Advanced Statistical Programming in R

Functional Programming in R

Thomas Mailund

Table of Contents

Table of Contents **3**

1 Introduction **5**
About the series 5
About this book 6

2 Functions in R **9**
Writing functions in R 9
Lazy evaluation 18
Vectorised functions 26
Infix operators 31
Replacement functions 34

3 Pure Functional Programming **43**
Writing pure functions 44
Recursion as loops 46
The structure of a recursive function 50
Tail-recursion 60
Runtime considerations 62

4 Scope and Closures **67**
Scopes and environments 67
Environment chains, scope, and function calls 71
Scopes, lazy-evaluation, and default parameters 79
Nested functions and scopes 80
Closures 86
Reaching outside your inner-most scope 87
Lexical scope and dynamic scope 90

5 Higher-order Functions **93**

Currying 96
A parameter binding function 102
Continuation-passing style 103
Thunks and trampolines 107

6 Filter, Map, and Reduce **113**
The general sequence object in R is a list 113
Filtering sequences 116
Mapping over sequences 118
Reducing sequences 121
Bringing the functions together 124
The apply family of functions 127
Functional programming in `purrr` 131

7 Point-free Programming **139**
Function composition 139
Pipelines 142

8 Conclusions **147**
Acknowledgements 148

Introduction

Welcome to the *Advanced Statistical Programming in R* series and this book, *Functional Programming in R*. I wrote the series, and this book, to have teaching material beyond the typical introductory level most textbooks on R have. It covers more advanced techniques used in R programming such as fully exploiting functional programming, writing meta-programs (programming the actual language structures), and writing domain specific languages to embed in R.

About the series

The *Advanced Statistical Programming in R* series is intended to consist of short single-topic books where each book can be used alone for teaching or learning R. That said, there will, of course, be some dependencies in topics, if not in content, among the books. For instance, functional programming is essential to understand for any serious R programming and the first book in the series covers that. Reading the other books without understanding functional programming will not be fruitful. However, if you are already familiar with functional programming, then you can safely skip the first book.

For each book, I will make clear what I think the prerequisites for reading the book are, but for the entire series I will expect you to already be familiar with programming and the basic R you will see in any introductory book. None of the books will give you a tutorial introduction to the R programming language.

If you have used R before for data analysis and are familiar with writing expressions and functions, and want to take it further and write more advanced R code, then the series is for you.

About this book

This book is intended to give an introduction to functions in R, and how to write functional programs in R. Functional programming is a style of programming, like object-oriented programming, but one that focuses on data transformations and calculations rather than objects and state.

Where in object-oriented programming you model your programs by describing which states an object can be in and how methods will reveal or modify that state—in functional programming you model programs by describing how functions translate input data to output data. Functions themselves are considered data that you can manipulate, and much of the strength of functional programming comes from manipulating functions, building more complex functions by combining simpler functions.

The R programming language supports both object-oriented programming and functional programming, but it is mainly a functional language. It is not a "pure" functional language. Pure functional languages will not allow you to modify the state of the program by changing values parameters hold and will not allow functions to have side-effects (and need various tricks to deal with program input and output because of it).

R is somewhat close to "pure" functional languages. In general, data is immutable, so changes to data inside a function do ordinarily not alter the state of data outside that function. But R does allow side-effects, such as printing data or making plots and of course allows variables to change values.

Pure functions are functions that have no side-effects and where a function called with the same input will always return the same output. Pure functions are easier to debug and to reason about because of this. They can be reasoned about in isolation and will not depend on the context in which they are called. The R language does not guarantee that the functions you write are pure, but you can write most of your programs using only pure functions. By keeping your code mostly purely functional, you will write more robust code and code that is easier to modify when the need arises.

You will just have to move the impure functions to a small subset of your program. These functions are typically those that need to sample random data, or that produces output (either text or plots). If you know where your impure functions are, you know when to be extra careful with modifying code.

The next chapter contains a short introduction to functions in R. Some parts you might already know, and then feel free to skip ahead, but I give an

exhaustive description of how functions are defined and used to make sure that we are all on the same page. The following chapters then move on to more complex issues.

Functions in R

In this chapter we cover how to write functions in R. If you already know much of what is covered, feel free to skip ahead. We will discuss the way parameters are passed to functions as "promises", a way of passing parameters known as lazy evaluation. If you are not familiar with that but know how to write functions, you can jump forward to that section. We will also cover how to write infix operators and replacement functions, so if you do not know what those are, and how to write them, you can skip ahead to those sections. If you are new to R functions, continue reading.

Writing functions in R

You create an R function using the `function` keyword. For example, we can write a function that squares numbers like this:

```
square <- function(x) x**2
```

and use it like this

```
square(1:5)
```

```
## [1]  1  4  9 16 25
```

The function we have written takes one argument, `x`, and returns the result `x**2`. The return value of a function is always the last expression evaluated in it. If you write a function with a single expression, you can write it as above,

but for more complex functions you will typically need several statements in it. If you do, you can put the body of the function in curly brackets.

The following function does this by having three statements, one for computing the mean of its input, one for getting the standard deviation, and a final expression that returns the input scaled to be centred on the mean and having one standard deviation.

```
rescale <- function(x) {
    m <- mean(x)
    s <- sd(x)
    (x - m) / s
}
```

The first two statements are just there to define some variables we can use in the final expression. This is typical for writing short functions.

Assignments are really also expressions. They return an object, the value that is being assigned, they just do so quietly. This is why, if you put an assignment in parenthesis you will still get the value you assign printed. The parenthesis makes R remove the invisibility of the expression result so you see the actual value.

```
(x <- 1:5)
```

```
## [1] 1 2 3 4 5
```

We usually use assignments for their side-effect, assigning a name to a value so you might not think of them as expressions, but everything you do in R is actually an expression. That includes control structures like `if` statements and `for` loops. They return values. They are actually functions, and they return the last expression evaluated in them, just like all other functions. Even parenthesis and sub-scripting are functions.

If you want to return a value from a function before its last expression, you can use the `return` function. It might look like a keyword but it *is* a function, and you need to include the parenthesis when you use it. Many languages will let you return a value by writing

```
  return expression
```

Not R. In R you need to write

```
  return(expression)
```

Return is usually used to exit a function early and isn't used that much in most R code. It is easier to return a value by just making it the last expression in a function than it is to explicitly use `return`. But you can use it to return early like this:

```
rescale <- function(x, only_translate) {
    m <- mean(x)
    translated <- x - m
    if (only_translate) return(translated)
    s <- sd(x)
    translated / s
}
rescale(1:4, TRUE)
```

```
## [1] -1.5 -0.5  0.5  1.5
```

```
rescale(1:4, FALSE)
```

```
## [1] -1.1618950 -0.3872983  0.3872983  1.1618950
```

This function has two arguments, `x` and `only_translate`. Your functions can have any number of parameters. When a function takes many arguments, however, it becomes harder to remember in which order you have to put them. To get around that problem, R allows you to provide the arguments to a function using their names. So the two function calls above can also be written as

```
rescale(x = 1:4, only_translate = TRUE)
rescale(x = 1:4, only_translate = FALSE)
```

Named parameters and default parameters

If you use named arguments, the order doesn't matter, so this is also equivalent to these function calls:

```
rescale(only_translate = TRUE, x = 1:4)
rescale(only_translate = FALSE, x = 1:4)
```

You can mix positional and named arguments. The positional arguments have to come in the same order as used in the function definition and the named arguments can come in any order. All the four function calls below are equivalent:

```
rescale(1:4, only_translate = TRUE)
rescale(only_translate = TRUE, 1:4)
rescale(x = 1:4, TRUE)
rescale(TRUE, x = 1:4)
```

When you provide a named argument to a function, you don't need to use the full parameter name. Any unique prefix will do. So we could also have used the two function calls below:

```
rescale(1:4, o = TRUE)
rescale(o = TRUE, 1:4)
```

This is convenient for interactive work with R because it saves some typing, but I do not recommend it when you are writing programs. It can easily get confusing and if the author of the function adds a new argument to the function with the same prefix as the one you use it will break your code. If the function author provides a default value for that parameter, your code will *not* break, if you use the full argument name.

Now default parameters are provided when the function is defined. We could have given `rescale` a default parameter for `only_translate` like this:

```
rescale <- function(x, only_translate = FALSE) {
    m <- mean(x)
    translated <- x - m
```

```
    if (only_translate) return(translated)
    s <- sd(x)
    translated / s
}
```

Then, if we call the function we only need to provide `x` if we are happy with the default value for `only_translate`.

```
rescale(1:4)
```

```
## [1] -1.1618950 -0.3872983  0.3872983  1.1618950
```

R makes heavy use of default parameters. Many commonly used functions, such as plotting functions and model fitting functions, have lots of arguments. These arguments let you control in great detail what the functions do, making them very flexible, and because they have default values you usually only have to worry about a few of them.

The "gobble up everything else" parameter: "..."

There is a special parameter all functions can take called `...`. This parameter is typically used to pass parameters on to functions *called within* a function. To give an example, we can use it to deal with missing values, `NA`, in the `rescale` function.

We can write (where I'm building from the shorter version):

```
rescale <- function(x, ...) {
    m <- mean(x, ...)
    s <- sd(x, ...)
    (x - m) / s
}
```

If we give this function a vector `x` that contains missing values it will return `NA`.

```
x <- c(NA, 1:3)
rescale(x)
```

```
## [1] NA NA NA NA
```

It would also have done that before because that is how the functions `mean` and `sd` work. But both of these functions take an additional parameter, `na.rm`, that will make them remove all `NA` values before they do their computations. Our `rescale` function can do the same now:

```
rescale(x, na.rm = TRUE)
```

```
## [1] NA -1  0  1
```

The first value in the output is still `NA`. Rescaling an `NA` value can't be anything else. But the rest are rescaled values where that `NA` was ignored when computing the mean and standard deviation.

The "`...`" parameter allows a function to take any named parameter at all. If you write a function without it, it will only take the parameters, but if you add this parameter, it will accept any named parameter at all.

```
f <- function(x) x
g <- function(x, ...) x
f(1:4, foo = "bar")
```

```
## Error in f(1:4, foo = "bar"): unused argument (foo = "bar")
```

```
g(1:4, foo = "bar")
```

```
## [1] 1 2 3 4
```

If you then call another function with "`...`" as a parameter then all the parameters the first function doesn't know about will be passed on to the second function.

```
f <- function(...) list(...)
g <- function(x, y, ...) f(...)
g(x = 1, y = 2, z = 3, w = 4)
```

```
## $z
## [1] 3
##
## $w
## [1] 4
```

In the example above, function `f` creates a list of named elements from "`...`" and as you can see it gets the parameters that `g` doesn't explicitly takes.

Using "`...`" is not particularly safe. It is often very hard to figure out what it actually does in a particular piece of code. What is passed on to other functions depend on what the first function explicitly takes as arguments, and when you call a second function using it you pass on all the parameters in it. If the function doesn't know how to deal with them, you get an error.

```
f <- function(w) w
g <- function(x, y, ...) f(...)
g(x = 1, y = 2, z = 3, w = 4)
```

```
## Error in f(...): unused argument (z = 3)
```

In the `rescale` function it would have been much better to add the `rm.na` parameter explicitly.

That being said, "`...`" is frequently used in R. Particularly because many functions take very many parameters with default values, and adding these parameters to all functions calling them would be tedious and error-prone. It is also the best way to add parameters when specialising generic functions, which is a topic for another book in this series: *Object Oriented Programming in R*.

To explicitly get hold of the parameters passed along in "`...`" you can use this invocation: `eval(substitute(alist(...)))`.

```
parameters <- function(...) eval(substitute(alist(...)))
parameters(a = 4, b = a**2)
```

```
## $a
## [1] 4
```

```
##
## $b
## a^2
```

The `alist` function creates a list of names for each parameter and values for the expressions given.

```
alist(a = 4, b = a**2)
```

```
## $a
## [1] 4
##
## $b
## a^2
```

You cannot use the `list` function for this unless you want all the expression evaluated. If you try to use `list` you can get errors like this:

```
list(a = 4, b = a**2)
```

```
## Error in eval(expr, envir, enclos): object 'a' not found
```

Because R uses so-called lazy evaluation for function parameters, something we return to shortly, it will be perfectly fine to define a function with default parameters that are expressions that can only right-hand. Inside a function that knows the parameters `a` and `b` you can evaluate expressions that use these parameters, even when they are not defined outside the function. So the parameters given to `alist` above is something you can use as default parameters when defining a function. But you cannot create the list using `list` because it will try to evaluate the expressions.

The reason you also need to substitute and evaluate is that `alist` will give you exactly the parameters you provide it. If you tried to use `alist` on "`...`" you would just get "`...`" back.

```
parameters <- function(...) alist(...)
parameters(a = 4, b = x**2)
```

```
## [[1]]
## ...
```

By substituting we translate "`...`" into the actual parameters given and by evaluating we get the list `alist` would give us in this context: the list of parameters and their associated expressions.

Functions don't have names

The last thing I want to stress when we talk about defining functions is that functions do not have names. Variables have names, and variables can refer to functions, but these are two separate things.

In many languages, such as Java, Python, or C++, you define a function and at the same time, you give it an argument. When possible at all, you need a special syntax to define a function without a name.

Not so in R. In R, functions do not have names, and when you define them, you are not giving them a name. We have given names to all the functions we have used above by assigning them to variables right where we defined them. We didn't have to. It is the "`function(...) ...`" syntax that defines a function. We are defining a function whether we assign it to a variable or not.

We can define a function and call it immediately like this:

```
(function(x) x**2)(2)
```

```
## [1] 4
```

We would never do this, of course. Anywhere we would want to define an anonymous function and immediately call it we could instead just put the body of the function. Functions we want to reuse we *have* to give a name so we can get to them again.

The syntax for defining functions, however, doesn't force us to give them names. When you start to write higher-order functions, that is functions that take other functions as input or return functions, this is convenient.

Such higher-order functions are an important part of functional programming, and we will see much of them later in the book.

Lazy evaluation

Expressions used in a function call are not evaluated before they are passed to the function. Most common languages have so-called pass-by-value semantics, which means that all expressions given to parameters of a function are evaluated before the function is called. In R, the semantic is "call-by-promise", also known as "lazy evaluation".

When you call a function and give it expressions as its arguments, these are not evaluated at that point. What the function gets is not the result of evaluating them but the actual expressions, called "promises" (they are promises of an evaluation to a value you can get when you need it). Thus the term "call-by-promise". These expressions are only evaluated when they are actually needed, thus the term "lazy evaluation".

This has several consequences for how functions work. First of all, an expression that isn't used in a function isn't evaluated.

```
f <- function(a, b) a
f(2, stop("error if evaluated"))
```

```
## [1] 2
```

```
f(stop("error if evaluated"), 2)
```

```
## Error in f(stop("error if evaluated"), 2): error if evaluated
```

If you have a parameter that can only be meaningfully evaluated in certain contexts, it is safe enough to have it as a parameter as long as you only refer to it when those necessary conditions are met.

It is also very useful for default values of parameters. These are evaluated inside the scope of the function, so you can write default values that depend on other parameters:

```
f <- function(a, b = a) a + b
f(a = 2)
```

```
## [1] 4
```

This does *not* mean that all the expressions are evaluated inside the scope of the function, though. We discuss scopes in a later chapter but for now, you can think of two scopes: the global scope where global variables live, and the function scope that has parameters and local variables as well.

If you call a function like this

```
f(a = 2, b = a)
```

you will get an error if you expect `b` to be the same as `a` inside the function. If you are lucky, and there isn't any global variable called `a` you will get a runtime error. If you are unlucky and there *is* a global variable called `a`, that is what `b` will be set to, and if you expect it to be set to 2 here your code will just give you an incorrect answer.

Using other parameters work for default values because these are evaluated inside the function. The expressions you give to function calls are evaluated in the scope outside the function.

This also means that you cannot change what an expression evaluates to just by changing a local variable.

```
a <- 4
f <- function(x) {
    a <- 2
    x
}
f(1 + a)
```

```
## [1] 5
```

In this example, the expression `1 + a` is evaluated inside `f`, but the `a` in the expression is the `a` outside of `f` and not the `a` local variable inside `f`.

This is of course what you want. If expressions really *were* evaluated inside the scope of the function, then you would have no idea what they evaluated to if you called a function with an expression. It would depend on any local variables the function might use.

Because expressions are evaluated in the calling scope and not the scope of the function, you mostly won't notice the difference between call-by-value or

call-by-promise. There are certain cases where the difference can bite you, though, if you are not careful.

As an example we can consider this function:

```
f <- function(a) function(b) a + b
```

This might look a bit odd if you are not used to it, but it is a function that returns another function. We will see many examples of this kind of functions in later chapters.

When we call `f` with a parameter `a` we get a function back that will add `a` to its argument.

```
f(2)(2)
```

```
## [1] 4
```

We can create a list of functions from this `f`:

```
ff <- vector("list", 4)
for (i in 1:4) {
  ff[[i]] <- f(i)
}
ff
```

```
## [[1]]
## function (b)
## a + b
## <environment: 0x7fb0afa49870>
##
## [[2]]
## function (b)
## a + b
## <environment: 0x7fb0afa4a800>
##
## [[3]]
## function (b)
```

```
## a + b
## <environment: 0x7fb0afa4ac60>
##
## [[4]]
## function (b)
## a + b
## <environment: 0x7fb0afa4b988>
```

Here, `ff` contains four functions and the idea is that the first of these adds 1 to its argument, the second add 2, and so on.

If we call the functions in `ff`, however, weird stuff happens.

```
ff[[1]](1)
```

```
## [1] 5
```

When we get the element `ff[[1]]` we get the first function we created in the loop. If we substitute into `f` the value we gave in the call, this is

```
function(b) i + b
```

The parameter `a` from `f` has been set to the parameter we gave it, `i`, but `i` has not been evaluated at this point!

When we call the function, the expression is evaluated, in the global scope, but there `i` now has the value 4 because that is the last value it was assigned in the loop. The value of `i` was 1 when we called it to create the function but it is 5 when the expression is actually evaluated.

This also means that we can chance the value of `i` before we evaluate one of the functions, and this changes it from the value we intended when we created the function.

```
i <- 1
ff[[2]](1)
```

```
## [1] 2
```

This laziness is only in effect the first time we call the function. If we change `i` again and call the function, we get the same value as the first time the function was evaluated.

```
i <- 2
ff[[2]](1)
```

```
## [1] 2
```

We can see this in effect by looping over the functions and evaluating them.

```
results <- vector("numeric", 4)
for (i in 1:4) {
  results[i] <- ff[[i]](1)
}
results
```

```
## [1] 5 2 4 5
```

We have already evaluated the first two functions so they are stuck at the values they got at the time of the first evaluation. The other two get the intended functionality but only because we are setting the variable `i` in the loop where we evaluate the functions. If we had used a different loop variable, we wouldn't have been so lucky.

This problem is caused by having a function with an unevaluated expression that depends on variables in the outer scope. Functions that return functions are not uncommon, though, if you want to exploit the benefits of having a functional language. It is also a consequence of allowing variables to change values, something most functional programming languages do not allow for such reasons. You cannot entirely avoid it, though, by avoiding `for`-loops. If you call functions that loop for you, you do not necessarily have control over how they do that, and you can end up in the same situation.

The way to avoid the problem is to force and evaluation of the parameter. If you evaluate it once, it will remember the value, and the problem is gone.

You can do that by just writing the parameter as a single statement. That will evaluate it. It is better to use the function `force` though, to make it explicit

that this is what you are doing. It really just gives you the expression back, so it works exactly as if you just wrote the parameter, but the code makes clear why you are doing it.

If you do this, the problem is gone.

```
f <- function(a) {
  force(a)
  function(b) a + b
}

ff <- vector("list", 4)
for (i in 1:4) {
  ff[[i]] <- f(i)
}

ff[[1]](1)
```

```
## [1] 2
```

```
i <- 1
ff[[2]](1)
```

```
## [1] 3
```

Getting back to parameters that are given expressions as arguments we can take a look at what they actually are represented as. We can use the function `parameters` we wrote above.

If we give a function an actual value, that is what the function gets.

```
parameters <- function(...) eval(substitute(alist(...)))

p <- parameters(x = 2)
class(p$x)
```

```
## [1] "numeric"
```

The `class` function here tells us the type of the parameter. For the number 2, this is "`numeric`".

