Advanced Statistical Programming in R

Meta-programming in R

Thomas Mailund

Table of Contents

Introduction

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

About the series

The *Advanced Statistical Programming in R* series consists 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 be already 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 gives an introduction to meta-programming. Meta-programming is when you write programs manipulating other programs: you treat code as data that you can generate, analyse, or modify. R is a very high-level language where all operations are functions, and all functions are data that we can manipulate. Functions are objects, and you can, within the language, extract their components, modify them, or create new functions from their constituent components.

There is great flexibility in how function calls and expressions are evaluated. The lazy evaluation semantics of R means that arguments to functions are passed as unevaluated expressions, and these expressions can be modified before they are evaluated, or they can be evaluated in other environments than the context where a function is defined. This can be exploited to create small domain-specific languages and is a fundamental component in the "tidy verse" in packages such as `dplyr` or `ggplot2` where expressions are evaluated in contexts defined by data frames.

There is some danger in modifying how the language evaluates function calls and expressions, of course. It makes it harder to reason about code. On the other hand, adding small embedded languages for dealing with everyday programming tasks adds expressiveness to the language that far outweighs the risks of programming confusion, as long as such meta-programming is used sparingly and in well-understood (and well documented) frameworks.

In this book, you will learn how to manipulate functions and expressions, and how to evaluate expressions in non-standard ways. Prerequisites for reading this book are familiarity with functional programming, at least familiarity with higher-order functions, that is, functions that take other functions as an input or that return functions.

Anatomy of a function

Everything you do in R involves defining functions or calling functions. You cannot do any action without evaluating some function or other. Even assigning values to variables or sub-scripting vectors or lists involves evaluating functions. But functions are more than just recipes for how to perform different actions, they are also data objects in themselves, and we have ways of probing and modifying them.

Manipulating functions

If we define a very simple function like this

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

we can examine the components it consists of. There are three parts to a function: its formal parameters, its body, and the environment it is defined in. The functions `formals`, `body`, and `environment` gives us these:

```
formals(f)
```

```
## $x
```

```
body(f)
```

```
## x
```

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

Formals

The formal parameters are given as a list where element names are the parameter names and values are default parameters.

```
g <- function(x = 1, y = 2, z = 3) x + y + z
parameters <- formals(g)
for (param in names(parameters)) {
  cat(param, "=>", parameters[[param]], "\n")
}
```

```
## x => 1
## y => 2
## z => 3
```

Strictly speaking, it is a so-called `pairlist`, but that is an implementation detail that has no bearing on how you treat it. You can treat it as if it is a `list`.

```
g <- function(x = 1, y = 2, z = 3) x + y + z
parameters <- formals(g)
for (param in names(parameters)) {
  cat(param, " => ", '"', parameters[[param]], '"', "\n", sep = "")
}
```

```
## x => "1"
## y => "2"
## z => "3"
```

For variables in this list that do not have default values, the list represents the values as the empty name. This is a special symbol that you cannot assign to, so it cannot be confused with a real value. You cannot use the `missing` function to check for a missing value in a `formals`—that function is only useful inside a function call, and in any case there is a difference between a missing parameter and one that doesn't have a default value—but you can always check if the value is the empty symbol.

```
g <- function(x, y, z = 3) x + y + z
parameters <- formals(g)
for (param in names(parameters)) {
  cat(param, " => ", '"', parameters[[param]], '"',
      " (", class(parameters[[param]]), ")\n", sep = "")
}
```

```
## x => "" (name)
## y => "" (name)
## z => "3" (numeric)
```

Primitive functions, those that call into the runtime system, such as `` `+` `` do not have formals. Only functions defined in R.

```
formals(`+`)
```

```
## NULL
```

Function bodies

The function body is an expression. For `f` it is a very simple expression

```
body(f)
```

```
## x
```

but even multi-statement function bodies are expressions. They just evaluate to the result of the last expression in the sequence.

```
g <- function(x) {
  y <- 2*x
  z <- x**2
  x + y + z
}
body(g)
```

```
## {
##     y <- 2 * x
##     z <- x^2
##     x + y + z
## }
```

When a function is called, R sets up an environment for it to evaluate this expression in; this environment is called the *evaluation environment* for the function call. The evaluation environment is first populated with values for the function's formal parameters, either provided in the function call or given as default parameters and then the body executes inside this environment. Assignments will modify this local environment unless you use the `` `<<-` `` operator, and the result of the function is the last expression evaluated in the body. This is either the last expression in a sequence or an expression explicitly given to the `return` function.

When we just have the body of a function as an expression, we don't get this function call semantics, but we can still try to evaluate the expression.

```
eval(body(f))
```

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

It fails because we do not have a variable `x` defined anywhere. If we had a global `x`, the evaluation would use that and not any function parameter, because the expression here doesn't know it is part of a function. It will be when it is evaluated as part of a call to `f` but not when we use it like this. We can give it a value for `x`, though, like this:

```
eval(body(f), list(x = 2))
```

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

The `eval` function evaluates an expression and uses the second argument to look up parameters. You can give it an environment, and the expression will then be evaluated in it, or you can use a list. We cover how to work with expressions and how to evaluate them in the *Expressions and environments* chapter; for now all you have to know is that we can evaluate an expression using `eval` if the variables in the expression are either found in the scope where we call `eval` or provided in the second argument to `eval`.

We can also set `x` as a default parameter and use that when we evaluate the expression:

```
f <- function(x = 2) x
formals(f)
```

```
## $x
## [1] 2
```

```
eval(body(f), formals(f))
```

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

Things get a little more complicated if default parameters refer to each other. This has to do with the evaluation environment is set up and not so much with how expressions are evaluated but consider an example where one default parameter refers to another.

```
f <- function(x = y, y = 5) x + y
```

Both parameters have default values so we can call `f` without any arguments.

```
f()
```

```
## [1] 10
```

We cannot, however, evaluate it just from the formal arguments without providing values:

```
eval(body(f), formals(f))
```

```
## Error in x + y: non-numeric argument to binary operator
```

In `formals(f)`, `x` points to the symbol `y` and `y` points to the numeric 5. But `y` is not used in the expression and if we simply look up `x` we just get the symbol `y`, we don't evaluate it further to figure out what `y` is. Therefore we get an error.

Formal arguments are not evaluated this way when we call a function. They are transformed into so-called promises: unevaluated expressions with an associated scope. This is how the formal language definition[1] puts it:

> When a function is called, each formal argument is assigned a promise in the local environment of the call with the expression slot containing the actual argument (if it exists) and the environment slot containing the environment of the caller. If no actual argument for a formal argument is given in the call and there is a default expression, it is similarly assigned to the expression slot of the formal argument, but with the environment set to the local environment.

What it means is that in the evaluating environment R first assign all variables to these "promises". The promises are place-holders for values but represented as expressions we haven't evaluated yet. As soon as you access them, though, they *will* be evaluated (and R will remember the value). For default parameters, the promises will be evaluated in the evaluating environment, and for parameters passed to the function in the function call the promises will be evaluated in the calling scope.

Since all the promises are unevaluated expressions, we don't have to worry about the order in which we assign the variables. As long as the variables exist when we evaluate a promise we are fine, and as long as there are no circular dependencies between the expressions we can figure out all the values when we need them.

Don't make circular dependencies. Don't do something like this:

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

We can try to make a similar setup for `f` where we build an environment of its formals as promises. We can use the function `delayedAssign` to assign values to promises like this:

[1] https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Argument-evaluation

```
fenv <- new.env()
parameters <- formals(f)
for (param in names(parameters)) {
  delayedAssign(param, parameters[[param]], fenv, fenv)
}
eval(body(f), fenv)
```

```
## [1] 10
```

Here we assign the expression `y` to variable `x` and the value 5 to variable `y`. Primitive values like a numeric vector are not handled as unevaluated expressions. They could be, but there is no point. So before we evaluate the body of `f`, the environment has `y` pointing to 5 and `x` pointing to the expression `y`, wrapped as a promise that says that the expression should be evaluated in `fend` when we need to know the value of `y`.

Function environments

The environment of a function is the simplest of its components. It is just the environment where the function was defined. This environment is used to capture the enclosing scope and is what makes closures possible in R. The evaluating environment will be set up with the function's environment when it is created such that variables not found in the local environment, consisting of local variables and formal parameters, will be searched for in the enclosing scope.

Calling a function

Before we continue, it might be worthwhile to see how these components fit together when a function is called. I explained this in some detail in *Functional Programming in R* but it is essential to know to understand how expressions are evaluated. When we start to fiddle around with non-standard evaluation, it becomes even more important. So it bears repeating.

When expressions are evaluated, they are evaluated in an environment. Environments are chained in a tree-structure. Each environment has a "parent," and when R needs to look up a variable, it first looks in the current environment to see if that environment holds the variable. If it doesn't, R will look in the parent. If it doesn't find it there either, it will look in the grandparent, and it will continue going up the tree until it either finds the variable or hits the global environment and see that it isn't there, at which point it will raise an error. We call the variables an expression can find by searching this way its scope. Since the search always picks the first place it finds a given variable, local variables overshadow global variables, and while several environments on this parent-chain might contain the same variable name, only the inner-most environment, the first we find, will be used.

When a function, `f`, is created, it gets associated `environment(f)`. This environment is the environment where `f` is defined. When `f` is invoked, R creates an evaluation environment for `f`, let's call it `evalenv`. The parent of `evalenv` is set to `environment(f)`. Since `environment(f)` is the environment where `f` is defined, having it as the parent of the evaluation environment means that the body of `f` can see its enclosing scope if `f` is a closure.

After the evaluation environment is created, the formals of `f` are added to it as promises. As we saw in the quote from the language definition earlier, there is a difference between default parameter and parameters given to the function where it is called in how these promises are set up. Default parameters will be promises that should be evaluated in the evaluation scope, `evalenv`. This means they can refer to other local variables or formal parameters, since these will be put in `evalenv`, and since `evalenv`'s parent is `environment(f)`, these promises can also refer to variables in the scope where `f` was defined. Expressions given to `f` where it is called, however, will be stored as promises that should be called in the calling environment. Let's call that `callenv`. If they were evaluated in the `evalenv` they would not be able to refer to variables in the scope were we call `f`, only local variables or variables in the scope were `f` was defined.

We can see it all in action in the example below:

```
enclosing <- function() {
  z <- 2
  function(x, y = x) {
    x + y + z
  }
}

f <- enclosing()
```

```
calling <- function() {
  w <- 5
  f(x = 2 * w)
}

calling()
```

```
## [1] 22
```

We start out in the global environment where we define `enclosing` to be a function. When we call `enclosing`, we create an evaluation environment in which we store the variable `z` and then return a function which we store in the global environment as `f`. Since this function was defined in the evaluation environment of `enclosing`, this environment is the `environment` of `f`.

Then we create `calling` and store that in the global environment and call it. This creates, once again, an evaluation environment. In this, we store the variable `w` and then call `f`. We don't have `f` in the evaluation environment, but because the parent of the evaluation environment is the global environment we can find it. When we call `f` we give it the expression `2 * w` as parameter `x`.

Inside the call to `f`, we have another evaluation environment. Its parent is the closer we got from `enclosing`. Here we need to evaluate `f`'s body: `x + y + z`, but before that the evaluation environment needs to be set up. Since `x` and `y` are formal parameters, they will be stored in the evaluation environment as promises. We provided `x` as a parameter when we called `f`, so this promise must be evaluated in the calling environment – the environment inside `calling` – while `y` has the default value so it must be evaluated in the evaluation environment. In this environment, it can see `x` and `y` and through the parent environment `z`. We evaluate `x`, which is the expression `2 * w` in the calling environment, where `w` is known and `y` in the local environment where `x` is known, so we can get the value of those two variables, and then from the enclosing environment `z`.

We can try to emulate all this using explicit environments and `delayedAssign` to store promises. We need three environments since we don't need to simulate the global environment for this. We need the environment where the `f` function was defined; we call it `defenv`, then we need the evaluating environment for the call to `f`, and we need the environment in which `f` is called.

```
defenv <- new.env()
```

```
evalenv <- new.env(parent = defenv)
callenv <- new.env()
```

Here, `defenv` and `calling` have the global environment as their parent, but we don't worry about that. The evaluating environment has `defend` as its parent.

In the definition environment we save the value of `z`:

```
defenv$z <- 2
```

In the calling environment we save the value of `w`:

```
callenv$w <- 5
```

In the evaluation environment, we set up the promises. The `delayedAssign` function takes two environments as arguments. The first is the environment where the promise should be evaluated and the second where it should be stored. For `x` we want the expression to be evaluated in the calling environment, and for `y` we want it to be evaluated in the evaluation environment. Both variables should be stored in the evaluation environment.

```
delayedAssign("x", 2 * w, callenv, evalenv)
delayedAssign("y", x, evalenv, evalenv)
```

In the `evalenv` we can now evaluate `f`:

```
f <- function(x, y = x) x + y + z
eval(body(f), evalenv)
```

```
## [1] 22
```

There is surprisingly much going on behind a function call, but it all follows these rules for how arguments are passed along as promises.

Modifying functions

We can do more than just inspect functions. The three functions for inspecting also come in assignment versions, and we can use those to change the three components of a function. If we go back to our simple definition of `f`

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

## function(x) x
```

we can try modifying its formal arguments by setting a default value for `x`.

```
formals(f) <- list(x = 3)
f

## function (x = 3)
## x
```

where, with a default value for `x`, we can evaluate its body in the environment of its formals.

```
eval(body(f), formals(f))

## [1] 3
```

I will stress again, though, that evaluating a function is not quite as simple as evaluating its body in the context of its formals. It doesn't matter that we change a function's formal arguments outside of its definition when the function is invoked the formal arguments will still be evaluated in the context where the function was defined.

If we define a closure, we can see this in action.

```
nested <- function() {
  y <- 5
  function(x) x
}
f <- nested()
```

Since `f` was defined inside the evaluating environment of `nested`, its `environment(f)` will be that environment. When we call it, it will, therefore, be able to see the local variable `y` from `nested`. It doesn't refer to that, but we can change this by modifying its formals

```
formals(f) <- list(x = quote(y))
f
```

```
## function (x = y)
## x
## <environment: 0x7ff8d458c108>
```

Here, we have to use the function `quote` to make `y` a name. If we didn't, we would get an error, or we would get a reference to a `y` in the global environment. In function definitions, default arguments are automatically quoted to turn them into expressions, but when we modify `formals`, we have to do this explicitly.

If we now call `f` without arguments, `x` will take its default value as specified by `formals(f)`, that is, it will refer to `y`. Since this is a default argument, it will be turned into a promise that will be evaluated in `f`'s evaluation environment. There is no local variable named `y`, so R will look in `environment(f)` for `y` and find it inside the `nested` environment.

```
f()
```

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

Just because we modified `formals(f)` in the global environment, we do not change in which environment R evaluates promises for default parameters. If we have a global `y`, the `y` in `f`'s formals still refer to the one in `nested`.

```
y <- 2
f()
```

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

Of course, if we provide `y` as a parameter when calling `f` things change. Now it will be a promise that should be evaluated in the calling environment, so in that case, we get a reference to the global `y`.

```
f(x = y)
```

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

We can modify the body of `f` as well. Instead of having its body refer to `x` we can, for example, make it return the constant 6:

```
body(f) <- 6
f
```

```
## function (x = y)
## 6
## <environment: 0x7ff8d458c108>
```

Now it evaluates that constant six when we call it, regardless of what `x` is.

```
f()
```

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

```
f(x = 12)
```

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

We can also try making `f`'s body more complex and make it an actual expression.

```
body(f) <- 2 * y
f()
```

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

Here, however, we don't get quite what we want. We don't want the body of a function to be evaluated before we call the function, but when we assign an expression like this we *do* evaluate it before we assign. There is a limit to how far lazy evaluation goes. Since `y` was 2, we are in effect setting the body of `f` to 4. Changing `y` afterwards doesn't change this.

```
y <- 3
f()
```

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

To get an unevaluated body we must, again, use `quote`.

```
body(f) <- quote(2 * y)
f
```

```
## function (x = y)
## 2 * y
## <environment: 0x7ff8d458c108>
```

Now, however, we get back to the semantics for function calls which means that the body is evaluated in an evaluation environment which parent is the environment inside `nested`, so `y` refers to the local and not the global parameter.

