Here are the R packages that I’ll be using:

library(tidyverse)

library(rlang)

library(rsample)

library(yardstick)

library(purrr)

First, duplicate the process of reading in the data and adding two new columns:

stocks <- read.csv('http://blog.mindymallory.com/wp-content/uploads/2018/02/stocks.csv') %>%

as\_tibble() %>%

mutate(

USStockUse = USEndingStocks/USTotalUse,

WorldStockUse = ROWEndingStocks/WorldTotalUse

)

stocks

## # A tibble: 43 x 8

## Year USEndingStocks ROWEndingStocks USTotalUse

##

## 1 1975 633. 36411 5767.

## 2 1976 1136. 39491 5789.

## 3 1977 1436. 40833 6207.

## 4 1978 1710. 47957 6995.

## 5 1979 2034. 59481 7604.

## 6 1980 1392. 67180 7282.

## 7 1981 2537. 62725 6975.

## 8 1982 3523. 60273 7249.

## 9 1983 1006. 63421 6693.

## 10 1984 1648. 76287 7032.

## # ... with 33 more rows, and 4 more variables:

## # WorldTotalUse , PriceRecievedFarmers ,

## # USStockUse , WorldStockUse

The blog post has an excellent description of cross-validation and looked at five different models that encoded the US and World stock-use predictors. Either a log- or inverse-transformation was applied and then polynomial basis functions were used on these features to demonstrate overfitting.

The blog post has some for loops to do the resampling and I volunteered to show how to do it with some tidy modeling packages.

**Tidy Cross-Validation**

First, let’s take the *easy* part. Instead of using for loops, we can use the new infrastructure in the tidyverse to resample the model. The rsample package has some functions for different types of resampling and we will use the same procedure:

Library(rsample)

set.seed(918)

resamp\_info <- vfold\_cv(stocks, v = 5)

resamp\_info

## # 5-fold cross-validation

## # A tibble: 5 x 2

## splits id

##

## 1 Fold1

## 2 Fold2

## 3 Fold3

## 4 Fold4

## 5 Fold5

The first column in the tibble is a set of “rsplit” objects that define how the data are split for each fold of cross-validation. Each one fully and efficiently encapsulates everything what is needed to get the two divisions of the original data. In rsample, to avoid naming confusion, we label the two resulting data sets as:

* The **analysis data** are those that we selected in the resample. For a bootstrap, this is the sample with replacement. For 5-fold cross-validation, this is the 80% of the data. These data are often used to fit a model or calculate a statistic in traditional bootstrapping.
* The **assessment data** are usually the section of the original data not covered by the analysis set. Again, in 5-fold CV, this is the 20% held out. These data are often used to evaluate the performance of a model that was fit to the analysis data.

To get these partitions for the first split there are functions analysis and assessment that return the appropriate data frames when given an rsplit:

# printing just shows the #rows per analysis/assessment/overall

resamp\_info$splits[[1]]

## <34/9/43>

# data used for modeling:

analysis(resamp\_info$splits[[1]])

## # A tibble: 34 x 8

## Year USEndingStocks ROWEndingStocks USTotalUse

##

## 1 1975 633. 36411 5767.

## 2 1976 1136. 39491 5789.

## 3 1977 1436. 40833 6207.

## 4 1979 2034. 59481 7604.

## 5 1980 1392. 67180 7282.

## 6 1981 2537. 62725 6975.

## 7 1982 3523. 60273 7249.

## 8 1983 1006. 63421 6693.

## 9 1984 1648. 76287 7032.

## 10 1985 4040. 75069 6494.

## # ... with 24 more rows, and 4 more variables:

## # WorldTotalUse , PriceRecievedFarmers ,

## # USStockUse , WorldStockUse

The first model for the data contained the US stock-use data with an inverse transformation. Let’s side-step the polynomial model tuning for now and just fit a quadratic model. To make things easier, I’ll define a function that can be used to fit the model when given an rsplit object and return the holdout mean squared error (MSE):

glm\_results <- function(split, ...) {

# Get the data used ot fit the model aka the "analysis" set

# and fit the model with a formula given in the ...

mod <- glm(data = analysis(split), ...)

# Get predictions on the other data (aka the "assessment" set

# and compute some metrics

holdout <- assessment(split)

# Compute performance using the yardstick package

rmse <- holdout %>%

mutate(pred = predict(mod, holdout)) %>%

rmse(truth = PriceRecievedFarmers, estimate = pred)

rmse^2

}

We can use this with any formula since it is just passed to glm using the ellipses. For example to get the holdout MSE estimate for the first fold:

glm\_results(resamp\_info$splits[[1]], formula = PriceRecievedFarmers ~ poly(1 / USStockUse, 2))

## [1] 1.67

To get these statistics for all folds, purrr::map\_dbl is used to add another column:

resamp\_info <- resamp\_info %>%

mutate(

model\_1\_deg\_2 =

map\_dbl(

splits,

glm\_results,

PriceRecievedFarmers ~ poly(1 / USStockUse, 2)

)

)

resamp\_info

## # 5-fold cross-validation

## # A tibble: 5 x 3

## splits id model\_1\_deg\_2

## \*

## 1 Fold1 1.67

## 2 Fold2 0.558

## 3 Fold3 0.596

## 4 Fold4 9.58

## 5 Fold5 0.248

That’s a lot of variation in the outcome! The mean value is fairly consistent with the blog post though:

resamp\_info %>% select(model\_1\_deg\_2) %>% colMeans()

## model\_1\_deg\_2

## 2.53

# MSE = 2.675 in the blog post

*K*-fold cross-validation is one of the noisiest resampling methods so this difference isn’t too surprising.

This same process could be repeated for each polynomial degree to get new columns for this model (we’ll discuss this below). The good things about doing things this way:

* It is a lot cleaner (so far) than doing for loops.
* Other tidyverse infrastructure can be used. For example, tidyposterior is a great way to do model comparisons with resampling and Bayesian analysis.
* It is simple to change resampling methods. Suppose you wanted to change to a larger number of bootstrap resamples (given the variance shown above). The same infrastructure can be easily exchanged; resample::bootstraps is used in place of rsample::vfold\_cv.