If we give it a name we are giving it an expression, even if the name refers to a single value.

```
a <- 2
p <- parameters(x = a)
class(p$x)
```

```
## [1] "name"
```

The type is a "`name`". If we try to evaluate it, it will evaluate to the value of that parameter.

```
eval(p$x)
```

```
## [1] 2
```

It knows that the variable is in the global scope, so if we change it the expression will reflect this if we evaluate it.

```
a <- 4
eval(p$x)
```

```
## [1] 4
```

If we call the function with an expression the type will be "`call`".

```
p <- parameters(x = 2 * y)
class(p$x)
```

```
## [1] "call"
```

This is because all expressions are function calls. In this case, it is the function `*` that is being called.

We can only evaluate it if the variables the expression refers to are in the scope of the expression. So evaluating this expression will give us an error because `y` is not defined.

```
eval(p$x)
```

```
## Error in eval(expr, envir, enclos): object 'y' not found
```

The parameter `y` has to be in the calling scope, just as we saw earlier. Expressions are evaluated in the calling scope, not inside the function, so we cannot define `y` inside the `parameters` function and get an expression we can evaluate.

```
parameters2 <- function(...) {
  y <- 2
  eval(substitute(alist(...)))
}
p2 <- parameters(x = 2 * y)
eval(p2$x)
```

```
## Error in eval(expr, envir, enclos): object 'y' not found
```

We can set the variable and then evaluate it, though.

```
y <- 2
eval(p$x)
```

```
## [1] 4
```

Alternatively, we can explicitly set the variable in an environment given to the `eval` function.

```
eval(p$x, list(y = 4))
```

```
## [1] 8
```

Actually, manipulating expressions and the scope they are evaluated in is a very powerful tool, but beyond what we will cover in this book. It is the topic for a later book in the series: *Meta-programming in R*.

Vectorised functions

Expressions in R are vectorised. When you write an expression, it is implicitly working on vectors of values, not single values. Even simple expressions that only involves numbers really are vector operations. They are just vectors of length 1.

For longer vectors, expressions work component-wise. So if you have vectors `x` and `y` you can subtract the first from the second component wise just by writing `x - y`.

```
x <- 1:5
y <- 6:10
x - y
```

```
## [1] -5 -5 -5 -5 -5
```

If the vectors are not of the same length, the shorter vector is just repeated as many times as is necessary. This is why you can, for example, multiply a number to a vector.

```
2 * x
```

```
## [1]  2  4  6  8 10
```

Here `2` is a vector of length 1 and `x` a vector of length 5, and `2` is just repeated five times. You will get a warning if the length of the longer vector is not divisible by the length of the shorter vector, and you generally want to avoid this. The semantic is the same, though: R just keep repeating the shorter vector as many times as needed.

```
x <- 1:6
y <- 1:3
x - y
```

```
## [1] 0 0 0 3 3 3
```

Depending on how a function is written it can also be used in vectorised expressions. R is happy to use the result of a function call in a vector expression as long as this result is a vector. This is not quite the same as the function operating component-wise on vectors. Such functions we can call *vectorised* functions.

Most mathematical functions such as `sqrt`, `log`, `cos` and `sin` are vectorised and you can use them in expressions.

```
log(1:3) - sqrt(1:3)
```

```
## [1] -1.0000000 -0.7210664 -0.6334385
```

Functions you write yourself will also be vectorised if their body consist only of vectorised expressions.

```
f <- function(a, b) log(a) - sqrt(b)
f(1:3, 1:3)
```

```
## [1] -1.0000000 -0.7210664 -0.6334385
```

The very first function we wrote in this book, `square`, was also a vectorised function. The function `scale` was also, although the functions it used, `mean` and `sd` are not; they take vector input but return a summary of the entire vector and do not operate on the vector component-wise.

A function that uses control structures usually will not be vectorised. We can write a comparison function that returns -1 if the first argument is smaller than the second and 1 if the second is larger than the first, and zero otherwise like this:

```
compare <- function(x, y) {
    if (x < y) {
        -1
    } else if (y < x) {
        1
```

```
    } else {
        0
    }
}
```

This function will work fine on single values but not on vectors. The problem is that the `if`-expression only looks at the first element in the logical vector `x < y`. (Yes, `x < y` is a vector because $<$ is a vectorised function).

To handle `if`-expressions we can get around this problem by using the `ifelse` function. This is a vectorized function that behaves just as an `if-else`-expression.

```
compare <- function(x, y) {
    ifelse(x < y, -1, ifelse(y < x, 1, 0))
}
compare(1:6, 1:3)
```

```
## [1] 0 0 0 1 1 1
```

The situation is not always so simple that we can replace `if`-statements with `ifelse`. In most cases, we can, but when we cannot we can instead use the function `Vectorize`. This function takes a function that can operate on single values and translate it into a function that can work component-wise on vectors.

As an example, we can take the `compare` function from before and vectorize it.

```
compare <- function(x, y) {
    if (x < y) {
        -1
    } else if (y < x) {
        1
    } else {
        0
    }
}
compare <- Vectorize(compare)
compare(1:6, 1:3)
```

```
## [1] 0 0 0 1 1 1
```

By default, `Vectorize` will vectorise on all parameters of a function. As an example, imagine that we want a `scale` function that doesn't scale all variables in a vector by the same vector's mean and standard deviation but use the mean and standard deviation of another vector.

```
scale_with <- function(x, y) {
    (x - mean(y)) / sd(y)
}
```

This function is already vectorised on its first parameter since it just consists of a vectorised expression, but if we use `Vectorize` on it, we break it.

```
scale_with(1:6, 1:3)
```

```
## [1] -1  0  1  2  3  4
```

```
scale_with <- Vectorize(scale_with)
scale_with(1:6, 1:3)
```

```
## [1] NA NA NA NA NA NA
```

The function we create with `Vectorize` is vectorized for both `x` and `y`, which means that it operates on these component-wise. When scaling, the function only sees one component of `y`, not the whole vector. The result is a vector of missing values, `NA` because the standard deviation of a single value is not defined.

We can fix this by explicitly telling `Vectorize` which parameters should be vectorised. In this example only parameter `x`.

```
scale_with <- function(x, y) {
    (x - mean(y)) / sd(y)
}
scale_with <- Vectorize(scale_with, vectorize.args="x")
scale_with(1:6, 1:3)
```

```
## [1] -1  0  1  2  3  4
```

Simple functions are usually already vectorised, or can easily be made vectorised using `ifelse`, but for functions more complex the `Vectorize` function is needed.

As an example we can consider a tree data structure and a function for computing the node depth of a named node—the node depth defined as the distance from the root. For simplicity we consider only binary trees. We can implement trees using lists:

```
make_node <- function(name, left = NULL, right = NULL)
  list(name = name, left = left, right = right)

tree <- make_node("root",
                  make_node("C", make_node("A"),
                                 make_node("B")),
                  make_node("D"))
```

To compute the node depth we can traverse the tree recursively:

```
node_depth <- function(tree, name, depth = 0) {
    if (is.null(tree))     return(NA)
    if (tree$name == name) return(depth)

    left <- node_depth(tree$left, name, depth + 1)
    if (!is.na(left)) return(left)
    right <- node_depth(tree$right, name, depth + 1)
    return(right)
}
```

This is not an unreasonably complex function, but it is a function that is harder to vectorise than the `scale_with` function. As it is, it works well for single names

```
node_depth(tree, "D")
```

```
## [1] 1
```

```
node_depth(tree, "A")
```

```
## [1] 2
```

but you will get an error if you call it on a sequence of names

```
node_depth(tree, c("A", "B", "C", "D"))
```

It is not hard to imagine that a vectorised version could be useful, however. For example to get the depth of a sequence of names.

```
node_depth <- Vectorize(node_depth, vectorize.args = "name",
                        USE.NAMES = FALSE)
node_depth(tree, c("A", "B", "C", "D"))
```

```
## [1] 2 2 1 1
```

Here the `USE.NAMES = FALSE` is needed to get a simple vector out. If we did not include it, the vector would have names based on the input variables. See the documentation for `Vectorize` for details.

Infix operators

Infix operators in R are also functions. You can over-write them with other functions (but you really shouldn't since you could mess up a lot of code), and you can also make your own infix operators.

User defined infix operators are functions with special names. A function with a name that starts and ends with `%` will be considered an infix operator by R. There is no special syntax for creating a function to be used as an infix operator, except that it should take two arguments. There is a special syntax for assigning, variables, though, including variables with names starting and ending with `%`.

To work with special variables you need to quote them with back-ticks. You cannot write `+(2, 2)` even though `+` is a function. R will not understand this as a function call when it sees it written like this. But you can take `+` and quote it, `` `+` ``, and then it is just a variable name like all other variable names.

```
`+`(2, 2)
```

```
## [1] 4
```

The same goes for all other infix operators in R and even control structures.

```
`if`(2 > 3, "true", "false")
```

```
## [1] "false"
```

The R parser recognises control structure keywords and various operators, e.g. the arithmetic operators, and therefore these get to have a special syntax. But they are all really just functions. Even parentheses are a special syntax for the function `` `(` `` and the subscript operators are as well, `` `[` `` and `` `[[` `` respectively. If you quote them, you get a variable name that you can use just like any other function name.

Just like all these operators have a special syntax, variables starting and ending with `%` get a special syntax. R expects them to be infix operators and will treat an expression like this:

```
exp1 %op% exp2
```

as the function call

```
`%op%`(exp1, exp2)
```

Knowing that you can translate an operator name into a function name just by quoting it also tells you how to define new infix operators. If you assign a function to a variable name, you can refer to it by that name. If that name is a quoted special name, it gets to use the special syntax for that name.

So, to define an operator `%x%` that does multiplication we can write

```
`%x%` <- `*`
3 %x% 2
```

```
## [1] 6
```

Here we used quotes twice, first to get a variable name we could assign to for `%x%` and once to get the function that the multiplication operator, `*`, points to.

Just because all control structures and operators in R are functions that you can overwrite, you shouldn't go doing that without extreme caution. You should *never* change the functions the control structures point to, and you should not change the other operators unless you are defining operators on new types where it is relatively safe to do so (and I will tell you how to in the *Object Oriented Programming in R* in this series). Defining entirely new infix operators, however, can be quite useful if they simplify expressions you write often.

As an example let us do something else with the `%x%` operator—after all, there is no point in having two different operators for multiplication. We can make it replicate the left-hand side a number of times given by the right-hand side:

```
`%x%` <- function(expr, num) replicate(num, expr)
3 %x% 5
```

```
## [1] 3 3 3 3 3
```

```
cat("This is ", "very " %x% 3, "much fun")
```

```
## This is  very  very  very  much fun
```

We are using the `replicate` function to achieve this. It does the same thing. It repeatedly evaluates an expression a fixed number of times. Using `%x%` infix might give more readable code, depending on your taste.

In the interest of honesty, I must mention, though, that we haven't just given `replicate` a new name here, switching of arguments aside. The `%x%` operator works slightly differently. In `%x%` the `expr` parameter is evaluated when we call `replicate`. So if we call `%x%` with a function that samples random numbers we will get the same result repeated `num` times; we will not sample `num` times.

```
rnorm(1) %x% 4
```

```
## [1] 0.7096367 0.7096367 0.7096367 0.7096367
```

Lazy evaluation only takes you so far.

To actually get the same behaviour as `replicate` we need a little more trickery:

```
`%x%` <- function(expr, num) {
  m <- match.call()
  replicate(num, eval.parent(m$expr))
}
rnorm(1) %x% 4
```

```
## [1]  0.3572523  0.3733601 -0.8712691 -0.9961701
```

Here the `match.call` function just gets us a representation of the current function call from which we can extract the expression without evaluating it. We then use `replicate` to evaluate it a number of times in the calling function's scope.

If you don't quite get this, don't worry. We cover scopes in a later chapter.

Replacement functions

Another class of functions with special names is the so-called *replacement functions*. Data in R is immutable. You can change what data parameters point to, but you cannot change the actual data. Even when it looks like you are modifying data you are, at least conceptually, creating a copy, modifying that, and making a variable point to the new copy.

We can see this in a simple example with vectors. If we create two vectors that point to the same initial vector, and then modify one of them, the other remains unchanged.

```
x <- y <- 1:5
x
```

```
## [1] 1 2 3 4 5
```

```
y
```

```
## [1] 1 2 3 4 5
```

```
x[1] <- 6
x
```

```
## [1] 6 2 3 4 5
```

```
y
```

```
## [1] 1 2 3 4 5
```

R is smart about it. It won't make a copy of values if it doesn't have to. Semantically it is best to think of any modification as creating a copy, but for performance reasons R will only make a copy when it is necessary. At least for built-in data like vectors. Typically, this happens when you have two variables referring to the same data and you "modify" one of them.

We can use the `address` function to get the memory address of an object. This will change when a copy is made but will remain the same when it isn't. And we can use the `mem_change` function from the `pryr` package to see how much memory is allocated for an expression. Using these two functions, we can dig a little deeper into this copying mechanism.

We can start by creating a long-ish vector and modifying it.

```
library(pryr)
```

```
rm(x) ; rm(y)
mem_change(x <- 1:10000000)
```

```
## 40 MB
```

```
address(x)
```

```
## [1] "0x10e5d6000"
```

```
mem_change(x[1] <- 6)
```

```
## 40 MB
```

```
address(x)
```

```
## [1] "0x110bfc000"
```

When we assign to the first element in this vector, we see that the entire vector is being copied. This might look odd since I just told you that R would only copy a vector if it had to, and here we are just modifying an element in it, and no other variable refers to it.

The reason we get a copy here is that the expression we used to create the vector, `1:10000000`, creates an integer vector. The value `6` we assign to the first element is a floating point, called "numeric" in R. If we want an actual integer we have to write "L" after the number.

```
class(6)
```

```
## [1] "numeric"
```

```
class(6L)
```

```
## [1] "integer"
```

When we assign a numeric to an integer vector, R has to convert the entire vector into numeric, and that is why we get a copy.

```
z <- 1:5
class(z)
```

```
## [1] "integer"
```

```
z[1] <- 6
class(z)
```

```
## [1] "numeric"
```

If we assign another numeric to it, after it has been converted, we no longer get a copy.

```
mem_change(x[3] <- 8)
```

```
## -360 B
```

```
address(x)
```

```
## [1] "0x110bfc000"
```

All expression evaluations modify the memory a little, up or down, but the change is much smaller than the entire vector so we can see that the vector isn't being copied, and the address remains the same.

If we assign `x` to another variable, we do not get a copy. We just have the two names refer to the same value.

```
mem_change(y <- x)
```

```
## 1.15 kB
```

```
address(x)
```

```
## [1] "0x110bfc000"
```

```
address(y)
```

```
## [1] "0x110bfc000"
```

If we change `x` again, though, we need a copy to make the other vector point to the original, unmodified data.

```
mem_change(x[3] <- 8)
```

```
## 80 MB
```

```
address(x)
```

```
## [1] "0x115848000"
```

```
address(y)
```

```
## [1] "0x110bfc000"
```

But after that copy, we can again assign to `x` without making additional copies.

```
mem_change(x[4] <- 9)
```

```
## 888 B
```

```
address(x)
```

```
## [1] "0x115848000"
```

When you assign to a variable in R, you are calling the assignment function, `` `<-` ``. When you assign to an element in a vector or list you are using the `` `[<-` `` function. But there is a whole class of such functions you can use to make the appearance of modifying an object, without actually doing it of course. These are called replacement functions and have names that ends in `<-`. An example is the `` `names<-` `` function. If you have a vector `x` you can give its elements names using this syntax:

```
x <- 1:4
x
```

```
## [1] 1 2 3 4
```

```
names(x) <- letters[1:4]
x
```

```
## a b c d
## 1 2 3 4
```

```
names(x)
```

```
## [1] "a" "b" "c" "d"
```

There are two different functions in play here. The last expression, which gives us `x`'s names, is the `names` function. The function we use to *assign* the names to `x` is the `` `names<-` `` function.

Any function you define whose name ends with `<-` becomes a replacement function, and the syntax for it is that it evaluates whatever is on the right-hand side of the assignment operator and assigns the result to the variable that it takes as its argument.

So this syntax

```
names(x) <- letters[1:4]
```

is translated into

```
x <- `names<-`(x, letters[1:4])
```

No values are harmed in the evaluation of this, but the variable is set to the new value.

We can write our own replacement functions using this syntax. There are just two requirements. The function name has to end with `<-` —so we need to quote the name when we assign to it—and the argument for the value that goes to the right-hand side of the assignment has to be named `value`. The last requirement is there so replacement functions can take more than two arguments.

The `` `attr<-` `` function is an example of this. Attributes are key-value maps that can be associated with objects. You can get the attributes associated with an object using the `attributes` function and set all attributes with the `` `attributes<-` `` function, but you can assign individual attributes using `` `attr<-` ``. It takes three arguments, the object to modify, a `which` parameter that is the name of the attribute to set, and the `value` argument that goes to the right-hand side of the assignment. The `which` argument is passed to the function on the left-hand side together with the object to modify.

```
x <- 1:4
attributes(x)

## NULL

attributes(x) <- list(foo = "bar")
attributes(x)

## $foo
## [1] "bar"

attr(x, "baz") <- "qux"
attributes(x)

## $foo
## [1] "bar"
##
## $baz
## [1] "qux"
```

We can write a replacement function to make the tree construction we had earlier in the chapter slightly more readable. Earlier we constructed the tree like this:

```
tree <- make_node("root",
                  make_node("C", make_node("A"),
                                 make_node("B")),
                  make_node("D"))
```

but we can make functions for setting the children of an object like this:

```
`left<-` <- function(node, value) {
    node$left = value
    node
}
`right<-` <- function(node, value) {
    node$right = value
    node
}
```

and then construct the tree like this:

```
A <- make_node("A")
B <- make_node("B")
C <- make_node("C")
D <- make_node("D")
root <- make_node("root")
left(C) <- A
right(C) <- B
left(root) <- C
right(root) <- D
tree <- root
```

To see the result, we can write a function for printing a tree. To keep the function simple, I assume that either both children are `NULL` or both are trees. It is simple to extend it to deal with trees that do not satisfy that, it just makes the function a bit longer.

```
print_tree <- function(tree) {
  build_string <- function(node) {
    if (is.null(node$left) && is.null(node$right)) {
        node$name
    } else {
        left <- build_string(node$left)
        right <- build_string(node$right)
        paste0("(", left, ",", right, ")")
    }
  }
  build_string(tree)
}
print_tree(tree)
```

```
## [1] "((A,B),D)"
```

This function shows the tree in what is known as the Newick format and doesn't show the names of inner nodes, but you can see the structure of the tree.

The order in which we build the tree using `children` is important. When we set the children for `root`, we refer to the variable `C`. If we set the children for `C` *after* we set the children for `root` we get the *old* version of `C`, no the new modified version.

```
A <- make_node("A")
B <- make_node("B")
C <- make_node("C")
D <- make_node("D")
root <- make_node("root")
left(root) <- C
right(root) <- D
left(C) <- A
right(C) <- B
tree <- root
print_tree(tree)
```

```
## [1] "(C,D)"
```

Replacement functions only look like they are modifying data. They are not. They only reassign values to variables.

Pure Functional Programming

A "pure" function is a function that behaves like a mathematical function: it maps values from one space to another, the same value always maps to the same result, and there is no such thing as "side effects" in a mathematical function.

The level to which programming languages go to ensure that functions are pure varies and R does precious little. Because values are immutable you have some guarantee about which side-effects functions can have, but not much. Functions can modify variables outside their scope, for example, modify global variables. They can print or plot and alter the state of the R process this way. They can also sample random numbers and use them for their computations, making the result of a function non-deterministic, for all intents and purposes, so the same value does not always map to the same result.

Pure functions are desirable because they are easier to reason about. If a function does not have any side-effects, you can treat it as a black box that just maps between values. If it has side-effects, you will also need to know how it modifies other parts of the program and that means you have to understand, at least at some level, what the body of the function is doing. If all you need to know about a function is how it maps from input parameters to results, you can change their implementation at any point without breaking any code that relies on the functions.

Non-deterministic functions, functions whose result is not always the same on the same input, are not necessarily hard to reason about. They are just harder to test and debug if their results depend on random numbers.

Since pure functions are easier to reason about, and to test, you will want to write as much of your programs using pure functions. You cannot necessarily write *all* your programs in pure functions. Sometimes you need randomness to

implement Monte Carlo methods, or sometimes you need functions to produce output. But if you write most of your program using pure functions, and know where the impure functions are, you are writing better and more robust programs.

Writing pure functions

There is not much work involved in guaranteeing that a function you write is pure. You should avoid sampling random numbers and stay away from modifying variables outside the scope of the function.

