Brian Lee Yung Rowe

Modeling Data With Functional Programming In R

Contents

List of Figures

List of Tables

1

Functions as a lingua franca

As a language paradigm, functional programming is not language-specific. Rather, functional programming is a theory for structuring programs based on function composition. In addition to function composition, functional programming is comprised of a (mostly) standard set of syntactic and semantic features. Many of these concepts originate from the lambda calculus, a mathematical framework for describing computation via functions. While each functional language supports a slightly different set of features, there is a minimal set of overlapping concepts that we can consider to form the basis of functional programming. This set consists of first-class functions, higher-order functions, and closures. Once these concepts are mastered, it is easy to identify and apply them in any language. In principle this is the same as learning the syntax of a new language: you begin by looking for the delimiter for statements, expressions, and blocks as well as how to create variables and call functions. These conceptual building blocks of a language act as a lingua franca irrespective of the specific language in question. The same is true within a language paradigm. Just like the semantics of classes and objects in an object-oriented language act as a lingua franca in the world of object-oriented programming, the function exclusively serves this purpose in a functional programming paradigm.

With just a few concepts the bulk of application design problems can be simply solved, particularly in data analysis. It is no secret that modeling data involves a lot of data processing. The steps involved typically include retrieving data, cleaning and normalizing data, persisting data for later user. And that's just to prepare for doing the real work, which involves analyzing the data, creating and validating models, and finally running them on new data. A curiosity of data analysis is that mental effort is split roughly 20 to 80 between data processing and modeling, but program code is often the opposite, with data processing taking up the majority of the lines. This is due to the steps involved and the inherently messy nature of data versus the pure and ideal world of mathematics and models. As data moves between libraries, components, and systems, the formats and data structures are often incompatible. Making these disparate pieces of software interoperable requires ad hoc data transformation to fit all the pieces together.

In this chapter, we'll see that first-class functions provide the necessary foundation to make it all possible. Higher-order functions provide the semantics for transforming data. The three primary constructions are iteration

(map), recursion (fold), and set operations (filter). Closures complete the picture by providing semantics for conforming function interfaces. Functions are not generally compatible so it is typical that the interface to one model is not immediately compatible with another function. Closures act as the glue between the data structure returned by one function and the expected format of another. With this core set of semantic constructs it is unnecessary to learn additional patterns and frameworks, meaning more time can be spent modeling and less on the dirty work of data transformation.

1.1 First-Class Functions

Structuring computer programs often begins by dividing programs into data structures and functions [1] that operate on the data. In an object-oriented programming (OOP) paradigm, data structures (objects) have associated methods that automatically passes the object as an operand to the function. In many OOP languages, functions can only exist as a part of a class. Despite the current popularity of this approach, this organizational structure is somewhat arbitrary. Consider that a Turing Machine operates on an infinite tape containing symbols that represent both instructions and data. [] The same is true at the hardware level, where data and instructions are ultimately both represented as a sequence of bits (data). Hence, at a fundamental level there is not much to distinguish functions from data. This indifference is also present in the lambda calculus, where Λ consists of lambda terms that are either lambda abstractions (functions) or variables. Consequently, functional languages treat everything as data. When functions are treated like variables, they are referred to as first-class entities. []

All functions are first-class in R. [] As a reminder, the syntax for function definition assigns a function to a variable. This is no different from assignment of a data strucure to a variable.

Example 1.1.1. Let's start by defining a univariate function that increments its argument. In mechanical terms, we are assigning a function to the variable named `increment`.

```
> increment ← function(x) x + 1
```

□

In Example 1.1.1 we've declared the variable `increment` and assigned a function as its value. This function can now be used like any other variable.

[1] Or procedures, sub-routines, etc.

Example 1.1.2. To convince yourself further that functions are treated like any other variable, we can add functions as elements of data structures. We can then extract the element, assign it to another variable, and finally call it.

```
> some.funs ← list(sum, increment)
> some.funs
[[1]]
function (..., na.rm = FALSE)  .Primitive("sum")

[[2]]
function (x)
x + 1

> f ← some.funs[[2]]
> f(4)
[1] 5
```

□

Functions can also be passed as arguments to other functions. This is common practice in R, typically with one of the `apply` functions. `apply` is used to iteratively process each element in an `array`, `matrix`, or `data.frame`. In two dimensions, an element is meant to be a row or column of the table-like structure. The function passed to `apply` is sequentially applied to each element in the data structure.

Example 1.1.3. Let's look at the classic `iris` dataset to illustrate how `apply` works. The signature of `apply` takes a data structure, the margin, which controls whether the iteration is along rows (columns), and a function that is applied to each row (column). Therefore, the function is treated as a value that is passed into `apply`. For this first example, we want to compute the mean for each attribute. This implies that each element is a column of the `iris` dataset, so the function `mean` is passed a vector in each iteration.

```
> apply(iris[,1:4], 2, mean)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

The purpose of `apply` is to provide the machinery around iteration, which is a generalized operation. We can verify this by replacing `mean` with another statistic, like standard deviation.

```
> apply(iris[,1:4], 2, sd)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
   0.8280661    0.4358663    1.7652982    0.7622377
```

Since `apply` uses the same machinery in both cases, the *structure* of the result is the same irrespective of our choice of statistic. The `apply` function is an example of a map operation, which is one of three primary higher-order functions. Map operations will be discussed in detail in Chapter **??**.

□

By using `apply`, the argument to the first-class function only needs to know how to process a single element instead of a set of elements. By separating the mechanics of iteration, the same function can be used for a single vector or multiple vectors without modification or ceremony. One condition is that any first-class function being passed to `apply` must have the same interface, since `apply` only has a single implementation. R provides a mechanism via the ellipsis to handle this situation, although a more idiomatically consistent functional approach is to use a closure, which is discussed in Section 1.3.

1.2 Higher-order functions

Treating functions as variables is a nice feature, but their value truly shines when coupled with higher-order functions. In general these functions provide the machinery for transforming data in a repeatable way. Since data analysis involves many individual records having the same general structure (e.g. vectors or table-like structures), it is beneficial to divide the data processing into a function that is responsible for manipulating a single record at a time, and a function that is responsible for the iteration over the records. The first function is a first-class function passed to the second function, which is a higher-order function. This is the separation of concerns that we saw in the previous section with `mean` and `apply`, respectively. We'll see in Chapter **??** that there are other types of machinery to manage alternate iterative processes.

Definition 1.2.1. A *higher-order function* is any function that takes a function as an operand, returns a function, or both.

If we didn't use higher-order functions, what would the `iris` code look like? Generally it requires initializing some data structure that represents the result, iterating over the original data structure and storing the values in a loop.

