**Unit 5 Reading Guide – MetaProgramming**

One of the most intriguing things about R is its ability to do metaprogramming. This is the idea that code is data that can be inspected and modified programmatically. This is a powerful idea; one that deeply influences much R code.

Closely related to metaprogramming is **non-standard evaluation**, NSE for short. This term, which is commonly used to describe the behaviour of R functions, is problematic in two ways. Firstly, NSE is actually a property of the argument (or arguments) of a function, so talking about NSE functions is a little sloppy. Secondly, it’s confusing to define something by what it’s not (standard), so in this book I’ll introduce more precise vocabulary.

**Chapter 17 Notes - Big Picture**

**17.2 Code is data**

The first big idea is that code is data: you can capture code and compute on it as you can with any other type of data. The first way you can capture code is with [rlang::expr()](https://rlang.r-lib.org/reference/nse-defuse.html). You can think of [expr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) as returning exactly what you pass in:

More formally, captured code is called an **expression**

**17.3 Code is a tree**

To do more complex manipulation with expressions, you need to fully understand their structure. Behind the scenes, almost every programming language represents code as a tree, often called the **abstract syntax tree**, or AST for short. R is unusual in that you can actually inspect and manipulate this tree.

**17.4 Code can generate code**

As well as seeing the tree from code typed by a human, you can also use code to create new trees. There are two main tools: [call2()](https://rlang.r-lib.org/reference/call2.html) and unquoting.

[call2()](https://rlang.r-lib.org/reference/call2.html) is often convenient to program with, but is a bit clunky for interactive use. An alternative technique is to build complex code trees by combining simpler code trees with a template. [expr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) and [enexpr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) have built-in support for this idea via !! (pronounced bang-bang), the **unquote operator**.

Unquoting gets even more useful when you wrap it up into a function, first using [enexpr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) to capture the user’s expression, then [expr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) and !! to create a new expression using a template. The example below shows how you can generate an expression that computes the coefficient of variation:

**17.5 Evaluation runs code**

Inspecting and modifying code gives you one set of powerful tools. You get another set of powerful tools when you **evaluate**, i.e. execute or run, an expression. Evaluating an expression requires an environment, which tells R what the symbols in the expression mean.

One of the big advantages of evaluating code manually is that you can tweak the environment. There are two main reasons to do this:

* To temporarily override functions to implement a domain specific language.
* To add a data mask so you can refer to variables in a data frame as if they are variables in an environment.

**17.6 Customising evaluation with functions**

It’s less obvious that you also bind names to functions, allowing you to override the behaviour of existing functions. This is a big idea that we’ll come back to in Chapter [21](https://adv-r.hadley.nz/translation.html#translation) where I explore generating HTML and LaTeX from R.

**17.7 Customising evaluation with data**

Rebinding functions is an extremely powerful technique, but it tends to require a lot of investment. A more immediately practical application is modifying evaluation to look for variables in a data frame instead of an environment. This idea powers the base [subset()](https://rdrr.io/r/base/subset.html) and [transform()](https://rdrr.io/r/base/transform.html) functions, as well as many tidyverse functions like [ggplot2::aes()](https://ggplot2.tidyverse.org/reference/aes.html) and [dplyr::mutate()](https://dplyr.tidyverse.org/reference/mutate.html)

As well as expression and environment, [eval\_tidy()](https://rlang.r-lib.org/reference/eval_tidy.html) also takes a **data mask**, which is typically a data frame:

**17.8 Quosures**

Fortunately we can solve this problem by using a new data structure: the **quosure** which bundles an expression with an environment. [eval\_tidy()](https://rlang.r-lib.org/reference/eval_tidy.html) knows how to work with quosures so all we need to do is switch out [enexpr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) for [enquo()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html):

**Chapter 18 Notes**

**18.2 Abstract syntax trees**

Expressions are also called **abstract syntax trees** (ASTs) because the structure of code is hierarchical and can be naturally represented as a tree.

**18.2.1 Drawing**

* The leaves of the tree are either symbols, like f and x, or constants, like 1 or "y". Symbols are drawn in purple and have rounded corners. Constants have black borders and square corners. Strings and symbols are easily confused, so strings are always surrounded in quotes.
* The branches of the tree are call objects, which represent function calls, and are drawn as orange rectangles. The first child (f) is the function that gets called; the second and subsequent children (x, "y", and 1) are the arguments to that function.

