```
main = do
  s <- readFile "somefile"
  let i = f s
  putStrLn (show i)
```

Here, we use the arrow convention to “get the string out of the IO action” and then apply `f` to the string (called `s`). We then, for example, print `i` to the screen. Note that the **let** here doesn’t have a corresponding **in**. This is because we are in a **do** block. Also note that we don’t write `i <- f s` because `f` is just a normal function, not an IO action.

4.4.4 Explicit Type Declarations

It is sometimes desirable to explicitly specify the types of some elements or functions, for one (or more) of the following reasons:

- Clarity
- Speed
- Debugging

Some people consider it good software engineering to specify the types of all top-level functions. If nothing else, if you’re trying to compile a program and you get type errors that you cannot understand, if you declare the types of some of your functions explicitly, it may be easier to figure out where the error is.

Type declarations are written separatly from the function definition. For instance, we could explicitly type the function `square` as in the following code (an explicitly declared type is called a *type signature*):

```
square :: Num a => a -> a
square x = x*x
```

These two lines do not even have to be next to eachother. However, the type that you specify must match the inferred type of the function definition (or be more specific). In this definition, you could apply `square` to anything which is an instance of **Num**: Int, Double, etc. However, if you knew apriori that `square` were only going to be applied to value of type Int, you could *refine* its type as:

```
square :: Int -> Int
square x = x*x
```

Now, you could only apply `square` to values of type Int. Moreover, with this definition, the compiler doesn’t have to generate the general code specified in the original function definition since it knows you will only apply `square` to Ints, so it may be able to generate faster code.

If you have extensions turned on ("-98" in Hugs or "-fglasgow-exts" in GHC(i)), you can also add a type signature to expressions and not just functions. For instance, you could write:

```
square (x :: Int) = x*x
```

which tells the compiler that `x` is an Int; however, it leaves the compiler alone to infer the type of the rest of the expression. What is the type of `square` in this example? Make your guess then you can check it either by entering this code into a file and loading it into your interpreter or by asking for the type of the expression:

```
Prelude> :t (\(x :: Int) -> x*x)
```

since this lambda abstraction is equivalent to the above function declaration.

4.4.5 Functional Arguments

In Section 3.3 we saw examples of functions taking other functions as arguments. For instance, `map` took a function to apply to each element in a list, `filter` took a function that told it which elements of a list to keep, and `foldl` took a function which told it how to combine list elements together. As with every other function in Haskell, these are well-typed.

Let's first think about the `map` function. It's job is to take a list of elements and produce another list of elements. These two lists don't necessarily have to have the same types of elements. So `map` will take a value of type $[a]$ and produce a value of type $[b]$. How does it do this? It uses the user-supplied function to convert. In order to convert an a to a b, this function must have type $a \to b$. Thus, the type of `map` is $(a \to b) \to [a] \to [b]$, which you can verify in your interpreter with ":t".

We can apply the same sort of analysis to `filter` and discern that it has type $(a \to \mathsf{Bool}) \to [a] \to [a]$. As we presented the `foldl` function, you might be tempted to give it type $(a \to a \to a) \to a \to [a] \to a$, meaning that you take a function which combines two as into another one, an initial value of type a, a list of as to produce a final value of type a. In fact, `foldl` has a more general type: $(a \to b \to a) \to a \to [b] \to a$. So it takes a function which turn an a and a b into an a, an initial value of type a and a list of bs. It produces an a.

To see this, we can write a function `count` which counts how many members of a list satisfy a given constraint. You can of course you `filter` and `length` to do this, but we will also do it using `foldr`:

```
module Count
    where

import Char
```

```
count1 p l = length (filter p l)
count2 p l = foldr (\x c -> if p x then c+1 else c) 0 l
```

The functioning of `count1` is simple. It filters the list `l` according to the predicate `p`, then takes the length of the resulting list. On the other hand, `count2` uses the intial value (which is an integer) to hold the current count. For each element in the list `l`, it applies the lambda expression shown. This takes two arguments, `c` which holds the current count and `x` which is the current element in the list that we're looking at. It checks to see if `p` holds about `x`. If it does, it returns the new value `c+1`, increasing the count of elements for which the predicate holds. If it doesn't, it just returns `c`, the old count.

Exercises

Exercise 4.3 *Figure out for yourself, and then verify the types of the following expressions, if they have a type. Also note if the expression is a type error:*

1. `\x -> [x]`
2. `\x y z -> (x,y:z:[])`
3. `\x -> x + 5`
4. `\x -> "hello, world"`
5. `\x -> x 'a'`
6. `\x -> x x`
7. `\x -> x + x`

4.5 Data Types

Tuples and lists are nice, common ways to define structured values. However, it is often desirable to be able to define our own data structures and functions over them. So-called "datatypes" are defined using the **data** keyword.

4.5.1 Pairs

For instance, a definition of a pair of elements (much like the standard, build-in pair type) could be:

```
data Pair a b = Pair a b
```

Let's walk through this code one word at a time. First we say "data" meaning that we're defining a datatype. We then give the name of the datatype, in this case, "Pair." The "a" and "b" that follow "Pair" are type parameters, just like the "a" is the type of the function `map`. So up until this point, we've said that we're going to define a data structure called "Pair" which is parameterized over two types, a and b.

After the equals sign, we specify the *constructors* of this data type. In this case, there is a single constructor, "Pair" (this doesn't necessarily have to have the same name as the type, but in this case it seems to make more sense). After this pair, we again write "a b", which means that in order to construct a `Pair` we need two values, one of type a and one of type b.

This definition introduces a function, `Pair :: a -> b -> Pair a b` that you can use to construct `Pair`s. If you enter this code into a file and load it, you can see how these are constructed:

```
Datatypes> :t Pair
Pair :: a -> b -> Pair a b
Datatypes> :t Pair 'a'
Pair 'a' :: a -> Pair Char a
Datatypes> :t Pair 'a' "Hello"
:t Pair 'a' "Hello"
Pair 'a' "Hello" :: Pair Char [Char]
```

So, by giving `Pair` two values, we have completely constructed a value of type Pair. We can write functions involving pairs as:

```
pairFst (Pair x y) = x
pairSnd (Pair x y) = y
```

In this, we've used the *pattern matching* capabilities of Haskell to look at a pair an extract values from it. In the definition of `pairFst` we take an entire `Pair` and extract the first element; similarly for `pairSnd`. We'll discuss pattern matching in much more detail in Section 7.4.

Exercises

Exercise 4.4 *Write a data type declaration for* `Triple`*, a type which contains three elements, all of different types. Write functions* `tripleFst`*,* `tripleSnd` *and* `tripleThr` *to extract respectively the first, second and third elements.*

Exercise 4.5 *Write a datatype* `Quadruple` *which holds four elements. However, the first two elements must be the same type and the last two elements must be the same type. Write a function* `firstTwo` *which returns a list containing the first two elements and a function* `lastTwo` *which returns a list containing the last two elements. Write type signatures for these functions*

4.5.2 Multiple Constructors

We have seen an example of the data type with one constructor: `Pair`. It is also possible (and extremely useful) to have multiple constructors.

Let us consider a simple function which searches through a list for an element satisfying a given predicate and then returns the first element satisfying that predicate. What should we do if none of the elements in the list satisfy the predicate? A few options are listed below:

- Raise an error
- Loop indefinitely
- Write a check function
- Return the first element
- ...

Raising an error is certainly an option (see Section 10.1 to see how to do this). The problem is that it is difficult/impossible to recover from such errors. Looping indefinitely is possible, but not terribly useful. We could write a sister function which checks to see if the list contains an element satisfying a predicate and leave it up to the user to always use this function first. We could return the first element, but this is very ad-hoc and difficult to remember.

The fact that there is no basic option to solve this problem simply means we have to think about it a little more. What are we trying to do? We're trying to write a function which might succeed and might not. Furthermore, if it does succeed, it returns some sort of value. Let's write a datatype:

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

This is one of the most common datatypes in Haskell and is defined in the Prelude.

Here, we're saying that there are two possible ways to create something of type `Maybe a`. The first is to use the nullary constructor `Nothing`, which takes no arguments (this is what "nullary" means). The second is to use the constructor `Just`, together with a value of type `a`.

The `Maybe` type is useful in all sorts of circumstances. For instance, suppose we want to write a function (like `head`) which returns the first element of a given list. However, we don't want the program to die if the given list is empty. We can accomplish this with a function like:

```
firstElement :: [a] -> Maybe a
firstElement []     = Nothing
firstElement (x:xs) = Just x
```

The type signature here says that `firstElement` takes a list of `a`s and produces something with type `Maybe a`. In the first line of code, we match against the empty list `[]`. If this match succeeds (i.e., the list is, in fact, empty), we return `Nothing`. If the first match fails, then we try to match against `x:xs` which must succeed. In this case, we return `Just x`.

For our `findElement` function, we represent failure by the value `Nothing` and success with value `a` by `Just a`. Our function might look something like this:

```
findElement :: (a -> Bool) -> [a] -> Maybe a
findElement p [] = Nothing
findElement p (x:xs) =
    if p x then Just x
    else findElement p xs
```

The first line here gives the type of the function. In this case, our first argument is the predicate (and takes an element of type `a` and returns `True` if and only if the element satisfies the predicate); the second argument is a list of `a`s. Our return value is *maybe* an `a`. That is, if the function succeeds, we will return `Just a` and if not, `Nothing`.

Another useful datatype is the `Either` type, defined as:

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

This is a way of expressing alternation. That is, something of type `Either a b` is *either* a value of type `a` (using the `Left` constructor) or a value of type `b` (using the `Right` constructor).

Exercises

Exercise 4.6 *Write a datatype* `Tuple` *which can hold one, two, three or four elements, depending on the constructor (that is, there should be four constructors, one for each number of arguments). Also provide functions* `tuple1` *through* `tuple4` *which take a tuple and return* `Just` *the value in that position, or* `Nothing` *if the number is invalid (i.e., you ask for the* `tuple4` *on a tuple holding only two elements).*

Exercise 4.7 *Based on our definition of* `Tuple` *from the previous exercise, write a function which takes a* `Tuple` *and returns either the value (if it's a one-tuple), a Haskell-pair (i.e.,* `('a',5)`*) if it's a two-tuple, a Haskell-triple if it's a three-tuple or a Haskell-quadruple if it's a four-tuple. You will need to use the* `Either` *type to represent this.*

4.5.3 Recursive Datatypes

We can also define *recursive datatypes.* These are datatypes whose definitions are based on themselves. For instance, we could define a list datatype as:

```
data List a = Nil
            | Cons a (List a)
```

In this definition, we have defined what it means to be of type `List a`. We say that a list is either empty (`Nil`) or it's the `Cons` of a value of type `a` and another value of type `List a`. This is almost identical to the actual definition of the list datatype in Haskell, except that uses special syntax where `[]` corresponds to `Nil` and `:` corresponds to `Cons`. We can write our own `length` function for our lists as:

```
listLength Nil = 0
listLength (Cons x xs) = 1 + listLength xs
```

This function is slightly more complicated and uses *recursion* to calculate the length of a `List`. The first line says that the length of an empty list (a `Nil`) is 0. This much is obvious. The second line tells us how to calculate the length of a non-empty list. A non-empty list must be of the form `Cons x xs` for some values of `x` and `xs`. We know that `xs` is another list and we know that whatever the length of the current list is, it's the length of its tail (the value of `xs`) plus one (to account for `x`). Thus, we apply the `listLength` function to `xs` and add one to the result. This gives us the length of the entire list.

Exercises

Exercise 4.8 *Write functions* `listHead`, `listTail`, `listFoldl` *and* `listFoldr` *which are equivalent to their Prelude twins, but function on our* `List` *datatype. Don't worry about exceptional conditions on the first two.*

4.5.4 Binary Trees

We can define datatypes that are more complicated than lists. Suppose we want to define a structure that looks like a binary tree. A binary tree is a structure that has a single root node; each node in the tree is either a "leaf" or a "branch." If it's a leaf, it holds a value; if it's a branch, it holds a value and a left child and a right child. Each of these children is another node. We can define such a data type as:

```
data BinaryTree a
    = Leaf a
    | Branch (BinaryTree a) a (BinaryTree a)
```

In this datatype declaration we say that a `BinaryTree` of `as` is either a `Leaf` which holds an `a`, or it's a branch with a left child (which is a `BinaryTree` of `as`), a node value (which is an a), and a right child (which is also a `BinaryTree` of as). It is simple to modify the `listLength` function so that instead of calculating the length of lists, it calculates the number of nodes in a `BinaryTree`. Can you figure out how? We can call this function `treeSize`. The solution is given below:

```
treeSize (Leaf x) = 1
treeSize (Branch left x right) =
  1 + treeSize left + treeSize right
```

Here, we say that the size of a leaf is 1 and the size of a branch is the size of its left child, plus the size of its right child, plus one.

Exercises

Exercise 4.9 *Write a function* `elements` *which returns the elements in a* `BinaryTree` *in a bottom-up, left-to-right manner (i.e., the first element returned in the left-most leaf, followed by its parent's value, followed by the other child's value, and so on). The result type should be a normal Haskell list.*

Exercise 4.10 *Write a fold function for* `BinaryTree`*s and rewrite* `elements` *in terms of it (call the new one* `elements2`*).*

4.5.5 Enumerated Sets

You can also use datatypes to define things like enumerated sets, for instance, a type which can only have a constrained number of values. We could define a color type:

```
data Color
    = Red
    | Orange
    | Yellow
    | Green
    | Blue
    | Purple
    | White
    | Black
```

This would be sufficient to deal with simple colors. Suppose we were using this to write a drawing program, we could then write a function to convert between a Color and a RGB triple. We can write a `colorToRGB` function, as:

```
colorToRGB Red    = (255,0,0)
colorToRGB Orange = (255,128,0)
colorToRGB Yellow = (255,255,0)
colorToRGB Green  = (0,255,0)
colorToRGB Blue   = (0,0,255)
colorToRGB Purple = (255,0,255)
colorToRGB White  = (255,255,255)
colorToRGB Black  = (0,0,0)
```

If we wanted also to allow the user to define his own custom colors, we could change the Color datatype to something like:

```
data Color
    = Red
    | Orange
    | Yellow
    | Green
    | Blue
    | Purple
    | White
    | Black
    | Custom Int Int Int  -- R G B components
```

And add a final definition for `colorToRGB`:

```
colorToRGB (Custom r g b) = (r,g,b)
```

4.5.6 The Unit type

A final useful datatype defined in Haskell (from the Prelude) is the unit type. It's definition is:

```
data () = ()
```

The only true value of this type is `()`. This is essentially the same as a *void* type in a langauge like C or Java and will be useful when we talk about IO in Chapter 5.

We'll dwell much more on data types in Sections 7.4 and 8.3.

4.6 Continuation Passing Style

There is a style of functional programming called "Continuation Passing Style" (also simply "CPS"). The idea behind CPS is to pass around as a function argument what to do next. I will handwave through an example which is too complex to write out at this point and then give a real example, though one with less motivation.

Consider the problem of parsing. The idea here is that we have a sequence of tokens (words, letters, whatever) and we want to ascribe structure to them. The task of converting a string of Java tokens to a Java abstract syntax tree is an example of a parsing problem. So is the task of taking an English sentence and creating a parse tree (though the latter is quite a bit harder).

Suppose we're parsing something like C or Java where functions take arguments in parentheses. But for simplicity, assume they are not separated by commas. That is, a function call looks like `myFunction(x y z)`. We want to convert this into something like a pair containing first the string "myFunction" and then a list with three string elements: "x", "y" and "z".

The general approach to solving this would be to write a function which parses function calls like this one. First it would look for an identifier ("myFunction"), then for an open parenthesis, then for zero or more identifiers, then for a close parenthesis.

One way to do this would be to have two functions:

```
parseFunction   ::
    [Token] -> Maybe ((String, [String]), [Token])

parseIdentifier ::
    [Token] -> Maybe (String, [Token])
```

The idea would be that if we call `parseFunction`, if it doesn't return `Nothing`, then it returns the pair described earlier, together with whatever is left after parsing the function. Similarly, `parseIdentifier` will parse one of the arguments. If it returns `Nothing`, then it's not an argument; if it returns `Just` something, then that something is the argument paired with the rest of the tokens.

What the `parseFunction` function would do is to parse an identifier. If this fails, it fails itself. Otherwise, it continues and tries to parse a open parenthesis. If that succeeds, it repeatedly calls `parseIdentifier` until that fails. It then tries to parse a close parenthesis. If that succeeds, then it's done. Otherwise, it fails.

There is, however, another way to think about this problem. The advantage to this solution is that functions no longer need to return the remaining tokens (which tends to get ugly). Instead of the above, we write functions:

```
parseFunction   ::
    [Token] -> ((String, [String]) -> [Token] -> a) ->
    ([Token] -> a) -> a

parseIdentifier ::
    [Token] -> (String -> [Token] -> a) ->
    ([Token] -> a) -> a
```

Let's consider `parseIdentifier`. This takes three arguments: a list of tokens and two *continuations*. The first continuation is what to do when you succeed. The second continuation is what to do if you fail. What `parseIdentifier` does, then, is try to read an identifier. If this succeeds, it calls the first continuation with that identifier and the remaining tokens as arguments. If reading the identifier fails, it calls the second continuation with all the tokens.

Now consider `parseFunction`. Recall that it wants to read an identifier, an open parenthesis, zero or more identifiers and a close parenthesis. Thus, the first thing it does is call `parseIdentifier`. The first argument it gives is the list of tokens. The first continuation (which is what `parseIdentifier` should do if it succeeds) is in turn a function which will look for an open parenthesis, zero or more arguments and a close parethesis. The second argument (the failure argument) is just going to be the failure function given to `parseFunction`.

Now, we simply need to define this function which looks for an open parenthesis, zero or more arguments and a close parethesis. This is easy. We write a function which looks for the open parenthesis and then calls `parseIdentifier` with a success continuation that looks for more identifiers, and a "failure" continuation which looks for the close parenthesis (note that this failure doesn't really mean failure – it just means there are no more arguments left).

I realize this discussion has been quite abstract. I would willingly give code for all this parsing, but it is perhaps too complex at the moment. Instead, consider the problem of folding across a list. We can write a CPS fold as:

```
cfold' f z [] = z
cfold' f z (x:xs) = f x z (\y -> cfold' f y xs)
```

In this code, `cfold'` take a function `f` which takes three arguments, slightly different from the standard folds. The first is the current list element, `x`, the second is the accumulated element, `z`, and the third is the continuation: basicially, what to do *next*.

We can write a wrapper function for `cfold'` that will make it behave more like a normal fold:

```
cfold f z l = cfold' (\x t g -> f x (g t)) z l
```

We can test that this function behaves as we desire:

```
CPS> cfold (+) 0 [1,2,3,4]
10
CPS> cfold (:) [] [1,2,3]
[1,2,3]
```

One thing that's nice about formulating `cfold` in terms of the helper function `cfold'` is that we can use the helper function directly. This enables us to change, for instance, the evaluation order of the fold very easily:

```
CPS> cfold' (\x t g -> (x : g t)) [] [1..10]
[1,2,3,4,5,6,7,8,9,10]
CPS> cfold' (\x t g -> g (x : t)) [] [1..10]
[10,9,8,7,6,5,4,3,2,1]
```

The only difference between these calls to `cfold'` is whether we call the continuation before or after constructing the list. As it turns out, this slight difference changes the behavior for being like `foldr` to being like `foldl`. We can evaluate both of these calls as follows (let `f` be the folding function):

```
     cfold' (\x t g -> (x : g t)) [] [1,2,3]
==>  cfold' f [] [1,2,3]
==>  f 1 [] (\y -> cfold' f y [2,3])
==>  1 : ((\y -> cfold' f y [2,3]) [])
==>  1 : (cfold' f [] [2,3])
==>  1 : (f 2 [] (\y -> cfold' f y [3]))
==>  1 : (2 : ((\y -> cfold' f y [3]) []))
==>  1 : (2 : (cfold' f [] [3]))
==>  1 : (2 : (f 3 [] (\y -> cfold' f y [])))
==>  1 : (2 : (3 : (cfold' f [] [])))
==>  1 : (2 : (3 : []))
==>  [1,2,3]

     cfold' (\x t g -> g (x:t)) [] [1,2,3]
==>  cfold' f [] [1,2,3]
==>  (\x t g -> g (x:t)) 1 [] (\y -> cfold' f y [2,3])
==>  (\g -> g [1]) (\y -> cfold' f y [2,3])
==>  (\y -> cfold' f y [2,3]) [1]
==>  cfold' f [1] [2,3]
==>  (\x t g -> g (x:t)) 2 [1] (\y -> cfold' f y [3])
==>  cfold' f (2:[1]) [3]
==>  cfold' f [2,1] [3]
==>  (\x t g -> g (x:t)) 3 [2,1] (\y -> cfold' f y [])
==>  cfold' f (3:[2,1]) []
==>  [3,2,1]
```

In general, continuation passing style is a very powerful abstraction, though it can be difficult to master. We will revisit the topic more thoroughly later in the book.

Exercises

Exercise 4.11 *Test whether the CPS-style fold mimicks either of* `foldr` *and* `foldl`. *If not, where is the difference?*

Exercise 4.12 *Write* `map` *and* `filter` *using continuation passing style.*

Chapter 5

Basic Input/Output

As we mentioned earlier, it is difficult to think of a good, clean way to integrate operations like input/output into a pure functional language. Before we give the solution, let's take a step back and think about the difficulties inherent in such a task.

Any IO library should provide a host of functions, containing (at a minimum) operations like:

- print a string to the screen
- read a string from a keyboard
- write data to a file
- read data from a file

There are two issues here. Let's first consider the initial two examples and think about what their types should be. Certainly the first operation (I hesitate to call it a "function") should take a `String` argument and produce something, but what should it produce? It could produce a unit `()`, since there is essentially no return value from printing a string. The second operation, similarly, should return a `String`, but it doesn't seem to require an argument.

We want both of these operations to be functions, but they are by definition not functions. The item that reads a string from the keyboard cannot be a function, as it will not return the same `String` every time. And if the first function simply returns `()` every time, there should be no problem with replacing it with a function `f _ = ()`, due to referential transparency. But clearly this does not have the desired effect.

5.1 The RealWorld Solution

In a sense, the reason that these items are not functions is that they interact with the "real world." Their values depend directly on the real world. Supposing we had a type RealWorld, we might write these functions as having type:

```
printAString :: RealWorld -> String -> RealWorld
readAString  :: RealWorld -> (RealWorld, String)
```

That is, `printAString` takes a current state of the world and a string to print; it then modifies the state of the world in such a way that the string is now printed and returns this new value. Similarly, `readAString` takes a current state of the world and returns a *new* state of the world, paired with the String that was typed.

This would be a possible way to do IO, though it is more than somewhat unweildy. In this style (assuming an initial RealWorld state were an argument to `main`), our "Name.hs" program from Section 3.8 would look something like:

```
main rW =
  let rW' = printAString rW "Please enter your name: "
      (rW'',name) = readAString rW'
  in  printAString rW''
          ("Hello, " ++ name ++ ", how are you?")
```

This is not only hard to read, but prone to error, if you accidentally use the wrong version of the RealWorld. It also doesn't model the fact that the program below makes no sense:

```
main rW =
  let rW' = printAString rW "Please enter your name: "
      (rW'',name) = readAString rW'
  in  printAString rW'                 -- OOPS!
          ("Hello, " ++ name ++ ", how are you?")
```

In this program, the reference to `rW''` on the last line has been changed to a reference to `rW'`. It is completely unclear what this program should do. Clearly, it must read a string in order to have a value for `name` to be printed. But that means that the RealWorld has been updated. However, then we try to ignore this update by using an "old version" of the RealWorld. There is clearly something wrong happening here.

Suffice it to say that doing IO operations in a pure lazy functional language is not trivial.

5.2 Actions

The breakthrough for solving this problem came when Phil Wadler realized that monads would be a good way to think about IO computations. In fact, monads are able to express much more than just the simple operations described above; we can use them to express a variety of constructions like concurrence, exceptions, IO, non-determinism and much more. Moreover, there is nothing special about them; they can be defined *within* Haskell with no special handling from the compiler (though compilers often choose to optimize monadic operations).

As pointed out before, we cannot think of things like "print a string to the screen" or "read data from a file" as functions, since they are not (in the pure mathematical sense). Therefore, we give them another name: *actions*. Not only do we give them a special name, we give them a special type. One particularly useful action is `putStrLn`, which prints a string to the screen. This action has type:

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

As expected, `putStrLn` takes a string argument. What it returns is of type `IO ()`. This means that this function is actually an action (that is what the IO means). Furthermore, when this action is *evaluated* (or "run") , the result will have type ().

■ NOTE ■ Actually, this type means that `putStrLn` is an action *within the IO monad*, but we will gloss over this for now.

You can probably already guess the type of `getLine`:

```
getLine :: IO String
```

This means that `getLine` is an IO action that, when run, will have type String.

The question immediately arises: "how do you 'run' an action?". This is something that is left up to the compiler. You cannot actually run an action yourself; instead, a program is, itself, a single action that is run when the compiled program is executed. Thus, the compiler requires that the `main` function have type IO (), which means that it is an IO action that returns nothing. The compiled code then executes this action.