It is trivial to avoid sampling random numbers. Don't call any function that does it, either directly or through other functions. If a function only calls deterministic functions and doesn't introduce any randomness itself, then the function will be deterministic.

It is only slightly less trivial to guarantee that you are not modifying something outside of the scope of a function. You need a special function, `<<-`, to modify variables outside of a function's local scope (we return to this operator in the next chapter), so avoid using that. Then all assignments will be to local variables, and you cannot modify variables in other scopes. If you avoid this operator, then the only risk you have of modifying functions outside of your scope is through lazy-evaluation. Remember the example from the previous chapter where we created a list of functions. The functions contained unevaluated expressions whose values depended on a variable we were changing before we evaluated the expressions.

Strictly speaking, we would still have a pure function if we returned functions with such an unevaluated expression, that depended on local variables. Even the functions we would be returning would be pure. They would be referring to local variables in the first function, variables that cannot be changed without the `<<-` operator once the first function returns. They just wouldn't necessarily be the functions you intended to return.

While such a function would be pure by the strict definition, they do have the problem that the functions we return depend on the state of local variables inside the first function. From a programming perspective, it doesn't much help us that the functions are pure if they are hard to reason about, and in this case, we would need to know how changing the state in the first function affects the functionality of the others.

A solution to avoid problems with functions depending on variables outside their own scope, that is variables that are not either arguments or local variables, is to simply never change what a variable points to.

Programming languages that guarantee that functions are pure enforce this. There simply isn't any way to modify the value of a variable. You can define constants, but there is no such thing as variables. Since R does have variables, you will have to avoid assigning to the same variable more than once if you want to guarantee that your functions are pure in this way.

It is very easy to accidentally reuse a variable name in a long function, even if you never intended to change the value of a variable. Keeping your functions short and simple alleviates this somewhat, but there is no way to guarantee that it doesn't happen. The only control structure that actually forces you to change variables is `for`-loops. You simply cannot write a `for`-loop without having a variable to loop over.

Now `for`-loops have a bad reputation in R, and many will tell you to avoid them because they are slow. This is not true. They are slow compared to loops in more low-level languages, but this has nothing to do with them being loops. Because R is a very dynamic language where everything you do involves calling functions, and functions that can be changed at any point if someone redefines a variable, R code is just generally slow. When you call built-in functions like `sum` or `mean` they are fast because they are implemented in C. By using vectorized expressions and built-in functions you do not pay the penalty of the dynamism. If you write a loop yourself in R, then you do. You pay the same price, however, if you use some of the other constructions that people recommend instead of loops; constructions we will return to in the *Filter, Map, and Reduce* chapter.

The real reason you should use such other constructions is that they make the intent behind your code clearer than a loop will do, at least once you get familiar with functional programming, and because you avoid the looping variable that can cause problems. The reason people often find that their loops are inefficient is that it is very easy to write loops that "modify" data, forcing R to make copies. This problem doesn't go away just because we avoid loops and is something we return to later towards the end of this chapter.

Recursion as loops

The way functional programming languages avoid loops is by using recursion instead. Anything you can write as a loop you can also write using recursive function calls and most of this chapter will be focusing on getting used to thinking in terms of recursive functions.

Thinking of problems as recursive is not just a programming trick for avoiding loops. It is generally a method of breaking problems into simpler sub-problems that are easier to solve. When you have to address a problem, you can first consider whether there are base cases that are trivial to solve. If we want to search for an element in a sequence, it is trivial to determine if it is there if the sequence is empty. Then it obviously isn't there. Now, if the sequence isn't empty we have a harder problem, but we can break it into two smaller problems. Is the element we are searching for the first element in the sequence? If so, the element is there. If the first element is not equal to the element we are searching for, then it is only in the sequence if it is the remainder of the sequence.

We can write a linear search function based on this breakdown of the problem. We will first check for the base case and return `FALSE` if we are searching in an empty sequence. Otherwise, we check the first element, if we find it, we return `TRUE`, and if it wasn't the first element we call the function recursively on the rest of the sequence.

```
lin_search <- function(element, sequence) {
    if (is_empty(sequence))              FALSE
    else if (first(sequence) == element) TRUE
    else lin_search(element, rest(sequence))
}

x <- 1:5
lin_search(0, x)

## [1] FALSE

lin_search(1, x)

## [1] TRUE
```

```
lin_search(5, x)
```

```
## [1] TRUE
```

```
lin_search(6, x)
```

```
## [1] FALSE
```

I have hidden away the test for emptiness, the extraction of the first element and the remainder of the sequence in three functions, `is_empty`, `first`, and `rest`. For a vector they can be implemented like this:

```
is_empty <- function(x) length(x) == 0
first <- function(x) x[1]
rest <- function(x) {
    if (length(x) == 1) NULL else x[2:length(x)]
}
```

A vector is empty if it has length zero. The first element is of course just the first element. The `rest` function is a little more involved. The indexing `2:length(x)` will give us the vector `2 1` if `x` has length 1, so I handle that case explicitly.

Now this search algorithm works, but you should never write code like the `rest` function I just wrote. The way we extract the rest of a vector by slicing will make R copy that sub-vector. The first time we call `rest`, we get the entire vector minus the first element, the second time we get the entire vector minus the first two, and so on. This adds up to about half the length of the vector squared. So while the search algorithm should be linear time, the way we extract the rest of a vector makes it run in quadratic time.

In practice, this doesn't matter. There is a limit in R on how deep we can go in recursive calls and we will reach that limit long before performance becomes an issue. We return to these issues at the end of the chapter, but for now let we will just, for aesthetic reasons, avoid a quadratic running time algorithm if we can make a linear time algorithm.

Languages that are built for using recursion instead of loops usually represent sequences in a different way; a way where you can get the rest of a sequence in

constant time. We can implement a version of such sequences by representing the elements in the sequence by a structure that has a `next` variable that points to the remainder of the sequence. Let us call that kind of structure a *next list*. This is an example of a *linked list*, but there are different variants of linked lists and this one just has a "next-pointer" to the rest of the sequence, so I prefer to call it a *next list*. We can translate a single element into such a sequence using this function:

```
next_list <- function(element, rest = NULL)
    list(element = element, rest = rest)
```

and construct a sequence by nested calls of the function, similarly to how we constructed a tree in the previous chapter

```
x <- next_list(1,
               next_list(2,
                         next_list(3,
                                   next_list(4))))
```

For this structure we can define the functions we need for the search algorithm like this:

```
nl_is_empty <- function(nl) is.null(nl)
nl_first <- function(nl) nl$element
nl_rest <- function(nl) nl$rest
```

and the actual search algorithm like this:

```
nl_lin_search <- function(element, sequence) {
    if (nl_is_empty(sequence))              FALSE
    else if (nl_first(sequence) == element) TRUE
    else nl_lin_search(element, nl_rest(sequence))
}
```

This works fine, and in linear time, but constructing lists is a bit cumbersome. We should write a function for translating a vector into a next list. To construct such function, we can again think recursively. If we have a vector of length

zero, the base case, then the next list should be `NULL`. Otherwise, we want to make a next list where the first element is the first element of the vector, and the rest of the list is the next list of the remainder of the vector.

```
vector_to_next_list <- function(x) {
    if (is_empty(x)) NULL
    else next_list(first(x), vector_to_next_list(rest(x)))
}
```

This works, but of course, we have just moved the performance problem from the search algorithm to the `vector_to_next_list` function. This function still needs to get the rest of a vector, and it does it by copying. The translation from vectors to next lists takes quadratic time. We need a way to get the rest of a vector without copying.

One solution is to keep track of an index into the vector. If that index is interpreted as the index where the vector really starts, we can get the rest of the vector just by increasing the index. We could use these helper functions

```
i_is_empty <- function(x, i) i > length(x)
i_first <- function(x, i) x[i]
```

and write the conversion like this:

```
i_vector_to_next_list <- function(x, i = 1) {
    if (i_is_empty(x, i)) NULL
    else next_list(i_first(x, i), i_vector_to_next_list(x, i + 1))
}
```

Of course, with the same trick we could just have implemented the search algorithm using an index.

```
i_lin_search <- function(element, sequence, i = 1) {
    if (i_is_empty(sequence, i))              FALSE
    else if (i_first(sequence, i) == element) TRUE
    else i_lin_search(element, sequence, i + 1)
}
```

Using the index implementation, we can't really write a `rest` function. Writing a function that returns a pair of a vector and an index is harder to work with than just incrementing the index itself.

When you write recursive functions on a sequence, the key abstractions you need will be checking if the sequence is empty, getting the first element, and getting the rest of the sequence. Using functions for these three operations doesn't help us unless these functions would let us work on different data types for sequences. It is possible to make such abstractions, but it is the topic for the *Object oriented programming in R* book of this series, and we will not consider it more in this book. Here will just implement the abstractions directly in our recursive functions from now on. If we do this, the linear search algorithm simply becomes.

```
lin_search <- function(element, sequence, i = 1) {
    if (i > length(sequence)) FALSE
    else if (sequence[i] == element) TRUE
    else lin_search(element, sequence, i + 1)
}
```

The structure of a recursive function

Recursive functions all follow the same pattern: figure out the base cases, that are easy to solve, and understand how you can break down the problem into smaller pieces that you can solve recursively. It is the same approach that is called "divide and conquer" in algorithm design theory. Reducing a problem to smaller problems is the hard part. There are two things to be careful about: Are the smaller problems *really* smaller? How do you combine solutions from the smaller problems to solve the larger problem?

In the linear search we have worked on so far, we know that the recursive call is looking at a smaller problem because each call is looking at a shorter sequence. It doesn't matter if we implement the function using lists or use an index into a vector, we know that when we call recursively, we are looking at a shorter sequence. For functions working on sequences, this is generally the case, and if you know that each time you call recursively you are moving closer to a base case you know that the function will eventually finish its computation.

The recursion doesn't always have to be on everything except the first element in a sequence. For a binary search, for example, we can search in logarithmic

time in a sorted sequence by reducing the problem to half the size in each recursive call. The algorithm works like this: if you have an empty sequence you can't find the element you are searching for, so you return `FALSE`. If the sequence is not empty, you check if the middle element is the element you are searching for, in which case you return `TRUE`. If it isn't, check if it is smaller than the element you are looking for, in which case you call recursively on the last half of the sequence, and if not, you call recursively on the first half of the sequence.

This sounds simple enough but first attempts at implementing this often end up calling recursively on the same sequence again and again, never getting closer to finishing. This happens if we are not careful when we pick the first or last half.

This implementation will not work. If you search for 0 or 5, you will get an infinite recursion.

```
binary_search <- function(element, x,
                          first = 1, last = length(x)) {

    if (last < first) return(FALSE) # empty sequence

    middle <- (last - first) %/% 2 + first
    if (element == x[middle]) {
        TRUE
    } else if (element < x[middle]) {
        binary_search(element, x, first, middle)
    } else {
        binary_search(element, x, middle, last)
    }
}
```

This is because you get a `middle` index that equals `first`, so you call recursively on the same problem you were trying to solve, not a simpler one.

You can solve it by never including `middle` in the range you try to solve recursively—after all, you only call the recursion if you know that `middle` is not the element you are searching for.

```
binary_search <- function(element, x,
```

```
                            first = 1, last = length(x)) {

    if (last < first) return(FALSE) # empty sequence

    middle <- (last - first) %/% 2 + first
    if (element == x[middle]) {
        TRUE
    } else if (element < x[middle]) {
        binary_search(element, x, first, middle - 1)
    } else {
        binary_search(element, x, middle + 1, last)
    }
}
```

It is crucial that you make sure that all recursive calls actually are working on a smaller problem. For sequences, that typically means making sure that you call recursively on shorter sequences.

For trees, a data structure that is fundamentally recursive—a tree is either a leaf or an inner node containing a number of children that are themselves also trees—we call recursively on sub-trees, thus making sure that we are looking at smaller problems in each recursive call.

The `node_depth` function we wrote in the first chapter is an example of this.

```
node_depth <- function(tree, name, depth = 0) {
    if (is.null(tree))     return(NA)
    if (tree$name == name) return(depth)

    left <- node_depth(tree$left, name, depth + 1)
    if (!is.na(left)) return(left)
    right <- node_depth(tree$right, name, depth + 1)
    return(right)
}
```

The base cases deal with an empty tree—an empty tree doesn't contain the node we are looking for, so we trivially return `NA`. If the tree isn't empty, we either have found the node we are looking for, in which case we can return the result. If not, we call recursively on the left tree. We return the result if we found the node we were looking for. Otherwise, we return the result of a

recursive call on the right tree (whether we found it or not, if the node wasn't in the tree at all the final result will be `NA`).

The functions we have written so far do not combine the results of the sub-problems we solve recursively. The functions are all search functions, and the result they return is either directly found or the result one of the recursive functions return. It is not always that simple, and often you need to do something with the result from the recursive call(s) to solve the larger problem.

A simple example is computing the factorial. The factorial of a number n, $n!$, is equal to $n \times (n-1)!$ with a basis case $1! = 1$. It is very simple to write a recursive function to return the factorial, but we cannot just return the result of a recursive call. We need to multiply the result we get from the recursive call with n.

```
factorial <- function(n) {
    if (n == 1) 1
    else n * factorial(n - 1)
}
```

Here I am assuming that n is an integer and $n > 0$. If it is not, the recursion doesn't move us closer to the basis case and we will (in principle) keep going forever. So here is another case where we need to be careful to make sure that when we call recursively, we are actually making progress on solving the problem. In this function, I am only guaranteeing this for positive integers.

In most algorithms, we will need to do something to the results of recursive calls to complete the function we are writing. As another example, besides `factorial`, we can consider a function for removing duplicates in a sequence. Duplicates are elements that are equal to the next element in the sequence. It is similar to the `unique` function built into R except that this function only removes repeated elements that are right next to each other.

To write it we follow the recipe for writing recursive functions. What is the base case? An empty sequence doesn't have duplicates, so the result is just an empty sequence. The same is the case for a sequence with only one element. Such a sequence does not have duplicated elements, so the result is just the same sequence. If we always know that the input to our function has at least one element, we don't have to worry about the first base case, but if the function might be called on empty sequences we need to take care of both. For sequences with more than one element, we need to check if the first element

equals the next. If it does, we should just return the rest of the sequence, thus removing a duplicated element. If it does not, we should return the first element together with the rest of the sequence where duplicates have been removed.

A solution using next lists could look like this:

```
nl_rm_duplicates <- function(x) {
    if (is.null(x)) return(NULL)
    else if (is.null(x$rest)) return(x)

    rest <- nl_rm_duplicates(x$rest)
    if (x$element == rest$element) rest
    else next_list(x$element, rest)
}

(x <- next_list(1, next_list(1, next_list(2, next_list(2)))))

## $element
## [1] 1
##
## $rest
## $rest$element
## [1] 1
##
## $rest$rest
## $rest$rest$element
## [1] 2
##
## $rest$rest$rest
## $rest$rest$rest$element
## [1] 2
##
## $rest$rest$rest$rest
## NULL

nl_rm_duplicates(x)

## $element
```

```
## [1] 1
##
## $rest
## $rest$element
## [1] 2
##
## $rest$rest
## NULL
```

To get a solution to the general problem, we have to combine the smaller solution we get from the recursive call with information in the larger problem. If the first element is equal to the first element we get back from the recursive call we have a duplicate and should just return the result of the recursive call, if not we need to combine the first element with the next list from the recursive call.

We can also implement this function for vectors. To avoid copying vectors each time we remove a duplicate we can split that function into two parts. First, we find the indices of all duplicates and then we remove these from the vector in a single operation.

```
vector_rm_duplicates <- function(x) {
    dup <- find_duplicates(x)
    x[-dup]
}
vector_rm_duplicates(c(1, 1, 2, 2))
```

```
## [1] 1 2
```

R already has a built-in function for finding duplicates, called `duplicated`, and we could implement `find_duplicates` using it (it returns a boolean vector, but we can use the function `which` to get the indices of the `TRUE` values). It is a good exercise to implement it ourselves, though.

```
find_duplicates <- function(x, i = 1) {
    if (i >= length(x)) return(c())

    rest <- find_duplicates(x, i + 1)
    if (x[i] == x[i + 1]) c(i, rest)
```

```
    else rest
}
```

The structure is very similar to the list version, but here we return the result of the recursive call together with the current index if it is a duplicate and just the result of the recursive call otherwise.

This solution isn't perfect. Each time we create an index vector by combining it with the recursive result we are making a copy so the running time will be quadratic in the length of the result (but linear in the length of the input). We can turn it into a linear time algorithm in the output as well by making a next-list instead of a vector of the indices, and then translate that into a vector in the remove duplicates function before we index into the vector `x` to remove the duplicates. I will leave that as an exercise.

As another example of a recursive function where we need to combine results from recursive calls we can consider computing the size of a tree. The base case is when the tree is a leaf. There it has size 1. Otherwise, the size of a tree is the sum of the size of its sub-trees plus one.

```
size_of_tree <- function(node) {
  if (is.null(node$left) && is.null(node$right)) {
    size <- 1
  } else {
    left_size <- size_of_tree(node$left)
    right_size <- size_of_tree(node$right)
    size <- left_size + right_size + 1
  }
  size
}
```

Here, again, I assume that either both or none of the sub-trees in a binary tree are `NULL`. I also use a slightly different approach to the function by setting the result in a local variable that I return at the end.

```
tree <- make_node("root",
                  make_node("C", make_node("A"),
                                 make_node("B")),
                  make_node("D"))
```

```
size_of_tree(tree)
```

```
## [1] 5
```

If I wanted to remember the size of sub-trees so I didn't have to recompute them, I could attempt something like this:

```
set_size_of_subtrees <- function(node) {
  if (is.null(node$left) && is.null(node$right)) {
    node$size <- 1
  } else {
    left_size <- set_size_of_subtrees(node$left)
    right_size <- set_size_of_subtrees(node$right)
    node$size <- left_size + right_size + 1
  }
  node$size
}
```

but remember that data in R cannot be changed. If I run this function on a tree, it would create nodes that knew the size of a sub-tree, but these nodes would be copies and not the nodes in the tree I call the function on.

```
set_size_of_subtrees(tree)
```

```
## [1] 5
```

```
tree$size
```

```
## NULL
```

To actually remember the sizes I would have to construct a whole new tree where the nodes knew their size. So I would need this function:

```
set_size_of_subtrees <- function(node) {
  if (is.null(node$left) && is.null(node$right)) {
    node$size <- 1
  } else {
    left <- set_size_of_subtrees(node$left)
    right <- set_size_of_subtrees(node$right)
    node$size <- left$size + right$size + 1
  }
  node
}

tree <- set_size_of_subtrees(tree)
tree$size
```

```
## [1] 5
```

Another thing we can do with a tree is to compute the depth-first-numbers of nodes. There is a neat trick for trees you can use to determine if a node is in a given sub-tree. If each node knows the range of depth-first-numbers, and we can map leaves to their depth-first-number, we can determine if it is in sub-tree just by checking if its depth-first-number is in the right range.

To compute the depth-first-numbers and annotate the tree with ranges, we need a slightly more complex function than the ones we have seen so far. It still follows the recursive function formula, though. We have a basis case for leaves and a recursive case where we need to call the function on both the left and the right sub-tree. The complexity lies only in what we have to return from the function. We need to keep track of the depth-first-numbers we have seen so far, we need to return a new node that has the range for its sub-tree, and we need to return a table of the depth-first-numbers.

```
depth_first_numbers <- function(node, dfn = 1) {
  if (is.null(node$left) && is.null(node$right)) {
    node$range <- c(dfn, dfn)
    new_table <- table
    table <- c()
    table[node$name] <- dfn
    list(node = node, new_dfn = dfn + 1, table = table)
```

```
  } else {
    left <- depth_first_numbers(node$left, dfn)
    right <- depth_first_numbers(node$right, left$new_dfn)

    new_dfn <- right$new_dfn
    new_node <- make_node(node$name, left$node, right$node)
    new_node$range <- c(left$node$range[1], new_dfn)
    table <- c(left$table, right$table)
    table[node$name] <- new_dfn
    list(node = new_node, new_dfn = new_dfn + 1, table = table)
  }
}

df <- depth_first_numbers(tree)
df$node$range

## [1] 1 5

df$table

##    A    B    C    D root
##    1    2    3    4    5
```

We can now use this depth-first-numbering to write a version of `node_depth` that only searches in the sub-tree where a node is actually found. We use a helper function for checking if a depth-first-number is in a node's range:

```
in_df_range <- function(i, df_range)
    df_range[1] <= i && i <= df_range[2]
```

and then simply check for this before we call recursively.

```
node_depth <- function(tree, name, dfn_table, depth = 0) {
    dfn <- dfn_table[name]

    if (is.null(tree) || !in_df_range(dfn, tree$range)) {
       return(NA)
```

```
    }
    if (tree$name == name) {
        return(depth)
    }

    if (in_df_range(dfn, tree$left$range)) {
        node_depth(tree$left, name, dfn_table, depth + 1)
    } else if (in_df_range(dfn, tree$right$range)) {
        node_depth(tree$right, name, dfn_table, depth + 1)
    } else {
        NA
    }
}

node_depth <- Vectorize(node_depth,
                        vectorize.args = "name",
                        USE.NAMES = FALSE)
node_depth(df$node, LETTERS[1:4], df$table)


## [1] 2 2 1 1
```

Tail-recursion

Functions, such as our search functions that return the result of recursive call without doing further computation on it are called *tail recursive*. Such functions are particularly desired in functional programming languages because they can be translated into loops, removing the overhead involved in calling functions. R, however, does not implement this tail-recursion optimisation. There are good but technical reasons why, having to do with scopes. This doesn't mean that we cannot exploit tail-recursion and the optimisations possible if we write our functions to be tail-recursive, we just have to translate our functions into loops explicitly. We cover that in the next section. First I will show you a technique for translating an otherwise not tail-recursive function into one that is.

