Stat 8054 Lecture Notes: R as a Functional Programming Language

Charles J. Geyer

December 22, 2023

1 License

2 R

- The version of R used to make this document is 4.3.2.
- The version of the `rmarkdown` package used to make this document is 2.25.
- The version of the `magrittr` package used to make this document is 2.0.3.
- The version of the `CatDataAnalysis` package used to make this document is 0.1.5.
- The version of the `glmbb` package used to make this document is 0.5.1.

3 Reading

- A blog post at R Bloggers: Functional programming in R.
- Three chapters in *Advanced R:*
  - Functional programming
  - Functionals
  - Function Operators
- A question in the R FAQ: What are the differences between R and S?.
- Some sections of my 3701 handout on the basics of R:
  - 4 Functions
  - 6 More on Functions
  - 7 Still More on Functions
  - 7.5 A Long Example (Maximum Likelihood Estimation), especially subsections 7.5.4 and 7.5.5 and 7.5.6
- Some sections of the book *The R Language Definition*, which is one of the R manuals that can be found at CRAN and also in your own R installation (R function `help.start()` provides access to local versions of them):
  - Section 2 Objects, which is very long, covering all the kinds of objects that R has, and is just background, not assigned reading

– Section 4.3.3 Argument evaluation

4 Some Keywords

4.1 Functional Programming

Functional programming has been defined many different ways (Wikipedia page), but at the very least definitions require that functions are first-class objects. You can do with a function whatever you can do with any other object. R has this.

4.2 Closures

> This [closures] is the best idea in the history of programming languages.
>
> — Douglas Crockford

Crockford was talking about Javascript, but R has the same idea. Both inherited it from Scheme (a flavor of LISP). It is the function named `function`, which creates functions. Technically, they are called "closures"

```
typeof(function(x) x)
```

```
## [1] "closure"
```

because they capture variables in the environment where they are defined that are used in the function definition. If that environment is the R global environment, then this works as users (some of them anyway) expect

```
x <- 2.5
fred <- function(y) x + y
environment(fred)
```

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

```
fred(pi)
```

```
## [1] 5.641593
```

```
x <- 0
fred(pi)
```

```
## [1] 3.141593
```

```
rm(x)
fred(pi)
```

```
## Error in fred(pi): object 'x' not found
```

If that environment is another environment, then this works as users (most of them anyway) do not expect

```
fred_factory <- function(x) function(y) x + y
fred <- fred_factory(2.5)
environment(fred)
```

```
## <environment: 0x560b235f86f0>
```

```
fred(pi)
```

```
## [1] 5.641593
```

```
x <- 0
fred(pi)
```

```
## [1] 5.641593
```

```
ls(envir = environment(fred))
```

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

```
environment(fred)$x
```

```
## [1] 2.5
```

```
environment(fred)$x <- 0
fred(pi)
```

```
## [1] 3.141593
```

```
rm(x, envir = environment(fred))
ls(envir = environment(fred))
```

```
## character(0)
```

```
fred(pi)
```

```
## [1] 3.141593
```

```
rm(x)
fred(pi)
```

```
## Error in fred(pi): object 'x' not found
```

Nevertheless, closures (or functions as most R users call them) are super important. They enable many good programming practices.

4.3 Pure Functional Programming

A functional programming language is "pure" if all functions act like math functions: for the same inputs (arguments) they always produce the same output (value). They cannot do assignment (except to local variables inside the function that are not visible to callers of the function), input, or output. They cannot use random numbers.

Some would say that a functional programming language is "pure" only if the whole language behaves that way. Pure functions are the only feature of the language.

R is not a pure functional programming language. In R functions do everything. Even assignment is really a function

```
x <- 2
x
```

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

```
assign("x", 3)
x
```

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

```
`<-`
```

```
## .Primitive("<-")
```

```
`<-`(x, 4)
x
```

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

The last bit you do not ever want to use in real life. No reader of your code would understand it. But it does show that everything that happens in R happens via a function call.

