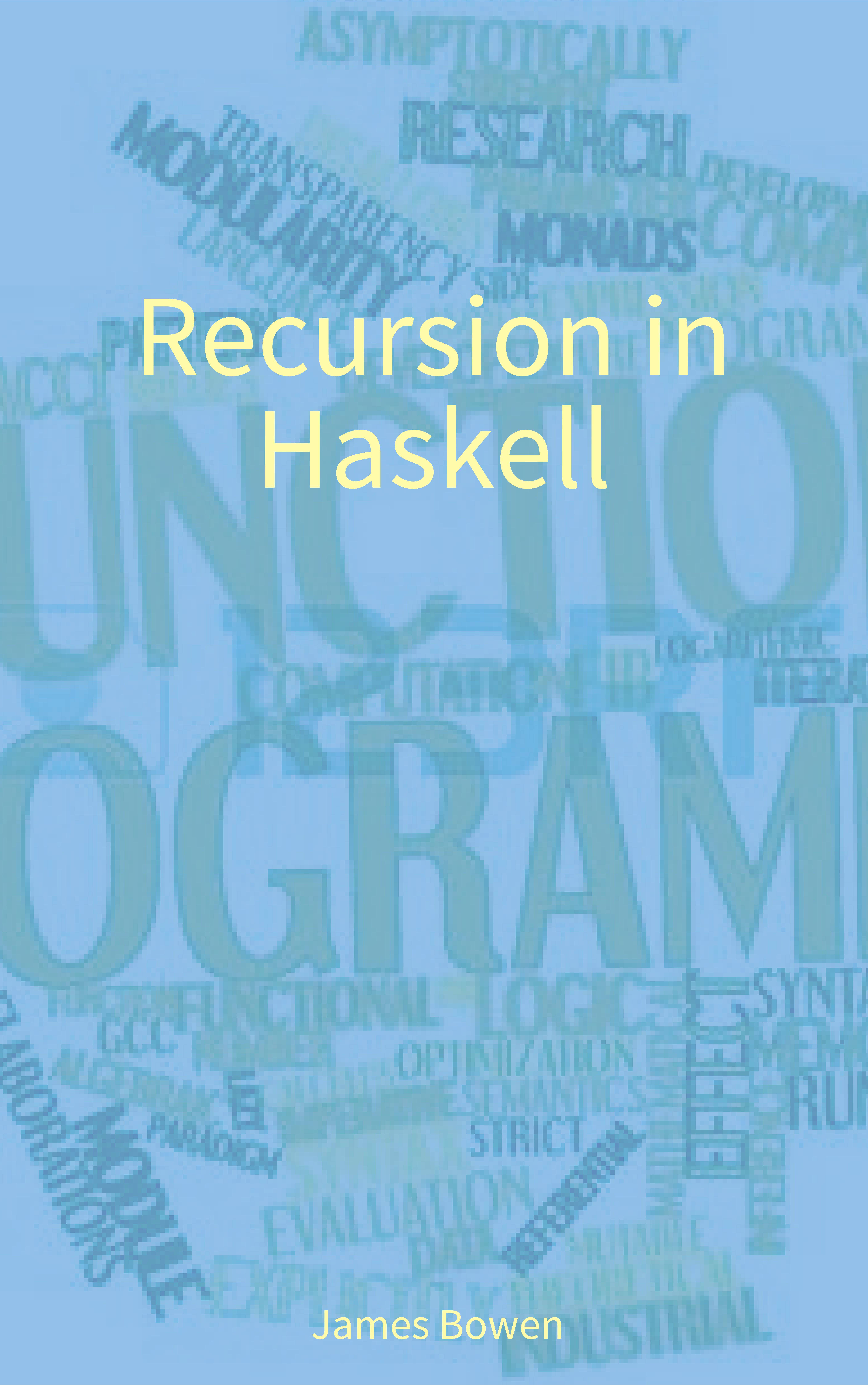
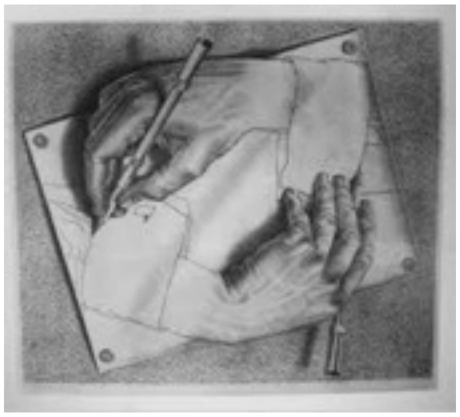