However, while you are not allowed to run actions yourself, you *are* allowed to `combine` actions. In fact, we have already seen one way to do this using the **do** notation (how to *really* do this will be revealed in Chapter 9). Let's consider the original name program:

```
main = do
  hSetBuffering stdin LineBuffering
  putStrLn "Please enter your name: "
  name <- getLine
  putStrLn ("Hello, " ++ name ++ ", how are you?")
```

We can consider the **do** notation as a way to combine a sequence of actions. Moreover, the `<-` notation is a way to get the value out of an action. So, in this program, we're sequencing four actions: setting buffering, a `putStrLn`, a `getLine` and another `putStrLn`. The `putStrLn` action has type String $\rightarrow$ IO (), so we provide it a String, so the fully applied action has type IO (). This is something that we are allowed to execute.

The `getLine` action has type IO String, so it is okay to execute it directly. However, in order to get the value out of the action, we write `name <- getLine`, which basically means "run `getLine`, and put the results in the variable called `name`."

Normal Haskell constructions like **if/then/else** and **case/of** can be used within the **do** notation, but you need to be somewhat careful. For instance, in our "guess the number" program, we have:

```
    do ...
       if (read guess) < num
         then do putStrLn "Too low!"
                 doGuessing num
         else if read guess > num
                then do putStrLn "Too high!"
                        doGuessing num
                else do putStrLn "You Win!"
```

If we think about how the **if/then/else** construction works, it essentially takes three arguments: the condition, the "then" branch, and the "else" branch. The condition needs to have type Bool, and the two branches can have any type, provided that they have the *same* type. The type of the entire **if/then/else** construction is then the type of the two branches.

In the outermost comparison, we have `(read guess) < num` as the condition. This clearly has the correct type. Let's just consider the "then" branch. The code here is:

```
              do putStrLn "Too low!"
                 doGuessing num
```

Here, we are sequencing two actions: `putStrLn` and `doGuessing`. The first has type IO (), which is fine. The second also has type IO (), which is fine. The type result of the entire computation is precisely the type of the final computation. Thus, the type of the "then" branch is also IO (). A similar argument shows that the type of the "else" branch is also IO (). This means the type of the entire **if/then/else** construction is IO (), which is just what we want.

■ NOTE ■ In this code, the last line is "`else do putStrLn "You Win!"`". This is somewhat overly verbose. In fact, "`else putStrLn "You Win!"`" would have been sufficient, since **do** is only necessary to sequence actions. Since we have only one action here, it is superfluous.

It is *incorrect* to think to yourself "Well, I already started a **do** block; I don't need another one," and hence write something like:

```
    do if (read guess) < num
         then putStrLn "Too low!"
              doGuessing num
         else ...
```

Here, since we didn't repeat the **do**, the compiler doesn't know that the `putStrLn` and `doGuessing` calls are supposed to be sequenced, and the compiler will think you're trying to call `putStrLn` with three arguments: the string, the function `doGuessing` and the integer `num`. It will certainly complain (though the error may be somewhat difficult to comprehend at this point).

We can write the same `doGuessing` function using a **case** statement. To do this, we first introduce the Prelude function `compare`, which takes two values of the same type (in the **Ord** class) and returns one of `GT`, `LT`, `EQ`, depending on whether the first is greater than, less than or equal to the second.

```
doGuessing num = do
  putStrLn "Enter your guess:"
  guess <- getLine
  case compare (read guess) num of
    LT -> do putStrLn "Too low!"
             doGuessing num
    GT -> do putStrLn "Too high!"
             doGuessing num
    EQ -> putStrLn "You Win!"
```

Here, again, the **do**s after the `->`s are necessary on the first two options, because we are sequencing actions.

If you're used to programming in an imperative language like C or Java, you might think that **return** will exit you from the current function. This is not so in Haskell. In Haskell, **return** simply takes a normal value (for instance, one of type IO Int) and makes it into an action that returns the given value (for instance, the value of type Int). In particular, in an imperative language, you might write this function as:

```
void doGuessing(int num) {
  print "Enter your guess:";
  int guess = atoi(readLine());
  if (guess == num) {
    print "You win!";
    return ();
  }

  // we won't get here if guess == num
  if (guess < num) {
    print "Too low!";
    doGuessing(num);
  } else {
    print "Too high!";
    doGuessing(num);
  }
}
```

Here, because we have the `return ()` in the first `if` match, we expect the code to exit there (and in mode imperative languages, it does). However, the equivalent code in Haskell, which might look something like:

```
doGuessing num = do
  putStrLn "Enter your guess:"
  guess <- getLine
  case compare (read guess) num of
    EQ -> do putStrLn "You win!"
             return ()

  -- we don't expect to get here unless guess == num
  if (read guess < num)
    then do print "Too low!";
            doGuessing
    else do print "Too high!";
            doGuessing
```

will not behave as you expect. First of all, if you guess correctly, it will first print "You win!," but it won't exit, and it will check whether `guess` is less than `num`. Of course it is not, so the else branch is taken, and it will print "Too high!" and then ask you to guess again.

On the other hand, if you guess incorrectly, it will try to evaluate the case statement and get either `LT` or `GT` as the result of the `compare`. In either case, it won't have a pattern that matches, and the program will fail immediately with an exception.

Exercises

Exercise 5.1 *Write a program that asks the user for his or her name. If the name is one of Simon, John or Phil, tell the user that you think Haskell is a great programming language. If the name is Koen, tell them that you think debugging Haskell is fun (Koen Classen is one of the people who works on Haskell debugging); otherwise, tell the user that you don't know who he or she is.*
*Write two different versions of this program, one using **if** statements, the other using a **case** statement.*

5.3 The IO Library

The IO Library (available by **import**ing the `IO` module) contains many definitions, the most common of which are listed below:

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

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

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

hIsEOF       :: Handle -> IO Bool

hGetChar     :: Handle -> IO Char
hGetLine     :: Handle -> IO String
hGetContents :: Handle -> IO String

getChar      :: IO Char
getLine      :: IO String
getContents  :: IO String

hPutChar     :: Handle -> Char -> IO ()
hPutStr      :: Handle -> String -> IO ()
hPutStrLn    :: Handle -> String -> IO ()

putChar      :: Char -> IO ()
putStr       :: String -> IO ()
putStrLn     :: String -> IO ()

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

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

■ NOTE ■ The type FilePath is a *type synonym* for String. That is, there is no difference between FilePath and String. So, for instance, the `readFile` function takes a String (the file to read) and returns an action that, when run, produces the contents of that file. See Section 8.1 for more about type synonyms.

Most of these functions are self-explanatory. The `openFile` and `hClose` functions open and close a file, respectively, using the IOMode argument as the mode for opening the file. `hIsEOF` tests for end-of file. `hGetChar` and `hGetLine` read a character or line (respectively) from a file. `hGetContents` reads the entire file. The `getChar`, `getLine` and `getContents` variants read from standard input. `hPutChar` prints a character to a file; `hPutStr` prints a string; and `hPutStrLn` prints a string with a newline character at the end. The variants without the `h` prefix work on standard output. The `readFile` and `writeFile` functions read an entire file without having to open it first.

The `bracket` function is used to perform actions safely. Consider a function that opens a file, writes a character to it, and then closes the file. When writing such a function, one needs to be careful to ensure that, if there were an error at some point, the file is still successfully closed. The `bracket` function makes this easy. It takes three arguments: The first is the action to perform at the beginning. The second is the action to perform at the end, regardless of whether there's an error or not. The third is the action to perform in the middle, which might result in an error. For instance, our character-writing function might look like:

```
writeChar :: FilePath -> Char -> IO ()
writeChar fp c =
    bracket
      (openFile fp ReadMode)
      hClose
      (\h -> hPutChar h c)
```

This will open the file, write the character and then close the file. However, if writing the character fails, `hClose` will still be executed, and the exception will be reraised afterwards. That way, you don't need to worry too much about catching the exceptions and about closing all of your handles.

5.4 A File Reading Program

We can write a simple program that allows a user to read and write files. The interface is admittedly poor, and it does not catch all errors (try reading a non-existant file). Nevertheless, it should give a fairly complete example of how to use IO. Enter the following code into "FileRead.hs," and compile/run:

```
module Main
    where

import IO

main = do
  hSetBuffering stdin LineBuffering
  doLoop

doLoop = do
  putStrLn "Enter a command rFN wFN or q to quit:"
  command <- getLine
  case command of
    'q':_ -> return ()
    'r':filename -> do putStrLn ("Reading " ++ filename)
                       doRead filename
                       doLoop
    'w':filename -> do putStrLn ("Writing " ++ filename)
                       doWrite filename
                       doLoop
    _            -> doLoop
```

```
doRead filename =
    bracket (openFile filename ReadMode) hClose
            (\h -> do contents <- hGetContents h
                      putStrLn "The first 100 chars:"
                      putStrLn (take 100 contents))

doWrite filename = do
  putStrLn "Enter text to go into the file:"
  contents <- getLine
  bracket (openFile filename WriteMode) hClose
          (\h -> hPutStrLn h contents)
```

What does this program do? First, it issues a short string of instructions and reads a command. It then performs a **case** switch on the command and checks first to see if the first character is a 'q.' If it is, it returns a value of unit type.

■ NOTE ■ The `return` function is a function that takes a value of type a and returns an action of type IO a. Thus, the type of `return ()` is IO ().

If the first character of the command wasn't a 'q,' the program checks to see if it was an 'r' followed by some string that is bound to the variable `filename`. It then tells you that it's reading the file, does the read and runs `doLoop` again. The check for 'w' is nearly identical. Otherwise, it matches _, the wildcard character, and loops to `doLoop`.

The `doRead` function uses the `bracket` function to make sure there are no problems reading the file. It opens a file in ReadMode, reads its contents and prints the first 100 characters (the `take` function takes an integer n and a list and returns the first n elements of the list).

The `doWrite` function asks for some text, reads it from the keyboard, and then writes it to the file specified.

■ NOTE ■ Both `doRead` and `doWrite` could have been made simpler by using `readFile` and `writeFile`, but they were written in the extended fashion to show how the more complex functions are used.

The only major problem with this program is that it will die if you try to read a file that doesn't already exists or if you specify some bad filename like `*\^#_@`. You may think that the calls to `bracket` in `doRead` and `doWrite` should take care of this, but they don't. They only catch exceptions within the main body, not within the startup or shutdown functions (`openFile` and `hClose`, in these cases). We would need to catch exceptions raised by `openFile`, in order to make this complete. We will do this when we talk about exceptions in more detail in Section 10.1.

Exercises

Exercise 5.2 *Write a program that first asks whether the user wants to read from a file, write to a file or quit. If the user responds quit, the program should exit. If he responds read, the program should ask him for a file name and print that file to the screen (if the file doesn't exist, the program may crash). If he responds write, it should ask him for a file name and then ask him for text to write to the file, with "." signaling completion. All but the "." should be written to the file.*
For example, running this program might produce:

```
Do you want to [read] a file, [write] a file or [quit]?
read
Enter a file name to read:
foo
...contents of foo...
Do you want to [read] a file, [write] a file or [quit]?
write
Enter a file name to write:
foo
Enter text (dot on a line by itself to end):
this is some
text for
foo
.
Do you want to [read] a file, [write] a file or [quit]?
read
Enter a file name to read:
foo
this is some
text for
foo
Do you want to [read] a file, [write] a file or [quit]?
read
Enter a file name to read:
foof
Sorry, that file does not exist.
Do you want to [read] a file, [write] a file or [quit]?
blech
I don't understand the command blech.
Do you want to [read] a file, [write] a file or [quit]?
quit
Goodbye!
```

Chapter 6

Modules

In Haskell, program subcomponents are divided into modules. Each module sits in its own file and the name of the module should match the name of the file (without the ".hs" extension, of course), if you wish to ever use that module in a larger program.

For instance, suppose I am writing a game of poker. I may wish to have a separate module called "Cards" to handle the generation of cards, the shuffling and the dealing functions, and then use this "Cards" module in my "Poker" modules. That way, if I ever go back and want to write a blackjack program, I don't have to rewrite all the code for the cards; I can simply import the old "Cards" module.

6.1 Exports

Suppose as suggested we are writing a cards module. I have left out the implementation details, but suppose the skeleton of our module looks something like this:

```
module Cards
    where

data Card = ...
data Deck = ...

newDeck :: ... -> Deck
newDeck = ...

shuffle :: ... -> Deck -> Deck
shuffle = ...

-- 'deal deck n' deals 'n' cards from 'deck'
deal :: Deck -> Int -> [Card]
deal deck n = dealHelper deck n []
```

```
dealHelper = ...
```

In this code, the function `deal` calls a helper function `dealHelper`. The implementation of this helper function is very dependent on the exact data structures you used for `Card` and `Deck` so we don't want other people to be able to call this function. In order to do this, we create an *export list*, which we insert just after the module name declaration:

```
module Cards ( Card(),
               Deck(),
               newDeck,
               shuffle,
               deal
             )
    where

...
```

Here, we have specified exactly what functions the module exports, so people who use this module won't be able to access our `dealHelper` function. The `()` after `Card` and `Deck` specify that we are exporting the *type* but none of the constructors. For instance if our definition of `Card` were:

```
data Card = Card Suit Face
data Suit = Hearts
          | Spades
          | Diamonds
          | Clubs
data Face = Jack
          | Queen
          | King
          | Ace
          | Number Int
```

Then users of our module would be able to *use* things of type `Card`, but wouldn't be able to construct their own `Card`s and wouldn't be able to extract any of the suit/face information stored in them.

If we wanted users of our module to be able to access all of this information, we would have to specify it in the export list:

```
module Cards ( Card(Card),
               Suit(Hearts,Spades,Diamonds,Clubs),
               Face(Jack,Queen,King,Ace,Number),
               ...
             )
```

```
    where

...
```

This can get frustrating if you’re exporting datatypes with many constructors, so if you want to export them all, you can simply write `(..)`, as in:

```
module Cards ( Card(..),
               Suit(..),
               Face(..),
               ...
             )
    where

...
```

And this will automatically export all the constructors.

6.2 Imports

There are a few idiosyncracies in the module import system, but as long as you stay away from the corner cases, you should be fine. Suppose, as before, you wrote a module called “Cards” which you saved in the file “Cards.hs”. You are now writing your poker module and you want to *import* all the definitions from the “Cards” module. To do this, all you need to do is write:

```
module Poker
    where

import Cards
```

This will enable to you use any of the functions, types and constructors exported by the module “Cards”. You may refer to them simply by their name in the “Cards” module (as, for instance, `newDeck`), or you may refer to them explicitely as imported from “Cards” (as, for instance, `Cards.newDeck`). It may be the case that two module export functions or types of the same name. In these cases, you can import one of the modules **qualified** which means that you would no longer be able to simply use the `newDeck` format but must use the longer `Cards.newDeck` format, to remove ambiguity. If you wanted to import “Cards” in this qualified form, you would write:

```
import qualified Cards
```

Another way to avoid problems with overlapping function definitions is to import only certain functions from modules. Suppose we knew the only function from “Cards” that we wanted was `newDeck`, we could import only this function by writing:

```
import Cards (newDeck)
```

On the other hand, suppose we knew that that the `deal` function overlapped with another module, but that we didn't need the "Cards" version of that function. We could hide the definition of `deal` and import everything else by writing:

```
import Cards hiding (deal)
```

Finally, suppose we want to import "Cards" as a qualified module, but don't want to have to type `Cards.` out all the time and would rather just type, for instance, `C.` – we could do this using the **as** keyword:

```
import qualified Cards as C
```

These options can be mixed and matched – you can give explicit import lists on qualified/as imports, for instance.

6.3 Hierarchical Imports

Though technically not part of the Haskell 98 standard, most Haskell compilers support hierarchical imports. This was designed to get rid of clutter in the directories in which modules are stored. Hierarchical imports allow you to specify (to a certain degree) where in the directory structure a module exists. For instance, if you have a "haskell" directory on your computer and this directory is in your compiler's path (see your compiler notes for how to set this; in GHC it's "-i", in Hugs it's "-P"), then you can specify module locations in subdirectories to that directory.

Suppose instead of saving the "Cards" module in your general haskell directory, you created a directory specifically for it called "Cards". The full path of the `Cards.hs` file is then `haskell/Cards/Cards.hs` (or, for Windows `haskell\Cards\Cards.hs`). If you then change the name of the Cards module to "Cards.Cards", as in:

```
module Cards.Cards(...)
    where

...
```

You could then import it in any module, regardless of this module's directory, as:

```
import Cards.Cards
```

If you start importing these module qualified, I highly recommend using the **as** keyword to shorten the names, so you can write:

```
import qualified Cards.Cards as Cards

... Cards.newDeck ...
```

instead of:

```
import qualified Cards.Cards

... Cards.Cards.newDeck ...
```

which tends to get ugly.

6.4 Literate Versus Non-Literate

The idea of literate programming is a relatively simple one, but took quite a while to become popularized. When we think about programming, we think about the *code* being the default mode of entry and *comments* being secondary. That is, we write code without any special annotation, but comments are annotated with either `--` or {`- ... -`}. Literate programming swaps these preconceptions.

There are two types of literate programs in Haskell; the first uses so-called Bird-scripts and the second uses LaTeX-style markup. Each will be discussed individually. No matter which you use, literate scripts must have the extension lhs instead of hs to tell the compiler that the program is written in a literate style.

6.4.1 Bird-scripts

In a Bird-style literate program, comments are default and code is introduced with a leading greater-than sign (">"). Everything else remains the same. For example, our Hello World program would be written in Bird-style as:

```
This is a simple (literate!) Hello World program.

> module Main
>     where

All our main function does is print a string:

> main = putStrLn "Hello World"
```

Note that the spaces between the lines of code and the "comments" are necessary (your compiler will probably complain if you are missing them). When compiled or loaded in an interpreter, this program will have the exact same properties as the non-literate version from Section 3.4.

6.4.2 LaTeX-scripts

LaTeX is a text-markup language very popular in the academic community for publishing. If you are unfamiliar with LaTeX, you may not find this section terribly useful.

Again, a literate Hello World program written in LaTeX-style would look like:

```
This is another simple (literate!) Hello World program.

\begin{code}
module Main
    where
\end{code}

All our main function does is print a string:

\begin{code}
main = putStrLn "Hello World"
\end{code}
```

In LaTeX-style scripts, the blank lines are *not* necessary.

Chapter 7

Advanced Features

Discussion

7.1 Sections and Infix Operators

We've already seen how to double the values of elements in a list using `map`:

```
Prelude> map (\x -> x*2) [1,2,3,4]
[2,4,6,8]
```

However, there is a more concise way to write this:

```
Prelude> map (*2) [1,2,3,4]
[2,4,6,8]
```

This type of thing can be done for any infix function:

```
Prelude> map (+5) [1,2,3,4]
[6,7,8,9]
Prelude> map (/2) [1,2,3,4]
[0.5,1.0,1.5,2.0]
Prelude> map (2/) [1,2,3,4]
[2.0,1.0,0.666667,0.5]
```

You might be tempted to try to subtract values from elements in a list by mapping `-2` across a list. This won't work, though, because while the `+` in `+2` is parsed as the standard plus operator (as there is no ambiguity), the `-` in `-2` is interpreted as the unary minus, not the binary minus. Thus `-2` here is the *number* -2, not the function $\lambda x.x - 2$.

In general, these are called sections. For binary infix operators (like `+`), we can cause the function to become prefix by enclosing it in paretheses. For example:

```
Prelude> (+) 5 3
8
Prelude> (-) 5 3
2
```

Additionally, we can provide either of its argument to make a section. For example:

```
Prelude> (+5) 3
8
Prelude> (/3) 6
2.0
Prelude> (3/) 6
0.5
```

Non-infix functions can be made infix by enclosing them in backquotes (“`’). For example:

```
Prelude> (+2) ‘map‘ [1..10]
[3,4,5,6,7,8,9,10,11,12]
```

7.2 Local Declarations

Recall back from Section 3.5, there are many computations which require using the result of the same computation in multiple places in a function. There, we considered the function for computing the roots of a quadratic polynomial:

```
roots a b c =
    ((-b + sqrt(b*b - 4*a*c)) / (2*a),
     (-b - sqrt(b*b - 4*a*c)) / (2*a))
```

In addition to the **let** bindings introduced there, we can do this using a **where** clause. `where` clauses come immediately after function definitions and introduce a new level of layout (see Section 7.11). We write this as:

```
roots a b c =
    ((-b + det) / (2*a), (-b - det) / (2*a))
    where det = sqrt(b*b-4*a*c)
```

Any values defined in a **where** clause *shadow* any other values with the same name. For instance, if we had the following code block:

```
det = "Hello World"

roots a b c =
    ((-b + det) / (2*a), (-b - det) / (2*a))
    where det = sqrt(b*b-4*a*c)

f _ = det
```

The value of `roots` doesn't notice the top-level declaration of `det`, since it is shadowed by the local definition (the fact that the types don't match doesn't matter either). Furthermore, since `f` cannot "see inside" of `roots`, the only thing it knows about `det` is what is available at the top level, which is the string "Hello World." Thus, `f` is a function which takes any argument to that string.

Where clauses can contain any number of subexpressions, but they must be aligned for layout. For instance, we could also pull out the `2*a` computation and get the following code:

```
roots a b c =
    ((-b + det) / (a2), (-b - det) / (a2))
    where det = sqrt(b*b-4*a*c)
          a2 = 2*a
```

Sub-expressions in **where** clauses must come after function definitions. Sometimes it is more convenient to put the local definitions before the actual expression of the function. This can be done by using **let**/**in** clauses. We have already seen **let** clauses; **where** clauses are virtually identical to their **let** clause cousins except for their placement. The same `roots` function can be written using **let** as:

```
roots a b c =
    let det = sqrt (b*b - 4*a*c)
        a2 = 2*a
    in  ((-b + det) / a2, (-b - det) / a2)
```

Using a **where** clause, it looks like:

```
roots a b c = ((-b + det) / a2, (-b - det) / a2)
  where
    det = sqrt (b*b - 4*a*c)
    a2  = 2*a
```

These two types of clauses can be mixed (i.e., you can write a function which has both a **let** cause and a **where** clause). This is strongly advised *against*, as it tends to make code difficult to read. However, if you choose to do it, values in the **let** clause shadow those in the **where** clause. So if you define the function:

```
f x =
    let y = x+1
    in  y
    where y = x+2
```

The value of `f 5` is `6`, not `7`. Of course, I plead with you to never ever write code that looks like this. No one should have to remember this rule and by shadowing **where**-defined values in a **let** clause only makes your code difficult to understand.

In general, whether you should use **let** clauses or **where** clauses is largely a matter of personal preference. Usually, the names you give to the subexpressions should be sufficiently expressive that without reading their definitions any reader of your code should be able to figure out what they do. In this case, **where** clauses are probably more desirable because they allow the reader to see immediately what a function does. However, in real life, values are often given cryptic names. In which case **let** clauses may be better. Either is probably okay, though I think **where** clauses are more common.

7.3 Partial Application

Partial application is when you take a function which takes n arguments and you supply it with $< n$ of them. When discussing sections in Section 7.1, we saw a form of "partial application" in which functions like `+` were partially applied. For instance, in the expression `map (+1) [1,2,3]`, the section `(+1)` is a partial application of `+`. This is because `+` really takes two arguments, but we've only given it one.

eta reduction

Partial application is very common in function definitions and sometimes goes by the name "eta reduction". For instance, suppose we are writting a function `lcaseString` which converts a whole string into lower case. We could write this as:

```
lcaseString s = map toLower s
```

Here, there is no partial application (though you could argue that applying no arguments to `toLower` could be considered partial application). However, we notice that the application of `s` occurs at the end of both `lcaseString` and of `map toLower`. In fact, we can remove it by performing eta reduction, to get:

```
lcaseString = map toLower
```

Now, we have a partial application of `map`: it expects a function and a list, but we've only given it the function.

This all is related to type type of `map`, which is $(\mathsf{a} \rightarrow \mathsf{b}) \rightarrow ([\mathsf{a}] \rightarrow [\mathsf{b}])$, when parentheses are all included. In our case, `toLower` is of type $\mathsf{Char} \rightarrow \mathsf{Char}$. Thus, if we supply this function to `map`, we get a function of type $[\mathsf{Char}] \rightarrow [\mathsf{Char}]$, as desired.

Now, consider the task of converting a string to lowercase and remove all non letter characters. We might write this as:

```
lcaseLetters s = map toLower (filter isAlpha s)
```

But note that we can actually write this in terms of function composition:

```
lcaseLetters s = (map toLower . filter isAlpha) s
```

And again, we're left with an eta reducible function:

```
lcaseLetters = map toLower . filter isAlpha
```

Writing functions in this style is very common among advanced Haskell users. In fact it has a name: point-free programming (not to be confused with *pointless* programming). It is call point free because in the original definition of `lcaseLetters`, we can think of the value `s` as a point on which the function is operating. By removing the point from the function definition, we have a point-free function.

point-free programming

A function similar to `(.)` is `($)`. Whereas `(.)` is function composition, `($)` is function application. The definition of `($)` from the Prelude is very simple:

$
function application

```
f $ x = f x
```

However, this function is given very low fixity, which means that it can be used to replace parentheses. For instance, we might write a function:

```
foo x y = bar y (baz (fluff (ork x)))
```

However, using the function application function, we can rewrite this as:

```
foo x y = bar y $ baz $ fluff $ ork x
```

This moderately resembles the function composition syntax. The `($)` function is also useful when combined with other infix functions. For instance, we cannot write:

```
Prelude> putStrLn "5+3=" ++ show (5+3)
```

because this is interpreted as `(putStrLn "5+3=") ++ (show (5+3))`, which makes no sense. However, we can fix this by writing instead:

```
Prelude> putStrLn $ "5+3=" ++ show (5+3)
```

Which works fine.

