5

HIGHER-ORDER FUNCTIONS

Haskell functions can take functions as parameters and return functions as return values. A function that does either of these things is called a *higher-order function*. Higher-order functions are a really powerful way of solving problems and thinking about programs, and they're indispensable when using a functional programming language like Haskell.

Curried Functions

Every function in Haskell officially takes only one parameter. But we have defined and used several functions that take more than one parameter so far—how is that possible?

Well, it's a clever trick! All the functions we've used so far that accepted multiple parameters have been *curried functions*. A curried function is a function that, instead of taking several parameters, always takes exactly one parameter.

Then when it's called with that parameter, it returns a function that takes the next parameter, and so on.

This is best explained with an example. Let's take our good friend, the `max` function. It looks as if it takes two parameters and returns the one that's bigger. For instance, consider the expression `max 4 5`. We call the function `max` with two parameters: `4` and `5`. First, `max` is applied to the value `4`. When we apply `max` to `4`, the value that is returned is actually another function, which is then applied to the value `5`. The act of applying this function to `5` finally returns a number value. As a consequence, the following two calls are equivalent:

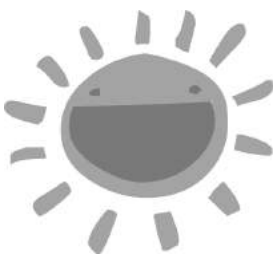
```
ghci> max 4 5
5
ghci> (max 4) 5
5
```

To understand how this works, let's examine the type of the `max` function:

```
ghci> :t max
max :: (Ord a) => a -> a -> a
```

This can also be written as follows:

```
max :: (Ord a) => a -> (a -> a)
```



Whenever we have a type signature that features the arrow `->`, that means it's a function that takes whatever is on the left side of the arrow and returns a value whose type is indicated on the right side of the arrow. When we have something like `a -> (a -> a)`, we're dealing with a function that takes a value of type `a`, and it returns a function that also takes a value of type `a` and returns a value of type `a`.

So how is that beneficial to us? Simply speaking, if we call a function with too few parameters, we get back a *partially applied* function, which is a function that takes as many parameters as we left out. For example, when we did `max 4`, we got back a function that takes one parameter. Using partial application (calling functions with too few parameters, if you will) is a neat way to create functions on the fly, so we can pass them to other functions.

Take a look at this simple little function:

```
multThree :: Int -> Int -> Int -> Int
multThree x y z = x * y * z
```

What really happens when we call `multThree 3 5 9`, or `((multThree 3) 5) 9`? First, `multThree` is applied to `3`, because they're separated by a space. That creates a function that takes one parameter and returns a function. Then that function is applied to `5`, which creates a function that will take one parameter, multiply 3 and 5 together, and then multiply that by the parameter. That function is applied to `9`, and the result is 135.

You can think of functions as tiny factories that take some materials and produce something. Using that analogy, we feed our `multThree` factory the number `3`, but instead of producing a number, it churns out a slightly smaller factory. That factory receives the number `5` and also spits out a factory. The third factory receives the number `9`, and then produces our resulting number, `135`.

Remember that this function's type can also be written as follows:

```
multThree :: Int -> (Int -> (Int -> Int))
```

The type (or type variable) before the `->` is the type of the values that a function takes, and the type after it is the type of the values it returns. So our function takes a value of type `Int` and returns a function of type `(Int -> (Int -> Int)`. Similarly, *this* function takes a value of type `Int` and returns a function of type `Int -> Int`. And finally, *this* function just takes a value of type `Int` and returns another value of type `Int`.

Let's look at an example of how we can create a new function by calling a function with too few parameters:

```
ghci> let multTwoWithNine = multThree 9
ghci> multTwoWithNine 2 3
54
```

In this example, the expression `multThree 9` results in a function that takes two parameters. We name that function `multTwoWithNine`, because `multThree 9` is a function that takes two parameters. If both parameters are supplied, it will multiply the two parameters between them, and then multiply that by `9`, because we got the `multTwoWithNine` function by applying `multThree` to `9`.

What if we wanted to create a function that takes an `Int` and compares it to `100`? We could do something like this:

```
compareWithHundred :: Int -> Ordering
compareWithHundred x = compare 100 x
```

As an example, let's try calling the function with `99`:

```
ghci> compareWithHundred 99
GT
```

100 is greater than 99, so the function returns GT, or greater than.

Now let's think about what compare 100 would return: a function that takes a number and compares it with 100, which is exactly what we were trying to get in our example. In other words, the following definition and the previous one are equivalent:

```
compareWithHundred :: Int -> Ordering
compareWithHundred = compare 100
```

The type declaration stays the same, because compare 100 returns a function. compare has a type of (Ord a) => a -> (a -> Ordering). When we apply it to 100, we get a function that takes a number and returns an Ordering.

Sections

Infix functions can also be partially applied by using *sections*. To section an infix function, simply surround it with parentheses and supply a parameter on only one side. That creates a function that takes one parameter and then applies it to the side that's missing an operand. Here's an insultingly trivial example:

```
divideByTen :: (Floating a) => a -> a
divideByTen = (/10)
```

As you can see in the following code, calling divideByTen 200 is equivalent to calling 200 / 10 or (/10) 200:

```
ghci> divideByTen 200
20.0
ghci> 200 / 10
20.0
ghci> (/10) 200
20.0
```

Let's look at another example. This function checks if a character supplied to it is an uppercase letter:

```
isUpperAlphanum :: Char -> Bool
isUpperAlphanum = (`elem` ['A'..'Z'])
```

The only thing to watch out for with sections is when you're using the - (negative or minus) operator. From the definition of sections, (-4) would result in a function that takes a number and subtracts 4 from it. However, for convenience, (-4) means negative four. So if you want to make a function that subtracts 4 from the number it gets as a parameter, you can partially apply the subtract function like so: (subtract 4).