Recursion in Haskell

James Bowen

Table of Contents

Chapter 1: Recursion



When writing Haskell, you never use for-loops. Many of the tasks that require for-loops in other languages are done with recursion in Haskell (though this is often hidden under library functions). Recursion is a fundamental pattern which is key to being able to write Haskell code. This chapter will focus on recursion on lists.

Pattern Matching

Recursion consists of a function which has a simple base case and at least one case where the function calls itself, only with “reduced” arguments. The first thing to understand is how to cause a function to have different behavior for different inputs. There are multiple ways to do this, but we’ll start with the idea of pattern matching. Consider this starter code for a function, “sumInts”, which will take a list of integers and return their sum.

```
sumInts :: [Int] -> Int
sumInts [] = ...
sumInts (i : is) = ...
```

The first line declares the type of this function. The second line uses the “[]” pattern to say “this is what happens when the list parameter is an empty list.” The third line uses the concatenation pattern to say “this is what happens when the list consists of a single element appended to another list”. These two are enough to account for all possible lists the function could receive as input. The list is either empty, or it has a head element and a “remainder” of the list (which could be empty).

There are other ways to pattern a list, such as saying it has a particular set of values, or an exact number of elements. You can also have a “catch-all” pattern at the end.

```
sumInts :: [Int] -> Int
sumInts [1,2] = ...
sumInts a : b : [] = ...
sumInts [a,b,c,d] = ...
sumInts catchAll = ...
```

In this example, the first pattern matches only the exact list with the numbers 1 and 2. The second pattern uses double concatenation with the empty list to match any list with two elements. The third pattern matches any list with 4 elements. The last pattern will match any list. It is important to note how the names given in the pattern bind the elements to names. In the third pattern above, we pattern match on a list which has exactly 4 elements. The first element can then be referred to as a, the second as “b”, and so on. The last pattern matches the entire list as the name “catchAll”. We could perform list operations such as “head” or “tail” on “catchAll”.

We can also use an underscore when a pattern is a catchall but we don’t need the value. We can also assign a name to the entire list as well as its parts using the “@” symbol.

```
sumInts :: [Int] -> Int
sumInts myList@[a,b] = ...
sumInts _ = ...
```

Note that the possible cases are checked in order! Thus you need to put the most restrictive cases first. In this next example, the second pattern is never called, because the first catches everything!

```
sumInts :: [Int] -> Int
sumInts myList = ...
sumInts [1,2,3] = -- Never called!
```

There are also other ways to achieve the same effect as pattern matching in the original example. You could either use an if statement to check the length of the list, or you could use a case statement to check the form of the list. The case statement is more flexible:

```
sumInts :: [Int] -> Int
sumInts myList = if length myList == 0
  then ...
  else ...

sumInts :: [Int] -> Int
sumInts myList = case myList of
  [] -> ...
  [1,2] -> ...
  (a : as) -> ...
```

Implementing “sumInts”

So now let’s get into the specifics of implementing this sum function. We do this with the original 2 patterns of empty and nonempty. The empty list case is our base case. This is an easy case! We know the sum of an empty list is 0.

```
sumInts :: [Int] -> Int
sumInts [] = 0
sumInts (i : is) = ...
```

Now for the other case, we simply consider how the current element “i” relates to the solution on the smaller list, “is”. We know that the sum of the whole list is just the current element added to the sum of the rest of the list. This is where recursion comes in. We can call the “sumInts” function from within “sumInts”! Since the new list is smaller, it will eventually terminate. Here’s what the code looks like:

```
sumInts :: [Int] -> Int
sumInts [] = 0
sumInts (i : is) = i + sumInts is
```

And we’re done! This code will correctly calculate the sum of a list of ints! Let’s review the basic process of recursion:

1. Determine the base case (what’s the simplest case we eventually break down to?)
2. How can we separate the current step from the remainder?
3. Recurse on the remainder.
4. Combine the current step with the recursive solution.