```
f()
```

```
## [1] 10
```

```
y <- 2
f()
```

```
## [1] 10
```

We can change `environment(f)` if we want to make `f` use the global `y`:

```
environment(f) <- globalenv()
f()
```

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

```
y <- 3
f()
```

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

If we do this, though, it will no longer know about the environment inside `nested`. It takes a lot of environment hacking if you want to pick and choose which environments a function finds its variables in, and if that is what you want you are probably better off rewriting the function to get access to variables in other ways.

If you want to set the formals of a function to missing values, that is, you want them to be parameters without default values, then you need to use `list` to create the arguments.

If we define a function `f` like this:

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

it takes one parameter, `x`, and add it to a global parameter `y`. If instead, we want `y` to be a parameter, but not give it a default value, we could try something like this:

```
formals(f) <- list(x =, y =)
```

This will not work, however, because `list` doesn't like empty values. Instead, you can use `alist`. This function creates a pair-list, a data structure used internally in R for formal arguments. It is the only thing this data structure is used for, but if you start hacking around with modifying parameters of a function, it is the one to use.

```
formals(f) <- alist(x =, y =)
```

Using `alist`, expressions are also automatically quoted. In the example above where we wanted the parameter `x` to default to `y`, we needed to use `quote(y)` to keep `y` as a promise to be evaluated in `environment(f)` rather than the calling scope. With `alist`, we do not have to quote `y`.

```
nested <- function() {
  y <- 5
  function(x) x
}
f <- nested()
formals(f) <- alist(x = y)
f
```

```
## function (x = y)
## x
## <environment: 0x7ff8d6287720>
```

```
f()
```

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

Constructing functions

We can also construct new functions by piecing together their components. The function to use for this is `as.function`. It takes an `alist` as input and interprets the last element in it as the new function's body and the rest as the formal arguments.

```
f <- as.function(alist(x =, y = 2, x + y))
f
```

```
## function (x, y = 2)
## x + y
```

```
f(2)
```

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

Don't try to use a `list` here; it doesn't do what you want.

```
f <- as.function(list(x = 2, y = 2, x + y))
```

If you give `as.function` a list, it interprets that as just an expression that then becomes the body of the new function. Here, if you have global definitions of `x` and `y` so you can evaluate `x + y`; you would get a body that is `c(2, 2, x+y)` where `x+y` refers to the value, not the expression, of the sum of global variables `x` and `y`.

The environment of the new function is by default the environment in which we call `as.function`. So to make a closure, we can just call `as.function` inside another function.

```
nested <- function(z) {
  as.function(alist(x =, y = z, x + y))
}
(g <- nested(3))
```

```
## function (x, y = z)
## x + y
## <environment: 0x7ff8d4ef0bf8>
```

```
(h <- nested(4))
```

```
## function (x, y = z)
## x + y
## <environment: 0x7ff8d4f61f78>
```

Here we call `as.function` inside `nested`, so the `environment` of the functions created here will know about the `z` parameter of `nested` and be able to use it in the default value for `y`.

Don't try this:

```
nested <- function(y) {
  as.function(alist(x =, y = y, x + y))
}
```

Remember that expressions that are default parameters are lazily evaluated inside the body of the function we define. Here, we say that `y` should evaluate to `y` which is a circular dependency. It has nothing to do with `as.function` you have the same problem in this definition:

```
nested <- function(y) {
  function(x, y = y) x + y
}
```

If you want something like that, where you make a function with a given default `y`, and you absolutely want the created function to call that parameter `y`, you need to evaluate the expression in the nesting scope and refer to it under a different name to avoid the nesting function's argument to overshadow it.

```
nested <- function(y) {
  z <- y
  function(x, y = z) x + y
}
nested(2)(2)
```

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

We can give an environment to `as.function` to specify the definition scope of the new function if we do not want it to be the current environment. In the example below we have two functions for constructing closures, the first create a function that can see the enclosing `z` while the other, instead, use the global environment, and is thus no closure at all, and so the argument `z` is not visible; instead the global `z` is.

```
nested <- function(z) {
  as.function(alist(x =, y = z, x + y))
}
nested2 <- function(z) {
  as.function(alist(x =, y = z, x + y),
              envir = globalenv())
}
```

If we evaluate functions created with these two functions, the `nested` one will add `x` to the `z` value we provide in the call to `nested` while the `nested2` will ignore its input and look for `z` in the global scope.

```
z <- -1
nested(3)(1)
```

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

```
nested2(3)(1)
```

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

Inside a function-call

When we execute the body of a function, as we have seen, we do this in the evaluation environment, that is linked through its parent to the environment where the function was defined, and with arguments stored as promises that will be evaluated either in the environment where the function was defined, for default parameters, or in the environment where the function was called, for parameters provided to the function there. In the last chapter, we saw how we could get hold of the formal parameters of a function, the body of the function, and the environment in which the function was defined. In this chapter, we will examine how we can access these, and more, from inside a function while the function is being evaluated.

Getting the components of the current function

In the last chapter, we could get the `formals`, `body`, and `environment` of a function we had a reference to. Inside a function body, we do not have such a reference. Functions do not have names as such; we give functions names when we assign them to variables, but that is a property of the environment where we have the name, not of the function itself, and functions we use as closures are often never assigned to any name at all. So how do we get hold of the current function to access its components?

To get hold of the current function, we can use the function `sys.function`. This function gives us the definition of the current function, which is what we need, not its name.

We can define this function to see how `sys.function` works:

```
f <- function(x = 5) {
  y <- 2 * x
  sys.function()
}
```

If we just write the function name on the prompt we get the function definition:

```
f
```

```
## function(x = 5) {
##   y <- 2 * x
##   sys.function()
## }
```

and since the function returns the definition of itself, we get the same when we evaluate it:

```
f()
```

```
## function(x = 5) {
##   y <- 2 * x
##   sys.function()
## }
```

When we call any of `formals`, `body`, or `environment`, we don't use a function name as the first parameter, we give them a reference to a function, and they get the function definition from that.

We don't need to explicitly call `sys.function` for `formals` and `body`, though, because these two functions already use a call to `sys.function` for the default value for the function parameter. If we want the components of the current function, we can simply leave out the function parameter.

Thus, to get the formal parameter of a function, inside the function body, we can just use `formals` without any parameters.

```
f <- function(x, y = 2 * x) formals()
params <- f(1, 2)
class(params)
```

```
## [1] "pairlist"
```

```
params
```

```
## $x
##
##
## $y
## 2 * x
```

The same goes for the body of the current function:

```
f <- function(x, y = 2 * x) {
  z <- x - y
  body()
}
f(2)
```

```
## {
##     z <- x - y
##     body()
## }
```

The `environment` function works slightly different. If we call it without parameters, we get the current (evaluating) environment.

```
f <- function() {
  x <- 1
  y <- 2
  z <- 3
  environment()
}
env <- f()
as.list(env)
```

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

```
##
## $y
## [1] 2
##
## $x
## [1] 1
```

This is not what we would get with `environment(f)`:

```
environment(f)
```

```
## <environment: R_GlobalEnv>
```

The `f` function is defined in the global environment, and `environment(f)` gives us the environment in which `f` is defined. If we call `environment()` inside `f` we get the evaluating environment. The local variables `x`, `y`, and `z` can be found in the evaluating environment, but they are not part of `environment(f)` – or if they are, they are different, global, parameters.

To get the equivalent of `environment(f)` from inside `f` we must get hold of the parent of the evaluating environment. We can get the parent of an environment using the function `parent.env`, so we can get the definition environment like this:

```
f <- function() {
  x <- 1
  y <- 2
  z <- 3
  parent.env(environment())
}
f()
```

```
## <environment: R_GlobalEnv>
```

When we have hold of a function definition, as in the previous chapter, we do not have an evaluating environment. That environment only exists when the function is called. A function we have defined, but not invoked, has the three components we covered in the previous chapter, but there are more components to a function that is actively executed; there is a difference between a function definition, a description of what a function should do when it is called, and a function instantiation, the actual running code. One such difference is the evaluating environment. Another is that a function instantiation has actual parameters while a function definition only has formal parameters. The latter are part of the function definition; the former are provided by the caller of the function.

Accessing actual function-parameters

We can see the difference between formal and actual parameters in the example below:

```
f <- function(x = 1:3) {
  print(formals()$x)
  x
}
f(x = 4:6)
```

```
## 1:3
```

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

The `formals` give us the arguments as we gave them in the function definition, where `x` is set to the expression `1:3`. It is a *promise*, to be evaluated in the defining scope when we access `x` in the cases where no parameters were provided in the function call, so in the `formals` list it is not the values 1, 2, and 3, but the expression `1:3`. In the actual call, though, we *have* provided the `x` parameter, so what this function call returns is `4:6`, but because we return it as the result of an expression this promise is evaluated so `f` returns 4, 5, and 6.

If we actually want the arguments passed to the current function in the form of the promises they are really represented as, we need to get hold of them without evaluating them. If we take an argument and use it as an expression, the promise will be evaluated. This goes for both default parameters and parameters provided in the function call; they are all promises that will be evaluated in different environments, but they are all promises nonetheless.

One way to get the expression that the promises represent is to use the function `substitute`. This function, which we will get intimately familiar with in the chapter *Manipulating expressions*, substitutes into an expression the values that variables refer to. This means that variables are replaced by the verbatim expressions, the expressions are not evaluated before they are substituted into an expression.

This small function illustrate how we can get the expression passed to a function:

```
f <- function(x = 1:3) substitute(x)
f()
```

```
## 1:3
```

Here we see that calling `f` with default parameters gives us the expression `1:3` back. This is similar to the `formals` we saw earlier in the section. We substitute `x` with the expression it has in its formal arguments; we do not evaluate the expression. We can, of course, once we have the expression

```
eval(f())
```

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

but it isn't done when we call `substitute`.

```
f(5 * x)
```

```
## 5 * x
```

```
f(foo + bar)
```

```
## foo + bar
```

Because the substituted expression is not evaluated, we don't even need to call the function with an expression that *can* be evaluated.

```
f(5 + "string")
```

```
## 5 + "string"
```

The substitution is verbatim. If we set up default parameters that depend on others, we just get them substituted with variable names; we do not get the value assigned to other variables.

```
f <- function(x = 1:3, y = x) substitute(x + y)
f()
```

```
## 1:3 + x
```

```
f(x = 4:6)
```

```
## 4:6 + x
```

```
f(y = 5 * x)
```

```
## 1:3 + 5 * x
```

In this example, we also see that we can call `substitute` with an expression instead of a single variable, we see that `x` gets replaced with the argument given to `x`, whether default or actual, and `y` gets replaced with `x` as the default parameter—not the values we provide for `x` in the function call, and with the actual argument when we provide it.

If we try to evaluate the expression we get back from the call to `f` we will not be evaluating it in the evaluation environment of `f`. That environment is not preserved in the substitution.

```
x <- 5
f(x = 5 * x)
```

```
## 5 * x + x
```

```
eval(f(x = 5 * x))
```

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

The expression we evaluate is `5 * x + x`, not `5 * x + 5 * x` as it would be if we substituted the value of `x` into `y`, as we would if we evaluated the expression inside the function.

```
g <- function(x = 1:3, y = x) x + y
g(x = 5 * x)
```

```
## [1] 50
```

A common use for `substitute` is to get the expression provided to a function as a string. This is used in the `plot` function, for instance, to set the default labels of a plot to the expressions `plot` is called with. Here, `substitute` is used in combination with the `departs` function. This function takes an expression and translates it into its text representation.

```
f <- function(x) {
  cat(deparse(substitute(x)), "==", x)
}
f(2 + x)
```

```
## 2 + x == 7
```

```
f(1:4)
```

```
## 1:4 == 1 2 3 4
```

Here, we use the `deparse(substitute(x))` pattern to get a textual representation of the argument `f` was called with, and the plain `x` to get it evaluated.

The actual type of object returned by `substitute` depends on the expression we give the function, and what the expression's variables refer to. If the expression, after variables have been substituted, is a simple type, that is what `substitute` returns.

```
f <- function(x) substitute(x)
f(5)
```

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

```
class(f(5))
```

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

If you give `substitute` a local variable you have assigned to, you also get a value back. This is not because `substitute` does anything special here; local variables like these are not promises, we have evaluated an expression when we assigned to one.

```
f <- function(x) {
  y <- 2 * x
  substitute(y)
}
f(5)
```

```
## [1] 10
```

```
class(f(5))
```

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

This behaviour only works inside functions, though. If you call `substitute` in the global environment, it considers variables as names and does not substitute them for their values.

```
x <- 5
class(substitute(5))
```

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

```
class(substitute(x))
```

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

It will substitute variables for values if you give a function a simple type as argument.

```
f <- function(x, y = x) substitute(y)
f(5)
```

```
## x
```

```
class(f(5))
```

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

```
f(5, 5)
```

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

```
class(f(5, 5))
```

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

If the expression that `substitute` evaluates to is a single variable, the type it returns is `name`, as we just saw. For anything more complicated, `substitute` will return a `call` object. Even if it is an expression that could easily be evaluated to a simple value; `substitute` does not evaluate expressions, it just substitutes variables.

```
f <- function(x, y) substitute(x + y)
f(5, 5)
```

```
## 5 + 5
```

```
class(f(5, 5))
```

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

A `call` object refers to an unevaluated function call. In this case, we have the expression `5 + 5`, which is the function call `` `+`(5, 5) ``.

Such `call` objects can also be manipulated. We can translate a `call` into a list to get its components, and we can evaluate it to invoke the actual function call.

```
my_call <- f(5, 5)
as.list(my_call)
```

```
## [[1]]
## ‘+‘
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] 5
```

```
eval(my_call)
```

```
## [1] 10
```

Since `substitute` doesn't evaluate a call, we can create function call objects with variables we can later evaluate in different environments.

```
rm(x) ; rm(y)
my_call <- f(x, y)
as.list(my_call)
```

```
## [[1]]
## ‘+‘
##
## [[2]]
## x
##
## [[3]]
## y
```

Here, we have created the call `x + y`, but removed the global variables `x` and `y`, so we cannot evaluate the call.

```
eval(my_call)
```

```
## Error in eval(expr, envir, enclos): object ’x’ not found
```

We can, however, provide the variables when we evaluate the call

```
eval(my_call, list(x = 5, y = x))
```

or we can set global variables and evaluate the call in the global environment.

```
x <- 5; y <- 2
eval(my_call)
```

```
## [1] 7
```

We can treat a `call` as a list and modify it. The first element in a `call` is the function we will call, and we can replace it like this:

```
(my_call <- f(5, 5))
```

```
## 5 + 5
```

```
my_call[[1]] <- ‘-‘
eval(my_call)
```

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

The remaining elements in the `call` are the arguments to the function call, and we can modify these as well:

```
my_call[[2]] <- 10
eval(my_call)
```

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

You can also create `call` objects manually using the `call` function. The first argument to `call` is the name of the function to call, and any additional arguments are passed on this function when the `call` object is evaluated.

```
(my_call <- call("+", 2, 2))
```

```
## 2 + 2
```

```
eval(my_call)
```

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

Unlike `substitute` inside a function, however, the arguments to `call` *are* evaluated when the call object is constructed. These are not lazy-evaluated.

```
(my_call <- call("+", x, y))
```

```
## 5 + 2
```

```
(my_call <- call("+", x - y, x + y))
```

```
## 3 + 7
```

From inside a function, you can get the call used to invoke it using the `match.call` function.

```
f <- function(x, y, z) {
  match.call()
}
(my_call <- f(2, 4, sin(2 + 4)))
```

```
## f(x = 2, y = 4, z = sin(2 + 4))
```

```
as.list(my_call)
```

```
## [[1]]
## f
##
## $x
## [1] 2
##
## $y
## [1] 4
##
## $z
## sin(2 + 4)
```

From the first element in this call, you can get the name of the function as it was called. Remember that the function itself doesn't have a name, but in the call to the function we have a reference to it, and we can get hold of that reference through the `match.call` function.