So the assign function named "<-" is not pure. Its whole reason for existence is to have a side effect: assignment. Similarly all input, output, and graphics functions are not pure. Similarly, all random number functions are not pure. Similarly, R functions `date` and `Sys.getenv` and `file.create` and `list.files` and many other functions that allow R to act as a scripting language are not pure.

Nevertheless, most R functions used for computation are pure. The only thing most R functions do that is observable from the outside is return a value. They do not refer to global variables. If they are using variables defined in their environment, then they are perhaps not strictly pure (strictly speaking), but so long as users do not modify that environment — and if it is not the R global environment, then most users will not even know that environment exists — such a function is practically pure.

4.4 Anonymous Functions

A function does not need a name. Like every function, the R function named "function" returns a value. In this case the value happens to be a function. Other functions can also return functions as values. Usually, these functions are created by a call to the R function named "function" inside the function returning a function as a value (as we have already seen in some of the examples above).

Like every other R object, the value returned by the R function named "function" does not get associated with a name (symbol) except by assignment.

```
(function() 2)()
```

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

We don't want to do trickery like the above in real life. No readers of our code would understand. But anonymous functions are quite useful as arguments to other functions, as examples below will show.

4.5 Immutability

Until recently, you never heard programmers talking about immutability. Most programming languages don't have it. Some languages like Javascript have recently gotten it as a feature but not as the default. In a few languages like Haskell and Clojure, it is the default.

In R immutability has always been the default. Most R objects cannot be changed once created. Only environments and reference classes, which most R users do not understand and do not explicitly use, are mutable. All assignments mutate an environment. R function `rm` also mutates environments. So users mutate environments all the time, but don't think about environments while doing so.

The less said about S4 classes in general and reference classes in particular, the better (they are in the process of being replaced, anyway).

So R environments are mutable, but most R objects are not.

```
x <- y <- 1:5
x[3] <- NA
x
```

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

```
y
```

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

After the first assignment, "x" and "y" are *names* for the *same object*.

The second assignment does not mutate an object, it creates a *new* object and assigns it to the name "x".

In Java or C++ the analogous code would mutate the object named both "x" and "y". Both "x" and "y" would name the same object before and after it was mutated. Their objects are mutable. R objects (except environments and reference classes) are immutable.

This is something most programmers who have been introduced to programming via Java or C++ think is bad about R. They think it is inefficient.

But it is actually something that is very good about R. It is what makes R a language that naive users can use. Even expert Java and C++ programmers make lots of mistakes that naive R programmers cannot make because of the nature of R.

4.6 Recursion

In R, like in most programming languages (even Java and C++), functions can call themselves. This is called recursion. It is a valuable programming technique that can simplify many problems. It is not widely used in R, but R does have it. R functions can call themselves.

This is not even a special feature of R. Function bodies can have any valid R code that can appear anywhere in R. Function calls are part of such. Hence functions can call other functions. If the function being called from inside another function happens to be the same function, then that is what other computer languages fuss about and call "recursion". In R, recursive function calls are no different from any other function calls.

4.7 Lazy Evaluation (Promises)

Like some other functional programming languages (Haskell), R has lazy evaluation. Function arguments are not evaluated until needed. If not needed, they are not evaluated at all. Section 4.3.3 of the *R Language Definition* explains.

Mostly R users and programmers do not need to pay attention to lazy evaluation. But occasionally they do. Section 4.3.3 of the *R Language Definition* gives an example where it is necessary to force evaluation to get what one wants.

There is an R function named "force" that has no effect except forcing evaluation that is useful for making clear the programmers intent. I would change the example in the section cited in the preceding paragraph so that the statement

```
label
```

is changed to

```
force(label)
```

Both have the same effect, but the latter makes the intention clearer.

Understanding lazy evaluation is the key to understanding how default arguments to functions work in tricky situations. That is illustrated by the example cited in the preceding paragraph. It is also the point of the discussion of default arguments of R function `svd` which are discussed in Section 7.2 of my 3701 handout on the basics of R.