Consider now the task of extracting from a list of tuples all the ones whose first component is greater than zero. One way to write this would be:

```
fstGt0 l = filter (\ (a,b) -> a>0) l
```

We can first apply eta reduction to the whole function, yielding:

```
fstGt0 = filter (\ (a,b) -> a>0)
```

Now, we can rewrite the lambda function to use the `fst` function instead of the pattern matching:

```
fstGt0 = filter (\x -> fst x > 0)
```

Now, we can use function composition between `fst` and `>` to get:

```
fstGt0 = filter (\x -> ((>0) . fst) x)
```

And finally we can eta reduce:

```
fstGt0 = filter ((>0).fst)
```

This definition is simultaneously shorter and easier to understand than the original. We can clearly see exactly what it is doing: we're filtering a list by checking whether something is greater than zero. What are we checking? The `fst` element.

While converting to point free style often results in clearer code, this is of course not always the case. For instance, converting the following map to point free style yields something nearly uninterpretable:

```
foo = map (\x -> sqrt (3+4*(x^2)))
foo = map (sqrt . (3+) . (4*) . (^2))
```

There are a handful of combinators defined in the Prelude which are useful for point free programming:

- `uncurry` takes a function of type $a \to b \to c$ and converts it into a function of type $(a, b) \to c$. This is useful, for example, when mapping across a list of pairs:

  ```
  Prelude> map (uncurry (*)) [(1,2),(3,4),(5,6)]
  [2,12,30]
  ```

- `curry` is the opposite of `uncurry` and takes a function of type $(a, b) \to c$ and produces a function of type $a \to b \to c$.

- `flip` reverse the order of arguments to a function. That is, it takes a function of type $a \to b \to c$ and produces a function of type $b \to a \to c$. For instance, we can sort a list in reverse order by using `flip compare`:

```
Prelude> List.sortBy compare [5,1,8,3]
[1,3,5,8]
Prelude> List.sortBy (flip compare) [5,1,8,3]
[8,5,3,1]
```

This is the same as saying:

```
Prelude> List.sortBy (\a b -> compare b a) [5,1,8,3]
[8,5,3,1]
```

only shorter.

Of course, not all functions can be written in point free style. For instance:

```
square x = x*x
```

Cannot be written in point free style, without some other combinators. For instance, if we can define other functions, we can write:

```
pair x = (x,x)
square = uncurry (*) . pair
```

But in this case, this is not terribly useful.

Exercises

Exercise 7.1 *Convert the following functions into point-free style, if possible.*

```
func1 x l = map (\y -> y*x) l

func2 f g l = filter f (map g l)

func3 f l = l ++ map f l

func4 l = map (\y -> y+2)
              (filter (\z -> z ‘elem‘ [1..10])
                      (5:l))

func5 f l = foldr (\x y -> f (y,x)) 0 l
```

7.4 Pattern Matching

Pattern matching is one of the most powerful features of Haskell (and most functional programming languages). It is most commonly used in conjunction with **case** expressions, which we have already seen in Section 3.5. Let's return to our Color example from Section 4.5. I'll repeat the definition we already had for the datatype:

```
data Color
    = Red
    | Orange
    | Yellow
    | Green
    | Blue
    | Purple
    | White
    | Black
    | Custom Int Int Int  -- R G B components
    deriving (Show,Eq)
```

We then want to write a function that will convert between something of type Color and a triple of Ints, which correspond to the RGB values, respectively. Specifically, if we see a Color which is `Red`, we want to return `(255,0,0)`, since this is the RGB value for red. So we write that (remember that piecewise function definitions are just **case** statements):

```
colorToRGB Red = (255,0,0)
```

If we see a Color which is `Orange`, we want to return `(255,128,0)`; and if we see `Yellow`, we want to return `(255,255,0)`, and so on. Finally, if we see a custom color, which is comprised of three components, we want to make a triple out of these, so we write:

```
colorToRGB Orange = (255,128,0)
colorToRGB Yellow = (255,255,0)
colorToRGB Green  = (0,255,0)
colorToRGB Blue   = (0,0,255)
colorToRGB Purple = (255,0,255)
colorToRGB White  = (255,255,255)
colorToRGB Black  = (0,0,0)
colorToRGB (Custom r g b) = (r,g,b)
```

Then, in our interpreter, if we type:

```
Color> colorToRGB Yellow
(255,255,0)
```

What is happening is this: we create a value, call it x, which has value `Red`. We then apply this to `colorToRGB`. We check to see if we can "match" x against `Red`. This match fails because according to the definition of **Eq** Color, `Red` is not equal to `Yellow`. We continue down the definitions of `colorToRGB` and try to match `Yellow` against `Orange`. This fails, too. We the try to match `Yellow` against `Yellow`, which succeeds, so we use this function definition, which simply returns the value `(255,255,0)`, as expected.

Suppose instead, we used a custom color:

```
Color> colorToRGB (Custom 50 200 100)
(50,200,100)
```

We apply the same matching process, failing on all values from `Red` to `Black`. We then get to try to match `Custom 50 200 100` against `Custom r g b`. We can see that the `Custom` part matches, so then we go see if the subelements match. In the matching, the variables `r`, `g` and `b` are essentially wild cards, so there is no trouble matching `r` with 50, `g` with 200 and `b` with 100. As a "side-effect" of this matching, `r` gets the value 50, `g` gets the value 200 and `b` gets the value 100. So the entire match succeeded and we look at the definition of this part of the function and bundle up the triple using the matched values of `r`, `g` and `b`.

We can also write a function to check to see if a `Color` is a custom color or not:

```
isCustomColor (Custom _ _ _) = True
isCustomColor _ = False
```

When we apply a value to `isCustomColor` it tries to match that value against `Custom _ _ _`. This match will succeed if the value is `Custom x y z` for any `x`, `y` and `z`. The _ (underscore) character is a "wildcard" and will match anything, but will not do the binding that would happen if you put a variable name there. If this match succeeds, the function returns `True`; however, if this match fails, it goes on to the next line, which will match anything and then return `False`.

For some reason we might want to define a function which tells us whether a given color is "bright" or not, where my definition of "bright" is that one of its RGB components is equal to 255 (admittedly and arbitrary definition, but it's simply an example). We could define this function as:

```
isBright = isBright' . colorToRGB
    where isBright' (255,_,_) = True
          isBright' (_,255,_) = True
          isBright' (_,_,255) = True
          isBright' _         = False
```

Let's dwell on this definition for a second. The `isBright` function is the composition of our previously defined function `colorToRGB` and a helper function `isBright'`, which tells us if a given RGB value is bright or not. We could replace the first line here with `isBright c = isBright' (colorToRGB c)` but there is no need to explicitly write the parameter here, so we don't. Again, this function composition style of programming takes some getting used to, so I will try to use it frequently in this tutorial.

The `isBright'` helper function takes the RGB triple produced by `colorToRGB`. It first tries to match it against `(255,_,_)` which succeeds if the value has 255 in its first position. If this match succeeds, `isBright'` returns `True` and so does `isBright`. The second and third line of definition check for 255 in the second and third position in the triple, respectively. The fourth line, the *fallthrough*, matches everything else and reports it as not bright.

We might want to also write a function to convert between RGB triples and Colors. We could simple stick everything in a `Custom` constructor, but this would defeat the purpose; we want to use the `Custom` slot only for values which don't match the predefined colors. However, we don't want to allow the user to construct custom colors like (600,-40,99) since these are invalid RGB values. We could throw an error if such a value is given, but this can be difficult to deal with. Instead, we use the `Maybe` datatype. This is defined (in the Prelude) as:

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

The way we use this is as follows: our `rgbToColor` function returns a value of type `Maybe Color`. If the RGB value passed to our function is *invalid*, we return `Nothing`, which corresponds to a failure. If, on the other hand, the RGB value is valid, we create the appropriate Color value and return `Just` that. The code to do this is:

```
rgbToColor 255   0   0 = Just Red
rgbToColor 255 128   0 = Just Orange
rgbToColor 255 255   0 = Just Yellow
rgbToColor   0 255   0 = Just Green
rgbToColor   0   0 255 = Just Blue
rgbToColor 255   0 255 = Just Purple
rgbToColor 255 255 255 = Just White
rgbToColor   0   0   0 = Just Black
rgbToColor   r   g   b =
    if 0 <= r && r <= 255 &&
       0 <= g && g <= 255 &&
       0 <= b && b <= 255
      then Just (Custom r g b)
      else Nothing   -- invalid RGB value
```

The first eight lines match the RGB arguments against the predefined values and, if they match, `rgbToColor` returns `Just` the appropriate color. If none of these matches, the last definition of `rgbToColor` matches the first argument against `r`, the second against g and the third against b (which causes the side-effect of binding these values). It then checks to see if these values are valid (each is greater than or equal to zero and less than or equal to 255). If so, it returns Just (Custom r g b); if not, it returns Nothing corresponding to an invalid color.

Using this, we can write a function that checks to see if a right RGB value is valid:

```
rgbIsValid r g b = rgbIsValid’ (rgbToColor r g b)
    where rgbIsValid’ (Just _) = True
          rgbIsValid’ _        = False
```

Here, we compose the helper function rgbIsValid’ with our function rgbToColor. The helper function checks to see if the value returned by rgbToColor is Just anything (the wildcard). If so, it returns True. If not, it matches anything and returns False.

Pattern matching isn’t magic, though. You can only match against datatypes; you cannot match against functions. For instance, the following is invalid:

```
f x = x + 1

g (f x) = x
```

Even though the intended meaning of g is clear (i.e., g x = x - 1), the compiler doesn’t know in general that f has an inverse function, so it can’t perform matches like this.

7.5 Guards

Guards can be thought of as an extension to the pattern matching facility. They enable you to allow piecewise function definitions to be taken according to arbitrary boolean expressions. Guards appear after all arguments to a function but before the equals sign, and are begun with a vertical bar. We could use guards to write a simple function which returns a string telling you the result of comparing two elements:

```
comparison x y | x < y = "The first is less"
               | x > y = "The second is less"
               | otherwise = "They are equal"
```

You can read the vertical bar as “such that.” So we say that the value of comparison x y “such that” x is less than y is “The first is less.” The value such that x is greater than y is “The second is less” and the value **otherwise** is “They are equal”. The keyword otherwise is simply defined to be equal to True and thus matches anything that falls through that far. So, we can see that this works:

```
Guards> comparison 5 10
"The first is less"
Guards> comparison 10 5
"The second is less"
Guards> comparison 7 7
"They are equal"
```

Guards are applied in conjunction with pattern matching. When a pattern matches, all of its guards are tried, consecutively, until one matches. If none match, then pattern matching continues with the next pattern.

One nicety about guards is that **where** clauses are common to all guards. So another possible definition for our `isBright` function from the previous section would be:

```
isBright2 c | r == 255 = True
            | g == 255 = True
            | b == 255 = True
            | otherwise = False
    where (r,g,b) = colorToRGB c
```

The function is equivalent to the previous version, but performs its calculation slightly differently. It takes a color, `c`, and applies `colorToRGB` to it, yielding an RGB triple which is matched (using pattern matching!) against `(r,g,b)`. This match succeeds and the values `r`, `g` and `b` are bound to their respective values. The first guard checks to see if `r` is 255 and, if so, returns true. The second and third guard check `g` and `b` against 255, respectively and return true if they match. The last guard fires as a last resort and returns `False`.

7.6 Instance Declarations

In order to declare a type to be an instance of a class, you need to provide an instance declaration for it. Most classes provide what's called a "minimal complete definition." This means the functions which must be implemented for this class in order for its definition to be satisfied. Once you've written these functions for your type, you can declare it an instance of the class.

7.6.1 The `Eq` Class

The **Eq** class has two members (i.e., two functions):

```
(==) :: Eq a => a -> a -> Bool
(/=) :: Eq a => a -> a -> Bool
```

The first of these type signatures reads that the function `==` is a function which takes two as which are members of **Eq** and produces a `Bool`. The type signature of `/=` (not equal) is identical. A minimal complete definition for the **Eq** class requires that either one of these functions be defined (if you define `==`, then `/=` is defined automatically by negating the result of `==`, and vice versa). These declarations must be provided inside the instance declaration.

This is best demonstrated by example. Suppose we have our color example, repeded here for convenience:

```
data Color
    = Red
    | Orange
    | Yellow
    | Green
    | Blue
    | Purple
    | White
    | Black
    | Custom Int Int Int  -- R G B components
```

We can define Color to be an instance of **Eq** by the following declaration:

```
instance Eq Color where
    Red == Red = True
    Orange == Orange = True
    Yellow == Yellow = True
    Green == Green = True
    Blue == Blue = True
    Purple == Purple = True
    White == White = True
    Black == Black = True
    (Custom r g b) == (Custom r' g' b') =
        r == r' && g == g' && b == b'
    _ == _ = False
```

The first line here begins with the keyword **instance** telling the compiler that we're making an instance declaration. It then specifies the class, **Eq**, and the type, Color which is going to be an instance of this class. Following that, there's the **where** keyword. Finally there's the method declaration.

The first eight lines of the method declaration are basically identical. The first one, for instance, says that the value of the expression `Red == Red` is equal to `True`. Lines two through eight are identical. The declaration for custom colors is a bit different. We pattern match `Custom` on both sides of `==`. On the left hand side, we bind `r`, `g` and `b` to the components, respectively. On the right hand side, we bind `r'`, `g'` and `b'` to the components. We then say that these two custom colors are equal precisely when `r == r'`, `g == g'` and `b == b'` are all equal. The fallthrough says that any pair we haven't previously declared as equal are unequal.

7.6.2 The `Show` Class

The **Show** class is used to display arbitrary values as strings. This class has three methods:

```
show :: Show a => a -> String
showsPrec :: Show a => Int -> a -> String -> String
showList :: Show a => [a] -> String -> String
```

A minimal complete definition is either `show` or `showsPrec` (we will talk about `showsPrec` later – it's in there for efficiency reasons). We can define our Color datatype to be an instance of **Show** with the following instance declaration:

```
instance Show Color where
    show Red = "Red"
    show Orange = "Orange"
    show Yellow = "Yellow"
    show Green = "Green"
    show Blue = "Blue"
    show Purple = "Purple"
    show White = "White"
    show Black = "Black"
    show (Custom r g b) =
        "Custom " ++ show r ++ " " ++
        show g ++ " " ++ show b
```

This declaration specifies exactly how to convert values of type Color to Strings. Again, the first eight lines are identical and simply take a Color and produce a string. The last line for handling custom colors matches out the RGB components and creates a string by concattenating the result of `show`ing the components individually (with spaces in between and "Custom" at the beginning).

7.6.3 Other Important Classes

There are a few other important classes which I will mention briefly because either they are commonly used or because we will be using them shortly. I won't provide example instance declarations; how you can do this should be clear by now.

The `Ord` Class

The ordering class, the functions are:

```
compare :: Ord a => a -> a -> Ordering
(<=) :: Ord a => a -> a -> Bool
(>) :: Ord a => a -> a -> Bool
(>=) :: Ord a => a -> a -> Bool
(<) :: Ord a => a -> a -> Bool
min :: Ord a => a -> a -> a
max :: Ord a => a -> a -> a
```

The almost any of the functions alone is a minimal complete definition; it is recommended that you implement `compare` if you implement only one, though. This function returns a value of type Ordering which is defined as:

```
data Ordering = LT | EQ | GT
```

So, for instance, we get:

```
Prelude> compare 5 7
LT
Prelude> compare 6 6
EQ
Prelude> compare 7 5
GT
```

In order to declare a type to be an instance of **Ord** you must already have declared it an instance of **Eq** (in other words, **Ord** is a *subclass* of **Eq** – more about this in Section 8.4).

The `Enum` Class

The **Enum** class is for enumerated types; that is, for types where each element has a successor and a predecessor. It's methods are:

```
pred :: Enum a => a -> a
succ :: Enum a => a -> a
toEnum :: Enum a => Int -> a
fromEnum :: Enum a => a -> Int
enumFrom :: Enum a => a -> [a]
enumFromThen :: Enum a => a -> a -> [a]
enumFromTo :: Enum a => a -> a -> [a]
enumFromThenTo :: Enum a => a -> a -> a -> [a]
```

The minimal complete definition contains both `toEnum` and `fromEnum`, which converts from and to Ints. The `pred` and `succ` functions give the predecessor and successor, respectively. The `enum` functions enumerate lists of elements. For instance, `enumFrom x` lists all elements after `x`; `enumFromThen x step` lists all elements starting at `x` in steps of size `step`. The `To` functions end the enumeration at the given element.

The `Num` Class

The **Num** class provides the standard arithmetic operations:

```
(-) :: Num a => a -> a -> a
(*) :: Num a => a -> a -> a
(+) :: Num a => a -> a -> a
negate :: Num a => a -> a
signum :: Num a => a -> a
abs :: Num a => a -> a
fromInteger :: Num a => Integer -> a
```

All of these are obvious except for perhaps `negate` which is the unary minus. That is, `negate x` means $-x$.

The `Read` Class

The **Read** class is the opposite of the **Show** class. It is a way to take a string and read in from it a value of arbitrary type. The methods for **Read** are:

```
readsPrec :: Read a => Int -> String -> [(a, String)]
readList :: String -> [([a], String)]
```

The minimal complete definition is `readsPrec`. The most important function related to this is `read`, which uses `readsPrec` as:

```
read s = fst (head (readsPrec 0 s))
```

This will fail if parsing the string fails. You could define a `maybeRead` function as:

```
maybeRead s =
    case readsPrec 0 s of
      [(a,_)] -> Just a
      _ -> Nothing
```

How to write and use `readsPrec` directly will be discussed further in the examples.

7.6.4 Class Contexts

Suppose we are definition the Maybe datatype from scratch. The definition would be something like:

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

Now, when we go to write the instance declarations, for, say, **Eq**, we need to know that a is an instance of **Eq** otherwise we can't write a declaration. We express this as:

```
instance Eq a => Eq (Maybe a) where
    Nothing == Nothing = True
    (Just x) == (Just x') = x == x'
```

This first line can be read "That a is an instance of **Eq** *implies* (=>) that Maybe a is an instance of **Eq**."

7.6.5 Deriving Classes

Writing obvious **Eq**, **Ord**, **Read** and **Show** classes like these is tedious and should be automated. Luckily for us, it is. If you write a datatype that's "simple enough" (almost any datatype you'll write unless you start writing fixed point types), the compiler can automatically *derive* some of the most basic classes. To do this, you simply add a **deriving** clause to after the datatype declaration, as in:

```
data Color
    = Red
    | ...
    | Custom Int Int Int  -- R G B components
    deriving (Eq, Ord, Show, Read)
```

This will automatically create instances of the Color datatype of the named classes. Similarly, the declaration:

```
data Maybe a = Nothing
             | Just a
             deriving (Eq, Ord, Show, Read)
```

derives these classes just when a is appropriate.

All in all, you are allowed to derive instances of **Eq**, **Ord**, **Enum**, **Bounded**, **Show** and **Read**. There is considerable work in the area of "polytypic programming" or "generic programming" which, among other things, would allow for instance declarations for *any* class to be derived. This is much beyond the scope of this tutorial; instead, I refer you to the literature.

7.7 Datatypes Revisited

I know by this point you're probably terribly tired of hearing about datatypes. They are, however, incredibly important, otherwise I wouldn't devote so much time to them. Datatypes offer a sort of notational convenience if you have, for instance, a datatype that holds many many values. These are called named fields.

7.7.1 Named Fields

Consider a datatype whose purpose is to hold configuration settings. Usually when you extract members from this type, you really only care about one or possibly two of the many settings. Moreover, if many of the settings have the same type, you might often find yourself wondering "wait, was this the fourth or *fifth* element?" One thing you could do would be to write accessor functions. Consider the following made-up configuration type for a terminal program:

```
data Configuration =
    Configuration String          -- user name
                  String          -- local host
                  String          -- remote host
                  Bool            -- is guest?
                  Bool            -- is super user?
                  String          -- current directory
                  String          -- home directory
                  Integer         -- time connected
              deriving (Eq, Show)
```

You could then write accessor functions, like (I've only listed a few):

```
getUserName (Configuration un _ _ _ _ _ _ _) = un
getLocalHost (Configuration _ lh _ _ _ _ _ _) = lh
getRemoteHost (Configuration _ _ rh _ _ _ _ _) = rh
getIsGuest (Configuration _ _ _ ig _ _ _ _) = ig
...
```

You could also write update functions to update a single element. Of course, now if you add an element to the configuration, or remove one, all of these functions now have to take a different number of arguments. This is highly annoying and is an easy place for bugs to slip in. However, there's a solution. We simply give names to the fields in the datatype declaration, as follows:

```
data Configuration =
    Configuration { username      :: String,
                    localhost     :: String,
                    remotehost    :: String,
                    isguest       :: Bool,
                    issuperuser   :: Bool,
                    currentdir    :: String,
                    homedir       :: String,
                    timeconnected :: Integer
                  }
```

This will automatically generate the following accessor functions for us:

```
username :: Configuration -> String
localhost :: Configuration -> String
...
```

Moreover, it gives us very convenient update methods. Here is a short example for a "post working directory" and "change directory" like functions that work on Configurations:

```
changeDir :: Configuration -> String -> Configuration
changeDir cfg newDir =
    -- make sure the directory exists
    if directoryExists newDir
      then -- change our current directory
           cfg{currentdir = newDir}
      else error "directory does not exist"

postWorkingDir :: Configuration -> String
  -- retrieve our current directory
postWorkingDir cfg = currentdir cfg
```

So, in general, to update the field `x` in a datatype y to `z`, you write `y{x=z}`. You can change more than one; each should be separated by commas, for instance, `y{x=z, a=b, c=d}`.

You can of course continue to pattern match against Configurations as you did before. The named fields are simply syntactic sugar; you can still write something like:

```
getUserName (Configuration un _ _ _ _ _ _ _) = un
```

But there is little reason to. Finally, you can pattern match against named fields as in:

```
getHostData (Configuration {localhost=lh,remotehost=rh})
  = (lh,rh)
```

This matches the variable `lh` against the `localhost` field on the Configuration and the variable `rh` against the `remotehost` field on the Configuration. These matches of course succeed. You could also constrain the matches by putting values instead of variable names in these positions, as you would for standard datatypes.

You can create values of Configuration in the old way as shown in the first definition below, or in the named-field's type, as shown in the second definition below:

```
initCFG =
    Configuration "nobody" "nowhere" "nowhere"
                  False False "/" "/" 0
initCFG' =
    Configuration
       { username="nobody",
         localhost="nowhere",
         remotehost="nowhere",
         isguest=False,
         issuperuser=False,
         currentdir="/",
         homedir="/",
         timeconnected=0 }
```

Though the second is probably much more understandable unless you litter your code with comments.

7.8 More Lists

todo: put something here

7.8.1 Standard List Functions

Recall that the definition of the built-in Haskell list datatype is equivalent to:

```
data List a = Nil
            | Cons a (List a)
```

With the exception that `Nil` is called `[]` and `Cons x xs` is called `x:xs`. This is simply to make pattern matching easier and code smaller. Let's investigate how some of the standard list functions may be written. Consider `map`. A definition is given below:

```
map _ [] = []
map f (x:xs) = f x : map f xs
```

Here, the first line says that when you `map` across an empty list, no matter what the function is, you get an empty list back. The second line says that when you `map` across a list with `x` as the head and `xs` as the tail, the result is `f` applied to `x` consed onto the result of mapping `f` on `xs`.

The `filter` can be defined similarly:

```
filter _ [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

How this works should be clear. For an empty list, we return an empty list. For a non empty list, we return the filter of the tail, perhaps with the head on the front, depending on whether it satisfies the predicate `p` or not.

We can define `foldr` as:

```
foldr _ z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Here, the best interpretation is that we are replacing the empty list (`[]`) with a particular value and the list constructor (`:`) with some function. On the first line, we can see the replacement of `[]` for `z`. Using backquotes to make `f` infix, we can write the second line as:

```
foldr f z (x:xs) = x ‘f‘ (foldr f z xs)
```

From this, we can directly see how `:` is being replaced by `f`.

Finally, `foldl`:

```
foldl _ z [] =  z
foldl f z (x:xs) = foldl f (f z x) xs
```

This is slightly more complicated. Remember, `z` can be thought of as the current state. So if we're folding across a list which is empty, we simply return the current state. On the other hand, if the list is not empty, it's of the form `x:xs`. In this case, we get a new state by appling `f` to the current state `z` and the current list element `x` and then recursively call `foldl` on `xs` with this new state.

