CHAPTER 10

Code Case Study: Parsing a Binary Data Format

In this chapter, we'll discuss a common task: parsing a binary file. We will use it for two purposes. Our first is indeed to talk a little about parsing, but our main goal is to talk about program organization, refactoring, and "boilerplate removal." We will demonstrate how you can tidy up repetitious code, and set the stage for our discussion of monads in Chapter 14.

The file formats that we will work with come from the *netpbm suite*, an ancient and venerable collection of programs and file formats for working with bitmap images. These file formats have the dual advantages of being widely used and being fairly easy, though not completely trivial, to parse. Most importantly for our convenience, netpbm files are not compressed.

Grayscale Files

The name of netpbm's grayscale file format is PGM (*portable gray map*). It is actually not one format, but two; the *plain* (or P2) format is encoded as ASCII, while the more common *raw* (P5) format is mostly binary.

A file of either format starts with a header, which in turn begins with a "magic" string describing the format. For a plain file, the string is `P2`, and for raw, it's `P5`. The magic string is followed by whitespace, and then by three numbers: the width, height, and maximum gray value of the image. These numbers are represented as ASCII decimal numbers, separated by whitespace.

After the maximum gray value comes the image data. In a raw file, this is a string of binary values. In a plain file, the values are represented as ASCII decimal numbers separated by single-space characters.

A raw file can contain a sequence of images, one after the other, each with its own header. A plain file contains only one image.

Parsing a Raw PGM File

For our first try at a parsing function, we'll only worry about raw PGM files. We'll write our PGM parser as a *pure* function. It's won't be responsible for obtaining the data to parse, just for the actual parsing. This is a common approach in Haskell programs. By separating the reading of the data from what we subsequently do with it, we gain flexibility in where we take the data from.

We'll use the `ByteString` type to store our graymap data, because it's compact. Since the header of a PGM file is ASCII text but its body is binary, we import both the text- and binary-oriented `ByteString` modules:

```
-- file: ch10/PNM.hs
import qualified Data.ByteString.Lazy.Char8 as L8
import qualified Data.ByteString.Lazy as L
import Data.Char (isSpace)
```

For our purposes, it doesn't matter whether we use a lazy or strict `ByteString`, so we've somewhat arbitrarily chosen the lazy kind.

We'll use a straightforward data type to represent PGM images:

```
-- file: ch10/PNM.hs
data Greymap = Greymap {
      greyWidth :: Int
    , greyHeight :: Int
    , greyMax :: Int
    , greyData :: L.ByteString
    } deriving (Eq)
```

Normally, a Haskell `Show` instance should produce a string representation that we can read back by calling `read`. However, for a bitmap graphics file, this would potentially produce huge text strings, for example, if we were to `show` a photo. For this reason, we're not going to let the compiler automatically derive a `Show` instance for us; we'll write our own and intentionally simplify it:

```
-- file: ch10/PNM.hs
instance Show Greymap where
    show (Greymap w h m _) = "Greymap " ++ show w ++ "x" ++ show h ++
                             " " ++ show m
```

Because our `Show` instance intentionally avoids printing the bitmap data, there's no point in writing a `Read` instance, as we can't reconstruct a valid `Greymap` from the result of `show`.

Here's an obvious type for our parsing function:

```
-- file: ch10/PNM.hs
parseP5 :: L.ByteString -> Maybe (Greymap, L.ByteString)
```

This will take a `ByteString`, and if the parse succeeds, it will return a single parsed `Greymap`, along with the string that remains after parsing. That residual string will be available for future parses.

Our parsing function has to consume a little bit of its input at a time. First, we need to assure ourselves that we're really looking at a raw PGM file; then we need to parse the numbers from the remainder of the header; and then we consume the bitmap data. Here's an obvious way to express this, which we will use as a base for later improvements:

```
-- file: ch10/PNM.hs
matchHeader :: L.ByteString -> L.ByteString -> Maybe L.ByteString

-- "nat" here is short for "natural number"
getNat :: L.ByteString -> Maybe (Int, L.ByteString)

getBytes :: Int -> L.ByteString
         -> Maybe (L.ByteString, L.ByteString)

parseP5 s =
  case matchHeader (L8.pack "P5") s of
    Nothing -> Nothing
    Just s1 ->
      case getNat s1 of
        Nothing -> Nothing
        Just (width, s2) ->
          case getNat (L8.dropWhile isSpace s2) of
            Nothing -> Nothing
            Just (height, s3) ->
              case getNat (L8.dropWhile isSpace s3) of
                Nothing -> Nothing
                Just (maxGrey, s4)
                  | maxGrey > 255 -> Nothing
                  | otherwise ->
                      case getBytes 1 s4 of
                        Nothing -> Nothing
                        Just (_, s5) ->
                          case getBytes (width * height) s5 of
                            Nothing -> Nothing
                            Just (bitmap, s6) ->
                              Just (Greymap width height maxGrey bitmap, s6)
```