```
g <- f
(my_call <- g(2, 4, sin(2 + 4)))
```

```
## g(x = 2, y = 4, z = sin(2 + 4))
```

```
my_call[[1]]
```

```
## g
```

This function is often used to remember a function call in statistical models, where the call to the model constructor is saved together with the fitted model.

Accessing the calling scope

Inside a function, expressions are evaluated in the scope defined by the evaluating environment and its parent environment, the environment where the the function was defined, except for promises provided in the function call, which are evaluated in the calling scope. If we want direct access to the calling environment, inside a function, we can get hold of it using the function `parent.frame`.[1]

We can see this in action in this function:

```
nested <- function(x) {
  function(local) {
    if (local) x
    else get("x", parent.frame())
  }
}
```

We have a function, `nested`, whose local environment knows the value of the parameter `x`. Inside it, we create and return a function that, depending on its argument, either returns the value of the argument to `nested` or looks for `x` in the scope where the function is called.

```
f <- nested(2)
f(TRUE)
```

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

```
x <- 1
f(FALSE)
```

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

In the first call to `f` we get the local value of `x`, the number two, and in the second call to `f` we bypass the local scope and instead find `x` in the calling scope, which in this case is the global environment, where we find that `x` has the value one.

In a slightly more complicated version, we can try evaluating an expression in either the local evaluating environment or in the calling scope:

[1]This is an unfortunate name since the `parent.frame` has nothing to do with the parent environment, which we get using the `parent.env` function. The "frame" refers to environments on the call stack, often called stack frames, while the parent environment refers to the parents in environments.

```
nested <- function(x) {
  y <- 2
  function(local) {
    z <- 2
    expr <- expression(x + y + z)
    if (local) eval(expr)
    else eval(expr, envir = parent.frame())
  }
}
```

The logic is the same as the previous function, except in this function we define an expression and use `eval` to evaluate it either in the local or the calling scope. We need to create the expression using the `expression` function; if we did not, the expression would be evaluated (in the local scope) before `eval` gets to it. As the function is defined, we can explicitly choose which environment to use when we evaluate the expression.

```
f <- nested(2)
x <- y <- z <- 1
f(TRUE)
```

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

```
f(FALSE)
```

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

As the last example, we get a bit more inventive with what we can do with scopes, variables, and expressions. We want to write a function that lets us assign several variables at once from an expression, such a function call, that returns a sequence of values. Rather than having to write

```
x <- 1
y <- 2
z <- 3
```

we want to be able to write

```
bind(x, y, z) <- 1:3
```

and get the same effect. We can't *quite* get there because of how R deal with replacement functions, as it would interpret this expression to be, but we can modify the assignment operator to our infix function `` `%<-%` `` and get

```
bind(x, y, z) %<-% 1:3
```

Maybe not the prettiest syntax, but good enough as an example. We can, however, get even more ambitious and have this `bind` function assign to variables based on expressions so

```
bind(x, y = 2 * x, z = 3 * x) %<-% 2
```

assigns 2 to `x`, 4 to `y`, and 6 to `z`. The first because it is a positional parameter and the other two because we give them as expressions that can be evaluated once we know `x`.

To implement this syntax, we need to define the `bind` function and the `%<-%` operator. Of these two, the `bind` function is the simplest:

```
bind <- function(...) {
  bindings <- eval(substitute(alist(...)))
  scope <- parent.frame()
  structure(list(bindings = bindings, scope = scope),
            class = "bindings")
}
```

We use the `eval(substitute(alist(...)))` expression to get all the function's arguments into a pair-list without evaluating any potential expressions. We want to preserve lazy evaluation because expressions provided as arguments cannot be evaluated before we try to assign to variables we bind. Using `eval(substitute(alist(...)))` we can achieve this. The `substitute` call puts the actual arguments of the function into the expression `alist(...)`, and when we then evaluate this expression, we get the pair-list. The scope where we should bind variables, we get from `parent.frame`, and we then just combine the bindings and the scope in a class we call `"bindings"`. We don't need to make it into a class, but it doesn't hurt so we might as well.

The real work is done in the %<-% operator. Here, we need to do several things. We need to figure out which of the parameter bindings are just names, where we should assign values based on their position, and which are expressions that we need to evaluate. Positional parameters we can just assign a value and then store them in the scope we remembered in the bindings. Expressions should both have a name and an expression – we cannot assign to an actual expression in any scope, so we need these expressions to be named parameters – and the expression we need to evaluate. If expressions refer to other parameters we name in the `bind` call, they need to know what those are, so we need to evaluate the expressions in a scope given by `bind`. If the values we are assigning to the expressions have names we also want to be able to refer to them, for example, to write an assignment like this:

```
bind(y = 2 * x, z = 3 * x) %<-% c(x = 4)
```

To achieve this, we can make the values into an environment and make the parent scope of `bind` the parent of this environment as well. This way, they can refer to variables both in the values we assign and variables we assign to in the binding. The only tricky part about having expressions refer to other parameters we define is then the order in which to evaluate the expressions. For an expression to be evaluated, all the variables it refers to must be assigned to first. So it seems we would need to parse the expressions and figure out an order, a topological sorting of the expressions based on which variables are used in which expressions, but we can instead steal a trick from how functions evaluate arguments: lazy evaluation. If instead of assigning a value to each parameter we assign a promise, we won't have to worry about the order in which we assign the variables. R will handle this order whenever it sees a reference to any of these promises. This would mean that if we modify one of the assigned variables before we access another, we could get the lazy evaluation behaviour of functions. For example, if we did this:

```
bind(y = 2 * z, z = 3 * x) %<-% c(x = 4)
z <- 5
```

then `y` would refer to 10 and not 24. To avoid this problem, we can force evaluation of all the expressions once we are done with assigning them all.

The entire function is this:

```
.unpack <- function(x) unname(unlist(x, use.names = FALSE))[1]
`%<-%` <- function(bindings, value) {

  var_names <- names(bindings$bindings)
  val_names <- names(value)
  has_names <- which(nchar(val_names) > 0)
  value_env <- list2env(as.list(value[has_names]),
                        parent = bindings$scope)

  for (i in seq_along(bindings$bindings)) {
    name <- var_names[i]
    if (length(var_names) == 0 || nchar(name) == 0) {
      # we don't have a name so the expression
      # should be a name and we are
      # going for a positional value
      variable <- bindings$bindings[[i]]
      if (!is.name(variable)) {
        stop(paste0("Positional variables cannot be expressions ",
                    deparse(variable), "\n"))
      }
      val <- .unpack(value[i])
      assign(as.character(variable), val, envir = bindings$scope)

    } else {
      # if we have a name we also have an expression
      # and we evaluate that in the
      # environment of the value followed by the
      # enclosing environment and assign
      # the result to the name.
      assignment <- substitute(
        delayedAssign(name, expr,
                      eval.env = value_env,
                      assign.env = bindings$scope),
        list(expr = bindings$bindings[[i]]))
      eval(assignment)
    }
  }

  # force evaluation of variables to get rid of the lazy
  # promises.
  for (name in var_names) {
    if (nchar(name) > 0) force(bindings$scope[[name]])
  }
}
```

and it works as intended.

```
bind(x, y, z) %<-% 1:3
c(x, y, z)

## [1] 1 2 3

bind(y = 2 * x, z = 3 * x) %<-% c(x = 4)
c(y, z)

## [1]  8 12

bind(y = 2 * z, z = 3 * x) %<-% c(x = 4)
c(y, z)

## [1] 24 12
```

The only complicated part of it is how we handle the lazy assignment. We need to use `delayedAssign` for this, and we need the evaluation environment to be the environment that includes the values and the assignment environment to be the one we stored in the `bind` function. The difficult bit is getting the expression evaluated. We cannot evaluate it, that is what we are actively trying to avoid, so we need to give it as an expression. This expression, however, will not be evaluated until later, and in a different scope, so we cannot simply use the `bindings$bindings` list for the expression. We need to substitute the expression into an expression for the entire assignment and then evaluate it. The `eval(substitute(...))` pattern is how we can achieve this; in this function, it is split over two lines for readability, but it is the simple trick of using substitute to get an expression into another expression and then evaluating it.

If this whole exercise in expressions, substitutions, and evaluation makes your head spin, then take a deep breath and read on. We will have a deeper look at this in the next two chapters.

Expressions and environments

In this chapter, we dig deeper into how environments work and how we can evaluate expressions in different environments. Understanding how environments are chained together helps you understand how the language finds variables and being able to create, manipulate, and chain together environments when evaluating expressions is a key trick for meta-programming.

Expressions

You can consider everything that is evaluated in R to be an expression. Every statement you have in your programs is also an expression that evaluates to some value (which, of course, might be `NULL`). This includes control structures and function bodies. You can consider it all expressions; just some expressions involves evaluating several contained expressions for their side-effect before returning the result of the last expression they evaluate. From a meta-programming perspective, though, we are most interested in expressions we can get our hands on and examine, modify, or evaluate within a program.

Believe it or not, we have already seen most of the ways to get expression objects. We can get function bodies using the `body` function, or we can construct expressions using `quote` or `call`. Using any of these methods, we get an object we can manipulate and evaluate from within a program. Usually, expressions are just automatically evaluated when R gets to them during a program's execution, but if we have an expression as the kind of object we can manipulate, we have to evaluate it explicitly. If we don't evaluate it, its value is the actual expression; if we evaluate it, we get the value it corresponds to in a given scope.

In this chapter, we will not concern ourselves with manipulation of expressions. That is the topic for the next chapter. Instead, we will focus on how expressions are evaluated and how we can chance the scope I which we evaluate an expression. If we just write an expression as R source code, it will be evaluated in the scope where it is written. This is what you are used to. To evaluate it in a different scope, you need to use the `eval` function. If we don't give `eval` an environment, it will just evaluate an expression in the scope where `eval` is called, similar to if we had just written the expression there. If we give it an environment, however, that environment determines the scope in which the expression is evaluated.

To understand how we can exploit scopes, though, we first need to understand how environments define scopes in detail.

Chains of linked environments

We have seen how functions have associated environments that capture where they were defined, and that when we evaluate a function call, we have an evaluation environment that is linked to this definition environment. We have also see how this works through a linked list of parent environments. So far, though, we have claimed that this chain ends in the global environment, where we look for variables if we don't find them in any nested scope. For all the examples we have seen so far, this might as well be true, but in any real use of R, you have packages loaded. The reason that you can find functions from packages is that these packages are also found in the chain of environments. The global environment also has a parent, and when you load packages, the last package you loaded will be the parent of the global environment and the previous package will be the parent of the new package. This explains both how you can find variables in loaded packages and why loading new packages can overshadow variables defined in other packages.

It has to stop at some point, of course, and there is a special environment that terminates the sequence. This is known as the empty environment, and it is the only environment that doesn't have a parent. When you start up R, the empty environment is there, then R puts the base environment, where all the base language functionality lives, then it puts an `Autoload` environment, responsible for loading data on demand, and on top of that, it puts the global environment. The base environment and the empty environment are sufficiently important that we have functions to get hold of them. These are `baseenv` and `emptying`, respectively.

When you import packages, or generally `attach` a namespace, it gets put of the this list just below the global environment. The function `search` will give you the sequence of environments from the global environment and down. We can see how loading a library affects this list in this example:

```
search()
```

```
##  [1] ".GlobalEnv"
##  [2] "package:ggplot2"
##  [3] "package:purrr"
##  [4] "package:microbenchmark"
##  [5] "package:pryr"
##  [6] "package:magrittr"
##  [7] "package:knitr"
##  [8] "package:stats"
##  [9] "package:graphics"
## [10] "package:grDevices"
## [11] "package:utils"
## [12] "package:datasets"
## [13] "Autoloads"
## [14] "package:base"
```

```
library(MASS)
search()
```

```
##  [1] ".GlobalEnv"
##  [2] "package:MASS"
##  [3] "package:ggplot2"
##  [4] "package:purrr"
##  [5] "package:microbenchmark"
##  [6] "package:pryr"
##  [7] "package:magrittr"
##  [8] "package:knitr"
##  [9] "package:stats"
## [10] "package:graphics"
## [11] "package:grDevices"
## [12] "package:utils"
## [13] "package:datasets"
## [14] "Autoloads"
## [15] "package:base"
```

The `search` function is an internal function, but we can write our own version to get a feeling for how it could work. While `search` search from the global environment, though, we will make our function more general and give it an environment to start from. We simply need it to print out environment names and then move from the current environment to the parent until we hit the empty environment.

To get the name of an environment, we can use the function `environmentName`. Not all environments have names—those we create when we nest or call functions, or those we create with `new.env` have not—but environments created when we are loading packages do.[1] If an environment doesn't have a name, though, `environmentName`, will give us empty strings when environments do not have names. If so, we will instead just use `str` to get a representation of environments we can `cat` and make sure we only get a simple representation by not getting any potential attributes printed as well.

To check if we have reached the end of the environment chain, we check `identical(env, emptyenv())`. You cannot compare two environments with `==`, but you can use `identical`. Our function could look like this:

```
my_search <- function(env) {
  repeat {
    name <- environmentName(env)
    if (nchar(name) != 0)
      name <- paste0(name, "\n")
    else
      name <- str(env, give.attr = FALSE)
    cat(name)
    env <- parent.env(env)
    if (identical(env, emptyenv())) break
  }
}
```

Calling it with the global environment as the argument should give us a result similar to `search`, except we are printing the environments instead of returning a list of names.

[1]If you want to name your environments, you can set the attribute "name". It is generally not something you need, though.

```
my_search(globalenv())
```

```
## R_GlobalEnv
## package:MASS
## package:ggplot2
## package:purrr
## package:microbenchmark
## package:pryr
## package:magrittr
## package:knitr
## package:stats
## package:graphics
## package:grDevices
## package:utils
## package:datasets
## Autoloads
## base
```

Since we can give it any environment, we can try to get hold of the environment of a function. If we write a function nested into other function scopes, we can see that we get (nameless) environments for the functions.

```
f <- function() {
  g <- function() {
    h <- function() {
      function(x) x
    }
    h()
  }
  g()
}
my_search(environment(f()))
```

```
## <environment: 0x7ff12c76fcd0>
## <environment: 0x7ff12c76fbb8>
## <environment: 0x7ff12c76faa0>
## R_GlobalEnv
## package:MASS
## package:ggplot2
## package:purrr
## package:microbenchmark
## package:pryr
## package:magrittr
## package:knitr
## package:stats
## package:graphics
## package:grDevices
## package:utils
## package:datasets
## Autoloads
## base
```

We can also get hold of the environment of imported functions. For example, we can get the chain of environments starting at the `ls` function like this:

```
my_search(environment(ls))
```

```
## base
## R_GlobalEnv
## package:MASS
## package:ggplot2
## package:purrr
## package:microbenchmark
## package:pryr
## package:magrittr
## package:knitr
## package:stats
## package:graphics
## package:grDevices
## package:utils
## package:datasets
## Autoloads
## base
```

Here we see something that is a little weird. It looks like `"base"` is found both at the top and at the bottom of the list. Clearly, this can't be the same environment; it would have to have, as its parent, both the global environment and the parent, and environments only have one parent. The only reason it looks like it is the same environment is that environmentName gives us the same name for two different environments. They are different.