```
> y ← c()
> for (i in 1:4) {
+   y ← c(y, mean(iris[,i]))
+ }
> names(y) ← colnames(iris)[1:4]
> y
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

□

While there is nothing conceptually wrong with an imperative approach, notice how the mechanics of the iteration must be implemented explicitly. This means getting dirty with indices and initial values. It also means comingling the scopes of two distinct operations. If we want to preserve the labels, that also must be done explicitly. All these additional steps add complexity and make code more error prone. [2] The purpose of functions is to abstract and encapsulate general operations, and this applies equally to mathematical operations as well as to algorithmic operations. Functional programming gives us the tools to leverage both with ease.

Continuing with our hypothetical situation, suppose that the same generality achieved with `apply` and `mean` is desired. What options are available? A naive approach is to use a function with a string argument to generalize the statistic being used. In this case the whole loop is bundled up in a function and a large if-else expression or case statement is used to control flow.

```
function(data, statistic) {
  for (i in 1:ncol(data)) {
    if (statistic == 'mean')
      y ← c(y, mean(data[,i]))
    else if (statistic == 'sd')
      y ← c(y, sd(data[,i]))
    else
      ...
  }
}
```

This approach is clearly not generalized since the supported statistics are hard-coded in the function. A more viable approach is to use object-oriented techniques. In this approach a class needs to be defined that manages the dispatching for each different statistic. Figure 1.1 illustrates a typical approach using ReferenceClasses. The implementation is based on the Strategy design pattern [], which codifies the behavior in an abstract class (or interface) followed by a concrete implementation for each specific statistic.

Applying the Strategy pattern to the `iris` dataset involves replacing the explicit call to `mean` with a call to the instance method `execute` as seen in Figure 1.2. As a matter of convenience, the loop is encapulated inside a function.

The final result is obtained by first instantiating an instance of the `Mean` class and then passing it to our newly minted function.

```
> m ← Mean$new()
> aggregate_iris(iris, m)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

[2] To hammer this point home, even in this simple example the author made a syntax error when first implementing it.

```
AbstractStatistic ← setRefClass(
  Class="AbstractStatistic",
  fields=list(),
  methods=list(
    initialize=function(...) { callSuper(...) },
    execute=function(data, ...) {
      stop("Interface should not be called directly")
    }
  )
)

Mean ← setRefClass(
  Class="Mean",
  fields=list(),
  contains="AbstractStatistic",
  methods=list(
    initialize=function(...) { callSuper(...) },
    execute=function(data, ...) { mean(data, ...) }
  )
)
```

FIGURE 1.1: Generalization of `iris` statistics using object-oriented programming

```
aggregate_iris ← function(data, statistic) {
  result ← c()
  for (i in 1:4) {
    result ← c(result, statistic$execute(data[,i]))
  }
  names(result) ← colnames(data)[1:4]
  result
}
```

FIGURE 1.2: Using the Strategy design pattern

```
library(randomForest)
classify_iris ← function(x) {
  x$Sepal.LW ← x$Sepal.Length / x$Sepal.Width
  x$Petal.LW ← x$Petal.Length / x$Petal.Width
  x$SP.Length ← x$Sepal.Length / x$Petal.Length
  x$SP.Width ← x$Sepal.Width / x$Petal.Width
  randomForest(Species ~ ., x)
}
```

FIGURE 1.3: Iris classification with added features

Now the function is general in the way we want, but at what cost did we achieve this? Without first-class functions a simple iteration over a dataset becomes quite complicated. Not only is there a lot of ceremony required to use the function, it is harder to understand what the purpose of the code is. In general it is best to avoid complexity unless there is a tangible benefit from it. A good example of this is adding complexity to improve the performance of a function. When there is no tangible benefit from complexity, you are essentially paying to make your life more difficult.

1.2.1 Functions that take functions as arguments

Not all higher-order functions manage the machinery of iteration. A common pattern is to create a higher-order function to support arbitrary implementations of a specific operation in the function. Some common examples are how `NA`s are handled in a function or to support different models. The advantage of using first-class functions is that the possibilities are infinite, so the author of a function does not have to guess at which implementations to provide. Instead, a package can author focus on the *ideal* interface, knowing that a user of the package can use functional programming concepts to conform the data to the package interface.

Suppose we want to train a random forest to classify the iris dataset. The original dataset only has four features, so we will create a function that adds some more features and then executes the random forest, as in Figure 1.3. This function expects the standard `iris` dataset and appends additional columns to the `data.frame` prior to calling the random forest. Now suppose that the dataset contains `NA`s. How should these be handled? The simplest solution is to use a scalar value and replace all `NA`s with this value. However, this approach is clearly limited in its functionality. What if we wanted to provide arbitrary handling of `NA` values? Then it is better to pass a function with a defined interface to handle this. An example of such a function is in Figure 1.4.

When calling the function, we need to decide what the `na.fn` function should do. A first approach is to create a function that computes the mean of

```
classify_iris ← function(x, na.fn) {
  cols ← c('Sepal.Length', 'Sepal.Width',
           'Petal.Length', 'Petal.Width')
  x[,cols] ← apply(x[,cols], 2,
    function(z) ifelse(is.na(z), na.fn(z), z))

  x$Sepal.LW ← x$Sepal.Length / x$Sepal.Width
  x$Petal.LW ← x$Petal.Length / x$Petal.Width
  x$SP.Length ← x$Sepal.Length / x$Petal.Length
  x$SP.Width ← x$Sepal.Width / x$Petal.Width
  randomForest(Species ~ ., x)
}
```

FIGURE 1.4: Custom handling of `NA`s

the non-`NA` values. But first we need to modify the `iris` dataset by adding some `NA`s. We'll create a new `data.frame` instead of modifying `iris` directly.