4.8 Higher-Order Functions

Any function that takes a function as an argument or returns a function as a value is called a *higher-order function*. This terminology is not widely used in R. Most R functions are not higher-order functions. Most higher-order functions in R are not called higher-order functions by users.

4.8.1 R Functions that work on mathematical functions.

- R functions `D` and `deriv` differentiate R expressions and return R expressions. Since R expressions are not R functions, these are not higher-order functions, strictly speaking, except that `deriv` can be made to return a function rather than an expression by use of an optional argument.

- R functions `grad` and `hessian` and `jacobian` in R package `numDeriv` differentiate R functions (approximately) and return the values of the derivatives evaluated at specified points. These are vector or matrix valued derivatives of vector-to-scalar or vector-to-vector functions.
- R function `integrate` does numerical integration of a mathematical function provided to it as an R function.
- R functions `nlm` and `optim` and `optimize` and many other R functions that do optimization, optimize a mathematical function provided to them as R functions.
- R function `uniroot` finds a root of a univariate mathematical function provided to it as an R function.
- R function `boot` in R recommended package `boot` (installed by default in every installation of R) approximately simulates the sampling distribution of any estimator provided to it as an R function applied to any data using Efron's nonparametric bootstrap.
- R function `metrop` in CRAN package mcmc simulates the distribution of any continuous random vector whose log unnormalized probability density function is provided to it as an R function.
- R function `glm` that fits generalized linear models takes an argument named `family` that is a function that produces a family object, but this argument can also be a family object or the name of a family function, so this is not exactly a higher-order function but more R-ish. That is, `glm(y ~ x, family = binomial)` works and here argument `binomial` is a function, so this is a higher-order function call. But `glm(y ~ x, family = "binomial")` and `glm(y ~ x, family = binomial())` also work, and those are not higher-order function calls because argument `family` is not, strictly speaking, a function.

So this is a very widely used R design pattern. To tell an R function about a mathematical function, provide it an R function that evaluates that mathematical function (and that function should be pure).

4.8.2 R Functions of the Apply Family

R function `apply` and friends (`eapply`, `lapply`, `mapply`, `rapply`, `sapply`, `tapply`, and `vapply` in R package `base` and `mclapply` and `clusterApply` in R package `parallel`) apply an arbitrary function (provided as an R function) to each element of some compound object (vector, list, data frame, table, etc.) or on each node of a cluster in parallel programming.

These also are not usually called higher-order functions by R users (although, strictly speaking, they are higher-order functions).

4.8.3 R Functions of the Higher-Order Family

In languages much newer than R a bunch of terminology for higher-order functions has grown up, and R has copied it. See `help("funprog")` for the list. The most widely used are `Map`, `Reduce`, and `Filter`. Some of these are built on functions of the apply family.

4.8.4 Some Odds and Ends

Some other R functions take functions as arguments like `sweep`, `aggregate`, `outer`, and `kronecker`.

4.8.5 Functionals

This is Hadley Wickham's name for functions that take a function argument and perhaps some other arguments and return a vector. It includes many of the higher-order functions discussed above.

4.8.6 Function Operators

This is Hadley Wickham's name for functions that take a function argument and return a function value. That AFAICS doesn't include any widely used R higher-order functions.

Oops! R function `Vectorize` is an example of a function operator.

4.8.7 Function Factories

This is Hadley Wickham's name for functions whose only job is to create other functions, like R function `fred_factory` in an example above.

In Section 7.5.4 of my 3701 handout on the basics of R the R function `make.logl` defined in its example is a function factory (which makes log likelihood functions for gamma shape families), although I didn't call it that when I wrote those notes.

5 Functional Programming versus Imperative Programming

R has the stuff of imperative programming languages, such as loops and assignment. So it isn't *just* a functional programming language.

The way users trained in Java or C++ and unfamiliar with the way of R program R, it looks just like C or Java or C++. But expert R programmers familiar with the way of R use functions instead of loops. They even use much less assignment.

I used to think I used far fewer loops than most R programmers, and I was right. But since writing the 3701 notes cited above, I use even fewer loops and even more functions. I am trying to exemplify the way of R.