There is another class of functions: the `zip` and `unzip` functions, which respectively take multiple lists and make one or take one lists and split them apart. For instance, `zip` does the following:

```
Prelude> zip "hello" [1,2,3,4,5]
[(’h’,1),(’e’,2),(’l’,3),(’l’,4),(’o’,5)]
```

Basically, it pairs the first elements of both lists and makes that the first element of the new list. It then pairs the second elements of both lists and makes that the second element, etc. What if the lists have unequal length? It simply stops when the shorter one stops. A reasonable definition for `zip` is:

```
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

The `unzip` function does the opposite. It takes a zipped list and returns the two “original” lists:

```
Prelude> unzip [('f',1),('o',2),('o',3)]
("foo",[1,2,3])
```

There are a whole slew of `zip` and `unzip` functions, named `zip3`, `unzip3`, `zip4`, `unzip4` and so on; the `...3` functions use triples instead of pairs; the `...4` functions use 4-tuples, etc.

Finally, the function `take` takes an integer n and a list and returns the first n elements off the list. Correspondingly, `drop` takes an integer n and a list and returns the result of throwing away the first n elements off the list. Neither of these functions produces an error; if n is too large, they both will just return shorter lists.

7.8.2 List Comprehensions

There is some syntactic sugar for dealing with lists whose elements are members of the **Enum** class (see Section 7.6), such as Int or Char. If we want to create a list of all the elements from 1 to 10, we can simply write:

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

We can also introduce an amount to step by:

```
Prelude> [1,3..10]
[1,3,5,7,9]
Prelude> [1,4..10]
[1,4,7,10]
```

These expressions are short hand for `enumFromTo` and `enumFromThenTo`, respectively. Of course, you don’t need to specify an upper bound. Try the following (but be ready to hit Control+C to stop the computation!):

```
Prelude> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12{Interrupted!}
```

Probably yours printed a few thousand more elements than this. As we said before, Haskell is lazy. That means that a list of all numbers from 1 on is perfectly well formed and that's exactly what this list is. Of course, if you attempt to print the list (which we're implicitly doing by typing it in the interpreter), it won't halt. But if we only evaluate an initial segment of this list, we're fine:

```
Prelude> take 3 [1..]
[1,2,3]
Prelude> take 3 (drop 5 [1..])
[6,7,8]
```

This comes in useful if, say, we want to assign an ID to each element in a list. Without laziness we'd have to write something like this:

```
assignID :: [a] -> [(a,Int)]
assignID l = zip l [1..length l]
```

Which means that the list will be traversed twice. However, because of laziness, we can simply write:

```
assignID l = zip l [1..]
```

And we'll get exactly what we want. We can see that this works:

```
Prelude> assignID "hello"
[('h',1),('e',2),('l',3),('l',4),('o',5)]
```

Finally, there is some useful syntactic sugar for `map` and `filter`, based on standard set-notation in mathematics. In math, we would write something like $\{f(x)|x \in s \wedge p(x)\}$ to mean the set of all values of f when applied to elements of s which satisfy p. This is equivalent to the Haskell statement `map f (filter p s)`. However, we can also use more math-like notation and write `[f x | x <- s, p x]`. While in math the ordering of the statements on the side after the pipe is free, it is not so in Haskell. We could not have put `p x` before `x <- s` otherwise the compiler wouldn't know yet what `x` was. We can use this to do simple string processing. Suppose we want to take a string, remove all the lower-case letters and convert the rest of the letters to upper case. We could do this in either of the following two equivalent ways:

```
Prelude> map toLower (filter isUpper "Hello World")
"hw"
Prelude> [toLower x | x <- "Hello World", isUpper x]
"hw"
```

These two are equivalent, and, depending on the exact functions you're using, one might be more readable than the other. There's more you can do here, though. Suppose you want to create a list of pairs, one for each point between (0,0) and (5,7) below the diagonal. Doing this manually with lists and maps would be cumbersome and possibly difficult to read. It couldn't be easier than with list comprehensions:

```
Prelude> [(x,y) | x <- [1..5], y <- [x..7]]
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(2,2),(2,3),
(2,4),(2,5),(2,6),(2,7),(3,3),(3,4),(3,5),(3,6),(3,7),
(4,4),(4,5),(4,6),(4,7),(5,5),(5,6),(5,7)]
```

If you reverse the order of the `x <-` and `y <-` clauses, the order in which the space is traversed will be reversed (of course, in that case, `y` could no longer depend on `x` and you would need to make `x` depend on `y` but this is trivial).

7.9 Arrays

Lists are nice for many things. It is easy to add elements to the beginning of them and to manipulate them in various ways that change the length of the list. However, they are bad for random access, having average complexity $\mathcal{O}(n)$ to access an arbitrary element (if you don't know what $\mathcal{O}(\dots)$ means, you can either ignore it or take a quick detour and read Appendix A, a two-page introduction to complexity theory). So, if you're willing to give up fast insertion and deletion because you need random access, you should use arrays instead of lists.

In order to use arrays you must import the `Array` module. There are a few methods for creating arrays, the `array` function, the `listArray` function, and the `accumArray` function. The `array` function takes a pair which is the bounds of the array, and an association list which specifies the initial values of the array. The `listArray` function takes bounds and then simply a list of values. Finally, the `accumArray` function takes an accumulation function, an initial value and an association list and accumulates pairs from the list into the array. Here are some examples of arrays being created:

```
Arrays> array (1,5) [(i,2*i) | i <- [1..5]]
array (1,5) [(1,2),(2,4),(3,6),(4,8),(5,10)]
Arrays> listArray (1,5) [3,7,5,1,10]
array (1,5) [(1,3),(2,7),(3,5),(4,1),(5,10)]
Arrays> accumArray (+) 2 (1,5) [(i,i) | i <- [1..5]]
array (1,5) [(1,3),(2,4),(3,5),(4,6),(5,7)]
```

When arrays are printed out (via the show function), they are printed with an association list. For instance, in the first example, the association list says that the value of the array at 1 is 2, the value of the array at 2 is 4, and so on.

You can extract an element of an array using the `!` function, which takes an array and an index, as in:

```
Arrays> (listArray (1,5) [3,7,5,1,10]) ! 3
5
```

Moreover, you can update elements in the array using the `//` function. This takes an array and an association list and updates the positions specified in the list:

```
Arrays> (listArray (1,5) [3,7,5,1,10]) //
              [(2,99),(3,-99)]
array (1,5) [(1,3),(2,99),(3,-99),(4,1),(5,10)]
```

There are a few other functions which are of interest:

`bounds`	returns the bounds of an array
`indices`	returns a list of all indices of the array
`elems`	returns a list of all the values in the array in order
`assocs`	returns an association list for the array

If we define `arr` to be `listArray (1,5) [3,7,5,1,10]`, the result of these functions applied to `arr` are:

```
Arrays> bounds arr
(1,5)
Arrays> indices arr
[1,2,3,4,5]
Arrays> elems arr
[3,7,5,1,10]
Arrays> assocs arr
[(1,3),(2,7),(3,5),(4,1),(5,10)]
```

Note that while arrays are $\mathcal{O}(1)$ access, they are not $\mathcal{O}(1)$ update. They are in fact $\mathcal{O}(n)$ update, since in order to maintain purity, the array must be *copied* in order to make an update. Thus, functional arrays are pretty much only useful when you're filling them up once and then only reading. If you need fast access and update, you should probably use `FiniteMap`s, which are discussed in Section 7.10 and have $\mathcal{O}(\log n)$ access and update.

7.10 Finite Maps

The FiniteMap datatype (which is available in the `FiniteMap` module, or `Data.FiniteMap` module in the hierarchical libraries) is a purely functional implementation of balanced trees. Finite maps can be compared to lists and arrays in terms of the time it takes to perform various operations on those datatypes of a fixed size, n. A brief comparison is:

	List	**Array**	**FiniteMap**
insert	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
update	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
delete	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
find	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
map	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$

As we can see, lists provide fast insertion (but slow everything else), arrays provide fast lookup (but slow everything else) and finite maps provide moderately fast everything (except mapping, which is a bit slower than lists or arrays).

The type of a finite map is for the form FiniteMapkeyelt where key is the type of the keys and elt is the type of the elements. That is, finite maps are lookup tables from type key to type elt.

The basic finite map functions are:

```
emptyFM   :: FiniteMap key elt
addToFM   :: FiniteMap key elt -> key -> elt ->
               FiniteMap key elt
delFromFM :: FiniteMap key elt -> key ->
               FiniteMap key elt
elemFM    :: key -> FiniteMap key elt -> Bool
lookupFM  :: FiniteMap key elt -> key -> Maybe elt
```

In all these cases, the type key must be an instance of **Ord** (and hence also an instance of **Eq**).

There are also function `listToFM` and `fmToList` to convert lists to and from finite maps. Try the following:

```
Prelude> :m FiniteMap
FiniteMap> let fm = listToFM
                      [('a',5),('b',10),('c',1),('d',2)]
FiniteMap> let myFM = addToFM fm 'e' 6
FiniteMap> fmToList fm
[('a',5),('b',10),('c',1),('d',2)]
FiniteMap> fmToList myFM
[('a',5),('b',10),('c',1),('d',2),('e',6)]
FiniteMap> lookupFM myFM 'e'
Just 6
FiniteMap> lookupFM fm 'e'
Nothing
```

You can also experiment with the other commands. Note that you cannot `show` a finite map, as they are not instances of **Show**:

```
FiniteMap> show myFM
```

```
<interactive>:1:
    No instance for (Show (FiniteMap Char Integer))
    arising from use of ‘show’ at <interactive>:1
    In the definition of ‘it’: show myFM
```

In order to inspect the elements, you first need to use `fmToList`.

7.11 Layout

7.12 The Final Word on Lists

You are likely tired of hearing about lists at this point, but they are so fundamental to Haskell (and really all of functional programming) that it would be terrible not to talk about them some more.

It turns out that `foldr` is actually quite a powerful function: it can compute an *primitive recursive* function. A primitive recursive function is essentially one which can be calculated using only "for" loops, but not "while" loops.

primitive recursive

In fact, we can fairly easily define `map` in terms of `foldr`:

```
map2 f = foldr (\a b -> f a : b) []
```

Here, `b` is the accumulator (i.e., the result list) and `a` is the element being currently considered. In fact, we can simplify this definition through a sequence of steps:

```
     foldr (\a b -> f a : b) []
==>  foldr (\a b -> (:) (f a) b) []
==>  foldr (\a -> (:) (f a)) []
==>  foldr (\a -> ((:) . f) a) []
==>  foldr ((:) . f) []
```

This is directly related to the fact that `foldr (:)  []` is the identity function on lists. This is because, as mentioned before, `foldr f z` can be thought of as replacing the `[]` in lists by `z` and the `:` by `f`. In this case, we're keeping both the same, so it is the identity function.

In fact, you can convert any function of the following style into a `foldr`:

```
myfunc [] = z
myfunc (x:xs) = f x (myfunc xs)
```

By writing the last line with `f` in infix form, this should be obvious:

```
myfunc [] = z
myfunc (x:xs) = x ‘f‘ (myfunc xs)
```

Clearly, we are just replacing `[]` with `z` and `:` with `f`. Consider the `filter` function:

```
filter p [] = []
filter p (x:xs) =
  if p x
    then x : filter p xs
    else filter p xs
```

This function also follows the form above. Based on the first line, we can figure out that `z` is supposed to be `[]`, just like in the `map` case. Now, suppose that we call the result of calling `filter p xs` simply `b`, then we can rewrite this as:

```
filter p [] = []
filter p (x:xs) =
  if p x then x : b else b
```

Given this, we can transform `filter` into a fold:

```
filter p = foldr (\a b -> if p a then a:b else b) []
```

Let's consider a slightly more complicated function: `++`. The definition for `++` is:

```
(++) []     ys = ys
(++) (x:xs) ys = x : (xs ++ ys)
```

Now, the question is whether we can write this in fold notation. First, we can apply eta reduction to the first line to give:

```
(++) [] = id
```

Through a sequence of steps, we can also eta-reduce the second line:

```
     (++) (x:xs) ys = x : ((++) xs ys)
==>  (++) (x:xs) ys = (x:) ((++) xs ys)
==>  (++) (x:xs) ys = ((x:) . (++) xs) ys
==>  (++) (x:xs) = (x:) . (++) xs
```

Thus, we get that an eta-reduced defintion of `++` is:

```
(++) []     = id
(++) (x:xs) = (x:) . (++) xs
```

Now, we can try to put this into fold notation. First, we notice that the base case converts `[]` into `id`. Now, if we assume `(++) xs` is called `b` and `x` is called `a`, we can get the following definition in terms of `foldr`:

```
(++) = foldr (\a b -> (a:) . b) id
```

This actually makes sense intuitively. If we only think about applying `++` to one argument, we can think of it as a function which takes a list and creates a function which, when applied, will prepend this list to another list. In the lambda function, we assume we have a function `b` which will do this for the rest of the list and we need to create a function which will do this for `b` as well as the single element `a`. In order to do this, we first apply `b` and then further add `a` to the front.

We can further reduce this expression to a point-free style through the following sequence:

```
==>  (++) = foldr (\a b -> (a:) . b) id
==>  (++) = foldr (\a b -> (.) (a:) b) id
==>  (++) = foldr (\a -> (.) (a:)) id
==>  (++) = foldr (\a -> (.) ((:) a)) id
==>  (++) = foldr (\a -> ((.) . (:)) a) id
==>  (++) = foldr ((.) . (:)) id
```

This final version is point free, though not necessarily understandable. Presumbably the original version is clearer.

As a final example, consider `concat`. We can write this as:

```
concat []     = []
concat (x:xs) = x ++ concat xs
```

It should be immediately clear that the `z` element for the fold is `[]` and that the recursive function is `++`, yielding:

```
concat = foldr (++) []
```

Exercises

Exercise 7.2 *The function* `and` *takes a list of booleans and returns* `True` *if and only if all of them are* `True`*. It also returns* `True` *on the empty list. Write this function in terms of* `foldr`*.*

Exercise 7.3 *The function* `concatMap` *behaves such that* `concatMap f` *is the same as* `concat .  map f`*. Write this function in terms of* `foldr`*.*

Chapter 8

Advanced Types

As you've probably ascertained by this point, the type system is integral to Haskell. While this chapter is called "Advanced Types", you will probably find it to be more general than that and it must not be skipped simply because you're not interested in the type system.

8.1 Type Synonyms

Type synonyms exist in Haskell simply for convenience: their removal would not make Haskell any less powerful.

Consider the case when you are constantly dealing with lists of three-dimensional points. For instance, you might have a function with type $[(\mathsf{Double}, \mathsf{Double}, \mathsf{Double})] \rightarrow \mathsf{Double} \rightarrow [(\mathsf{Double}, \mathsf{Double}, \mathsf{Double})]$ Since you are a good software engineer, you want to place type signatures on all your top-level functions. However, typing $[(\mathsf{Double}, \mathsf{Double}, \mathsf{Double})]$ all the time gets very tedious. To get around this, you can define a type synonym:

```
type List3D = [(Double,Double,Double)]
```

Now, the type signature for your functions may be written $\mathsf{List3D} \rightarrow \mathsf{Double} \rightarrow \mathsf{List3D}$.

We should note that type synonyms cannot be self-referential. That is, you cannot have:

```
type BadType = Int -> BadType
```

This is because this is an "infinite type." Since Haskell removes type synonyms very early on, any instance of $\mathsf{BadType}$ will be replaced by $\mathsf{Int} \rightarrow \mathsf{BadType}$, which will result in an infinite loop.

Type synonyms can also be parameterized. For instance, you might want to be able to change the types of the points in the list of 3D points. For this, you could define:

```
type List3D a = [(a,a,a)]
```

Then your references to [(Double, Double, Double)] would become List3D Double.

8.2 Newtypes

Consider the problem in which you need to have a type which is very much like Int, but its ordering is defined differently. Perhaps you wish to order Ints first by even numbers then by odd numbers (that is, all odd numbers are greater than any even number and within the odd/even subsets, ordering is standard).

Unfortunately, you cannot define a new instance of **Ord** for Int because then Haskell won't know which one to use. What you want is to define a type which is *isomorphic* to Int.

■ NOTE ■ "Isomorphic" is a common term in mathematics which basically means "structurally identical." For instance, in graph theory, if you have two graphs which are identical except they have different labels on the nodes, they are isomorphic. In our context, two types are isomorphic if they have the same underlying structure.

One way to do this would be to define a new datatype:

```
data MyInt = MyInt Int
```

We could then write appropriate code for this datatype. The problem (and this is very subtle) is that this type is not truly isomorphic to Int: it has one more value. When we think of the type Int, we usually think that it takes all values of integers, but it really has one more value: _|_ (pronounced "bottom"), which is used to represent erroneous or undefined computations. Thus, MyInt has not only values `MyInt 0`, `MyInt 1` and so on, but also `MyInt` _|_. However, since datatypes can themselves be undefined, it has an additional value: _|_ which differs from `MyInt` _|_ and this makes the types non-isomorphic. (See Section **??** for more information on bottom.)

Disregarding that subtlety, there may be efficiency issues with this representation: now, instead of simply storing an integer, we have to store a pointer to an integer and have to follow that pointer whenever we need the value of a MyInt.

To get around these problems, Haskell has a **newtype** construction. A **newtype** is a cross between a datatype and a type synonym: it has a constructor like a datatype, but it can have only one constructor and this constructor can have only one argument. For instance, we can define:

```
newtype MyInt = MyInt Int
```

But we cannot define any of:

```
newtype Bad1 = Bad1a Int | Bad1b Double
newtype Bad2 = Bad2 Int Double
```

Of course, the fact that we cannot define Bad2 as above is not a big issue: we can simply define the following by pairing the types:

```
newtype Good2 = Good2 (Int,Double)
```

Now, suppose we've defined MyInt as a **newtype**. This enables use to write our desired instance of **Ord** as:

```
instance Ord MyInt where
  MyInt i < MyInt j
      | odd i  && odd  j = i < j
      | even i && even j = i < j
      | even i           = True
      | otherwise        = False
      where odd x = (x ‘mod‘ 2) == 0
            even  = not . odd
```

Like datatype, we can still derive classes like **Show** and **Eq** over newtypes (in fact, I'm implicitly assuming we have derived **Eq** over MyInt – where is my assumption in the above code?).

Moreover, in recent versions of GHC (see Section 2.2), on newtypes, you are allowed to derive *any* class of which the base type (in this case, Int) is an instance. For example, we could derive **Num** on MyInt to provide arithmetic functions over it.

Pattern matching over newtypes is exactly as in datatypes. We can write constructor and destructor functions for MyInt as follows:

```
mkMyInt i = MyInt i
unMyInt (MyInt i) = i
```

8.3 Datatypes

We've already seen datatypes used in a variety of contexts. This section concludes some of the discussion and introduces some of the common datatypes in Haskell. It also provides a more theoretical underpinning to what datatypes actually are.

8.3.1 Strict Fields

One of the great things about Haskell is that computation is performed lazily. However, sometimes this leads to inefficiencies. One way around this problem is to use datatypes with strict fields. Before we talk about the solution, let's spend some time to get a bit more comfortable with how bottom works in to the picture (for more theory, see Section **??**).

Suppose we've defined the unit datatype (this one of the simplest datatypes you can define):

```
data Unit = Unit
```

This datatype has exactly one constructor, `Unit`, which takes no arguments. In a strict language like ML, there would be exactly one value of type Unit: namely, `Unit`. This is not quite so in Haskell. In fact, there are *two* values of type Unit. One of them is `Unit`. The other is bottom (written $\perp$).

You can think of bottom as representing a computation which won't halt. For instance, suppose we define the value:

```
foo = foo
```

This is perfectly valid Haskell code and simply says that when you want to evaluate `foo`, all you need to do is evaluate `foo`. Clearly this is an "infinite loop."

What is the type of `foo`? Simply a. We cannot say anything more about it than that. The fact that `foo` has type a in fact tells us that it must be an infinite loop (or some other such strange value). However, since `foo` has type a and thus can have any type, it can also have type Unit. We could write, for instance:

```
foo :: Unit
foo = foo
```

Thus, we have found a second value with type Unit. In fact, we have found all values of type Unit. Any other non-terminating function or error-producing function will have exactly the same effect as `foo` (though Haskell provides some more utility with the function `error`).

This means, for instance, that there are actually *four* values with type Maybe Unit. They are: $\perp$, `Nothing`, `Just` $\perp$ and `Just Unit`. However, it could be the fact that you, as a programmer, know that you will never come across the third of these. Namely, you want the argument to `Just` to be *strict*. This means that if the argument to `Just` is bottom, then the entire structure becomes bottom. You use an exclamation point to specify a constructor as strict. We can define a strict version of Maybe as:

```
data SMaybe a = SNothing | SJust !a
```

There are now only three values of SMaybe. We can see the difference by writing the following program:

```
module Main where

import System

data SMaybe a = SNothing | SJust !a  deriving Show
```

```
main = do
  [cmd] <- getArgs
  case cmd of
    "a" -> printJust   undefined
    "b" -> printJust   Nothing
    "c" -> printJust  (Just undefined)
    "d" -> printJust  (Just ())

    "e" -> printSJust  undefined
    "f" -> printSJust  SNothing
    "g" -> printSJust (SJust undefined)
    "h" -> printSJust (SJust ())

printJust :: Maybe () -> IO ()
printJust Nothing = putStrLn "Nothing"
printJust (Just x) = do putStr "Just "; print x

printJust :: SMaybe () -> IO ()
printSJust SNothing = putStrLn "Nothing"
printSJust (SJust x) = do putStr "Just "; print x
```

Here, depending on what command line argument is passed, we will do something different. The outputs for the various options are:

```
\% ./strict a
Fail: Prelude.undefined

\% ./strict b
Nothing

\% ./strict c
Just
Fail: Prelude.undefined

\% ./strict d
Just ()

\% ./strict e
Fail: Prelude.undefined

\% ./strict f
Nothing

\% ./strict g
Fail: Prelude.undefined
```

```
\% ./strict h
Just ()
```

The thing worth noting here is the difference between cases "c" and "g". In the "c" case, the `Just` is printed, because this is printed *before* the undefined value is evaluated. However, in the "g" case, since the constructor is strict, as soon as you match the `SJust`, you also match the value. In this case, the value is undefined, so the whole thing fails before it gets a chance to do *anything*.

8.4 Classes

We have already encountered type classes a few times, but only in the context of previously existing type classes. This section is about how to define your own. We will begin the discussion by talking about Pong and then move on to a useful generalization of computations.

8.4.1 Pong

The discussion here will be motivated by the construction of the game Pong (see Appendix **??** for the full code). In Pong, there are three things drawn on the screen: the two paddles and the ball. While the paddles and the ball are different in a few respects, they share many commonalities, such as position, velocity, acceleration, color, shape, and so on. We can express these commonalities by defining a class for Pong entities, which we call **Entity**. We make such a definition as follows:

```
class Entity a where
    getPosition :: a -> (Int,Int)
    getVelocity :: a -> (Int,Int)
    getAcceleration :: a -> (Int,Int)
    getColor :: a -> Color
    getShape :: a -> Shape
```

This code defines a typeclass **Entity**. This class has five methods: `getPosition`, `getVelocity`, `getAcceleration`, `getColor` and `getShape` with the corresponding types.

The first line here uses the keyword **class** to introduce a new typeclass. We can read this typeclass definition as "There is a typeclass 'Entity'; a type 'a' is an instance of Entity if it provides the following five functions: . . . ". To see how we can write an instance of this class, let us define a player (paddle) datatype:

```
data Paddle =
   Paddle { paddlePosX, paddlePosY,
            paddleVelX, paddleVelY,
            paddleAccX, paddleAccY :: Int,
            paddleColor :: Color,
            paddleHeight :: Int,
            playerNumber :: Int }
```

Given this data declaration, we can define Paddle to be an instance of **Entity**:

```
instance Entity Paddle where
  getPosition p = (paddlePosX p, paddlePosY p)
  getVelocity p = (paddleVelX p, paddleVelY p)
  getAcceleration p = (paddleAccX p, paddleAccY p)
  getColor = paddleColor
  getShape = Rectangle 5 . paddleHeight
```

The actual Haskell types of the class functions all have included the context `Entity a =>`. For example, `getPosition` has type $\mathbf{Entity}\ \mathsf{a} \Rightarrow \mathsf{a} \rightarrow (\mathsf{Int}, \mathsf{Int})$. However, it will turn out that many of our routines will need entities to also be instances of **Eq**. We can therefore choose to make **Entity** a subclass of **Eq**: namely, you can only be an instance of **Entity** if you are already an instance of **Eq**. To do this, we change the first line of the class declaration to:

```
class Eq a => Entity a where
```

Now, in order to define Paddles to be instances of **Entity** we will first need them to be instances of **Eq** – we can do this by deriving the class.

8.4.2 Computations

Let's think back to our original motivation for defining the Maybe datatype from Section **??**. We wanted to be able to express that functions (i.e., computations) can fail.

Let us consider the case of performing search on a graph. Allow us to take a small aside to set up a small graph library:

```
data Graph v e = Graph [(Int,v)] [(Int,Int,e)]
```

The Graph datatype takes two type arguments which correspond to vertex and edge labels. The first argument to the Graph constructor is a list (set) of vertices; the second is the list (set) of edges. We will assume these lists are always sorted and that each vertex has a unique id and that there is at most one edge between any two vertices.

Suppose we want to search for a path between two vertices. Perhaps there is no path between those vertices. To represent this, we will use the Maybe datatype. If it succeeds, it will return the list of vertices traversed. Our search function could be written (naively) as follows:

```
search :: Graph v e -> Int -> Int -> Maybe [Int]
search g@(Graph vl el) src dst
    | src == dst = Just [src]
    | otherwise  = search' el
    where search' [] = Nothing
          search' ((u,v,_):es)
              | src == u  =
                case search g v dst of
                  Just p  -> Just (u:p)
                  Nothing -> search' es
              | otherwise = search' es
```

This algorithm works as follows (try to read along): to search in a graph `g` from `src` to `dst`, first we check to see if these are equal. If they are, we have found our way and just return the trivial solution. Otherwise, we want to traverse the edge-list. If we're traversing the edge-list and it is empty, we've failed, so we return `Nothing`. Otherwise, we're looking at an edge from `u` to `v`. If `u` is our source, then we consider this step and recursively search the graph from `v` to `dst`. If this fails, we try the rest of the edges; if this succeeds, we put our current position before the path found and return. If `u` is not our source, this edge is useless and we continue traversing the edge-list.

This algorithm is terrible: namely, if the graph contains cycles, it can loop indefinitely. Nevertheless, it is sufficent for now. Be sure you understand it well: things only get more complicated.

Now, there are cases where the Maybe datatype is not sufficient: perhaps we wish to include an error message together with the failure. We could define a datatype to express this as:

```
data Failable a = Success a | Fail String
```

Now, failures come with a failure string to express what went wrong. We can rewrite our search function to use this datatype:

```
search2 :: Graph v e -> Int -> Int -> Failable [Int]
search2 g@(Graph vl el) src dst
    | src == dst = Success [src]
    | otherwise  = search' el
    where search' [] = Fail "No path"
          search' ((u,v,_):es)
              | src == u  =
                case search2 g v dst of
                  Success p -> Success (u:p)
                  _         -> search' es
              | otherwise = search' es
```

This code is a straightforward translation of the above.

There is another option for this computation: perhaps we want not just one path, but all possible paths. We can express this as a function which returns a list of lists of vertices. The basic idea is the same:

```
search3 :: Graph v e -> Int -> Int -> [[Int]]
search3 g@(Graph vl el) src dst
    | src == dst = [[src]]
    | otherwise  = search' el
    where search' [] = []
          search' ((u,v,_):es)
              | src == u  =
                   map (u:) (search3 g v dst) ++
                   search' es
              | otherwise = search' es
```

The code here has gotten a little shorter, thanks to the standard prelude `map` function, though it is essentially the same.

We may ask ourselves what all of these have in common and try to gobble up those commonalities in a class. In essense, we need some way of representing success and some way of representing failure. Furthermore, we need a way to combine two successes (in the first two cases, the first success is chosen; in the third, they are strung together). Finally, we need to be able to augment a previous success (if there was one) with some new value. We can fit this all into a class as follows:

```
class Computation c where
    success :: a -> c a
    failure :: String -> c a
    augment :: c a -> (a -> c b) -> c b
    combine :: c a -> c a -> c a
```

In this class declaration, we're saying that c is an instance of the class **Computation** if it provides four functions: `success`, `failure`, `augment` and `combine`. The `success` function takes a value of type a and returns it wrapped up in c, representing a successful computation. The `failure` function takes a String and returns a computation representing a failure. The `combine` function takes two previous computation and produces a new one which is the combination of both. The `augment` function is a bit more complex.

The `augment` function takes some previously given computation (namely, c a) and a function which takes the value of that computation (the a) and returns a b and produces a b inside of that computation. Note that in our current situation, giving `augment` the type $c\ a \rightarrow (a \rightarrow a) \rightarrow c\ a$ would have been sufficient, since a is always [Int], but we make it this more general time just for generality.

How `augment` works is probably best shown by example. We can define Maybe, Failable and [] to be instances of **Computation** as:

```
instance Computation Maybe where
    success = Just
    failure = const Nothing
    augment (Just x) f = f x
    augment Nothing  _ = Nothing
    combine Nothing y = y
    combine x _ = x
```

Here, success is represented with `Just` and `failure` ignores its argument and returns `Nothing`. The `combine` function takes the first success we found and ignores the rest. The function `augment` checks to see if we succeeded before (and thus had a `Just` something) and, if we did, applies `f` to it. If we failed before (and thus had a `Nothing`), we ignore the function and return `Nothing`.