Having a base case is key! If you don’t have any base cases, you’ll find yourself in an infinite loop!

Implementing “map”

For another example of basic recursion, let’s look at how the “map” library function works. This function takes a list of inputs, a function converting those inputs, and returns a list of the outputs:

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

This is eerily similar to our “sumInts” function from above. The only real difference is the combination step. We apply the function to the first element of the list, recurse over the remainder of the list, and then combine the two with concatenation.

Tail Recursion

The last concept we will talk about in this chapter is tail recursion. It turns out the way we wrote “sumInts” earlier is potentially problematic. Each number will all be stored in memory as Haskell waits for each of the recursive calls to finish. If we have a lot of numbers, we could run out of memory and our program would crash!

However, by cleverly rewriting our function to use an accumulator argument, we can avoid this. We will use a helper function within sumInts which takes two parameters: the sum of the elements we’ve already seen, and the remaining items in the list. Then we simply call this helper with the initial sum (0) and the full list:

```
sumInts :: [a] -> a
sumInts myList = sumIntsTail 0 myList
  where
    sumIntsTail accumulatedSum [] = accumulatedSum
    sumIntsTail accumulatedSum (a : as) =
      sumIntsTail (a + accumulatedSum) as
```

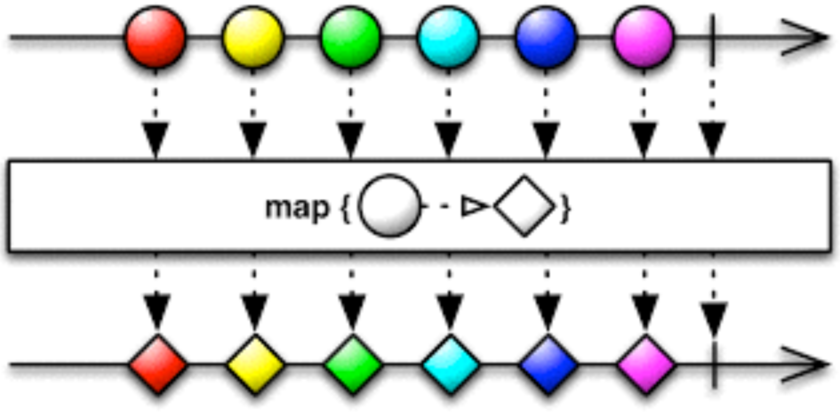
The goal with tail recursion is to make the recursive call the last thing the recursive function does. In the earlier example, the call to “sumInts” was not the last thing. Once the call returned, we would need to add its result to our current value. This forces us to store the value until the recursive call finishes, but tail recursion does not.

Summary

So here’s the 5 step summary for solving recursive problems:

1. Determine the base case
2. Break the input into a current step and a remainder.
3. Recurse on the remainder.
4. Combine the current step with the recursive solution.
5. Try to accumulate your current steps along the way so that you can make the recursive call the last thing that happens in a method (tail recursion).

Chapter 2: Higher Order Functions



Now that we know about recursion, let’s talk about some helper functions which use recursion under the hood. Many of these are higher order functions, meaning they take other functions as input. We’ll see how this works in practice. These functions are super useful in manipulating any kind of list data. They can make your code a lot cleaner.