As long as you have a function that only calls recursively zero or one time, it is a very simple trick. You pass along values in recursive calls that can be used to compute the final value once the recursion gets to a base case.

As a simple example, we can take the factorial function. The way we wrote it above was not tail-recursive. We called recursively and then multiplied n to the result.

```
factorial <- function(n) {
    if (n == 1) 1
    else n * factorial(n - 1)
}
```

We can translate it into a tail-recursive function by passing the product of the numbers we have seen so far along to the recursive call. Such a value that is passed along is typically called an accumulator. The tail-recursive function would look like this:

```
factorial <- function(n, acc = 1) {
    if (n == 1) acc
    else factorial(n - 1, acc * n)
}
```

Similarly, we can take the `find_duplicates` function we wrote and turn it into a tail-recursive function. The original function looks like this:

```
find_duplicates <- function(x, i = 1) {
    if (i >= length(x)) return(c())
    rest <- find_duplicates(x, i + 1)
    if (x[i] == x[i + 1]) c(i, rest) else rest
}
```

It needs to return a list of indices so that is what we pass along as the accumulator:

```
find_duplicates <- function(x, i = 1, acc = c()) {
    if (i >= length(x)) return(acc)
    if (x[i] == x[i + 1]) find_duplicates(x, i + 1, c(acc, i))
    else find_duplicates(x, i + 1, acc)
}
```

All functions that call themselves recursively at most once can equally easily be translated into tail-recursive functions using an appropriate accumulator.

It is harder for functions that make more than one recursive call, like the tree functions we wrote earlier. It is not impossible to make them tail-recursive, but it requires a trick called *continuation passing* which I will show you in the chapter on Higher-order Functions.

Runtime considerations

Now for the bad news. All the techniques I have shown you in this chapter for writing pure functional programs using recursion instead of loops are not actually the best way to write programs in R.

You will want to write pure functions, but relying on recursion instead of loops come at a runtime cost. We can our recursive linear search function with one that uses a `for`-loop to see how much overhead we incur.

The recursive function looked like this:

```
r_lin_search <- function(element, sequence, i = 1) {
  if (i > length(sequence)) FALSE
  else if (sequence[i] == element) TRUE
  else r_lin_search(element, sequence, i + 1)
}
```

A version using a `for`-loop could look like this:

```
l_lin_search <- function(element, sequence) {
  for (e in sequence) {
    if (e == element) return(TRUE)
  }
  return(FALSE)
}
```

We can use the function `microbenchmark` from the `microbenchmark` package to compare the two. If we search for an element that is not contained in the sequence we search in we will have to search through the entire sequence, so we can use that worst-case scenario for the performance measure.

```
library(microbenchmark)


x <- 1:1000
microbenchmark(r_lin_search(-1, x),
               l_lin_search(-1, x))


## Unit: microseconds
##                  expr     min        lq      mean
##  r_lin_search(-1, x) 1208.45 1349.9925 1683.4491
##  l_lin_search(-1, x)  197.24  215.0905  247.7755
##    median        uq      max neval
##  1495.173 1723.3685 5864.094   100
##   239.297  261.8335  466.682   100
```

The recursive function is almost an order of magnitude slower than the function that uses a loop. Keep that in mind if people tell you that loops are slow in R; they might be, but recursive functions are slower.

It gets worse than that. R has a limit to how deep you can call a function recursively, and if we were searching in a sequence longer than about a thousand elements we would reach this limit, and R would terminate the call with an error.

This doesn't mean that reading this chapter was a complete waste of time. It can be very useful to think in terms of recursion when you are constructing a function. There is a reason why divide-and-conquer is frequently used to solve algorithmic problems. You just want to translate the recursive solution into a loop once you have designed the function.

For functions such as linear search, we would never program a solution as a recursive function in the first place. The `for`-loop version is much easier to write and much more efficient. Other problems are much easier to solve with a recursive algorithm, and there the implementation is also easier done by first thinking in terms of a recursive function. The binary search is an example of such a problem. It is inherently recursive since we solve the search by another search on a shorter string. It is also less likely to hit the allowed recursion limit since it will only call recursively a number of times that is logarithmic in the length of the input, but that is another issue.

The recursive binary search looked like this:

```
r_binary_search <- function(element, x,
                            first = 1, last = length(x)) {
  if (last < first) return(FALSE) # empty sequence

  middle <- (last - first) %/% 2 + first
  if (element == x[middle]) TRUE
  else if (element < x[middle]) {
    r_binary_search(element, x, first, middle - 1)
  } else {
    r_binary_search(element, x, middle + 1, last)
  }
}
```

It is a tail-recursive function and we can exploit that to translate it into a version that uses a loop instead of recursive calls. To translate a tail-recursive function into a looping function you put the body of the function in a `repeat`-loop. A `repeat`-loop will loop forever unless you explicitly exit from it, but the base case tests in the recursive function can be used to exit the loop using an explicit `return` call. When you would normally call recursively you instead just update the local parameters you passed as arguments to the function. The result looks like this:

```
l_binary_search <- function(element, x,
                            first = 1, last = length(x)) {
  repeat {
    if (last < first) return(FALSE) # empty sequence

    middle <- (last - first) %/% 2 + first
    if (element == x[middle]) return(TRUE)

    else if (element < x[middle]) {
      last <- middle - 1
    } else {
      first <- middle + 1
    }
  }
}
```

The translation always follows this simple pattern, which is why many programming languages will do it for you automatically. We don't get as massive

a performance boost by changing this algorithm into a looping version, simply because there aren't that many function calls in a binary search—the power of logarithmic runtime algorithms—but we do get a slightly more efficient version. We can again compare the two using `microbenchmark` to measure exactly how much improvement we get:

```
x <- 1:10000000
microbenchmark(r_binary_search(-1, x),
               l_binary_search(-1, x))
```

```
## Unit: microseconds
##                     expr    min      lq     mean
##  r_binary_search(-1, x) 47.442 55.6955 69.77639
##  l_binary_search(-1, x) 35.678 39.7615 47.62955
##   median      uq     max neval
##  58.9795 80.7550 156.367   100
##  41.8440 43.7525 117.331   100
```

If your function is *not* tail-recursive it is a lot more work to translate it into a version that uses loops. You will essentially have to simulate the function call stack yourself. That involves a lot more programming, but it is not functional programming and thus is beyond the scope of this book. Not to worry, though, using continuations, a topic we cover later in the book, you can generally translate your functions into tail-recursive functions and then use a trick called a "trampoline" to replace recursion with loops.

Scope and Closures

A *scope* is something functions or expressions are associated with that tells them what values variables refer to. It is used to figure out which environment expressions are evaluated in. The same variable name can be used many places in a program, but the scope of an expression tells R exactly which variable of a given name is referred to in the expression.

A *closure* is a function with an associated scope. All functions in R have at least two different environments where they can find out what value a given variable is referring to. There is the *local* environment of the function, where function parameters and local variables can be looked up, and the *global* environment where global variables can be found. So by this definition of closure all functions in R are closures. We typically reserve the term for functions that have more than these two environments, though; functions defined inside other functions that can refer to both the local and global environment and also the environment of the enclosing function and are used outside of the enclosing function. There is nothing special about closures. They are just functions. We use them to remember environments that existed at the time they were created. If this sounds confusing right now, I hope it becomes clearer after you have read the section on scopes later in the chapter.

Scopes and environments

There are really two conceptual mappings going on when looking up a variable in an R expression. Expressions that contain variables know the variable name, not the value that the variable points to. When an expression is evaluated, the expression needs that value, so R needs to figure out what the value is. Since a variable name is not necessarily unique, it first needs to determine which of

the potentially many variables with the same name is being referred to, and then to figure out what the value that variable is pointing to.

In the code below we have two variables named `x`. One is a global variable that refers to a vector. The other is a function argument. Inside the function, we have an expression that refers to `x`. When we evaluate the function, the expression needs to figure out that the variable `x` is the function argument rather than the global variable before it can get to the value that the variable is referring to.

```
x <- 1:100
f <- function(x) sqrt(sum(x))
f(x**2)
```

When we call the function we also have an expression that refers to a variable named `x` but this is a different `x` than the variable inside the function. This `x` is the global variable, so when we evaluate the function call expression R needs to figure out that `x` refers to the global variable and then look up the value that *this* variable is referring to.

We can make this clearer by changing the names, so the variables become unique. We call the global variable `gx` and the parameter variable `px`.

```
gx <- 1:100
f <- function(px) sqrt(sum(px))
f(gx**2)
```

Because of lazy evaluation both the expression `gx**2` and the expression `sqrt(sum(px))` is actually evaluated inside function `f`. The expression being evaluated is `sqrt(sum(gx**2))`. The reason we can write the original version and let R figure out which `x` we are referring to when we have two different `x`s is that the scopes of the two `x`s are different. To evaluate the expression R needs to figure out what `gx` is pointing to, the vector `1:100`.

At the risk of causing some confusion with terms used in R, I will call the mapping from variable names to values the *environment* of an expression and the mapping from variable names to actual variables the *scope* of the expression. The risk of confusion is because if you evaluate an expression in R using the `eval` function you can provide an environment to evaluate the expression in, but this environment works both as the scope and environment. It can define

mappings from variables to values, so it is an environment, but it also changes the scope the expression is evaluated in. If a variable in the expression exists in the environment, then the expression will refer to that variable and not the variable in the scope it would otherwise refer to.

We will not be doing a lot of manipulations and evaluations of expressions in this book. It is the topic of another book in the series, *Meta-programming in R*. I will just give you a short example of what I mean here.

When you write an expression such `x + y` you have an expression where variables `x` and `y` are defined in a scope. If you evaluate the expression, you get the value of the expression using the values that the variables in the scope are referring to.

```
x <- 2 ; y <- 2
x + y
```

```
## [1] 4
```

You can also create an expression that doesn't have a scope associated. It can have variables, but these are just variable names. As long as we do not evaluate the expression, they do not even have to exist in any scope. We can create such an expression using the `quote` function.

```
quote(x + y)
```

```
## x + y
```

We can use the `eval` function to evaluate such an expression. To evaluate the expression we, of course, need both a scope, to figure out which variables the variable names refer to, and an environment that tells us which values these variables are pointing to. By default, `eval` will use the scope where you call `eval`.

If we write

```
eval(x + y)
```

```
## [1] 4
```

there is nothing special going on. Lazy-evaluation aside, the expression `x + y` already has a scope, here the global scope, and the expression will be evaluated there.

If we instead write

```
eval(quote(x + y))
```

```
## [1] 4
```

the quoted expression is put in a scope, so the variable names in the expression that were just names before now refers to variables in the scope, and the expression is then evaluated in that scope.

The result is the same in this example because the scope that is used is the same (global) scope.

To see the difference we need to provide `eval` with an environment in which to evaluate the expression.

You can create an environment using the `new.env` function and put a variable in it by assigning to a name in it using `$`.

```
env <- new.env()
env$x <- 4
```

We can also use a shortcut and create the environment and assign a value to a variable inside it using the function `lst2env`. This would look like this

```
env <- list2env(list(x = 4))
```

If we evaluate the unquoted expression in this environment, we get the same result as before. In that expression `x` and `y` already have a scope and that is what is being used.

```
eval(x + y, env)
```

```
## [1] 4
```

However, if we use the quoted expression then `env` overrides the scope we are using. When `quote(x + y)` is evaluated `eval` figures out that `x` should be found in the scope defined by `env`, and looks up the value in the corresponding environment, while `y`, which is not defined in `env`, should be found in the enclosing scope and found in that environment.

```
eval(quote(x + y), env)
```

```
## [1] 6
```

The function `eval` changes both scope and environment at the same time, but conceptually scope and environment are two different things. What `eval` considers an environment is what I describe as both scope and environment. The reason that scope and environment are conflated in `eval` is that the two things are inherently linked in R. R has an explicit representation of environments but only an implicit representation of scopes; scopes are defined by the algorithm R uses to figure out which actual variable a variable name is referring to.

Environment chains, scope, and function calls

When you call a function in R, you first create an environment for that function. That environment is where its parameters will be stored and any local variables the function assigns to will go there as well. That environment is linked to another environment. If the function is defined inside another function it will be the environment in that function instantiation; if the function is called directly from the outermost level, it will be linked to the global environment. Depending on how the function is defined there might be many such linked environment and it is this chain of environments that determines the scope used to find a variable and get its value.

We need another example to see this in action.

```
f <- function(x) 2 * x
x <- 4
f(2)
```

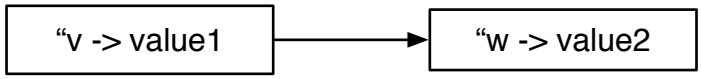

Figure 4.1: Environment chain graph

I will use the following notation to explain how the scopes and environments work: `"v` refers to a variable name. I will write environments as mappings from variable names to values like this `["v -> value]`. Environments are chained so I will write `["v -> value1] -> ["w -> value2]` to mean that I have a chain of environments where the first environment maps variable name `"v` to `value1` and the next in the chain maps variable `"w` to `value2`.

When I have need for more complex chain graphs I will use a graphical notation as shown in fig. 4.1.

If we assume that there are no variables in the global environment when we start the program, we have the global environment `[]`.[1] After we evaluate the first expression, the definition of function `f`, we have changed the global environment, so it now maps `"f` to that function.

```
["f -> function(x) 2 * x]
```

The next expression assigns a value to the variable `x` so after that the global environment is this:

```
["f -> function(x) 2 * x, "x -> 4]
```

In the third expression, we call function `f` and a lot is going on here. First R needs to figure out what the variable name `"f` is referring to. It searches in the chain of environments—in this case a chain of only one environment—and finds it in the global environment. So at this point, the scope of the variable `f` is the global environment. It can get the value from that environment, and it gets the function `function(x) 2 * x`.

[1] The global environment is actually a little more complex than the empty one we use here. It is nested inside environments where imported packages live. But for the purpose of this chapter, we do not need to worry about that.

When we call the function R creates a new environment to execute the function instance in. This environment is first empty, but it is linked to the global scope.

```
[] -> ["f -> function(x) 2 * x, "x -> 4]
```

Before any of the code in the function starts executing, though, the function parameters are put into this environment, so when we start executing the code in function `f` the environment chain looks like this:

```
["x -> 2] -> ["f -> function(x) 2 * x, "x -> 4]
```

Inside the function, we need to evaluate the expression `2 * x`. To find out how to, we first need to figure out what the variable name `"x` refers to. Here R starts searching in the chain of environments and find it in the first environment. So the scope of `"x` is the local environment; it is not the variable in the global environment that also has variable `x` defined. The result of the function call is therefore 4 rather than 8 as it would have been if `"x` was referring to the global variable `x`.

After the function returns, R removes the first environment from the chain, and future code will be evaluated in the environment chain

```
["f -> function(x) 2 * x, "x -> 4]
```

This is simple enough, but what happens if we next call the function like this?

```
f(3 * x)
```

The first steps are the same as before. R looks for `"f`, finds that it is a function, instantiate it, and creates an environment for it execute in.

```
[] -> ["f -> function(x) 2 * x, "x -> 4]
```

As before, it then puts the function parameter into this environment, but now it gets a little more complicated. Remember that R doesn't evaluate the expression used for function arguments before it calls the function. When

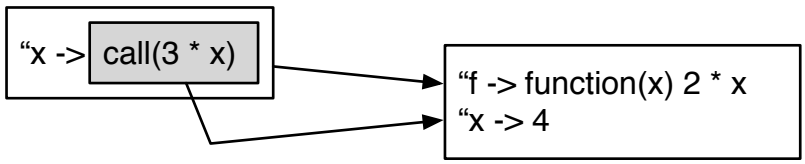

Figure 4.2: Environment chain graph

we called `f` with the value `1` it actually did pass the value along. If we had written `f(x)` it would as well, but here we call `f` with an expression and that expression is not evaluated before it is used inside the function call.

Such an expression is passed to the function in a type called “call” and such a type has its own associated environment chain. This environment chain starts in the environment where the function is *called*, not the environment inside the function. So the call object has an environment chain that starts at the global environment. So the chain (now a graph) of environments is as shown in fig. 4.2.

When we evaluate the expression inside the function, `2 * x`, R goes searching for `"x` and finds it in the first environment. It sees that it is referring to a call so to get a value it needs to evaluate this call. Because the call has its own environment chain it will use this chain to evaluate the expression. So evaluating the *call* `2 * x` it uses this chain:

```
["f -> function(x) 2 * x, "x -> 4]
```

Here it finds that the variable name `"x` is mapped to 4, so it evaluates `3 * 4` and we get the value 12 back. This is then inserted into the environment for the function call so future evaluations will refer to the value and not the call expression.

```
["x -> 12] -> ["f -> function(x) 2 * x, "x -> 4]
```

It is now in this environment we search for `"x` and find 12 that we use to evaluate `2 * x` and get 24.

We have two different variables `x` in play here. The both use the variable name `"x` but they are associated with different environment chains and are therefore in different scopes.

We can now go back to the `eval` example and see what is really going on.

```
x <- 2 ; y <- 2
env <- new.env()
env$x <- 4
eval(x + y, env)
```

The first two assignments just puts values in the global environment. After these the global environment looks like this:

```
["x -> 2, "y -> 2]
```

Then we create a new environment. By default, that environment will be linked to the current environment chain, so we are actually creating this environment chain

```
[] -> ["x -> 2, "y -> 2]
```

and we are inserting that into the global environment by assigning it to the name `env`. So we have a new global environment that knows about `env` but `env` also knows about the global environment because it has a chain to it.

```
                              .---.
["x -> 2, "y -> 2, "env -> []]<-'
```

Into the new environment we assign the value 4 to variable `x` so we now have

```
                                .--------.
["x -> 2, "y -> 2, "env -> ["x -> 4]]<-'
```

When we then call `eval`, we create yet another environment, the one for the function instantiation, and gives it the expression `x + y` together with the environment `env`. The parameter that `eval` uses to refer to the expression

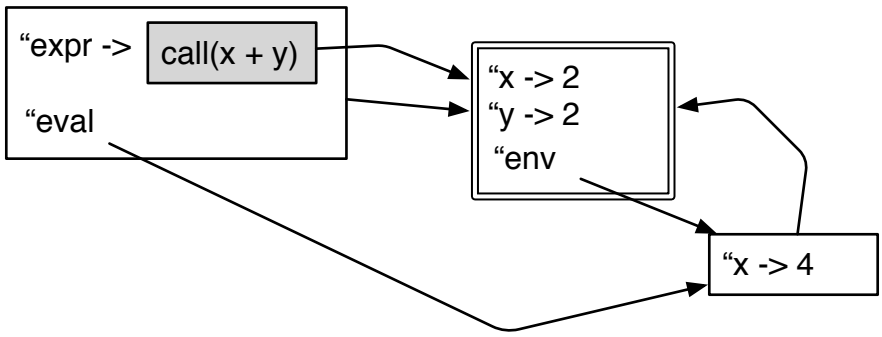

Figure 4.3: Environment chain graph in eval(x + y, env).

is called `expr`, so that is put into its environment together with `env`. (There actually is another parameter, but we ignore this in this example). So when `eval` is ready to evaluate the expression we give it, the environment chain looks like in fig. 4.3. In the figure the global environment is shown with double-strokes and the environment inside the `eval` function is shown on the left.