**18.2.2 Non-code components**

 They are abstract because they only capture important structural details of the code, not whitespace or comments

**18.2.3 Infix calls**

Every call in R can be written in tree form because any call can be written in prefix form

**18.3 Expressions**

Collectively, the data structures present in the AST are called expressions. An **expression** is any member of the set of base types created by parsing code: constant scalars, symbols, call objects, and pairlists. These are the data structures used to represent captured code from [expr()](https://rlang.r-lib.org/reference/nse-defuse.html), and [is\_expression(expr(...))](https://rlang.r-lib.org/reference/is_expression.html) is always true89. Constants, symbols and call objects are the most important,

**expression vectors**

In base R documentation “expression” is used to mean two things. As well as the definition above, expression is also used to refer to the type of object returned by [expression()](https://rdrr.io/r/base/expression.html) and [parse()](https://rdrr.io/r/base/parse.html)

**18.3.1 Constants**

Scalar constants are the simplest component of the AST. More precisely, a **constant** is either NULL or a length-1 atomic vector (or scalar, Section [3.2.1](https://adv-r.hadley.nz/vectors-chap.html#scalars)) like TRUE, 1L, 2.5 or "x".

Constants are self-quoting in the sense that the expression used to represent a constant is the same constant:

**18.3.2 Symbols**

A **symbol** represents the name of an object like x, mtcars, or mean. In base R, the terms symbol and name are used interchangeably (i.e. [is.name()](https://rdrr.io/r/base/name.html) is identical to [is.symbol()](https://rdrr.io/r/base/name.html))

You can create a symbol in two ways: by capturing code that references an object with [expr()](https://rlang.r-lib.org/reference/nse-defuse.html), or turning a string into a symbol with [rlang::sym()](https://rlang.r-lib.org/reference/sym.html):

**18.3.3 Calls**

A call object represents a captured function call. Call objects are a special type of list90 where the first component specifies the function to call (usually a symbol), and the remaining elements are the arguments for that call. Call objects create branches in the AST, because calls can be nested inside other calls.

**18.3.3.1 Subsetting**

Calls generally behave like lists, i.e. you can use standard subsetting tools. The first element of the call object is the function to call, which is usually a symbol

**18.3.3.2 Function position**

The first element of the call object is the **function position**. This contains the function that will be called when the object is evaluated, and is usually a symbol92:

**18.3.4 Summary**

The following table summarises the appearance of the different expression subtypes in [str()](https://rdrr.io/r/utils/str.html) and [typeof()](https://rdrr.io/r/base/typeof.html):

|  | [**str()**](https://rdrr.io/r/utils/str.html) | [**typeof()**](https://rdrr.io/r/base/typeof.html) |
| --- | --- | --- |
| Scalar constant | logi/int/num/chr | logical/integer/double/character |
| Symbol | symbol | symbol |
| Call object | language | language |
| Pairlist | Dotted pair list | pairlist |
| Expression vector | [expression()](https://rdrr.io/r/base/expression.html) | expression |

Both base R and rlang provide functions for testing for each type of input, although the types covered are slightly different. You can easily tell them apart because all the base functions start with is. and the rlang functions start with is\_.

|  | **base** | **rlang** |
| --- | --- | --- |
| Scalar constant | — | [is\_syntactic\_literal()](https://rlang.r-lib.org/reference/is_expression.html) |
| Symbol | [is.symbol()](https://rdrr.io/r/base/name.html) | [is\_symbol()](https://rlang.r-lib.org/reference/is_symbol.html) |
| Call object | [is.call()](https://rdrr.io/r/base/call.html) | [is\_call()](https://rlang.r-lib.org/reference/is_call.html) |
| Pairlist | [is.pairlist()](https://rdrr.io/r/base/list.html) | [is\_pairlist()](https://rlang.r-lib.org/reference/is_pairlist.html) |
| Expression vector | [is.expression()](https://rdrr.io/r/base/expression.html) | — |

**18.4.1 Operator precedence**

Infix functions introduce two sources of ambiguity93. The first source of ambiguity arises from infix functions

Programming languages use conventions called **operator precedence** to resolve this ambiguity.

In R, most operators are **left-associative**, i.e. the operations on the left are evaluated first:

**18.4.3 Parsing and deparsing**

Most of the time you type code into the console, and R takes care of turning the characters you’ve typed into an AST. But occasionally you have code stored in a string, and you want to parse it yourself. You can do so using [rlang::parse\_expr()](https://rlang.r-lib.org/reference/parse_expr.html)

The inverse of parsing is **deparsing**: given an expression, you want the string that would generate it.