Code Chunks – tidyposterior

INSTALLATION

You can install the released version of tidyposterior from CRAN with:

install.packages("tidyposterior")

EXAMPLE

To illustrate, here are some example objects using 10-fold cross-validation for a simple two-class problem:

library(tidymodels)

#> ── Attaching packages ───────────────────────────────── tidymodels 0.1.2.9000 ──

#> ✓ broom 0.7.4 ✓ recipes 0.1.15

#> ✓ dials 0.0.9 ✓ rsample 0.0.9

#> ✓ dplyr 1.0.4 ✓ tibble 3.1.0

#> ✓ ggplot2 3.3.3 ✓ tidyr 1.1.3

#> ✓ infer 0.5.4 ✓ tune 0.1.3

#> ✓ modeldata 0.1.0 ✓ workflows 0.2.2

#> ✓ parsnip 0.1.5 ✓ yardstick 0.0.7

#> ✓ purrr 0.3.4

#> ── Conflicts ───────────────────────────────────────── tidymodels\_conflicts() ──

#> x purrr::discard() masks scales::discard()

#> x dplyr::filter() masks stats::filter()

#> x dplyr::lag() masks stats::lag()

#> x recipes::step() masks stats::step()

library(tidyposterior)

data(two\_class\_dat, package = "modeldata")

set.seed(100)

folds <- vfold\_cv(two\_class\_dat)

We can define two different models (for simplicity, with no tuning parameters).

logistic\_reg\_glm\_spec <-

logistic\_reg() %>%

set\_engine('glm')

mars\_earth\_spec <-

mars(prod\_degree = 1) %>%

set\_engine('earth') %>%

set\_mode('classification')

For tidymodels, the  
*tune* :  : *fit*\_*resamples*()  
function can be used to estimate performance for each model/resample:

rs\_ctrl <- control\_resamples(save\_workflow = TRUE)

logistic\_reg\_glm\_res <-

logistic\_reg\_glm\_spec %>%

fit\_resamples(Class ~ ., resamples = folds, control = rs\_ctrl)

mars\_earth\_res <-

mars\_earth\_spec %>%

fit\_resamples(Class ~ ., resamples = folds, control = rs\_ctrl)

From these, there are several ways to pass the results to the perf\_mod() function. The most general approach is to have a data frame with the resampling labels (i.e., one or more id columns) as well as columns for each model that you would like to compare.

For the model results above,  
*tune* :  : *collect*\_*metrics*()  
can be used along with some basic data manipulation steps:

logistic\_roc <-

collect\_metrics(logistic\_reg\_glm\_res, summarize = FALSE) %>%

dplyr::filter(.metric == "roc\_auc") %>%

dplyr::select(id, logistic = .estimate)

mars\_roc <-

collect\_metrics(mars\_earth\_res, summarize = FALSE) %>%

dplyr::filter(.metric == "roc\_auc") %>%

dplyr::select(id, mars = .estimate)

resamples\_df <- full\_join(logistic\_roc, mars\_roc, by = "id")

resamples\_df

#> # A tibble: 10 x 3

#> id logistic mars

#> <chr> <dbl> <dbl>

#> 1 Fold01 0.856 0.845

#> 2 Fold02 0.933 0.951

#> 3 Fold03 0.934 0.937

#> 4 Fold04 0.864 0.858

#> 5 Fold05 0.847 0.854

#> 6 Fold06 0.911 0.840

#> 7 Fold07 0.867 0.858

#> 8 Fold08 0.886 0.876

#> 9 Fold09 0.898 0.898

#> 10 Fold10 0.906 0.894

We can then give this directly to perf\_mod():

set.seed(101)

roc\_model\_via\_df <- perf\_mod(resamples\_df, iter = 2000)

#>

#> SAMPLING FOR MODEL 'continuous' NOW (CHAIN 1).

#> Chain 1:

#> Chain 1: Gradient evaluation took 0.0001 seconds

#> Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 1 seconds.

#> Chain 1: Adjust your expectations accordingly!

#> Chain 1:

#> Chain 1:

#> Chain 1: Iteration: 1 / 2000 [ 0%] (Warmup)

#> Chain 1: Iteration: 200 / 2000 [ 10%] (Warmup)

#> Chain 1: Iteration: 400 / 2000 [ 20%] (Warmup)

#> Chain 1: Iteration: 600 / 2000 [ 30%] (Warmup)

#> Chain 1: Iteration: 800 / 2000 [ 40%] (Warmup)

#> Chain 1: Iteration: 1000 / 2000 [ 50%] (Warmup)

#> Chain 1: Iteration: 1001 / 2000 [ 50%] (Sampling)

#> Chain 1: Iteration: 1200 / 2000 [ 60%] (Sampling)

#> Chain 1: Iteration: 1400 / 2000 [ 70%] (Sampling)

#> Chain 1: Iteration: 1600 / 2000 [ 80%] (Sampling)

#> Chain 1: Iteration: 1800 / 2000 [ 90%] (Sampling)

#> Chain 1: Iteration: 2000 / 2000 [100%] (Sampling)

#> Chain 1:

#> Chain 1: Elapsed Time: 0.46459 seconds (Warm-up)

#> Chain 1: 0.17115 seconds (Sampling)

#> Chain 1: 0.63574 seconds (Total)

#> Chain 1:

#>

#> SAMPLING FOR MODEL 'continuous' NOW (CHAIN 2).

#> Chain 2:

#> Chain 2: Gradient evaluation took 1.8e-05 seconds

#> Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 0.18 seconds.

#> Chain 2: Adjust your expectations accordingly!

#> Chain 2:

#> Chain 2:

#> Chain 2: Iteration: 1 / 2000 [ 0%] (Warmup)

#> Chain 2: Iteration: 200 / 2000 [ 10%] (Warmup)

#> Chain 2: Iteration: 400 / 2000 [ 20%] (Warmup)

#> Chain 2: Iteration: 600 / 2000 [ 30%] (Warmup)

#> Chain 2: Iteration: 800 / 2000 [ 40%] (Warmup)

#> Chain 2: Iteration: 1000 / 2000 [ 50%] (Warmup)

#> Chain 2: Iteration: 1001 / 2000 [ 50%] (Sampling)

#> Chain 2: Iteration: 1200 / 2000 [ 60%] (Sampling)

#> Chain 2: Iteration: 1400 / 2000 [ 70%] (Sampling)

#> Chain 2: Iteration: 1600 / 2000 [ 80%] (Sampling)

#> Chain 2: Iteration: 1800 / 2000 [ 90%] (Sampling)

#> Chain 2: Iteration: 2000 / 2000 [100%] (Sampling)

#> Chain 2:

#> Chain 2: Elapsed Time: 0.477826 seconds (Warm-up)

#> Chain 2: 0.109854 seconds (Sampling)

#> Chain 2: 0.58768 seconds (Total)

#> Chain 2:

#>

#> SAMPLING FOR MODEL 'continuous' NOW (CHAIN 3).

#> Chain 3:

#> Chain 3: Gradient evaluation took 1.6e-05 seconds

#> Chain 3: 1000 transitions using 10 leapfrog steps per transition would take 0.16 seconds.

#> Chain 3: Adjust your expectations accordingly!

