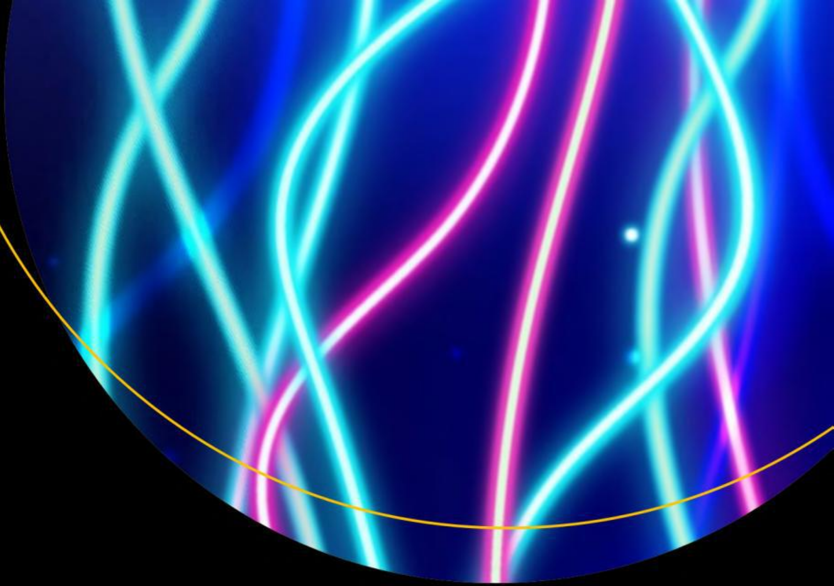

Metaprogramming in R

Advanced Statistical Programming for Data Science, Analysis and Finance

—

Thomas Mailund

Apress®

Metaprogramming in R

Advanced Statistical Programming for Data Science, Analysis and Finance

Thomas Mailund

Apress®

Metaprogramming in R: Advanced Statistical Programming for Data Science, Analysis and Finance

Thomas Mailund
Aarhus N, Denmark

ISBN-13 (pbk): 978-1-4842-2880-7 ISBN-13 (electronic): 978-1-4842-2881-4
DOI 10.1007/978-1-4842-2881-4

Library of Congress Control Number: 2017943347

Cover image designed by Freepik

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Technical Reviewer: Massimo Nardone
Coordinating Editor: Mark Powers
Copy Editor: Kim Wimpsett

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484228807. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

Contents at a Glance

Contents

About the Author

Thomas Mailund is an associate professor in bioinformatics at Aarhus University, Denmark. His background is in math and computer science, but for the last decade his main focus has been on genetics and evolutionary studies, particularly comparative genomics, speciation, and gene flow between emerging species.

About the Technical Reviewer

Massimo Nardone has more than 22 years of experience in security, web/mobile development, the cloud, and IT architecture. His true IT passions are security and Android.

He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a master of science degree in computing science from the University of Salerno, Italy.

He has worked as a project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect for many years.

He currently works as a chief information security officer (CISO) for Cargotec Oyj.

He was a visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University), and he holds four international patents (PKI, SIP, SAML, and proxy areas).

Massimo has reviewed more than 40 IT books for different publishing companies, and he is the coauthor of *Pro Android Games* (Apress, 2015).

Introduction

Welcome to *Metaprogramming in R*. I am writing this book, and my books on R programming in general, to help make more advanced teaching material available beyond the typical introductory level most textbooks on R have. This book covers some of the more advanced techniques used in R programming such as fully exploiting functional programming, writing metaprograms (code for actually manipulating the language structures), and writing domain-specific languages to embed in R.

This book introduces metaprogramming. *Metaprogramming* is when you write programs that manipulate other programs; in other words, you treat code as data that you can generate, analyze, or modify. R is a very high-level language where all operations are functions, and all functions are data that you can manipulate.

There is great flexibility in how function calls and expressions are evaluated. The lazy evaluation semantics of R mean 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 metaprogramming 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 nonstandard 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.

CHAPTER 1

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 there are ways of probing and modifying them.

Manipulating Functions

If you define a simple function like the following, you can examine the components it consists of:

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

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` give you 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.

© Thomas Mailund 2017
T. Mailund, *Metaprogramming in R*, DOI 10.1007/978-1-4842-2881-4_1

```
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` function (that function is useful only 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 whether 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 that are defined in R.