This is a very literal piece of code, performing all of the parsing in one long staircase of `case` expressions. Each function returns the residual `ByteString` left over after it has consumed all it needs from its input string. We pass each residual string along to the next step. We deconstruct each result in turn, either returning `Nothing` if the parsing step fails, or building up a piece of the final result as we proceed. Here are the bodies of the functions that we apply during parsing (their types are commented out because we already presented them):

```
-- file: ch10/PNM.hs
-- L.ByteString -> L.ByteString -> Maybe L.ByteString
matchHeader prefix str
    | prefix `L8.isPrefixOf` str
        = Just (L8.dropWhile isSpace (L.drop (L.length prefix) str))
    | otherwise
        = Nothing
```

```
-- L.ByteString -> Maybe (Int, L.ByteString)
getNat s = case L8.readInt s of
             Nothing -> Nothing
             Just (num,rest)
                 | num <= 0    -> Nothing
                 | otherwise -> Just (fromIntegral num, rest)

-- Int -> L.ByteString -> Maybe (L.ByteString, L.ByteString)
getBytes n str = let count           = fromIntegral n
                     both@(prefix,_) = L.splitAt count str
                 in if L.length prefix < count
                    then Nothing
                    else Just both
```

Getting Rid of Boilerplate Code

While our `parseP5` function works, the style in which we wrote it is somehow not pleasing. Our code marches steadily to the right of the screen, and it's clear that a slightly more complicated function would soon run out of visual real estate. We repeat a pattern of constructing and then deconstructing `Maybe` values, only continuing if a particular value matches `Just`. All of the similar `case` expressions act as *boilerplate code*, busywork that obscures what we're really trying to do. In short, this function is begging for some abstraction and refactoring.

If we step back a little, we can see two patterns. First is that many of the functions that we apply have similar types. Each takes a `ByteString` as its last argument and returns `Maybe` something else. Second, every step in the "ladder" of our `parseP5` function deconstructs a `Maybe` value, and either fails or passes the unwrapped result to a function.

We can quite easily write a function that captures this second pattern:

```
-- file: ch10/PNM.hs
(>>?) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>? _ = Nothing
Just v  >>? f = f v
```

The `(>>?)` function acts very simply: it takes a value as its left argument, and a function as its right. If the value is not `Nothing`, it applies the function to whatever is wrapped in the `Just` constructor. We have defined our function as an operator so that we can use it to chain functions together. Finally, we haven't provided a fixity declaration for `(>>?)`, so it defaults to `infixl 9` (left-associative, strongest operator precedence). In other words, `a >>? b >>? c` will be evaluated from left to right, as `(a >>? b) >>? c`).

With this chaining function in hand, we can take a second try at our parsing function:

```
-- file: ch10/PNM.hs
parseP5_take2 :: L.ByteString -> Maybe (Greymap, L.ByteString)
parseP5_take2 s =
    matchHeader (L8.pack "P5") s      >>?
    \s -> skipSpace ((), s)           >>?
    (getNat . snd)                    >>?
```

```
    skipSpace                       >>?
    \(width, s) ->   getNat s       >>?
    skipSpace                       >>?
    \(height, s) ->  getNat s       >>?
    \(maxGrey, s) -> getBytes 1 s   >>?
    (getBytes (width * height) . snd) >>?
    \(bitmap, s) -> Just (Greymap width height maxGrey bitmap, s)

skipSpace :: (a, L.ByteString) -> Maybe (a, L.ByteString)
skipSpace (a, s) = Just (a, L8.dropWhile isSpace s)
```

The key to understanding this function is to think about the chaining. On the left side of each `(>>?)` is a `Maybe` value; on the right is a function that returns a `Maybe` value. Each left-and-right-side expression is thus of type `Maybe`, suitable for passing to the following `(>>?)` expression.

The other change that we've made to improve readability is add a `skipSpace` function. With these changes, we've halved the number of lines of code compared to our original parsing function. By removing the boilerplate `case` expressions, we've made the code easier to follow.

While we warned against overuse of anonymous functions in "Anonymous (lambda) Functions" on page 99, we use several in our chain of functions here. Because these functions are so small, we wouldn't improve readability by giving them names.

Implicit State

We're not yet out of the woods. Our code explicitly passes pairs around, using one element for an intermediate part of the parsed result and the other for the current residual `ByteString`. If we want to extend the code, for example, to track the number of bytes we've consumed so that we can report the location of a parse failure, we already have eight different spots that we will need to modify, just to pass a three-tuple around.

This approach makes even a small body of code difficult to change. The problem lies with our use of pattern matching to pull values out of each pair: we have embedded the knowledge that we are always working with pairs straight into our code. As pleasant and helpful as pattern matching is, it can lead us in some undesirable directions if we do not use it carefully.