Printing Functions

So far, we've bound our partially applied functions to names and then supplied the remaining parameters to view the results. However, we never tried to print the functions themselves to the terminal. Let's give that a go then, shall we? What happens if we try entering `multThree 3 4` into GHCi, instead of binding it to a name with a `let` or passing it to another function?

```
ghci> multThree 3 4
<interactive>:1:0:
    No instance for (Show (a -> a))
      arising from a use of `print' at <interactive>:1:0-12
    Possible fix: add an instance declaration for (Show (a -> a))
    In the expression: print it
    In a 'do' expression: print it
```

GHCi is telling us that the expression produced a function of type `a -> a`, but it doesn't know how to print it to the screen. Functions aren't instances of the `Show` type class, so we can't get a neat string representation of a function. This is different, for example, than when we enter `1 + 1` at the GHCi prompt. In that case, GHCi calculates `2` as the result, and then calls `show` on `2` to get a textual representation of that number. The textual representation of `2` is just the string `"2"`, which is then printed to the screen.

NOTE *Make sure you thoroughly understand how curried functions and partial application work, because they're really important!*

Some Higher-Orderism Is in Order

In Haskell, functions can take other functions as parameters, and as you've seen, they can also return functions as return values. To demonstrate this concept, let's write a function that takes a function, and then applies it twice to some value:

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

Notice the type declaration. For our earlier examples, we didn't need parentheses when declaring function types, because `->` is naturally right-associative. However, here parentheses are mandatory. They indicate that the first parameter is a function that takes one parameter and returns a value of the same type (`a -> a`). The second parameter is something of type `a`, and the return value's type is also `a`. Notice that it doesn't matter what type `a` is—it can be `Int`, `String`, or whatever—but all the values must be the same type.

NOTE *You now know that under the hood, functions that seem to take multiple parameters are actually taking a single parameter and returning a partially applied function. However, to keep things simple, I'll continue to say that a given function takes multiple parameters.*

The body of the `applyTwice` function is very simple. We just use the parameter `f` as a function, applying `x` to it by separating the `f` and `x` with a space. We then apply the result to `f` again. Here are some examples of the function in action:

```
ghci> applyTwice (+3) 10
16
ghci> applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"
ghci> applyTwice ("HAHA " ++) "HEY"
"HAHA HAHA HEY"
ghci> applyTwice (multThree 2 2) 9
144
ghci> applyTwice (3:) [1]
[3,3,1]
```

The awesomeness and usefulness of partial application is evident. If our function requires us to pass it a function that takes only one parameter, we can just partially apply a function to the point where it takes only one parameter and then pass it. For instance, the `+` function takes two parameters, and in this example, we partially applied it so that it takes only one parameter by using sections.

Implementing zipWith

Now we're going to use higher-order programming to implement a really useful function in the standard library called `zipWith`. It takes a function and two lists as parameters, and then joins the two lists by applying the function between corresponding elements. Here's how we'll implement it:

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

First let's look at the type declaration. The first parameter is a function that takes two arguments and returns one value. They don't have to be of the same type, but they can be. The second and third parameters are lists, and the final return value is also a list.

The first list must be a list of type `a` values, because the joining function takes `a` types as its first argument. The second must be a list of `b` types, because the second parameter of the joining function is of type `b`. The result is a list of type `c` elements.

NOTE *Remember that if you're writing a function (especially a higher-order function), and you're unsure of the type, you can try omitting the type declaration and checking what Haskell infers it to be by using* `:t`.

This function is similar to the normal `zip` function. The base cases are the same, although there's an extra argument (the joining function). However, that argument doesn't matter in the base cases, so we can just use the `_` character for it. The function body in the last pattern is also similar to `zip`, though instead of doing `(x, y)`, it does `f x y`.

Here's a little demonstration of all the different things our `zipWith'` function can do:

```
ghci> zipWith' (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]
ghci> zipWith' max [6,3,2,1] [7,3,1,5]
[7,3,2,5]
ghci> zipWith' (++) ["foo ", "bar ", "baz "] ["fighters", "hoppers", "aldrin"]
["foo fighters","bar hoppers","baz aldrin"]
ghci> zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]
ghci> zipWith' (zipWith' (*)) [[1,2,3],[3,5,6],[2,3,4]] [[3,2,2],[3,4,5],[5,4,3]]
[[3,4,6],[9,20,30],[10,12,12]]
```

As you can see, a single higher-order function can be used in very versatile ways.

Implementing flip

Now we'll implement another function in the standard library, called `flip`. The `flip` function takes a function and returns a function that is like our original function, but with the first two arguments flipped. We can implement it like this:

```
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = g
    where g x y = f y x
```

You can see from the type declaration that `flip'` takes a function that takes `a` and `b` types, and returns a function that takes `b` and `a` types. But because functions are curried by default, the second pair of parentheses actually is not necessary. The arrow `->` is right-associative by default, so `(a -> b -> c) -> (b -> a -> c)` is the same as `(a -> b -> c) -> (b -> (a -> c))`, which is the same as `(a -> b -> c) -> b -> a -> c`. We wrote that `g x y = f y x`. If that's true, then `f y x = g x y` must also hold, right? Keeping that in mind, we can define this function in an even simpler manner:

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f y x = f x y
```

In this new version of `flip'`, we take advantage of the fact that functions are curried. When we call `flip' f` without the parameters `y` and `x`, it will return an `f` that takes those two parameters but calls them flipped.

Even though flipped functions are usually passed to other functions, we can take advantage of currying when making higher-order functions by thinking ahead and writing what their end result would be if they were fully applied.