#> Chain 3:

#> Chain 3:

#> Chain 3: Iteration: 1 / 2000 [ 0%] (Warmup)

#> Chain 3: Iteration: 200 / 2000 [ 10%] (Warmup)

#> Chain 3: Iteration: 400 / 2000 [ 20%] (Warmup)

#> Chain 3: Iteration: 600 / 2000 [ 30%] (Warmup)

#> Chain 3: Iteration: 800 / 2000 [ 40%] (Warmup)

#> Chain 3: Iteration: 1000 / 2000 [ 50%] (Warmup)

#> Chain 3: Iteration: 1001 / 2000 [ 50%] (Sampling)

#> Chain 3: Iteration: 1200 / 2000 [ 60%] (Sampling)

#> Chain 3: Iteration: 1400 / 2000 [ 70%] (Sampling)

#> Chain 3: Iteration: 1600 / 2000 [ 80%] (Sampling)

#> Chain 3: Iteration: 1800 / 2000 [ 90%] (Sampling)

#> Chain 3: Iteration: 2000 / 2000 [100%] (Sampling)

#> Chain 3:

#> Chain 3: Elapsed Time: 0.41594 seconds (Warm-up)

#> Chain 3: 0.127105 seconds (Sampling)

#> Chain 3: 0.543045 seconds (Total)

#> Chain 3:

#>

#> SAMPLING FOR MODEL 'continuous' NOW (CHAIN 4).

#> Chain 4:

#> Chain 4: Gradient evaluation took 1.6e-05 seconds

#> Chain 4: 1000 transitions using 10 leapfrog steps per transition would take 0.16 seconds.

#> Chain 4: Adjust your expectations accordingly!

#> Chain 4:

#> Chain 4:

#> Chain 4: Iteration: 1 / 2000 [ 0%] (Warmup)

#> Chain 4: Iteration: 200 / 2000 [ 10%] (Warmup)

#> Chain 4: Iteration: 400 / 2000 [ 20%] (Warmup)

#> Chain 4: Iteration: 600 / 2000 [ 30%] (Warmup)

#> Chain 4: Iteration: 800 / 2000 [ 40%] (Warmup)

#> Chain 4: Iteration: 1000 / 2000 [ 50%] (Warmup)

#> Chain 4: Iteration: 1001 / 2000 [ 50%] (Sampling)

#> Chain 4: Iteration: 1200 / 2000 [ 60%] (Sampling)

#> Chain 4: Iteration: 1400 / 2000 [ 70%] (Sampling)

#> Chain 4: Iteration: 1600 / 2000 [ 80%] (Sampling)

#> Chain 4: Iteration: 1800 / 2000 [ 90%] (Sampling)

#> Chain 4: Iteration: 2000 / 2000 [100%] (Sampling)

#> Chain 4:

#> Chain 4: Elapsed Time: 0.397339 seconds (Warm-up)

#> Chain 4: 0.181545 seconds (Sampling)

#> Chain 4: 0.578884 seconds (Total)

#> Chain 4:

From this, the posterior distributions for each model can be obtained from the tidy() method:

roc\_model\_via\_df %>%

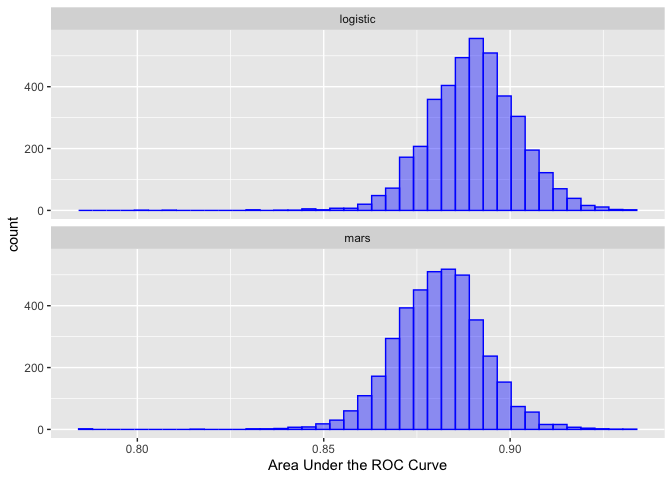
tidy() %>%

ggplot(aes(x = posterior)) +

geom\_histogram(bins = 40, col = "blue", fill = "blue", alpha = .4) +

facet\_wrap(~ model, ncol = 1) +

xlab("Area Under the ROC Curve")



See contrast\_models() for how to analyze these distributions

**Tidy Model Specification (maybe)**

The model specification part is, for me, a lot more difficult to tidy. It would be good to be able to state what predictors that we want, specify the polynomial degree, and have a function to generate the appropriate formula. The original post sensibly just types the terms out.

I spent some time thinking about how we could use expressions and tidy evaluation to make it a little less script-like. The problem is that the solution took me a while to write and, arguably, it doesn’t really buy you much more than the original code (apart from the potential copy/paste duplication errors).

Code Chunks – Tidy Expressions

To compute on the language, we first need to understand its structure. That requires some new vocabulary, some new tools, and some new ways of thinking about R code. The first of these is the distinction between an operation and its result. Take the following code, which multiplies a variable x by 10 and saves the result to a new variable called y. It doesn’t work because we haven’t defined a variable called x:

y <- x \* 10

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

Copy

It would be nice if we could capture the intent of the code without executing it. In other words, how can we separate our description of the action from the action itself?

One way is to use rlang::expr():

z <- rlang::expr(y <- x \* 10)

z

#> y <- x \* 10

Copy

expr() returns an expression, an object that captures the structure of the code without evaluating it (i.e. running it). If you have an expression, you can evaluate it with base::eval():

x <- 4

eval(z)

y

#> [1] 40

Copy

The focus of this chapter is the data structures that underlie expressions. Mastering this knowledge will allow you to inspect and modify captured code, and to generate code with code. We’ll come back to expr() in Chapter 19, and to eval() in Chapter 20.

### **Outline**

* Section 18.2 introduces the idea of the abstract syntax tree (AST), and reveals the tree like structure that underlies all R code.
* Section 18.3 dives into the details of the data structures that underpin the AST: constants, symbols, and calls, which are collectively known as expressions.
* Section 18.4 covers parsing, the act of converting the linear sequence of character in code into the AST, and uses that idea to explore some details of R’s grammar.
* Section 18.5 shows you how you can use recursive functions to compute on the language, writing functions that compute with expressions.
* Section 18.6 circles back to three more specialised data structures: pairlists, missing arguments, and expression vectors.

### **Prerequisites**

Make sure you’ve read the metaprogramming overview in Chapter 17 to get a broad overview of the motivation and the basic vocabulary. You’ll also need the rlang package to capture and compute on expressions, and the lobstr package to visualise them.

library(rlang)

library(lobstr)

Copy

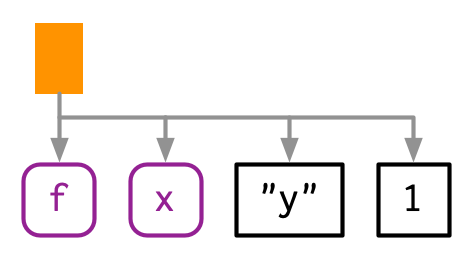
## 18.2 Abstract syntax trees

Expressions are also called **abstract syntax trees** (ASTs) because the structure of code is hierarchical and can be naturally represented as a tree. Understanding this tree structure is crucial for inspecting and modifying expressions (i.e. metaprogramming).