6 More on Functions, Closures, Promises, Names, Objects, and Environments

6.1 Objects

> To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call.
>
> — John Chambers, quoted in Section 6.3 of *Advanced R* by Hadley Wickham

Unlike in many so-called object-oriented programming languages where not *everything* is an object, in R *everything* is an object, even expressions of the language itself.

Section 2 Objects of *The R language Definition* covers all the kinds of R objects, but that is more than we want to know about here.

6.2 Names

In R, objects can have *names*, also called *symbols*,

```
an <- as.name("arrg")
is.name(an)
```

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

```
mode(an)
```

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

```
typeof(an)
```

```
## [1] "symbol"
```

```
storage.mode(an)
```

```
## [1] "symbol"
```

```
class(an)
```

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

These correspond to what many users call "variables". Where many users say the R statement

```
x <- 2
```

creates a variable `x` which is assigned the value 2, the pedantically correct thing to say is that the R *name* or *symbol* `x` has been associated with the R object which is the numeric vector having one component 2.

A quotation from the R help for R function `make.names`

> A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. Names such as ' ".2way" ' are not valid, and neither are the reserved words.
>
> The definition of a *letter* depends on the current locale, but only ASCII digits are considered to be digits.

To understand this we have to know what "reserved words" are, and the R help page titled "Reserved" shows them (do `?Reserved` or `help("Reserved")` to see that page).

For example, `function` is a reserved word, so it cannot be a valid R symbol. So when we say "the R function named function" that is pedantically incorrect (more on this below).

Anything can be a symbol if it is put in backquotes. So

```
get("function")
```

```
## .Primitive("function")
```

```
`function`
```

```
## .Primitive("function")
```

But just saying `function` without the quotes is an incomplete R statement (R requires a function definition to follow the reserved word `function`).

Similarly for other bits of R syntax

```
`+`
```

```
## function (e1, e2)  .Primitive("+")
```

```
`[<-`
```

```
## .Primitive("[<-")
```

Sometimes names can be provided as character strings as arguments to other functions (like we saw in `get("function")` above) but this is just what certain particular functions require. That function undertakes to turn character strings into symbols or their corresponding objects. There is no automatic conversion of character strings to symbols.

```
outer(1:3, 4:5)
```

```
##      [,1] [,2]
## [1,]    4    5
## [2,]    8   10
## [3,]   12   15
```

```
outer(1:3, 4:5, "+")
```

```
##      [,1] [,2]
## [1,]    5    6
```

```
## [2,]    6    7
## [3,]    7    8
```

6.3 The Other Kind of Names

Names, also called symbols, are not to be confused with names, *not* also called symbols. Any R object can have an attribute named "names" which is helpful in indexing

```
x <- seq(along = letters)
names(x) <- letters
x["c"]
```

```
## c
## 3
```

```
y <- as.list(x)
y[["c"]]
```

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

```
y$c
```

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

There is a weak connection between names of the first kind and names of the second kind in that in order for `y$c` to work it is necessary that `c` be a valid symbol name (`y$function` is invalid, even if `"function"` is the name of some component of `y`).

6.4 Environments

Environment is a type of R object

```
typeof(globalenv())
```

```
## [1] "environment"
```

A quotation from Section 2.1.10 Environments of *The R Language Definition*

> Environments can be thought of as consisting of two things. A *frame*, consisting of a set of symbol-value pairs, and an *enclosure*, a pointer to an enclosing environment. When R looks up the value for a symbol the frame is examined and if a matching symbol is found its value will be returned. If not, the enclosing environment is then accessed and the process repeated. Environments form a tree structure in which the enclosures play the role of parents. The tree of environments is rooted in an empty environment, available through `emptyenv()`, which has no parent.

So environments are where objects can be looked up by name, and the lookup is not just in the specified environment, but in the parent, parent of parent, parent of parent of parent, and so forth, of the specified environment.