```
instance Computation Failable where
    success = Success
    failure = Fail
    augment (Success x) f = f x
    augment (Fail s) _ = Fail s
    combine (Fail _) y = y
    combine x _ = x
```

These definitions are obvious. Finally:

```
instance Computation [] where
    success a = [a]
    failure = const []
    augment l f = concat (map f l)
    combine = (++)
```

Here, the value of a successful computation is a singleton list containing that value. Failure is represented with the empty list and to combine previous successes we simply catenate them. Finally, augmenting a computation amounts to mapping the function across the list of previous computations and concatentate them. we apply the function to each element in the list and then concatenate the results.

Using these computations, we can express all of the above versions of search as:

```
searchAll g@(Graph vl el) src dst
    | src == dst = success [src]
    | otherwise  = search’ el
    where search’ [] = failure "no path"
          search’ ((u,v,_):es)
              | src == u  = (searchAll g v dst ‘augment‘
                             (success . (u:)))
                            ‘combine‘ search’ es
              | otherwise = search’ es
```

In this, we see the uses of all the functions from the class **Computation**.

If you've understood this discussion of computations, you are in a very good position as you have understood the concept of *monads*, probably the most difficult concept in Haskell. In fact, the **Computation** class is almost exactly the **Monad** class, except that `success` is called `return`, `failure` is called `fail` and `augment` is called `>>=` (read "bind"). The `combine` function isn't actually required by monads, but is found in the **MonadPlus** class for reasons which will become obvious later.

If you didn't understand everything here, read through it again and then wait for the proper discussion of monads in Chapter 9.

8.5 Instances

We have already seen how to declare instances of some simple classes; allow us to consider some more advanced classes here. There is a **Functor** class defined in the **Functor** module.

■ NOTE ■ The name "functor", like "monad" comes from category theory. There, a functor is like a function, but instead of mapping elements to elements, it maps structures to structures.

The definition of the functor class is:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

The type definition for `fmap` (not to mention its name) is very similar to the function `map` over lists. In fact, `fmap` is essentially a generalization of `map` to arbitrary structures (and, of course, lists are already instances of **Functor**). However, we can also define other structures to be instances of functors. Consider the following datatype for binary trees:

```
data BinTree a = Leaf a
               | Branch (BinTree a) (BinTree a)
```

We can immediately identify that the BinTree type essentially "raises" a type a into trees of that type. There is a naturally associated functor which goes along with this raising. We can write the instance:

```
instance Functor BinTree where
    fmap f (Leaf a) = Leaf (f a)
    fmap f (Branch left right) =
        Branch (fmap f left) (fmap f right)
```

Now, we've seen how to make something like BinTree an instance of **Eq** by using the **deriving** keyword, but here we will do it by hand. We want to make BinTree as instances of **Eq** but obviously we cannot do this unless a is itself an instance of **Eq**. We can specify this dependence in the instance declaration:

```
instance Eq a => Eq (BinTree a) where
    Leaf a == Leaf b = a == b
    Branch l r == Branch l' r' = l == l' && r == r'
    _ == _ = False
```

The first line of this can be read "if a is an instance of **Eq**, then BinTree a is also an instance of **Eq**". We then provide the definitions. If we did not include the "`Eq a =>`" part, the compiler would complain because we're trying to use the == function on as in the second line.

The "`Eq a =>`" part of the definition is called the "context." We should note that there are some restrictions on what can appear in the context and what can appear in the declaration. For instance, we're not allowed to have instance declarations that don't contain type constructors on the right hand side. To see why, consider the following declarations:

```
class MyEq a where
    myeq :: a -> a -> Bool

instance Eq a => MyEq a where
    myeq = (==)
```

As it stands, there doesn't seem to be anything wrong with this definition. However, if elsewhere in a program we had the definition:

```
instance MyEq a => Eq a where
    (==) = myeq
```

In this case, if we're trying to establish if some type is an instance of **Eq**, we could reduce it to trying to find out if that type is an instance of **MyEq**, which we could in turn reduce to trying to find out if that type is an instance of **Eq**, and so on. The compiler protects itself against this by refusing the first instance declaration.

This is commonly known as the *closed-world assumption*. That is, we're assuming, when we write a definition like the first one, that there won't be any declarations like the second. However, this assumption is invalid because there's nothing to prevent the second declaration (or some equally evil declaration). The closed world assumption can also bite you in cases like:

```
class OnlyInts a where
    foo :: a -> a -> Bool
```

```
instance OnlyInts Int where
    foo == (==)

bar :: OnlyInts a => a -> Bool
bar = foo 5
```

We've again made the closed-world assumption: we've assumed that the only instance of **OnlyInts** is Int, but there's no reason another instance couldn't be defined elsewhere, ruining our defintion of `bar`.

8.6 Kinds

Let us take a moment and think about what types are available in Haskell. We have simple types, like Int, Char, Double and so on. We then have type constructors like Maybe which take a type (like Char) and produce a new type, Maybe Char. Similarly, the type constructor [] (lists) takes a type (like Int) and produces [Int]. We have more complex things like $\rightarrow$ (function arrow) which takes *two* types (say Int and Bool) and produces a new type Int $\rightarrow$ Bool.

In a sense, these types themselves have type. Types like Int have some sort of basic type. Types like Maybe have a type which takes something of basic type and returns something of basic type. And so forth.

Talking about the types of types becomes unwieldy and highly ambiguous, so we call the types of types "kinds." What we have been calling "basic types" have kind "`*`". Something of kind `*` is something which can have an actual value. There is also a single kind constructor, $\rightarrow$ with which we can build more complex kinds.

Consider Maybe. This takes something of kind `*` and produces something of kind `*`. Thus, the kind of Maybe is `* -> *`. Recall the definition of `Pair` from Section 4.5.1:

```
data Pair a b = Pair a b
```

Here, Pair is a type constructor which takes two arguments, each of kind `*` and produces a type of kind `*`. Thus, the kind of Pair is `* -> (* -> *)`. However, we again assume associativity so we just write `* -> * -> *`.

Let us make a slightly strange datatype definition:

```
data Strange c a b =
    MkStrange (c a) (c b)
```

Before we analyze the kind of Strange, let's think about what it does. It is essentially a pairing constructor, though it doesn't pair actual elements, but elements within another constructor. For instance, think of c as Maybe. Then MkStrange pairs Maybes of the two types a and b. However, c need not be Maybe but could instead by [], or many other things.

What do we know about c, though? We know that it must have kind `* -> *`. This is because we have `c a` on the right hand side. The type variables a and b each have kind `*` as before. Thus, the kind of Strange is `(* -> *) -> * -> * -> *`. That is, it takes a constructor (c) of kind `* -> *` together with two types of kind `*` and produces something of kind `*`.

A question may arise regarding how we know a has kind `*` and not some other kind `k`. In fact, the inferred kind for Strange is `(k -> *) -> k -> k -> *`. However, this requires polymorphism on the kind level, which is too complex, so we make a default assumption that `k = *`.

■ NOTE ■ There are extensions to GHC which allow you to specify the kind of constructors directly. For instance, if you wanted a different kind, you could write this explicitly:

```
data Strange (c :: (* -> *) -> *) a b = MkStrange (c a) (c b)
```

to give a different kind to Strange.

The notation of kinds suggests that we can perform partial application, as we can for functions. And, in fact, we can. For instance, we could have:

```
type MaybePair = Strange Maybe
```

The kind of MaybePair is, not surprisingly, `* -> * -> *`.

We should note here that all of the following definitions are acceptable:

```
type MaybePair1     = Strange Maybe
type MaybePair2 a   = Strange Maybe a
type MaybePair3 a b = Strange Maybe a b
```

These all appear to be the same, but they are in fact not identical as far as Haskell's type system is concerned. The following are all valid type definitions using the above:

```
type MaybePair1a = MaybePair1
type MaybePair1b = MaybePair1 Int
type MaybePair1c = MaybePair1 Int Double

type MaybePair2b = MaybePair2 Int
type MaybePair2c = MaybePair2 Int Double

type MaybePair3c = MaybePair3 Int Double
```

But the following are *not* valid:

```
type MaybePair2a = MaybePair2

type MaybePair3a = MaybePair3
type MaybePair3b = MaybePair3 Int
```

This is because while it is possible to partially apply type constructors on datatypes, it is not possible on type synonyms. For instance, the reason MaybePair2a is invalid is because MaybePair2 is defined as a type synonym with one argument and we have given it none. The same applies for the invalid MaybePair3 definitions.

8.7 Class Hierarchies

8.8 Default

what is it?

Chapter 9

Monads

The most difficult concept to master, while learning Haskell, is that of understanding and using monads. We can distinguish two subcomponents here: (1) learning how to use existing monads and (2) learning how to write new ones. If you want to use Haskell, you must learn to use existing monads. On the other hand, you will only need to learn to write your own monads if you want to become a "super Haskell guru." Still, if you can grasp writing your own monads, programming in Haskell will be much more pleasant.

So far we've seen two uses of monads. The first use was IO actions: We've seen that, by using monads, we can abstract get away from the problems plaguing the Real-World solution to IO presented in Chapter 5. The second use was representing different types of computations in Section 8.4.2. In both cases, we needed a way to sequence operations and saw that a sufficient definition (at least for computations) was: computations

```
class Computation c where
    success :: a -> c a
    failure :: String -> c a
    augment :: c a -> (a -> c b) -> c b
    combine :: c a -> c a -> c a
```

Let's see if this definition will enable us to also perform IO. Essentially, we need a way to represent taking a value out of an action and performing some new operation on it (as in the example from Section 4.4.3, rephrased slightly):

```
main = do
  s <- readFile "somefile"
  putStrLn (show (f s))
```

But this is exactly what `augment` does. Using `augment`, we can write the above code as:

```
main =  -- note the lack of a "do"
  readFile "somefile" ‘augment‘ \s ->
  putStrLn (show (f s))
```

This certainly seems to be sufficient. And, in fact, it turns out to be more than sufficient.

The definition of a monad is a slightly trimmed-down version of our **Computation** class. The **Monad** class has four methods (but the fourth method can be defined in terms of the third):

```
class Monad m where
    return  :: a -> m a
    fail    :: String -> m a
    (>>=)   :: m a -> (a -> m b) -> m b
    (>>)    :: m a -> m b -> m b
```

bind
then

In this definition, `return` is equivalent to our `success`; `fail` is equivalent to our `failure`; and `>>=` (read: "bind") is equivalent to our `augment`. The `>>` (read: "then") method is simply a version of `>>=` that ignores the `a`. This will turn out to be useful; although, as mentioned before, it can be defined in terms of `>>=`:

```
a >> x = a >>= \_ -> x
```

9.1 Do Notation

syntactic sugar

We have hinted that there is a connection between monads and the **do** notation. Here, we make that relationship concrete. There is actually nothing magic about the **do** notation – it is simply "syntactic sugar" for monadic operations.

As we mentioned earlier, using our **Computation** class, we could define our above program as:

```
main =
    readFile "somefile" ‘augment‘ \s ->
    putStrLn (show (f s))
```

But we now know that `augment` is called `>>=` in the monadic world. Thus, this program really reads:

```
main =
    readFile "somefile" >>= \s ->
    putStrLn (show (f s))
```

And this is completely valid Haskell at this point: if you defined a function `f :: Show a => String -> a`, you could compile and run this program)

This suggests that we can translate:

```
x <- f
g x
```

into `f >>= \x -> g x`. This is exactly what the compiler does. Talking about **do** becomes easier if we do not use implicit layout (see Section **??** for how to do this). There are four translation rules:

1. `do {e}` → `e`
2. `do {e; es}` → `e >> do {es}`
3. `do {let decls; es}` → `let decls in do {es}`
4. `do {p <- e; es}` → `let ok p = do {es} ; ok _ = fail "..."`
   `in e >>= ok`

Again, we will elaborate on these one at a time:

Translation Rule 1

The first translation rule, `do {e}` → `e`, states (as we have stated before) that when performing a single action, having a **do** or not is irrelevant. This is essentially the base case for an inductive definition of **do**. The base case has one action (namely `e` here); the other three translation rules handle the cases where there is more than one action.

Translation Rule 2

This states that `do {e; es}` → `e >> do {es}`. This tells us what to do if we have an action (`e`) followed by a list of actions (`es`). Here, we make use of the `>>` function, defined earlier. This rule simple states that to **do** `{e; es}`, we first perform the action `e`, throw away the result, and then **do** `es`.

For instance, if `e` is `putStrLn s` for some string `s`, then the translation of `do {e; es}` is to perform `e` (i.e., print the string) and then **do** `es`. This is clearly what we want.

Translation Rule 3

This states that `do {let decls; es}` → `let decls in do {es}`. This rule tells us how to deal with **let**s inside of a **do** statement. We lift the declarations within the **let** out and **do** whatever comes after the declarations. let

Translation Rule 4

This states that `do {p <- e; es}` $\rightarrow$ `let ok p = do {es} ; ok _ = fail "..." in e >>= ok`. Again, it is not exactly obvious what is going on here. However, an alternate formulation of this rule, which is roughly equivalent, is: `do {p <- e; es}` $\rightarrow$ `e >>= \p -> es`. Here, it is clear what is happening. We run the action `e`, and then send the results into `es`, but first give the result the name `p`.

The reason for the complex definition is that `p` doesn't need to simply be a variable; it could be some complex pattern. For instance, the following is valid code:

```
foo = do ('a':'b':'c':x:xs) <- getLine
         putStrLn (x:xs)
```

In this, we're assuming that the results of the action `getLine` will begin with the string "abc" and will have at least one more character. The question becomes what should happen if this pattern match fails. The compiler could simply throw an error, like usual, for failed pattern matches. However, since we're within a monad, we have access to a special `fail` function, and we'd prefer to fail using that function, rather than the "catch all" `error` function. Thus, the translation, as defined, allows the compiler to fill in the `...` with an appropriate error message about the pattern matching having failed. Apart from this, the two definitions are equivalent.

9.2 Definition

monad laws

There are three rules that all monads must obey called the "Monad Laws" (and it is up to *you* to ensure that your monads obey these rules) :

1. `return a >>= f` $\equiv$ `f a`
2. `f >>= return` $\equiv$ `f`
3. `f >>= (\x -> g x >>= h)` $\equiv$ `(f >>= g) >>= h`

Let's look at each of these individually:

Law 1

This states that `return a >>= f` $\equiv$ `f a`. Suppose we think about monads as computations. This means that if we create a trivial computation that simply returns the value `a` regardless of anything else (this is the `return a` part); and then bind it together with some other computation `f`, then this is equivalent to simply performing the computation `f` on `a` directly.

For example, suppose `f` is the function `putStrLn` and `a` is the string "Hello World." This rule states that binding a computation whose result is "Hello World" to `putStrLn` is the same as simply printing it to the screen. This seems to make sense.

In **do** notation, this law states that the following two programs are equivalent:

```
law1a = do
  x <- return a
  f x

law1b = do
  f a
```

Law 2

The second monad law states that `f >>= return` $\equiv$ `f` for some computation `f`. In other words, the law states that if we perform the computation `f` and then pass the result on to the trivial `return` function, then all we have done is to perform the computation.

That this law must hold should be obvious. To see this, think of `f` as `getLine` (reads a string from the keyboard). This law states that reading a string and then returning the value read is exactly the same as just reading the string.

In **do** notation, the law states that the following two programs are equivalent:

```
law2a = do
  x <- f
  return x

law2b = do
  f
```

Law 3

This states that `f >>= (\x -> g x >>= h)` $\equiv$ `(f >>= g) >>= h`. At first glance, this law is not as easy to grasp as the other two. It is essentially an associativity law for monads.

associative

■ NOTE ■ Outside the world of monads, a function $\cdot$ is associative if $(f \cdot g) \cdot h = f \cdot (g \cdot h)$. For instance, `+` and `*` are associative, since bracketing on these functions doesn't make a difference. On the other hand, `-` and `/` are not associative since, for example, $5 - (3 - 1) \neq (5 - 3) - 1$.

If we throw away the messiness with the lambdas, we see that this law states: `f >>= (g >>= h)` $\equiv$ `(f >>= g) >>= h`. The intuition behind this law is that when we string together actions, it doesn't matter how we group them.

For a concrete example, take `f` to be `getLine`. Take `g` to be an action which takes a value as input, prints it to the screen, reads another string via `getLine`, and then returns that newly read string. Take `h` to be `putStrLn`.

Let's consider what `(\x -> g x >>= h)` does. It takes a value called `x`, and runs `g` on it, feeding the results into `h`. In this instance, this means that it's going to take a value, print it, read another value and then print that. Thus, the entire left hand side of the law first reads a string and then does what we've just described.

On the other hand, consider `(f >>= g)`. This action reads a string from the keyboard, prints it, and then reads another string, returning that newly read string as a result. When we bind this with `h` as on the right hand side of the law, we get an action that does the action described by `(f >>= g)`, and then prints the results.

Clearly, these two actions are the same.

While this explanation is quite complicated, and the text of the law is also quite complicated, the actual meaning is simple: if we have three actions, and we compose them in the same order, it doesn't matter where we put the parentheses. The rest is just notation.

In **do** notation, the law says that the following two programs are equivalent:

```
law3a = do
  x <- f
  do y <- g x
     h y

law3b = do
  y <- do x <- f
          g x
  h y
```

9.3 A Simple State Monad

One of the simplest monads that we can craft is a state-passing monad. In Haskell, all state information usually must be passed to functions explicitly as arguments. Using monads, we can effectively hide some state information.

Suppose we have a function `f` of type $\mathsf{a} \to \mathsf{b}$, and we need to add state to this function. In general, if state is of type state, we can encode it by changing the type of `f` to $\mathsf{a} \to \mathsf{state} \to (\mathsf{state}, \mathsf{b})$. That is, the new version of `f` takes the original parameter of type `a` and a new state parameter. And, in addition to returning the value of type b, it also returns an updated state, encoded in a tuple.

For instance, suppose we have a binary tree defined as:

```
data Tree a
  = Leaf a
  | Branch (Tree a) (Tree a)
```

Now, we can write a simple map function to apply some function to each value in the leaves:

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf a) = Leaf (f a)
mapTree f (Branch lhs rhs) =
    Branch (mapTree f lhs) (mapTree f rhs)
```

This works fine until we need to write a function that numbers the leaves left to right. In a sense, we need to add state, which keeps track of how many leaves we've numbered so far, to the `mapTree` function. We can augment the function to something like:

```
mapTreeState :: (a -> state -> (state, b)) ->
                Tree a -> state -> (state, Tree b)
mapTreeState f (Leaf a) state =
    let (state', b) = f a state
    in  (state', Leaf b)
mapTreeState f (Branch lhs rhs) state =
    let (state' , lhs') = mapTreeState f lhs state
        (state'', rhs') = mapTreeState f rhs state'
    in  (state'', Branch lhs' rhs')
```

This is beginning to get a bit unweildy, and the type signature is getting harder and harder to understand. What we want to do is abstract away the state passing part. That is, the differences between `mapTree` and `mapTreeState` are: (1) the augmented `f` type, (2) we replaced the type `-> Tree b` with `-> state -> (state, Tree b)`. Notice that both types changed in exactly the same way. We can abstract this away with a type synonym declaration:

```
type State st a = st -> (st, a)
```

To go along with this type, we write two functions:

```
returnState :: a -> State st a
returnState a = \st -> (st, a)

bindState :: State st a -> (a -> State st b) ->
             State st b
bindState m k = \st ->
    let (st', a) = m st
        m'       = k a
    in  m' st'
```

Let's examine each of these in turn. The first function, `returnState`, takes a value of type `a` and creates something of type `State st a`. If we think of the `st` as the state, and the value of type `a` as the value, then this is a function that doesn't change the state and returns the value `a`.

The `bindState` function looks distinctly like the interior **let** declarations in `mapTreeState`. It takes two arguments. The first argument is an action that returns something of type a with state st. The second is a function that takes this a and produces something of type b also with the same state. The result of bindState is essentially the result of transforming the a into a b.

The definition of bindState takes an initial state, st. It first applies this to the State st a argument called m. This gives back a new state st' and a value a. It then lets the function k act on a, producing something of type State st b, called m'. We finally run m' with the new state st'.

We write a new function, mapTreeStateM and give it the type:

```
mapTreeStateM :: (a -> State st b) -> Tree a -> State st (Tree b)
```

Using these "plumbing" functions (returnState and bindState) we can write this function without ever having to explicitly talk about the state:

```
mapTreeStateM f (Leaf a) =
  f a `bindState` \b ->
  returnState (Leaf b)
mapTreeStateM f (Branch lhs rhs) =
  mapTreeStateM f lhs `bindState` \lhs' ->
  mapTreeStateM f rhs `bindState` \rhs' ->
  returnState (Branch lhs' rhs')
```

In the Leaf case, we apply f to a and then *bind* the result to a function that takes the result and returns a Leaf with the new value.

In the Branch case, we recurse on the left-hand-side, binding the result to a function that recurses on the right-hand-side, binding that to a simple function that returns the newly created Branch.

As you have probably guessed by this point, State st is a monad, returnState is analogous to the overloaded return method, and bindState is analogous to the overloaded >>= method. In fact, we can verify that State st a obeys the monad laws:

Law 1 states: return a >>= f ≡ f a. Let's calculate on the left hand side, substituting our names:

```
     returnState a `bindState` f
==>
     \st -> let (st', a) = (returnState a) st
                m'       = f a
            in  m' st'
==>
     \st -> let (st', a) = (\st -> (st, a)) st
            in  (f a) st'
==>
     \st -> let (st', a) = (st, a)
            in  (f a) st'
```

```
==>
     \st -> (f a) st
==>
     f a
```

In the first step, we simply substitute the definition of `bindState`. In the second step, we simplify the last two lines and substitute the definition of `returnState`. In the third step, we apply `st` to the lambda function. In the fourth step, we rename `st'` to `st` and remove the **let**. In the last step, we eta reduce.