The `eval` function doesn't evaluate the expression inside its own environment, however, but inside the environment pointed to by its `env` parameter. So the expression is evaluated in the environment to the right in fig. 4.3.

It doesn't matter, though, whether it evaluated the expression in its own environment or in `env` because the expression is a call, with its own environment chain, consisting in this case just of the global environment, and that is the environment the call object will be evaluated in. In this environment, it finds both variable names `"x` and `"y`, and find them to refer to value 2 and 2, so that determines the result.

For the `eval` call with the quoted expression

```
eval(quote(x + y), env)
```

the situation is different. Here `eval` gets a quoted expression, not a call, so this expression does not carry its own environment along with it. The environment chain graph is shown in fig. 4.4. The `eval` function does the same thing, it

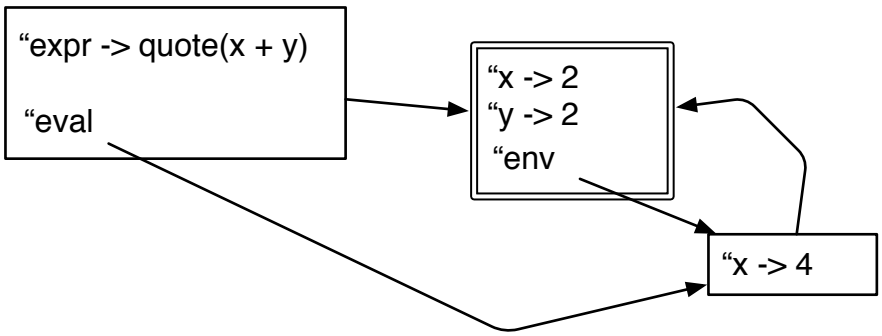

Figure 4.4: Environment chain graph in eval(quote(x + y), env).

evaluates the expression in the environment pointed to by `env`, which is the rightmost environment in the figure. So when R needs to look up variable names `"x` and `"y` it finds `"x` in the first environment and `"y` in the global environment. The `"x` in the global environment is overshadowed by the `"x` in the first environment so is not the variable in the scope of the evaluation. Thus the values used in evaluating the expression are 4 and 2.

Scopes are not static. They always depend on the environment chain expressions are evaluated in and what these environments look like at the time the expressions are evaluated. Consider the program below.

```
y <- 2
f <- function(x) {
    result <- x + y
    y <- 3
    return(result)
}
f(2)
```

```
## [1] 4
```

After evaluating the two first expressions, the assignment to `y` and `f`, we have a global environment that looks like this:

```
["y -> 2, "f -> function(x) ...]
```

When we start evaluating the body of function `f`, after the parameter has been added to its environment, the current environment chain looks like this:

```
["x -> 2] -> ["y -> 2, "f -> function(x) ...]
```

When we evaluate the `x + y` expression, R will search for these parameter names and find `"x` in the local environment and `"y` in the global environment, so the scope of `"x` is local and the scope of `"y` is global. The result is therefore 4 which is put in the local variable `result`. So now the environment chain is

```
["x -> 2, "result -> 4] -> ["y -> 2, "f -> function(x) ...]
```

We then assign the value 3 to a local variable and get the environment chain

```
["x -> 2, "result -> 4, "y -> 3] ->
    ["y -> 2, "f -> function(x) ...]
```

Now both `x` and `y` have local scope. It doesn't matter for the result, though. We have already evaluated the expression `x + y` to get the result so what we return is 4, not 6.

Situations like this don't just happen when we assign to a local variable later in a function. If we conditionally assign to a local variable, this is also in effect. In the function below, when we evaluate `x + y` the scope of these parameters depend on whether we assigned to them before we evaluated the expressions. So these variables can have local or global scope depending on which parameters we called the function with.

```
f <- function(condx, x, dondy, y) {
    if (condx) x <- 2
    if (condy) y <- 2
    x + y
}
```

So to briefly summarise how scopes and environments work in R: whenever you evaluate an expression, there is an associated chain of environments. The scope of the variables in the expression, which variables the actual variable names refer to, depends on where in this chain the variables can be found. While data is immutable in R, environments are not; whenever you assign to a variable with the `<-` operator you are modifying the top environment in the environment chain. This can not only change the value a variable refers to but also its scope. If we assign to a variable inside a function, we are only changing the environment inside that function. Other functions that might be referring to a global variable with the same name will still be referring to the global variable, not the new local variable because the environment that will be created when these functions will not be chained to the local environment where a new variable has been put.

The rules for how variables are mapped to values are always the same. It involves a search in the chain of environments that are active at the time the expression is evaluated. The only difficulty is knowing which environments are in the chain at any given time.

Here the rules are not that complex either. We can explicitly create an environment and put it at the top of the chain of environments using the `eval` function, we can create a "call" environment when we pass expressions as arguments to a function—where the environment will be the same environment chain as where we call the function—or we can create a new environment by running code inside a function.

Scopes, lazy-evaluation, and default parameters

Knowing the rules for how variables are mapped into values also helps us understand why we can use function parameters in expressions use for default parameters when defining a function, but we cannot use them when we call a function.

If we define a function with a default parameter set to an expression that refers to another parameter

```
f <- function(x, y = 2 * x) x + y
```

we can call it like this

```
f(x = 2)
```

```
## [1] 6
```

but not necessarily like this

```
f(x = 2, y = 2 * x)
```

In both cases the function body will execute in an environment chain where the parameters have been put, and in both cases `y` will refer to a call object.

```
["x -> 2, "y -> call(2 * x)] -> <global environment>
```

The difference between the two calls is which environment the call object is associated with. The difference is illustrated in fig. 4.5. The call object defined as the default parameter will be evaluated in an environment chain starting with the local function environment. The call object passed along in the function call, however, will be evaluated in the global environment because that is where we call the function from.

In the second function call, `x` will, therefore, be referring to a variable in the global environment when we evaluate the expression for `y`. If `x` is not defined there, we get an error. If `x` *is* defined there, but we meant `y` to be referring to the parameter `x` and not the global variable `x`, we have a potentially hard to debug error.

It is not an error that pops up often. I have never seen it in the wild. When people call a function, they expect the expressions in the function call to be referring to variables in the environment where the function is called, not some local variables inside the function body, and that is what they get by this semantics.

Nested functions and scopes

Whenever you instantiate a function, you create a new environment in which to execute its body. That environment is chained to the global environment in all the functions we have considered so far in this chapter, but for functions

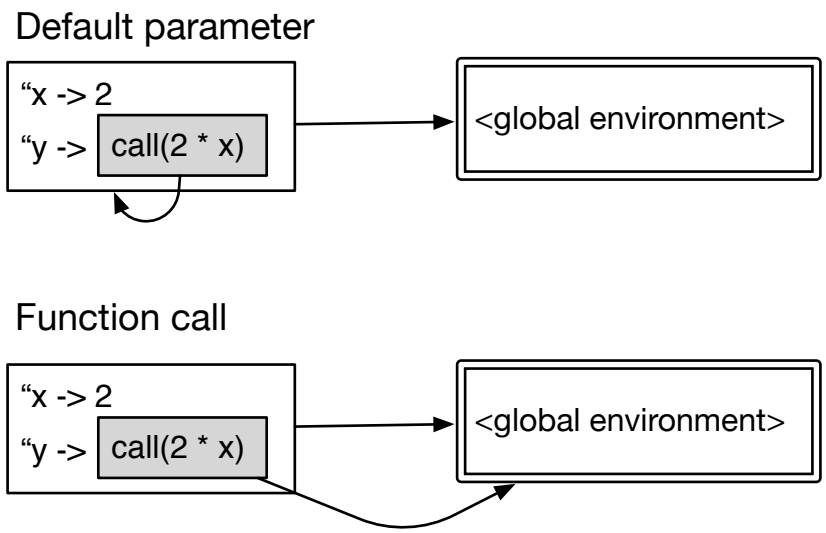

Figure 4.5: Environment chain graph in f(x = 2, y = 2 * x) and f(x = 2).

defined inside the body of other functions, the environment will instead be chained to the environment created by the outer function.

Let us first see a simple example of this. Consider the program below:

```
f <- function(x) {
    g <- function(y) x + y
    g(x)
}
f(2)
```

```
## [1] 4
```

Ignoring that there can be other variables in the global environment, the environment chain just before we call `g` inside `f` looks like this:

```
["x -> 2, "g -> function(y) x + 1] ->
    ["f -> function(x) ...]
```

Here `x` is referring to the value 2 and not a call since we passed a constant value along to `f` when we called the function.

When we call `g` we now get a new environment, as we do for all function calls and where function parameters are put before we evaluate anything else, but this environment is linked not to the global environment but the environment inside the function call to `f` where `g` was defined. So when we evaluate the expression `x + y` we have the following chain of environments

```
["y -> 2] ->
    ["x -> 2, "g -> function(y) x + 1] ->
        ["f -> function(x) ...]
```

and it is in this chain we find the variables `x` and `y` to get their values.

Here is a slightly more complex example:

```
f <- function(x) {
    g <- function(y) x + y
    g
}
h <- f(2)
h(2)
```

```
## [1] 4
```

Just before we return from the function call to `f` the environment chain looks like before

```
["x -> 2, "g -> function(y) x + 1] ->
    ["f -> function(x) ...]
```

and after we return it looks like this:

```
["f -> function(x) ..., "h -> function(y) ...]
```

We have defined two functions and assigned them to variables `f` and `h`. What happens when we then call `h`? It turns out that the environment chain, after we have put parameter variables into the new function call environment, will look like this:

```
["y -> 2] ->
    ["x -> 2, "g -> function(y) x + 1] ->
        ["f -> function(x) ...]
```

The environment from the function call to `f` is back in play. When we call the function `h` we instantiate a function, `g`, that was defined inside a call to `f`, and this function remembers that environment. When we call this function, it will chain its local environment to the environment in which it was defined, which is a local environment inside `f`.

Functions, when called, will always chain their local environment to the environment in which they were defined. There are not actually two rules for how the environments are chained together; it is just that functions defined in the global environment will be chained to that environment and functions defined in other environments will be chained to those.

All functions have an associated environment they will chain their local environments to. I just haven't shown that in the environment chains so far. From now on I will. Think of functions as similar to call objects. Call objects are actually function calls, so they behave in much the same way; functions you just have to explicitly call where call objects are evaluated when you need their value.

Just for fun, let us call `f` twice and create two functions referring to the inner function `g`.

```
h1 <- f(1)
h2 <- f(2)
```

The environment chain graph after we have defined these two functions is shown in fig. 4.6. Here functions are shown in grey, and each function points to the environment its instances will be chained to. There are three functions, `f`, `h1`, and `h2`. Even though both `h1` and `h2` were constructed from the function `g` inside `f` they were constructed in different calls to `f`, so they are different functions with different environments.

If we call `h1` we will create an environment chain that first has its local environment, then the environment created when `h1` was defined, the environment that remembers that variable `x` refers to 1, and then the global environment. If we instead call `h2` we will have the chain from the local environment to the instance of `f` where `x` was 2 and then the global environment.

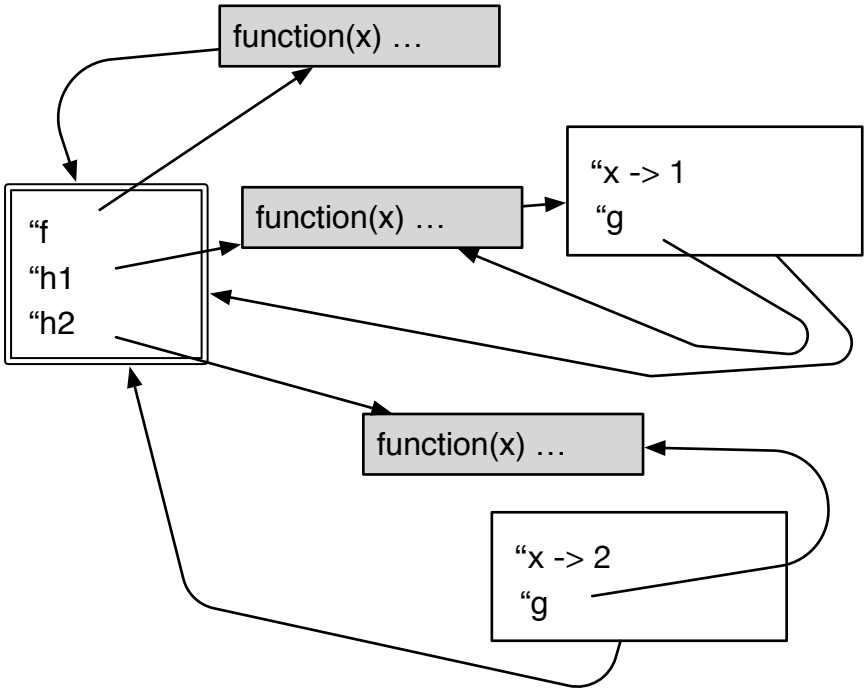

Figure 4.6: Environment chain graph after defining h1 and h2.

This environment chain graph is determined by where the functions are defined not where they are called. If we define a global function `g` that takes a function as an argument and calls that function with the value 1, we can see what happens.

```
gg <- function(ff) ff(1)
gg(h1)
```

```
## [1] 2
```

```
gg(h2)
```

```
## [1] 3
```

The global environment will know `f`, `gg`, `h1`, and `h2`. When we call `gg` we get a local environment where `ff` now refers either `h1` and `h2` depending on which call we consider. Inside the call to `gg` the environment chain looks like this

```
["ff -> <a h function>] -> <global environment>
```

where <`a h function`> refers to the functions that either `h1` or `h2`.

When `gg` then calls the (local) function `ff` we create another local environment in which to execute the function. This environment is chained to the environment the function remembers, not the local environment for `gg`. We call `gg(h1)` we need to evaluate the expression `x + y` and we will do that in an environment chain that looks like this:

```
["y -> 1] ->
    ["x -> 1, "g -> <function(y) ...>] ->
        <global environment>
```

The local environment for `gg` is nowhere in this chain. When we call `h1` through `gg` the function `h1` doesn't know anything about `gg`. It knows the environment in which it was created, the instance of `f`, and because this environment is chained to the global environment it knows about that as well. It doesn't know where it is being called from, only where it was defined.

This rule for finding variable's values, based on the environment where functions are defined, is called *lexical scoping* and is the most common standard for scopes. They are called that because you can, in principle, figure out what variables in an expression are defined where. You first check if the variables are set in the local environment, so either local variables or function parameters. If not, then you look at the enclosing code and check if the variable is in the environment. The enclosing code can be the global environment or a function. If is a function and the variables are not defined there you work your way further out in the code where the function is defined. If the mapping from variables to values depended on where a function was *called* rather than where it was defined, something called *dynamic scope*, you couldn't figure out what variables variable names were referring to just from reading the code where the function is defined.

Figuring out the variable name to variable mapping in R is not quite so easy that you can just figure it out from the code. The problem is that which

variables are defined depends on the code executed in function calls, so it is only known at runtime. We saw an example of this earlier. That is why I told you the whole complicated story rather than just the simple rule of thumb; sometimes the rule just isn't true. If you are careful and never do conditional assignments to variables without first giving them a default value, though, the rule of thumb applies.

Closures

Functions like `h1` and `h2` that remembers the environment of another function invocation are what we call closures. The term derives from *enclosing scope* and refers to the property these functions have, of remembering the enclosing environment in which they were created.

With the definition I gave at the beginning of the chapter, that closures are functions that carry a scope with them, in practise functions that have an environment chain, all functions in R are closures. But we will restrict the term to mean functions that remember an environment from a previous function instantiation that is no longer active. Because functions in R remember the environment in which they were created, the only thing that is required for a function to be a closure is that we return it from a function call.

The function `f` in the previous example creates closures. It creates an environment in which `x` is known and returns a function that adds that `x` to its input. With more sensible names, the creation of `h1` and `h2` can be written like this, making it clearer what the functions are really doing.

```
make_adder <- function(x) {
    add_y <- function(y) x + y
    add_y
}
add1 <- make_adder(1)
add2 <- make_adder(2)
```

In themselves, closures are not that useful. Making a function that takes one argument and returns another function that takes a second argument just so we can call the second function to do some operation, is just a very complex way of doing the operation; writing a function that takes two arguments is much simpler. The usefulness of closures is in combination with higher-order

functions. Higher-order functions are functions that either take other functions as arguments or return functions. A function that creates a closure is thus a higher-order function, and where closures are used is with higher-order functions that take functions as input. We return to such functions in the chapter Higher-order Functions where we will see many uses for closures.

Reaching outside your inner-most scope

When we assign to a variable using the <- operator we modify the environment at the top of the current environment chain. We modify the local environment. So what does this code do?

```
make_counter <- function() {
    x <- 0
    count <- function() {
        x <- x + 1
        x
    }
}
counter <- make_counter()
```

The intent behind the function is to create a function, a closure, that returns an increasing number each time we call it. It is of course not a pure function, but it is something we could find useful. In the depth-first-numbering algorithm, we wrote in the previous chapter we had to pass along in recursive calls the current number, but if we had such a counter we could just use *it* to get the next number each time we needed it.

It doesn't work, though.

```
counter()
```

```
## [1] 1
```

```
counter()
```

```
## [1] 1
```

```
counter()
```

```
## [1] 1
```

We can unwrap the function and see what is really going on. When we create the counter, we call the `make_counter` function, which creates an environment where `x` is set to zero and then the `count` function, which it returns.

When we call the `counter` function it knows `x` because it is a closure, so it can evaluate `x + 1` which it then assigns to `x`. This is where the problem is. The `x` used in evaluating `x + 1` is found by searching up the environment chain but the `x` the function assigns to is put in the `counter` function's environment. Then the function returns and that environment is gone. The next time `counter` is called it is a new function instance, so it is a new environment that is created. The variable `x` from the `make_counter` function is never updated.

When you use `<-` you create a new local variable if it didn't exist before. Even if the variable name is found deeper in the environment chain, it doesn't matter. The assignment always is to the local environment.

To assign to a variable deeper in the environment chain you need to use the operator `<<-` instead. This operator will search through the environment chain, the same way as R does to figure out what expressions should evaluate to, and update the environment where it finds the variable (or add it to the global environment if it doesn't find it anywhere in the environment chain).

If we change the assignment operator in the example, we can see what happens.

```
make_counter <- function() {
    x <- 0
    count <- function() {
        x <<- x + 1
        x
    }
}
counter <- make_counter()
counter()
```

```
## [1] 1
```

```
counter()
```

```
## [1] 2
```

```
counter()
```

```
## [1] 3
```

This time around, when we do the assignment we find that there is an `x` in the enclosing environment, the `x` that was initialised when we called `make_counter`, so the assignment is to that environment instead of to the local environment. Each time we call `counter` we create a new function instance environment but all the instances are linked to the same enclosing environment so each time we call the function we are updating the same environment.

We can use this counter function together with the `<<-` operator to make a much simpler version of the `depth_first_numbers` function where we do not need to pass data along in the recursive calls. We can create a table and a counter function in the outermost scope and simply use the counter and assign, with `<<-` to the table.

```
depth_first_numbers <- function(tree) {
  table <- c()
  counter <- make_counter()

  traverse_tree <- function(node) {
    if (is.null(node$left) && is.null(node$right)) {
      dfn <- counter()
      node$range <- c(dfn, dfn)
      table[node$name] <<- dfn
      node

    } else {
      left <- traverse_tree(node$left)
      right <- traverse_tree(node$right)
      new_node <- make_node(node$name, left, right)
      new_node$range <- c(left$range[1], right$range[2])
      new_node
    }
  }
```

```
  new_tree <- traverse_tree(tree)
  list(tree = new_tree, table = table)
}

result <- depth_first_numbers(tree)
print_tree(result$tree)

## [1] "((A,B),D)"

result$table

## A B D
## 1 2 3
```

We still need to create a new tree here if we want to annotate all nodes with their depth-first-number ranges, we still cannot modify data, but we can use the variables in the outer scope inside the recursive function.

Lexical scope and dynamic scope

The last thing I want to mention in this chapter is how R supports dynamic scope in addition to lexical scope. It is not something we will use further in this book, but a discussion of R's scope rules would be incomplete without it.

I will return to the `eval` example we had before.

```
x <- 2; y <- 2
eval(quote(x + y))

## [1] 4
```

Here we create a quoted expression, `x + y`, so `x` and `y` do not refer to any variables, they are just variable names, and then we evaluate that expression. In doing so, `eval` manages to find the variables to do so in the global environment. There is nothing surprising here; all functions can find the variables in the global environment.