R environments are very different from all other kinds of R objects. When an environment is changed (by assignment or by R function `rm`) the object is modified. Unlike all other kinds of R objects, environments are *mutable* (being mutable is their whole *raison d'être*). Compare the example below with the one in the section on immutability above.

```
foo <- bar <- new.env(parent = emptyenv())
foo$x <- 1:5
bar$x
```

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

A very special environment is the R global environment, which is returned by R function `globalenv` or by the R global variable (name, symbol) `.GlobalEnv`. This is where assignments done at top level (on the R command line) are done

```
x <- "This is the assignment we just made"
x
```

```
## [1] "This is the assignment we just made"
```

```
globalenv()$x
```

```
## [1] "This is the assignment we just made"
```

```
.GlobalEnv$x
```

```
## [1] "This is the assignment we just made"
```

The parent, parent of parent, and so forth, of the R global environment depends on what packages have been loaded

```
search()
```

```
##  [1] ".GlobalEnv"        "package:rmarkdown" "package:stats"
##  [4] "package:graphics"  "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:methods"   "Autoloads"
## [10] "package:base"
```

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

```
##  [1] ".GlobalEnv"        "package:MASS"      "package:rmarkdown"
##  [4] "package:stats"     "package:graphics"  "package:grDevices"
##  [7] "package:utils"     "package:datasets"  "package:methods"
## [10] "Autoloads"         "package:base"
```

6.5 Function Execution Environment

When a function is called (one also says *invoked*), local variables go in the function *evaluation environment*, whose parent is the function environment. By local variables we mean those that are defined in the function body and created by execution of statements in the function body.

6.6 Promises

A promise is another kind of R object, but not one that can be accessed by users. They can be *used* by users and in fact are used whenever one invokes a function that has one or more arguments.

A quotation from Section 4.3.3 Argument Evaluation of *The R Language Definition*

> One of the most important things to know about the evaluation of arguments to a function is that supplied arguments and default arguments are treated differently. The supplied arguments to a function are evaluated in the evaluation frame of the calling function. The default arguments to a function are evaluated in the evaluation frame of the function.

It is this latter fact that makes some tricky definitions of default arguments work, like those of R function `svd` as discussed in Section 7.2 of my 3701 handout on the basics of R which was already referred to above in the previous section on promises.

6.7 Pipes

To repeat part of the quotation from John Chambers above, "everything that happens is a function call", but that depends on what "everything" means. There is also parsing. The R parser (the part of R that figures

out what to do from the code you write) does some of the work. It is true that

```
2 + 2
```

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

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

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

are equivalent R statements. They are equivalent because the R parser turns the first into the second. The code `2 + 2` means call R function `` `+` `` with arguments 2 and 2.

Similarly, `+ 2` (that is unary plus) means call R function `` `+` `` with one argument.

```
+ 2
```

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

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

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

But R function `` `+` `` is not R function `sum`.

```
`+`(1, 2, 3)
```

```
## Error in `+`(1, 2, 3): operator needs one or two arguments
```

```
sum(1, 2, 3)
```

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

The only purpose of R function `` `+` `` is to be the function that backs up the plus operation in code.

Similar syntax transformations are involved in turning all R code into R function calls.

All of this is preliminary to discussing a new bit of R syntax transformation that is introduced in R 4.1.0. This is the pipe operation, which is denoted `|>`. This is part of base R that replaces the pipe operation from R package `magrittr`, which appeared on CRAN in 2014 and since then has become very popular. As I write this, https://cran.r-project.org/package=magrittr lists 2525 R packages that need R package `magrittr`

- 61 reverse depends (52 CRAN, 9 Bioconductor)
- 2147 reverse imports (1941 CRAN, 206, Bioconductor)
- 317 reverse suggests (281 CRAN, 36 Bioconductor)

The new base R pipe operation replaces the main functionality of R package `magrittr` (but not all of the functionality). But it does it in a completely different way. The `magrittr` pipe operator is an R function `%>%` that uses nonstandard evaluation to do what it does (pipe the output of the function call on the left-hand side into the function call on the right-hand side). The new R pipe operation is not implemented as an R function but rather as a syntax transformation, as the help page for this new operation (`help("|>")`) shows in some of its examples

```
# the pipe operator is implemented as a syntax transformation:
quote(mtcars |> subset(cyl == 4) |> nrow())
```

```
## nrow(subset(mtcars, cyl == 4))
```

```
quote(x |> (function(x) x + 1)())
```

```
## (function(x) x + 1)(x)
```