Moving on to *Law 2*, we need to show that `f >>= return` ≡ `f`. This is shown as follows:

```
     f `bindState` returnState
==>
     \st -> let (st', a) = f st
            in  (returnState a) st'
==>
     \st -> let (st', a) = f st
            in  (\st -> (st, a)) st'
==>
     \st -> let (st', a) = f st
            in  (st', a)
==>
     \st -> f st
==>
     f
```

Finally, we need to show that `State` obeys the third law: `f >>= (\x -> g x >>= h)` ≡ `(f >>= g) >>= h`. This is much more involved to show, so we will only sketch the proof here. Notice that we can write the left-hand-side as:

```
     \st -> let (st', a) = f st
            in  (\x -> g x `bindState` h) a st'
==>
     \st -> let (st', a) = f st
            in  (g a `bindState` h) st'
==>
     \st -> let (st', a) = f st
            in  (\st' -> let (st'', b) = g a
                         in  h b st'') st'
==>
     \st -> let (st' , a) = f st
                (st'', b) = g a st'
                (st''',c) = h b st''
            in  (st''',c)
```

The interesting thing to note here is that we have both action applications on the same **let** level. Since **let** is associative, this means that we can put whichever bracketing we prefer and the results will not change. Of course, this is an informal, "hand waving" argument and it would take us a few more derivations to actually prove, but this gives the general idea.

Now that we know that `State st` is actually a monad, we'd like to make it an instance of the **Monad** class. Unfortunately, the straightforward way of doing this doesn't work. We can't write:

```
instance Monad (State st) where { ... }
```

This is because you cannot make instances out of non-fully-applied type synonyms. Instead, what we need to do instead is convert the type synonym into a **newtype**, as:

```
newtype State st a = State (st -> (st, a))
```

Unfortunately, this means that we need to do some packing and unpacking of the `State` constructor in the **Monad** instance declaration, but it's not terribly difficult:

```
instance Monad (State state) where
    return a = State (\state -> (state, a))
    State run >>= action = State run'
        where run' st =
                  let (st', a)    = run st
                      State run'' = action a
                  in  run'' st'
```

mapTreeM

Now, we can write our `mapTreeM` function as:

```
mapTreeM :: (a -> State state b) -> Tree a ->
            State state (Tree b)
mapTreeM f (Leaf a) = do
  b <- f a
  return (Leaf b)
mapTreeM f (Branch lhs rhs) = do
  lhs' <- mapTreeM f lhs
  rhs' <- mapTreeM f rhs
  return (Branch lhs' rhs')
```

which is significantly cleaner than before. In fact, if we remove the type signature, we get the more general type:

```
mapTreeM :: Monad m => (a -> m b) -> Tree a ->
            m (Tree b)
```

That is, `mapTreeM` can be run in *any* monad, not just our `State` monad.

Now, the nice thing about encapsulating the stateful aspect of the computation like this is that we can provide functions to get and change the current state. These look like:

getState
putState

```
getState :: State state state
getState = State (\state -> (state, state))

putState :: state -> State state ()
putState new = State (\_ -> (new, ()))
```

Here, `getState` is a monadic operation that takes the current state, passes it through unchanged, and then returns it as the value. The `putState` function takes a new state and produces an action that ignores the current state and inserts the new one.

Now, we can write our `numberTree` function as:

```
numberTree :: Tree a -> State Int (Tree (a, Int))
numberTree tree = mapTreeM number tree
    where number v = do
            cur <- getState
            putState (cur+1)
            return (v,cur)
```

Finally, we need to be able to run the action by providing an initial state:

```
runStateM :: State state a -> state -> a
runStateM (State f) st = snd (f st)
```

Now, we can provide an example Tree:

```
testTree =
  Branch
    (Branch
      (Leaf 'a')
      (Branch
        (Leaf 'b')
        (Leaf 'c')))
    (Branch
      (Leaf 'd')
      (Leaf 'e'))
```

and number it:

```
State> runStateM (numberTree testTree) 1
Branch (Branch (Leaf ('a',1)) (Branch (Leaf ('b',2))
       (Leaf ('c',3)))) (Branch (Leaf ('d',4))
       (Leaf ('e',5)))
```

This may seem like a large amount of work to do something simple. However, note the new power of `mapTreeM`. We can also print out the leaves of the tree in a left-to-right fashion as:

```
State> mapTreeM print testTree
'a'
'b'
'c'
'd'
'e'
```

This crucially relies on the fact that `mapTreeM` has the more general type involving arbitrary monads – not just the state monad. Furthermore, we can write an action that will make each leaf value equal to its old value as well as all the values preceeding:

```
fluffLeaves tree = mapTreeM fluff tree
    where fluff v = do
            cur <- getState
            putState (v:cur)
            return (v:cur)
```

and can see it in action:

```
State> runStateM (fluffLeaves testTree) []
Branch (Branch (Leaf "a") (Branch (Leaf "ba")
       (Leaf "cba"))) (Branch (Leaf "dcba")
       (Leaf "edcba"))
```

In fact, you don't even need to write your own monad instance and datatype. All this is built in to the `Control.Monad.State` module. There, our `runStateM` is called `evalState`; our `getState` is called `get`; and our `putState` is called `put`.

This module also contains a *state transformer monad*, which we will discuss in Section 9.7.

9.4 Common Monads

lists

It turns out that many of our favorite datatypes are actually monads themselves. Consider, for instance, lists. They have a monad definition that looks something like:

```
instance Monad [] where
    return x = [x]
    l >>= f  = concatMap f l
    fail _   = []
```

This enables us to use lists in do notation. For instance, given the definition:

```
cross l1 l2 = do
  x <- l1
  y <- l2
  return (x,y)
```

we get a cross-product function:

```
Monads> cross "ab" "def"
[('a','d'),('a','e'),('a','f'),('b','d'),('b','e'),
 ('b','f')]
```

It is not a coincidence that this looks very much like the list comprehension form: list comprehensions

```
Prelude> [(x,y) | x <- "ab", y <- "def"]
[('a','d'),('a','e'),('a','f'),('b','d'),('b','e'),
 ('b','f')]
```

List comprehension form is simply an abbreviated form of a monadic statement using lists. In fact, in older versions of Haskell, the list comprehension form could be used for *any* monad – not just lists. However, in the current version of Haskell, this is no longer allowed.

The Maybe type is also a monad, with failure being represented as `Nothing` and with success as `Just`. We get the following instance declaration: Maybe

```
instance Monad Maybe where
    return a      = Just a
    Nothing >>= f = Nothing
    Just x  >>= f = f x
    fail _        = Nothing
```

We can use the *same* cross product function that we did for lists on Maybes. This is because the **do** notation works for any monad, and there's nothing specific to lists about the `cross` function.

```
Monads> cross (Just 'a') (Just 'b')
Just ('a','b')
Monads> cross (Nothing :: Maybe Char) (Just 'b')
Nothing
Monads> cross (Just 'a') (Nothing :: Maybe Char)
Nothing
Monads> cross (Nothing :: Maybe Char)
                   (Nothing :: Maybe Char)
Nothing
```

What this means is that if we write a function (like `searchAll` from Section 8.4) only in terms of monadic operators, we can use it with any monad, depending on what we mean. Using real monadic functions (not **do** notation), the `searchAll` function looks something like:

sea

```
searchAll g@(Graph vl el) src dst
    | src == dst = return [src]
    | otherwise  = search' el
    where search' [] = fail "no path"
          search' ((u,v,_):es)
              | src == u  =
                  searchAll g v dst >>= \path ->
                  return (u:path)
              | otherwise = search' es
```

The type of this function is `Monad m => Graph v e -> Int -> Int -> m [Int]`. This means that no matter what monad we're using at the moment, this function will perform the calculation. Suppose we have the following graph:

```
gr = Graph [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
           [(0,1,'l'), (0,2,'m'), (1,3,'n'), (2,3,'m')]
```

This represents a graph with four nodes, labelled **a,b,c** and **d**. There is an edge from **a** to both **b** and **c**. There is also an edge from both **b** and **c** to **d**. Using the Maybe monad, we can compute the path from **a** to **d**:

```
Monads> searchAll gr 0 3 :: Maybe [Int]
Just [0,1,3]
```

We provide the type signature, so that the interpreter knows what monad we're using. If we try to search in the opposite direction, there is no path. The inability to find a path is represented as `Nothing` in the Maybe monad:

```
Monads> searchAll gr 3 0 :: Maybe [Int]
Nothing
```

Note that the string "no path" has disappeared since there's no way for the Maybe monad to record this.

If we perform the same impossible search in the list monad, we get the empty list, indicating no path:

```
Monads> searchAll gr 3 0 :: [[Int]]
[]
```

If we perform the possible search, we get back a list containing the first path:

```
Monads> searchAll gr 0 3 :: [[Int]]
[[0,1,3]]
```

You may have expected this function call to return *all* paths, but, as coded, it does not. See Section 9.6 for more about using lists to represent nondeterminism. nondeterminism

If we use the IO monad, we can actually get at the error message, since IO knows how to keep track of error messages:

```
Monads> searchAll gr 0 3 :: IO [Int]
Monads> it
[0,1,3]
Monads> searchAll gr 3 0 :: IO [Int]
*** Exception: user error
Reason: no path
```

In the first case, we needed to type `it` to get GHCi to actually evaluate the search.

There is one problem with this implementation of `searchAll`: if it finds an edge that does not lead to a solution, it won't be able to backtrack. This has to do with the recursive call to `searchAll` inside of `search'`. Consider, for instance, what happens if `searchAll g v dst` doesn't find a path. There's no way for this implementation to recover. For instance, if we remove the edge from node **b** to node **d**, we should still be able to find a path from **a** to **d**, but this algorithm can't find it. We define:

```
gr2 = Graph [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
            [(0,1,'l'), (0,2,'m'), (2,3,'m')]
```

and then try to search:

```
Monads> searchAll gr2 0 3
*** Exception: user error
Reason: no path
```

To fix this, we need a function like `combine` from our **Computation** class. We will see how to do this in Section 9.6.

Exercises

Exercise 9.1 *Verify that* Maybe *obeys the three monad laws.*

Exercise 9.2 *The type* Either String *is a monad that can keep track of errors. Write an instance for it, and then try doing the search from this chapter using this monad.*
Hint: Your instance declaration should begin: `instance Monad (Either String) where`.

9.5 Monadic Combinators

The Monad/Control.Monad library contains a few very useful monadic combinators, which haven't yet been thoroughly discussed. The ones we will discuss in this section, together with their types, are:

- `(=<<) :: (a -> m b) -> m a -> m b`
- `mapM :: (a -> m b) -> [a] -> m [b]`
- `mapM_ :: (a -> m b) -> [a] -> m ()`
- `filterM :: (a -> m Bool) -> [a] -> m [a]`
- `foldM :: (a -> b -> m a) -> a -> [b] -> m a`
- `sequence :: [m a] -> m [a]`
- `sequence_ :: [m a] -> m ()`
- `liftM :: (a -> b) -> m a -> m b`
- `when :: Bool -> m () -> m ()`
- `join :: m (m a) -> m a`

In the above, m is always assumed to be an instance of Monad.

In general, functions with an underscore at the end are equivalent to the ones without, except that they do not return any value.

The =<< function is exactly the same as >>=, except it takes its arguments in the opposite order. For instance, in the IO monad, we can write either of the following:

```
Monads> writeFile "foo" "hello world!" >>
             (readFile "foo" >>= putStrLn)
hello world!
Monads> writeFile "foo" "hello world!" >>
             (putStrLn =<< readFile "foo")
hello world!
```

mapM
filterM
foldM

The mapM, filterM and foldM are our old friends map, filter and foldr wrapped up inside of monads. These functions are incredibly useful (particularly foldM) when working with monads. We can use mapM_, for instance, to print a list of things to the screen:

```
Monads> mapM_ print [1,2,3,4,5]
1
2
3
4
5
```

We can use `foldM` to sum a list and print the intermediate sum at each step:

```
Monads> foldM (\a b ->
               putStrLn (show a ++ "+" ++ show b ++
                         "=" ++ show (a+b)) >>
               return (a+b)) 0 [1..5]
0+1=1
1+2=3
3+3=6
6+4=10
10+5=15
Monads> it
15
```

The `sequence` and `sequence_` functions simply "execute" a list of actions. For instance: sequence

```
Monads> sequence [print 1, print 2, print 'a']
1
2
'a'
*Monads> it
[(),(),()]
*Monads> sequence_ [print 1, print 2, print 'a']
1
2
'a'
*Monads> it
()
```

We can see that the underscored version doesn't return each value, while the non-underscored version returns the list of the return values.

The `liftM` function "lifts" a non-monadic function to a monadic function. (Do not confuse this with the `lift` function used for monad transformers in Section 9.7.) This is useful for shortening code (among other things). For instance, we might want to write a function that prepends each line in a file with its line number. We can do this with: liftM

```
numberFile :: FilePath -> IO ()
numberFile fp = do
  text <- readFile fp
  let l = lines text
  let n = zipWith (\n t -> show n ++ ' ' : t) [1..] l
  mapM_ putStrLn n
```

However, we can shorten this using `liftM`:

```
numberFile :: FilePath -> IO ()
numberFile fp = do
  l <- lines ‘liftM‘ readFile fp
  let n = zipWith (\n t -> show n ++ ’ ’ : t) [1..] l
  mapM_ putStrLn n
```

In fact, you can apply any sort of (pure) processing to a file using `liftM`. For instance, perhaps we also want to split lines into words; we can do this with:

```
  ...
  w <- (map words . lines) ‘liftM‘ readFile fp
  ...
```

Note that the parentheses are required, since the `(.)` function has the same fixity has `‘liftM‘`.

Lifting pure functions into monads is also useful in other monads. For instance `liftM` can be used to apply function inside of `Just`. For instance:

```
Monads> liftM (+1) (Just 5)
Just 6
*Monads> liftM (+1) Nothing
Nothing
```

when

The `when` function executes a monadic action only if a condition is met. So, if we only want to print non-empty lines:

```
Monads> mapM_ (\l -> when (not $ null l) (putStrLn l))
                   ["","abc","def","","","ghi"]
abc
def
ghi
```

Of course, the same could be accomplished with `filter`, but sometimes `when` is more convenient.

join

Finally, the `join` function is the monadic equivalent of `concat` on lists. In fact, when `m` is the list monad, `join` is exactly `concat`. In other monads, it accomplishes a similar task:

```
Monads> join (Just (Just ’a’))
Just ’a’
Monads> join (Just (Nothing :: Maybe Char))
Nothing
Monads> join (Nothing :: Maybe (Maybe Char))
Nothing
```

```
Monads> join (return (putStrLn "hello"))
hello
Monads> return (putStrLn "hello")
Monads> join [[1,2,3],[4,5]]
[1,2,3,4,5]
```

These functions will turn out to be even more useful as we move on to more advanced topics in Chapter 10.

9.6 MonadPlus

Given only the `>>=` and `return` functions, it is impossible to write a function like `combine` with type c a → c a → c a. However, such a function is so generally useful that it exists in another class called **MonadPlus**. In addition to having a `combine` function, instances of **MonadPlus** also have a "zero" element that is the identity under the "plus" (i.e., combine) action. The definition is:

combine
MonadPlus

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

In order to gain access to **MonadPlus**, you need to import the `Monad` module (or `Control.Monad` in the hierarchical libraries).

In Section 9.4, we showed that Maybe and list are both monads. In fact, they are also both instances of **MonadPlus**. In the case of Maybe, the zero element is `Nothing`; in the case of lists, it is the empty list. The `mplus` operation on Maybe is `Nothing`, if both elements are `Nothing`; otherwise, it is the first `Just` value. For lists, `mplus` is the same as `++`.

Maybe
lists

That is, the instance declarations look like:

```
instance MonadPlus Maybe where
  mzero = Nothing
  mplus Nothing y = y
  mplus x       _ = x

instance MonadPlus [] where
  mzero = []
  mplus x y = x ++ y
```

We can use this class to reimplement the search function we've been exploring, such that it will explore all possible paths. The new function looks like:

```
searchAll2 g@(Graph vl el) src dst
    | src == dst = return [src]
    | otherwise  = search’ el
    where search’ [] = fail "no path"
          search’ ((u,v,_):es)
              | src == u  =
                 (searchAll2 g v dst >>= \path ->
                  return (u:path)) ‘mplus‘
                 search’ es
              | otherwise = search’ es
```

Now, when we’re going through the edge list in `search’`, and we come across a matching edge, not only do we explore this path, but we also continue to explore the out-edges of the current node in the recursive call to `search’`.

The IO monad is not an instance of **MonadPlus**; we we’re not able to execute the search with this monad. We can see that when using lists as the monad, we (a) get all possible paths in `gr` and (b) get a path in `gr2`.

```
MPlus> searchAll2 gr 0 3 :: [[Int]]
[[0,1,3],[0,2,3]]
MPlus> searchAll2 gr2 0 3 :: [[Int]]
[[0,2,3]]
```

You might be tempted to implement this as:

```
searchAll2 g@(Graph vl el) src dst
    | src == dst = return [src]
    | otherwise  = search’ el
    where search’ [] = fail "no path"
          search’ ((u,v,_):es)
              | src == u  = do
                 path <- searchAll2 g v dst
                 rest <- search’ es
                 return ((u:path) ‘mplus‘ rest)
              | otherwise = search’ es
```

But note that this doesn’t do what we want. Here, if the recursive call to `searchAll2` fails, we don’t try to continue and execute `search’ es`. The call to `mplus` must be at the top level in order for it to work.

Exercises

Exercise 9.3 *Suppose that we changed the order of arguments to* `mplus`*. I.e., the matching case of* `search’` *looked like:*

```
                 search’ es ‘mplus‘
                 (searchAll2 g v dst >>= \path ->
                  return (u:path))
```

How would you expect this to change the results when using the list monad on `gr`*? Why?*

9.7 Monad Transformers

Often we want to "piggyback" monads on top of each other. For instance, there might be a case where you need access to both IO operations through the IO monad and state functions through some state monad. In order to accomplish this, we introduce a **MonadTrans** class, which essentially "lifts" the operations of one monad into another. You can think of this as *stacking* monads on top of eachother. This class has a simple method: `lift`. The class declaration for **MonadTrans** is:

MonadTrans

lift

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

The idea here is that `t` is the outer monad and that `m` lives inside of it. In order to execute a command of type `Monad m => m a`, we first `lift` it into the transformer.

The simplest example of a transformer (and arguably the most useful) is the state transformer monad, which is a state monad wrapped around an arbitrary monad. Before, we defined a state monad as:

state monad

```
newtype State state a = State (state -> (state, a))
```

Now, instead of using a function of type `state -> (state, a)` as the monad, we assume there's some other monad `m` and make the internal action into something of type `state -> m (state, a)`. This gives rise to the following definition for a state transformer:

state transformer

```
newtype StateT state m a =
        StateT (state -> m (state, a))
```

For instance, we can think of `m` as IO. In this case, our state transformer monad is able to execute actions in the IO monad. First, we make this an instance of **MonadTrans**:

```
instance MonadTrans (StateT state) where
    lift m = StateT (\s -> do a <- m
                              return (s,a))
```

Here, lifting a function from the realm of `m` to the realm of `StateT state` simply involves keeping the state (the `s` value) constant and executing the action.

Of course, we also need to make `StateT` a monad, itself. This is relatively straightforward, provided that `m` is already a monad:

```
instance Monad m => Monad (StateT state m) where
  return a = StateT (\s -> return (s,a))
  StateT m >>= k = StateT (\s -> do
    (s', a) <- m s
    let StateT m' = k a
    m' s')
  fail s = StateT (\_ -> fail s)
```

The idea behind the definition of `return` is that we keep the state constant and simply return the state/a pair in the enclosed monad. Note that the use of `return` in the definition of `return` refers to the enclosed monad, not the state transformer.

In the definition of bind, we create a new `StateT` that takes a state `s` as an argument. First, it applies this state to the first action (`StateT m`) and gets the new state and answer as a result. It then runs the `k` action on this new state and gets a new transformer. It finally applies the new state to this transformer. This definition is nearly identical to the definition of bind for the standard (non-transformer) `State` monad described in Section 9.3.

The `fail` function passes on the call to `fail` in the enclosed monad, since state transformers don't natively know how to deal with failure.

getT
putT
evalStateT

Of course, in order to actually use this monad, we need to provide function `getT` , `putT` and `evalStateT` . These are analogous to `getState`, `putState` and `runStateM` from Section 9.3:

```
getT :: Monad m => StateT s m s
getT = StateT (\s -> return (s, s))

putT :: Monad m => s -> StateT s m ()
putT s = StateT (\_ -> return (s, ()))

evalStateT :: Monad m => StateT s m a -> s -> m a
evalStateT (StateT m) state = do
  (s', a) <- m state
  return a
```

These functions should be straightforward. Note, however, that the result of `evalStateT` is actually a monadic action in the enclosed monad. This is typical of monad transformers: they don't know how to actually run things in their enclosed monad (they only know how to `lift` actions). Thus, what you get out is a monadic action in the inside monad (in our case, IO), which you then need to run yourself.

We can use state transformers to reimplement a version of our `mapTreeM` function from Section 9.3. The only change here is that when we get to a leaf, we print out the value of the leaf; when we get to a branch, we just print out "Branch."

```
mapTreeM action (Leaf a) = do
  lift (putStrLn ("Leaf " ++ show a))
  b <- action a
  return (Leaf b)
mapTreeM action (Branch lhs rhs) = do
  lift (putStrLn "Branch")
  lhs’ <- mapTreeM action lhs
  rhs’ <- mapTreeM action rhs
  return (Branch lhs’ rhs’)
```

The only difference between this function and the one from Section 9.3 is the calls to `lift (putStrLn ...)` as the first line. The `lift` tells us that we’re going to be executing a command in an enclosed monad. In this case, the enclosed monad is `IO`, since the command lifted is `putStrLn`.

The type of this function is relatively complex:

```
mapTreeM :: (MonadTrans t, Monad (t IO), Show a) =>
            (a -> t IO a1) -> Tree a -> t IO (Tree a1)
```

Ignoring, for a second, the class constraints, this says that `mapTreeM` takes an action and a tree and returns a tree. This just as before. In this, we require that `t` is a monad transformer (since we apply `lift` in it); we require that `t IO` is a monad, since we use `putStrLn` we know that the enclosed monad is `IO`; finally, we require that `a` is an instance of show – this is simply because we use `show` to show the value of leaves.

Now, we simply change `numberTree` to use this version of `mapTreeM`, and the new versions of `get` and `put`, and we end up with:

```
numberTree tree = mapTreeM number tree
    where number v = do
            cur <- getT
            putT (cur+1)
            return (v,cur)
```

Using this, we can run our monad:

```
MTrans> evalStateT (numberTree testTree) 0
Branch
Branch
Leaf ’a’
Branch
Leaf ’b’
Leaf ’c’
Branch
Leaf ’d’
Leaf ’e’
*MTrans> it
```

```
Branch (Branch (Leaf ('a',0))
       (Branch (Leaf ('b',1)) (Leaf ('c',2))))
       (Branch (Leaf ('d',3)) (Leaf ('e',4)))
```

cycles

One problem not specified in our discussion of **MonadPlus** is that our search algorithm will fail to terminate on graphs with cycles. Consider:

```
gr3 = Graph [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
            [(0,1,'l'), (1,0,'m'), (0,2,'n'),
             (1,3,'o'), (2,3,'p')]
```

In this graph, there is a back edge from node **b** back to node **a**. If we attempt to run `searchAll2`, regardless of what monad we use, it will fail to terminate. Moreover, if we move this erroneous edge to the end of the list (and call this `gr4`), the result of `searchAll2 gr4 0 3` will contain an infinite number of paths: presumably we only want paths that don't contain cycles.

In order to get around this problem, we need to introduce state. Namely, we need to keep track of which nodes we have visited, so that we don't visit them again.

We can do this as follows:

```
searchAll5 g@(Graph vl el) src dst
  | src == dst = do
      visited <- getT
      putT (src:visited)
      return [src]
  | otherwise  = do
      visited <- getT
      putT (src:visited)
      if src `elem` visited
        then mzero
        else search' el
  where
    search' [] = mzero
    search' ((u,v,_):es)
        | src == u  =
          (do path <- searchAll5 g v dst
              return (u:path)) `mplus`
          search' es
        | otherwise = search' es
```

Here, we implicitly use a state transformer (see the calls to `getT` and `putT`) to keep track of visited states. We only continue to recurse, when we encounter a state we haven't yet visited. Futhermore, when we recurse, we add the current state to our set of visited states.

Now, we can run the state transformer and get out only the correct paths, even on the cyclic graphs:

```
MTrans> evalStateT (searchAll5 gr3 0 3) [] :: [[Int]]
[[0,1,3],[0,2,3]]
MTrans> evalStateT (searchAll5 gr4 0 3) [] :: [[Int]]
[[0,1,3],[0,2,3]]
```

Here, the empty list provided as an argument to `evalStateT` is the initial state (i.e., the initial visited list). In our case, it is empty.

We can also provide an `execStateT` method that, instead of returning a result, returns the final state. This function looks like:

```
execStateT :: Monad m => StateT s m a -> s -> m s
execStateT (StateT m) state = do
  (s', a) <- m state
  return s'
```

This is not so useful in our case, as it will return exactly the reverse of `evalStateT` (try it and find out!), but can be useful in general (if, for instance, we need to know how many numbers are used in `numberTree`).