Map

The first example we're going to talk about is “map”, whose implementation we showed in the last chapter. Map is a simple function. It takes as its first argument a function between two data types (they could be the same). It also takes a list of the first type in this function. It will then apply the function to each element in the input list, and return the result as a list. Let ‘s take a look at a use case of map. Suppose we have a list of integers and we want to double all of them. The expression (2 *) is a function which takes an int and returns an int. Thus we can map it over a list of ints to double them all:

```
>> let a = [2, 5, 7, 8]
>> map (2 *) a
[4, 10, 14, 16]
```

Let’s look at another example using a function between different data types. Suppose we wanted to return a list of strings, each with the first n letters of the alphabet based on n, an integer. Since the list “[a..z]” describes the list of all the lowercase letters, the expression “(\n -> take n [a..z])” is a function from an integer to a list of characters. So we can pass it using map:

```
Let a = [2, 5, 7, 8]
>> map (\n -> take n [a..z]) a
[“ab”, “abcde”, “abcdefg”, “abcdefgh”]
```

Filter

The second function we're going to talk about is filter. Filter takes a predicate as its function argument. A predicate is a function from a particular type to a boolean, effectively telling us if the element passes some test. Filter then also takes a list of the type that the predicate evaluates, and then returns a new list containing only the original items which returned true when passed into the predicate.

Let's take a look at a couple examples. First, consider the problem of taking all the even numbers from a list. We can make a function from an integer to a boolean with the expression (\x -> x `mod` 2 == 0). Let's apply this as a filter:

```
>> let a = [2, 5, 7, 8]
>> filter (\x -> x `mod` 2 == 0) a
[2, 8]
```

We could also filter strings by their length. Suppose we wanted to take only strings longer than 5. The predicate function would be (\str -> length str > 5).

```
>> let a = [“hello”, “are you”, “there”, “today?”, “I”, “am”]
>> filter (\str -> length str > 5) a
[“are you”, “today?”]
```

Fold

A fold takes a series of elements and combines them in some way. There are 3 arguments to any fold. We'll look specifically at “foldl”, but there are two other common folding functions “**foldr**” and “**foldl'**” which do the same fundamental action.

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

The first argument is a combining function. The type “a” is the resulting type of the fold. The type “b” is the input type of the fold. Thus the combining function takes the accumulated “a” type, combines it with another “b” element, consuming it, and then returns a new “a” type. We'll ultimately return the final “a” value.

The second argument is the initial element of type “a”. This is our “starting” value. It will be returned if the list is empty. Otherwise it will be combined with the first element from the list in the combining function on the first step. The third argument is the list of “b” elements that we want to fold. Many simple recursion examples like “sumInts” from the previous chapter can be easily described with a fold.

```
sumInts :: [Int] -> Int
sumInts myList = foldl (+) 0 myList
```

We ultimately want an Int as the result, and we consume ints one by one. The combining function is simply addition. The initial argument is just 0, and then we apply this over our list. If the list is initially empty, we'll return 0. If there is a first element, this will get added to 0. Then subsequent elements will get added to the result.

Let's try a more complicated example. Suppose we want our result to be a tuple of two Ints. The first of this pair is the product of all the ints. The second is the sum of the even numbers. Here's how we would do it with a fold.

```
combineFunction :: (Int, Int) -> Int -> (Int, Int)
combineFunction (sum, evenProduct) newNumber =
  (newFirst, newSecond)
  where
    newFirst = sum + newNumber
    newSecond = if newNumber `mod` 2 == 0
      then eventProduct * newNumber
      else evenProduct
```

```
findTupleResult :: [Int] -> (Int, Int)
findTupleResult myList = foldl combineFunction (0, 1) myList
```

Zip

Zip takes two lists, and matches up their elements. Let's first look at its type signature:

```
zip :: [a] -> [b] -> [(a,b)]
```

A simple example would be if you had a list of names of people and a list of phone numbers, and you know that the names match up exactly with the entry of the same index in the numbers list. You could combine these to get a list of tuples of names and numbers.

```
>> let names = ["John", "Jane", "Jack"]
>> let numbers = ["888-555-0001","888-555-0002","888-555-0003"]
>> zip names numbers
[("John", "888-555-0001"),
 ("Jane", "888-555-0002"),
 ("Jack", "888-555-0003")]
```