Or more simply

```
quote(x |> f(y))
```

```
## f(x, y)
```

The R code `x |> f(y)` is literally turned into the code `f(x, y)` before anything else happens. The right-hand side of of a pipe expression must be a function call. It is an error for it to be a function *definition* or a function *name*.

```
mtcars |> class
```

```
## Error: The pipe operator requires a function call as RHS (<text>:1:11)
```

The way to do this is to turn the right-hand side into a function *call* rather than a function *name*.

```
mtcars |> class()
```

```
## [1] "data.frame"
```

Note that this applies to anonymous functions too. You cannot put an anonymous function in a pipeline, only the *evaluation* of an anonymous function

```
"Foo!" |> function(x) x
```

```
## Error: function 'function' not supported in RHS call of a pipe (<text>:1:11)
```

```
"Foo!" |> (function(x) x)()
```

```
## [1] "Foo!"
```

New in R-4.2.0 is the "placeholder" option in the right-hand-side (RHS) of a pipe expression. The placeholder character is the underscore character, and this can be used exactly once in the RHS as the value of a named argument (the argument must be named) to indicate that the input is piped into this argument rather than the first argument. This is a `magritter` feature that the core R pipe operator did not have before R-4.2.0 and was one of the most frequent reasons for needing an anonymous function, Use of the placeholder is illustrated in the following example.

So why is this popular? R function `%>%` in R package `magrittr` is already very popular. R operation `|>` in base R is much cleaner, much easier to debug, and much more efficient. So it should eventually replace most usages of `%>%`. It allows complicated nested function calls to be read left to right rather than inside out. Here is the first example from the `magrittr` package vignette. Here it is in `magrittr`

```
library(magrittr)
car_data <-
    mtcars %>%
    subset(hp > 100) %>%
    aggregate(. ~ cyl, data = ., FUN = . %>% mean %>% round(2)) %>%
    transform(kpl = mpg %>% multiply_by(0.4251)) %>%
    print
```

```
##   cyl   mpg   disp     hp drat   wt  qsec   vs   am gear carb       kpl
## 1   4 25.90 108.05 111.00 3.94 2.15 17.75 1.00 1.00 4.50 2.00 11.010090
## 2   6 19.74 183.31 122.29 3.59 3.12 17.98 0.57 0.43 3.86 3.43  8.391474
## 3   8 15.10 353.10 209.21 3.23 4.00 16.77 0.00 0.14 3.29 3.50  6.419010
```

And here it is rewritten to use `|>` instead of `%>%`.

```
car_data <-
    mtcars |>
    subset(hp > 100) |>
    aggregate(. ~ cyl, data = _,
        FUN = function(x) x |> mean() |> round(2)) |>
    transform(kpl = mpg |> (function(x) x * 0.4251)()) |>
    print()
```

```
##   cyl   mpg   disp     hp drat   wt  qsec   vs   am gear carb       kpl
## 1   4 25.90 108.05 111.00 3.94 2.15 17.75 1.00 1.00 4.50 2.00 11.010090
## 2   6 19.74 183.31 122.29 3.59 3.12 17.98 0.57 0.43 3.86 3.43  8.391474
## 3   8 15.10 353.10 209.21 3.23 4.00 16.77 0.00 0.14 3.29 3.50  6.419010
```

and this is (supposedly) easier to read (see below) than without pipes

```
print(car_data <-
    transform(aggregate(. ~ cyl,
            data = subset(mtcars, hp > 100),
            FUN = function(x) round(mean(x), 2)),
        kpl = mpg * 0.4251))
```

```
##   cyl   mpg   disp     hp drat   wt  qsec   vs   am gear carb       kpl
## 1   4 25.90 108.05 111.00 3.94 2.15 17.75 1.00 1.00 4.50 2.00 11.010090
## 2   6 19.74 183.31 122.29 3.59 3.12 17.98 0.57 0.43 3.86 3.43  8.391474
## 3   8 15.10 353.10 209.21 3.23 4.00 16.77 0.00 0.14 3.29 3.50  6.419010
```