Exercises

Exercise 9.4 *Write a function* `searchAll6`*, based on the code for* `searchAll2`*, that, at every entry to the main function (not the recursion over the edge list), prints the search being conducted. For instance, the output generated for* `searchAll6 gr 0 3` *should look like:*

```
Exploring 0 -> 3
Exploring 1 -> 3
Exploring 3 -> 3
Exploring 2 -> 3
Exploring 3 -> 3
MTrans> it
[[0,1,3],[0,2,3]]
```

In order to do this, you will have to define your own list monad transformer and make appropriate instances of it.

Exercise 9.5 *Combine the* `searchAll5` *function (from this section) with the* `searchAll6` *function (from the previous exercise) into a single function called* `searchAll7`*. This function should perform IO as in* `searchAll6` *but should also keep track of state using a state transformer.*

9.8 Parsing Monads

It turns out that a certain class of parsers are all monads. This makes the construction of parsing libraries in Haskell very clean. In this chapter, we begin by building our own (small) parsing library in Section 9.8.1 and then introduce the Parsec parsing library in Section 9.8.2.

9.8.1 A Simple Parsing Monad

Consider the task of parsing. A simple parsing monad is much like a state monad, where the state is the unparsed string. We can represent this exactly as:

```
newtype Parser a = Parser
    { runParser :: String -> Either String (String, a) }
```

We again use `Left err` to be an error condition. This yields standard instances of **Monad** and **MonadPlus**:

```
instance Monad Parser where
    return a = Parser (\xl -> Right (xl,a))
    fail   s = Parser (\xl -> Left  s)
    Parser m >>= k = Parser $ \xl ->
      case m xl of
        Left  s -> Left s
        Right (xl’, a) ->
            let Parser n = k a
            in  n xl’

instance MonadPlus Parser where
    mzero = Parser (\xl -> Left "mzero")
    Parser p ‘mplus‘ Parser q = Parser $ \xl ->
      case p xl of
        Right a -> Right a
        Left  err -> case q xl of
                       Right a -> Right a
                       Left  _ -> Left err
```

primitives

Now, we want to build up a library of paring “primitives.” The most basic primitive is a parser that will read a specific character. This function looks like:

```
char :: Char -> Parser Char
char c = Parser char’
    where char’ [] = Left ("expecting " ++ show c ++
                          " got EOF")
          char’ (x:xs)
              | x == c    = Right (xs, c)
```

```
                | otherwise = Left  ("expecting " ++
                                     show c ++ " got " ++
                                     show x)
```

Here, the parser succeeds only if the first character of the input is the expected character.

We can use this parser to build up a parser for the string “Hello”:

```
helloParser :: Parser String
helloParser = do
  char ’H’
  char ’e’
  char ’l’
  char ’l’
  char ’o’
  return "Hello"
```

This shows how easy it is to combine these parsers. We don’t need to worry about the underlying string – the monad takes care of that for us. All we need to do is combine these parser primatives. We can test this parser by using `runParser` and by supplying input:

runParser

```
Parsing> runParser helloParser "Hello"
Right ("","Hello")
Parsing> runParser helloParser "Hello World!"
Right (" World!","Hello")
Parsing> runParser helloParser "hello World!"
Left "expecting ’H’ got ’h’"
```

We can have a slightly more general function, which will match any character fitting a description:

```
matchChar :: (Char -> Bool) -> Parser Char
matchChar c = Parser matchChar’
    where matchChar’ [] =
              Left ("expecting char, got EOF")
          matchChar’ (x:xs)
              | c x       = Right (xs, x)
              | otherwise =
                Left  ("expecting char, got " ++
                       show x)
```

Using this, we can write a case-insensitive “Hello” parser:

```
ciHelloParser = do
  c1 <- matchChar (‘elem‘ "Hh")
  c2 <- matchChar (‘elem‘ "Ee")
  c3 <- matchChar (‘elem‘ "Ll")
  c4 <- matchChar (‘elem‘ "Ll")
  c5 <- matchChar (‘elem‘ "Oo")
  return [c1,c2,c3,c4,c5]
```

Of course, we could have used something like `matchChar ((==’h’) . toLower)`, but the above implementation works just as well. We can test this function:

```
Parsing> runParser ciHelloParser "hELlO world!"
Right (" world!","hELlO")
```

Finally, we can have a function, which will match any character:

```
anyChar :: Parser Char
anyChar = Parser anyChar’
    where anyChar’ []     =
             Left  ("expecting character, got EOF")
          anyChar’ (x:xs) = Right (xs, x)
```

many

On top of these primitives, we usually build some combinators. The `many` combinator, for instance, will take a parser that parses entities of type `a` and will make it into a parser that parses entities of type `[a]` (this is a Kleene-star operator):

```
many :: Parser a -> Parser [a]
many (Parser p) = Parser many’
    where many’ xl =
              case p xl of
                Left  err -> Right (xl, [])
                Right (xl’,a) ->
                    let Right (xl’’, rest) = many’ xl’
                    in  Right (xl’’, a:rest)
```

The idea here is that first we try to apply the given parser, `p`. If this fails, we *succeed* but return the empty list. If `p` succeeds, we recurse and keep trying to apply `p` until it fails. We then return the list of successes we’ve accumulated.

In general, there would be many more functions of this sort, and they would be hidden away in a library, so that users couldn’t actually look inside the `Parser` type. However, using them, you could build up, for instance, a parser that parses (non-negative) integers:

```
int :: Parser Int
int = do
  t1 <- matchChar isDigit
  tr <- many (matchChar isDigit)
  return (read (t1:tr))
```

In this function, we first match a digit (the `isDigit` function comes from the module `Char`/`Data.Char`) and then match as many more digits as we can. We then `read` the result and return it. We can test this parser as before:

```
Parsing> runParser int "54"
Right ("",54)
*Parsing> runParser int "54abc"
Right ("abc",54)
*Parsing> runParser int "a54abc"
Left "expecting char, got 'a'"
```

Now, suppose we want to parse a Haskell-style list of Ints. This becomes somewhat difficult because, at some point, we're either going to parse a comma or a close brace, but we don't know when this will happen. This is where the fact that `Parser` is an instance of **MonadPlus** comes in handy: first we try one, then we try the other.

Consider the following code:

```
intList :: Parser [Int]
intList = do
  char '['
  intList' `mplus` (char ']' >> return [])
    where intList' = do
            i <- int
            r <- (char ',' >> intList') `mplus`
                 (char ']' >> return [])
            return (i:r)
```

The first thing this code does is parse and open brace. Then, using `mplus`, it tries one of two things: parsing using `intList'`, or parsing a close brace and returning an empty list. mplus

The `intList'` function assumes that we're not yet at the end of the list, and so it first parses an int. It then parses the rest of the list. However, it doesn't know whether we're at the end yet, so it again uses `mplus`. On the one hand, it tries to parse a comma and then recurse; on the other, it parses a close brace and returns the empty list. Either way, it simply prepends the int it parsed itself to the beginning.

One thing that you should be careful of is the order in which you supply arguments to `mplus`. Consider the following parser:

```
tricky =
  mplus (string "Hal") (string "Hall")
```

You might expect this parser to parse both the words “Hal” and “Hall;” however, it only parses the former. You can see this with:

```
Parsing> runParser tricky "Hal"
Right ("","Hal")
Parsing> runParser tricky "Hall"
Right ("l","Hal")
```

This is because it tries to parse “Hal,” which succeeds, and then it doesn’t bother trying to parse “Hall.”

You can attempt to fix this by providing a parser primitive, which detects end-of-file (really, end-of-string) as:

```
eof :: Parser ()
eof = Parser eof'
    where eof' [] = Right ([], ())
          eof' xl = Left ("Expecting EOF, got " ++
                          show (take 10 xl))
```

You might then rewrite `tricky` using `eof` as:

```
tricky2 = do
  s <- mplus (string "Hal") (string "Hall")
  eof
  return s
```

But this also doesn’t work, as we can easily see:

```
Parsing> runParser tricky2 "Hal"
Right ("",())
Parsing> runParser tricky2 "Hall"
Left "Expecting EOF, got \"l\""
```

This is because, again, the `mplus` doesn’t know that it needs to parse the whole input. So, when you provide it with “Hall,” it parses just “Hal” and leaves the last “l” lying around to be parsed later. This causes `eof` to produce an error message.

The correct way to implement this is:

```
tricky3 =
  mplus (do s <- string "Hal"
            eof
            return s)
        (do s <- string "Hall"
            eof
            return s)
```

We can see that this works:

```
Parsing> runParser tricky3 "Hal"
Right ("","Hal")
Parsing> runParser tricky3 "Hall"
Right ("","Hall")
```

This works precisely because each side of the `mplus` knows that it must read the end.

In this case, fixing the parser to accept both "Hal" and "Hall" was fairly simple, due to the fact that we assumed we would be reading an end-of-file immediately afterwards. Unfortunately, if we cannot disambiguate immediately, life becomes significantly more complicated. This is a general problem in parsing, and has little to do with monadic parsing. The solution most parser libraries (e.g., Parsec, see Section 9.8.2) have adopted is to only recognize "LL(1)" grammars: that means that you must be able to disambiguate the input with a one token look-ahead.

Exercises

Exercise 9.6 *Write a parser* `intListSpace` *that will parse int lists but will allow arbitrary white space (spaces, tabs or newlines) between the commas and brackets.*

Given this monadic parser, it is fairly easy to add information regarding source position. For instance, if we're parsing a large file, it might be helpful to report the line number on which an error occurred. We could do this simply by extending the `Parser` type and by modifying the instances and the primitives:

line numbers

```
newtype Parser a = Parser
    { runParser :: Int -> String ->
                   Either String (Int, String, a) }

instance Monad Parser where
  return a = Parser (\n xl -> Right (n,xl,a))
  fail   s = Parser (\n xl -> Left  (show n ++
                                     ": " ++ s))
  Parser m >>= k = Parser $ \n xl ->
    case m n xl of
      Left  s -> Left s
      Right (n', xl', a) ->
          let Parser m2 = k a
          in  m2 n' xl'

instance MonadPlus Parser where
  mzero = Parser (\n xl -> Left "mzero")
  Parser p `mplus` Parser q = Parser $ \n xl ->
    case p n xl of
      Right a -> Right a
      Left  err -> case q n xl of
                     Right a -> Right a
                     Left  _ -> Left err

matchChar :: (Char -> Bool) -> Parser Char
matchChar c = Parser matchChar'
  where matchChar' n [] =
            Left ("expecting char, got EOF")
        matchChar' n (x:xs)
            | c x       =
              Right (n+if x=='\n' then 1 else 0
                    , xs, x)
            | otherwise =
              Left  ("expecting char, got " ++
                     show x)
```

The definitions for `char` and `anyChar` are not given, since they can be written in terms of `matchChar`. The `many` function needs to be modified only to include the new state.

Now, when we run a parser and there is an error, it will tell us which line number contains the error:

```
Parsing2> runParser helloParser 1 "Hello"
Right (1,"","Hello")
Parsing2> runParser int 1 "a54"
Left "1: expecting char, got 'a'"
Parsing2> runParser intList 1 "[1,2,3,a]"
Left "1: expecting ']' got '1'"
```

We can use the `intListSpace` parser from the prior exercise to see that this does in fact work:

```
Parsing2> runParser intListSpace 1
               "[1 ,2 , 4  \n\n ,a\n]"
Left "3: expecting char, got 'a'"
Parsing2> runParser intListSpace 1
               "[1 ,2 , 4  \n\n\n ,a\n]"
Left "4: expecting char, got 'a'"
Parsing2> runParser intListSpace 1
```

```
               "[1 ,\n2 , 4  \n\n\n ,a\n]"
Left "5: expecting char, got 'a'"
```

We can see that the line number, on which the error occurs, increases as we add additional newlines before the erroneous "a".

9.8.2 Parsec

As you continue developing your parser, you might want to add more and more features. Luckily, Graham Hutton and Daan Leijen have already done this for us in the Parsec library. This section is intended to be an introduction to the Parsec library; it by no means covers the whole library, but it should be enough to get you started.

Like our libarary, Parsec provides a few basic functions to build parsers from characters. These are: `char`, which is the same as our `char`; `anyChar`, which is the same as our `anyChar`; `satisfy`, which is the same as our `matchChar`; `oneOf`, which takes a list of Chars and matches any of them; and `noneOf`, which is the opposite of `oneOf`.

char
anyChar
satisfy
oneOf
noneOf

The primary function Parsec uses to run a parser is `parse`. However, in addition to a parser, this function takes a string that represents the name of the file you're parsing. This is so it can give better error messages. We can try parsing with the above functions:

parse

```
ParsecI> parse (char 'a') "stdin" "a"
Right 'a'
ParsecI> parse (char 'a') "stdin" "ab"
Right 'a'
ParsecI> parse (char 'a') "stdin" "b"
Left "stdin" (line 1, column 1):
unexpected "b"
expecting "a"
ParsecI> parse (char 'H' >> char 'a' >> char 'l')
            "stdin" "Hal"
Right 'l'
ParsecI> parse (char 'H' >> char 'a' >> char 'l')
            "stdin" "Hap"
Left "stdin" (line 1, column 3):
unexpected "p"
expecting "l"
```

Here, we can see a few differences between our parser and Parsec: first, the rest of the string isn't returned when we run `parse`. Second, the error messages produced are much better.

In addition to the basic character parsing functions, Parsec provides primitives for: `spaces`, which is the same as ours; `space` which parses a single space; `letter`, which parses a letter; `digit`, which parses a digit; `string`, which is the same as ours; and a few others.

spaces
space
letter
digit
string

We can write our `int` and `intList` functions in Parsec as:

```
int :: CharParser st Int
int = do
  i1 <- digit
  ir <- many digit
  return (read (i1:ir))

intList :: CharParser st [Int]
intList = do
  char '['
  intList' `mplus` (char ']' >> return [])
    where intList' = do
            i <- int
            r <- (char ',' >> intList') `mplus`
                 (char ']' >> return [])
            return (i:r)
```

First, note the type signatures. The `st` type variable is simply a state variable that we are not using. In the `int` function, we use the `many` function (built in to Parsec) together with the `digit` function (also built in to Parsec). The `intList` function is actually identical to the one we wrote before.

Note, however, that using `mplus` explicitly is not the preferred method of combining parsers: Parsec provides a `<|>` function that is a synonym of `mplus`, but that looks nicer:

```
intList :: CharParser st [Int]
intList = do
  char '['
  intList' <|> (char ']' >> return [])
    where intList' = do
            i <- int
            r <- (char ',' >> intList') <|>
                 (char ']' >> return [])
            return (i:r)
```

We can test this:

```
ParsecI> parse intList "stdin" "[3,5,2,10]"
Right [3,5,2,10]
ParsecI> parse intList "stdin" "[3,5,a,10]"
Left "stdin" (line 1, column 6):
unexpected "a"
expecting digit
```

In addition to these basic combinators, Parsec provides a few other useful ones:

- `choice` takes a list of parsers and performs an *or* operation (`<|>`) between all of them.
- `option` takes a default value of type `a` and a parser that returns something of type `a`. It then tries to parse with the parser, but it uses the default value as the return, if the parsing fails.
- `optional` takes a parser that returns `()` and optionally runs it.
- `between` takes three parsers: an open parser, a close parser and a between parser. It runs them in order and returns the value of the between parser. This can be used, for instance, to take care of the brackets on our `intList` parser.
- `notFollowedBy` takes a parser and returns one that succeeds only if the given parser would have failed.

Suppose we want to parse a simple calculator language that includes only plus and times. Furthermore, for simplicity, assume each embedded expression must be enclosed in parentheses. We can give a datatype for this language as:

```
data Expr = Value Int
          | Expr :+: Expr
          | Expr :*: Expr
          deriving (Eq, Ord, Show)
```

And then write a parser for this language as:

```
parseExpr :: Parser Expr
parseExpr = choice
  [ do i <- int; return (Value i)
  , between (char '(') (char ')') $ do
      e1 <- parseExpr
      op <- oneOf "+*"
      e2 <- parseExpr
      case op of
        '+' -> return (e1 :+: e2)
        '*' -> return (e1 :*: e2)
  ]
```

Here, the parser alternates between two options (we could have used `<|>`, but I wanted to show the `choice` combinator in action). The first simply parses an int and then wraps it up in the `Value` constructor. The second option uses `between` to parse text between parentheses. What it parses is first an expression, then one of plus or times, then another expression. Depending on what the operator is, it returns either `e1 :+: e2` or `e1 :*: e2`.

We can modify this parser, so that instead of computing an `Expr`, it simply computes the value:

```
parseValue :: Parser Int
parseValue = choice
  [int
  ,between (char '(') (char ')') $ do
     e1 <- parseValue
     op <- oneOf "+*"
     e2 <- parseValue
     case op of
       '+' -> return (e1 + e2)
       '*' -> return (e1 * e2)
  ]
```

We can use this as:

```
ParsecI> parse parseValue "stdin" "(3*(4+3))"
Right 21
```

bindings

getState

setState

updateState

Now, suppose we want to introduce bindings into our language. That is, we want to also be able to say "let x = 5 in" inside of our expressions and then use the variables we've defined. In order to do this, we need to use the `getState` and `setState` (or `updateState`) functions built in to Parsec.

```
parseValueLet :: CharParser (FiniteMap Char Int) Int
parseValueLet = choice
  [ int
  , do string "let "
       c <- letter
       char '='
       e <- parseValueLet
       string " in "
       updateState (\fm -> addToFM fm c e)
       parseValueLet
  , do c  <- letter
       fm <- getState
       case lookupFM fm c of
         Nothing -> unexpected ("variable " ++ show c ++
                                " unbound")
         Just  i -> return i
  , between (char '(') (char ')') $ do
      e1 <- parseValueLet
      op <- oneOf "+*"
      e2 <- parseValueLet
      case op of
        '+' -> return (e1 + e2)
      '*' -> return (e1 * e2)
  ]
```

The `int` and recursive cases remain the same. We add two more cases, one to deal with let-bindings, the other to deal with usages.

In the let-bindings case, we first parse a "let" string, followed by the character we're binding (the `letter` function is a Parsec primitive that parses alphabetic characters), followed by it's value (a `parseValueLet`). Then, we parse the " in " and update the state to include this binding. Finally, we continue and parse the rest.

In the usage case, we simply parse the character and then look it up in the state. However, if it doesn't exist, we use the Parsec primitive `unexpected` to report an error.

We can see this parser in action using the `runParser` command, which enables us to provide an initial state:

runParser

```
ParsecI> runParser parseValueLet emptyFM "stdin"
                 "let c=5 in ((5+4)*c)"
Right 45
*ParsecI> runParser parseValueLet emptyFM "stdin"
                 "let c=5 in ((5+4)*let x=2 in (c+x))"
Right 63
*ParsecI> runParser parseValueLet emptyFM "stdin"
                 "((let x=2 in 3+4)*x)"
Right 14
```

Note that the bracketing does not affect the definitions of the variables. For instance, in the last example, the use of "x" is, in some sense, outside the scope of the definition. However, our parser doesn't notice this, since it operates in a strictly left-to-right fashion. In order to fix this omission, bindings would have to be removed (see the exercises).

Exercises

Exercise 9.7 *Modify the* `parseValueLet` *parser, so that it obeys bracketing. In order to do this, you will need to change the state to something like* `FiniteMap Char [Int]`*, where the* `[Int]` *is a stack of definitions.*

Chapter 10

Advanced Techniques

Appendix A

Brief Complexity Theory

Complexity Theory is the study of how long a program will take to run, depending on the size of its input. There are many good introductory books to complexity theory and the basics are explained in any good algorithms book. I'll keep the discussion here to a minimum.

The idea is to say how well a program scales with more data. If you have a program that runs quickly on very small amounts of data but chokes on huge amounts of data, it's not very useful (unless you know you'll only be working with small amounts of data, of course). Consider the following Haskell function to return the sum of the elements in a list:

```
sum [] = 0
sum (x:xs) = x + sum xs
```

How long does it take this function to complete? That's a very difficult question; it would depend on all sorts of things: your processor speed, your amount of memory, the exact way in which the addition is carried out, the length of the list, how many other programs are running on your computer, and so on. This is far too much to deal with, so we need to invent a simpler model. The model we use is sort of an arbitrary "machine step." So the question is "how many machine steps will it take for this program to complete?" In this case, it only depends on the length of the input list.

If the input list is of length 0, the function will take either 0 or 1 or 2 or some very small number of machine steps, depending exactly on how you count them (perhaps 1 step to do the pattern matching and 1 more to return the value 0). What if the list is of length 1. Well, it would take however much time the list of length 0 would take, plus a few more steps for doing the first (and only element).

If the input list is of length n, it will take however many steps an empty list would take (call this value y) and then, for each element it would take a certain number of steps to do the addition and the recursive call (call this number x). Then, the total time this function will take is $nx + y$ since it needs to do those additions n many times. These x and y values are called *constant values*, since they are independent of n, and actually dependent only on exactly how we define a machine step, so we really don't want to consider them all that important. Therefore, we say that the complexity of this `sum` function is $\mathcal{O}(n)$ (read "order n"). Basically saying something is $\mathcal{O}(n)$ means that for some constant factors x and y, the function takes $nx+y$ machine steps to complete.

Consider the following sorting algorithm for lists (commonly called "insertion sort"):

```
sort []  = []
sort [x] = [x]
sort (x:xs) = insert (sort xs)
    where insert [] = [x]
          insert (y:ys) | x <= y    = x : y : ys
                        | otherwise = y : insert ys
```

The way this algorithm works is as follow: if we want to sort an empty list or a list of just one element, we return them as they are, as they are already sorted. Otherwise, we have a list of the form `x:xs`. In this case, we sort `xs` and then want to insert `x` in the appropriate location. That's what the `insert` function does. It traverses the now-sorted tail and inserts `x` wherever it naturally fits.

Let's analyze how long this function takes to complete. Suppose it takes $f(n)$ stepts to sort a list of length n. Then, in order to sort a list of n-many elements, we first have to sort the tail of the list first, which takes $f(n-1)$ time. Then, we have to insert `x` into this new list. If `x` has to go at the end, this will take $\mathcal{O}(n-1) = \mathcal{O}(n)$ steps. Putting all of this together, we see that we have to do $\mathcal{O}(n)$ amount of work $\mathcal{O}(n)$ many times, which means that the entire complexity of this sorting algorithm is $\mathcal{O}(n^2)$. Here, the squared is not a constant value, so we cannot throw it out.

What does this mean? Simply that for really long lists, the `sum` function won't take very long, but that the `sort` function will take quite some time. Of course there are algorithms that run much more slowly that simply $\mathcal{O}(n^2)$ and there are ones that run more quickly than $\mathcal{O}(n)$.

Consider the random access functions for lists and arrays. In the worst case, accessing an arbitrary element in a list of length n will take $\mathcal{O}(n)$ time (think about accessing the last element). However with arrays, you can access any element immediately, which is said to be in *constant* time, or $\mathcal{O}(1)$, which is basically as fast an any algorithm can go.

There's much more in complexity theory than this, but this should be enough to allow you to understand all the discussions in this tutorial. Just keep in mind that $\mathcal{O}(1)$ is faster than $\mathcal{O}(n)$ is faster than $\mathcal{O}(n^2)$, etc.

Appendix B

Recursion and Induction

Informally, a function is recursive if its definition depends on itself. The prototypical example is factorial, whose definition is:

$$fact(n) = \begin{cases} 1 & n = 0 \\ n * fact(n-1) & n > 0 \end{cases}$$

Here, we can see that in order to calculate $fact(5)$, we need to calculate $fact(4)$, but in order to calculate $fact(4)$, we need to calculate $fact(3)$, and so on.

Recursive function definitions always contain a number of non-recursive base cases and a number of recursive cases. In the case of factorial, we have one of each. The base case is when $n = 0$ and the recursive case is when $n > 0$.

One can actually think of the natural numbers themselves as recursive (in fact, if you ask set theorists about this, they'll say this *is* how it is). That is, there is a zero element and then for every element, it has a successor. That is $1 = succ(0), 2 = succ(1), \ldots, 573 = succ(572), \ldots$ and so on forever. We can actually implement this system of natural numbers in Haskell:

```
data Nat = Zero | Succ Nat
```

This is a recursive type definition. Here, we represent one as `Succ Zero` and three as `Succ (Succ (Succ Zero))`. One thing we might want to do is be able to convert back and forth beween `Nat`s and `Int`s. Clearly, we can write a base case as:

```
natToInt Zero = 0
```

In order to write the recursive case, we realize that we're going to have something of the form `Succ n`. We can make the assumption that we'll be able to take `n` and produce an `Int`. Assuming we can do this, all we need to do is add one to this result. This gives rise to our recursive case:

```
natToInt (Succ n) = natToInt n + 1
```

There is a close connection between recursion and mathematical induction. Induction is a proof technique which typically breaks problems down into base cases and "inductive" cases, very analogous to our analysis of recursion.

Let's say we want to prove the statement $n! \geq n$ for all $n \geq 0$. First we formulate a base case: namely, we wish to prove the statement when $n = 0$. When $n = 0$, $n! = 1$ by definition. Since $n! = 1 > 0 = n$, we get that $0! \geq 0$ as desired.