```
ghci> zip [1,2,3,4,5] "hello"
[(1,'h'),(2,'e'),(3,'l'),(4,'l'),(5,'o')]
ghci> flip' zip [1,2,3,4,5] "hello"
[('h',1),('e',2),('l',3),('l',4),('o',5)]
ghci> zipWith div [2,2..] [10,8,6,4,2]
[0,0,0,0,1]
ghci> zipWith (flip' div) [2,2..] [10,8,6,4,2]
[5,4,3,2,1]
```

If we `flip'` the `zip` function, we get a function that is like `zip`, except that the items from the first list are placed into the second components of the tuples and vice versa. The `flip' div` function takes its second parameter and divides that by its first, so when the numbers `2` and `10` are passed to `flip' div`, the result is the same as using `div 10 2`.

The Functional Programmer's Toolbox

As functional programmers, we seldom want to operate on just one value. We usually want to take a bunch of numbers, letters, or some other type of data, and transform the set to produce our results. In this section, we'll look at some useful functions that can help us work with multiple values.

The map Function

The `map` function takes a function and a list, and applies that function to every element in the list, producing a new list. Here is its definition:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

The type signature says that `map` takes a function from `a` to `b` and a list of `a` values, and returns a list of `b` values.

`map` is a versatile higher-order function that can be used in many different ways. Here it is in action:

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!") ["BIFF", "BANG", "POW"]
["BIFF!","BANG!","POW!"]
```

```
ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

You've probably noticed that each of these examples could also be achieved with a list comprehension. For instance, `map (+3) [1,5,3,1,6]` is technically the same as `[x+3 | x <- [1,5,3,1,6]]`. However, using the `map` function tends to make your code much more readable, especially once you start dealing with maps of maps.

The filter Function

The `filter` function takes a predicate and a list, and returns the list of elements that satisfy that predicate. (Remember that a *predicate* is a function that tells whether something is true or false; that is, a function that returns a Boolean value.) The type signature and implementation look like this:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

If `p x` evaluates to `True`, the element is included in the new list. If it doesn't evaluate to `True`, it isn't included in the new list.

Here are some `filter` examples:

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> let notNull x = not (null x) in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]
[[1,2,3],[3,4,5],[2,2]]
ghci> filter (`elem` ['a'..'z']) "u LaUgH aT mE BeCaUsE I aM diFfeRent"
"uagameasadifeent"
ghci> filter (`elem` ['A'..'Z']) "i LAuGh at you bEcause u R all the same"
"LAGER"
```

As with the `map` function, all of these examples could also be achieved by using comprehensions and predicates. There's no set rule for when to use `map` and `filter` versus using list comprehensions. You just need to decide what's more readable depending on the code and the context.

The `filter` equivalent of applying several predicates in a list comprehension is either filtering something several times or joining the predicates with the logical `&&` function. Here's an example:

```
ghci> filter (<15) (filter even [1..20])
[2,4,6,8,10,12,14]
```

In this example, we take the list `[1..20]` and filter it so that only even numbers remain. Then we pass that list to `filter (<15)` to get rid of numbers 15 and up. Here's the list comprehension version:

```
ghci> [x | x <- [1..20], x < 15, even x]
[2,4,6,8,10,12,14]
```

We use a list comprehension where we draw from the list `[1..20]`, and then say what conditions need to hold for a number to be in the resulting list.

Remember our `quicksort` function from Chapter 4? We used list comprehensions to filter out the list elements that were less than (or equal to) or greater than the pivot. We can achieve the same functionality in a more readable way by using `filter`:

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerOrEqual = filter (<= x) xs
        larger = filter (> x) xs
    in  quicksort smallerOrEqual ++ [x] ++ quicksort larger
```

More Examples of map and filter

As another example, let's find the largest number under 100,000 that's divisible by 3,829. To do that, we'll just filter a set of possibilities in which we know the solution lies:

```
largestDivisible :: Integer
largestDivisible = head (filter p [100000,99999..])
    where p x = x `mod` 3829 == 0
```

First, we make a descending list of all numbers less than 100,000. Then we filter it by our predicate. Because the numbers are sorted in a descending manner, the largest number that satisfies our predicate will be the first element of the filtered list. And because we end up using only the head of the filtered list, it doesn't matter if the filtered list is finite or infinite. Haskell's laziness causes the evaluation to stop when the first adequate solution is found.

As our next example, we'll find the sum of all odd squares that are smaller than 10,000. In our solution, we'll use the `takeWhile` function. This function takes a predicate and a list. Starting at the beginning of the list, it returns the list's elements as long as the predicate holds true. Once an element is found for which the predicate doesn't hold true, the function stops and returns the resulting list. For example, to get the first word of a string, we can do the following:

```
ghci> takeWhile (/=' ') "elephants know how to party"
"elephants"
```

To find the sum of all odd squares that are less than 10,000, we begin by mapping the `(^2)` function over the infinite list `[1..]`. Then we filter this list so we get only the odd elements. Next, using `takeWhile`, we take elements from that list only while they are smaller than 10,000. Finally, we get the sum of that list (using the `sum` function). We don't even need to define a function for this example, because we can do it all in one line in GHCi:

```
ghci> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
```

Awesome! We start with some initial data (the infinite list of all natural numbers), and then we map over it, filter it, and cut it until it suits our needs. Finally, we just sum it up!

We could have also written this example using list comprehensions, like this:

```
ghci> sum (takeWhile (<10000) [m | m <- [n^2 | n <- [1..]], odd m])
166650
```

For our next problem, we'll be dealing with Collatz sequences. A *Collatz sequence* (also known as a *Collatz chain*) is defined as follows:

- Start with any natural number.
- If the number is 1, stop.
- If the number is even, divide it by 2.
- If the number is odd, multiply it by 3 and add 1.
- Repeat the algorithm with the resulting number.

In essence, this gives us a chain of numbers. Mathematicians theorize that for all starting numbers, the chain will finish at the number 1. For example, if we start with the number 13, we get this sequence: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. ($13 \times 3 + 1$ equals 40. 40 divided by 2 equals 20, and so on.) We can see that the chain that starts with 13 has 10 terms.

Here is the problem we want to solve: For all starting numbers between 1 and 100, how many Collatz chains have a length greater than 15?

Our first step will be to write a function that produces a chain:

```
chain :: Integer -> [Integer]
chain 1 = [1]
chain n
    | even n =  n:chain (n `div` 2)
    | odd n  =  n:chain (n*3 + 1)
```

This is a pretty standard recursive function. The base case is one, because all our chains will end at one. We can test the function to see if it's working correctly:

```
ghci> chain 10
[10,5,16,8,4,2,1]
ghci> chain 1
[1]
ghci> chain 30
[30,15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1]
```

Now we can write the `numLongChains` function, which actually answers our question:

```
numLongChains :: Int
numLongChains = length (filter isLong (map chain [1..100]))
    where isLong xs = length xs > 15
```

We map the `chain` function to `[1..100]` to get a list of chains, which are themselves represented as lists. Then we filter them by a predicate that checks whether a list's length is longer than 15. Once we've done the filtering, we see how many chains are left in the resulting list.

NOTE *This function has a type of `numLongChains :: Int` because `length` returns an `Int` instead of a `Num a`. If we wanted to return a more general `Num a`, we could have used `fromIntegral` on the resulting length.*

Mapping Functions with Multiple Parameters

So far, we've mapped functions that take only one parameter (like `map (*2) [0..]`). However, we can also map functions that take multiple parameters. For example, we could do something like `map (*) [0..]`. In this case, the function `*`, which has a type of `(Num a) => a -> a -> a`, is applied to each number in the list.

As you've seen, giving only one parameter to a function that takes two parameters will cause it to return a function that takes one parameter. So if we map `*` to the list `[0..]`, we will get back a list of functions that take only one parameter.

Here's an example:

```
ghci> let listOfFuns = map (*) [0..]
ghci> (listOfFuns !! 4) 5
20
```

Getting the element with the index 4 from our list returns a function that's equivalent to (4*). Then we just apply 5 to that function, which is the same as (4*) 5, or just 4 * 5.

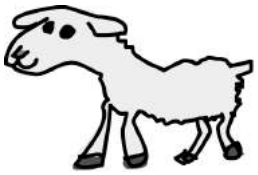
Lambdas

Lambdas are anonymous functions that we use when we need a function only once.

Normally, we make a lambda with the sole purpose of passing it to a higher-order function. To declare a lambda, we write a \ (because it kind of looks like the Greek letter lambda (λ) if you squint hard enough), and then we write the function's parameters, separated by spaces. After that comes a ->, and then the function body. We usually surround lambdas with parentheses.

In the previous section, we used a `where` binding in our `numLongChains` function to make the `isLong` function for the sole purpose of passing it to `filter`. Instead of doing that, we can also use a lambda, like this:

```
numLongChains :: Int
numLongChains = length (filter (\xs -> length xs > 15) (map chain [1..100]))
```



Lambdas are expressions, which is why we can just pass them to functions like this. The expression (\xs -> length xs > 15) returns a function that tells us whether the length of the list passed to it is greater than 15.

People who don't understand how currying and partial application work often use lambdas where they are not necessary. For instance, the following expressions are equivalent:

```
ghci> map (+3) [1,6,3,2]
[4,9,6,5]
ghci> map (\x -> x + 3) [1,6,3,2]
[4,9,6,5]
```

Both (+3) and (\x -> x + 3) are functions that take a number and add 3 to it, so these expressions yield the same results. However, we don't want to make a lambda in this case, because using partial application is much more readable.

Like normal functions, lambdas can take any number of parameters:

```
ghci> zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
[153.0,61.5,31.0,15.75,6.6]
```

And like normal functions, you can pattern match in lambdas. The only difference is that you can't define several patterns for one parameter (like making a [] and a (x:xs) pattern for the same parameter and then having values fall through).

```
ghci> map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
[3,8,9,8,7]
```

NOTE *If a pattern match fails in a lambda, a runtime error occurs, so be careful!*

Let's look at another interesting example:

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z

addThree :: Int -> Int -> Int -> Int
addThree' = \x -> \y -> \z -> x + y + z
```

Due to the way functions are curried by default, these two functions are equivalent. Yet the first addThree function is far more readable. The second one is little more than a gimmick to illustrate currying.