**18.5 Walking AST with recursive functions**

* The **recursive case** handles the nodes in the tree. Typically, you’ll do something to each child of a node, usually calling the recursive function again, and then combine the results back together again. For expressions, you’ll need to handle calls and pairlists (function arguments).
* The **base case** handles the leaves of the tree. The base cases ensure that the function eventually terminates, by solving the simplest cases directly. For expressions, you need to handle symbols and constants in the base case.

**Chapter 19 notes**

**Quasiquotation**

 Where quotation is the act of capturing an unevaluated expression,**unquotation** is the ability to selectively evaluate parts of an otherwise quoted expression.

**19.2 Motivation**

We need some way to explicitly *unquote* the input to tell [cement()](https://rdrr.io/pkg/MASS/man/cement.html) to remove the automatic quote marks. Here we need time and name to be treated differently to Good. Quasiquotation gives us a standard tool to do so: !!, called “unquote”, and pronounced bang-bang. !! tells a quoting function to drop the implicit quotes:

It’s useful to compare [cement()](https://rdrr.io/pkg/MASS/man/cement.html) and [paste()](https://rdrr.io/r/base/paste.html) directly. [paste()](https://rdrr.io/r/base/paste.html) evaluates its arguments, so we must quote where needed; [cement()](https://rdrr.io/pkg/MASS/man/cement.html) quotes its arguments, so we must unquote where needed.

paste("Good", time, name)

cement(Good, !!time, !!name)

**19.2.1 Vocabulary**

The distinction between quoted and evaluated arguments is important:

* An **evaluated** argument obeys R’s usual evaluation rules.
* A **quoted** argument is captured by the function, and is processed in some custom way.

[paste()](https://rdrr.io/r/base/paste.html) evaluates all its arguments; [cement()](https://rdrr.io/pkg/MASS/man/cement.html) quotes all its arguments.

**19.3 Quoting**

The first part of quasiquotation is quotation: capturing an expression without evaluating it. We’ll need a pair of functions because the expression can be supplied directly or indirectly, via lazily-evaluated function argument.

**19.3.1 Capturing expressions**

There are four important quoting functions.

[expr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) is great for interactive exploration, because it captures what you, the developer, typed.

[enexpr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html). This captures what the caller supplied to the function by looking at the internal promise object that powers lazy evaluation

In short, use [enexpr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) and [enexprs()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) to capture the expressions supplied as arguments *by the user*. Use [expr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) and [exprs()](https://ggplot2.tidyverse.org/reference/tidyeval.html) to capture expressions that *you* supply.

**19.3.2 Capturing symbols**

Sometimes you only want to allow the user to specify a variable name, not an arbitrary expression. In this case, you can use [ensym()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) or [ensyms()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html). These are variants of [enexpr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) and [enexprs()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) that check the captured expression is either symbol or a string (which is converted to a symbol96). [ensym()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) and [ensyms()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) throw an error if given anything else.

**19.3.3 With base R**

The base equivalent of [expr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) is [quote()](https://rdrr.io/r/base/substitute.html):

The base function closest to [enexpr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) is [substitute()](https://rdrr.io/r/base/substitute.html):

The base equivalent to [exprs()](https://ggplot2.tidyverse.org/reference/tidyeval.html) is [alist()](https://rdrr.io/r/base/list.html):

The equivalent to [enexprs()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) is an undocumented feature of [substitute()](https://rdrr.io/r/base/substitute.html)97:

* [bquote()](https://rdrr.io/r/base/bquote.html) provides a limited form of quasiquotation, and is discussed in Section [19.5](https://adv-r.hadley.nz/quasiquotation.html#base-nonquote).
* [~](https://rdrr.io/r/base/tilde.html), the formula, is a quoting function that also captures the environment. It’s the inspiration for quosures, the topic of the next chapter, and is discussed in Section [20.3.4](https://adv-r.hadley.nz/evaluation.html#quosure-impl).

**19.3.4 Substitution**

You’ll most often see [substitute()](https://rdrr.io/r/base/substitute.html) used to capture unevaluated arguments. However, as well as quoting, [substitute()](https://rdrr.io/r/base/substitute.html) also does substitution (as its name suggests!). If you give it an expression, rather than a symbol, it will substitute in the values of symbols defined in the current environment.