Let's do something to address the inflexibility of our new code. First, we will change the type of state that our parser uses:

```
-- file: ch10/Parse.hs
data ParseState = ParseState {
      string :: L.ByteString
    , offset :: Int64           -- imported from Data.Int
    } deriving (Show)
```

In our switch to an algebraic data type, we added the ability to track both the current residual string and the offset into the original string since we started parsing. The more

important change was our use of record syntax: we can now *avoid* pattern matching on the pieces of state that we pass around and use the accessor functions `string` and `offset` instead.

We have given our parsing state a name. When we name something, it can become easier to reason about. For example, we can now look at parsing as a kind of function: it consumes a parsing state and produces both a new parsing state and some other piece of information. We can directly represent this as a Haskell type:

```
-- file: ch10/Parse.hs
simpleParse :: ParseState -> (a, ParseState)
simpleParse = undefined
```

To provide more help to our users, we would like to report an error message if parsing fails. This requires only a minor tweak to the type of our parser:

```
-- file: ch10/Parse.hs
betterParse :: ParseState -> Either String (a, ParseState)
betterParse = undefined
```

In order to future-proof our code, it is best if we do not expose the implementation of our parser to our users. When we explicitly used pairs for state earlier, we found ourselves in trouble almost immediately, once we considered extending the capabilities of our parser. To stave off a repeat of that difficulty, we will hide the details of our parser type using a `newtype` declaration:

```
-- file: ch10/Parse.hs
newtype Parse a = Parse {
      runParse :: ParseState -> Either String (a, ParseState)
    }
```

Remember that the `newtype` definition is just a compile-time wrapper around a function, so it has no runtime overhead. When we want to use the function, we will apply the `runParser` accessor.

If we do not export the `Parse` value constructor from our module, we can ensure that nobody else will be able to accidentally create a parser, nor will they be able to inspect its internals via pattern matching.

The Identity Parser

Let's try to define a simple parser, the *identity* parser. All it does is turn whatever it is passed into the result of the parse. In this way, it somewhat resembles the `id` function:

```
-- file: ch10/Parse.hs
identity :: a -> Parse a
identity a = Parse (\s -> Right (a, s))
```

This function leaves the parse state untouched and uses its argument as the result of the parse. We wrap the body of the function in our `Parse` type to satisfy the type checker. How can we use this wrapped function to parse something?

The first thing we must do is peel off the `Parse` wrapper so that we can get at the function inside. We do so using the `runParse` function. We also need to construct a `ParseState`, and then run our parsing function on it. Finally, we'd like to separate the result of the parse from the final `ParseState`:

```
-- file: ch10/Parse.hs
parse :: Parse a -> L.ByteString -> Either String a
parse parser initState
    = case runParse parser (ParseState initState 0) of
        Left err          -> Left err
        Right (result, _) -> Right result
```

Because neither the `identity` parser nor the `parse` function examines the parse state, we don't even need to create an input string in order to try our code:

```
ghci> :load Parse
[1 of 2] Compiling PNM              ( PNM.hs, interpreted )
[2 of 2] Compiling Parse            ( Parse.hs, interpreted )
Ok, modules loaded: Parse, PNM.
ghci> :type parse (identity 1) undefined
parse (identity 1) undefined :: (Num t) => Either String t
ghci> parse (identity 1) undefined
Loading package array-0.1.0.0 ... linking ... done.
Loading package bytestring-0.9.0.1.1 ... linking ... done.
Right 1
ghci> parse (identity "foo") undefined
Right "foo"
```

A parser that doesn't even inspect its input might not seem interesting, but we will see shortly that in fact it is useful. Meanwhile, we have gained confidence that our types are correct and that we understand the basic workings of our code.

Record Syntax, Updates, and Pattern Matching

Record syntax is useful for more than just accessor functions—we can use it to copy and partly change an existing value. In use, the notation looks like this:

```
-- file: ch10/Parse.hs
modifyOffset :: ParseState -> Int64 -> ParseState
modifyOffset initState newOffset =
    initState { offset = newOffset }
```

This creates a new `ParseState` value identical to `initState`, but with its `offset` field set to whatever value we specify for `newOffset`:

```
ghci> let before = ParseState (L8.pack "foo") 0
ghci> let after = modifyOffset before 3
ghci> before
ParseState {string = Chunk "foo" Empty, offset = 0}
ghci> after
ParseState {string = Chunk "foo" Empty, offset = 3}
```

We can set as many fields as we want inside the curly braces, separating them using commas.

A More Interesting Parser

Let's focus now on writing a parser that does something meaningful. We're not going to get too ambitious yet—all we want to do is parse a single byte:

```
-- file: ch10/Parse.hs
-- import the Word8 type from Data.Word
parseByte :: Parse Word8
parseByte =
    getState ==> \initState ->
    case L.uncons (string initState) of
      Nothing ->
          bail "no more input"
      Just (byte,remainder) ->
          putState newState ==> \_ ->
          identity byte
        where newState = initState { string = remainder,
                                     offset = newOffset }
              newOffset = offset initState + 1
```

There are a number of new functions in our definition.