```
environment(ls)
```

```
## <environment: namespace:base>
```

```
baseenv()
```

```
## <environment: base>
```

The environment of ls is a namespace defined by the base package. The baseenv() environment is how this package is exported into the environment below the global environment, making base functions available to you, outside of the base package.

Having such extra environments is how packages manage to have private functions within a package and have other functions that are exported to users of a package. The base package is special in only defining two environments, the namespace for the package and the package environment. All other packages have three packages set up in their environment chain before the global environment: the package namespace, a namespace containing imported symbols, and then the base environment, which, as we just saw, connects to the global environment. A graph of environments when three packages are loaded, MASS, stats and graphics, is shown in fig. 4.1 (here graphics was loaded first, then stats and then MASS, so MASS appears first, followed by stats and then graphics on the path from the global environment to the base environment). The solid arrows indicate parent pointers for the environments and the dashed arrows indicate from which package symbols are exported into package environments.

If you try to access a symbol from a package, starting in the global environment, then you only get access to the exported functions, and some of these might be overshadowed by other packages you have loaded. For functions defined inside the package, their parent environment contains all the functions (and other variables) defined in the package, and because this package namespace environment sits before the global environment in the chain, variables from other packages do not overshadow variables inside the package when we execute functions from the package.

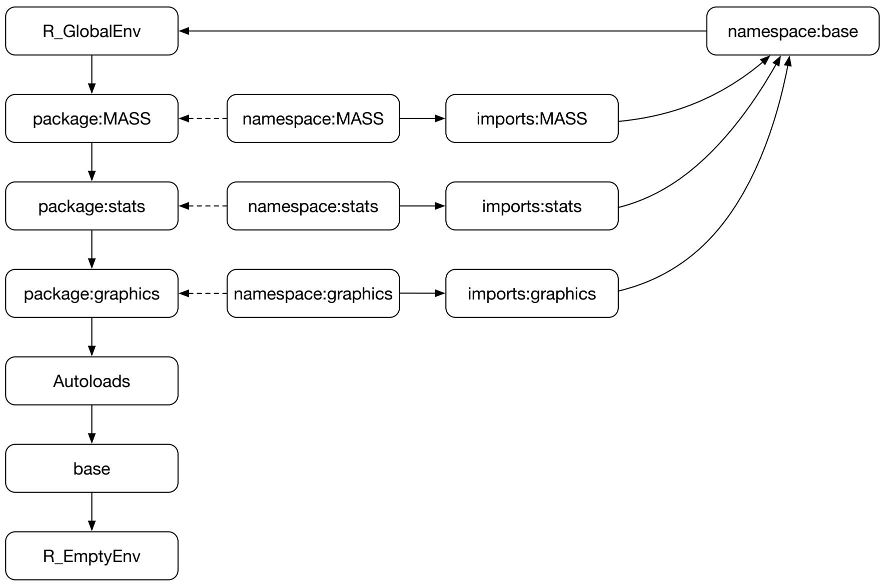

Figure 4.1: Environment graph with three loaded packages: `MASS`, `stats`, and `graphics`.

If a package imports other packages, these goes into the import environment, below the package namespace and before the base environment. So functions inside a package can see imported symbols, but only if they aren't overshadowed inside the package. If a user of the package imports other packages, these cannot overshadow any symbols the package functions can see, since such imports come later in the environment chain, as seen from inside the package. After the imports environment comes the base namespace. This gives all packages access to the basic R functionality, even if some of it should be overshadowed as seen from the global environment.[2]

There are three ways of specifying that a package depends on anther in the `DESCRIPTION` file: Using `Suggests:`, `Depends:`, and `Imports:`. The first doesn't actually set up any dependencies; it is just a suggestion of other packages that might enhance the functionality of the one being defined.

The packages specified in the `Depends:` directive will be loaded into the `search` path when you load the package. For packages specified here, you will clutter

[2]Strictly speaking, there is a lot more to importing other packages than what I just explained here. Since this book is not about R packages, I will only give a very short explanation in this footnote.

up the global namespace—not the global environment, but the search path below it—and you risk that functions you depend on will be overshadowed by packages that are loaded into the global namespace later. You should avoid using `Depends:` when you can, for these reasons.

Using `Imports:` you just require that a set of other packages are installed before your own package can be installed. Those packages, however, are not put on the search path, nor are they imported in the `imports` environment. Using `Imports:` just enable you to access functions and data in another package using the package namespace prefix, so if you `Imports:` the `stats` package you know you can access `stats::sd` because that function is guaranteed to exist on the installation when your package is used.

Actually importing variables into the `imports` namespace, you need to modify the `NAMESPACE` file, using the directives

- `imports()`
- `importFrom()`
- `importClassesFrom()`
- `importMethodsFrom()`

for importing an entire package, functions, S4 classes and S4 methods, respectively. The easiest way to handle the `NAMESPACE` file, though, is using `Roxygen`, and here you can import names using

- `@import <package>`
- `@importFrom <package> <name>`
- `@importClassesFrom <package> <classes>`
- `@importMethodsFrom <package> <methods>`

To ensure that packages you write play well with other namespaces you should use `Imports:` for dependencies you absolutely need (and `Suggests:` for other dependencies) and either use the package prefixes for dependencies in other packages or import the dependencies in the `NAMESPACE`.

The function `sd` sits in the package `stats`. Its parent is `namespace:stats` and its grandparent is `imports:stats`, and its great-grandparent is `namespace:base`. If we access `sd` from the global environment, though, we find it in `package:stats`.

```
my_search(environment(sd))
```

```
## stats
## imports:stats
## base
## R_GlobalEnv
## package:MASS
## package:ggplot2
## package:purrr
## package:microbenchmark
## package:pryr
## package:magrittr
## package:knitr
## package:stats
## package:graphics
## package:grDevices
## package:utils
## package:datasets
## Autoloads
## base
```

```
environment(sd)
```

```
## <environment: namespace:stats>
```

```
parent.env(environment(sd))
```

```
## <environment: 0x7ff12b24b5c8>
## attr(,"name")
## [1] "imports:stats"
```

```
parent.env(parent.env(environment(sd)))
```

```
## <environment: namespace:base>
```

```
parent.env(parent.env(parent.env(environment(sd))))
```

```
## <environment: R_GlobalEnv>
```

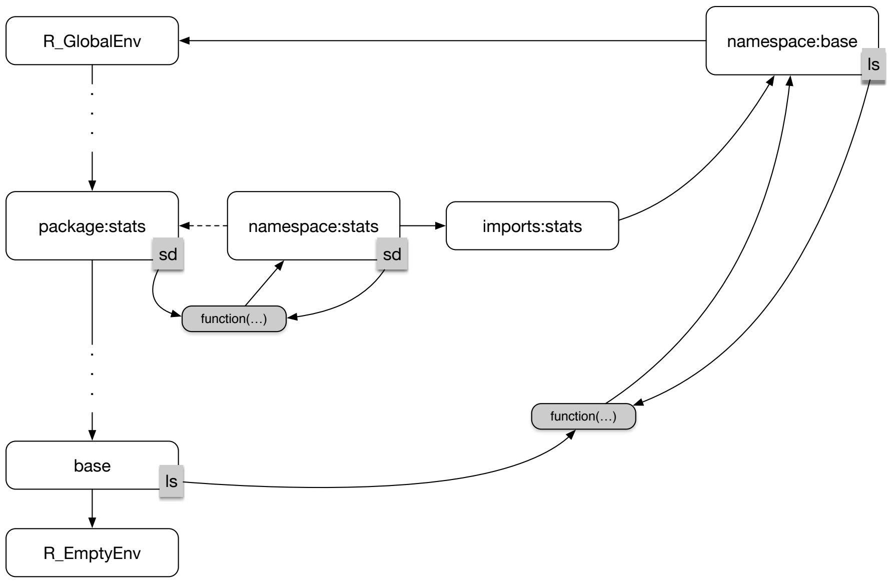

Figure 4.2: Environment graph showing the positions of `stats::sd` and `base::ls`.

Figure 4.2 shows how both `ls` and `sd` sits in package and namespace environments and how their parents are the namespace rather than the package environment.

As we now see, this simple way of chaining environments give us not only lexical scope in functions; it also explains how namespaces in packages work.

Environments and function calls

How environments and package namespaces work together is interesting to understand, and might give you some inspiration for how namespaces can be implemented for other uses, but in day-to-day programming, we are more interested in how environments and functions work together. So from now on, we are going to pretend that the scope graph ends at the global environment and focus on how environments are chained together when we define and call functions.

We already know the rules for this: when we call a function we get an evaluation environment, its parent points to the environment in which the function was defined, and if we want the environment of the caller of the function we can get it using the `parent.frame` function. Just to test our knowledge, though, we should consider a more complex example of nested functions than we have done so far. Consider the code below at the point where we evaluate the `i(3)` call. Figure 4.3 shows how environments are chained together; here solid lines indicate parent pointers and dashed lines are pointers from variables to their values. Figure 4.4 adds the call stack as parent frame environments. Follow along on the figures while we go through the code.

```
f <- function(x) {
  g <- function(y, z) x + y + z
  g
}
h <- function(a) {
  g <- f(x)
  i <- function(b) g(a + b, 5)
}
x <- 2
i <- h(1)
i(3)
```

Ok, the first two statements in the code defines functions `f` and `h`. We don't call them, we just define them, so we are not creating any new environments. The environment associated with both functions is the global environment. Then we set `x` to two. Nothing interesting happens here either. When we call `h`, however, we start creating new evaluation environments. We first create one to evaluate `h` inside. This environment sets `a` to 1 since we called `h` with 1. It then calls `f` with `x`.

Already here, it gets a little tricky. Since we call `f` with a variable, which will be lazy-evaluated inside `f`, we are not calling `f` with the value of the global parameter `x`. So `f` gets a promise it can later use to get a value for `x`, and this promise knows that `x` should be found in the scope of the `h` call we are currently evaluating. That `x` happens to be the global variable in this example, but you could assign to a local variable `x` inside `h` after you called `f` and then `f` would be using this local `x` instead. When we create a promise in a function call, the promise knows it should be evaluated in the calling scope, but it doesn't find any variables just yet; that only happens when the promise is evaluated.

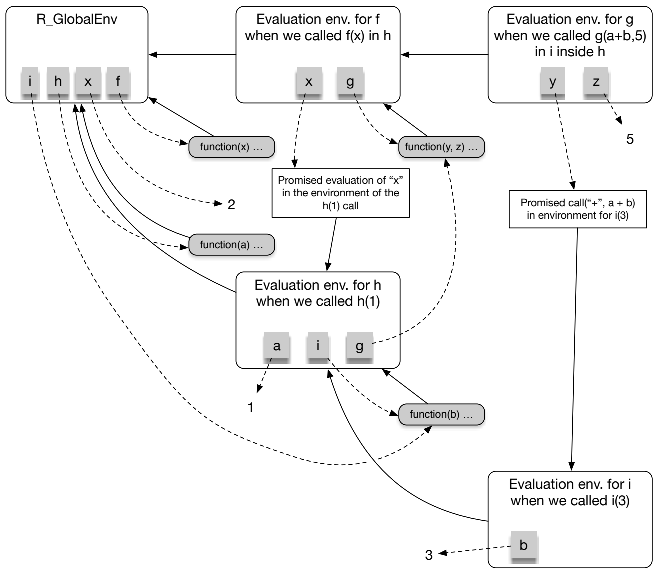
R_GlobalEnv
i
h
x
f
Evaluation env. for f when we called f(x) in h
x
g
Evaluation env. for g when we called g(a+b,5) in i inside h
y
z
5
function(x) …
function(y, z) …
Promised evaluation of "x" in the environment of the h(1) call
Promised call("+", a + b) in environment for i(3)
2
function(a) …
Evaluation env. for h when we called h(1)
a
i
g
1
function(b) …
Evaluation env. for i when we called i(3)
b
3

Figure 4.3: Environment graph for a complex example of nested functions.

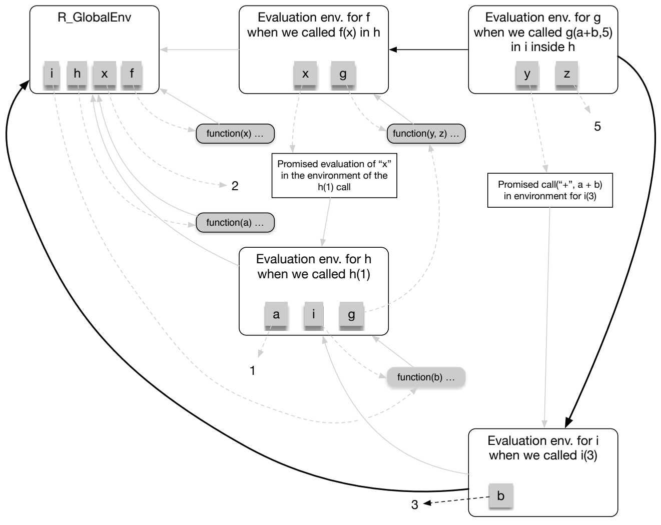
R_GlobalEnv
i
h
x
f
Evaluation env. for f when we called f(x) in h
x
g
Evaluation env. for g when we called g(a+b,5) in i inside h
y
z
5
function(x) …
function(y, z) …
Promised evaluation of "x" in the environment of the h(1) call
Promised call("+", a + b) in environment for i(3)
2
function(a) …
Evaluation env. for h when we called h(1)
a
i
g
1
function(b) …
Evaluation env. for i when we called i(3)
b
3

Figure 4.4: Environment graph for a complex example of nested functions highlighting the call stack as we can get it with `parent.frame`.

Inside the call to `f`, we define the function `g` and then return it. Since `g` is defined inside the `f` call, its environment is set to the evaluation scope of the call. This will later let `g` know how to get a value for `x`. It doesn't store `x` itself, but it can find it in the scope of the `f` call, where it will find it to be a promise that should be evaluated in the scope of the `h` call, where it will be found to be the global variable `x`. It already looks very complicated, but whenever you need a value, you should just follow the parent environment chain to see where you will eventually find it.

The `h(1)` call then defines a function, `i`, returns it, and we save that in the global variable `i`. The environment of this function is the evaluation scope of the `h(1)` call, so this is where the function will be looking for the names `a` and `g`.

Now, finally, we call `i(3)`. This first creates an evaluation environment where we set the variable `b` to `3` since that is what the argument for `b` is. Then we find the `g` function, which is the closer we created earlier, and we call `g` with parameters `a+b` and 5. The 5 is just passed along as a number, we don't translate constants into promises, but the `a+b` will be lazily evaluated, so it is a promise in the call to `g`. Calling `g`, we create a new evaluation environment for it. Inside this environment, we store the variables `y` and `z`. The former is set to the promise `a+b` and the latter to `5`. Since promises should be evaluated where they are defined, here in the calling scope, this is stored together with the promise.

We evaluate `g(a + b, 5)` as such: We first need to figure out what `x` is, so we look for it in the local evaluation environment, where we don't find it. Then we look in the parent environment where we do find it, but see that it is a promise from the `h(1)` environment, so we have to look there for it now. It isn't there either, so we continue down the parent chain and find it in the global environment where it is 2. Then we need to figure out what `y` is. We can find `y` in the local environment where we see that it is a promise, `a+b`, that should be evaluated in the `i(3)` environment. Here we need to find `a` and `b`. We can find `b` directly in the environment, so that is easy, but `a` we need to search for in the parent environment. Here we find it so, we can evaluate `a+b` as `1+3`. This value now replaces the promise in `y`. Finally, we need to find `z`, but at least this is easy. That is just the number 5 stored in the local environment. We now have all the values we need to compute `x + y + z`, they are `2 + (1+3) + 5` so when we return from the `i(3)` call, we get the return value 11.

The environment graphs can get rather complicated, but the rules for finding values are quite simple. You just follow environment chains. The only pitfall that tends to confuse programmers is the lazy evaluation. Here, the rules are also simple; they are just not as familiar. Promises are evaluated in the scope where they are defined. So a default parameter will be evaluated in the environment where the function is defined, and actual parameters will be evaluated in the calling scope. They will always be evaluated when you access a parameter, so if you don't want side-effects of modifying closure environments by changing variables in other scopes, you should use `force` before you create closures.

Take some time to work through this example. Once you understand it, you understand how environments work. It doesn't get more complicated than this. Well, unless we start messing with the environments as we are wont to do…

Manipulating environments

So how can we make working with environments even more complicated? We can, of course, start modifying them and chaining them up in all kinds of new ways. You see, we can not only access environments and put variables in them, but we can also modify the chain of parent environments. We can, for example, change the environment we execute a function in by changing the parent of the evaluation environment like this:

```
f <- function() {
  my_env <- environment()
  parent.env(my_env) <- parent.frame()
  x
}
g <- function(x) f()
g(2)
```

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

We are not changing the local environment—that would be hard to do, you don't have anywhere to put values if you don't have that—but we are making its parent point to the function call rather than the environment where the function was defined. If we didn't mess with the parent environment, `x` would be searched for in the global environment, but because we set the parent environment to the parent frame, we will instead start the search in the caller where we find `x`.

The power to modify scopes in this way can be used for good but certainly also for evil. There is nothing complicated in how it works; if you understood how environment graphs work from the previous section, you will also understand how they work if we start changing parent pointers. The main problem is just that environments are mutable, so if you start modifying them one place, it has consequences elsewhere.

Consider this example of a closure:

```
f <- function() {
  my_env <- environment()
  call_env <- parent.frame()
  parent.env(my_env) <- call_env
  y
}
g <- function(x) {
  closure <- function(y) {
    z <- f()
    z + x
  }
  closure
}
add1 <- g(1)
add2 <- g(2)
```

It is just a complicated way of writing a closure for adding numbers, but we are not going for elegance here—we aim to see how modifying environments can affect us. Here we have a function `f` that sets up its calling environment as its scope and then returns `y`. Since `y` is not a local variable it must be found in the enclosing scope with a search that starts in the parent environment; this is the environment we just changed to the calling scope. In the function `g` we then define a closure that takes one argument, `y`, and then calls `f` and adds `x` to the result of the call. Since `y` is a parameter of the closure, `f` will be able to see it when it searches from its (modified) parent scope. Since `x` is not local to the closure, it will be searched for in the enclosing scope, where it was a parameter of the enclosing function.

It works as we would expect it to. Even though it is a complicated way of achieving this effect, there are no traps in the code.