```
> iris1 ← iris
> iris1[,1:4] ← apply(iris1[,1:4], 2, function(x) {
+   x[sample(length(x),10)] ← NA
+   x
+ })
```

This function randomly adds 10 `NA`s to each column of the dataset, which is sufficient for our purposes.

Computing the mean for each column should be as simple as calling `mean`. It would be nice to reference the function directly, but the default behavior is to return `NA` if any of the values are `NA`. Instead we need to wrap `mean` inside a function that calls `mean` and explicitly sets `na.rm=TRUE`.

```
> classify_iris(iris1, function(x) mean(x, na.rm=TRUE))

Call:
 randomForest(formula = Species ~ ., data = x)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 2

        OOB estimate of  error rate: 4%
Confusion matrix:
           setosa versicolor virginica class.error
setosa         50          0         0        0.00
versicolor      0         47         3        0.06
virginica       0          3        47        0.06
```

By following this approach `classify_iris` is now a higher-order function. Removing the function implementation that handles `NA`s and instead adding

```
classify_iris ← function(x, na.fn, model=randomForest) {
  cols ← c('Sepal.Length', 'Sepal.Width',
           'Petal.Length', 'Petal.Width')
  x[,cols] ← apply(x[,cols], 2,
    function(z) ifelse(is.na(z), na.fn(z), z))

  x$Sepal.LW ← x$Sepal.Length / x$Sepal.Width
  x$Petal.LW ← x$Petal.Length / x$Petal.Width
  x$SP.Length ← x$Sepal.Length / x$Petal.Length
  x$SP.Width ← x$Sepal.Width / x$Petal.Width
  model(Species ~ ., x)
}
```

FIGURE 1.5: A generalized classification function for `iris` data

it to the function signature points to the separation of concerns that was mentioned earlier. In essence, the function `classify_iris` becomes focused on data management, while the logic inherent in the model is isolated in the function argument. Continuing this pattern, it is possible to abstract the model call as well, supporting any arbitrary model. This version is implemented in Figure 1.5. Notice that we set the default model to the original `randomForest` function. This has the effect of preserving past behavior despite adding functionality, which is generally advisable when refactoring code.

Suppose we want to evaluate the performance of a support vector machine. This is as trivial as specifying the model parameter with the `ksvm` function reference.

```
> library(kernlab)
> classify_iris(iris1, function(x) mean(x, na.rm=TRUE), ksvm)
Using automatic sigma estimation (sigest) for RBF or laplace
    kernel
Support Vector Machine object of class "ksvm"

SV type: C-svc  (classification)
 parameter : cost C = 1

Gaussian Radial Basis kernel function.
 Hyperparameter : sigma =  0.292382695546997

Number of Support Vectors : 61

Objective Function Value : -4.3539 -5.0384 -24.2008
Training error : 0.026667
```

Notice how our original function is now exclusively responsible for data management and wiring, while the actual model logic has been separated from this function. This separation of concerns is similar to how `apply` sep-

```
setup_svm ← function(...) {
  function(formula, data) {
    ksvm(formula, data, ...)
  }
}
```

FIGURE 1.6: A higher-order function used to configure a SVM

arates general data management machinery from specific application logic. Achieving an explicit separation of concerns is one of the key benefits of functional programming. Compare this to object-oriented programming where class hierarchies must be created to support the generalization we accomplished above with a simple change to the function signature.

1.2.2 Functions that return functions

Instead of calling a function directly, sometimes it is better to first call a function that returns a function and then call the resulting function. The rationale is that by having two functions, it is easier to understand the purpose of both via explicit separation of concerns. The outer function acts as a constructor of sorts, initializing certain values of the returned function. This keeps the interface of the returned function clean and concise. Often this is required to make two interfaces compatible.

Continuing the iris example from the previous section, what if you want to tune some parameters of the classification model? Notice that we've codified a de facto model interface in our classifier function: `function(formula, data)`. It isn't possible to support all the tuning parameters for every individual model, as it would make the interface extremely cluttered, while also having finite utility. This is similar to the hard-coding conundrum on page 5. Instead it's better to write a function that knows how to call a model with our specific parameters, thus preserving the interface defined by `classify_iris`. This requires calling a function that returns a function with the correct signature. Hence the higher-order function is responsible for matching the function signatures. Let's say we want to swap out the kernel in our SVM. The native function call is

```
ksvm(Species ~ ., iris1, kernel='besseldot').
```

Without modifying `classify_iris`, let's create a new function `setup_svm` that knows how to configure our model, as shown in Figure 1.6. By using the ellipsis argument, our new function is generic enough that it supports any arbitrary parameter that we may want to tune. This function is then called like this.

```
model ← setup_svm(kernel='besseldot')
classify_iris(iris1, function(x) mean(x, na.rm=TRUE), model)
```

This same pattern happens often with the `apply` functions when a parameter must be set in a function. In fact, this is what we did when defining the function for handling `NA`s. The only difference is that we defined the function inline, so there was no need to create an explicit higher-order function. Any generic higher-order function is defining a de facto interface for its function argument. To ensure compatibility with this expected signature, a higher-order function can be used.

1.3 Closures

When a higher-order function returns a function, that function is typically a closure. What differentiates a closure from a basic function is that a closure has an associated external scope bound to the function. This means that variables can be referenced outside the function scope and accessed as immutable values. The significance is that the closure provides a way to track interstitial state strictly within the context of the function in question. In pure terms these variables are immutable, such that the values are guaranteed to be constant within the closure. This property is essential for deterministic behavior and local reasoning of a program.

Example 1.3.1. To see how a closure works let's first define a simple function that references a variable in the global environment. This is not a recommended practice as it is unsafe, but for pedagogical purposes it is particularly illuminating.

```
x ← -5
f ← function() {
  x ← abs(x)
  function(y) y - x
}
```

When calling this function, note that the original `x` is not modified.

```
> g ← f()
> g(6)
[1] 1
> x
[1] -5
```

□

Thanks to lexical scoping, where the scope of a function is determined lexically from the source code [], closures are easy to create in R. This means that variables defined in an outer scope can be referenced within a function body. Doing so will make a copy of the variable in the closure environment.

This implies that if the original variable changes value later in a program, it will not affect the closure. In the above example, changing `x` to another value will not affect the result of `g(6)`.

Example 1.3.2. Avoiding the use of the global environment, let's instead create a higher-order function that defines the same variable `x`, which is referenced in the returned function.