Now, suppose that $n > 0$. Then $n = k + 1$ for some value k. We now invoke the *inductive hypothesis* and claim that the statement holds for $n = k$. That is, we assume that $k! \geq k$. Now, we use k to formate the statement for our value of n. That is, $n! \geq n$ if and only iff $(k+1)! \geq (k+1)$. We now apply the definition of factorial and get $(k+1)! = (k+1) * k!$. Now, we know $k! \geq k$, so $(k+1) * k! \geq k + 1$ if and only if $k + 1 \geq 1$. But we know that $k \geq 0$, which means $k + 1 \geq 1$. Thus it is proven.

It may seem a bit counter-intuitive that we are assuming that the claim is true for k in our proof that it is true for n. You can think of it like this: we've proved the statement for the case when $n = 0$. Now, we know it's true for $n = 0$ so using this we use our inductive argument to show that it's true for $n = 1$. Now, we know that it is true for $n = 1$ so we reuse our inductive argument to show that it's true for $n = 2$. We can continue this argument as long as we want and then see that it's true for all n.

It's much like pushing down dominoes. You know that when you push down the first domino, it's going to knock over the second one. This, in turn will knock over the third, and so on. The base case is like pushing down the first domino, and the inductive case is like showing that pushing down domino k will cause the $k+1$st domino to fall.

In fact, we can use induction to prove that our `natToInt` function does the right thing. First we prove the base case: does `natToInt Zero` evaluate to 0? Yes, obviously it does. Now, we can assume that `natToInt n` evaluates to the correct value (this is the inductive hypothesis) and ask whether `natToInt (Succ n)` produces the correct value. Again, it is obvious that it does, by simply looking at the definition.

Let's consider a more complex example: addition of `Nat`s. We can write this concisely as:

```
addNat Zero m = m
addNat (Succ n) m = addNat n (Succ m)
```

Now, let's prove that this does the correct thing. First, as the base case, suppose the first argument is `Zero`. We know that $0 + m = m$ regardless of what m is; thus in the base case the algorithm does the correct thing. Now, suppose that `addNat n m` does the correct thing for all `m` and we want to show that `addNat (Succ n) m` does the correct thing. We know that $(n+1) + m = n + (m+1)$ and thus since `addNat n (Succ m)` does the correct thing (by the inductive hypothesis), our program is correct.

Appendix C

Solutions To Exercises

Solution 3.1

It binds more tightly; actually, function application binds more tightly than anything else. To see this, we can do something like:

```
Prelude> sqrt 3 * 3
5.19615
```

If multiplication bound more tightly, the result would have been 3.

Solution 3.2

Solution: `snd (fst ((1,’a’),"foo"))`. This is because first we want to take the first half the the tuple: `(1,’a’)` and then out of this we want to take the second half, yielding just `’a’`.

If you tried `fst (snd ((1,’a’),"foo"))` you will have gotten a type error. This is because the application of `snd` will leave you with `fst "foo"`. However, the string “foo” isn’t a tuple, so you cannot apply `fst` to it.

Solution 3.3

Solution: map Char.isLower ”aBCde”

Solution 3.4

Solution: length (filter Char.isLower ”aBCde”)

Solution 3.5

foldr max 0 [5,10,2,8,1]. You could also use foldl. The foldr case is easier to explain: we replace each cons with an application of max and the empty list with 0. Thus, the inner-most application will take the maximum of 0 and the last element of the list (if it exists). Then, the next-most inner application will return the maximum of whatever was the maximum before and the second-to-last element. This will continue on, carrying to current maximum all the way back to the beginning of the list.

In the foldl case, we can think of this as looking at each element in the list in order. We start off our "state" with 0. We pull off the first element and check to see if it's bigger than our current state. If it is, we replace our current state with that number and the continue. This happens for each element and thus eventually returns the maximal element.

Solution 3.6

fst (head (tail [(5,'b'),(1,'c'),(6,'a')]))

Solution 3.7

We can define a fibonacci function as:

```
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)
```

We could also write it using explicit **if** statements, like:

```
fib n =
  if n == 1 || n == 2
    then 1
    else fib (n-1) + fib (n-2)
```

Either is acceptable, but the first is perhaps more natural in Haskell.

Solution 3.8

We can define:

$$a * b = \begin{cases} a & b = 1 \\ a + a * (b-1) & \text{otherwise} \end{cases}$$

And then type out code:

```
mult a 1 = a
mult a b = a + mult a (b-1)
```

Note that it doesn't matter that of a and b we do the recursion on. We could just as well have defined it as:

```
mult 1 b = b
mult a b = b + mult (a-1) b
```

Solution 3.9

We can define `my_map` as:

```
my_map f [] = []
my_map f (x:xs) = f x : my_map f xs
```

Recall that the `my_map` function is supposed to apply a function `f` to every element in the list. In the case that the list is empty, there are no elements to apply the function to, so we just return the empty list.

In the case that the list is non-empty, it is an element `x` followed by a list `xs`. Assuming we've already properly applied `my_map` to `xs`, then all we're left to do is apply `f` to `x` and then stick the results together. This is exactly what the second line does.

Solution 3.10

The code below appears in `Numbers.hs`. The only tricky parts are the recursive calls in `getNums` and `showFactorials`.

```
module Main
    where

import IO

main = do
  nums <- getNums
  putStrLn ("The sum is " ++ show (sum nums))
  putStrLn ("The product is " ++ show (product nums))
  showFactorials nums

getNums = do
  putStrLn "Give me a number (or 0 to stop):"
  num <- getLine
  if read num == 0
    then return []
    else do rest <- getNums
            return ((read num :: Int):rest)

showFactorials []     = return ()
showFactorials (x:xs) = do
  putStrLn (show x ++ " factorial is " ++
            show (factorial x))
  showFactorials xs

factorial 1 = 1
factorial n = n * factorial (n-1)
```

The idea for `getNums` is just as spelled out in the hint. For `showFactorials`, we consider first the recursive call. Suppose we have a list of numbers, the first of which is `x`. First we print out the string showing the factorial. Then we print out the rest, hence the recursive call. But what should we do in the case of the empty list? Clearly we are done, so we don't need to do anything at all, so we simply `return ()`.

Note that this must be `return ()` instead of just `()` because if we simply wrote `showFactorials [] = ()` then this wouldn't be an IO action, as it needs to be. For more clarification on this, you should probably just keep reading the tutorial.

Solution 4.1

1. String or $[\mathsf{Char}]$

2. type error: lists are homogenous

3. $\mathbf{Num}\ \mathsf{a} \Rightarrow (\mathsf{a}, \mathsf{Char})$

4. Int

5. type error: cannot add values of different types

Solution 4.2

The types:

1. $(\mathsf{a}, \mathsf{b}) - > \mathsf{b}$

2. $[\mathsf{a}] - > \mathsf{a}$

3. $[\mathsf{a}] - > \mathsf{Bool}$

4. $[\mathsf{a}] - > \mathsf{a}$

5. $[[\mathsf{a}]] - > \mathsf{a}$

Solution 4.3

The types:

1. $\mathsf{a} - > [\mathsf{a}]$. This function takes an element and returns the list containing only that element.

2. $\mathsf{a} - > \mathsf{b} - > \mathsf{b} - > (\mathsf{a}, [\mathsf{b}])$. The second and third argument must be of the same type, since they go into the same list. The first element can be of any type.

3. $\mathbf{Num}\ \mathsf{a} => \mathsf{a} - > \mathsf{a}$. Since we apply `(+)` to a, it must be an instance of $\mathbf{Num}$.

4. $\mathsf{a} - > \mathsf{String}$. This ignores the first argument, so it can be any type.

5. $(\mathsf{Char} - > \mathsf{a}) - > \mathsf{a}$. In this expression, `x` must be a function which takes a Char as an argument. We don't know anything about what it produces, though, so we call it a.

6. Type error. Here, we assume `x` has type `a`. But `x` is applied to itself, so it must have type $b->c$. But then it must have type $(b->c)->c$, but then it must have type $((b->c)->c)->c$ and so on, leading to an infinite type.

7. **Num** $a => a -> a$. Again, since we apply `(+)`, this must be an instance of **Num**.

Solution 4.4

The definitions will be something like:

```
data Triple a b c = Triple a b c

tripleFst (Triple x y z) = x
tripleSnd (Triple x y z) = y
tripleThr (Triple x y z) = z
```

Solution 4.5

The code, with type signatures, is:

```
data Quadruple a b = Quadruple a a b b

firstTwo :: Quadruple a b -> [a]
firstTwo (Quadruple x y z t) = [x,y]

lastTwo :: Quadruple a b -> [b]
lastTwo (Quadruple x y z t) = [z,t]
```

We note here that there are only two type variables, `a` and `b` associated with `Quadruple`.

Solution 4.6

The code:

```
data Tuple a b c d e = One a
                     | Two a b
                     | Three a b c
                     | Four a b c d

tuple1 (One   a      ) = Just a
tuple1 (Two   a b    ) = Just a
tuple1 (Three a b c  ) = Just a
tuple1 (Four  a b c d) = Just a

tuple2 (One   a      ) = Nothing
tuple2 (Two   a b    ) = Just b
tuple2 (Three a b c  ) = Just b
```

```
tuple2 (Four  a b c d) = Just b

tuple3 (One   a      ) = Nothing
tuple3 (Two   a b    ) = Nothing
tuple3 (Three a b c  ) = Just c
tuple3 (Four  a b c d) = Just c

tuple4 (One   a      ) = Nothing
tuple4 (Two   a b    ) = Nothing
tuple4 (Three a b c  ) = Nothing
tuple4 (Four  a b c d) = Just d
```

Solution 4.7

The code:

```
fromTuple :: Tuple a b c d -> Either (Either a (a,b)) (Either (a,b,c) (a,b,
fromTuple (One   a      ) = Left  (Left  a        )
fromTuple (Two   a b    ) = Left  (Right (a,b)    )
fromTuple (Three a b c  ) = Right (Left  (a,b,c)  )
fromTuple (Four  a b c d) = Right (Right (a,b,c,d))
```

Here, we use embedded `Either`s to represent the fact that there are four (instead of two) options.

Solution 4.8

The code:

```
listHead (Cons x xs) = x
listTail (Cons x xs) = xs

listFoldl f y Nil = y
listFoldl f y (Cons x xs) = listFoldl f (f y x) xs

listFoldr f y Nil = y
listFoldr f y (Cons x xs) = f x (listFoldr f y xs)
```

Solution 4.9

The code:

```
elements (Leaf x) = [x]
elements (Branch lhs x rhs) =
  elements lhs ++ [x] ++ elements rhs
```

Solution 4.10

The code:

```
foldTree :: (a -> b -> b) -> b -> BinaryTree a -> b
foldTree f z (Leaf x) = f x z
foldTree f z (Branch lhs x rhs) =
    foldTree f (f x (foldTree f z rhs)) lhs

elements2 = foldTree (:) []
```

or:

```
elements2 tree = foldTree (\a b -> a:b) [] tree
```

The first `elements2` is simply a more compact version of the second.

Solution 4.11

It mimicks neither exactly. It's behavior most closely resembles `foldr`, but differs slightly in its treatment of the initial value. We can observe the difference in an interpreter:

```
CPS> foldr (-) 0 [1,2,3]
2
CPS> foldl (-) 0 [1,2,3]
-6
CPS> fold  (-) 0 [1,2,3]
-2
```

Clearly it behaves differently. By writing down the derivations of `fold` and `foldr` we can see exactly where they diverge:

```
     foldr (-) 0 [1,2,3]
==>  1 - foldr (-) 0 [2,3]
==>  ...
==>  1 - (2 - (3 - foldr (-) 0 []))
==>  1 - (2 - (3 - 0))
==>  2

     fold  (-) 0 [1,2,3]
==>  fold' (-) (\y -> 0 - y) [1,2,3]
==>  0 - fold' (-) (\y -> 1 - y) [2,3]
==>  0 - (1 - fold' (-) (\y -> 2 - y) [3])
==>  0 - (1 - (2 - 3))
==>  -2
```

Essentially, the primary difference is that in the `foldr` case, the "initial value" is used at the end (replacing `[]`), whereas in the CPS case, the initial value is used at the beginning.

Solution 4.12

Solution 5.1

Using **if**, we get something like:

```
main = do
  putStrLn "Please enter your name:"
  name <- getLine
  if name == "Simon" || name == "John" || name == "Phil"
    then putStrLn "Haskell is great!"
    else if name == "Koen"
           then putStrLn "Debugging Haskell is fun!"
           else putStrLn "I don't know who you are."
```

Note that we don't need to repeat the **do**s inside the **if**s, since these are only one action commands.

We could also be a bit smarter and use the `elem` command which is built in to the Prelude:

```
main = do
  putStrLn "Please enter your name:"
  name <- getLine
  if name `elem` ["Simon", "John", "Phil"]
    then putStrLn "Haskell is great!"
    else if name == "Koen"
           then putStrLn "Debugging Haskell is fun!"
           else putStrLn "I don't know who you are."
```

Of course, we needn't put all the `putStrLn`s inside the **if** statements. We could instead write:

```
main = do
  putStrLn "Please enter your name:"
  name <- getLine
  putStrLn
    (if name `elem` ["Simon", "John", "Phil"]
       then "Haskell is great!"
       else if name == "Koen"
              then "Debugging Haskell is fun!"
              else "I don't know who you are.")
```

Using **case**, we get something like:

```
main = do
  putStrLn "Please enter your name:"
  name <- getLine
  case name of
    "Simon" -> putStrLn "Haskell is great!"
    "John"  -> putStrLn "Haskell is great!"
    "Phil"  -> putStrLn "Haskell is great!"
    "Koen"  -> putStrLn "Debugging Haskell is fun!"
    _       -> putStrLn "I don’t know who you are."
```

Which, in this case, is actually not much cleaner.

Solution 5.2

The code might look something like:

```
module DoFile where

import IO

main = do
  putStrLn "Do you want to [read] a file, ...?"
  cmd <- getLine
  case cmd of
    "quit"  -> return ()
    "read"  -> do doRead; main
    "write" -> do doWrite; main
    _       -> do putStrLn
                    ("I don’t understand the command "
                     ++ cmd ++ ".")
                  main

doRead = do
  putStrLn "Enter a file name to read:"
  fn <- getLine
  bracket (openFile fn ReadMode) hClose
          (\h -> do txt <- hGetContents h
                    putStrLn txt)

doWrite = do
  putStrLn "Enter a file name to write:"
  fn <- getLine
  bracket (openFile fn WriteMode) hClose
          (\h -> do putStrLn
                      "Enter text (...):"
                    writeLoop h)

writeLoop h = do
  l <- getLine
  if l == "."
    then return ()
    else do hPutStrLn h l
            writeLoop h
```

The only interesting things here are the calls to `bracket`, which ensure the that the program lives on, regardless of whether there's a failure or not; and the `writeLoop` function. Note that we need to pass the handle returned by `openFile` (through `bracket` to this function, so it knows where to write the input to).

Solution 7.1

Function `func3` cannot be converted into point-free style. The others look something like:

```
func1 x = map (*x)

func2 f g = filter f . map g

func4 = map (+2) . filter (‘elem‘ [1..10]) . (5:)

func5 = foldr (flip $ curry f) 0
```

You might have been tempted to try to write `func2` as `filter f . map`, trying to eta-reduce off the `g`. In this case, this isn't possible. This is because the function composition operator `(.)` has type $(b \to c) \to (a \to b) \to (a \to c)$. In this case, we're trying to use `map` as the second argument. But `map` takes two arguments, while `(.)` expects a function which takes only one.

Solution 7.2

We can start out with a recursive definition:

```
and [] = True
and (x:xs) = x && and xs
```

From here, we can clearly rewrite this as:

```
and = foldr (&&) True
```

Solution 7.3

We can write this recursively as:

```
concatMap f [] = []
concatMap f (x:xs) = f x ++ concatMap f xs
```

This hints that we can write this as:

```
concatMap f = foldr (\a b -> f a ++ b) []
```

Now, we can do point elimination to get:

```
     foldr (\a b -> f a ++ b) []
==>  foldr (\a b -> (++) (f a) b) []
==>  foldr (\a -> (++) (f a)) []
==>  foldr (\a -> ((++) . f) a) []
==>  foldr ((++) . f) []
```

Solution 9.1

The first law is: `return a >>= f` ≡ `f a`. In the case of Maybe, we get:

```
     return a >>= f
==>  Just a   >>= \x -> f x
==>  (\x -> f x) a
==>  f a
```

The second law is: `f >>= return` ≡ `f`. Here, we get:

```
     f >>= return
==>  f >>= \x -> return x
==>  f >>= \x -> Just x
```

At this point, there are two cases depending on whether `f` is `Nothing` or not. In the first case, we get:

```
==>  Nothing >>= \x -> Just x
==>  Nothing
==>  f
```

In the second case, `f` is `Just a`. Then, we get:

```
==>  Just a >>= \x -> Just x
==>  (\x -> Just x) a
==>  Just a
==>  f
```

And the second law is shown. The third law states: `f >>= (\x -> g x >>= h)` ≡ `(f >>= g) >>= h`.

If `f` is `Nothing`, then the left-hand-side clearly reduces to `Nothing`. The right-hand-side reduces to `Nothing >>= h` which in turn reduces to `Nothing`, so they are the same.

Suppose `f` is `Just a`. Then the LHS reduces to `g a >>= h` and the RHS reduces to `(Just a >>= \x -> g x) >>= h` which in turn reduces to `g a >>= h`, so these two are the same.

Solution 9.2

The idea is that we wish to use the `Left` constructor to represent errors on the `Right` constructor to represent successes. This leads to an instance declaration like:

```
instance Monad (Either String) where
    return x      = Right x
    Left  s >>= _ = Left s
    Right x >>= f = f x
    fail  s       = Left s
```

If we try to use this monad to do search, we get:

```
Monads> searchAll gr 0 3 :: Either String [Int]
Right [0,1,3]
Monads> searchAll gr 3 0 :: Either String [Int]
Left "no path"
```

which is exactly what we want.

Solution 9.3

The order to `mplus` essentially determins the search order. When the recursive call to `searchAll2` comes first, we are doing depth-first search. When the recursive call to `search'` comes first, we are doing breadth-first search. Thus, using the list monad, we expect the solutions to come in the other order:

```
MPlus> searchAll3 gr 0 3 :: [[Int]]
[[0,2,3],[0,1,3]]
```

Just as we expected.

Solution 9.4

This is a very difficult problem; if you found that you were stuck immediately, please just read as much of this solution as you need to try it yourself.

First, we need to define a list transformer monad. This looks like:

```
newtype ListT m e = ListT { unListT :: m [e] }
```

The ListT constructor simply wraps a monadic action (in monad m) which returns a list.

We now need to make this a monad:

```
instance Monad m => Monad (ListT m) where
    return x = ListT (return [x])
    fail   s = ListT (return [] )
    ListT m >>= k = ListT $ do
      l  <- m
      l' <- mapM (unListT . k) l
      return (concat l')
```

Here, success is designated by a monadic action which returns a singleton list. Failure (like in the standard list monad) is represented by an empty list: of course, it's actually an empty list returned from the enclosed monad. Binding happens essentially by running the action which will result in a list l. This has type [e]. We now need to apply k to each of these elements (which will result in something of type ListT m [e2]. We need to get rid of the ListTs around this (by using unListT) and then concatenate them to make a single list.

Now, we need to make it an instance of **MonadPlus**

```
instance Monad m => MonadPlus (ListT m) where
    mzero = ListT (return [])
    ListT m1 `mplus` ListT m2 = ListT $ do
      l1 <- m1
      l2 <- m2
      return (l1 ++ l2)
```

Here, the zero element is a monadic action which returns an empty list. Addition is done by executing both actions and then concatenating the results.

Finally, we need to make it an instance of **MonadTrans**:

```
instance MonadTrans ListT where
    lift x = ListT (do a <- x; return [a])
```

Lifting an action into ListT simply involves running it and getting the value (in this case, a) out and then returning the singleton list.

Once we have all this together, writing searchAll6 is fairly straightforward:

```
searchAll6 g@(Graph vl el) src dst
    | src == dst = do
      lift $ putStrLn $
```

```
        "Exploring " ++ show src ++ " -> " ++ show dst
      return [src]
    | otherwise  = do
      lift $ putStrLn $
        "Exploring " ++ show src ++ " -> " ++ show dst
      search’ el
  where
    search’ [] = mzero
    search’ ((u,v,_):es)
        | src == u  =
          (do path <- searchAll6 g v dst
              return (u:path)) ‘mplus‘
          search’ es
        | otherwise = search’ es
```

The only change (besides changing the recursive call to call `searchAll6` instead of `searchAll2`) here is that we call `putStrLn` with appropriate arguments, lifted into the monad.

If we look at the type of `searchAll6`, we see that the result (i.e., after applying a graph and two ints) has type `MonadTrans t, MonadPlus (t IO) => t IO [Int])`. In theory, we could use this with any appropriate monad transformer; in our case, we want to use `ListT`. Thus, we can run this by:

```
MTrans> unListT (searchAll6 gr 0 3)
Exploring 0 -> 3
Exploring 1 -> 3
Exploring 3 -> 3
Exploring 2 -> 3
Exploring 3 -> 3
MTrans> it
[[0,1,3],[0,2,3]]
```

This is precisely what we were looking for.

Solution 9.5

This exercise is actually simpler than the previous one. All we need to do is incorporate the calls to `putT` and `getT` into `searchAll6` and add an extra lift to the IO calls. This extra lift is required because now we’re stacking two transformers on top of IO instead of just one.

```
searchAll7 g@(Graph vl el) src dst
    | src == dst = do
      lift $ lift $ putStrLn $
        "Exploring " ++ show src ++ " -> " ++ show dst
      visited <- getT
      putT (src:visited)
      return [src]
    | otherwise  = do
      lift $ lift $ putStrLn $
        "Exploring " ++ show src ++ " -> " ++ show dst
      visited <- getT
      putT (src:visited)
      if src ‘elem‘ visited
        then mzero
        else search’ el
  where
    search’ [] = mzero
    search’ ((u,v,_):es)
        | src == u  =
          (do path <- searchAll7 g v dst
              return (u:path)) ‘mplus‘
          search’ es
        | otherwise = search’ es
```

The type of this has grown significantly. After applying the graph and two ints, this has type `Monad (t IO), MonadTrans t, MonadPlus (StateT [Int] (t IO)) => StateT [Int] (t IO) [Int]`.

Essentially this means that we’ve got something that’s a state transformer wrapped on top of some other arbitrary transformer (`t`) which itself sits on top of `IO`. In our case, `t` is going to be `ListT`. Thus, we run this beast by saying:

```
MTrans> unListT (evalStateT (searchAll7 gr4 0 3) [])
Exploring 0 -> 3
Exploring 1 -> 3
Exploring 3 -> 3
Exploring 0 -> 3
Exploring 2 -> 3
Exploring 3 -> 3
MTrans> it
[[0,1,3],[0,2,3]]
```

And it works, even on `gr4`.

Solution 9.6

First we write a function `spaces` which will parse out whitespaces:

```
spaces :: Parser ()
spaces = many (matchChar isSpace) >> return ()
```

Now, using this, we simply sprinkle calls to `spaces` through `intList` to get `intListSpace`:

```
intListSpace :: Parser [Int]
intListSpace = do
  char '['
  spaces
  intList' `mplus` (char ']' >> return [])
    where intList' = do
            i <- int
            spaces
            r <- (char ',' >> spaces >> intList')
                 `mplus`
                 (char ']' >> return [])
            return (i:r)
```

We can test that this works:

```
Parsing> runParser intListSpace "[1 ,2 , 4  \n\n ,5\n]"
Right ("",[1,2,4,5])
Parsing> runParser intListSpace "[1 ,2 , 4  \n\n ,a\n]"
Left "expecting char, got 'a'"
```

Solution 9.7

We do this by replacing the state functions with push and pop functions as follows:

```
parseValueLet2 :: CharParser (FiniteMap Char [Int]) Int
parseValueLet2 = choice
  [ int
  , do string "let "
       c <- letter
       char '='
       e <- parseValueLet2
       string " in "
       pushBinding c e
       v <- parseValueLet2
       popBinding c
       return v
  , do c  <- letter
       fm <- getState
       case lookupFM fm c of
         Nothing    -> unexpected ("variable " ++
                                   show c ++
                                   " unbound")
         Just (i:_) -> return i
  , between (char '(') (char ')') $ do
      e1 <- parseValueLet2
      op <- oneOf "+*"
      e2 <- parseValueLet2
      case op of
        '+' -> return (e1 + e2)
        '*' -> return (e1 * e2)
  ]
  where
    pushBinding c v = do
      fm <- getState
      case lookupFM fm c of
        Nothing -> setState (addToFM fm c [v])
        Just  l -> setState (addToFM fm c (v:l))
    popBinding c = do
      fm <- getState
      case lookupFM fm c of
        Just [_]   -> setState (delFromFM fm c)
        Just (_:l) -> setState (addToFM fm c l)
```

The primary difference here is that instead of calling `updateState`, we use two local functions, `pushBinding` and `popBinding`. The `pushBinding` function takes a variable name and a value and adds the value onto the head of the list pointed to in the state `FiniteMap`. The `popBinding` function looks at the value and if there is only one element on the stack, it completely removes the stack from the `FiniteMap`; otherwise it just removes the first element. This means that if something is in the `FiniteMap`, the stack is never empty.

This enables us to modify only slightly the usage case; this time, we simply take the top element off the stack when we need to inspect the value of a variable.

We can test that this works:

```
ParsecI> runParser parseValueLet2 emptyFM "stdin"
               "((let x=2 in 3+4)*x)"
Left "stdin" (line 1, column 20):
unexpected variable 'x' unbound
```

Index