NOTE *Notice that in the second example, the lambdas are not surrounded with parentheses. When you write a lambda without parentheses, it assumes that everything to the right of the arrow -> belongs to it. So in this case, omitting the parentheses saves some typing. Of course, you can include the parentheses if you prefer them.*

However, there are times when using the currying notation instead is useful. I think that the flip function is the most readable when it's defined like this:

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f = \x y -> f y x
```

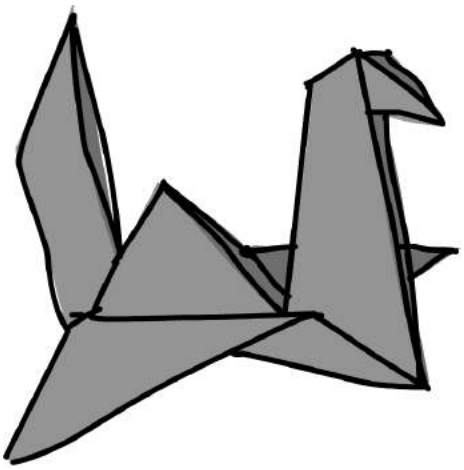
Even though this is the same as writing flip' f x y = f y x, our new notation makes it obvious that this will often be used for producing a new function.The most common use case with flip is calling it with just the function

parameter, or the function parameter and one extra parameter, and then passing the resulting function on to a `map` or a `zipWith`:

```
ghci> zipWith (flip (++)) ["love you", "love me"] ["i ", "you "]
["i love you","you love me"]
ghci> map (flip subtract 20) [1,2,3,4]
[19,18,17,16]
```

You can use lambdas this way in your own functions when you want to make it explicit that your functions are meant to be partially applied and then passed on to other functions as a parameter.

I Fold You So



Back when we were dealing with recursion in Chapter 4, many of the recursive functions that operated on lists followed the same pattern. We had a base case for the empty list, we introduced the `x:xs` pattern, and then we performed some action involving a single element and the rest of the list. It turns out this is a very common pattern, so the creators of Haskell introduced some useful functions, called *folds*, to encapsulate it. Folds allow you to reduce a data structure (like a list) to a single value.

Folds can be used to implement any function where you traverse a list once, element by element, and then return something based on that. Whenever you want to traverse a list to return something, chances are you want a fold.

A fold takes a *binary function* (one that takes two parameters, such as `+` or `div`), a starting value (often called the *accumulator*), and a list to fold up.

Lists can be folded up from the left or from the right. The fold function calls the given binary function, using the accumulator and the first (or last) element of the list as parameters. The resulting value is the new accumulator. Then the fold function calls the binary function again with the new accumulator and the new first (or last) element of the list, resulting in another new accumulator. This repeats until the function has traversed the entire list and reduced it down to a single accumulator value.

Left Folds with foldl

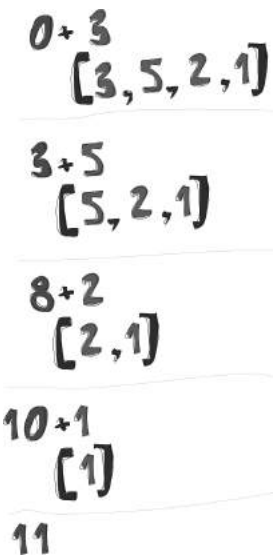
First, let's look at the `foldl` function. This is called a *left fold*, since it folds the list up from the left side. In this case, the binary function is applied between the starting accumulator and the head of the list. That produces a new accumulator value, and the binary function is called with that value and the next element, and so on.

Let's implement the `sum` function again, this time using a fold instead of explicit recursion:

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

Now we can test it:

```
ghci> sum' [3,5,2,1]
11
```



Let's take an in-depth look at how this fold happens. `\acc x -> acc + x` is the binary function. `0` is the starting value, and `xs` is the list to be folded up. First, `0` and `3` are passed to the binary function as the `acc` and `x` parameters, respectively. In this case, the binary function is simply an addition, so the two values are added, which produces `3` as the new accumulator value. Next, `3` and the next list value (`5`) are passed to the binary function, and they are added together to produce `8` as the new accumulator value. In the same way, `8` and `2` are added together to produce `10`, and then `10` and `1` are added together to produce the final value of `11`. Congratulations, you've folded your first list!

The diagram on the left illustrates how a fold happens, step by step. The number that's on the left side of the `+` is the accumulator value. You can see how the list is consumed up from the left side by the accumulator. (Om nom nom nom!) If we take into account that functions are curried, we can write this implementation even more succinctly, like so:

```
sum' :: (Num a) => [a] -> a
sum' = foldl (+) 0
```

The lambda function (`\acc x -> acc + x`) is the same as (`+`). We can omit the `xs` as the parameter because calling `foldl (+) 0` will return a function that takes a list. Generally, if you have a function like `foo a = bar b a`, you can rewrite it as `foo = bar b` because of currying.

Right Folds with foldr

The right fold function, `foldr`, is similar to the left fold, except the accumulator eats up the values from the right. Also, the order of the parameters in the right fold's binary function is reversed: The current list value is the first parameter, and the accumulator is the second. (It makes sense that the right fold has the accumulator on the right, since it folds from the right side.)

The accumulator value (and hence, the result) of a fold can be of any type. It can be a number, a Boolean, or even a new list. As an example, let's implement the `map` function with a right fold. The accumulator will be a list, and we'll be accumulating the mapped list element by element. Of course, our starting element will need to be an empty list:

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs
```

If we're mapping `(+3)` to `[1,2,3]`, we approach the list from the right side. We take the last element, which is `3`, and apply the function to it, which gives `6`. Then we prepend it to the accumulator, which was `[]`. `6:[]` is `[6]`, so that's now the accumulator. We then apply `(+3)` to `2`, yielding `5`, and prepend `(:)` that to the accumulator. Our new accumulator value is now `[5,6]`. We then apply `(+3)` to `1` and prepend the result to the accumulator again, giving a final result of `[4,5,6]`.