### **18.2.1 Drawing**

We’ll start by introducing some conventions for drawing ASTs, beginning with a simple call that shows their main components: f(x, "y", 1). I’ll draw trees in two ways87:

* By “hand” (i.e. with OmniGraffle):



* With lobstr::ast():
* lobstr::ast(f(x, "y", 1))
* #> █─f
* #> ├─x
* #> ├─"y"

#> └─1

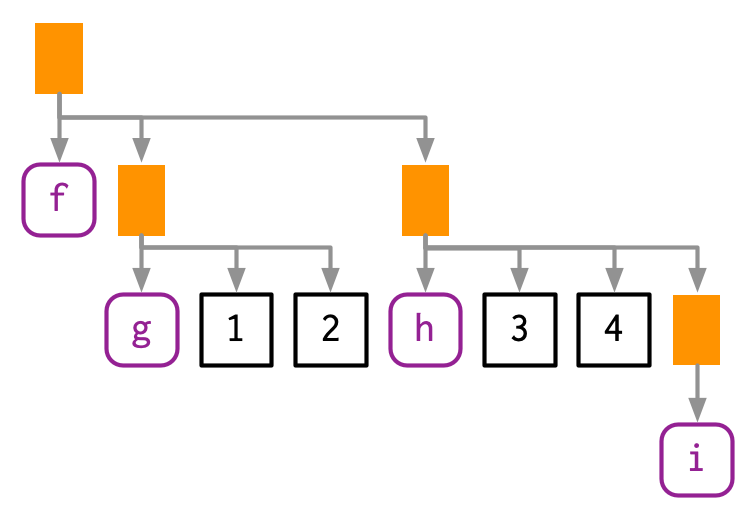
Copy

Both approaches share conventions as much as possible:

* 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.

Colours will be shown when you call ast(), but do not appear in the book for complicated technical reasons.

The above example only contained one function call, making for a very shallow tree. Most expressions will contain considerably more calls, creating trees with multiple levels. For example, consider the AST for f(g(1, 2), h(3, 4, i())):



lobstr::ast(f(g(1, 2), h(3, 4, i())))

#> █─f

#> ├─█─g

#> │ ├─1

#> │ └─2

#> └─█─h

#> ├─3

#> ├─4

#> └─█─i

Copy

You can read the hand-drawn diagrams from left-to-right (ignoring vertical position), and the lobstr-drawn diagrams from top-to-bottom (ignoring horizontal position). The depth within the tree is determined by the nesting of function calls. This also determines evaluation order, as evaluation generally proceeds from deepest-to-shallowest, but this is not guaranteed because of lazy evaluation (Section 6.5). Also note the appearance of i(), a function call with no arguments; it’s a branch with a single (symbol) leaf.

### **18.2.2 Non-code components**

You might have wondered what makes these abstract syntax trees. They are abstract because they only capture important structural details of the code, not whitespace or comments:

ast(

f(x, y) # important!

)

#> █─f

#> ├─x

#> └─y

Copy

There’s only one place where whitespace affects the AST:

lobstr::ast(y <- x)

#> █─`<-`

#> ├─y

#> └─x

lobstr::ast(y < -x)

#> █─`<`

#> ├─y

#> └─█─`-`

#> └─x

Copy

### **18.2.3 Infix calls**

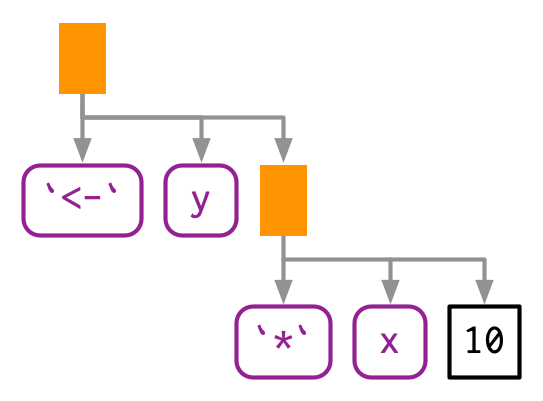
Every call in R can be written in tree form because any call can be written in prefix form (Section 6.8.1). Take y <- x \* 10 again: what are the functions that are being called? It is not as easy to spot as f(x, 1) because this expression contains two infix calls: <- and \*. That means that these two lines of code are equivalent:

y <- x \* 10

`<-`(y, `\*`(x, 10))

Copy

And they both have this AST88:



lobstr::ast(y <- x \* 10)

#> █─`<-`

#> ├─y

#> └─█─`\*`

#> ├─x

#> └─10

Copy

There really is no difference between the ASTs, and if you generate an expression with prefix calls, R will still print it in infix form:

expr(`<-`(y, `\*`(x, 10)))

#> y <- x \* 10

Copy

The order in which infix operators are applied is governed by a set of rules called operator precedence, and we’ll use lobstr::ast() to explore them in Section 18.4.1.

### **18.2.4 Exercises**

1. Reconstruct the code represented by the trees below:
2. #> █─f
3. #> └─█─g
4. #> └─█─h
5. #> █─`+`
6. #> ├─█─`+`
7. #> │ ├─1
8. #> │ └─2
9. #> └─3
10. #> █─`\*`
11. #> ├─█─`(`
12. #> │ └─█─`+`
13. #> │ ├─x
14. #> │ └─y

#> └─z

Copy

1. Draw the following trees by hand and then check your answers with lobstr::ast().
2. f(g(h(i(1, 2, 3))))
3. f(1, g(2, h(3, i())))

f(g(1, 2), h(3, i(4, 5)))

Copy

1. What’s happening with the ASTs below? (Hint: carefully read ?"^".)
2. lobstr::ast(`x` + `y`)
3. #> █─`+`
4. #> ├─x
5. #> └─y
6. lobstr::ast(x \*\* y)
7. #> █─`^`
8. #> ├─x
9. #> └─y
10. lobstr::ast(1 -> x)
11. #> █─`<-`
12. #> ├─x

#> └─1

Copy

1. What is special about the AST below? (Hint: re-read Section 6.2.1.)
2. lobstr::ast(function(x = 1, y = 2) {})
3. #> █─`function`
4. #> ├─█─x = 1
5. #> │ └─y = 2
6. #> ├─█─`{`

#> └─<inline srcref>

Copy

1. What does the call tree of an if statement with multiple else if conditions look like? Why?

## 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(), and is\_expression(expr(...)) is always true89. Constants, symbols and call objects are the most important, and are discussed below. Pairlists and empty symbols are more specialised and we’ll come back to them in Sections 18.6.1 and Section 18.6.2.