```
add1(3)
```

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

```
add2(4)
```

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

For reasons that will soon be apparent I just want to show you that setting a global variable `x` does not change the behaviour:

```
x <- 3
add1(3)
```

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

```
add2(4)
```

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

It also shouldn't. When we read the definition of the closure, we can see that the `x` it refers to is the parameter of `g`, not a global variable.

But now I am going to break the closure without even touching it. I just do one simple extra thing in `f`: I don't just change the enclosing scope of `f`, I do the same for the caller of `f`. I set the parent of the caller of `f` to be its caller instead of its enclosing environment:

```
f <- function() {
  my_env <- environment()
  call_env <- parent.frame()
  parent.env(my_env) <- call_env
  parent.env(call_env) <- parent.frame(2)
  y
}
```

I haven't touched the closure, `g`, of the `add1` and `add2` functions. I have just made a small change to `f`. Now, however, if I don't have a global variable for `x` the addition functions do not work. This would give me an error:

```
rm(x)
add1(3)
```

Even worse, if I *do* have a global variable `x` I don't get an error, but I don't get the expected results either.

```
x <- 3
add1(3)
```

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

```
add2(4)
```

```
## [1] 7
```

What happens here, of course, is that we change the enclosing scope of the closure from the `g` call to the global environment (which is the calling scope of `g` as well as its parent environment), so this is where we now start the search for `x`. The evaluation environment for the `g` call is not on the search path any longer.

While you *can* modify environments in this way, you should need an excellent reason to do so. Changing the behaviour of completely unrelated functions is the worst kind of side effects. It is one thing to mess up your own function, but don't mess up other people's functions. Of course, we are only modifying active environments here. We have not permanently damaged any function; we have just messed up the behaviour of function calls on the stack—if we wanted to mess up functions permanently we can do so using the `environment` function as well, though, but doing that tends to be a more deliberate and thought-through choice.

Don't go modifying the scope of calling functions. If you want to change the scope of expressions you evaluate, you are better off creating new environment chains for this, rather than modifying existing ones; the latter solution can easily have unforeseen consequences while the former at least has consequences restricted to the function you are writing.

Explicitly creating environments

You create a new environment with the `new.env` function. By default, this environment will have the current environment as its parent[3] and you can use functions such as `exists` and `get` to check what it contains.

```
env <- new.env()
x <- 5
exists("x", env)
```

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

```
get("x", env)
```

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

```
f <- function() {
  x <- 7
  new.env()
}
env2 <- f()
get("x", env2)
```

```
## [1] 7
```

You can also use the `$` subscript operator to access it, but in this case, R will not search up the parent list to find a variable; only if a variable is in the actual environment can you get to it.

```
env$x
```

```
## NULL
```

[3]If you check the documentation for `new.env` you will see that the default argument is actually `parent.frame()`. If you think about it, this is how it becomes the current environment: when you call `new.env` the current environment will be its parent frame.

You can assign variables to environments using `assign` or through the `$<-` function.

```
assign("x", 3, envir = env)
env$x

## [1] 3

env$x <- 7
env$x

## [1] 7
```

Depending on what you want to do with the environment, you might not want it to have a parent environment. There is no way to achieve that.

```
env <- new.env(parent = NULL) # This won't work!
```

All environments have a parent except the empty environment, but you can get the next best thing by making this environment the parent of your new one.

```
global_x <- "foo"
env <- new.env(parent = emptyenv())
exists("global_x", env)

## [1] FALSE
```

We can try to do something a little more interesting with manually created environments: build a parallel call stack we can use to implement dynamic scoping rather than lexical scoping. Lexical scoping is the scoping we already have in R, where a function call's parent is the definition scope of the function. Dynamic scope instead has the calling environment. It is terribly hard to reason about programs in languages with dynamic scope, so I would advise that you avoid them, but for the purpose of education, we can try implementing it.

Since we don't want to mess around with the actual call stack and modify parent pointers, we need to make a parallel sequence of environments, and we need to copy the content of each call stack frame into these. We can copy an environment like this:

```
copy_env <- function(from, to) {
  for (name in ls(from, all.names = TRUE)) {
    assign(name, get(name, from), to)
  }
}
```

Just for illustration purposes, we need a function that will show us what names we see in each environment when moving down towards the global environment—we don't want to go all the way down to the empty environment, here, so we stop a little early. This function lets us do that:

```
show_env <- function(env) {
  if (!identical(env, globalenv())) {
    print(env)
    print(names(env))
    show_env(parent.env(env))
  }
}
```

Now comes the function for creating the parallel sequence of environments. It is not that difficult; we can use `parent.frame` to get the frames on the call stack arbitrarily deep—well, down to the first function call—and we can get the depth of the call stack using the function `sys.nframe`. The only thing we have to be careful about is adjusting the depth of the stack by one since we want to create the call stack chain for the *caller* of the function; not for the function itself. The rest is just a loop.

```
build_caller_chain <- function() {
  n <- sys.nframe() - 1
  env <- globalenv()
  for (i in seq(1,n)) {
    env <- new.env(parent = env)
    frame <- parent.frame(n - i + 1)
    copy_env(frame, env)
  }
  env
}
```

To see it in action we need to set up a rather convoluted example with both nested scopes and a call stack. It doesn't look pretty, but try to work through it and consider what the function environments must be and what the call stack must look like.

```
f <- function() {
  x <- 1
  function() {
    y <- 2
    function() {
      z <- 3

      print("---Enclosing environments---")
      show_env(environment())

      call_env <- build_caller_chain()
      print("---Calling environments---")
      show_env(call_env)
    }
  }
}
g <- f()()
h <- function() {
  x <- 4
  y <- 5
  g()
}
h()
## [1] "---Enclosing environments---"
## <environment: 0x1035937b8>
## [1] "z"
## <environment: 0x103596780>
## [1] "y"
## <environment: 0x1035964e0>
## [1] "x"
## [1] "---Calling environments---"
## <environment: 0x10354b748>
## [1] "z"
## <environment: 0x103553b20>
## [1] "x" "y"
```

When we call `h` it calls `g`, and we get the list of environments starting from the innermost level where we have a `z` and out till the outermost level, just before the global environment, where we have the `x`. On the (parallel) call-stack we also see a `z` first (it is in a copy o the function's environment, but it is there), but this chain is only two steps long and the second environment contains both `x` and `y`.

We can use the stack chain we constructed here to implement dynamic scoping. We simply need to evaluate expressions in the scope defined by this chain rather than the current evaluating environment. The `<<-` assignment operator won't work—it would require us to write a similar function to get that behaviour, and it would be a design choice to which degree changes made to this chain should be moved back into the actual call stack frames—but as long as it comes to evaluating expressions, we can use it for dynamic scoping.

Environments and expression evaluation

Now, finally, we come to what this chapter is all about: how we combine expressions and environments to compute values. The good news is that we are past all the hard stuff and it gets pretty simple after all of the stuff above. All you have to do to evaluate an expression in any selected environment chain is to provide it to the `eval` function. We can see it in the example below that evaluates the same expression in the lexical scope and the dynamic scope:

```
f <- function() {
  x <- 1
  function() {
    y <- 2
    function() {
      z <- 3

      cat("Lexical scope: ", x + y + z, "\n")

      call_env <- build_caller_chain()
      cat("Dynamic scope: ", eval(quote(x + y + z), call_env), "\n")
    }
  }
}
g <- f()()
```

```
h <- function() {
  x <- 4
  y <- 5
  g()
}
h()
```

```
## Lexical scope:  6
## Dynamic scope:  12
```

The hard part of working with environments really isn't evaluating them. It is manipulating them.

Manipulating expressions

Expressions, the kind we create using the `quote` function, come in four flavours: a primitive value, a name, a function call or a control structure, or a *pairlist*. Function calls include operators such as the arithmetic or logical operators as these are function calls as well in R, and control structures can be considered just a special kind of function calls—they only really differ from function calls in the syntax you use to invoke them.

```
class(quote(1))
```

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

```
class(quote("foo"))
```

```
## [1] "character"
```

```
class(quote(TRUE))
```

```
## [1] "logical"
```

```
class(quote(x))
```

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

```
class(quote(f(x)))
```

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

```
class(quote(2+2))
```

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

```
class(quote(if (TRUE) "foo" else "bar"))
```

```
## [1] "if"
```

```
class(quote(for (x in 1:3) x))
```

```
## [1] "for"
```

Of these, the calls and control structures are of course the more interesting; values and symbols are pretty simple, and we cannot do a lot with them. Pairlists are used for dealing with function parameters, so unless we are working with function arguments, we won't see them in expressions. Calls and control structures, on the other hand, capture the action in an expression; we can treat these as lists, and we can thus examine them and modify them.[1] Working with expressions this way is, I believe, the simplest approach and is the topic of this chapter. Substituting values for variables is another, complementary, way that is the topic of the next chapter.

After we have learned the basics of expressions in the next section, the rest of the chapter will go through some potential real-life examples of how we would use meta-programming. You can find a full version of the examples in the `dfdr` package on GitHub[2].

The basics of expressions

Both function calls and control structures can be manipulated as lists. Of those two, we will mostly focus on calls since those are, in my experience, more likely to be modified in a program. So let us get control structures out of the way first. I will only describe `if` and `for`; the rest are similar.

[1]To the extent that we can modify data in R; we are of course creating new objects with replacement operators.

[2]https://github.com/mailund/dfdr

Accessing and manipulating control structures

Statements involving control structures are expressions like any other expression in R, and we can create an unevaluated version of them using `quote`. As I explained above, we can then treat this expression object as a list. So we can get the length of the object, and we can get access to the elements in the object. For a single `if`-statement we get expressions of length three while for `if-else`-statements we get expressions of length four. The first element is the name `if`. For all control structures and function calls, the first element will always be the name of the function, so if you think of control structures as just functions with a slightly weird syntax, you don't have to consider them a special case at all.[3] The second element is the test condition in the `if`-statement, and after that, you get the body of the statement. If it is an `if-else`-statement, the fourth element is the `else` part of the expression.

```
x <- quote(if (foo) bar)
length(x)
```

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

```
x[[1]]
```

```
## `if`
```

```
x[[2]]
```

```
## foo
```

```
x[[3]]
```

```
## bar
```

```
y <- quote(if (foo) bar else baz)
length(y)
```

[3]They basically *are* just special cases of calls. The `is.call` function will return `TRUE` for them, and there is no difference in how you can treat them. The only difference is in the syntax for how you write control-structure expressions compared to function calls.

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

```
y[[1]]
```

```
## `if`
```

```
y[[2]]
```

```
## foo
```

```
y[[3]]
```

```
## bar
```

```
y[[4]]
```

```
## baz
```

With `for`-loops you get an expression of length four where the first element is, of course, the name `for`, the second is the iteration variable, the third the expression we iterate over and the fourth the loop body.

```
z <- quote(for (x in 1:4) print(x))
length(z)
```

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

```
z[[1]]
```

```
## `for`
```

```
z[[2]]
```

```
## x
```

```
z[[3]]
```

```
## 1:4
```

```
z[[4]]
```

```
## print(x)
```

We can evaluate these control structures as any other expression:

```
eval(z)
```

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

And we can modify them by assigning to their elements to change their behaviour before we evaluate them, e.g., changing what we loop over:

```
z[[3]] <- 1:2
eval(z)
```

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

Changing what we do in the function body:

```
z[[4]] <- quote(print(x + 2))
eval(z)
```

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

Or changing the index variable and the body:

```
z[[2]] <- quote(y)
z[[4]] <- quote(print(y))
eval(z)
```

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

Accessing and manipulating function calls

For function calls their class is `call` and when we treat them as lists the first element is the name of the function being called and the remaining elements are the function call arguments.

```
x <- quote(f(x,y,z))
class(x)
```

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

```
length(x)
```

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

```
x[[1]]
```

```
## f
```

```
x[[2]]
```

```
## x
```

```
x[[3]]
```

```
## y
```

```
x[[4]]
```

```
## z
```

We can test if an expression is a function call using the `is.call` function:

```
is.call(quote(x))
```

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

```
is.call(quote(f(x)))
```

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

You don't have access to any function body or environment or anything like that here. This is just the name of a function; it is not unless we evaluate the expression we will have to associate the name with an actual function. We can, of course, always evaluate the function call using `eval` and we can modify the expression if we want to change how it should be evaluated:

```
x <- quote(sin(2))
eval(x)
```

```
## [1] 0.9092974
```

```
x[[1]] <- quote(cos)
eval(x)
```

```
## [1] -0.4161468
```

```
x[[2]] <- 0
eval(x)
```

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

I didn't quote the zero in the last assignment; I didn't have to since numeric values are already expressions and do not need to be quoted.

To explore an expression, we usually need a recursive function. The two basic cases in a recursion are then `is.atomic`—for values—and `is.name`—for symbols—and the recursive cases are `is.call` for function calls and `is.pairlist` if we want to deal with those. In the function below, that just prints the structure of an expression, I do not bother.

```
f <- function(expr, indent = "") {
  if (is.atomic(expr) || is.name(expr)) { # basic case
    print(paste0(indent, expr))

  } else if (is.call(expr)) { # a function call / sub-expression
    print(paste0(indent, expr[[1]]))
    n <- length(expr)
    if (n > 1) {
      new_indent <- paste0(indent, "  ")
      for (i in 2:n) {
        f(expr[[i]], new_indent)
      }
    }

  } else {
    print(paste0(indent, "Unexpected expression: ", expr[[1]]))
  }
}

f(quote(2 + 3*(x + y)))
```

```
## [1] "+"
## [1] "  2"
## [1] "  *"
## [1] "    3"
## [1] "    ("
## [1] "      +"
## [1] "        x"
## [1] "        y"
```

You might find the output here a little odd, but it captures the structure of the expression `2+3*(x+y)`. The outermost function call is the function `+`, and it has two arguments, the number 2 and the call to `*`. The call to `*` also has two arguments—naturally—where one is 3 and the other is a call to the function `(`. If you find this odd, then welcome to the club, but parentheses are functions in R. The call to `(` only has a single argument which happens to be a function call to `+` with the arguments `x` and `y`.

This is all there is to the direct manipulation of function calls, but of course, the is much that can be done with these simple tools and the following sections will show how you can use them to achieve powerful effects.

Expression simplification

To see the manipulation of expressions in action we consider a scenario where we want to simplify an expression. We want to evaluate sub-expressions that we can immediately evaluate, because they only consist of atomic values where we do not depend on variables, and we want to reduce multiplication by one or addition by zero. Something like this:

```
simplify_expr(quote(2*(0 + ((4 + 5)*x)*1)))
## 2 * (9 * x)
```

It isn't quite perfect; if we really reduced the expression we would see that we could rearrange the parentheses and multiply 2 by 9, but we are going to simplify expressions locally and not attempt to rewrite them.

Since we are dealing with expressions, we need a recursive function that handles the basic cases—atomic values and names—and the recursive cases—calls and pair lists. I don't expect to see a pair list in an expression, so I simply give up if I see anything except atomic, name, or call objects. If I see any basic case I just return that; we can't simplify those further. For call objects, I call a function, `simplify_call` responsible for handling calls.