Of course, we could have implemented this function with a left fold instead, like this:

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldl (\acc x -> acc ++ [f x]) [] xs
```

However, the `++` function is much slower than `:`, so we usually use right folds when we're building up new lists from a list.

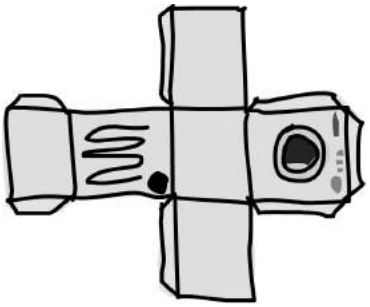
One big difference between the two types of folds is that right folds work on infinite lists, whereas left ones don't!

Let's implement one more function with a right fold. As you know, the `elem` function checks whether a value is part of a list. Here's how we can use `foldr` to implement it:

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldr (\x acc -> if x == y then True else acc) False ys
```

Here, the accumulator is a Boolean value. (Remember that the type of the accumulator value and the type of the end result are always the same when dealing with folds.) We start with a value of `False`, since we're assuming the value isn't in the list to begin with. This also gives us the correct value if we call it on the empty list, since calling a fold on an empty list just returns the starting value.

Next, we check if the current element is the element we want. If it is, we set the accumulator to `True`. If it's not, we just leave the accumulator unchanged. If it was `False` before, it stays that way because this current element is not the one we're seeking. If it was `True`, it stays that way as the rest of the list is folded up.



The foldl and foldr1 Functions

The `foldl1` and `foldr1` functions work much like `foldl` and `foldr`, except that you don't need to provide them with an explicit starting accumulator. They assume the first (or last) element of the list to be the starting accumulator, and then start the fold with the element next to it. With that in mind, the `maximum` function can be implemented like so:

```
maximum' :: (Ord a) => [a] -> a
maximum' = foldl1 max
```

We implemented `maximum` by using a `foldl1`. Instead of providing a starting accumulator, `foldl1` just assumes the first element as the starting accumulator and moves on to the second one. So all `foldl1` needs is a binary function and a list to fold up! We start at the beginning of the list and then compare each element with the accumulator. If it's greater than our accumulator, we keep it as the new accumulator; otherwise, we keep the old one. We passed `max` to `foldl1` as the binary function because it does exactly that: takes two values and returns the one that's larger. By the time we've finished folding our list, only the largest element remains.

Because they depend on the lists they're called with having at least one element, these functions cause runtime errors if called with empty lists. `foldl` and `foldr`, on the other hand, work fine with empty lists.

NOTE *When making a fold, think about how it acts on an empty list. If the function doesn't make sense when given an empty list, you can probably use a `foldl1` or `foldr1` to implement it.*

Some Fold Examples

To demonstrate how powerful folds are, let's implement some standard library functions using folds. First, we'll write our own version of `reverse`:

```
reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []
```

Here, we reverse a list by using the empty list as a starting accumulator and then approaching our original list from the left and placing the current element at the start of the accumulator.

The function `\acc x -> x : acc` is just like the `:` function, except that the parameters are flipped. That's why we could have also written `reverse'` like so:

```
reverse' :: [a] -> [a]
reverse' = foldl (flip (:)) []
```

Next, we'll implement `product`:

```
product' :: (Num a) => [a] -> a
product' = foldl (*) 1
```

To calculate the product of all the numbers in the list, we start with `1` as the accumulator. Then we fold left with the `*` function, multiplying each element with the accumulator.

Now we'll implement `filter`:

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc) []
```

Here, we use an empty list as the starting accumulator. Then we fold from the right and inspect each element. `p` is our predicate. If `p x` is `True`—meaning that if the predicate holds for the current element—we put it at the beginning of the accumulator. Otherwise, we just reuse our old accumulator.

Finally, we'll implement `last`:

```
last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

To get the last element of a list, we use a `foldl1`. We start at the first element of the list, and then use a binary function that disregards the accumulator and always sets the current element as the new accumulator. Once we've reached the end, the accumulator—that is, the last element—will be returned.

Another Way to Look at Folds

Another way to picture right and left folds is as successive applications of some function to elements in a list. Say we have a right fold, with a binary function `f` and a starting accumulator `z`. When we right fold over the list `[3,4,5,6]`, we're essentially doing this:

```
f 3 (f 4 (f 5 (f 6 z)))
```

`f` is called with the last element in the list and the accumulator, then that value is given as the accumulator to the next-to-last value, and so on.

If we take `f` to be `+` and the starting accumulator value to be `0`, we're doing this:

```
3 + (4 + (5 + (6 + 0)))
```

Or if we write `+` as a prefix function, we're doing this:

```
(+) 3 ((+) 4 ((+) 5 ((+) 6 0)))
```

Similarly, doing a left fold over that list with `g` as the binary function and `z` as the accumulator is the equivalent of this:

```
g (g (g (g z 3) 4) 5) 6
```

If we use `flip (:)` as the binary function and `[]` as the accumulator (so we're reversing the list), that's the equivalent of the following:

```
flip (:) (flip (:) (flip (:) (flip (:) [] 3) 4) 5) 6
```

And sure enough, if you evaluate that expression, you get `[6,5,4,3]`.

Folding Infinite Lists

Viewing folds as successive function applications on values of a list can give you insight as to why `foldr` sometimes works perfectly fine on infinite lists. Let's implement the `and` function with a `foldr`, and then write it out as a series of successive function applications, as we did with our previous examples. You'll see how `foldr` works with Haskell's laziness to operate on lists that have infinite length.

The `and` function takes a list of `Bool` values and returns `False` if one or more elements are `False`; otherwise, it returns `True`. We'll approach the list from the right and use `True` as the starting accumulator. We'll use `&&` as the binary function, because we want to end up with `True` only if all the elements are `True`. The `&&` function returns `False` if either of its parameters is `False`, so if we come across an element in the list that is `False`, the accumulator will be set as `False` and the final result will also be `False`, even if all the remaining elements are `True`:

```
and' :: [Bool] -> Bool
and' xs = foldr (&&) True xs
```

Knowing how `foldr` works, we see that the expression `and' [True,False,True]` will be evaluated like this:

```
True && (False && (True && True))
```

The last `True` represents our starting accumulator, whereas the first three `Bool` values are from the list `[True,False,True]`. If we try to evaluate the previous expression, we will get `False`.

