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, and some familiarity with classes and generic 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 subscripting 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): objekt 'x' blev ikke fundet
```

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 use 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 parsed 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`. Basic 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 in order 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 look 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 create, 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 know, 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 actually need to simulate the global environment for this. We need the environment where the `f` function was defined, we call it `defend`, then we need the evaluating environment for the call to `f`, and we need the environment in which `f` is called.

```r
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`:

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

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

```r
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.

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

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

```r
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: 0x7fdbf25e7e08>
```

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 actually provide `y` as a parameter when calling `f` things change. Now it is 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: 0x7fdbf25e7e08>
```

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: 0x7fdbf25e7e08>
```

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`:

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

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

```r
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:

```r
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:

```r
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 really used for, but if you start hacking around with modifying parameters of a function, it is the one to use.

```r
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: 0x7fdbf3079c38>
```

```
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: 0x7fdbf2be52b0>
```

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

```
## function (x, y = z)
## x + y
## <environment: 0x7fdbf2c55578>
```

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 lazy 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`, really, 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 really 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 return 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 actually 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, so 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 parsed 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 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 translate 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 the expressions 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
```

```r
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.

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

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

```r
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.

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

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

```r
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 substitute 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 actually 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 actually 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 parsed 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 actually 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 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 return the value of 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 complex 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 basically 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 a 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 own infix function `` `%<-%` `` and get

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

Maybe not the prettiest syntax, but good enough for an example. But we can get even more ambitions 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 really 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`, but 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 refer 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:

```r
.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 really 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 to evaluate into this function. 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.