```
simplify_expr <- function(expr) {
  if (is.atomic(expr) || is.name(expr)) {
    expr
  } else if (is.call(expr)) {
    simplify_call(expr)

  } else {
    stop(paste0("Unexpected expression ",
                deparse(expr),
                " in simplifying"))
  }
}
```

For call simplification, I don't attempt to simplify function calls. I don't know what any generic function is doing, so there is little I can do to simplify expressions that involve functions. I will assume, though, that if I am simplifying an expression, then functions in it behave as if they had call-by-value semantics and simplify their arguments. This is an assumption, it might be wrong, but for this exercises, I will assume it. So for general function calls, I will just simplify their arguments. For arithmetic expressions, I will try to simplify those further. I could also attempt to do that for other operations, but handling just the arithmetic operators show how we would handle operators in sufficient detail that I trust you, dear reader, to be able to handle other operators if you need to.

Call handling can then look like this:

```
simplify_call <- function(expr) {
  if (expr[[1]] == as.name("+"))
    return(simplify_addition(expr[[2]], expr[[3]]))
  if (expr[[1]] == as.name("-")) {
    if (length(expr) == 2)
      return(simplify_unary_subtraction(expr[[2]]))
    else
      return(simplify_subtraction(expr[[2]], expr[[3]]))
  }

  if (expr[[1]] == as.name("*"))
    return(simplify_multiplication(expr[[2]], expr[[3]]))
  if (expr[[1]] == as.name("/"))
    return(simplify_division(expr[[2]], expr[[3]]))

  if (expr[[1]] == as.name("^"))
```

```
    return(simplify_exponentiation(expr[[2]], expr[[3]]))

  if (expr[[1]] == as.name("(")) {
    subexpr <- simplify_expr(expr[[2]])
    if (is.atomic(subexpr) || is.name(subexpr))
      return(subexpr)
    else if (is.call(subexpr) && subexpr[[1]] == as.name("("))
      return(subexpr)
    else
      return(call("(", subexpr))
  }

  simplify_function_call(expr)
}
```

It is mostly self-explanatory, but a few comments are in order: First, we need to compare the call names with name objects. They are not actually character strings but have type `name`; thus we need to use `as.name`. Second, minus come in two flavours, binary subtraction and a unary negation. We can tell the two apart by checking if the call has one or two arguments (i.e., whether it has length two or three; remember that the first element is the call name), and we just use two different functions to handle the two cases. Third, parentheses are also calls, so we need to handle them. We just get hold of the expression inside the parentheses. If this is something that doesn't need parentheses—single values, names, or an expression already surrounded by parentheses—we just return that sub-expression. Otherwise, we put parentheses around it. Finally, if we don't know what else to do, we just treat the expression as a function call.

Now we just handle each operator in turn. They are all handled very similarly, only differing in what we can simplify given each operator. For addition, we can get rid of addition by zero, and if both our arguments are numbers we can evaluate them right away, otherwise we need to return a call to `+` with simplified operands:

```
simplify_addition <- function(f, g) {
  left <- simplify_expr(f)
  right <- simplify_expr(g)
  if (left == 0) return(right)
  if (right == 0) return(left)
  if (is.numeric(left) && is.numeric(right))
    return(left + right)
  call("+", left, right)
}
```

Unary minus we can just evaluate if its argument is numeric. Otherwise, we can get rid of an existing minus in the argument since two minuses make a plus, and if all else fails we just have to return the simplified expression with a minus in front of it:

```
simplify_unary_subtraction <- function(f) {
   simplified <- simplify_expr(f)
   if (is.numeric(simplified))
     -simplified
   else if (is.call(simplified) && simplified[[1]] == "-")
     simplified[[2]]
   else
     bquote(-.(simplified))
}
```

For the final case, here, we use the function `bquote`. It works similar to `quote` but substitutes a value in where we put `.(...)`. So we essentially write `quote(-simplified)` except that we put the simplified expression inside the expression.

Binary subtraction is similar to addition but with a little more work when we subtract from zero. Here we need to use `bquote` again:

```
simplify_subtraction <- function(f, g) {
  left <- simplify_expr(f)
  right <- simplify_expr(g)
  if (left == 0) {
    if (is.numeric(right))
      return(-right)
    else
      return(bquote(-.(right)))
  }
  if (right == 0)
    return(left)
  if (is.numeric(left) && is.numeric(right))
    return(left - right)
  call("-", left, right)
}
```

For multiplication, we can simplify cases where the multiplication involves zero or one, but otherwise, the function looks very similar to what we have seen before:

```
simplify_multiplication <- function(f, g) {
  left <- simplify_expr(f)
  right <- simplify_expr(g)
  if (left == 0 || right == 0)
    return(0)
  if (left == 1)
    return(right)
  if (right == 1)
    return(left)
  if (is.numeric(left) && is.numeric(right))
    return(left * right)
  call("*", left, right)
}
```

Division and exportation are just more of the same, with different cases to handle:

```
simplify_division <- function(f, g) {
  left <- simplify_expr(f)
  right <- simplify_expr(g)
  if (right == 1)
    return(left)
  if (is.numeric(left) && is.numeric(right))
    return(left / right)
  call("/", left, right)
}

simplify_exponentiation <- function(f, g) {
  left <- simplify_expr(f)
  right <- simplify_expr(g)
  if (right == 0) return(1)
  if (left == 0) return(0)
  if (left == 1) return(1)
  if (right == 1) return(left)
  if (is.numeric(left) && is.numeric(right))
    return(left ^ right)
  call("^", left, right)
}
```

The final function we need is a function-call simplification. Here we just have to simplify all the function's arguments before returning a call. We can collect the arguments in a list and create a function call with an expression like:

```
  do.call("call", c(list(function_name), arguments))
```

This would take the arguments, as a list, and make them into arguments in a call to `call`. This will work fine if the `function_name` is a function name, but expressions such as `f(x,y)(z)` are also function calls; here the function "name" is `f(x,y)` and the argument is `z`. We cannot wrap such an expression up in a call to `call`, but we can just take a list and make it into a call using `as.call`:

```
simplify_function_call <- function(expr) {
  function_name <- expr[[1]]
  arguments <- vector("list", length(expr) - 1)
  for (i in seq_along(arguments)) {
    arguments[i] <- list(simplify_expr(expr[[i + 1]]))
  }
  as.call(c(list(function_name), arguments))
}
```

For the same reason, we have to remedy the `simplify_call` function. There, we compare `expr[[1]]` with names to dispatch to the various arithmetic operators. This only works if `expr[[1]]` is a name, so we have to make sure that we only make these comparisons when it is:

```
simplify_call <- function(expr) {
  if (is.name(expr[[1]])) {
    # Dispatch to operators...
  }

  simplify_function_call(expr)
}
```

We could also get a little more ambitious and try to evaluate functions when all their arguments are values and when we know what the functions are—or at least have a reasonable expectation that we would know. We could always check if we can find the name in a relevant environment, and if it is a function, but since we are simplifying expressions where we don't expect to know variables that are not functions, it is probably too much to demand that all function symbols are known. Still, we could say that functions such as `sin` and `cos`, `exp` and `log` are their usual selves and then do something like this:

```
simplify_function_call <- function(expr) {
  function_name <- expr[[1]]
  arguments <- vector("list", length(expr) - 1)
  for (i in seq_along(arguments)) {
    arguments[i] <- list(simplify_expr(expr[[i + 1]]))
  }

  if (all(unlist(Map(is.numeric, arguments)))) {
    if (as.character(function_name) %in%
                c("sin", "cos", "exp", "log")) {
      result <- do.call(as.character(function_name), arguments)
      return(result)
    }
  }
  as.call(c(list(function_name), arguments))
}
```

We now have a simple program that lets us simplify expressions to a certain extend:

```
simplify_expr(quote(2*(0 + ((4 + 5)*x)*1)))
```

```
## 2 * (9 * x)
```

Neither function-call solution can handle named arguments. We simply work with positional arguments. We just throw away the name information.

```
f <- function(x, y) x
expr1 <- quote(f(x = 2, y = 1))
expr2 <- quote(f(y = 2, x = 1))
eval(expr1)
```

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

```
eval(expr2)
```

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

```
simplify_expr(expr1)
```

```
## f(2, 1)
```

```
simplify_expr(expr2)
```

```
## f(2, 1)
```

```
eval(simplify_expr(expr1))
```

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

```
eval(simplify_expr(expr2))
```

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

It isn't hard to remedy this, though. There is nothing special needed to work with named arguments when we deal with function calls; they are just accessed with the `named` function:

```
names(expr1)
```

```
## [1] ""  "x" "y"
```

```
names(expr2)
```

```
## [1] ""  "y" "x"
```

If we make sure that the result of our simplification gets the same name as the original expression, we will be okay:

```
simplify_function_call <- function(expr) {
  function_name <- expr[[1]]
  arguments <- vector("list", length(expr) - 1)
  for (i in seq_along(arguments)) {
    arguments[i] <- list(simplify_expr(expr[[i + 1]]))
  }
  result <- as.call(c(list(function_name), arguments))
  names(result) <- names(expr)
  result
}
```

```
simplify_expr(expr1)
```

```
## f(x = 2, y = 1)
```

```
simplify_expr(expr2)
```

```
## f(y = 2, x = 1)
```

```
eval(simplify_expr(expr1))
```

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

```
eval(simplify_expr(expr2))
```

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

Automatic differentiation

As a second, only slightly more involved, example, we consider *automatic differentiation:* automatically translating a function that computes an expression into a function that calculates the derived expression. We will assume that we have a function whose body contains only a single expression—one that doesn't involve control structures or sequences of statements but just a single arithmetic expression—and recurse through this expression, applying the rules of differentiation. Although what we do with this meta-program is more complicated than the expression simplification we just implemented, you will see that the form of the program is very similar.

We start with the main function, which we name `d` for differentiation. It takes two arguments: the function to be differentiated and the variable to take the derivative on. If we want the function to be able to handle the built-in mathematical functions we need to handle these as special cases. These are implemented as so-called *primitive* functions and do not have a body. We need to handle them explicitly in the `d` function. For all other functions, we just need to compute the derivative of the expression in the function body. If we want to return a new function for the derivative, we can just take the function we are modifying and replace its body. Since R doesn't let us modify arguments to a function, this will just create a copy we can return and leave the original function intact. Reusing the argument this way make sure that the new function has the same arguments, with the same names and same default values, as the original. It also ensures that the derivative will have the same enclosing environment as the original function, which is potentially important for when we evaluate it.

The `d` function can look like this, where I've only handled three of the primitive functions—you can add the remaining as an exercise:

```
d <- function(f, x) {
  if (is.null(body(f))) {
    if (identical(f, sin)) return(cos)
    if (identical(f, cos)) return(function(x) -sin(x))
    if (identical(f, exp)) return(exp)

    stop("unknown primitive")

  } else {
    e <- environment(f)
    body(df) <- simplify_expr(diff_expr(body(f), x, e))
    df
  }
}
```

We send the function environment along with the recursion because we will need it when we have to deal with function calls later. There, we will need to look up functions and analyse which parameters they take to apply the chain rule. For now, we just pass it along in the recursion.

For aesthetic reasons, we simplify the expression we get from differentiating the body of `f`, using the code we wrote in the previous section. We can use `d` like this:

```
f <- function(x) x^2 + sin(x)
df <- d(f, "x")
df
## function (x)
## 2 * x + cos(x)
```

For computing the derivative of the function body, we follow the pattern we used for the expression simplification: we write a recursive function for dealing with expressions, where we dispatch function calls to different cases for the different arithmetic operations.

The two basic cases for the recursive function are numbers and names—we assume that we do not get other atomic values such as logical vectors; we wouldn't know how to differentiate them anyway. For numbers, the derivative is always zero while for names it depends on whether we have the variable we are computing the derivative on or another variable. The recursive case for the function is function calls, where we just call another function to handle that case.

```
diff_expr <- function(expr, x, e) {
  if (is.numeric(expr)) {
    quote(0)

  } else if (is.name(expr)) {
    if (expr == x) quote(1)
    else quote(0)

  } else if (is.call(expr)) {
    diff_call(expr, x, e)

  } else {
    stop(paste0("Unexpected expression ",
                deparse(expr), " in parsing."))
  }
}
```

For calls, we dispatch based on the type of call, so we deal with arithmetic expressions through a function for each operator, we deal with parentheses similar to how we handled them in the expression simplification, and for differentiating other function calls. We have to handle primitive functions and user defined functions as two separate cases here as well. For user-defined functions, we can analyse these, figure out their formal arguments, and apply the chain rule. For primitive functions, `formals` will give us an empty list, so that strategy will not work for those. So we handle them as a special case. I assume, here, that we have a list of names of the primitive functions. E.g. we could have

```
.built_in_functions <- c("sin", "cos", "exp")
```

If we need to handle only those three cases. Extend it as needed.

The function handling calls looks like this:

```
diff_call <- function(expr, x, e) {
  if (is.name(expr[[1]])) {
    if (expr[[1]] == as.name("+"))
      return(diff_addition(expr[[2]], expr[[3]], x, e))

    if (expr[[1]] == as.name("-")) {
      if (length(expr) == 2)
        return(call("-", diff_expr(expr[[2]], x, e)))
      else
        return(diff_subtraction(expr[[2]], expr[[3]], x, e))
    }

    if (expr[[1]] == as.name("*"))
      return(diff_multiplication(expr[[2]], expr[[3]], x, e))
    if (expr[[1]] == as.name("/"))
      return(diff_division(expr[[2]], expr[[3]], x, e))

    if (expr[[1]] == as.name("^"))
      return(diff_exponentiation(expr[[2]], expr[[3]], x, e))

    if (expr[[1]] == as.name("(")) {
      subexpr <- diff_expr(expr[[2]], x, e)
      if (is.atomic(subexpr) || is.name(subexpr))
        return(subexpr)
      else if (is.call(subexpr) && subexpr[[1]] == as.name("("))
        return(subexpr)
      else
        return(call("(", subexpr))
    }
  }

  if (is.name(expr[[1]]) &&
      as.character(expr[[1]]) %in% .built_in_functions)
    return(diff_built_in_function_call(expr, x, e))
  else
    return(diff_general_function_call(expr, x, e))
}
```

We handle the arithmetic operations just by following the rules we learned in calculus class:

```
diff_addition <- function(f, g, x, e) {
  call("+", diff_expr(f, x, e), diff_expr(g, x, e))
}

diff_subtraction <- function(f, g, x, e) {
  call("-", diff_expr(f, x, e), diff_expr(g, x, e))
}
```

```
diff_multiplication <- function(f, g, x, e) {
  # f’ g + f g’
  call("+",
       call("*", diff_expr(f, x, e), g),
       call("*", f, diff_expr(g, x, e)))
}

diff_division <- function(f, g, x, e) {
  # (f’ g âĹŠ f g’ )/g**2
  call("/",
       call("-",
        call("*", diff_expr(f, x, e), g),
        call("*", f, diff_expr(g, x, e))),
       call("^", g, 2))
}

diff_exponentiation <- function(f, g, x, e) {
  # Using the chain rule to handle this generally.
  dydf <- call("*", g,
               call("^", f, substitute(n - 1, list(n = g))))
  dfdx <- diff_expr(f, x, e)
  call("*", dydf, dfdx)
}
```

For function calls, we have to apply the chain rule. For primitive functions we cannot get a list of formal arguments so we cannot handle these by inspecting the functions; we have to use their names to figure out what their arguments and derivatives are. I’ve shown a few cases below, but I will leave handling other functions as an exercise for the reader.