```
f ← function(x) {
  x ← abs(x)
  function(y) y - x
}
```

By debugging the closure, we can explicitly verify that the variables `x` and `y` are in separate environments.

```
> g ← f(-5)
> debug(g)
> g(7)
debugging in: g(7)
debug: y - x
Browse[2]> x
[1] 5
Browse[2]> ls()
[1] "y"
Browse[2]> ls(envir=parent.env(environment()))
[1] "x"
```

□

1.3.1 Interface compatibility

We now know that the `setup_svm` function defined in the previous section returns a closure. The resultant function references a single variable outside of its scope, which happens to be the ellipsis argument. This is easily verifiable by manually inspecting the definition of the closure.

```
> model
function(formula, data) {
  ksvm(formula, data, ...)
}
<environment: 0x7f985b8c1a38>
```

Notice that the ellipsis is present in the body of the function but the value is unspecified. To inspect its contents, a similar procedure as in Example 1.3.2 is required.

```
> ls(envir=parent.env(environment(model)))
FIX THIS
```

```
classify_iris ← function(x, method='rf') {
  if (method == "rf")
    randomForest(Species ~ ., x)
  else if (method == "svm")
    ksvm(Species ~ ., x)
}
```

Any time a higher-order function specifies a function signature that is different from the signature of the function we want to pass to it, a closure is used to bridge the gap in signatures. The key is that the signature of the closure must always match the expected signature, while the higher-order function generating the closure can be arbitrary. Below is a simple algorithm for extending a function signature to become a higher-order function.

1. Add a first-class function argument to the signature
2. Replace explicit function call with argument
3. Create a new higher-order function to return a closure that calls the replaced function
4. Add arguments as necessary to higher-order function

This simple procedure works for any function that you want to turn into a higher-order function. This is the process followed when refactoring `classify_iris` to use an arbitrary model.

Let's again pretend that we are not using functional programming. In this hypothetical scenario, we no longer have access to first-class functions. What are alternative ways to implement the same behavior? There are numerous approaches to this, but none of them are as simple and straight-forward as using a first-class function.

One approach is to follow the approach of the standard `optim` function, where a `method` character argument is used to describe the optimization method. Actual dispatching is then performed explicitly via a conditional block, which is the same naive approach we started with when we wanted to generalize the use of a statistic on the iris data. Clearly this approach holds constant the number of methods possible to use. In some circumstances where the methods are truly finite this is acceptable. In other cases where the set of methods are not known a priori it can be limiting.

One argument supporting a conditional block is that if each model expects slightly different data, it might be easier to use the conditional block to control the data transformations. This actually is the approach `optim` takes, which unfortunately also shows how easy it is to create complicated code. Functions provide explicit boundaries between blocks. This constraint forces a separation of concerns, which makes it easier to reuse functions and modify them later. In Chapter **??** we'll see how to effectively swap out models while conforming to their unique interfaces.

```
AbstractModel ← setRefClass('AbstractModel',
  fields=c("data"),
  methods=list(
    classify=function(formula) stop("not implemented")
  ))

SvmModel ← setRefClass('SvmModel',
  contains="AbstractModel",
  methods=list(
    classify=function(formula) ksvm(formula, data)
  ))

RandomForestModel ← setRefClass('RandomForestModel',
  contains="AbstractModel",
  methods=list(
    classify=function(formula) randomForest(formula, data)
  ))
```

FIGURE 1.7: A class hierarchy for classification models

Rather than using an explicit conditional block one might use dynamic function calls via `do.call`.

```
do.call(method, list(Species ~ ., data))
```

This approach supports any arbitrary function to be called, which is similar to using a first-class function directly. The danger here is that a syntax error will result in an execution error. Using a first-class function is safer since the object is guaranteed to be callable.

Using ReferenceClasses provide a more traditionally object-oriented approach but still incurs much software design overhead to accomplish a simple task. The strategy here is to create a class hierarchy to represent the different types of models. Then a method is called to execute the model for the given data. This design is similar to the design we used in Figure 1.1 for generalizing the choice of statistic.

```
model ← RandomForestModel(data=iris1)
model$classify(Species ~ .)
```

This approach is typical of an object-oriented paradigm. Notice how much additional work is required to implement this style of programming. In general, object-oriented design patterns stem from the need to create object structures that emulate the behavior of first-class functions. Consider that with the inclusion of first-class functions in a language, the need for design patterns all but disappears.

1.3.2 State representation

In certain cases shared mutable state is appropriate to use, particularly for representing external resources. These resources are often singletons in the physical world (or in the operating system environment), so modeling them as a single shared object with state makes sense. Connections are an obvious example, where a resource is opened, read, and finally closed. Here a file descriptor represents the state of the file and must be managed accordingly.

Object-oriented paradigms are often heralded for their ability to manage state. In an object-oriented paradigm a class represents a generic file, and an instance of the class is a specific file. This file object can then be opened, read, and closed. The power of the object-oriented approach is that all resources, variables, and operations associated with the file are encapsulated within the class definition. The challenge is that each resource and method returns its own instances of other classes. Knowing when to stop modeling the class hierarchy is one of the hardest problems in designing object-oriented systems as one must balance reusability with ease of use. Highly granular class libraries are good for reuse, but it leads to exceptionally verbose implementations that are difficult to learn. In Java, there are distinct classes for files, connections, streams, and buffers. Loading a file in Java requires interacting with objects from each of these classes, which means understanding how a file system is modeled along with their individual APIs, in addition to the implicit state machines embedded within the class. An example of this are connections that must be closed after opening. When resources aren't properly closed, it can lead to memory leaks as well as running out of operating system resources. Despite all this granularity, you still have to manually manage the actual resources being modeled. The saving grace is that all of the machinery for managing a resource can be encapsulated in a single class, which limits the hunt for documentation. On the other hand, languages that favor monolithic classes (like Objective-C) are also difficult to learn because so many permutations exist for performing an operation that it isn't immediately obvious which one to use.

So the benefit of object-oriented programming comes at the cost of complexity. Not surprisingly, functional programming provides a liberating alternative to the tyranny of all-encompassing class hierarchies. Rather than attempting to optimize an interface for the most common use cases, functional programming interfaces are restricted in quantity. Since closures are so easy to create (and their resources managed efficiently), it is often trivial to conform two interfaces together on an ad hoc basis. This approach preserves a simple and clear interface for functions while avoiding the slippery slope of optimal interface design.

In terms of state management, closures can provide the same encapsulation as a class can. The key difference is that creating a closure does not require a lot of ceremony and is therefore efficient in implementation. Closures can be created ad hoc as an anonymous function or more formally as the