Our rewritten example is a bit clumsy because the `magrittr` example uses pipes pointlessly in some cases just to be using them. If we get rid of some of the pipes, we can simplify our example.

```
car_data <-
    mtcars |>
    subset(hp > 100) |>
    aggregate(. ~ cyl, data = _,
        FUN = function(x) mean(x) |> round(2)) |>
    transform(kpl = mpg * 0.4251) |>
    print()
```

```
##   cyl   mpg   disp     hp drat   wt  qsec   vs   am gear carb       kpl
## 1   4 25.90 108.05 111.00 3.94 2.15 17.75 1.00 1.00 4.50 2.00 11.010090
## 2   6 19.74 183.31 122.29 3.59 3.12 17.98 0.57 0.43 3.86 3.43  8.391474
## 3   8 15.10 353.10 209.21 3.23 4.00 16.77 0.00 0.14 3.29 3.50  6.419010
```

But we still have an anonymous function definition that `magrittr` does not need (it is hidden inside R function `%>%`). Along with the pipe operation R 4.1.0 introduced a new syntax for anonymous function definitions that replaces `function (` with `\(`. Using this, our example can be shortened to

```
car_data <-
    mtcars |>
    subset(hp > 100) |>
    aggregate(. ~ cyl, data = _,
        FUN = \(x) x |> mean() |> round(2)) |>
    transform(kpl = mpg * 0.4251) |>
    print()
```

```
##   cyl   mpg   disp     hp drat   wt  qsec   vs   am gear carb       kpl
## 1   4 25.90 108.05 111.00 3.94 2.15 17.75 1.00 1.00 4.50 2.00 11.010090
## 2   6 19.74 183.31 122.29 3.59 3.12 17.98 0.57 0.43 3.86 3.43  8.391474
## 3   8 15.10 353.10 209.21 3.23 4.00 16.77 0.00 0.14 3.29 3.50  6.419010
```

Whether you like this stuff or not is a matter of taste. Pipes are function composition

```
quote(x |> f() |> g() |> h())
```

```
## h(g(f(x)))
```

And function composition is the most important method of structuring programs, far more important than object orientation and other such fads. In fact, favor composition over inheritance has become a

slogan of object-oriented design. It says: don't use the object-orientated feature (*inheritance*), instead use composition (which is not particularly object-oriented), because *inheritance*, despite being a key feature of object orientation, has many undesirable properties. (The OOP slogan, refers to *object* composition rather than *function* composition, so perhaps is not exactly on point. But in OOP languages, one may need object composition to do function composition.)

The oldest AFAIK method of structuring programs (at least higher-level language programs as opposed to assembly language programs, which no one writes today) is *assignment*. Consequently, it is the most familiar method, especially to R users. We can structure this example as

```
foo <- subset(mtcars, hp > 100)
bar <- aggregate(. ~ cyl, foo, FUN = function(x) round(mean(x), 2))
car_data <- transform(bar, kpl = mpg * 0.4251)
car_data
```

```
##   cyl   mpg   disp     hp drat   wt  qsec   vs   am gear carb       kpl
## 1   4 25.90 108.05 111.00 3.94 2.15 17.75 1.00 1.00 4.50 2.00 11.010090
## 2   6 19.74 183.31 122.29 3.59 3.12 17.98 0.57 0.43 3.86 3.43  8.391474
## 3   8 15.10 353.10 209.21 3.23 4.00 16.77 0.00 0.14 3.29 3.50  6.419010
```

And we note

- the silliness of having one column rounded differently than the rest and
- the silliness of storing rounded data, where it may lead to inaccuracy if ever used.