The `L8.uncons` function takes the first element from a `ByteString`:

```
ghci> L8.uncons (L8.pack "foo")
Just ('f',Chunk "oo" Empty)
ghci> L8.uncons L8.empty
Nothing
```

Our `getState` function retrieves the current parsing state, while `putState` replaces it. The `bail` function terminates parsing and reports an error. The `(==>)` function chains parsers together. We will cover each of these functions shortly.

Hanging lambdas

The definition of `parseByte` has a visual style that we haven't discussed before. It contains anonymous functions in which the parameters and `->` sit at the end of a line, with the function's body following on the next line.

This style of laying out an anonymous function doesn't have an official name, so let's call it a "hanging lambda." Its main use is to make room for more text in the body of the function. It also makes it more visually clear that there's a relationship between a function and the one that follows it. Often, for instance, the result of the first function is being passed as a parameter to the second.

Obtaining and Modifying the Parse State

Our `parseByte` function doesn't take the parse state as an argument. Instead, it has to call `getState` to get a copy of the state and `putState` to replace the current state with a new one:

```
-- file: ch10/Parse.hs
getState :: Parse ParseState
getState = Parse (\s -> Right (s, s))

putState :: ParseState -> Parse ()
putState s = Parse (\_ -> Right ((), s))
```

When reading these functions, recall that the left element of the tuple is the result of a `Parse`, while the right is the current `ParseState`. This makes it easier to follow what these functions are doing.

The `getState` function extracts the current parsing state so that the caller can access the string. The `putState` function replaces the current parsing state with a new one. This becomes the state that will be seen by the next function in the `(==>)` chain.

These functions let us move explicit state handling into the bodies of only those functions that need it. Many functions don't need to know what the current state is, and so they'll never call `getState` or `putState`. This lets us write more compact code than our earlier parser, which had to pass tuples around by hand. We will see the effect in some of the code that follows.

We've packaged up the details of the parsing state into the `ParseState` type, and we work with it using accessors instead of pattern matching. Now that the parsing state is passed around implicitly, we gain a further benefit. If we want to add more information to the parsing state, all we need to do is modify the definition of `ParseState` and the bodies of whatever functions need the new information. Compared to our earlier parsing code, where all of our state was exposed through pattern matching, this is much more modular: the only code we affect is code that needs the new information.

Reporting Parse Errors

We carefully defined our `Parse` type to accommodate the possibility of failure. The `(==>)` combinator checks for a parse failure and stops parsing if it runs into a failure. But we haven't yet introduced the `bail` function, which we use to report a parse error:

```
-- file: ch10/Parse.hs
bail :: String -> Parse a
bail err = Parse $ \s -> Left $
           "byte offset " ++ show (offset s) ++ ": " ++ err
```

After we call `bail`, `(==>)` will successfully pattern match on the `Left` constructor that it wraps the error message with, and it will not invoke the next parser in the chain. This will cause the error message to percolate back through the chain of prior callers.

Chaining Parsers Together

The `(==>)` function serves a similar purpose to our earlier `(>>?)` function—it is "glue" that lets us chain functions together:

```
-- file: ch10/Parse.hs
(==>) :: Parse a -> (a -> Parse b) -> Parse b

firstParser ==> secondParser  =  Parse chainedParser
  where chainedParser initState   =
          case runParse firstParser initState of
            Left errMessage ->
                Left errMessage
            Right (firstResult, newState) ->
                runParse (secondParser firstResult) newState
```

The body of `(==>)` is interesting and ever so slightly tricky. Recall that the `Parse` type represents really a function inside a wrapper. Since `(==>)` lets us chain two `Parse` values to produce a third, it must return a function, in a wrapper.

The function doesn't really "do" much, it just creates a *closure* to remember the values of `firstParser` and `secondParser`.

A closure is simply the pairing of a function with its *environment*, the bound variables that it can see. Closures are commonplace in Haskell. For instance, the section `(+5)` is a closure. An implementation must record the value `5` as the second argument to the `(+)` operator so that the resulting function can add `5` to whatever value it is passed.

This closure will not be unwrapped and applied until we apply `parse`. At that point, it will be applied with a `ParseState`. It will apply `firstParser` and inspect its result. If that parse fails, the closure will fail too. Otherwise, it will pass the result of the parse and the new `ParseState` to `secondParser`.

This is really quite fancy and subtle stuff. We're effectively passing the `ParseState` down the chain of `Parse` values in a hidden argument. (We'll be revisiting this kind of code in a few chapters, so don't fret if this description seems dense.)

