source Haskell projects. You will also find alternative explanations for things we've covered and even links to other Haskell tutorials, guides, and books in case you need a different view on things.

Also, the GHC User Guide is a fantastic resource with loads of articles and information about the language and GHC tooling around it. It is often the best place to learn about the Haskell language.

However, don't feel pressured to learn everything Haskell has to offer right away. Mastering Haskell is a journey that can take a lot of time. Most of us are definitely not there yet, but we can still be very productive with Haskell, build real-world projects, and even discover new techniques and concepts ourselves.

Remember that in a lazy language, we evaluate things only when needed. Maybe we can do that too, with Haskell concepts!

Frequently asked questions

Got a question? You can ask in the discussion board or the issue tracker!

General questions

Why should I learn Haskell

I've written a couple of articles on the topic:

- Consider Haskell (Alternative title, 'What can I do with Haskell?')
- 7 things I learned from Haskell

How to install editor tools

As far as I know, the most recommended setup today for Haskell development is using VSCode or VSCodium together with the marketplace Haskell extension.

The Haskell extension uses haskell-language-server which can be installed via GHCup or even via the Haskell extension itself.

If you already have a preferred editor, see if HLS supports it, or alternatively use GHCid which provides rapid feedback independently from an editor.

How to learn new things

The Haskell community keeps marching forward, developing new libraries, tools, and techniques as well as creating new material for older concepts. The Haskell planetarium aggregates feeds from several communities into one page, as well as a Haskell Weekly newsletter. You might also find quite a bit of Haskell presence on the Fediverse!

Debugging

How to debug Haskell code

Most imperative languages provide a step debugger. While the GHCi debugger, exists, it is not particularly easy to use, especially because of Haskell's lazy evaluation, where things might not be evaluated in the order we might intuitively expect. Because of that, Haskellers tend to use trace debugging and equational reasoning. With trace debugging, we try to *verify our assumptions* about the code - we use the various `trace` functions as a "hack" to print variables, functions inputs, functions output or even just say "got here" from anywhere in the code.

After finding something that does not match our assumptions, such as unexpected input or output of a function, we try to think what piece of code could be responsible for the discrepancy or even use trace debugging again to pinpoint the exact location, and try to use "equational reasoning" to evaluate the offending code that betrayed our expectations. If it's easy to do, we try running the function in `ghci` with different inputs to check our assumptions as well.

Because Haskell focuses on immutability, composability, and using types to eliminate many classes of possible errors, "local reasoning" becomes possible, and trace debugging becomes a viable strategy for debugging Haskell programs.

How to understand type errors

GHC type errors are often not the most friendly error messages, but they mean well! They are just trying to help us find inconsistencies in our code - often with regards to type usage, they help us avoid making errors.

When you run into error messages, start by reading them carefully until you get used to them, and then the offending code hinted at by the error message. As you gain experience, it is likely that the most important part of an error will be the location of the offending code, and by reading the code, we can find the error without the actual error message.

Adding type signatures and annotations to test your understanding of the types also helps greatly. We can even ask GHC for the expected type in a certain place by using typed holes.

My program is slow. Why?

There could be various reasons. From inefficient algorithms or unsuited data structures for the task in terms of time complexity of the common operations, to less efficient memory representations (this is another reminder to use `Text` over `String` in most cases), and laziness issues (again, the evaluation strategy!).

The performance section in my Haskell study plan links to various resources on Haskell evaluation, profiling, and case studies.

Design

How to structure programs

Start with the imperative shell functional core approach, define EDSLs with the combinator pattern for logic if needed, use capabilities such as `State` locally if needed, maybe add an environment configuration with `ReaderT`, and see how it goes.

If that approach fails you, look at why it fails and examine other solutions according to your needs.

How to model data

Modeling data using ADTs are usually the way to go. Often programmers coming from object-oriented background tend to look at type classes as a way to define methods similar to inheritance, but this often isn't the right approach, and ADTs with different constructors for different alternatives go a long way. Remember that even OOP people often preach for composition over inheritance.

Use functions to define behavior on data rather than trying to couple the two together.