Now what if we try this with an infinite list, say `repeat False`, which has an infinite number of elements, all of which are `False`? If we write that out, we get something like this:

```
False && (False && (False && (False ...
```

Haskell is lazy, so it will compute only what it really must. And the `&&` function works in such a way that if its first parameter is `False`, it disregards its second parameter, because the `&&` function returns `True` only if both of its parameters are `True`:

```
(&&) :: Bool -> Bool -> Bool
True && x = x
False && _ = False
```

In the case of the endless list of `False` values, the second pattern matches, and `False` is returned without Haskell needing to evaluate the rest of the infinite list:

```
ghci> and' (repeat False)
False
```

`foldr` will work on infinite lists when the binary function that we're passing to it doesn't always need to evaluate its second parameter to give us some sort of answer. For instance, `&&` doesn't care what its second parameter is if its first parameter is `False`.

Scans

The `scanl` and `scanr` functions are like `foldl` and `foldr`, except they report all the intermediate accumulator states in the form of a list. The `scanl1` and `scanr1` functions are analogous to `foldl1` and `foldr1`. Here are some examples of these functions in action:

```
ghci> scanl (+) 0 [3,5,2,1]
[0,3,8,10,11]
ghci> scanr (+) 0 [3,5,2,1]
[11,8,3,1,0]
ghci> scanl1 (\acc x -> if x > acc then x else acc) [3,4,5,3,7,9,2,1]
[3,4,5,5,7,9,9,9]
ghci> scanl (flip (:)) [] [3,2,1]
[[],[3],[2,3],[1,2,3]]
```

When using a `scanl`, the final result will be in the last element of the resulting list. `scanr` will place the result in the head of the list.

Scans are used to monitor the progress of a function that can be implemented as a fold. As an exercise in using scans, let's try answering this question: How many elements does it take for the sum of the square roots of all natural numbers to exceed 1,000?

To get the square roots of all natural numbers, we just call `map sqrt [1..]`. To get the sum, we could use a fold. However, because we're interested in how the sum progresses, we'll use a scan instead. Once we've done the scan, we can check how many sums are under 1,000.

```
sqrtSums :: Int
sqrtSums = length (takeWhile (<1000) (scanl1 (+) (map sqrt [1..]))) + 1
```

We use `takeWhile` here instead of `filter` because `filter` wouldn't cut off the resulting list once a number that's equal to or over 1,000 is found; it would keep searching. Even though we know the list is ascending, `filter` doesn't, so we use `takeWhile` to cut off the scan list at the first occurrence of a sum greater than 1,000.

The first sum in the scan list will be 1. The second will be 1 plus the square root of 2. The third will be that plus the square root of 3. If there are *x* sums under 1,000, then it takes *x*+1 elements for the sum to exceed 1,000:

```
ghci> sqrtSums
131
ghci> sum (map sqrt [1..131])
1005.0942035344083
ghci> sum (map sqrt [1..130])
993.6486803921487
```

And behold, our answer is correct! If we sum the first 130 square roots, the result is just below 1,000, but if we add another one to that, we go over our threshold.

Function Application with $

Now we'll look at the `$` function, also called the *function application operator*. First, let's see how it's defined:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

What the heck? What is this useless function? It's just function application! Well, that's almost true, but not quite. Whereas normal function application (putting a space between two things) has a really high precedence, the `$` function has the lowest precedence. Function application with a space is left-associative (so `f a b c` is the same as `((f a) b) c`), while function application with `$` is right-associative.

So how does this help us? Most of the time, it's a convenience function that lets us write fewer parentheses. For example, consider the expression `sum (map sqrt [1..130])`. Because `$` has such a low precedence, we can rewrite that expression as `sum $ map sqrt [1..130]`. When a `$` is encountered, the expression on its right is applied as the parameter to the function on its left.

How about `sqrt 3 + 4 + 9`? This adds together 9, 4, and the square root of 3. However, if we wanted the square root of 3 + 4 + 9, we would need to write `sqrt (3 + 4 + 9)`. With `$`, we can also write this as `sqrt $ 3 + 4 + 9`. You can imagine `$` as almost being the equivalent of writing an opening parenthesis and then writing a closing parenthesis on the far right side of the expression.

Let's look at another example:

```
ghci> sum (filter (> 10) (map (*2) [2..10]))
80
```

Whoa, that's a lot of parentheses! It looks kind of ugly. Here, `(*2)` is mapped onto `[2..10]`, then we filter the resulting list to keep only those numbers that are larger than `10`, and finally those numbers are added together.