But consider this example where we remove the global variables for `x` and `y` and call `eval` inside a function that has them as local variables:

```
rm(x); rm(y)
f <- function() {
    x <- 2; y <- 2
    eval(quote(x + y))
}
f()
```

```
## [1] 4
```

You might not be surprised that `eval` manages to do this, after all, it is what you would expect it to do, but it doesn't follow the rules I have told you about how functions know their environment chain. The `eval` function is not defined inside the `f` function so it shouldn't know about these parameters. Somehow, though, it manages to get them anyway.

This is because R supports dynamic scope as well as lexical scope. Remember, dynamic scope is where we find variables based on which functions are on the call-stack, not which functions are lexically enclosing the place where we define them.

The `eval` function manages to get the calling scope, instead of the enclosing scope, using the function `parent.frame`. Using this function, you can get to the environment of functions on the call stack.

These call-stack environments are not chained. They behave just as I have described earlier. So you cannot do this

```
g <- function(y) {
  y
  eval(quote(x + y))
}
f <- function(x) {
  g(2)
}
f(2)
```

but you can do this:

```
f <- function(x) {
  x <- x
```

```
  g <- function(y) {
    y
    eval(quote(x + y))
  }
  g(2)
}
f(2)
```

```
## [1] 4
```

To test if you have understood the environment and scope rules in R, I suggest you take a piece of paper and write down the environment chain graph for this example and work out why the first does not work but the second does.

Higher-order Functions

The term *higher-order functions* refers to functions that either take functions as arguments or return functions. The functions we used to create closures in the previous chapter are thus higher-order functions. Higher-order functions are frequently used in R instead of looping control structures, and we will cover how you can do this in general in the next chapter. I will just give you a quick example here before moving on to more interesting things we can do with functions working on functions.

The function `sapply` (it stands for simplifying apply) lets you call a function for all elements in a list or vector. You should mainly use this for interactive in R because it is a little unsafe. It tries to guess at which type of output you want, based on what the input and the function you give it does, and there are safer alternatives, `vapply` and `lapply`, where the output is always a predefined type or always a list, respectively.

In any case, if you want to evaluate a function for all elements in a vector you can write code like this:

```
sapply(1:4, sqrt)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000
```

This evaluates `sqrt` on all the element in the vector `1:4` and gives you the result. It is analogue to the vectorised expression

```
sqrt(1:4)
```

and the vectorised version is probably always preferable if your expressions can be vectorised. The vector expression is easier to read and usually much faster if it involves calling built-in functions.

You can use `sapply` to evaluate a function on all elements in a sequence when the function you want to call is not vectorised, however, and in fact, the `Vectorize` function we saw in the first chapter is implemented using `sapply`'s cousin, `lapply`.

We could implement our own version like this:

```
myapply <- function(x, f) {
  result <- x
  for (i in seq_along(x)) result[i] <- f(x[i])
  result
}

myapply(1:4, sqrt)

## [1] 1.000000 1.414214 1.732051 2.000000
```

Here I create a result vector of the right length just by copying `x`. The assignments into this vector might create copies. The first is guaranteed to because we are changing a vector, which R avoids by making a copy. Later assignments might do it again to convert the type of the results. Still, it is reasonably simple to see how it works and if the input and output have the same type it is running in linear time (in the number of function calls to `f` which of course can have any run-time complexity).

Using a function like `sapply` or `myapply`, or any of their cousins, lets us replace a loop, where all kinds of nastiness can happen with assignments to local variables that we do not necessarily want, with a single function call. We just need to wrap the body of the loop in another function we can give to the apply function.

Closures can be particularly useful in combination with apply functions if we need to pass some information along to the function. Let us, for example, consider a situation where we want to scale the element in a vector. We wrote such a function in the first chapter:

```
rescale <- function(x) {
```

```
  m <- mean(x)
  s <- sd(x)
  (x - m) / s
}
rescale(1:4)
```

```
## [1] -1.1618950 -0.3872983  0.3872983  1.1618950
```

This was written using vector expressions, which is the right thing to do, but let us see what we can do with an apply function. Obviously we still need to compute `m` and `s` and then a function for computing `(x - m) / s`.

```
rescale <- function(x) {
  m <- mean(x)
  s <- sd(x)
  f <- function(y) (y - m) / s
  myapply(x, f)
}
rescale(1:4)
```

```
## [1] -1.1618950 -0.3872983  0.3872983  1.1618950
```

There is really no need to give the function a name and then pass it along, we can also just write

```
rescale <- function(x) {
  m <- mean(x)
  s <- sd(x)
  myapply(x, function(y) (y - m) / s)
}
rescale(1:4)
```

```
## [1] -1.1618950 -0.3872983  0.3872983  1.1618950
```

The function we pass to `myapply` knows the environment in which it was created, so it knows `m` and `s` and can use these variables when it is being called from `myapply`.

Currying

It is not unusual to have a function of two arguments where you want to bind one of them and return a function that takes, as a parameter, the second argument. For example, if we want to use the function

```
f <- function(x, y) x + y
```

we might want to use it to add 2 to each element in a sequence using `myapply`. Once again I stress that this example is just to show you a general technique and you should never do these things when you can solve a problem with vector expressions, but let us for a second pretend that we don't have vector expressions. Then we would need to create a function for one of the parameters, say `y` and then when called evaluate `f(2,y)`.

We could write it like this, obviously:

```
g <- function(y) f(2, y)
myapply(1:4, g)
```

```
## [1] 3 4 5 6
```

which would suffice if we only ever needed to add 2 to all elements in a sequence. If we wanted a more general solution, we would want a function to create the function we need, a closure that remembers `x` and can be called with `y`. Such a function is only slightly more complicated to write, we simply need a function that returns a function like this:

```
h <- function(x) function(y) f(x, y)
myapply(1:4, h(2))
```

```
## [1] 3 4 5 6
```

The function `f` and the function `h` eventually do the same thing, the only difference is in how we provide parameters to them. Function `f` needs both parameters at once while function `h` takes one parameter at a time.

```
f(2, 2)
```

```
## [1] 4
```

```
h(2)(2)
```

```
## [1] 4
```

A function such as `h`, one that takes a sequence of parameters not in a single function call but through a sequence of function calls, each taking a single parameter and returning a new function, is known as a *curried* function. What we did to transform `f` into `h` is called *currying* `f`. The names refer to the logician Haskell Curry, from whom we also get the name of the functional programming language Haskell.

The transformation we did from `f` to `h` was manual, but we can write functions that transform functions: functions that take a function as input and return another function. We can thus write a general function for currying a function. Another high-level function. A function for currying functions of two parameters can be written like this:

```
curry2 <- function(f)
  function(x) function(y) f(x, y)
```

The argument `f` is the function we want to curry and the return value is the function

```
function(x) function(y) f(x, y)
```

that needs to be called first with parameter `x` and then with parameter `y` and then it will return the value `f(x, y)`. The name of the variables do not matter here, they are just names and need not have anything to do with the names of the variables that `f` actually takes.

Using this function we can automatically create `h` from `f`:

```
h <- curry2(f)
f(2, 3)
```

```
## [1] 5
```

```
h(2)(3)
```

```
## [1] 5
```

Since `+` is just a function in R we can also simply use *it* instead of writing the function `f` first

```
h <- curry2(‘+‘)
h(2)(3)
```

```
## [1] 5
```

and thus write the code that adds 2 to all elements in a sequence like this:

```
myapply(1:4, curry2(‘+‘)(2))
```

```
## [1] 3 4 5 6
```

Whether you find this clever or too terse to be easily understood is a matter of taste. It is clearly not as elegant as vector expressions, but once you are familiar with what currying does it is not difficult to parse either.

The function we wrote for currying only works on functions that take exactly two arguments. We explicitly created a function that returns a function that returns the final value, so the curried version has to work on functions with two arguments. But functions are data that we can examine in R so we can create a general version that can handle functions of any number of arguments. We simply need to know the number of arguments the function takes and then create a sequence of functions for taking each parameter.

The full implementation is shown below, and I will explain it in detail after the function listing.

```
curry <- function(f) {
  n <- length(formals(f))
  if (n == 1) return(f) # no currying needed

  arguments <- vector("list", length = n)
  last <- function(x) {
    arguments[n] <<- x
    do.call(f, arguments)
  }
  make_i <- function(i, continuation) {
    force(i) ; force(continuation)
    function(x) {
      arguments[i] <<- x
      continuation
    }
  }

  continuation <- last
  for (i in seq(n-1, 1)) {
    continuation <- make_i(i, continuation)
  }
  continuation
}
```

First we get the number of arguments that function `f` takes.

```
  n <- length(formals(f))
```

We can get these arguments using the `formals` function. It returns what is called a pair-list, which is essentially a linked list similar to the one we saw in *Pure Functional Programming*, and is used internally in R but not in actual R programming. You can treat it as a list, and it behaves the same way except for some runtime performance differences. We just need to know the length of the list.

We just return if `f` only takes a single argument. Then it is already as curried as it can get. Otherwise we create a table in which we will store variables when the chain of functions are called.

```
  arguments <- vector("list", length = n)
```

- We need to collect the arguments that are passed to the individual functions we create. We cannot simply use parameter arguments `x` and `y` as we did in `curry2` because we do not know how many arguments we would need before we have examined the input function, `f`. In any case, we would need to create names dynamically to make sure they don't clash. Just saving the arguments in a list is simpler, and since it is a list, we can put any kind of values that are passed as arguments in it.

Now we need to create the actual function we should return. We do this in steps. The final function we create should call `f` with all the arguments. It takes the last argument as input, so we need to store that in `arguments`—and since this means modifying a list outside of the scope of the actual function, we need the `<<-` assignment operator for that. To call `f` with a list of its arguments we need to use the function `do.call`. It lets us specify the arguments to the function in a list instead of giving them directly as comma-separated arguments.

```
last <- function(x) {
  arguments[n] <<- x
  do.call(f, arguments)
}
```

Each of the other functions needs to store an argument as well so they will need to know the corresponding index, and then they should return the next function in the curried chain. We can create such functions like this:

```
make_i <- function(i, continuation) {
  force(i) ; force(continuation)
  function(x) {
    arguments[i] <<- x
    continuation
  }
}
```

The parameter `i` is just the index, which we use to assign a value to an argument, and the parameter `continuation` is the functions that still remains to be called before the final result can be returned from function `last`. It is the continuation of the curried function. We need to evaluate both parameters

inside `make_i` before we return the function. Otherwise, we run into the lazy-evaluation problem where, when eventually call the function, values of variables might have changed since we created the function. We do this by calling `force` on the arguments.

We now simply need to bind it all together. We do this in reverse order, so we always have the continuation we need when we create the next function. The first continuation is the `last` function. It is the last function that should be called, and it will return the desired result of the entire curried chain. Each call to `make_i` will return a new continuation that we provide to the next function in the chain (in reverse order).

```
  continuation <- last
  for (i in seq(n-1, 1)) {
    continuation <- make_i(i, continuation)
  }
```

The final continuation we create is the curried function, so that is what we return at the end of `curry`.

Now we can use this `curry` function to translate any function into a curried version:

```
f <- function(x, y, z) x + 2*y + 3*z
f(1, 2, 3)
```

```
## [1] 14
```

```
curry(f)(1)(2)(3)
```

```
## [1] 14
```

It is not *quite* the same semantics as calling `f` directly; we are evaluating each expression when we assign it to the `arguments` table, so lazy-evaluation isn't in play here. To write a function that can deal with the lazy evaluation of the parameters requires a lot of mucking about with environments and expressions that is beyond the scope of this book, but that I cover in the *Meta-programming in R* book later. Aside from that, though, we have written a general function for translating normal functions into their curried form.

It is *much* harder to make a transformation in the other direction automatically. If we get a function, we cannot see how many times we would need to call it to get a value, if that is even independent of the parameters we give it, so there is no way to figure out how many parameters we should get for the uncurried version of a curried function.

The `curry` function isn't completely general. We cannot deal with default arguments—all arguments must be provided in the curried version—and we cannot create a function where some arguments are fixed, and others are not. The curried version always needs to take the arguments in the exact order the original function takes them.

A parameter binding function

We already have all the tools we need to write a function that binds a set of parameters for a function and return another function where we can give the remaining parameters and get a value.

```
bind_parameters <- function(f, ...) {
  remembered <- list(...)
  function(...) {
    new <- list(...)
    do.call(f, c(remembered, new))
  }
}

f <- function(x, y, z, w = 4) x + 2*y + 3*z + 4*w

f(1, 2, 3, 4)
```

```
## [1] 30
```

```
g <- bind_parameters(f, y = 2)
g(x = 1, z = 3)
```

```
## [1] 30
```

```
h <- bind_parameters(f, y = 1, w = 1)
f(2, 1, 3, 1)
```

```
## [1] 17
```

```
h(x = 2, z = 3)
```

```
## [1] 17
```

We get the parameters from the first function call and saves them in a list, and then we return a closure that can take the remaining parameters, turn them into a list, combine the remembered and the new parameters and call function `f` using `do.call`.

All the building blocks we have seen before, we are just combining them to translate one function into another.

Using `list` here to remember the parameters from `...` means that we are evaluating the parameters. We are explicitly turning off lazy-evaluation in this function. It is possible to keep the lazy evaluation semantics as well, but it requires more work. We would need to use the `eval(substitute(alist(...)))` trick to get the unevaluated parameters into a list—we saw this trick in the first chapter—but that would give us raw expressions in the lists, and we would need to be careful to evaluate these in the right environment chain to make it work. I leave this as an exercise to the reader, or you can look at the `partial` function from the `pryr` package to see how it is done.

Such partial binding functions aren't used that often in R. It is just as easy to write closures to bind parameters and those are usually easier to read, so use partial bindings with caution.

Continuation-passing style

The trick we used to create `curry` involved creating a chain of functions where each function returns the next function that should be called, the *continuation*. This idea of having the remainder of a computation as a function you can eventually call can be used in many other problems.

Consider the simple task of adding all elements in a list. We can write a function for doing this in the following three ways:

```
my_sum_direct <- function(lst) {
  if (is_empty(lst)) 0
  else first(lst) + my_sum_direct(rest(lst))
}
my_sum_acc <- function(lst, acc = 0) {
  if (is_empty(lst)) acc
  else my_sum_acc(rest(lst), first(lst) + acc)
}
my_sum_cont <- function(lst, cont = identity) {
  if (is_empty(lst)) cont(0)
  else my_sum_cont(rest(lst),
                   function(acc) cont(first(lst) + acc))
}
```

The first function handles the computation in an obvious way by adding the current element to the result of the recursive call. The second function uses an accumulator to make a tail-recursive version, where the accumulator carries a partial sum along with the recursion. The third version also gives us a tail-recursive function but in this case via a continuation function. This function works as the accumulator in the second function, it just wraps the computation inside a function that is passed along in the recursive call.

Here, the continuation captures the partial sum moving down the recursion—the same job as the accumulator has in the second function—but expressed as an as-yet not evaluated function. This function will eventually be called by the sum of values for the rest of the recursion, so the job at this place in the recursion is simply to take the value it will eventually be provided, add the current value, and then call the continuation it was passed earlier to complete the computation.

For something as simple as a adding the numbers in a list, continuation passing is of course overkill. If you need tail-recursion, the accumulator version is simpler and faster, and in any case, you are just replacing recursion going down the vector with function calls in the continuation moving up again (but see later for a solution to this problem). Still, seeing the three approaches to recursion—direct, accumulator and continuation-passing—in a trivial example makes it easier to see how they work and how they differ.

A common use of continuations is to translate non-tail-recursive functions into tail-recursive. As an example, we return to the function from *Pure Functional Programming* that we used to compute the size of a tree. In that solution we

needed to handle internal nodes by first calling recursively on the left subtree and then the right subtree, to get the sizes of these, and then combine them and adding one for the internal node. Because we needed the results from two recursive calls, we couldn't directly make the function tail-recursive. Using continuations we can.

The trick is to pass a continuation along that is used to wrap one of the recursions while we can handle the other recursion in a tail-recursive call. The solution looks like this:

```
size_of_tree <- function(node, continuation = identity) {
  if (is.null(node$left) && is.null(node$right)) {
    continuation(1)
  } else {
    new_continuation <- function(left_result) {
      continuation(left_result + size_of_tree(node$right) + 1)
    }
    size_of_tree(node$left, new_continuation)
  }
}

size_of_tree(tree)

## [1] 5
```

The function takes a continuation along in its call, and this function is responsible for computing "the rest of what needs to be done". If the node we are looking at is a leaf, we call it with 1, because a leaf has size one, and then it will do whatever computation is needed to compute the size of the tree we have seen earlier and wrapped in the `continuation` parameter.

The default continuation is the identity function

```
function(x) x
```

so if we just call the function with a leaf we will get 1 as the value. But we modify the continuation when we see an internal node to wrap some of the computation we cannot do yet because we do not have the full information for what we need to compute.

For internal nodes we do this:

```
    new_continuation <- function(left_result) {
      continuation(left_result + size_of_tree(node$right) + 1)
    }
    size_of_tree(node$left, new_continuation)
```

We create a continuation that, if we give it the size of the left subtree, can compute the size of the full tree by calling recursively on the right subtree and then adding the size of the left subtree. That wraps up the computations we need to do with the right subtree, so we only need to call recursively on the left subtree and that we can do with a tail-recursive call.

Because it is tail-recursive, we can replace the recursive calls with a loop. This time around we need to remember to `force` the evaluation of the continuation when we create the new continuation because when we loop we are modifying local parameters. Otherwise, it looks like you would expect from the general pattern for translating tail-recursive functions into looping functions.

```
size_of_tree <- function(node) {
  continuation <- identity # function(x) x
  repeat {
    if (is.null(node$left) && is.null(node$right)) {
      return(continuation(1))
    }
    new_continuation <- function(continuation) {
      force(continuation)
      function(left_result) {
        continuation(left_result + size_of_tree(node$right) + 1)
      }
    }
    # simulated recursive call
    node <- node$left
    continuation <- new_continuation(continuation)
  }
}

size_of_tree(tree)
```

```
## [1] 5
```

There is a catch, though. We avoid deep recursions to the left, but the continuation we create is going to call functions just as deep as we would earlier do the recursion. Each time we would usually call recursively, we are wrapping a function inside another, and when these need to be evaluated, we still have a deep number of function calls.

There is a trick to get around this. It won't give us the performance boost of using a loop instead of recursions, it will actually be slightly slower, but it will let us write functions with more than one recursion without having too deep recursive calls.

Thunks and trampolines

There are two pieces to the solution of too deep recursions. The first is something called a "thunk". It is simply a function that takes no arguments and returns a value. It is used to wrap up a little bit of computation that you can evaluate later. We can turn a function, with its arguments, into a thunk like this:

```
make_thunk <- function(f, ...) {
  force(f)
  params <- list(...)
  function() do.call(f, params)
}
```

We force the function parameter, `f`, just in case—we don't want it to change if it refers to an expression that might change after we have defined the thunk. Then we remember the parameters to the function—this evaluates the parameters, so no lazy evaluation here (it is much harder to keep track of the thunk if we need to keep the evaluation lazy), and then we return a function with no arguments that simply evaluates `f` on the remembered parameters.

Now we can turn any function into a thunk:

```
f <- function(x, y) x + y
thunk <- make_thunk(f, 2, 2)
thunk()
```

```
## [1] 4
```

If you are wondering why such functions are called "thunks", here is what the Hackers Dictionary has to say:

> Historical note: There are a couple of onomatopoeic myths circulating about the origin of this term. The most common is that it is the sound made by data hitting the stack; another holds that the sound is that of the data hitting an accumulator. Yet another holds that it is the sound of the expression being unfrozen at argument-evaluation time. In fact, according to the inventors, it was coined after they realized (in the wee hours after hours of discussion) that the type of an argument in Algol-60 could be figured out in advance with a little compile-time thought, simplifying the evaluation machinery. In other words, it had "already been thought of"; thus it was christened a thunk, which is "the past tense of 'think' at two in the morning". – The Hackers Dictionary[1].

We are going to wrap recursive calls into thunks where each thunk takes one step in a recursion. Each thunk evaluates one step and returns a new thunk that will evaluate the next step, and so on until a thunk eventually returns a value. The term for such evaluations is a "trampoline", and the imagery is that each thunk bounce on the trampoline, evaluates one step, and lands on the trampoline again as the next thunk.

A trampoline is just a function that keeps evaluating thunks until it gets a value, and the implementation looks like this:

```
trampoline <- function(thunk) {
  while (is.function(thunk)) thunk <- thunk()
  thunk
}
```

To see how thunks and trampolines can be combined to avoid recursion we will first consider the simpler case of calculating the factorial of a number instead of the size of a tree.