```
diff_built_in_function_call <- function(expr, x, e) {
  # chain rule with a known function to differentiate...
  if (expr[[1]] == as.name("sin"))
    return(call("*", call("cos", expr[[2]]),
                diff_expr(expr[[2]], x, e)))

  if (expr[[1]] == as.name("cos"))
    return(call("*", call("-", call("sin", expr[[2]])),
                diff_expr(expr[[2]], x, e)))
  if (expr[[1]] == as.name("exp"))
    return(call("*", call("exp", expr[[2]]),
                diff_expr(expr[[2]], x, e)))
}
```

For other function calls, we can inspect the function to work out which variables it has and apply the chain rules to those. This only works if we can figure out which function we are referring to, so we cannot handle cases where we have to compute it. In those cases, we just give up. If we have a symbol for the function, however, we can look it up and inspect it. This isn't *entirely* safe for general use. If we calculate the derivative of a function, then change a global function that it refers to, we will have a derivative that uses the old global function and while the actual function uses the new global function. There isn't much we can do about this, though. At the point where we apply the chain rule, we need to know which arguments the function takes. That means that we need to know which function we are working with.

I will assume that the arguments used in the function call are the relevant ones to consider when we apply the chain rules. Those that we are not passing along in the function call will have default values and will not depend on the arguments given to the derivative function so that we can ignore them. Therefore, we can take the arguments in the function call and sum over those in the chain rule. We need to know the names of the arguments to compute the derivatives of the function, and we need to handle both positional and named arguments, and this is where we have to look up the actual function.

In the environment we have passed along in the recursion—the environment of the original function we are computing the derivative of—we look up the function we have to apply the chain rule to. With that function in hand, we can use the function `match.call` to get all the names of the arguments in the function call. The `match.call` function takes care of merging named and positional arguments. For each argument, we build a function call by changing the function to its derivative to the appropriate variable. We use the `bquote` function to call `d` to compute these derivatives. We then multiply the function call with the argument differentiated with the original variable. Collecting all these terms in a sum completes the chain rule.

```
diff_general_function_call <- function(expr, x, e) {
  function_name <- expr[[1]]
  if (!is.name(function_name))
    stop(paste0("Unexpected call ", deparse(expr)))

  func <- get(as.character(function_name), e)
  full_call <- match.call(func, expr)
  variables <- names(full_call)

  arguments <- vector("list", length(full_call) - 1)
  for (i in seq_along(arguments)) {
    var <- variables[i + 1]
    dfdz <- full_call
    dfdz[[1]] <- bquote(d(.(function_name), .(var)))
    dzdx <- diff_expr(expr[[i + 1]], x, e)
    arguments[[i]] <- bquote(.(dfdz) * .(dzdx))
  }
  as.call(c(list(sum), arguments))
}
```

There is one caveat with this solution: even if the original function is vectorised, the derivative won't be. If we define these functions

```
f <- function(x, y) x^2 * y
g <- function(z) f(2*z, z^2)
h <- function(z) 4*z^4
```

then `g` and `h` should be the same functions. However, if we calculate `d(g,"z")` and `d(h,"z")` and call them with a vector of values, the former will add all the results together while the latter will return a vector of values. The `sum` call in the derivative of `g` will gobble up all the values. You can fix this by calling `Vectorize` on `d(g,"z")`.

Other than that, we now have a meta-program for translating a function into its derivative. It doesn't handle all possible functions; it has to be functions that evaluate simple expressions. The chain rule can only be applied to known functions mentioned by name, and we have only handled some of the primitive functions, but I trust you can see how you could build more functionality on top of what we have now.

Working with substitutions

We can take expressions and manipulate them by treating them as strings, but we can also modify them by substituting variables for expressions, we can build expressions by more advanced quoting, and we can move back and forth between strings and expressions.

A little more on quotes

We start this chapter by quickly revisiting the quote mechanism. We have already seen how to quote expressions, but you can do more than just create verbatim expressions. We have discussed the `quote` function in some detail, but only seen the `bquote` function in passing. The `bquote` function allows you create expressions where you only partially quote—you can evaluate some values when you create the expression and leave other parts for later. While `quote` wraps an entire expression in quotes, the `bquote` function does partial substitution in an expression. You call it with an expression, as you would call `quote`, but any sub-expression you wrap in a call to "dot", that is, you write `.(...)`, will not be quoted but will instead be evaluated and the value will be put into the expression you are constructing. If you just use `bquote` without using `.(...)`, it works just as `quote`.

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

```
## x + y
```

```
bquote(x + y)
```

```
## x + y
```

```
bquote(.(2+2) + y)
```

```
## 4 + y
```

We used `bquote` when we constructed terms for the chain rule in our differentiation program. Here we used `bquote(.(dfdz) * .(dzdx))` to construct the product of the differentiated function and the differentiated argument; we constructed `dfdz` and `dzdx` by constructing a differentiated function, using `bquote` again, for `dfdz`—`bquote(d(.(function_name), .(var)))`—and by calling the `diff_expr` function for `dzdx`.

In many situations where you need to create a call object, using `bquote` can be much simpler than constructing the call object explicitly. For the simple example from above, there is not a huge difference

```
bquote(.(2+2) + y)
```

```
## 4 + y
```

```
call("+", 2+2, quote(y))
```

```
## 4 + y
```

but once you start creating expressions with many nested calls—and you don't need particularly complex expressions to need multiple nested calls—then explicitly creating calls become much more cumbersome than using `bquote`.

```
bquote((.(2+2) + x) * y)
```

```
## (4 + x) * y
```

```
call("*", call("(", call("+", 2+2, quote(x))), quote(y))
```

```
## (4 + x) * y
```

Parsing and deparsing

When we use quoting to construct expressions, we get objects we can manipulate via recursive functions, but we can also work with expressions as strings and translate between strings and expressions using the functions `parse` and `deparse`.

The `deparse` function translates an expression into the R source code that would be used to create the expression—represented as a string—and the `parse` function parses a string or a file and returns the expressions in it.

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

```
## [1] "x + y"
```

```
parse(text = "x + y")
```

```
## expression(x + y)
```

For the call to `parse`, here, we need to specify that we are parsing a text (string). Otherwise, `parse` will assume that we are giving it a file name and try to parse the content of that file. The result of the call to `parse` is not, strictly speaking, and expression. Although the type it writes is `expression`. That is an unfortunate choice for this type because `expression` objects are actually lists of expressions. The `parse` function can parse more than one expression, and sequences of expressions are represented in the `expression` type. You can get the *actual* expressions by indexing into this object.

```
(expr <- parse(text = "x + y; z * x"))
```

```
## expression(x + y, z * x)
```

```
expr[[1]]
```

```
## x + y
```

```
expr[[2]]
```

```
## z * x
```

The `departs` function is often used when you need a string that represents an R object—for example for a label in a plot. Many functions extract expressions given as arguments and use those as default labels. There, you cannot just use `deparse`. That would evaluate the expression you are trying to departs before you turn it into a string. You need to get the actual argument, and for that, you need the `substitute` function.

```
f <- function(x) deparse(x)
g <- function(x) deparse(substitute(x))

x <- 1:4; y <- x**2
f(x + y)
```

```
## [1] "c(2, 6, 12, 20)"
```

```
g(x + y)
```

```
## [1] "x + y"
```

Substitution

The `substitute` function replaces variables for values in an expression. The `deparse(substitute(x))` construction we just saw exploits this by using `substitute` to get the expression that the function parameter `x` refers to before translating it into a string. If we just refer to `x` will well force an evaluation of the argument and get the value it evaluates to; instead, because we use `substitute`, we get the expression that `x` refers to.

Getting the expression used as a function argument, rather than the value of the expression, is a common use of `substitute`. Together with `deparse`, it is used to create labels for plots. It is also used for so-called *non-standard evaluation*—functions that do not evaluate their arguments following the default rules for environments. Non-standard evaluation, which we return to in the next section, obtains the expressions in arguments using `substitute` and then evaluating them, using `eval` in environments different from the function's evaluation environment.

Before we consider evaluating expressions, however, we should get a handle on how `substitute` works. This depends a little bit on where it is called. In the global environment, `substitute` doesn't do anything. At least not unless you give it more arguments than the expression—we get to that shortly. In all other environments, if you just call `substitute` with an expression, the function will search through the expression and find variables. If it finds a variable that has a value in the current environment—whether it is a promise for a function call or a variable we have assigned values to—it will substitute the variable with the value. If the variable does not have a value in the environment, it is left alone. In the global environment, it leaves all variables alone.

In the example below, we see that `substitute(x + y)` doesn't get modified in the global environment, even though the variables `x` and `y` are defined. Inside the function environment for `f`, however, we substitute the two variables with their values.

```
x <- 2; y <- 3
substitute(x + y)
```

```
## x + y
```

```
f <- function(x, y) substitute(x + y)
f(2, 3)
```

```
## 2 + 3
```

With `substitute`, variables are not found the same way as they are in `eval`. When `substitute` looks in an environment, it does not follow the parent pointer. If it doesn't find the variable to substitute in the exact environment in which it is called, it will not look further. So, if we write functions like these:

```
y - 3
```

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

```
f <- function(x) substitute(x + y)
f(2)
```

```
## 2 + y
```

```
g <- function(x) function(y) substitute(x + y)
h <- g(2)
h(3)
```

```
## x + 3
```

the function `f`, when called, will have `x` in its evaluation environment and `y` in the parent environment—which is the global environment—but `substitute` will only substitute the local variable, `x`. For `h`, it will know `y` as a local variable and `x` from its closure, but only `y`, the local variable, will be substituted.

The actual environment that `substitute` use to find variables is given as its second argument. The default is just the current evaluating environment. We can change that by providing either an environment or a list with a variable to value mapping.

```
e <- new.env(parent = emptyenv())
e$x <- 2
e$y <- 3
substitute(x + y, e)
```

```
## 2 + 3
```

```
substitute(x + y, list(x = 2, y = 3))
```

```
## 2 + 3
```

Again, `substitute` will not follow parent pointers, whether these are set implicitly or explicitly in the environment, we pass on to the function.

```
x <- 2 ; y <- 3
e <- new.env(parent = globalenv())
substitute(x + y, e)
```

```
## x + y
```

```
e <- new.env(parent = globalenv())
e$x <- 2
e$y <- 3
e2 <- new.env(parent = e)
substitute(x + y, e2)
```

```
## x + y
```

If you want a variable substituted, you need to make sure it is in the exact environment you provide to `substitute`.

Substituting expressions held in variables

A common case when you manipulate expressions is that you have a reference to an expression—for example from a function argument—and you want to modify it. In the global environment, you cannot do this directly with `substitute`. If you give `substitute` a variable, it will just return that variable.

```
expr <- quote(x + y)
substitute(expr)
```

```
## expr
```

This is because `substitute` doesn't replace the variable in the global environment. You can get the expression substituted by explicitly giving `substitute` the expression in an environment or a list:

```
substitute(expr, list(expr = expr))
```

```
## x + y
```

Usually, though, you don't manipulate expressions in the global environment, and inside a function you *can* substitute an expression:

```
f <- function() {
  expr <- quote(x + y)
  substitute(expr)
}
f()
```

```
## x + y
```

But what if you want to replace, say, `y` with 2 in the expression here. The substitution, both in the global environment with an explicit list or inside a function, will replace `expr` with `quote(x + y)`, but you want to take that then and replace `y` with 2. You cannot just get `y` from the local environment and give it to `substitute` explicitly won't work either.

```
f <- function() {
  expr <- quote(x + y)
  y <- 2
  substitute(expr)
}
f()
```

```
## x + y
```

```
f <- function() {
  expr <- quote(x + y)
  substitute(expr, list(y = 2))
}
f()
```

```
## expr
```

What you want to do is, first replace `expr` with the expression `quote(x + y)` and then replace `y` with 2. So the natural approach is to write this code, that will not work:

```
substitute(substitute(expr, list(expr = expr)), list(y = 2))
```

```
## substitute(expr, list(expr = expr))
```

The problem here is that `y` doesn't appear anywhere in the expression given to the outermost `substitute`, so it won't be substituted in anywhere. What you get is just the expression

```
substitute(expr, list(expr = expr))
```

which you can evaluate to get `x + y`

```
eval(substitute(substitute(expr, list(expr = expr)), list(y = 2)))
```

but the evaluation *first* substitutes `y` into the inner-most `substitute` expression—where there is no `y` variable—and *then* substitutes `expr` into the expression `expr`. The order is wrong.

To substitute variables in an expression, you hold in another variable you have to write the expression in the opposite order of what comes naturally. You don't want to substitute `expr` at the inner-most level and then `y` at the outer-most level; you want to first substitute `expr` into a substitute expression that takes care of substituting `y`. The outer-most level substitutes `expr` into `substitute(expr, list(y = 2))` which you can evaluate to get `y` substituted into the expression.

So we create the expression we need to evaluate like this:

```
substitute(substitute(expr, list(y = 2)), list(expr = expr))
```

```
## substitute(x + y, list(y = 2))
```

and we complete the substitute like this:

```
eval(substitute(substitute(expr, list(y = 2)), list(expr = expr)))
```

```
## x + 2
```

It might take a little getting used to, but you just have to remember that you need to do the substitutions in this order.

Substituting function arguments

Function arguments are passed as unevaluated promises, but the second we access them, they get evaluated. If you want to get hold of the promises without evaluating them, you can use the `substitute` function. This gives you the argument as an unevaluated—or quoted—expression.

This can be useful if you want to manipulate expressions or evaluate them in ways different from the norm—as we explore in the next section—but you do throw away information about which context the expression was supposed to be evaluated in. Consider the example below:

```
f <- function(x) function(y = x) substitute(y)
g <- f(2)
g()
```

```
## x
```

```
x <- 4
g(x)
```

```
## x
```

In the first call to `g`, `y` has its default parameter, which is the one it gets from its closure, so it substitutes to the `x` that has the value 2. In the second call, however, we have the expression `x` from the global environment where `x` is 4. In both cases, however, we just have the expression `quote(x)`. From inside R, there is no mechanism for getting the environment out of a promise, so you cannot write code that modifies input expressions and then evaluates them in the enclosing scope for default parameters and the calling scope for function arguments.[1]

You also have to be a little careful when you use `substitute` in functions that are called with other functions. The expression you get when you `substitute` is the exact expression a function gets called with. This expression doesn't propagate through other functions. In the example below, we call the function

[1]In the package `pryr`, which we return to at the end of this chapter, there are functions, written in C, that does provide access to the internals of promises. Using `pryr`, you *can* get hold of both the expression and associated environment of a promise. In case you need it.

g with the expression x + y, but since g calls f with expr, that is what we get in the substitution.

```
f <- function(expr) substitute(expr)
f(x + y)

## x + y

g <- function(expr) f(expr)
g(x + y)

## expr

x <- 2; y <- 3
eval(f(x + y))

## [1] 5

eval(g(x + y))

## x + y
```

The substitute function is harder to use safely and correctly than is using bquote and explicitly modifying call objects, but it is the function you need to use to implement non-standard evaluation.

Non-standard evaluation

Non-standard evaluation refers to any evaluation that doesn't follow the rules for how you evaluate expressions in the local evaluation environment. When we use eval to evaluate a function argument in an environment other than the default, which is what we get from a call to environment(), we are evaluating an expression in a non-standard way.

Typical uses of non-standard evaluation, or NSE, are evaluating expressions in the calling scope, which we have already seen examples of, and evaluating expressions in data frames. We have already seen that we can use a list to provide a variable to value mapping when using substitute, but we can also do the same when using eval.

```
eval(quote(x + y), list(x = 2, y = 3))
```

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

Since a data frame is just a list of vector elements of the same length, we can also evaluate expressions in the context of these.

```
d <- data.frame(x = 1:2, y = 3:3)
eval(quote(x + y), d)
```

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

When we use `eval` this way, where we explicitly quote the expression, we are not really doing NSE. The quoted expression would not be evaluated in any other, standard, way. After all, we explicitly quote it, and if we didn't quote it here, `x+y` would be evaluated in the calling scope, not inside the data frame.