Note that if either list is shorter, then the trailing elements of the longer list are ignored.

```
>> let a = [1,2,3]
>> let b = ["hi", "bye"]
>> zip a b
[(1, "hi"), (2, "bye")]
```

Zip has a corresponding function, unzip, which does exactly the opposite, giving you two lists instead of a list of tuples:

```
>> let a = [(1,2), (3,4)]
>> unzip a
([1,3], [2,4])
```

ZipWith

The zipWith function is similar to zip, but it allows you more control over how the elements in the lists are combined. Its type is:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Suppose we want to combine two lists of integers. Instead of making them into tuples, we could make a new list of their products:

```
>> let a = [1,2,3]
>> let b = [3,4,5]
>> zipWith (*) a b
[3, 8, 15]
```

You can get more creative when you try it with different types:

```
combiningFunction :: Int -> String -> String
combiningFunction n str = if n < 5 && length str < 5
  then “Small number and string!”
  else if n < 5
    then “Small number, but long string!”
    else if length str < 5
      Then “Small string, big number!”
      else “Big number and string!”

-- Results in
-- [ “Small number and string!”
-- , “Big number and string!”
-- , “Small number, but long string!”
-- , “Small string, big number!”]
coolResult :: [String]
coolResult = zipWith
  [1, 6, 4, 5]
  [“Hi”, “Hello”, “Goodbye”, “What”]
```

Concat

The concat function is a simple function for turning a list of lists into a single list. As long as the lists are all the same type, you can combine them, even if they're different lengths, or even empty!

```
>> let a =
  [[“Hi”],
   [“Bye”, “What”],
   [“Yo”, “Greetings!”, “Cheers!”],
   []]
>> concat a
[“Hi”, “Bye”, “What”, “Yo”, “Greetings!”, “Cheers!”]
```

Note that concat and map can be combined with concatMap! You pass a function which generates a list of the target type, and then concatMap automatically flattens this out to a single list.

```
>> :t concatMap
concatMap :: (a -> [b]) -> [a] -> [b]
>> let f a = [a+1, a+2, a+3]
>> concatMap f [3,6,9]
[4,5,6,7,8,9,10,11,12]
```

Summary

In this chapter we learned 6 possible library functions and concepts:

1. Map
2. Filter
3. Fold
4. Zip
5. ZipWith
6. Concat

Between these and what you learned in the recursion chapter, you should now have all the skills you need to solve the following problems! Go to it!

Problems

Fill in the missing definitions in src/WorkbookQuestions.hs, then run the “stack test” command to check your answers!

Problem 1 (evens)

Implement a function which takes a list and returns a new list containing all the even indexed elements of the original list (the first element of a list is “odd” indexed).

Problem 2 (addWhenMod3Is2)

Implement a function which takes a list of integers, and return a list containing all the original number which are equal to 2 modulo 3, except add 3 to each of them.

Problem 3 (reverse_)

Implement a function which reverses a normal list. For this problem do not worry about the ill effects of using the (++) operator.

Problem 4 (reverseAccum)

Implement reverse, but this time use a helper with an accumulator argument to avoid using (++).

Problem 5 (specialMultiples)

Write a function which takes a list of integers, and returns a new list containing twice, three times, and four times every element of the original list.

Problem 6 (manyStrings)

Given a list of integers, and a list of strings, return a list of strings where each string from the original list is repeated the number of times of the integer in the corresponding index in the integer list. If either list is longer, ignore excess elements.

Problem 7 (addPairs)

Given a list of integers, return a new list with the sum of each successive pair of numbers. If there is an odd number of elements, IGNORE the last number.

Problem 8 (listToMap)

Take a list, and return a map containing the elements of that list, but mapped by the string version of the elements.

Problem 9 (sumWithParity)

Given a list of integers, return the sum of twice each element at an even index (counting from 1), and three times each element at an odd index.

Problem 10 (jumpingStairs)

This function takes two arguments. The first is a list of integers representing the heights of successive jumps. The second is a list of tuples representing steps, each with a name, and an integer representing how high it is ABOVE the previous step (not absolute height). Return a tuple partitioning the step names into those which you are able to climb given the series of jumps. One jump can be used to climb multiple steps, but you can't otherwise "save" energy from past jumps to get over higher steps.