**19.3.5 Summary**

When quoting (i.e. capturing code), there are two important distinctions:

* Is it supplied by the developer of the code or the user of the code? In other words, is it fixed (supplied in the body of the function) or varying (supplied via an argument)?
* Do you want to capture a single expression or multiple expressions?

| **Developer** | **User** |
| --- | --- |
| One | [expr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) | [enexpr()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) |
| Many | [exprs()](https://ggplot2.tidyverse.org/reference/tidyeval.html) | [enexprs()](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) |

| Table 19.2: base R quoting functions | | |
| --- | --- | --- |
|  | **Developer** | **User** |
| One | [quote()](https://rdrr.io/r/base/substitute.html) | [substitute()](https://rdrr.io/r/base/substitute.html) |
| Many | [alist()](https://rdrr.io/r/base/list.html) | [as.list(substitute(...()))](https://rdrr.io/r/base/list.html) |

**19.4 Unquoting**

Unquoting allows you to selectively evaluate parts of the expression that would otherwise be quoted, which effectively allows you to merge ASTs using a template AST. Since base functions don’t use unquoting, they instead use a variety of other techniques

**19.4.1 Unquoting one argument**

Use !! to unquote a single argument in a function call. !! takes a single expression, evaluates it, and inlines the result in the AST.

**19.4.2 Unquoting a function**

!! is most commonly used to replace the arguments to a function, but you can also use it to replace the function. The only challenge here is operator precedence: [expr(!!f(x, y))](https://dplyr.tidyverse.org/reference/tidyeval-compat.html) unquotes the result of f(x, y), so you need an extra pair of parentheses.

**19.4.3 Unquoting a missing argument**

Very occasionally it is useful to unquote a missing argument (Section [18.6.2](https://adv-r.hadley.nz/expressions.html#empty-symbol)), but the naive approach doesn’t work:

**19.4.4 Unquoting in special forms**

There are a few special forms where unquoting is a syntax error. Take [$](https://rdrr.io/r/base/Extract.html) for example: it must always be followed by the name of a variable, not another expression. This means attempting to unquote with [$](https://rdrr.io/r/base/Extract.html) will fail with a syntax error:

**19.4.5 Unquoting many arguments**

!! is a one-to-one replacement. !!! (called “unquote-splice”, and pronounced bang-bang-bang) is a one-to-many replacement. It takes a list of expressions and inserts them at the location of the !!!:

!!! can be used in any rlang function that takes ... regardless of whether or not ... is quoted or evaluated.

**19.4.6 The polite fiction of !!**

So far we have acted as if !! and !!! are regular prefix operators like [+](https://rdrr.io/r/base/Arithmetic.html) , [-](https://rdrr.io/r/base/Arithmetic.html), and [!](https://rdrr.io/r/base/Logic.html). They’re not. From R’s perspective, !! and !!! are simply the repeated application of [!](https://rdrr.io/r/base/Logic.html):

**19.5 Non-quoting**

* It is only easily used with your code; it is hard to apply it to arbitrary code supplied by a user.
* It does not provide an unquote-splice operator that allows you to unquote multiple expressions stored in a list.
* It lacks the ability to handle code accompanied by an environment, which is crucial for functions that evaluate code in the context of a data frame, like [subset()](https://rdrr.io/r/base/subset.html) and friends.

Base functions that quote an argument use some other technique to allow indirect specification. Base R approaches selectively turn quoting off, rather than using unquoting, so I call them **non-quoting** techniques.

There are four basic forms seen in base R:

* A pair of quoting and non-quoting functions
* A pair of quoting and non-quoting arguments
* An argument that controls whether a different argument is quoting or non-quoting
* Quoting if evaluation fails

**19.6 ... (dot-dot-dot)**

!!! is useful because it’s not uncommon to have a list of expressions that you want to insert into a call. It turns out that this pattern is common elsewhere

We say functions that support these tools, without quoting arguments, have **tidy dots**100

**19.6.2 exec()**

What if you want to use this technique with a function that doesn’t have tidy dots? One option is to use [rlang::exec()](https://rlang.r-lib.org/reference/exec.html) to call a function with some arguments supplied directly (in ...) and others indirectly (in a list):