We wrote the recursive factorial function in *Pure Functional Programming* and the non-tail-recursive version looked like this:

[1] http://www.hacker-dictionary.com/terms/thunk

```
factorial <- function(n) {
  if (n == 1) 1
  else n * factorial(n - 1)
}
```

and the tail-recursive version, using an accumulator, looked like this:

```
factorial <- function(n, acc = 1) {
  if (n == 1) acc
  else factorial(n - 1, acc * n)
}
```

To get the thunk-trampoline version, we are first going to rewrite this using continuation passing. This is mostly just turning the accumulator in the tail-recursive version into a continuation for computing the final result:

```
cp_factorial <- function(n, continuation = identity) {
  if (n == 1) {
    continuation(1)
  } else {
    new_continuation <- function(result) {
      continuation(result * n)
    }
    cp_factorial(n - 1, new_continuation)
  }
}

factorial(10)

## [1] 3628800

cp_factorial(10)

## [1] 3628800
```

This function does the same as the accumulator version, and because there is no tail-recursion optimisation it will call `cp_factorial` all the way down from

`n` to `1` and then it will evaluate continuation functions just as many times. We can get it to work for n maybe up to a thousand or so, but after that, we hit the recursion stack limit. Before we reach that limit, the number will be too large to represent as floating point numbers in R anyway, but that is not the point; the point is that the number of recursive calls can get too large for us to handle.

So instead of calling recursively we want each "recursive" call to create a thunk instead. This will create a thunk that does the next step and returns a thunk for the step after that, but it will not call the next step, so no recursion. We need such thunks both for the recursions and the continuations. The implementation is simple, we just replace the recursions with calls to `make_thunk`:

```
thunk_factorial <- function(n, continuation = identity) {
  if (n == 1) {
    continuation(1)
  } else {
    new_continuation <- function(result) {
      make_thunk(continuation, n * result)
    }
    make_thunk(thunk_factorial, n - 1, new_continuation)
  }
}
```

Calling this function with 1 directly gives us a value:

```
thunk_factorial(1)
```

```
## [1] 1
```

Calling it with 2 creates a thunk. We need to call this thunk to move down the recursion to the base case, this will give us a thunk for the continuation there, and we need to evaluate that thunk to get the value:

```
thunk_factorial(2)()()
```

```
## [1] 2
```

For each additional step in the recursion we thus get two more thunks, one for going down the recursion and the next for evaluating the thunk, but eventually, we will have evaluated all the thunks and will get a value.

```
thunk_factorial(3)()()()()
```

```
## [1] 6
```

```
thunk_factorial(4)()()()()()()
```

```
## [1] 24
```

```
thunk_factorial(5)()()()()()()()()
```

```
## [1] 120
```

Of course, we don't want to call all these thunks explicitly, that is what the trampoline is for.

```
trampoline(thunk_factorial(100))
```

```
## [1] 9.332622e+157
```

We can write another higher-order function for translating such a thunk-enized function into one that uses the trampoline to do the calculation like this:

```
make_trampoline <- function(f) function(...) trampoline(f(...))
factorial <- make_trampoline(thunk_factorial)
factorial(100)
```

```
## [1] 9.332622e+157
```

For computing the size of a tree we just do exactly the same thing. It doesn't matter that the continuation we use here does something more complex—it calls the depth-first traversal on the right subtree instead of just computing an expression directly—because it is just a continuation and we just need to wrap it up as a thunk:

```
thunk_size <- function(node, continuation = identity) {
  if (is.null(node$left) && is.null(node$right)) {
    continuation(1)
  } else {
    new_continuation <- function(left_result)
      make_thunk(continuation,
                 left_result + thunk_size(node$right) + 1)
    make_thunk(thunk_size, node$left, new_continuation)
  }
}

size_of_tree <- make_trampoline(thunk_size)
size_of_tree(tree)
```

```
## [1] 5
```

The way we make the trampoline version is *exactly* the same as what we did for the factorial function. We make a continuation passing version of the recursion, then we translate the direct recursive calls into thunks, and we make our continuations return thunks. Using the trampoline, we never run into problems with hitting the call stack limit; we never call recursively we just create thunks on the fly whenever we would otherwise need to call a function.

Isn't this just mind-boggling clever?

Filter, Map, and Reduce

The last chapter covered some pieces of functional programming that can be hard to wrap your head around, but this chapter will be much simpler. We will just look at three general methods that are used in functional programming instead of loops, and instead of explicitly writing recursive functions. They are really three different patterns for computing on sequences, and they come in different flavours in different functions, but just these three lets you do almost anything you would otherwise do with loops.

Note the *almost* above. These three functions do not replace *everything* you can do with loops. You can replace `for`-loops, where you already know how much you are looping over, but they cannot substitute `while`- and `repeat`-loops. Still, by far the most loops you write in R are `for`-loops, and in general, you can use these functions to replace those.

The functions, or patterns, are `Filter`, `Map`, and `Reduce`. The first takes a sequence and a predicate, a function that returns a boolean value, and it returns a sequence where all elements where the predicate was true are included, and the rest are removed. The second, `Map`, evaluates a function on each item in a sequence and returns a sequence with the results of evaluating the function. It is similar to the `sapply` function we briefly saw in the previous chapter. The last, `Reduce`, takes a sequence and a function and evaluates the function repeatedly to reduce the sequence to a single value. This pattern is also called "fold" in some programming languages.

The general sequence object in R is a list

Sequences come in two flavours in R, vectors and lists. Vectors can only contain basic types and all elements in a vector must have the same type. Lists can

contain a sequence of any type and the elements in a list can have different types. Lists are thus more general than vectors and are often the building blocks of data structures such as the "next lists" and the trees we have used earlier in the book.

It, therefore, comes as no surprise that general functions for working on sequences would work on lists. The three functions, `Filter`, `Map`, and `Reduce` are also happy to take vectors, but they are treated just as if you explicitly converted them to lists first. The `Reduce` function returns a value, so not a sequence, of a type that depends on its input, while `Filter` and `Map` both return sequences in the form of a list.

From a programming perspective, it is just as easy to work with lists as it is to work with vectors, but some functions do expect vectors—plotting functions and functions for manipulating data frames for example—so sometimes you will have to translate a list from `Filter` or `Map` into a vector. You can do this with the function `unlist`. This function will convert a list into a vector when this is possible, that is when all elements are of the same basic type, and otherwise will just give you the list back. I will use `unlist` in many examples in this chapter just because it makes the output nicer to look at, but in most programs, I do not bother doing so until I really need a vector. A list is just as good for storing sequences.

It is just that

```
list(1, 2, 3, 4)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 4
```

gives us much longer output listings to put in the book than

```
1:4
```

```
## [1] 1 2 3 4
```

If you follow along in front of your compute you can try to see the results with and without `unlist` to get a feeling for the differences.

You rarely need to convert sequences the other way, from vectors to lists. Functions that work on lists usually also work on vectors, but if you want to you should use the `as.list` function and not the `list` function. The former gives you a list with one element per element in the vector

```
as.list(1:4)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 4
```

whereas the latter gives you a list with a single element that contains the vector

```
list(1:4)
```

```
## [[1]]
## [1] 1 2 3 4
```

Filtering sequences

The `Filter` function is the simplest of the three main functions we cover in this chapter. It simply selects a subset of a sequence based on a predicate. A predicate is a function that returns a single boolean value, and `Filter` will return a list of elements where the predicate returned `TRUE` and discard the elements where the predicate returned `FALSE`.

```
is_even <- function(x) x %% 2 == 0
unlist(Filter(is_even, 1:10))
```

```
## [1]  2  4  6  8 10
```

The function is often used together with closures so the predicate can depend on local variables

```
larger_than <- function(x) function(y) y > x
unlist(Filter(larger_than(5), 1:10))
```

```
## [1]  6  7  8  9 10
```

and of course works with the `curry` functions we wrote earlier

```
unlist(Filter(curry2(`<`)(5), 1:10))
```

```
## [1]  6  7  8  9 10
```

```
unlist(Filter(curry2(`>=`)(5), 1:10))
```

```
## [1] 1 2 3 4 5
```

Using `curry2` with a binary operator like here can look a little confusing, though. We have the left-hand-side of the operator immediately to the right of the operator, so the casual reader would expect `curry2(<)(5)` to pick numbers less than five while in fact it does the opposite since `curry2(<)(5)` translates to `function(y) 5 < y`. We can easily fix this by reversing the order of arguments in the curry function:

```
rcurry2 <- function(f) function(y) function(x) f(x, y)
unlist(Filter(rcurry2(‘>=‘)(5), 1:10))
```

```
## [1]  5  6  7  8  9 10
```

```
unlist(Filter(rcurry2(‘<‘)(5), 1:10))
```

```
## [1] 1 2 3 4
```

Here we have used a vector as input to `Filter`, but any list will do and we do not need to limit it to sequences of the same type.

```
s <- list(a = 1:10, b = list(1,2,3,4,5,6),
          c = y ~ x1 + x2 + x3, d = vector("numeric"))
Filter(function(x) length(x) > 5, s)
```

```
## $a
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $b
## $b[[1]]
## [1] 1
##
## $b[[2]]
## [1] 2
##
## $b[[3]]
## [1] 3
##
## $b[[4]]
## [1] 4
##
## $b[[5]]
## [1] 5
##
## $b[[6]]
## [1] 6
```

When printed, the result isn't pretty, but we can't solve that with `unlist` in this case. Using `unlist` we *would* get a vector, but not remotely one reflecting the structure of the result, the vector `a` and list `b` would be flattened into a single vector.

Mapping over sequences

The `Map` function evaluates a function for each element in a list and returns a list with the results.

```
unlist(Map(is_even, 1:5))
```

```
## [1] FALSE  TRUE FALSE  TRUE FALSE
```

As with `Filter`, `Map` is usually combined with closures

```
add <- function(x) function(y) x + y
unlist(Map(add(2), 1:5))
```

```
## [1] 3 4 5 6 7
```

```
unlist(Map(add(3), 1:5))
```

```
## [1] 4 5 6 7 8
```

and can be applied on lists of different types

```
s <- list(a = 1:10, b = list(1,2,3,4,5,6),
          c = y ~ x1 + x2 + x3, d = vector("numeric"))
unlist(Map(length, s))
```

```
##  a  b  c  d
## 10  6  3  0
```

`Map` can be applied to more than one sequences if the function you provide it takes a number of parameters that matches the number of sequences:

```
unlist(Map(‘+‘, 1:5, 1:5))
```

```
## [1]  2  4  6  8 10
```

In this example we use the function `+`, which takes two arguments, and we give the `Map` function two sequences, so the result is the component-wise addition.

You can pass along named parameters to a `Map` call, either directly as a named parameter

```
x <- 1:10
y <- c(NA, x)
s <- list(x = x, y = y)
unlist(Map(mean, s))
```

```
##   x   y
## 5.5  NA
```

```
unlist(Map(mean, s, na.rm = TRUE))
```

```
##   x   y
## 5.5 5.5
```

or as a list provided to the `MoreArgs` parameter.

```
unlist(Map(mean, s, MoreArgs = list(na.rm = TRUE)))
```

```
##   x   y
## 5.5 5.5
```

For a single value, the two approaches work the same, but their semantics is slightly different, which comes into play when providing arguments that are sequences. Providing a named argument directly to `Map` works just as providing an unnamed argument (except that you can pick a specific variable by name instead of by position), so `Map` assumes that you want to apply your function to every element of the argument. The reason this works with a single argument is that, as R generally does, the shorter sequence is repeated as

many times as needed. With a single argument that is exactly what we want, but it isn't necessarily with a sequence.

If we want that behaviour we can just use the named argument to `Map` but if we want the function to be called with the entire sequence each time it is called we must put the sequence as an argument to the `MoreArgs` parameter.

As an example we can consider our trusted friend the `scale` function and make a version where a vector, `x`, is scaled by the mean and standard deviation of another vector, `y`.

```
scale <- function(x, y) (x - mean(y))/sd(y)
```

If we just provide `Map` with two arguments for `scale` it will evaluate all pairs independently (and we will get a lot of `NA` values because we are calling the `sd` function on a single value.

```
unlist(Map(scale, 1:10, 1:5))
```

```
##  [1] NA NA NA NA NA NA NA NA NA NA
```

The same happens if we name parameter `y`

```
unlist(Map(scale, 1:10, y = 1:5))
```

```
##  [1] NA NA NA NA NA NA NA NA NA NA
```

but if we use `MoreArgs` the entire vector `y` is provided to `scale` in each call.

```
unlist(Map(scale, 1:10, MoreArgs = list(y = 1:5)))
```

```
##  [1] -1.2649111 -0.6324555  0.0000000  0.6324555
##  [5]  1.2649111  1.8973666  2.5298221  3.1622777
##  [9]  3.7947332  4.4271887
```

Just as `Filter`, `Map` is not restricted to work on vectors, so we can map over arbitrary types as long as our function can handle the different types.

```
s <- list(a = 1:10, b = list(1,2,3,4,5,6),
          c = y ~ x1 + x2 + x3, d = vector("numeric"))
unlist(Map(length, s))
```

```
##  a  b  c  d
## 10  6  3  0
```

Reducing sequences

While `Filter` and `Map` produces lists, the `Reduce` function transforms a list into a value. Of course, that value can also be a list, lists are also values, but `Reduce` doesn't simply process each element in its input list independently. Instead, it summarises the list by applying a function iteratively to pairs. You provide it a function, `f` of two elements, and it will first call `f` on the first two elements in the list. Then it will take the result of this and call `f` with this and the next element, and continue doing that through the list.

So calling `Reduce(f, 1:5)` will be equivalent to calling

```
f(f(f(f(1, 2), 3), 4), 5)
```

It is just more readable to write `Reduce(f, 1:5)`.

We can see it in action using `` `+` `` as the function:

```
Reduce(`+`, 1:5)
```

```
## [1] 15
```

You can also get the step-wise results back by using the parameter `accumulate`. This will return a list of all the calls to `f` and include the first value in the list, so `Reduce(f, 1:5)` will return the list

```
c(1, f(1, 2), f(f(1 ,2), 3), f(f(f(1, 2), 3), 4),
   f(f(f(f(1, 2), 3), 4), 5))
```

So for addition we get:

```
Reduce(‘+‘, 1:5, accumulate = TRUE)
```

```
## [1]  1  3  6 10 15
```

By default `Reduce` does its computations from left to right, but by setting the option `right` to `TRUE` you instead get the results from right to left.

```
Reduce(‘+‘, 1:5, right = TRUE, accumulate = TRUE)
```

```
## [1] 15 14 12  9  5
```

For an associative operation like `` `+` ``, this will, of course, be the same result if we do not ask for the accumulative function calls.

In many functional programming languages, which all have this function although it is sometimes called `fold` or `accumulate`, you need to provide an initial value for the computation. This is then used in the first call to `f`, so the folding instead starts with `f(init, x[1])` if `init` refers to the initial value and `x` is the sequence.

You can also get that behaviour in `R` by explicitly giving `Reduce` an initial value through parameter `init`:

```
Reduce(‘+‘, 1:5, init = 10, accumulate = TRUE)
```

```
## [1] 10 11 13 16 20 25
```

You just don't need to specify this initial value that often. In languages that require it, it is used to get the right starting points when accumulating results. For addition, we would use zero as an initial value if we want `Reduce` to compute a sum because adding zero to the first value in the sequence would just get us the first element. For multiplication, we would instead have to use one as the initial value since that is how the first function application will just give us the initial value.

In R we don't need to provide these initial values if we are happy with just having the first function call be on the first two elements in the list, so multiplication works just as well as addition without providing `init`:

```
Reduce(‘*‘, 1:5)
```

```
## [1] 120
```

```
Reduce(‘*‘, 1:5, accumulate = TRUE)
```

```
## [1]   1   2   6  24 120
```

```
Reduce(‘*‘, 1:5, right = TRUE, accumulate = TRUE)
```

```
## [1] 120 120  60  20   5
```

You wouldn't normally use `Reduce` for summarising values as their sum or product, there are already functions in R for this (`sum` and `prod`, respectively), and these are much faster as they are low-level functions implemented in C while `Reduce` has to be high level to handle arbitrary functions. For more complex data where we do not already have a function to summarise a list, `Reduce` is often the way to go.

Here is an example taken from Hadley Wickham's *Advanced R* book:

```
samples <- replicate(3, sample(1:10, replace = TRUE),
                                      simplify = FALSE)
str(samples)
```

```
## List of 3
##  $ : int [1:10] 7 10 4 3 7 10 4 4 4 3
##  $ : int [1:10] 3 6 2 1 8 10 3 3 10 3
##  $ : int [1:10] 7 10 10 3 9 7 9 8 1 4
```

```
Reduce(intersect, samples)
```

```
## [1] 10  3
```

We have a list of three vectors each with ten samples of the numbers from one to ten, and we want to get the intersection of these three lists. That means taking the intersection of the first two and then taking the intersection of that result and the third list. Perfect for `Reduce`. We just combine it with the `intersection` function.

Bringing the functions together

The three functions are often used together, where `Filter` first gets rid of elements that should not be processed, then `Map` processes the list, and finally `Reduce` combines all the results.

In this section, we will see a few examples of how we can use these functions together. We start with processing trees. Remember that we can construct trees using the `make_node` function we wrote earlier, and we can, of course, create a list of trees.

```
A <- make_node("A")
C <- make_node("C", make_node("A"),
                    make_node("B"))
E <- make_node("E",
               make_node("C", make_node("A"), make_node("B")),
               make_node("D"))

trees <- list(A = A, C = C, E = E)
```

Printing a tree gives us the list representation of it, if we `unlist` a tree we get the same representation, just flattened, so the structure is shown in the names of the resulting vector, but we wrote a `print_tree` function that gives us a string representation in Newick format.

```
trees[[2]]
```

```
## $name
## [1] "C"
##
## $left
## $left$name
## [1] "A"
##
## $left$left
## NULL
##
## $left$right
## NULL
```

```
##
##
## $right
## $right$name
## [1] "B"
##
## $right$left
## NULL
##
## $right$right
## NULL
```

```
unlist(trees[[2]])
```

```
##       name  left.name right.name
##        "C"        "A"        "B"
```

```
print_tree(trees[[2]])
```

```
## [1] "(A,B)"
```

We can use `Map` to translate a list of trees into their Newick format and flatten this list to just get a vector of characters.

```
Map(print_tree, trees)
```

```
## $A
## [1] "A"
##
## $C
## [1] "(A,B)"
##
## $E
## [1] "((A,B),D)"
```

```
unlist(Map(print_tree, trees))
```

```
##           A           C           E
##         "A"     "(A,B)" "((A,B),D)"
```

We can combine this with `Filter` to only get the trees that are not single leaves, here we can use the `size_of_tree` function we wrote earlier

```
unlist(Map(print_tree,
           Filter(function(tree) size_of_tree(tree) > 1, trees)))
```

```
##           C           E
##     "(A,B)" "((A,B),D)"
```

or we can get the size of all trees and compute their sum by combining `Map` with `Reduce`

```
unlist(Map(size_of_tree, trees))
```

```
## A C E
## 1 3 5
```

```
Reduce(‘+‘, Map(size_of_tree, trees), 0)
```

```
## [1] 9
```

We can also search for the node depth of a specific node and for example get the depth of “B” in all the trees:

```
node_depth_B <- function(tree) node_depth(tree, "B")
unlist(Map(node_depth_B, trees))
```

```
##  A  C  E
## NA  1  2
```

The names we get in the result are just confusing, they refer to the names we gave the trees when we constructed the list, and we can get rid of them by using the parameter `use.names` in `unlist`. In general, if you don’t need the names of a vector you should always do this, it speeds up computations when R doesn’t have to drag names along with the data you are working on.

```
unlist(Map(node_depth_B, trees), use.names = FALSE)
```

```
## [1] NA  1  2
```

For trees that do not have a "B" node we get `NA` when we search for the node depth, and we can easily remove those using `Filter`

```
Filter(function(x) !is.na(x),
       unlist(Map(node_depth_B, trees), use.names = FALSE))
```

```
## [1] 1 2
```

or we can explicitly check if a tree has node "B" before we `Map` over the trees

```
has_B <- function(node) {
  if (node$name == "B") return(TRUE)
  if (is.null(node$left) && is.null(node$right)) return(FALSE)
  has_B(node$left) || has_B(node$right)
}
unlist(Map(node_depth_B, Filter(has_B, trees)), use.names = FALSE)
```

```
## [1] 1 2
```

The solution with filtering after mapping is probably preferable since we do not have to remember to match the `has_B` with `node_depth_B` if we replace them with general functions that handle arbitrary node names, but either solution will work.