Problem Solutions

Note: The best form of practice is doing something yourself! Don't look at the answers until you've attempted to solve them! Note that in several answers I have used eta reduction to simplify the code a bit. Don't let the (seeming) lack of arguments throw you off!

Problem 1 (evens)

```
evens :: [a] -> [a]
evens [] = []
evens [_] = []
evens (_ : e2 : rest) = e2 : evens rest
```

Problem 2 (addWhenMod3Is2)

```
addWhenMod3Is2 :: [Int] -> [Int]
addWhenMod3Is2 ints = map (3 +) filteredInts
  where
    filteredInts = filter (\i -> i `mod` 3 == 2) ints
```

Problem 3 (reverse_)

```
reverse_ :: [a] -> [a]
reverse_ [] = []
reverse_ (e : es) = reverse_ es ++ [e]
```

Problem 4 (reverseAccum)

```
-- Watch out, eta reduction!
reverseAccum :: [a] -> [a]
reverseAccum = reverseAccumTail []
  where
    reverseAccumTail accum [] = accum
    reverseAccumTail accum (e : es) =
      reverseAccumTail (e : accum) es

-- Here's a cleaner but less intuitive version
-- using the built in (:) operator.
reverseAccum :: [a] -> [a]
reverseAccum = reverseAccumTail []
  where
    reverseAccumTail :: [a] -> [a] -> [a]
    reverseAccumTail = foldl (flip (:))
```

Problem 5 (specialMultiples)

```
-- Note: eta reduction
specialMultiples :: [Int] -> [Int]
specialMultiples = concatMap (\i -> [2 * i, 3 * i, 4 * i])

-- Bonus: Using list comprehension
specialMultiples :: [Int] -> [Int]
specialMultiples ints =  [i*x | i <- ints, x <- [2,3,4]]
```

Problem 6 (manyStrings)

```
manyStrings :: [Int] -> [String] -> [String]
manyStrings ints strings = concat $
  zipWith replicate ints strings
```

Problem 7 (addPairs)

```
addPairs :: [Int] -> [Int]
addPairs (a : b : rest) = a + b : addPairs rest
addPairs _ = []


-- Here's a version combining evens with zipWith
addPairs :: [Int] -> [Int]
addPairs ints = zipWith (+) es odds
  where
    es = evens ints
    odds = evens (0 : ints)
```

Problem 8 (listToMap)

```
-- Note: eta reduction
listToMap :: [Int] -> M.Map String Int
listToMap = foldl (\m i -> M.insert (show i) i m) M.empty
```

Problem 9 (sumWithParity)

```
-- Note: eta reduction
sumWithParity :: [Int] -> Int
sumWithParity = sumWithParityTail True 0
  where
    sumWithParityTail _ accum [] = accum
    sumWithParityTail isOdd accum (i : is) = if isOdd
      then sumWithParityTail False (3 * i + accum) is
      else sumWithParityTail True (2 * i + accum) is
```

Problem 10 (jumpingStairs)

```
-- Note: eta reduction
jumpingStairs :: [Int] -> [(String, Int)] -> ([String],
[String])
jumpingStairs ints = jumpingStairsTail ints []
  where
    -- No more jumps case
    jumpingStairsTail [] accStairs remStairs =
      (reverse accStairs, map fst remStairs)
    -- No more remaining stairs case
    jumpingStairsTail _ accStairs [] =
      (reverse accStairs, [])
    -- Normal case (at least one jump, at least one stair)
    jumpingStairsTail
      (j : js)
      accStairs
      remStairs@((sName, sHeight) : rest) =
        if j >= sHeight
          -- Case where we can jump the stair
          then jumpingStairsTail
            (j - sHeight : js)
            (sName : accStairs)
            rest
          -- Case where we cannot jump the step
          else jumpingStairsTail js accStairs remStairs
```