```
using ← function(resource, handler, exit=close) {
  tryCatch(handler(resource),
    error=stop, finally=function() exit(resource))
}
```

FIGURE 1.8: A resource management function

return value of a higher-order function. Any resources defined in the closure can be automatically garbage collected once all references to the closure are gone. The result is a cleaner code base since there are fewer formal type/class definitions.

A functional approach to managing resources involves, not surprisingly, a higher-order function. We will implement a function that is inspired by the `with` keyword in Python. A `with` statement automatically manages resources within the scope of a block. When the end of the block is encountered or an error is encountered, the specified resource is automatically closed. [3] Since R defines `with` as a technique to access objects as environments, we'll call our version `using`.

The function is used like

```
z ← using(file(path), readLines)
```

The value of a function like this is that any errors in the handler will automatically close the resource.

```
z ← using(file(path), function(x) { log('a'); readLines(x) })
```

Example 1.3.3. Another scenario is managing graphical parameters. Sometimes a function needs to change these parameters to display a custom plot. A good citizen will ensure that the original parameters are restored once the function exits. A typical implementation looks like

```
plot_handler ← function(x) {
  opar ← par(mfrow=c(2,2), ...)
  on.exit(par(opar))

  # Do stuff
}
```

The use of `on.exit` is required to properly account for errors that may arise in the function. Without this inlcusion, the parameters will not be restored properly if an error is encountered. This approach works well but is easily overlooked. The same can be accomplished with `using`. [4]

```
using(par(mfrow=c(2,2), ...), plot_handler, par)
```

[3] In Python, `with` operates on a callable object that has a `__enter__` and `__exit__` function defined.

[4] The removal of the `par` lines in `plot_handler` is implied.

```
setup_using ← function(resource, exit=close) {
  function(handler, destroy=FALSE) {
    if (destroy) return(exit(resource))
    tryCatch(handler(resource),
      error=function(e) { exit(resource); stop(e) })
  }
}
```

FIGURE 1.9: Using a closure to manage external resources

Notice how this approach cleanly separates the mechanics of managing the state of the graphics environment from the visualization code.

□

In the above cases no closure is required because the handler operation is effectively atomic. What if the resource must stay open for an indefinite period of time? Here a closure can be used to manage the resource. While the above technique is useful for a fixed set of operations, it doesn't work well for arbitrary operations in disconnected control sequences. Taking a cue from Javascript, we can overload a function with multiple behaviors to achieve the desired behavior. Named parameters makes this a simple and safe exercise as seen in Figure 1.9. The general method is to define the default operation as the primary interface for the signature. Other operations are then controlled by optional arguments to the function.

Our new function `using_fn` is the second type of higher-order function since it is returning a function instead of having a function as an argument. Working with this function involves naming the returned function and calling this in lieu of `using`.

```
> cat("line 1\n", file="example.data")
> using.resource ← setup_using(file("example.data"))
> using.resource(readLines)
[1] "line 1"

> cat("line 2\n", file="example.data")
> using.resource(readLines)
[1] "line 2"

> using.resource(destroy=TRUE)
> unlink("example.data")
```

The recurring theme of separation of concerns is yet again the main benefit. By clearly thinking about what is model logic versus general software machinery provides an opportunity to cleanly implement models according to the mathematical sequence of function composition. Once this distinction in code purpose is made, it also becomes clear that much of the data management machinery is general and can be easily reused at a level of sophistication

```
setup_svm ← function(...) {
  errors ← c()
  function(formula, data, get.error=FALSE) {
    if (get.error) return(errors)

    result ← ksvm(formula, data, ...)
    errors ↞ c(errors, result@error)
    result
  }
}
```

FIGURE 1.10: Evaluating the numerical stability of SVMs

that exceeds granular functions. This is because we have encoded a process workflow within a higher-order function and closure as opposed to a single operation.

1.3.3 Mutable state

In the previous section, the state being managed was static. Once a file resource is opened, the resultant connection doesn't change state. Other situations have dynamic state that requires updating this state. Typically variables retained in a closure are immutable, but with the special ↞ operator, it is possible to change the value of a variable.

Continuing the iris example, machine learning methods are often faced with the question of numerical stability. How do we know whether the solution from one iteration is representative of the model or is an outlier? Some algorithms have built-in stability tests [], whereas others require manual evaluation of the stability. We can answer this question by examining the distribution of the model results over multiple iterations.

To measure the classification error of the SVM we used earlier, we can modify the `setup_svm` function to track the errors over multiple iterations, as shown in Figure 1.10. By using a closure it is possible to preserve the previous function signature, allowing us to use this new function wherever the old one was used. Hence, we can perform as many iterations as we like in the same way as running any other model.

```
> do.svm ← setup_svm()
> z ← sapply(1:500, function(x) do.svm(Species ~ ., iris))
```

As an added benefit, extracting the accumulated error is a repeatable process, so it is easy to work with the data. The histogram in Figure 1.11 is one way to view the classfication error.

```
> hist(do.svm(get.error=TRUE), main='SVM class. error',
+   xlab='Error')
```

FIGURE 1.11: Classification error for SVM

```
seq.gen ← function(start)
{
  value ← start - 1
  function() {
    value ↞ value + 1
    return(value)
  }
}
```

FIGURE 1.12: A simple generator function

The significance of this approach cannot be stressed enough. By using functional programming techniques, we've added specialized functionality for measuring error that preserves existing function signatures. This means that you can measure the error of a numerical method from any external package with little effort.

If we wanted to compare the performance of different kernels, this can be done with the current function.