NB: 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() and parse(), which are basically lists of expressions as defined above. In this book I’ll call these **expression vectors**, and I’ll come back to them in Section 18.6.3.

### **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) like TRUE, 1L, 2.5 or "x". You can test for a constant with rlang::is\_syntactic\_literal().

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

identical(expr(TRUE), TRUE)

#> [1] TRUE

identical(expr(1), 1)

#> [1] TRUE

identical(expr(2L), 2L)

#> [1] TRUE

identical(expr("x"), "x")

#> [1] TRUE

Copy

### **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() is identical to is.symbol()), but in this book I used symbol consistently because “name” has many other meanings.

You can create a symbol in two ways: by capturing code that references an object with expr(), or turning a string into a symbol with rlang::sym():

expr(x)

#> x

sym("x")

#> x

Copy

You can turn a symbol back into a string with as.character() or rlang::as\_string(). as\_string() has the advantage of clearly signalling that you’ll get a character vector of length 1.

as\_string(expr(x))

#> [1] "x"

Copy

You can recognise a symbol because it’s printed without quotes, str() tells you that it’s a symbol, and is.symbol() is TRUE:

str(expr(x))

#> symbol x

is.symbol(expr(x))

#> [1] TRUE

Copy

The symbol type is not vectorised, i.e. a symbol is always length 1. If you want multiple symbols, you’ll need to put them in a list, using (e.g.) rlang::syms().

### **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.

You can identify a call object when printed because it looks just like a function call. Confusingly typeof() and str() print “language”91 for call objects, but is.call() returns TRUE:

lobstr::ast(read.table("important.csv", row.names = FALSE))

#> █─read.table

#> ├─"important.csv"

#> └─row.names = FALSE

x <- expr(read.table("important.csv", row.names = FALSE))

typeof(x)

#> [1] "language"

is.call(x)

#> [1] TRUE

Copy

#### 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:

x[[1]]

#> read.table

is.symbol(x[[1]])

#> [1] TRUE

Copy

The remainder of the elements are the arguments:

as.list(x[-1])

#> [[1]]

#> [1] "important.csv"

#>

#> $row.names

#> [1] FALSE

Copy

You can extract individual arguments with [[ or, if named, $:

x[[2]]

#> [1] "important.csv"

x$row.names

#> [1] FALSE

Copy

You can determine the number of arguments in a call object by subtracting 1 from its length:

length(x) - 1

#> [1] 2

Copy

Extracting specific arguments from calls is challenging because of R’s flexible rules for argument matching: it could potentially be in any location, with the full name, with an abbreviated name, or with no name. To work around this problem, you can use rlang::call\_standardise() which standardises all arguments to use the full name:

rlang::call\_standardise(x)

#> read.table(file = "important.csv", row.names = FALSE)

Copy

(NB: If the function uses ... it’s not possible to standardise all arguments.)

Calls can be modified in the same way as lists:

x$header <- TRUE

x

#> read.table("important.csv", row.names = FALSE, header = TRUE)

Copy

#### 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:

lobstr::ast(foo())

#> █─foo

Copy

While R allows you to surround the name of the function with quotes, the parser converts it to a symbol:

lobstr::ast("foo"())

#> █─foo

Copy

However, sometimes the function doesn’t exist in the current environment and you need to do some computation to retrieve it: for example, if the function is in another package, is a method of an R6 object, or is created by a function factory. In this case, the function position will be occupied by another call:

lobstr::ast(pkg::foo(1))

#> █─█─`::`

#> │ ├─pkg

#> │ └─foo

#> └─1

lobstr::ast(obj$foo(1))

#> █─█─`$`

#> │ ├─obj

#> │ └─foo

#> └─1

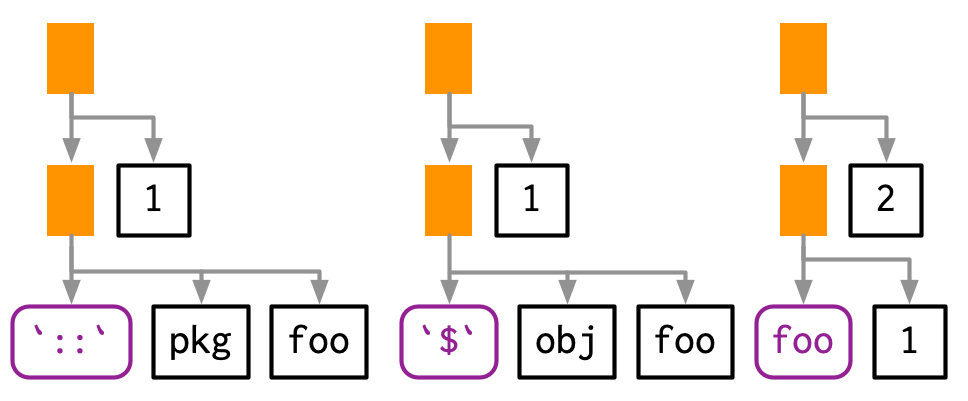
lobstr::ast(foo(1)(2))

#> █─█─foo

#> │ └─1

#> └─2

Copy



#### 18.3.3.3 Constructing

You can construct a call object from its components using rlang::call2(). The first argument is the name of the function to call (either as a string, a symbol, or another call). The remaining arguments will be passed along to the call:

call2("mean", x = expr(x), na.rm = TRUE)

#> mean(x = x, na.rm = TRUE)

call2(expr(base::mean), x = expr(x), na.rm = TRUE)

#> base::mean(x = x, na.rm = TRUE)

Copy

Infix calls created in this way still print as usual.

call2("<-", expr(x), 10)

#> x <- 10

Copy

Using call2() to create complex expressions is a bit clunky. You’ll learn another technique in Chapter 19.

### **18.3.4 Summary**

The following table summarises the appearance of the different expression subtypes in str() and typeof():

|  | **str()** | **typeof()** |
| --- | --- | --- |
| 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() | 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() |
| Symbol | is.symbol() | is\_symbol() |
| Call object | is.call() | is\_call() |
| Pairlist | is.pairlist() | is\_pairlist() |
| Expression vector | is.expression() | — |

### **18.3.5 Exercises**

1. Which two of the six types of atomic vector can’t appear in an expression? Why? Similarly, why can’t you create an expression that contains an atomic vector of length greater than one?
2. What happens when you subset a call object to remove the first element? e.g. expr(read.csv("foo.csv", header = TRUE))[-1]. Why?
3. Describe the differences between the following call objects.
4. x <- 1:10
5. call2(median, x, na.rm = TRUE)
6. call2(expr(median), x, na.rm = TRUE)
7. call2(median, expr(x), na.rm = TRUE)

call2(expr(median), expr(x), na.rm = TRUE)