```
x <- 2; y <-  3
eval(x + y, d)
```

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

To do NSE, we have to explicitly substitute an argument, so we do not evaluate the argument-promise in the calling scope, and then evaluate it in an alternative scope. For example, we can implement our own version of the `with` function like this:

```
my_with <- function(df, expr) {
  eval(substitute(expr), df)
}
d <- data.frame(x = rnorm(5), y = rnorm(5))
my_with(d, x + y)
```

```
## [1] -1.4437635  0.9121930 -1.8908632 -0.2156309
## [5]  1.8495491
```

Here, the expression `x + y` is *not* quoted in the function call, so normally we would expect `x + y` to be evaluated in the calling scope. Because we explicitly `substitute` the argument in `my_with` this does not happen. Instead, we evaluate the expression in the context of the data frame. This is non-standard evaluation.

The real `with` function works a little better than our version. If the expression we evaluate contains variables that are not found in the data frame, then it takes these variables from the calling scope. Our version can also handle variables that do not appear in the data frame, but it works slightly differently.

If we use the two functions in the global scope we don't see a difference:

```
z <- 1
with(d, x + y + z)
```

```
## [1] -0.4437635  1.9121930 -0.8908632  0.7843691
## [5]  2.8495491
```

```
my_with(d, x + y + z)
```

```
## [1] -0.4437635  1.9121930 -0.8908632  0.7843691
## [5]  2.8495491
```

but if we use them inside functions, we do:

```
f <- function(z) with(d, x + y + z)
f(2)
```

```
## [1] 0.5562365 2.9121930 0.1091368 1.7843691
## [5] 3.8495491
```

```
g <- function(z) my_with(d, x + y + z)
g(2)
```

```
## [1] -0.4437635  1.9121930 -0.8908632  0.7843691
## [5]  2.8495491
```

What is going on here?

Well, `eval` takes a third argument that gives the enclosing scope for the evaluation. In `my_with` we haven't provided this, so we use the default value, which is the enclosing scope where we call `eval`, which is the evaluating environment of `my_with`. We haven't defined `z` in this environment, but the enclosing scope includes the global environment where we have. When we evaluate the expression in `my_with`, we find `z` in the global environment. In contrast, when we use `with`, the enclosing environment is the calling environment.

We can change `my_with` to have the same behaviour thus:

```
my_with <- function(df, expr) {
  eval(substitute(expr), df, parent.frame())
}

f <- function(z) with(d, x + y + z)
f(2)

## [1] 0.5562365 2.9121930 0.1091368 1.7843691
## [5] 3.8495491

g <- function(z) my_with(d, x + y + z)
g(2)

## [1] 0.5562365 2.9121930 0.1091368 1.7843691
## [5] 3.8495491
```

Now, we have both typical uses of NSE: evaluating in a data frame and evaluating in the calling scope.

Non-standard evaluation from inside functions

Non-standard evaluation is very hard to get right once you start using it from inside other functions. It is a convenient approach to simplify the syntax for many operations when you work with R interactively or when you write analysis pipelines in the global scope, but because substitutions tend to work verbatim on the function arguments you give functions, once arguments get passed from one function to another, NSE gets tricky.

Consider the example below:

```
x <- 2; y <- 3
f <- function(d, expr) my_with(d, expr)
f(d, x + y)

## [1] 5

g <- function(d, expr) my_with(d, substitute(expr))
g(d, x + y)

## expr
```

Here, we make two attempts at using `my_with` from inside a function, and neither work as intended. In `f`, the `expr` gets evaluated in the global scope. When we use the variable inside the function, the promise gets evaluated before it is passed along to `my_with`. In `g`, we do substitute, but it is `substitute(expr)` that `my_with` sees—remember, it does not see the expression as a promise but substitutes it to get an expression—so we don't actually get the argument substituted. The NSE in `my_with` prevents this.

If you want functions that do NSE, you really should write functions that work with expressions and do "normal" evaluation on those instead. We can make a version of `my_with` that expects the expression to be already quoted, which we can use in other functions, and then define `my_with` to do the NSE like this:

```
my_with_q <- function(df, expr) {
  eval(expr, df, parent.frame())
}
my_with <- function(df, expr) my_with_q(d, substitute(expr))

g <- function(d, expr) my_with_q(d, substitute(expr))
g(d, x + y)

## [1] -1.4437635  0.9121930 -1.8908632 -0.2156309
## [5]  1.8495491

my_with(d, x + y)

## [1] -1.4437635  0.9121930 -1.8908632 -0.2156309
## [5]  1.8495491
```

Writing macros with NSE

Since NSE allows you to evaluate expressions in the calling scope, you can use it write macros, that is, functions that work as shortcuts for statements where they are called. These differ from functions, that cannot generally modify data outside of their own scope, and I will not recommend using them unless you have very good reasons to—the immutability of data is a very important feature of R and violating it with macros goes against this—but because it is a good example of what you can do with NSE I will include the example here.

Consider the code below. This example is a modified implementation of the macro code presented in [*Programmer's Niche: Macros in R*(https://www.r-project.org/doc/Rnews/Rnews_2001-3.pdf) by Thomas Lumley, R Journal, 2001]. I have simplified the macro making function a bit; you can see the full version in the original article online.

```
make_param_names <- function(params) {
  param_names <- names(params)
  if (is.null(param_names))
    param_names <- rep("", length(params))
  for (i in seq_along(param_names)) {
    if (param_names[i] == "") {
      param_names[i] <- paste(params[[i]])
    }
  }
  param_names
}

make_macro <- function(..., body) {
  params <- eval(substitute(alist(...)))
  body <- substitute(body)

  # Construct macro
  f <- eval(substitute(
    function() eval(substitute(body), parent.frame())
  ))

  # Set macro arguments
  param_names <- make_param_names(params)
  names(params) <- param_names
  params <- as.list(params)
  formals(f) <- params

  f
}
```

The first function simply extracts the names from a list of function parameters and make sure that all parameters have a name. This is necessary later when we construct the `formals` list of a function. In a `formals` list, all parameters must have a name, and we construct these names with this function. The second function is the interesting one. Here we construct a macro by constructing a function whose body will be evaluated in its calling scope. We first get hold of the parameters and body the macro should have. The parameters we get into a `list` by substituting `...` in and then evaluating the `alist(...)` expression. Without the substitution we would get the argument list that just contains `...` but with the substitution we get the arguments that are passed to the `make_macro` function, except for the named argument `body` that we just translate into the expression passed to the `make_macro` with another `substitute` call.

The function we construct is where the magic happens. We create the expression

```
substitute(function() eval(substitute(body), parent.frame()))
```

where `body` will be replaced with the expression that we pass to the macro. When we evaluate this, we get the function

```
function() eval(substitute(<body>), parent.frame())
```

where <`body`> is not the symbol `body` but the `body` expression. This is a function that doesn't take any arguments (yet) but will substitute variables in <`body`> that are known in its calling scope—the `parent.frame()`—before evaluating the resulting expression.

We can see this in action with this small example:

```
(m <- make_macro(body = x + y))
```

```
## function ()
## eval(substitute(x + y), parent.frame())
## <environment: 0x7ff916cc2778>
```

When we call `m` it will evaluate `x + y` in its calling scope, so we can set the variables `x` and `y` in the global scope and evaluate it thus:

```
x <- 2; y <- 4
m()
```

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

The last part of the `make_macro` code sets the formal arguments of the macro. It simply takes the parameters we have specified, makes sure they all have a name, and then make them into a list and sets `formals(f)`. After that, `make_macro` returns the constructed function.

We can use this function to create a macro that replaces specific values in a column in a data frame with `NA` like this:

```
set_NA_val <- make_macro(df, var, na_val,
                         body = df$var[df$var == na_val] <- NA)
```

The macro we construct takes three parameters, the data frame, `df`, the variable (column) in the data frame, `var`, and the value that corresponds to `NA`, `na_val`. Its body then is the expression

```
df$var[df$var == na_val] <- NA
```

with `df`, `var`, and `na_val` replaced by the arguments passed to the macro. We can use it like this:

```
(d <- data.frame(x = c(1,-9,3,4), y = c(1,2,-9,-9)))
```

```
##    x  y
## 1  1  1
## 2 -9  2
## 3  3 -9
## 4  4 -9
```

```
set_NA_val(d, x, -9); d
```

```
##    x  y
## 1  1  1
## 2 NA  2
## 3  3 -9
## 4  4 -9
```

```
set_NA_val(d, y, -9); d
```

```
##    x  y
## 1  1  1
## 2 NA  2
## 3  3 NA
## 4  4 NA
```

Here, we see that the constructed `set_NA_val` macro modifies a data frame in the calling scope. It saves some boilerplate code from being written, but at the cost of keeping parameter values immutable. The more traditional function solution where you return updated values is probably much more readable to most R programmers.

```
set_NA_val_fun <- function(df, var, na_val) {
  df[df[,var] == na_val, var] <- NA
  df
}
(d <- data.frame(x = c(1,-9,3,4), y = c(1,2,-9,-9)))
```

```
##    x  y
## 1  1  1
## 2 -9  2
## 3  3 -9
## 4  4 -9
```

```
(d <- set_NA_val_fun(d, "x", -9))
```

```
##    x  y
## 1  1  1
## 2 NA  2
## 3  3 -9
## 4  4 -9
```

```
(d <- set_NA_val_fun(d, "y", -9))
```

```
##    x  y
## 1  1  1
## 2 NA  2
## 3  3 NA
## 4  4 NA
```

Or even the `magrittr` pipeline version:

```
library(magrittr)
d <- data.frame(x = c(1,-9,3,4), y = c(1,2,-9,-9))
d %<>% set_NA_val_fun("x", -9) %>% set_NA_val_fun("y", -9)
d
```

```
##    x  y
## 1  1  1
## 2 NA  2
## 3  3 NA
## 4  4 NA
```

Modifying environments in evaluations

The reason we can modify variables in macros is that we can modify environments. Actual values are immutable, but when we modify the data frame in the example above, we are replacing the reference in the environment to the modified data. If other variables refer to the same data frame, they will still be referring to the original version. Even with macros, we cannot *actually* modify data, but we can modify environments, and because we can evaluate expressions in environments different from the current evaluating environment, we can make it appear as if we are modifying data in the calling environment.

We can see this by examining the evaluation environment explicitly when we evaluate an assignment. If we explicitly make an environment and evaluate an assignment in it, we see that it gets modified.

```
e <- list2env(list(x = 2, y = 3))
eval(quote(z <- x + y), e)
as.list(e)
```

```
## $z
## [1] 5
##
## $y
## [1] 3
##
## $x
## [1] 2
```

This environment has the global environment as its parent. Don't try this with the empty environment as its parent. If you do, it won't know the `<-` function. What we see here, is that we modify `e` by setting the variable `z`.

We can also evaluate expressions in lists, so we can attempt the same here:

```
l <- list(x = 2, y = 3)
eval(quote(z <- x + y), l)
l
```

```
## $x
## [1] 2
##
## $y
## [1] 3
```

Here we see that the list is *not* modified. It is only environments we can modify in lists, and while we can evaluate an assignment in a list using `eval`, we cannot actually modify the list.

Accessing promises using the pryr package

As I mentioned before, there is no mechanism in pure R to get access to the internals of promises, so if you use `substitute` to translate a function argument into its corresponding expression then you loose information about which environment the expression should be evaluated in. You can, however, use the `pryr` package to examine promises. This package has the function `promise_info` that tells you both what the expression is and the environment it belongs to.

Consider this:

```
library(pryr)
f <- function(x, y) function(z = x + y) promise_info(z)
g <- f(2, 3)
g()

## $code
## x + y
##
## $env
## <environment: 0x7ff917c4a308>
##
## $evaled
## [1] FALSE
##
## $value
## NULL

x <- 4; y <- 5
g(x + y)

## $code
## x + y
##
## $env
## <environment: R_GlobalEnv>
##
## $evaled
## [1] FALSE
##
## $value
## NULL
```

For a promise we get the expression it corresponds to in the `code` field, the environment it belongs to in the `env` field, whether it has been evaluated yet in the `evaled` field, and if it has been evaluated the corresponding value is in the `value` field. In the two different calls to `g`, we see that the `code` is the same, but the environment is different. In the first call, where we use the default values for parameter `z`, the environment is the `f` closure, and in the second, where we call `g` with an expression from the global environment, the promise environment is also the global environment.

We can see the difference between when a promise has been evaluated and before it is in the following example:

```
g <- function(x) {
  cat("=== Before evaluation =====\n")
  print(promise_info(x))
  force(x)
  cat("=== After evaluation ======\n")
  promise_info(x)
}
g(x + y)
```

```
## === Before evaluation =====
## $code
## x + y
##
## $env
## <environment: R_GlobalEnv>
##
## $evaled
## [1] FALSE
##
## $value
## NULL
##
## === After evaluation ======
## $code
## x + y
##
## $env
## NULL
##
## $evaled
## [1] TRUE
##
## $value
## [1] 9
```

The code doesn't change when we evaluate a promise, but the environment is removed—we do not need to hold a reference to an environment we no longer need, and if we are the only one holding on to this environment we can free it for garbage collection by no longer holding on to it when we don't need it any longer—and the result of evaluating the promise is put in `value` and `evaled` is set to `TRUE`.

We can use the promise info to modify an environment and still evaluate it in the right scope. We just need to get hold of the code, which we can get either by the expression

```
eval(substitute(
  substitute(code, list(y = quote(2 * y))),
  list(code = pi$code)))
```

or the expression

```
eval(substitute(
  substitute(expr, list(y = quote(2 * y)))))
```

where `expr` is the parameter that holds the promise, and `pi` is the result of calling `promise_info(expr)`. Neither is particularly pretty, and you have to remember to construct the expressions inside out, but that is the way you can get an expression substituted in for a variable that holds it and then modify it. Of the two, the traditional approach—the second of the two—is probably the simplest.

In both cases, we are creating the expression

```
substitute(<expr>, list(y = quote(2 * y)))
```

where <expr> refers to the expression in the promise, and we then evaluate this expression, to substitute `y` for `quote(2 * y)`.

Evaluating the expression once we have modified it is almost trivial in comparison. We can just use `eval(expr, pi$env)`.

```
f <- function(x, y) function(expr = x + y) {
  pi <- promise_info(expr)
  expr <- eval(substitute(
    substitute(expr, list(y = quote(2 * y)))))
  value <- eval(expr, pi$env)
  list(expr = expr, value = value)
}
g <- f(2, 2)
g()
```

```
## $expr
## x + 2 * y
##
## $value
## [1] 6
```

```
x <- y <- 4
g(x + y)
```

```
## $expr
## x + 2 * y
##
## $value
## [1] 12
```

```
z <- 4
g(z)
```

```
## $expr
## z
```

```
##
## $value
## [1] 4
```

In the substitution when we create `expr` it is important that we replace `y` with `quote(2 * y)` and not simply `2 * y`. If we did the latter, then `y` would be evaluated in the standard way and would refer to the `y` in the enclosing scope, the parameter given to the call to `f` that creates `g`. Of course, that could be what we wanted: substitute whatever `y` is in the input expression with the `y` we have in the enclosing scope. In that case, the code would simply look like this:

```
f <- function(x, y) function(expr = x + y) {
  pi <- promise_info(expr)
  expr <- eval(substitute(
    substitute(expr, list(y = 2 * y))))
  value <- eval(expr, pi$env)
  list(expr = expr, value = value)
}
g <- f(2, 2)
g()

## $expr
## x + 4
##
## $value
## [1] 6

x <- y <- 4
g(x + y)

## $expr
## x + 4
##
## $value
## [1] 8

z <- 4
g(z)
```

```
## $expr
## z
##
## $value
## [1] 4
```

In both cases, we modify the expression in the promise—just in two different ways—and then we evaluate it in the promise scope. In the first case, `y` is taken from the promise scope if it appears in the modified expression; in the second case, `y` is replaced by the value we have in the enclosing scope.

Conclusions

You have now seen the various techniques used for manipulating the actual language constructs of R from within R programs. Manipulating the actual language, doing meta-programming, gives you the tools to extend R in various ways. You can write functions for modifying other functions—as we did with the code for computing the derivative of a function—or you can write small embedded domain-specific languages for manipulating or querying data frames, as done in `dplyr` and `ggplot2`. You know how to do non-standard evaluation, changing how expressions are evaluated so you can evaluate them in different scopes than what they would usually be evaluated in, which can often simplify how functions are used in pipelines; but you should be careful with using such programming when writing programs since it can be hard to reason about expressions passed between functions.

You don't want to overdo meta-programming. Someone reading your code will, presumably, know how to read R code, so you shouldn't break his expectations for how the code will be evaluated. If you develop a domain-specific language and use meta-programming for this, you are generally fine; someone who can read the expressions you implement there will know how they behave, but use meta-programming with care. It is a very powerful tool, but also a very big gun to shoot yourself in the foot with if you are not careful.

This is the end of the book. I hope it has been useful in learning object-oriented programming as understood by the R programming language. If you liked this book, why not check out my list of other books[1] or sign up to my mailing list[2]?

[1] http://wp.me/P9B2l-DN
[2] http://eepurl.com/cwIbR5