The apply family of functions

The `Map` function is a general solution for mapping over elements in a list, but R has a whole family of `Map`-like functions that operate on different types of input. These are all named "something"-apply, and we have already seen `sapply` in the previous chapter. The `Map` function is actually just a wrapper around one of these, the function `mapply`, and since we have already seen `Map` in use I will not also discuss `mapply`, but I will give you a brief overview of the other functions in the apply family.

sapply, vapply, and lapply

The functions `sapply`, `vapply`, and `lapply` all operate on sequences. The difference is that `sapply` tries to *simplify* its output, `vapply` takes a value as an argument and will coerce its output to have the type of this value, and give an error if it cannot, and `lapply` maps over lists.

Using `sapply` is convenient for interactive sessions since it essentially works like `Map` combined with `unlist` when the result of a map can be converted into a vector. Unlike `unlist` it will not flatten a list, though, so if the result of a map is more complex than a vector, `sapply` will still give you a list as its result. Because of this, `sapply` can be dangerous to use in a program. You don't necessarily know which type the output will have so you have to program defensively and check if you get a vector or a list.

```
sapply(trees, size_of_tree)
```

```
## A C E
## 1 3 5
```

```
sapply(trees, identity)
```

```
##       A    C      E
## name  "A"  "C"    "E"
## left  NULL List,3 List,3
## right NULL List,3 List,3
```

Using `vapply` you get the same simplification as using `sapply` if the result can be transformed into a vector, but you have to tell the function what type the output should have. You do this by giving it an example of the desired output. If `vapply` cannot translate the result into that type, you get an error instead of a type of a different type, making the function safer to use in your programming. After all, getting errors is better than unexpected results due to type-confusion.

```
vapply(trees, size_of_tree, 1)
```

```
## A C E
## 1 3 5
```

The `lapply` is the function most similar to `Map`. It takes a list as input and returns a list. The main difference between `lapply` and `Map` is that `lapply` always operate on a *single* list, while `Map` can take multiple lists (which explains the name of `mapply`, the function that `Map` is a wrapper for).

```
lapply(trees, size_of_tree)
```

```
## $A
## [1] 1
##
## $C
## [1] 3
##
## $E
## [1] 5
```

The `apply` function

The `apply` function works on matrices and higher-dimensional arrays instead of sequences. It takes three parameters, plus any additional parameters that should just be passed along to the function called. The first parameter is the array to map over, the second which dimension(s) we should marginalise along, and the third the function we should apply.

We can see it in action by creating a matrix to apply over

```
(m <- matrix(1:6, nrow=2, byrow=TRUE))
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

To see what is actually happening we will create a function that collects the data that it gets so we can see exactly what it is called with

```
collaps_input <- function(x) paste(x, collapse = ":")
```

If we marginalise on rows it will be called on each of the two rows and the function will be called with the entire row vectors

```
apply(m, 1, collaps_input)
```

```
## [1] "1:2:3" "4:5:6"
```

If we marginalise on columns it will be called on each of the three columns and produce tree strings:

```
apply(m, 2, collaps_input)
```

```
## [1] "1:4" "2:5" "3:6"
```

If we marginalise on both rows and columns it will be called on each single element instead:

```
apply(m, c(1, 2), collaps_input)
```

```
##      [,1] [,2] [,3]
## [1,] "1"  "2"  "3"
## [2,] "4"  "5"  "6"
```

The `tapply` function

The `tapply` function works on so-called ragged tables, tables where the rows can have different lengths. You cannot directly make an array with different sizes of dimensions in rows, but you can use a flat vector combined with factors that indicate which virtual dimensions you are using. The `tapply` function groups the vector according to a factor and then calls its function with each group.

```
(x <- rnorm(10))
```

```
##  [1]  0.1311435 -2.8011897 -0.5525549  0.7913567
##  [5]  0.1799513  0.9231047 -0.7701383  0.2615430
##  [9]  0.4992433 -1.3999893
```

```
(categories <- sample(c("A", "B", "C"), size = 10, replace = TRUE))
```

```
##  [1] "B" "A" "A" "C" "C" "A" "A" "B" "B" "C"
```

```
tapply(x, categories, mean)
```

```
##          A          B          C
## -0.8001945  0.2973099 -0.1428937
```

You can use more than one factor if you wrap the factors in a list:

```
(categories2 <- sample(c("X", "Y"), size = 10, replace = TRUE))
```

```
##  [1] "Y" "X" "Y" "X" "X" "X" "X" "X" "X" "Y"
```

```
tapply(x, list(categories, categories2), mean)
```

```
##            X          Y
## A -0.8827411 -0.5525549
## B  0.3803931  0.1311435
## C  0.4856540 -1.3999893
```

Functional programming in purrr

The `Filter`, `Map`, and `Reduce` functions are the building blocks of many functional algorithms, in addition to recursive functions. However, many common operations require various combinations of the three functions, and combinations with `unlist`, so writing functions using *only* these three/four functions means building functions from the most basic building blocks. This is not efficient, so you want to have a toolbox of more specific functions for common operations.

The package `purrr`[1] implements a number of such functions for more efficient functional programming, in addition to its own versions of `Filter`, `Map`, and `Reduce`. A complete coverage of the `purrr` package is beyond the scope of this

[1] https://github.com/hadley/purrr

book but I will give a quick overview of the functions available in the package and urge you to explore the package more if you are serious in using functional programming in R.

```
library(purrr)
```

The functions in `purrr` all take the sequence they operate on as their first argument — similar to the apply family of functions but different from the `Filter`, `Map`, and `Reduce` functions.

`Filter`-like functions

The `purrr` analogue of `Filter` is called `keep` and works exactly like `Filter`. It takes a predicate and returns a list of the elements in a sequence where the predicate returns `TRUE`. The function `discard` works similarly but returns the elements where the predicate returns `FALSE`.

```
keep(1:5, rcurry2(`>`)(3))
```

```
## [1] 4 5
```

```
discard(1:5, rcurry2(`>`)(3))
```

```
## [1] 1 2 3
```

If you give these functions a vector you get a vector back, of the same type, and if you give them a list you will get a list back

```
keep(as.list(1:5), rcurry2(`>`)(3))
```

```
## [[1]]
## [1] 4
##
## [[2]]
## [1] 5
```

Two convenience functions that you could implement by checking the length of the list returned by `Filter`, are `every` and `some`, that checks if all elements in the sequence satisfy the predicate or if some elements satisfy the predicate.

```
every(1:5, rcurry2(`>`)(0))
```

```
## [1] TRUE
```

```
every(1:5, rcurry2(`>`)(3))
```

```
## [1] FALSE
```

```
some(1:5, rcurry2(`>`)(3))
```

```
## [1] TRUE
```

```
some(1:5, rcurry2(`>`)(6))
```

```
## [1] FALSE
```

In the examples here I have used the `rcurry2` function we defined earlier, but with `purrr` it is very easy to write anonymous functions in a less verbose way than the typical R functions. You can use the "formula notation" and define an anonymous function by writing `~` followed by the body of the function, where the function argument is referred to by the variable `.x`.

```
keep(1:5, ~ .x > 3)
```

```
## [1] 4 5
```

```
discard(1:5, ~ .x > 3)
```

```
## [1] 1 2 3
```

This short-hand for anonymous functions is only available within functions from `purrr`, though. Just because you have imported the `purrr` package, you will not get the functionality in other functions.

Map-like functions

The Map functionality comes in different flavours in purrr, depending on the type of output you want and the number of input sequences you need to map over.

The map function always returns a list while functions map_lgl, map_int, map_dbl, and map_chr return vectors of logical values, integer values, numeric (double) values, and characters, respectively.

```
map(1:5, ~ .x + 2)
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 5
##
## [[4]]
## [1] 6
##
## [[5]]
## [1] 7
```

```
map_dbl(1:5, ~ .x + 2)
```

```
## [1] 3 4 5 6 7
```

The map family of functions all take a single sequence as input, but there are corresponding functions for two sequences, the map2, family and for an arbitrary number of sequences, the pmap family.

For the map2 functions you can create anonymous functions that refer to the two input values they will be called with by using variables .x and .y.

```
map2(1:5, 6:10, ~ 2 * .x + .y)
```

```
## [[1]]
## [1] 8
##
## [[2]]
## [1] 11
##
## [[3]]
## [1] 14
##
## [[4]]
## [1] 17
##
## [[5]]
## [1] 20
```

```
map2_dbl(1:5, 6:10, ~ 2 * .x + .y)
```

```
## [1]  8 11 14 17 20
```

For arbitrary numbers of sequences, you must use the `pmap` family and wrap the sequences in a list that is given as the first parameter. If you use anonymous functions you will need to define them using the general R syntax; there are no shortcuts for specifying anonymous functions with three or more parameters.

```
pmap(list(1:5, 6:10, 11:15),
     function(x, y, z) x + y + z)
```

```
## [[1]]
## [1] 18
##
## [[2]]
## [1] 21
##
## [[3]]
## [1] 24
##
## [[4]]
## [1] 27
```

```
##
## [[5]]
## [1] 30
```

```
pmap_dbl(list(1:5, 6:10, 11:15),
         function(x, y, z) x + y + z)
```

```
## [1] 18 21 24 27 30
```

The function `map_if` provides a variation of `map` that only applies the function it is mapping if a predicate is `TRUE`. If the predicate returns `FALSE` for an element, that element is kept unchanged.

For example, we can use it to multiply only numbers that are not even by two like this:

```
unlist(map_if(1:5, ~ .x %% 2 == 1, ~ 2*.x))
```

```
## [1]  2  2  6  4 10
```

A particularly nice feature of `purrr`'s map functions is that you can provide them with a string instead of a function and this is used, when you are working with sequences of elements that have names, like data frames and lists, to extract named components. So if we map over the trees from earlier we can, for example, use `map` to extract the left child of all the trees

```
map_chr(map(keep(trees, ~ size_of_tree(.x) > 1), "left"),
        print_tree)
```

```
##       C       E
##     "A" "(A,B)"
```

Here we combine three different functions: we use `keep` so we only look at trees that actually *have* a left child. Then we use `map` with `"left"` to extract the left child, and finally, we use `map_chr` to translate the trees into Newick format for printing.

Reduce-like functions

The Reduce function is implemented in two different functions, reduce and reduce_right.

```
reduce(1:5, `+`)
```

```
## [1] 15
```

```
reduce_right(1:5, `*`)
```

```
## [1] 120
```

Point-free Programming

In this last chapter we will not so much discuss actual programming but a programming style called *point free programming* (not *pointless* programming), that is characterised by constructing functions through a composition of other functions and not by actually writing new functions.

A lot of computing can be expressed as the steps that data flow through and how data is transformed along the way. We write functions to handle all the most basic steps, the atoms of a program and then construct functions for more complex operations by combining more fundamental transformations, building program molecules from the program atoms.

The term *point free* refers to the intermediate states data can be in when computing a sequence of transformations. The *points* it refers to are the states the data is in after each transformation, and *point free* means that we do not focus on these points in any way. They are simply not mentioned anywhere in code written using point free programming.

This might all sound a bit abstract but if you are used to writing pipelines in shell scripts it should soon become very familiar because point free programming is exactly what you do when you write a pipeline. There, data flow through a number of programs, tied together, so the output of one program becomes the input for the next in the pipeline, and you never refer to the intermediate data, only the steps of transformations the data go through as it is processed by the pipeline.

Function composition

The simplest way to construct new functions from basic ones is through function composition. In mathematics, if we have a function, f, mapping from domain

A to domain B, which we can write as $f : A \to B$, and another function g, $g : A \to C$, we can create a new function $h : A \to C$ by composing the two: $h(x) = g(f(x))$.

We can do exactly the same thing in R and define `h` in terms of functions `f` and `g` like this:

```
h <- function(x) g(f(x))
```

or even more verbose

```
h <- function(x) {
  y <- f(x)
  g(y)
}
```

Either way, there is a lot of extra fluff in writing a new function explicitly just to combine two other functions. In mathematical notation, we don't write the combination of two functions that way. We write the function composition as $h = g \circ f$. Composing functions to define new functions, rather than defining functions that just explicitly call others, is what we call point-free programming, and it is easily done in R.

We can write a function composition function, a higher-order function that takes two functions as arguments and returns their composition.

```
compose <- function(g, f) function(...) g(f(...))
```

We can then use this function to handle a common case when we are using `Map` and frequently want to `unlist` the result:

```
umap <- compose(unlist, Map)
umap(curry2(`+`)(2), 1:4)
```

```
## [1] 3 4 5 6
```

To get something similar to the mathematical notation we want it to be an infix operator, but the package `pryr` has already defined it for us so we can write the same code as this:

```
library(pryr)
umap <- unlist %.% Map
umap(curry2(`+`)(2), 1:4)
```

```
## [1] 3 4 5 6
```

We are not limited to only composing two functions, and since function composition is associative we don't need to worry about the order in which they are composed, and so we don't need to use parentheses around compositions. We can combine three or more functions as well, and we can combine functions with anonymous functions if we need some extra functionality that isn't already implemented as a named function. For example, we could define a function for computing the root mean square error like this:

```
rmse <- sqrt %.% mean %.% function(x, y) (x - y)**2
rmse(1:4, 2:5)
```

```
## [1] 1
```

We need a new function for computing the squared distance between `x` and `y`, so we add an anonymous function for that, but then we can just use `mean` and `sqrt` to get the rest of the functionality.

In the mathematical notation for function composition you always write the functions you compose in the same order as you would write them if you explicitly called them, so $h \circ g \circ f$ would be evaluated on a value x as $h(g(f(x)))$. This is great if you are used to reading from right to left, but if you are used to reading left to right it is a bit backwards.

Of course, nothing prevents us from writing a function composition operator that reverses the order of the functions. To avoid confusion with the mathematical notation for function composition, we would use a different operator so we could define ; such that $f; g = g \circ f$ and in R use that for function composition:

```
`%;%` <- function(f, g) function(...) g(f(...))
rmse <- (function(x, y) (x - y)**2) %;% mean %;% sqrt
rmse(1:4, 2:5)
```

```
## [1] 1
```

Here I need parentheses around the anonymous function to prevent R from considering the composition as part of the function body, but otherwise, the functionality is the same as before, we can just read the composition from left to right.

Pipelines

The `magrittr` package already implements a "left-to-right" function composition as the operator `%>%`. The `magrittr` package aims at making pipelines of data analysis simpler. It allows you to chain together various data transformations and analyses in ways very similar to how you chain together command-line tools in shell pipelines. The various operations are function calls, and these functions are chained together with the `%>%` operator, moving the output of one function to the input of the next.

For example, to take the vector `1:4`, get the `mean` of it, and then take the `sqrt`, you can write a pipeline like this:

```
library(magrittr)
1:4 %>% mean %>% sqrt
```

```
## [1] 1.581139
```

The default is that the output from the previous step in the pipeline is passed as the first argument to the next function in the pipeline. To write your own functions to fit into this pattern, you just need to make the data that come through a pipeline is the first argument for your function. Because `magrittr` pipelines are now frequently used in R, this is the pattern that most functions follow, and it is used in popular packages like `dplyr` and `tidyr`. The `purrr` package is also designed to work well with `magrittr` pipelines. All its functions take the data they operate on as their first parameter, so stringing together several transformations using `%>%` is straightforward.

Not all functions follow the pattern, mostly older functions do not, but you can use the variable "`.`" to refer to the data being passed along the pipeline when you need it to go at a different position than the first.

```
data.frame(x = 1:4, y = 2:5) %>% plot(y ~ x, data = .)
```

If the input is a data frame, you can access its columns using the column names as you would with any other data frame, the argument is still "`.`" and you just need to use that parameter to refer to the data.

```
data.frame(x = 1:4, y = 2:5) %>% plot(.$x, .$y)
```

The "`.`" parameter can be used several times in a function call in the pipeline and can be used together with function calls in a pipeline, e.g.:

```
rnorm(4) %>% data.frame(x = ., y = cos(.))
```

```
##            x           y
## 1  2.4314053 -0.75823974
## 2 -0.1664974  0.98617130
## 3 -1.5018321  0.06890957
## 4  0.6464462  0.79822948
```

There is one caveat, though, if "`.`" *only* appears in function calls, it will *also* be given to the function as a first parameter. This means that code such as the example below will create a data frame with three variables, the first being a column named "`.`".

```
rnorm(4) %>% data.frame(x = sin(.), y = cos(.))
```

```
##            .          x           y
## 1  0.9582171  0.8181677  0.57497963
## 2  0.2870045  0.2830805  0.95909615
## 3  0.1195326  0.1192482  0.99286448
## 4 -1.6182240 -0.9988755 -0.04740988
```

While the package is mainly intended for writing such data processing pipelines, it can also be used for defining new functions by composing other functions, this time with the functions written in left-to-right order. You write such functions just as you would write data processing pipelines, but let the first step in the pipeline be "`.`", so this is a data pipeline

```
mean_sqrt <- 1:4 %>% mean %>% sqrt
mean_sqrt
```

```
## [1] 1.581139
```

while this is a function

```
mean_sqrt <- . %>% mean %>% sqrt
mean_sqrt(1:4)
```

```
## [1] 1.581139
```

which of course is a function that you can use in a pipeline

```
1:4 %>% mean_sqrt
```

```
## [1] 1.581139
```

You can only use this approach to define functions taking a single argument as input. If you need more than one argument, you will need to define your function explicitly. The whole pipeline pattern works by assuming that it is a single piece of data that is passed along between function calls, but nothing prevents you from having complex data, such as data frames, and emulate having more than one argument in this way.

Take for example the root mean square error function we wrote above:

```
rmse <- (function(x, y) (x - y)**2) %;% mean %;% sqrt
```

This function takes two arguments so we cannot create it using ".". We can instead change it to take a single argument, for example, a data frame, and get the two values from there.

```
rmse <- . %>% { (.$x - .$y)**2 } %>% mean %>% sqrt
data.frame(x = 1:4, y = 2:5) %>% rmse
```

```
## [1] 1
```

Here we also used another feature of `magrittr`, a less verbose syntax for writing anonymous functions. By writing the expression {`(.$x - .$y)**2`} in curly braces we are making a function, in which we can refer to the argument as "`.`".

Being able to write anonymous functions by just putting expressions in braces is very convenient when the data needs to be massaged just a little to fit the output of one function to the expected input of the next.

Anonymous functions are also the way to prevent the "`.`" parameter getting implicitly passed as the first argument to a function when it is otherwise only used in function calls. If the expression is wrapped in curly braces, then the function call is not modified in any way and so "`.`" is not implicitly added as a first parameter.

```
rnorm(4) %>% { data.frame(x = sin(.), y = cos(.)) }
```

```
##            x          y
## 1  0.7087547 -0.7054550
## 2  0.8377746  0.5460162
## 3 -0.7275843  0.6860183
## 4 -0.1222865  0.9924948
```

Conclusions

This concludes this book on functional programming in R. You now know all the basic programming patterns used in day to day functional programming and how to use them in the R programming language.

Getting used to writing functional programs might take a little effort if you are only used to imperative or object-oriented programming, but the combination of higher-order functions and closures is a very powerful paradigm for effective programming and writing pure functions whenever possible makes for code that is much simpler to reason about.

R is not a pure functional programming language, though, so your code will usually mix imperative programming—which in most cases means using loops instead of recursions, both for convenience and efficiency reasons—with functional patterns. With careful programming, you can still keep the changing states of a program to a minimum and keep most of your program pure.

Helping you keep programming pure is the immutability of data in R. Whenever you "modify" data, you will implicitly create a copy of the data and then modify the copy. For reasoning about your programs, that is good news. It is very hard to create side effects of functions. It does come with some drawbacks, however. Many classical data structures assume that you can modify data. Since you cannot do this in R, you will instead have to construct your data structures such that updating them means creating new, modified, data structures.

We have seen how we can use linked lists ("next lists" in the terminology I have used in this book) and trees with functions that modify the data when computing on it. Lists and trees form the basic constructions for data structures in functional programs, but efficient functional data structures are beyond the scope of this book. I plan to return to it in a later book in the series.

I will end the book here, but I hope it is not the end of your exploration of functional programming in R.

If you liked this book, why not check out my list of other books[1] or sign up to my mailing list[2]?

Acknowledgements

I would like to thank Duncan Murdoch and the people on the R-help mailing list for helping me work out a kink in lazy evaluation in the trampoline example.

[1] http://wp.me/P9B2l-DN
[2] http://eepurl.com/cwIbR5