We can use the `$` function to rewrite our previous example and make it a little easier on the eyes:

```
ghci> sum $ filter (> 10) (map (*2) [2..10])
80
```

The `$` function is right-associative, meaning that something like `f $ g $ x` is equivalent to `f $ (g $ x)`. With that in mind, the preceding example can once again be rewritten as follows:

```
ghci> sum $ filter (> 10) $ map (*2) [2..10]
80
```

Apart from getting rid of parentheses, `$` lets us treat function application like just another function. This allows us to, for instance, map function application over a list of functions, like this:

```
ghci> map ($ 3) [(4+), (10*), (^2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

Here, the function `($ 3)` gets mapped over the list. If you think about what the `($ 3)` function does, you'll see that it takes a function and then applies that function to `3`. So every function in the list gets applied to `3`, which is evident in the result.

Function Composition

In mathematics, *function composition* is defined like this: $(f \circ g)(x) = f(g(x))$. This means that composing two functions is the equivalent of calling one function with some value and then calling another function with the result of the first function.

In Haskell, function composition is pretty much the same thing. We do function composition with the `.` function, which is defined like this:

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

Notice the type declaration. `f` must take as its parameter a value that has the same type as `g`'s return value. So the resulting function takes a parameter of the same type that `g` takes and returns a value of the same type that `f` returns. For example, the expression `negate . (* 3)` returns a function that takes a number, multiplies it by 3, and then negates it.

One use for function composition is making functions on the fly to pass to other functions. Sure, we can use lambdas for that, but many times, function composition is clearer and more concise.

For example, say we have a list of numbers and we want to turn them all into negative numbers. One way to do that would be to get each number's absolute value and then negate it, like so:

```
ghci> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Notice the lambda and how it looks like the result of function composition. Using function composition, we can rewrite that as follows:

```
ghci> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Fabulous! Function composition is right-associative, so we can compose many functions at a time. The expression `f (g (z x))` is equivalent to `(f . g . z) x`. With that in mind, we can turn something messy, like this:

```
ghci> map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

into something much cleaner, like this:

```
ghci> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

`negate . sum . tail` is a function that takes a list, applies the `tail` function to it, then applies the `sum` function to the result of that, and finally applies `negate` to the previous result. So it's equivalent to the preceding lambda.

Function Composition with Multiple Parameters

But what about functions that take several parameters? Well, if we want to use them in function composition, we usually must partially apply them so that each function takes just one parameter. Consider this expression:

```
sum (replicate 5 (max 6.7 8.9))
```

This expression can be rewritten as follows:

```
(sum . replicate 5) max 6.7 8.9
```

which is equivalent to this:

```
sum . replicate 5 $ max 6.7 8.9
```

The function `replicate 5` is applied to the result of `max 6.7 8.9`, and then `sum` is applied to that result. Notice that we partially applied the `replicate` function to the point where it takes only one parameter, so that when the result of `max 6.7 8.9` gets passed to `replicate 5`, the result is a list of numbers, which is then passed to `sum`.

If we want to rewrite an expression with a lot of parentheses using function composition, we can start by first writing out the innermost function and its parameters. Then we put a `$` before it and compose all the functions that came before by writing them without their last parameter and putting dots between them. Say we have this expression:

```
replicate 2 (product (map (*3) (zipWith max [1,2] [4,5])))
```

We can write this as follows:

```
replicate 2 . product . map (*3) $ zipWith max [1,2] [4,5]
```

How did we turn the first example into the second one? Well, first we look at the function on the far right and its parameters, just before the bunch

of closing parentheses. That function is `zipWith max [1,2] [4,5]`. We're going to keep that as it is, so now we have this:

```
zipWith max [1,2] [4,5]
```

Then we look at which function was applied to `zipWith max [1,2] [4,5]` and see that it was `map (*3)`. So we put a `$` between it and what we had before:

```
map (*3) $ zipWith max [1,2] [4,5]
```

Now we start the compositions. We check which function was applied to all this, and we see that it was `product`, so we compose it with `map (*3)`:

```
product . map (*3) $ zipWith max [1,2] [4,5]
```

And finally, we see that the function `replicate 2` was applied to all this, and we can write the expression as follows:

```
replicate 2 . product . map (*3) $ zipWith max [1,2] [4,5]
```

If the expression ends with three parentheses, chances are that if you translate it into function composition by following this procedure, it will have two composition operators.

Point-Free Style

Another common use of function composition is defining functions in the *point-free style*. For example, consider a function we wrote earlier:

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (+) 0 xs
```

The `xs` is on the far right on both sides of the equal sign. Because of currying, we can omit the `xs` on both sides, since calling `foldl (+) 0` creates a function that takes a list. In this way, we are writing the function in point-free style:

```
sum' :: (Num a) => [a] -> a
sum' = foldl (+) 0
```

As another example, let's try writing the following function in point-free style:

```
fn x = ceiling (negate (tan (cos (max 50 x))))
```

We can't just get rid of the `x` on both right sides, since the `x` in the function body is surrounded by parentheses. `cos (max 50)` wouldn't make sense—

you can't get the cosine of a function. What we *can* do is express `fn` as a composition of functions, like this:

```
fn = ceiling . negate . tan . cos . max 50
```

Excellent! Many times, a point-free style is more readable and concise, because it makes you think about functions and what kinds of functions composing them results in, instead of thinking about data and how it's shuffled around. You can take simple functions and use composition as glue to form more complex functions.

However, if a function is too complex, writing it in point-free style can actually be less readable. For this reason, making long chains of function composition is discouraged. The preferred style is to use `let` bindings to give labels to intermediary results or to split the problem into subproblems that are easier for someone reading the code to understand.

Earlier in the chapter, we solved the problem of finding the sum of all odd squares that are smaller than 10,000. Here's what the solution looks like when put into a function:

```
oddSquareSum :: Integer
oddSquareSum = sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

With our knowledge of function composition, we can also write the function like this:

```
oddSquareSum :: Integer
oddSquareSum = sum . takeWhile (<10000) . filter odd $ map (^2) [1..]
```

It may seem a bit weird at first, but you will get used to this style quickly. There's less visual noise because we removed the parentheses. When reading this, you can just say that `filter odd` is applied to the result of `map (^2) [1..]`, then `takeWhile (<10000)` is applied to the result of that, and finally `sum` is applied to that result.