```
> bessel.svm ← setup_svm(kernel='besseldot')
> tanh.svm ← setup_svm(kernel='tanhdot')
```

Since each closure manages its own state, it is easy to compare the error from each kernel. All this was accomplished with just a few lines of code. As a thought-experiment, compare this to what is required in a non-functional approach.

Some care does need to used with the ↞ operator. This is due to the semantics involved: until a matching variable is found, the operator will continue to access enclosing environments until the global environment is found. At this point if no variable is found, one is created. Careless usage can therefore result in variables being created in the global environment.

Exercise 1.1. Rewrite `setup_svm` to be a general function that measures numerical stability for any model.

1.3.4 Generators

Building on the functionality of state management, closures can also be used to implement generator functions. Like all closures, generators have local state that is mutable but only accessible to the particular function. Generators are useful when a variable acts as a shared singleton, which requires explicit management of its internal state.

Example 1.3.4. A sequence generator is used for creating monotonically increasing IDs as seen in Figure 1.12. The returned closure does two things: increment a counter and return its value. Thus a monotonic sequence is produced by calling the function successive times.

```
> g ← seq.gen(5)
> g()
[1] 5
> g()
[1] 6
```

□

In general, generators are a convenient way to localize the side effects of a function. This abstraction also means that multiple instances of a generator can be created without worrying about namespace collisions. We'll see in Chapter **??** how to use the generator concept for implementing finite state machines as well as Markov Chains for simulation.

1.4 Functions In Mathematics

The idea of first-class functions is not some radical idea spawned in the age of computing. Disciplines like traditional calculus actually makes extensive use of this concept. The derivative and integral both take functions as operands so conceptually these functions are being treated as data. It is also clear from the notation that a function is treated as a first-class entity. Take for example the polynomial function $f(x) = ax^3 - bx + 4$. When we take the derivative of this function the Liebniz notation hints at the concept: $\frac{d}{dx}f$. Hence, f is a first-class function passed to the derivative operator, which just happens to be a higher-order function. Conceptually this is no different from writing the derivative as a function $d(f) \equiv \frac{d}{dx}f$ for the univariate case.

First-class functions make an appearance in other parts of mathematics as well. Finding such cases often involves transforming operators into functions. This is a legal transformation, as we can prove that there is exactly one unique function that represents the operator.

Theorem 1.4.1. *Given an operator* $\circ$, $\exists$ *exactly one function* $f : X \times X \to X$ *such that* $f(x, y) = x \circ y$, $\forall x, y \in X$.

Proof. We use a proof by contradiction to show this is true. Let $\circ$ be an operator $\circ : X \times X \to X$. Define $f_1(x, y) = x \circ y$. Suppose there is another function $f_2 \neq f_1$ where $f_2(x, y) = x \circ y$. But $x \circ y = f_1(x, y)$, so $f_2 = f_1$. □

For the time being what is important is looking at the operand of the derivative. Here the function f is being transformed into the function $f'(x)$. When writing functional programs it is useful to remember that this equivalence exists. In fact all operators in R are indeed functions. It is only the syntax that differentiates the two. However, any operator can be called using function notation.

Example 1.4.1.