Should be

```
foo <- subset(mtcars, hp > 100)
bar <- aggregate(. ~ cyl, foo, mean)
car_data <- transform(bar, kpl = mpg * 0.4251)
round(car_data, 2)
```

```
##   cyl   mpg   disp     hp drat   wt  qsec   vs   am gear carb   kpl
## 1   4 25.90 108.05 111.00 3.94 2.15 17.75 1.00 1.00 4.50 2.00 11.01
## 2   6 19.74 183.31 122.29 3.59 3.12 17.98 0.57 0.43 3.86 3.43  8.39
## 3   8 15.10 353.10 209.21 3.23 4.00 16.77 0.00 0.14 3.29 3.50  6.42
```

So now that we have used assignment to clean up this `magrittr` example we can turn it back into pipes and it stays cleaner.

```
car_data <- mtcars |> subset(hp > 100) |> aggregate(. ~ cyl, data = _, mean) |>
    transform(kpl = mpg * 0.4251)
car_data |> round(2)
```

```
##   cyl   mpg   disp     hp drat   wt  qsec   vs   am gear carb   kpl
## 1   4 25.90 108.05 111.00 3.94 2.15 17.75 1.00 1.00 4.50 2.00 11.01
## 2   6 19.74 183.31 122.29 3.59 3.12 17.98 0.57 0.43 3.86 3.43  8.39
## 3   8 15.10 353.10 209.21 3.23 4.00 16.77 0.00 0.14 3.29 3.50  6.42
```

Like it or not, use it or not, you will be seeing this in other people's code.

A pure functional language does not have assignment. Its only method of program structuring is definition of functions and composition of functions. Pipelines are just composition in disguise. So pipelining is more "functional" than anything else you can do in R (except actual composition: function calls as arguments to function calls).

As we can see from the example (not my example, `magrittr`'s example) pipelines are not really easier to read than code with assignments, especially for naive users. They are more functional. They do avoid the clutter of a lot of unmotivated temporary variables. But whether you use them or not is up to you. It's just a matter of taste.

I myself have not gotten on the pipeline bandwagon (in R, I have been using UNIX pipes for 35 years). This is partly because I do not like to require very recent versions of R for my CRAN packages (some users are slow to update). But also I do not like the ugly syntax required to use anonymous functions in pipes (but a lot of this ugliness is no longer required because of the new placeholder syntax). I expect that I will eventually start using pipes and they will be base R pipes rather than `magrittr` pipes (and this despite the super cool name of that package).

It might be an interesting exercise to point out where in my notes pipelines might make code a lot neater.

One more example, this time from my categorical data analysis notes but redone to use pipes.

```
library(CatDataAnalysis)
library(glmbb)
data(table_8.1)
gout <- table_8.1 |>
    transform(lake = factor(lake,
        labels = c("Hancock", "Oklawaha", "Trafford", "George"))) |>
    transform(gender = factor(gender, labels = c("Male", "Female"))) |>
    transform(size = factor(size, labels = c("<=2.3", ">2.3"))) |>
    transform(food = factor(food,
        labels = c("Fish", "Invertebrate", "Reptile", "Bird", "Other"))) |>
    glmbb(big = count ~ lake * gender * size * food,
        little = ~ lake * gender * size + food,
        family = poisson, data = _)
```

```
## Warning: glm.fit: fitted rates numerically 0 occurred

## Warning: glm.fit: fitted rates numerically 0 occurred
```

```
summary(gout)
```

```
##
## Results of search for hierarchical models with lowest AIC.
## Search was for all models with AIC no larger than min(AIC) + 10
## These are shown below.
##
##   criterion  weight    formula
##   288.0      0.903304  count ~ lake*food + size*food + lake*gender*size
##   293.7      0.050072  count ~ lake*food + gender*food + size*food + lake*gender*size
##   294.9      0.028388  count ~ lake*gender*size + lake*size*food
##   296.8      0.010643  count ~ size*food + lake*gender*size + lake*gender*food
##   297.5      0.007593  count ~ gender*food + lake*gender*size + lake*size*food
```

6.8 Named Arguments, Default Values for Arguments, Missing Arguments, Dot-Dot-Dot Arguments

These are covered in Section 7 of the 3701 lecture notes cited above.