Introducing Functors

We're by now thoroughly familiar with the `map` function, which applies a function to every element of a list, returning a list of possibly a different type:

```
ghci> map (+1) [1,2,3]
[2,3,4]
ghci> map show [1,2,3]
["1","2","3"]
ghci> :type map show
map show :: (Show a) => [a] -> [String]
```

This `map`-like activity can be useful in other instances. For example, consider a binary tree:

```
-- file: ch10/TreeMap.hs
data Tree a = Node (Tree a) (Tree a)
```

```
  | Leaf a
    deriving (Show)
```

If we want to take a tree of strings and turn it into a tree containing the lengths of those strings, we could write a function to do this:

```
-- file: ch10/TreeMap.hs
treeLengths (Leaf s) = Leaf (length s)
treeLengths (Node l r) = Node (treeLengths l) (treeLengths r)
```

Now that our eyes are attuned to looking for patterns that we can turn into generally useful functions, we can see a possible case of this here:

```
-- file: ch10/TreeMap.hs
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f (Leaf a)   = Leaf (f a)
treeMap f (Node l r) = Node (treeMap f l) (treeMap f r)
```

As we might hope, `treeLengths` and `treeMap length` give the same results:

```
ghci> let tree = Node (Leaf "foo") (Node (Leaf "x") (Leaf "quux"))
ghci> treeLengths tree
Node (Leaf 3) (Node (Leaf 1) (Leaf 4))
ghci> treeMap length tree
Node (Leaf 3) (Node (Leaf 1) (Leaf 4))
ghci> treeMap (odd . length) tree
Node (Leaf True) (Node (Leaf True) (Leaf False))
```

Haskell provides a well-known typeclass to further generalize `treeMap`. This typeclass is named `Functor`, and it defines one function, `fmap`:

```
-- file: ch10/TreeMap.hs
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

We can think of `fmap` as a kind of *lifting* function, as we introduced in "Avoiding Boilerplate with Lifting" on page 223. It takes a function over ordinary values `a -> b`, and lifts it to become a function over containers `f a -> f b`, where `f` is the container type.

If we substitute `Tree` for the type variable `f`, for example, then the type of `fmap` is identical to the type of `treeMap`, and in fact we can use `treeMap` as the implementation of `fmap` over `Tree`s:

```
-- file: ch10/TreeMap.hs
instance Functor Tree where
    fmap = treeMap
```

We can also use `map` as the implementation of `fmap` for lists:

```
-- file: ch10/TreeMap.hs
instance Functor [] where
    fmap = map
```

We can now use `fmap` over different container types:

```
ghci> fmap length ["foo","quux"]
[3,4]
```

```
ghci> fmap length (Node (Leaf "Livingstone") (Leaf "I presume"))
Node (Leaf 11) (Leaf 9)
```

The `Prelude` defines instances of `Functor` for several common types, notably lists and `Maybe`:

```
-- file: ch10/TreeMap.hs
instance Functor Maybe where
    fmap _ Nothing  = Nothing
    fmap f (Just x) = Just (f x)
```

The instance for `Maybe` makes it particularly clear what an `fmap` implementation needs to do. The implementation must have a sensible behavior for each of a type's constructors. If a value is wrapped in `Just`, for example, the `fmap` implementation calls the function on the unwrapped value, then rewraps it in `Just`.

The definition of `Functor` imposes a few obvious restrictions on what we can do with `fmap`. For example, we can only make instances of `Functor` from types that have exactly one type parameter.

We can't write an `fmap` implementation for `Either a b` or `(a, b)`, for example, because these have two type parameters. We also can't write one for `Bool` or `Int`, as they have no type parameters.

In addition, we can't place any constraints on our type definition. What does this mean? To illustrate, let's first look at a normal `data` definition and its `Functor` instance:

```
-- file: ch10/ValidFunctor.hs
data Foo a = Foo a

instance Functor Foo where
    fmap f (Foo a) = Foo (f a)
```

When we define a new type, we can add a type constraint just after the `data` keyword as follows:

```
-- file: ch10/ValidFunctor.hs
data Eq a => Bar a = Bar a

instance Functor Bar where
    fmap f (Bar a) = Bar (f a)
```

This says that we can only put a type `a` into a `Foo` if `a` is a member of the `Eq` typeclass. However, the constraint renders it impossible to write a `Functor` instance for `Bar`:

```
ghci> :load ValidFunctor
[1 of 1] Compiling Main             ( ValidFunctor.hs, interpreted )

ValidFunctor.hs:12:12:
    Could not deduce (Eq a) from the context (Functor Bar)
      arising from a use of `Bar' at ValidFunctor.hs:12:12-16
    Possible fix:
      add (Eq a) to the context of the type signature for `fmap'
    In the pattern: Bar a
    In the definition of `fmap': fmap f (Bar a) = Bar (f a)
```

```
    In the definition for method `fmap'