```
> `+`(4,5)
[1] 9
```

□

Example 1.4.2. The summation operator illustrates the equivalence between operators and functions. Suppose we want to take the sum of f applied to each element of a vector. The expression is written mathematically as $\sum_i ax_i^3 - bx_i + 4$, which is really just fancy notation for function application. With some basic symbolic manipulation, we can illustrate this point. We'll define a function `sum` as follows.

$$\begin{aligned}\text{sum}(\vec{x}, f) &= f(x_1) + f(x_2) + \cdots + f(x_n) \\ &= \sum_i f(x_i) \\ &= \sum_i ax_i^3 - bx_i + 4\end{aligned}$$

This example shows that the summation operator is really just a function that takes a vector and a first-class function as arguments. We'll see in Chapter **??** that $\sum$ and $\prod$ are examples of fold operations. The lesson here is that there shouldn't be any bias in terms of using functions as operands to other functions.

□

Transforms are another class of mathematical entities that operate on functions. Consider the Laplace, Z, or Fourier transform. Each of these transforms takes an expression or function as an argument and returns a new function with change of variable. Hence, these transforms are a special type of higher-order function.

Example 1.4.3. Recall the definition of the Laplace transform, which is $\mathcal{L}\{f(t\} = \int_0^\infty e^{-st} f(t)\, dt$. The notation is clearly indicates that $f(t)$ is the operand to the function $\mathcal{L}$.

□

When thinking of a transform $y = f(x)$ we often discuss the inverse $g(y)$ as well, which has the property of reversing the operation of f. In math terms we have $x = g(f(x))$, for all x in the domain of f. This is true of transforms and is reflected in the relationship between the derivative and the integral. While not all programming functions have inverses, thinking about functions as being analytic or as transforms helps to prime your thinking. Leveraging the tools of mathematical analysis for the act of model implementation facilitates reasoning about the program code. As we progress further in the book, numerous examples of this will be highlighted.

1.5 A lambda calculus primer

The astute reader will likely notice the gradual arc towards mathematical reasoning in this chapter. The goal is to highlight the shared semantic structures in the notation of mathematics and functional programming. This overlap yields remarkable clarity in thinking about models and data. Functional programming is possible thanks to the conceptual foundation laid by the lambda calculus. Invented by Alonso Church, the lambda calculus defined computable functions to answer the so-called *Entscheidungsproblem* [3].

As an outgrowth of this task, much of the mathematical landscape like numbers and algebra were defined using this system. Defining the whole of mathematics is out of scope for this book; what we care about is the ability to define higher-order functions and closures. We also need a notation for anonymous functions, which the lambda calculus provides us. Formalizing these concepts will enable us to perform symbolic transformations so that we can fully reason about our functional programs. To start we establish some basics regarding the lambda calculus. Our focus will be the untyped lambda calculus as it is readily compatible with a dynamically typed lanugage like R. In the untyped lambda calculus only variables v_1, v_2, etc., the symbols λ and ., and parentheses are allowed. The set of all lambda expressions is further defined inductively [3].

Definition 1.5.1. The set of all *lambda expressions* Λ is defined by

(a) If x is a variable, then $x \in \Lambda$.

(b) If x is a variable and $M \in \Lambda$, then $\lambda x.M \in \Lambda$.

(c) If $M, N \in \Lambda$, then $(MN) \in \Lambda$.

This definition tells us that variables, functions, and the result of functions are all lambda terms. Typically uppercase letters are used to denote lambda terms while lowercase letters represent simple variables. So long as the mathematical constructions we create satisfy this definition, then we can leverage the lambda calculus in our analysis.

In the lambda calculus all functions are anonymous and first-class. Anonymous functions are therefore synonomous with lambda abstractions. A named function is thus nothing more than a lambda abstraction bound to a variable. These are denoted as in conventional mathematics. Hence $f(x) \equiv f = \lambda x$. This equivalence can be extended to any function with an arbitrary number of arguments. For function application we note that $(\lambda x.M)[x := N] \equiv f(N)$, where $f(x) = M$.

In terms of symbolic notation, equality of expressions is denoted by $=$. For recursive functions, it can be difficult to distinguish between symbolic equality of an expression and equality based on a recursive application of a function. In these cases $\rightarrow$ is used instead.

1.5.1 Reducible expressions

Lambda terms can be transformed under certain conditions, referred to as either a conversion or a reduction. As one might expect a conversion changes the syntax but not the form of an expression. The most fundamental conversion is an α-conversion, which is commonly known as a change of variable. Any reference to a given variable can be replaced with a different variable without changing the meaning of the expression. For example $\lambda x.x * 2 = \lambda y.y * 2$. Reduction is the process of simplifying an expression using rewrite rules. The goal is to achieve a so-called normal form that cannot be reduced further. Applying arithmetic operations can be considered reductions since repeated application of the operations eventually yields a scalar value, which is a terminal value.

Most algebraic operations can be viewed as a conversion or reduction. Consider a polynomial that requires factorization in order to simplify it as

$$\begin{aligned} f(x) &= \frac{x^2 + 3x + 10}{x - 2} \\ &= \frac{(x-2)(x+5)}{x-2} \\ &= x + 5. \end{aligned}$$

We can think of the first step as a conversion since neither form is clearly simpler than the other. Hence these two forms could be interchanged for an indeterminate number of times without ever arriving at a normal form. Once the term $x - 2$ is factored out, then it is clear that a reduction operation can take place, eliminating this term from both the numerator and denominator.

Operators like the factorial are also governed by rewrite rules. For example $5! = 5 * 4! = 20 * 3! = 60 * 2! = 120$. We can look at the factorial as either an operator or a function. When viewed as a function we must describe the mechanics of function abstraction, which provides a syntax for defining a function. Furthermore we must consider function application and the role of variable substitution within an expression.

Function application is governed by β-reduction, which tells us how to apply a function M to a lambda term N, or MN. If M is a function of variable x, then this application is denoted $MN = M[x := N]$. Suppose that $M = \lambda x.X$, then $(\lambda x.X)N = (\lambda x.X)[x := N] = X[x := N]$. The latter expression can be read as X evaluated with x replaced with N. Referring back to the factorial function, we can define this as a lambda abstraction $\lambda x.x!$ and apply it to the parameter 5 as $(\lambda x.x!)[x := 5] = x![x := 5] = 5! = 120$.

The final conversion is known as η-conversion and is often characterized in terms of extentionality. I tend to think of η-conversion more as a proclamation of independence between lambda terms. In other words a function application has no effect on an embedded lambda term if there is no dependency on the argument. Recognizing the applicability of η-conversion can often lead to greater modularity and simplification of an algorithm.

Definition 1.5.2. Given $\lambda x.Mx$ where M is a lamba abstraction, if x is not free in M then the η-conversion of the lambda abstraction is $\lambda x.Mx \leftrightarrow_\eta M$.

The standard conversions and reductions provide mechanisms to reduce lambda terms into normal forms. Sometimes it is useful to go in the opposite direction and add structure to a lambda term. This is analogous to unconventional factorizations of polynomials to achieve a particular goal.

Proposition 1.5.3. *An equivalent higher-order function can be constructed from any existing function. This step is an intermediate one on the way to creating a closure as discussed in Section 1.3.*
$\lambda w.X = \lambda v.(\lambda w.X)[w := v]$

Proof. Given $\lambda w.X$ and $n \in \Lambda$. The left-hand side reduces by standard function application to $(\lambda w.X)n = X[w := n]$. The right-hand side reduces to

$$\lambda v.(\lambda w.X)[w := v] = \lambda v.X[w := v]$$
$$(\lambda v.X[w := v])[v := n] = X[w := n].$$

□

Example 1.5.1. Let $f = \lambda x.x + 1$. Then

$$\begin{aligned} f &= \lambda x.f(x) \\ &= \text{function(x) f(x)} \end{aligned}$$

The last line shows the equivalent syntax in R. Let's look at a concrete example in the interpreter.