Copy

1. rlang::call\_standardise() doesn’t work so well for the following calls. Why? What makes mean() special?
2. call\_standardise(quote(mean(1:10, na.rm = TRUE)))
3. #> mean(x = 1:10, na.rm = TRUE)
4. call\_standardise(quote(mean(n = T, 1:10)))
5. #> mean(x = 1:10, n = T)
6. call\_standardise(quote(mean(x = 1:10, , TRUE)))

#> mean(x = 1:10, , TRUE)

Copy

1. Why does this code not make sense?
2. x <- expr(foo(x = 1))

names(x) <- c("x", "y")

Copy

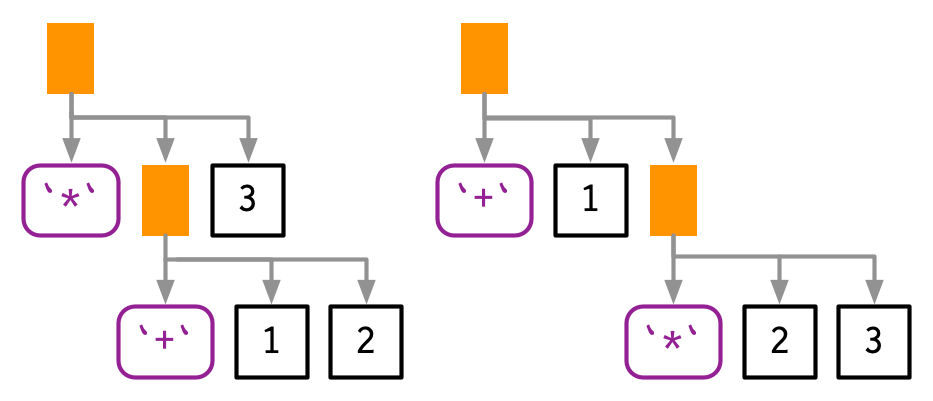
1. Construct the expression if(x > 1) "a" else "b" using multiple calls to call2(). How does the code structure reflect the structure of the AST?

## 18.4 Parsing and grammar

We’ve talked a lot about expressions and the AST, but not about how expressions are created from code that you type (like "x + y"). The process by which a computer language takes a string and constructs an expression is called **parsing**, and is governed by a set of rules known as a **grammar**. In this section, we’ll use lobstr::ast() to explore some of the details of R’s grammar, and then show how you can transform back and forth between expressions and strings.

### **18.4.1 Operator precedence**

Infix functions introduce two sources of ambiguity93. The first source of ambiguity arises from infix functions: what does 1 + 2 \* 3 yield? Do you get 9 (i.e. (1 + 2) \* 3), or 7 (i.e. 1 + (2 \* 3))? In other words, which of the two possible parse trees below does R use?



Programming languages use conventions called **operator precedence** to resolve this ambiguity. We can use ast() to see what R does:

lobstr::ast(1 + 2 \* 3)

#> █─`+`

#> ├─1

#> └─█─`\*`

#> ├─2

#> └─3

Copy

Predicting the precedence of arithmetic operations is usually easy because it’s drilled into you in school and is consistent across the vast majority of programming languages.

Predicting the precedence of other operators is harder. There’s one particularly surprising case in R: ! has a much lower precedence (i.e. it binds less tightly) than you might expect. This allows you to write useful operations like:

lobstr::ast(!x %in% y)

#> █─`!`

#> └─█─`%in%`

#> ├─x

#> └─y

Copy

R has over 30 infix operators divided into 18 precedence groups. While the details are described in ?Syntax, very few people have memorised the complete ordering. If there’s any confusion, use parentheses!

lobstr::ast((1 + 2) \* 3)

#> █─`\*`

#> ├─█─`(`

#> │ └─█─`+`

#> │ ├─1

#> │ └─2

#> └─3

Copy

Note the appearance of the parentheses in the AST as a call to the ( function.

### **18.4.2 Associativity**

The second source of ambiguity is introduced by repeated usage of the same infix function. For example, is 1 + 2 + 3 equivalent to (1 + 2) + 3 or to 1 + (2 + 3)? This normally doesn’t matter because x + (y + z) == (x + y) + z, i.e. addition is associative, but is needed because some S3 classes define + in a non-associative way. For example, ggplot2 overloads + to build up a complex plot from simple pieces; this is non-associative because earlier layers are drawn underneath later layers (i.e. geom\_point() + geom\_smooth() does not yield the same plot as geom\_smooth() + geom\_point()).

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

lobstr::ast(1 + 2 + 3)

#> █─`+`

#> ├─█─`+`

#> │ ├─1

#> │ └─2

#> └─3

Copy

There are two exceptions: exponentiation and assignment.

lobstr::ast(2^2^3)

#> █─`^`

#> ├─2

#> └─█─`^`

#> ├─2

#> └─3

lobstr::ast(x <- y <- z)

#> █─`<-`

#> ├─x

#> └─█─`<-`

#> ├─y

#> └─z

Copy

### **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():

x1 <- "y <- x + 10"

x1

#> [1] "y <- x + 10"

is.call(x1)

#> [1] FALSE

x2 <- rlang::parse\_expr(x1)

x2

#> y <- x + 10

is.call(x2)

#> [1] TRUE

Copy

parse\_expr() always returns a single expression. If you have multiple expression separated by ; or \n, you’ll need to use rlang::parse\_exprs(). It returns a list of expressions:

x3 <- "a <- 1; a + 1"

rlang::parse\_exprs(x3)

#> [[1]]

#> a <- 1

#>

#> [[2]]

#> a + 1

Copy

If you find yourself working with strings containing code very frequently, you should reconsider your process. Read Chapter 19 and consider whether you can generate expressions using quasiquotation more safely.

The base equivalent to parse\_exprs() is parse(). It is a little harder to use because it’s specialised for parsing R code stored in files. You need to supply your string to the text argument and it returns an expression vector (Section 18.6.3). I recommend turning the output into a list:

as.list(parse(text = x1))

#> [[1]]

#> y <- x + 10

Copy

The inverse of parsing is **deparsing**: given an expression, you want the string that would generate it. This happens automatically when you print an expression, and you can get the string with rlang::expr\_text():

z <- expr(y <- x + 10)

expr\_text(z)

#> [1] "y <- x + 10"

Copy

Parsing and deparsing are not perfectly symmetric because parsing generates an abstract syntax tree. This means we lose backticks around ordinary names, comments, and whitespace:

cat(expr\_text(expr({

# This is a comment

x <- `x` + 1

})))

#> {

#> x <- x + 1

#> }

Copy

Be careful when using the base R equivalent, deparse(): it returns a character vector with one element for each line. Whenever you use it, remember that the length of the output might be greater than one, and plan accordingly.

### **18.4.4 Exercises**

1. R uses parentheses in two slightly different ways as illustrated by these two calls:
2. f((1))

`(`(1 + 1)

Copy

Compare and contrast the two uses by referencing the AST.