```
formals(`+`)
## NULL
```

Function Bodies

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

```
body(f)
## x
```

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

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

It fails because you do not have a variable `x` defined anywhere. If you 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. If we call the function, the expression will know about the function context, of course, but not when we simply evaluate the function body like this. You 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. Chapter 3 covers how to work with expressions and how to evaluate them; for now all you have to know is that you can evaluate an expression using eval if the variables in the expression are either found in the scope where you call eval or provided in the second argument to eval.

You can also set x as a default parameter and use that when you 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 way the evaluation environment is set up and not so much with how expressions are evaluated, but consider the following example where one default parameter refers to another:

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

Both parameters have default values, so you can call f without any arguments.

```
f()
## [1] 10
```

You 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 you simply look up x, you just get the symbol y, and you don't evaluate it further to figure out what y is. Therefore, you get an error.

Formal arguments are not evaluated this way when you call a function. They are transformed into so-called promises, which are unevaluated expressions with an associated scope. This is how the formal language definition 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.

This means that in the evaluating environment, R first assigns all variables to these "promises." The promises are placeholders for values but represented as expressions you 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, you don't have to worry about the order in which you assign the variables. As long as the variables exist when you evaluate a promise, you are fine, and as long as there are no circular dependencies between the expressions, you can figure out all the values when you need them.

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

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

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

```
fenv <- new.env()
parameters <- formals(f)
for (param in names(parameters)) {
  delayedAssign(param, parameters[[param]], fenv, fenv)
}
eval(body(f), fenv)
## [1] 10
```

Here you 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 you 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 you 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 continuing, 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 understand how expressions are evaluated. When you start to fiddle around with nonstandard 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 whether 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 sees that it isn't there, at which point it will raise an error. You call the variables that 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 innermost environment, the first you find, will be used.

When a function, `f`, is created, it gets associated with `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 you saw from the language definition earlier, there is a difference between default parameters 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 where you call f; they would be able to refer only to local variables or variables in the scope where f was defined.

You can see it all in action in the following example:

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

f <- enclosing()

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

calling()
## [1] 22
```

You start out in the global environment where you define enclosing to be a function. When you call enclosing, you create an evaluation environment in which you store the variable z and then return a function that you 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 you create calling, store that in the global environment, and call it. This creates, once again, an evaluation environment. In this, you store the variable w and then call f. You don't have f in the evaluation environment, but because the parent of the evaluation environment is the global environment, you can find it. When you call f, you give it the expression 2 * w as parameter x.

Inside the call to f, you have another evaluation environment. Its parent is the closure you got from enclosing. Here you need to evaluate f's body: x + y + z. However, 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. You provided x as a parameter when you 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. You evaluate x, which is the expression 2 * w in the calling environment, where w is known, and you evaluate y in the local environment, where x is known. So, you can get the value of those two variables and then get z from the enclosing environment.

You can try to emulate all this using explicit environments and delayedAssign to store promises. You need three environments since you don't need to simulate the global environment for this. You need the environment where the f function was defined; you call it defenv. Then you need the evaluating environment for the call to f, and you 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 you don't need to worry about that. The evaluating environment has defend as its parent.

In the definition environment, you save the value of z.

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

In the calling environment, you save the value of w.

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

In the evaluation environment, you 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 is where it should be stored. For x you want the expression to be evaluated in the calling environment, and for y you 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 you 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

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

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

you 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, you 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 you 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 you define a closure, you 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 you call it, it will, therefore, be able to see the local variable y from nested. It doesn't refer to that, but you can change this by modifying its formals.

```
formals(f) <- list(x = quote(y))
f
## function (x = y)
## x
## <environment: 0x7fc0f8c85908>
```

Here, you have to use the function quote to make y a name. If you didn't, you would get an error, or you 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 you modify formals, you have to do this explicitly.

If you 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 you modified formals(f) in the global environment, you do not change in which environment R evaluates promises for default parameters. If you 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 you 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, you get a reference to the global y.

```
f(x = y)
## [1] 2
```

You can modify the body of f as well. Instead of having its body refer to x, you can, for example, make it return the constant 6.

```
body(f) <- 6
f
## function (x = y)
## 6
## <environment: 0x7fc0f8c85908>
```

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

```
f()
## [1] 6
f(x = 12)
## [1] 6
```

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

```
body(f) <- 2 * y
f()
## [1] 4
```

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

```
y <- 3
f()
## [1] 4
```

To get an unevaluated body, you must, again, use quote.

```
body(f) <- quote(2 * y)
f
## function (x = y)
## 2 * y
## <environment: 0x7fc0f8c85908>
```

Now, however, you get back to the semantics for function calls, which means that the body is evaluated in an evaluation environment whose 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
```