ValidFunctor.hs:12:21:
    Could not deduce (Eq b) from the context (Functor Bar)
      arising from a use of `Bar' at ValidFunctor.hs:12:21-29
    Possible fix:
      add (Eq b) to the context of the type signature for `fmap'
    In the expression: Bar (f a)
    In the definition of `fmap': fmap f (Bar a) = Bar (f a)
    In the definition for method `fmap'
Failed, modules loaded: none.
```

Constraints on Type Definitions Are Bad

Adding a constraint to a type definition is essentially never a good idea. It has the effect of forcing you to add type constraints to *every* function that will operate on values of that type. Let's say that we need a stack data structure that we want to be able to query to see whether its elements obey some ordering. Here's a naive definition of the data type:

```
-- file: ch10/TypeConstraint.hs
data (Ord a) => OrdStack a = Bottom
                           | Item a (OrdStack a)
                             deriving (Show)
```

If we want to write a function that checks the stack to see whether it is increasing (i.e., every element is bigger than the element below it), we'll obviously need an `Ord` constraint to perform the pairwise comparisons:

```
-- file: ch10/TypeConstraint.hs
isIncreasing :: (Ord a) => OrdStack a -> Bool
isIncreasing (Item a rest@(Item b _))
    | a < b     = isIncreasing rest
    | otherwise = False
isIncreasing _  = True
```

However, because we wrote the type constraint on the type definition, that constraint ends up infecting places where it isn't needed. We need to add the `Ord` constraint to `push`, which does not care about the ordering of elements on the stack:

```
-- file: ch10/TypeConstraint.hs
push :: (Ord a) => a -> OrdStack a -> OrdStack a
push a s = Item a s
```

Try removing that `Ord` constraint, and the definition of `push` will fail to typecheck.

This is why our attempt to write a `Functor` instance for `Bar` failed earlier: it would have required an `Eq` constraint to somehow get retroactively added to the signature of `fmap`.

Now that we've tentatively established that putting a type constraint on a type definition is a misfeature of Haskell, what's a more sensible alternative? The answer is simply to omit type constraints from type definitions, and instead place them on the functions that need them.

In this example, we can drop the `Ord` constraints from `OrdStack` and `push`. It needs to stay on `isIncreasing`, which otherwise couldn't call `(<)`. We now have the constraints where they actually matter. This has the further benefit of making the type signatures better document the true requirements of each function.

Most Haskell container types follow this pattern. The `Map` type in the `Data.Map` module requires that its keys be ordered, but the type itself does not have such a constraint. The constraint is expressed on functions such as `insert`, where it's actually needed, and not on `size`, where ordering isn't used.

Infix Use of fmap

Quite often, you'll see `fmap` called as an operator:

```
ghci> (1+) `fmap` [1,2,3] ++ [4,5,6]
[2,3,4,4,5,6]
```

Perhaps strangely, plain old `map` is almost never used in this way.

One possible reason for the stickiness of the `fmap`-as-operator meme is that this use lets us omit parentheses from its second argument. Fewer parentheses leads to reduced mental juggling while reading a function:

```
ghci> fmap (1+) ([1,2,3] ++ [4,5,6])
[2,3,4,5,6,7]
```

If you really want to use `fmap` as an operator, the `Control.Applicative` module contains an operator `(<$>)` that is an alias for `fmap`. The `$` in its name appeals to the similarity between applying a function to its arguments (using the `($)` operator) and lifting a function into a functor. We will see that this works well for parsing when we return to the code that we have been writing.

Flexible Instances

You might hope that we could write a `Functor` instance for the type `Either Int b`, which has one type parameter:

```
-- file: ch10/EitherInt.hs
instance Functor (Either Int) where
    fmap _ (Left n) = Left n
    fmap f (Right r) = Right (f r)
```

However, the type system of Haskell 98 cannot guarantee that checking the constraints on such an instance will terminate. A nonterminating constraint check may send a compiler into an infinite loop, so instances of this form are forbidden:

```
ghci> :load EitherInt
[1 of 1] Compiling Main             ( EitherInt.hs, interpreted )

EitherInt.hs:2:0:
    Illegal instance declaration for `Functor (Either Int)'
        (All instance types must be of the form (T a1 ... an)
```

```
          where a1 ... an are type *variables*,
          and each type variable appears at most once in the instance head.
          Use -XFlexibleInstances if you want to disable this.)
      In the instance declaration for `Functor (Either Int)'
Failed, modules loaded: none.
```

GHC has a more powerful type system than the base Haskell 98 standard. It operates in Haskell 98 compatibility mode by default, for maximal portability. We can instruct it to allow more flexible instances using a special compiler directive:

```
-- file: ch10/EitherIntFlexible.hs
{-# LANGUAGE FlexibleInstances #-}

instance Functor (Either Int) where
    fmap _ (Left n)  = Left n
    fmap f (Right r) = Right (f r)
```

The directive is embedded in the specially formatted `LANGUAGE` pragma.

With our `Functor` instance in hand, let's try out `fmap` on `Either Int`:

```
ghci> :load EitherIntFlexible
[1 of 1] Compiling Main             ( EitherIntFlexible.hs, interpreted )
Ok, modules loaded: Main.
ghci> fmap (== "cheeseburger") (Left 1 :: Either Int String)
Left 1
ghci> fmap (== "cheeseburger") (Right "fries" :: Either Int String)
Right False
```

Thinking More About Functors

We've made a few implicit assumptions about how functors ought to work. It's helpful to make these explicit and to think of them as rules to follow, because this lets us treat functors as uniform, well-behaved objects. We have only two rules to remember, and they're simple:

- Our first rule is functors must preserve *identity*. That is, applying `fmap id` to a value should give us back an identical value:

  ```
  ghci> fmap id (Node (Leaf "a") (Leaf "b"))
  Node (Leaf "a") (Leaf "b")
  ```

- Our second rule is functors must be *composable*. That is, composing two uses of `fmap` should give the same result as one `fmap` with the same functions composed:

  ```
  ghci> (fmap even . fmap length) (Just "twelve")
  Just True
  ghci> fmap (even . length) (Just "twelve")
  Just True
  ```

Another way of looking at these two rules is that functors must preserve *shape*. The structure of a collection should not be affected by a functor; only the values that it contains should change:

```
ghci> fmap odd (Just 1)
Just True
ghci> fmap odd Nothing
Nothing
```

If you're writing a `Functor` instance, it's useful to keep these rules in mind, and indeed to test them, because the compiler can't check the rules we've just listed. On the other hand, if you're simply *using* functors, the rules are "natural" enough that there's no need to memorize them. They just formalize a few intuitive notions of "do what I mean." Here is a pseudocode representation of the expected behavior:

```
-- file: ch10/FunctorLaws.hs
fmap id       ==  id
fmap (f . g)  ==  fmap f . fmap g
```

Writing a Functor Instance for Parse

For the types we have surveyed so far, the behavior we ought to expect of `fmap` has been obvious. This is a little less clear for `Parse`, due to its complexity. A reasonable guess is that the function we're `fmap`ping should be applied to the current result of a parse, and leave the parse state untouched:

```
-- file: ch10/Parse.hs
instance Functor Parse where
    fmap f parser = parser ==> \result ->
                    identity (f result)
```

This definition is easy to read, so let's perform a few quick experiments to see if we're following our rules for functors.

First, we'll check that identity is preserved. Let's try this first on a parse that ought to fail—parsing a byte from an empty string (remember that `(<$>)` is `fmap`):

```
ghci> parse parseByte L.empty
Left "byte offset 0: no more input"
ghci> parse (id <$> parseByte) L.empty
Left "byte offset 0: no more input"
```

Good. Now for a parse that should succeed:

```
ghci> let input = L8.pack "foo"
ghci> L.head input
102
ghci> parse parseByte input
Right 102
ghci> parse (id <$> parseByte) input
Right 102
```

Inspecting these results, we can also see that our `Functor` instance is obeying our second rule of preserving shape. Failure is preserved as failure, and success as success.

Finally, we'll ensure that composability is preserved:

```
ghci> parse ((chr . fromIntegral) <$> parseByte) input
Right 'f'
ghci> parse (chr <$> fromIntegral <$> parseByte) input
Right 'f'
```

On the basis of this brief inspection, our `Functor` instance appears to be well behaved.

Using Functors for Parsing

All this talk of functors has a purpose: they often let us write tidy, expressive code. Recall the `parseByte` function that we introduced earlier. In recasting our PGM parser to use our new parser infrastructure, we'll often want to work with ASCII characters instead of `Word8` values.

While we could write a `parseChar` function that has a similar structure to `parseByte`, we can now avoid this code duplication by taking advantage of the functor nature of `Parse`. Our functor takes the result of a parse and applies a function to it, so what we need is a function that turns a `Word8` into a `Char`:

```
-- file: ch10/Parse.hs
w2c :: Word8 -> Char
w2c = chr . fromIntegral

-- import Control.Applicative
parseChar :: Parse Char
parseChar = w2c <$> parseByte
```

We can also use functors to write a compact "peek" function. This returns `Nothing` if we're at the end of the input string. Otherwise, it returns the next character without consuming it (i.e., it inspects, but doesn't disturb, the current parsing state):

```
-- file: ch10/Parse.hs
peekByte :: Parse (Maybe Word8)
peekByte = (fmap fst . L.uncons . string) <$> getState
```

The same lifting trick that let us define `parseChar` lets us write a compact definition for `peekChar`:

```
-- file: ch10/Parse.hs
peekChar :: Parse (Maybe Char)
peekChar = fmap w2c <$> peekByte
```

Notice that `peekByte` and `peekChar` each make two calls to `fmap`, one of which is disguised as `(<$>)`. This is necessary because the type `Parse (Maybe a)` is a functor within a functor. We thus have to lift a function twice to "get it into" the inner functor.

Finally, we'll write another generic combinator, which is the `Parse` analogue of the familiar `takeWhile`. It consumes its input while its predicate returns `True`:

```
-- file: ch10/Parse.hs
parseWhile :: (Word8 -> Bool) -> Parse [Word8]
parseWhile p = (fmap p <$> peekByte) ==> \mp ->
               if mp == Just True
               then parseByte ==> \b ->
                    (b:) <$> parseWhile p
               else identity []
```

Once again, we're using functors in several places (doubled up, when necessary) to reduce the verbosity of our code. Here's a rewrite of the same function in a more direct style that does not use functors:

```
-- file: ch10/Parse.hs
parseWhileVerbose p =
    peekByte ==> \mc ->
    case mc of
      Nothing -> identity []
      Just c | p c ->
                 parseByte ==> \b ->
                 parseWhileVerbose p ==> \bs ->
                 identity (b:bs)
             | otherwise ->
                 identity []
```

The more verbose definition is likely easier to read when you are less familiar with functors. However, use of functors is sufficiently common in Haskell code that the more compact representation should become second nature (both to read and to write) fairly quickly.

Rewriting Our PGM Parser

With our new parsing code, what does the raw PGM parsing function look like now?

```
-- file: ch10/Parse.hs
parseRawPGM =
    parseWhileWith w2c notWhite ==> \header -> skipSpaces ==>&
    assert (header == "P5") "invalid raw header" ==>&
    parseNat ==> \width -> skipSpaces ==>&
    parseNat ==> \height -> skipSpaces ==>&
    parseNat ==> \maxGrey ->
    parseByte ==>&
    parseBytes (width * height) ==> \bitmap ->
    identity (Greymap width height maxGrey bitmap)
  where notWhite = (`notElem` " \r\n\t")
```

This definition makes use of a few more helper functions that we present here, following a pattern that should be familiar by now:

```
-- file: ch10/Parse.hs
parseWhileWith :: (Word8 -> a) -> (a -> Bool) -> Parse [a]
parseWhileWith f p = fmap f <$> parseWhile (p . f)

parseNat :: Parse Int
parseNat = parseWhileWith w2c isDigit ==> \digits ->
           if null digits
           then bail "no more input"
           else let n = read digits
                in if n < 0
                   then bail "integer overflow"
                   else identity n

(==>&) :: Parse a -> Parse b -> Parse b
p ==>& f = p ==> \_ -> f

skipSpaces :: Parse ()
skipSpaces = parseWhileWith w2c isSpace ==>& identity ()

assert :: Bool -> String -> Parse ()
assert True  _   = identity ()
assert False err = bail err
```

The `(==>&)` combinator chains parsers such as `(==>)`, but the righthand side ignores the result from the left. The `assert` function lets us check a property and abort parsing with a useful error message if the property is `False`.

Notice how few of the functions that we have written make any reference to the current parsing state. Most notably, where our old `parseP5` function explicitly passed two-tuples down the chain of dataflow, all of the state management in `parseRawPGM` is hidden from us.

Of course, we can't completely avoid inspecting and modifying the parsing state. Here's a case in point, the last of the helper functions needed by `parseRawPGM`:

```
-- file: ch10/Parse.hs
parseBytes :: Int -> Parse L.ByteString
parseBytes n =
    getState ==> \st ->
    let n' = fromIntegral n
        (h, t) = L.splitAt n' (string st)
        st' = st { offset = offset st + L.length h, string = t }
    in putState st' ==>&
       assert (L.length h == n') "end of input" ==>&
       identity h
```

Future Directions

Our main theme in this chapter has been abstraction. We found passing explicit state down a chain of functions to be unsatisfactory, so we abstracted this detail away. We noticed some recurring needs as we worked out our parsing code, and abstracted those into common functions. Along the way, we introduced the notion of a functor, which offers a generalized way to map over a parameterized type.

We will revisit parsing in Chapter 16, when we discuss `Parsec`, a widely used and flexible parsing library. And in Chapter 14, we will return to our theme of abstraction, where we will find that much of the code that we have developed in this chapter can be further simplified by the use of monads.

For efficiently parsing binary data represented as a `ByteString`, a number of packages are available via the Hackage package database. At the time of this writing, the most popular is `binary`, which is easy to use and offers high performance.

EXERCISES

1. Write a parser for "plain" PGM files.
2. In our description of "raw" PGM files, we omitted a small detail. If the "maximum gray" value in the header is less than 256, each pixel is represented by a single byte. However, it can range up to 65,535, in which case, each pixel will be represented by 2 bytes, in big-endian order (most significant byte first).

   Rewrite the raw PGM parser to accommodate both the single- and double-byte pixel formats.
3. Extend your parser so that it can identify a raw or plain PGM file, and then parse the appropriate file type.