1. = can also be used in two ways. Construct a simple example that shows both uses.
2. Does -2^2 yield 4 or -4? Why?
3. What does !1 + !1 return? Why?
4. Why does x1 <- x2 <- x3 <- 0 work? Describe the two reasons.
5. Compare the ASTs of x + y %+% z and x ^ y %+% z. What have you learned about the precedence of custom infix functions?
6. What happens if you call parse\_expr() with a string that generates multiple expressions? e.g. parse\_expr("x + 1; y + 1")
7. What happens if you attempt to parse an invalid expression? e.g. "a +" or "f())".
8. deparse() produces vectors when the input is long. For example, the following call produces a vector of length two:
9. expr <- expr(g(a + b + c + d + e + f + g + h + i + j + k + l +
10. m + n + o + p + q + r + s + t + u + v + w + x + y + z))

deparse(expr)

Copy

What does expr\_text() do instead?

1. pairwise.t.test() assumes that deparse() always returns a length one character vector. Can you construct an input that violates this expectation? What happens?

## 18.5 Walking AST with recursive functions

To conclude the chapter I’m going to use everything you’ve learned about ASTs to solve more complicated problems. The inspiration comes from the base codetools package, which provides two interesting functions:

* findGlobals() locates all global variables used by a function. This can be useful if you want to check that your function doesn’t inadvertently rely on variables defined in their parent environment.
* checkUsage() checks for a range of common problems including unused local variables, unused parameters, and the use of partial argument matching.

Getting all of the details of these functions correct is fiddly, so we won’t fully develop the ideas. Instead we’ll focus on the big underlying idea: recursion on the AST. Recursive functions are a natural fit to tree-like data structures because a recursive function is made up of two parts that correspond to the two parts of the tree:

* 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.

To make this pattern easier to see, we’ll need two helper functions. First we define expr\_type() which will return “constant” for constant, “symbol” for symbols, “call”, for calls, “pairlist” for pairlists, and the “type” of anything else:

expr\_type <- function(x) {

if (rlang::is\_syntactic\_literal(x)) {

"constant"

} else if (is.symbol(x)) {

"symbol"

} else if (is.call(x)) {

"call"

} else if (is.pairlist(x)) {

"pairlist"

} else {

typeof(x)

}

}

expr\_type(expr("a"))

#> [1] "constant"

expr\_type(expr(x))

#> [1] "symbol"

expr\_type(expr(f(1, 2)))

#> [1] "call"

Copy

We’ll couple this with a wrapper around the switch function:

switch\_expr <- function(x, ...) {

switch(expr\_type(x),

...,

stop("Don't know how to handle type ", typeof(x), call. = FALSE)

)

}

Copy

With these two functions in hand, we can write a basic template for any function that walks the AST using switch() (Section 5.2.3):

recurse\_call <- function(x) {

switch\_expr(x,

# Base cases

symbol = ,

constant = ,

# Recursive cases

call = ,

pairlist =

)

}

Copy

Typically, solving the base case is easy, so we’ll do that first, then check the results. The recursive cases are trickier, and will often require some functional programming.

### **18.5.1 Finding F and T**

We’ll start with a function that determines whether another function uses the logical abbreviations T and F because using them is often considered to be poor coding practice. Our goal is to return TRUE if the input contains a logical abbreviation, and FALSE otherwise.

Let’s first find the type of T versus TRUE:

expr\_type(expr(TRUE))

#> [1] "constant"

expr\_type(expr(T))

#> [1] "symbol"

Copy

TRUE is parsed as a logical vector of length one, while T is parsed as a name. This tells us how to write our base cases for the recursive function: a constant is never a logical abbreviation, and a symbol is an abbreviation if it’s “F” or “T”:

logical\_abbr\_rec <- function(x) {

switch\_expr(x,

constant = FALSE,

symbol = as\_string(x) %in% c("F", "T")

)

}

logical\_abbr\_rec(expr(TRUE))

#> [1] FALSE

logical\_abbr\_rec(expr(T))

#> [1] TRUE

Copy

I’ve written logical\_abbr\_rec() function assuming that the input will be an expression as this will make the recursive operation simpler. However, when writing a recursive function it’s common to write a wrapper that provides defaults or makes the function a little easier to use. Here we’ll typically make a wrapper that quotes its input (we’ll learn more about that in the next chapter), so we don’t need to use expr() every time.

logical\_abbr <- function(x) {

logical\_abbr\_rec(enexpr(x))

}

logical\_abbr(T)

#> [1] TRUE

logical\_abbr(FALSE)

#> [1] FALSE

Copy

Next we need to implement the recursive cases. Here we want to do the same thing for calls and for pairlists: recursively apply the function to each subcomponent, and return TRUE if any subcomponent contains a logical abbreviation. This is made easy by purrr::some(), which iterates over a list and returns TRUE if the predicate function is true for any element.

logical\_abbr\_rec <- function(x) {

switch\_expr(x,

# Base cases

constant = FALSE,

symbol = as\_string(x) %in% c("F", "T"),

# Recursive cases

call = ,

pairlist = purrr::some(x, logical\_abbr\_rec)

)

}

logical\_abbr(mean(x, na.rm = T))

#> [1] TRUE

logical\_abbr(function(x, na.rm = T) FALSE)

#> [1] TRUE

Copy

### **18.5.2 Finding all variables created by assignment**

logical\_abbr() is relatively simple: it only returns a single TRUE or FALSE. The next task, listing all variables created by assignment, is a little more complicated. We’ll start simply, and then make the function progressively more rigorous.

We start by looking at the AST for assignment:

ast(x <- 10)

#> █─`<-`

#> ├─x

#> └─10

Copy

Assignment is a call object where the first element is the symbol <-, the second is the name of variable, and the third is the value to be assigned.

Next, we need to decide what data structure we’re going to use for the results. Here I think it will be easiest if we return a character vector. If we return symbols, we’ll need to use a list() and that makes things a little more complicated.

With that in hand we can start by implementing the base cases and providing a helpful wrapper around the recursive function. Here the base cases are straightforward because we know that neither a symbol nor a constant represents assignment.

find\_assign\_rec <- function(x) {

switch\_expr(x,

constant = ,

symbol = character()

)

}

find\_assign <- function(x) find\_assign\_rec(enexpr(x))

find\_assign("x")

#> character(0)

find\_assign(x)

#> character(0)

Copy

Next we implement the recursive cases. This is made easier by a function that should exist in purrr, but currently doesn’t. flat\_map\_chr() expects .f to return a character vector of arbitrary length, and flattens all results into a single character vector.

flat\_map\_chr <- function(.x, .f, ...) {

purrr::flatten\_chr(purrr::map(.x, .f, ...))