**19.6.3 dots\_list()**

[list2()](https://rlang.r-lib.org/reference/list2.html) provides one other handy feature: by default it will ignore any empty arguments at the end. This is useful in functions like [tibble::tibble()](https://tibble.tidyverse.org/reference/tibble.html) because it means that you can easily change the order of variables without worrying about the final comma:

* .ignore\_empty allows you to control exactly which arguments are ignored. The default ignores a single trailing argument to get the behaviour described above, but you can choose to ignore all missing arguments, or no missing arguments.
* .homonyms controls what happens if multiple arguments use the same name:

**19.6.4 With base R**

Base R provides a Swiss army knife to solve these problems: [do.call()](https://rdrr.io/r/base/do.call.html). [do.call()](https://rdrr.io/r/base/do.call.html) has two main arguments. The first argument, what, gives a function to call. The second argument, args, is a list of arguments to pass to that function, and so [do.call("f", list(x, y, z))](https://rdrr.io/r/base/do.call.html) is equivalent to f(x, y, z)

**Chapter 20**

**Evaluation**

The user-facing inverse of quotation is unquotation: it gives the *user* the ability to selectively evaluate parts of an otherwise quoted argument. The developer-facing complement of quotation is evaluation: this gives the *developer* the ability to evaluate quoted expressions in custom environments to achieve specific goals.

* The quosure: a data structure that captures an expression along with its associated environment, as found in function arguments.
* The data mask, which makes it easier to evaluate an expression in the context of a data frame. This introduces potential evaluation ambiguity which we’ll then resolve with data pronouns.

Together, quasiquotation, quosures, and data masks form what we call **tidy evaluation**, or tidy eval for short.

**20.2 Evaluation basics**

Here we’ll explore the details of [eval()](https://rdrr.io/r/base/eval.html) which we briefly mentioned in the last chapter. It has two key arguments: expr and envir. The first argument, expr, is the object to evaluate, typically a symbol or expression104. None of the evaluation functions quote their inputs, so you’ll usually use them with [expr()](https://rlang.r-lib.org/reference/nse-defuse.html) or similar:

**20.2.1 Application: local()**

Sometimes you want to perform a chunk of calculation that creates some intermediate variables. The intermediate variables have no long-term use and could be quite large, so you’d rather not keep them around. One approach is to clean up after yourself using [rm()](https://rdrr.io/r/base/rm.html); another is to wrap the code in a function and just call it once. A more elegant approach is to use [local()](https://rdrr.io/r/base/eval.html):

**20.2.2 Application: source()**

We can create a simple version of [source()](https://rdrr.io/r/base/source.html) by combining [eval()](https://rdrr.io/r/base/eval.html) with [parse\_expr()](https://rlang.r-lib.org/reference/parse_expr.html) from Section [18.4.3](https://adv-r.hadley.nz/expressions.html#parsing). We read in the file from disk, use [parse\_expr()](https://rlang.r-lib.org/reference/parse_expr.html) to parse the string into a list of expressions, and then use [eval()](https://rdrr.io/r/base/eval.html) to evaluate each element in turn.

**20.3 Quosures**

Almost every use of [eval()](https://rdrr.io/r/base/eval.html) involves both an expression and environment. This coupling is so important that we need a data structure that can hold both pieces. Base R does not have such a structure105 so rlang fills the gap with the **quosure**, an object that contains an expression and an environment. The name is a portmanteau of quoting and closure, because a quosure both quotes the expression and encloses the environment.

**20.3.1 Creating**

There are three ways to create quosures:

* Use [enquo()](https://rlang.r-lib.org/reference/nse-defuse.html) and [enquos()](https://rlang.r-lib.org/reference/nse-defuse.html) to capture user-supplied expressions. The vast majority of quosures should be created this way.
* [quo()](https://rlang.r-lib.org/reference/nse-defuse.html) and [quos()](https://rlang.r-lib.org/reference/nse-defuse.html) exist to match to [expr()](https://rlang.r-lib.org/reference/nse-defuse.html) and [exprs()](https://rlang.r-lib.org/reference/nse-defuse.html), but they are included only for the sake of completeness and are needed very rarely. If you find yourself using them, think carefully if [expr()](https://rlang.r-lib.org/reference/nse-defuse.html) and careful unquoting can eliminate the need to capture the environment.
* [new\_quosure()](https://rlang.r-lib.org/reference/as_quosure.html) create a quosure from its components: an expression and an environment. This is rarely needed in practice, but is useful for learning, so is used a lot in this chapter.