}

flat\_map\_chr(letters[1:3], ~ rep(., sample(3, 1)))

#> [1] "a" "b" "b" "b" "c" "c" "c"

Copy

The recursive case for pairlists is straightforward: we iterate over every element of the pairlist (i.e. each function argument) and combine the results. The case for calls is a little bit more complex: if this is a call to <- then we should return the second element of the call:

find\_assign\_rec <- function(x) {

switch\_expr(x,

# Base cases

constant = ,

symbol = character(),

# Recursive cases

pairlist = flat\_map\_chr(as.list(x), find\_assign\_rec),

call = {

if (is\_call(x, "<-")) {

as\_string(x[[2]])

} else {

flat\_map\_chr(as.list(x), find\_assign\_rec)

}

}

)

}

find\_assign(a <- 1)

#> [1] "a"

find\_assign({

a <- 1

{

b <- 2

}

})

#> [1] "a" "b"

Copy

Now we need to make our function more robust by coming up with examples intended to break it. What happens when we assign to the same variable multiple times?

find\_assign({

a <- 1

a <- 2

})

#> [1] "a" "a"

Copy

It’s easiest to fix this at the level of the wrapper function:

find\_assign <- function(x) unique(find\_assign\_rec(enexpr(x)))

find\_assign({

a <- 1

a <- 2

})

#> [1] "a"

Copy

What happens if we have nested calls to <-? Currently we only return the first. That’s because when <- occurs we immediately terminate recursion.

find\_assign({

a <- b <- c <- 1

})

#> [1] "a"

Copy

Instead we need to take a more rigorous approach. I think it’s best to keep the recursive function focused on the tree structure, so I’m going to extract out find\_assign\_call() into a separate function.

find\_assign\_call <- function(x) {

if (is\_call(x, "<-") && is\_symbol(x[[2]])) {

lhs <- as\_string(x[[2]])

children <- as.list(x)[-1]

} else {

lhs <- character()

children <- as.list(x)

}

c(lhs, flat\_map\_chr(children, find\_assign\_rec))

}

find\_assign\_rec <- function(x) {

switch\_expr(x,

# Base cases

constant = ,

symbol = character(),

# Recursive cases

pairlist = flat\_map\_chr(x, find\_assign\_rec),

call = find\_assign\_call(x)

)

}

find\_assign(a <- b <- c <- 1)

#> [1] "a" "b" "c"

find\_assign(system.time(x <- print(y <- 5)))

#> [1] "x" "y"

Copy

The complete version of this function is quite complicated, it’s important to remember we wrote it by working our way up by writing simple component parts.

## 18.6 Specialised data structures

There are two data structures and one special symbol that we need to cover for the sake of completeness. They are not usually important in practice.

### **18.6.1 Pairlists**

Pairlists are a remnant of R’s past and have been replaced by lists almost everywhere. The only place you are likely to see pairlists in R94 is when working with calls to the function function, as the formal arguments to a function are stored in a pairlist:

f <- expr(function(x, y = 10) x + y)

args <- f[[2]]

args

#> $x

#>

#>

#> $y

#> [1] 10

typeof(args)

#> [1] "pairlist"

Copy

Fortunately, whenever you encounter a pairlist, you can treat it just like a regular list:

pl <- pairlist(x = 1, y = 2)

length(pl)

#> [1] 2

pl$x

#> [1] 1

Copy

Behind the scenes pairlists are implemented using a different data structure, a linked list instead of an array. That makes subsetting a pairlist much slower than subsetting a list, but this has little practical impact.

### **18.6.2 Missing arguments**

The special symbol that needs a little extra discussion is the empty symbol, which is used to represent missing arguments (not missing values!). You only need to care about the missing symbol if you’re programmatically creating functions with missing arguments; we’ll come back to that in Section 19.4.3.

You can make an empty symbol with missing\_arg() (or expr()):

missing\_arg()

typeof(missing\_arg())

#> [1] "symbol"

Copy

An empty symbol doesn’t print anything, so you can check if you have one with rlang::is\_missing():

is\_missing(missing\_arg())

#> [1] TRUE

Copy

You’ll find them in the wild in function formals:

f <- expr(function(x, y = 10) x + y)

args <- f[[2]]

is\_missing(args[[1]])

#> [1] TRUE

Copy

This is particularly important for ... which is always associated with an empty symbol:

f <- expr(function(...) list(...))

args <- f[[2]]

is\_missing(args[[1]])

#> [1] TRUE

Copy

The empty symbol has a peculiar property: if you bind it to a variable, then access that variable, you will get an error:

m <- missing\_arg()

m

#> Error in eval(expr, envir, enclos): argument "m" is missing, with no default

Copy

But you won’t if you store it inside another data structure!

ms <- list(missing\_arg(), missing\_arg())

ms[[1]]

Copy

If you need to preserve the missingness of a variable, rlang::maybe\_missing() is often helpful. It allows you to refer to a potentially missing variable without triggering the error. See the documentation for use cases and more details.

### **18.6.3 Expression vectors**

Finally, we need to briefly discuss the expression vector. Expression vectors are only produced by two base functions: expression() and parse():

exp1 <- parse(text = c("

x <- 4

x

"))

exp2 <- expression(x <- 4, x)

typeof(exp1)

#> [1] "expression"

typeof(exp2)

#> [1] "expression"

exp1

#> expression(x <- 4, x)

exp2

#> expression(x <- 4, x)

Copy

Like calls and pairlists, expression vectors behave like lists:

length(exp1)

#> [1] 2

exp1[[1]]

#> x <- 4

In any case, the function uses rlang to manipulate expressions to make the formula. The inputs are expressions of *how* the predictors are used (i.e. log, inverse, etc.) and the degree. It captures the expression (without evaluating it), substitutes it into the polynomial function, then creates the formula:

make\_formula <- function(..., degree = 1) {

# Capture the expression so it is not evaluated

var\_expr <- exprs(...)

# Create a template expression

inv\_poly <- quote(poly(x = x, degree = degree))

# Use a wrapper around rlang::call\_modify to reverse the

# order of the arguments so that we can map over the

# predictor expressions

add\_args <- function(arg, call, ...)

call\_modify(call, x = arg, ...)

# Add the variables and the degree into the template

poly\_expr <- map(var\_expr, add\_args, call = inv\_poly, degree = degree)

# Convert to character

poly\_char <- map\_chr(poly\_expr, deparse)

# Convert to a formula

poly\_char <- paste(poly\_char, collapse = " + ")

as.formula(paste("PriceRecievedFarmers ~", poly\_char))

}

# For example:

make\_formula(1/USStockUse, log(WorldStockUse), degree = 3)

## PriceRecievedFarmers ~ poly(x = 1/USStockUse, degree = 3) + poly(x = log(WorldStockUse),

## degree = 3)

##

From here, we could use a bunch of mutate commands like the one shown above or write a slightly smaller for loop to work across the polynomial degrees. While the function above works well, the overall approach to working across models isn’t particularly satisfying.