```
> f ← function(x) x + 1
> f(5) == (function(x) f(x))(5)
[1] TRUE
```

This example shows the equivalence between the two constructions for a single value. The lambda calculus gives us the tools to prove that the equivalence holds for all values of `x`.

□

Standard mathematical notation can be included as lambda expressions, since numbers are variables and operators are functions. When including function application using traditional notation, care must be taken with the variable naming.

Example 1.5.2. In example **??** we used the same variable `x` for both functions. To avoid confusion, it is wise to apply an η-conversion to one of the function definitions.

$$\begin{aligned} f &= \lambda y.f(y) \\ &= \text{function(y) f(y)} \end{aligned}$$

□

The lambda calculus also supports multivariate functions via Currying [3]. Additional arguments are appended after the λ symbol as $\lambda xyz.X$.

Example 1.5.3. The `ksvm` function has multiple arguments, but let's assume that it is defined as $ksvm = \lambda formula\ data.M$. Then

$$\begin{aligned} \text{ksvm} &= \lambda formula\ data.ksvm(formula, data) \\ &= \text{function(formula, data) ksvm(formula, data)} \end{aligned}$$

□

A closure can now be constructed by applying the proposition to a multivariate function. This means that at a fundamental level, we can create a closure from an existing function and be certain that its behavior is unchanged.

Example 1.5.4. As a shorthand I will often denote a set of function arguments as a sequence. Hence for $W = \langle x, y, z \rangle$, $\lambda xyz.X = \lambda W.X$. To illustrate the creation of a closure along with this point, let's rewrite a version of `setup_svm` in lambda calculus notation.

$$\begin{aligned} \text{setup\_svm} &= \lambda W.\lambda formula\ data.ksvm(formula, data, W) \\ &= \text{function(...) function(formula, data) ksvm(formula, data, ...)} \end{aligned}$$

□

There is no requirement that a lambda abstraction must only reference variables specified in the function signature. Those that are present in the argument list are known as *bound*, while those not present are *free*. The relationship between free and bound variables within lambda abstractions form an important basis for understanding how to transform functional programs into equivalent forms. We will explore this in depth in Chapter **??**.

1.6 Church numerals

To understand how functions can be treated as data, the Church numerals provide a good example of how this works in practice. Church numerals represent a technique for encoding the natural numbers based on the lambda calculus. The insight is that any structure or process that is countable can be mapped to the cardinal numbers. In traditional mathematics, set theory is often used to show how the fundamental entities of mathematics can be used to prove the existence of natural numbers. Doing so reduces the axioms

that mathematics must rely on. Once natural numbers are defined it is easy to derive the integers followed by the rational numbers.

As a product of the lambda calculus, Church numerals are simply functions. Yet these functions can be operated on just like the natural numbers.

Definition 1.6.1. Church numerals are based on the definition of function composition, which is defined inductively [3]. Let $F, M \in \Lambda$ and $n \in \mathbb{N}$. Then $F^0(M) = M$ and $F^{n+1}(M) = F(F^n(M))$. The Church numerals are then defined as $\mathbf{c}_n \equiv \lambda fx.f^n(x)$. For example,

$\mathbf{c}_0 \equiv \lambda f.\lambda x.x$
$\mathbf{c}_1 \equiv \lambda f.\lambda x.f(x)$
$\mathbf{c}_3 \equiv \lambda f.\lambda x.f(f(f(x)))$

The syntax in R is equivalent despite the nominal syntactic differences.

```
C0 ← function(f) function(x) x
C1 ← function(f) function(x) f(x)
C3 ← function(f) function(x) f(f(f(x)))
```

By definition these functions represent scalar values and thus can be considered data. This means that the functions can be operands to other functions and indeed this is the case with addition. The addition operator is derived from the successor function, `SUCC`, which simply increments a Church numeral.

```
SUCC ← function(n) function(f) function(x) f(n(f)(x))
```

When `SUCC` is applied to a Church numeral, we see the remarkable behavior that the Church numeral is acting as data and also a function simultaneously. Let's apply the rules of the lambda calculus to see this more clearly. [5]

$$\begin{aligned} SUCC\ \mathbf{c}_2 &= SUCC[n := \mathbf{c}_2] \\ &= (\lambda n.\lambda f.\lambda x f((nf)x)))[n := 2] \\ &= \lambda f.\lambda x.f((\lambda g.\lambda y.g(g(y)))[g := f])x) \\ &= \lambda f.\lambda x.f(\lambda y.f(f(y)))[y := x]) \\ &= \lambda f.\lambda x.f(f(f(x))) \\ &= \mathbf{c}_3 \end{aligned}$$

In the case of Church numerals, both the numerals and the operators are higher-order functions. One would think that in a computing environment, it would be easier to verify the operation performed by the `SUCC` function. Due to lazy evaluation, it is actually somewhat opaque and requires an additional step to verify the complete computation. Let's see what happens in the interpreter when we apply `C2` to the `SUCC` function.

[5] Note that an α-conversion is applied to 2 for clarity's sake.

```
TO_NAT ← function(x) x + 1
```

FIGURE 1.13: Mapping Church numerals to natural numbers

```
PLUS ← function(m) function(n) m(SUCC)(n)
```

FIGURE 1.14: Addition for Church numerals

```
> SUCC(C2)
function(f) function(x) f(n(f)(x))
<environment: 0x7fe521d88740>
```

Since the return value is an unevaluated function, the arguments of the `SUCC` function are unevaluated. This ties into the idea of a closure, where the bound variables are unevaluated. Hence to see a value that is meaningful to humans requires evaluating the function completely. To do so requires creating a function to map the function composition to the natural numbers, as in Figure 1.13. Then to verify the value of the Church numeral, simply call every function with the appropriate argument.

```
> SUCC(C2)(TO_NAT)(0)
[1] 3
```

Finishing up, we can now define addition in terms of the successor function, as seen in Figure **??**. The mechanics of the operation can be rather cryptic, so let's break down how it works. Let's evaluate $PLUS\ \mathbf{c}_2\ \mathbf{c}_3$. The equivalent definition of `PLUS` is $PLUS = \lambda m\ n.m\ SUCC\ n$. Recall that the definition of $SUCC = \lambda n\ f\ x.f(n\ f\ x)$. The first part of the definition applies $SUCC$ to $\mathbf{c}_2$, which gives

$$\begin{aligned}\mathbf{c}_2\ SUCC &= (\lambda f\ x.f^2(x))[f := SUCC] \\ &= (\lambda f\ x.f(f(x)))[f := SUCC] \\ &= \lambda x.SUCC(SUCC(x)).\end{aligned}$$

Now apply $\mathbf{c}_3$ to this intermediate result, yielding

$$\begin{aligned}(\lambda x.SUCC(SUCC(x)))\ \mathbf{c}_3 &= SUCC(SUCC(\mathbf{c}_3)) \\ &= \mathbf{c}_5.\end{aligned}$$

In R, we can evaluate the sum using the same technique as before.

```
> PLUS(C2)(C3)(TO_NAT)(0)
[1] 5
```

The *PLUS* function illustrates how a Church numeral can be used as both data and as a function within the same equation. Initially $\mathbf{c}_2$ is used

as a function that operates on *SUCC*. Then $\mathbf{c}_3$ is used as an argument to the resulting function, which yields a function representing a value. The value of Church numerals is that it provides a framework for thinking about function composition. We will revisit them later in terms of the fold concept.

1.7 Summary

The benefits of functional programming are legion, and this chapter highlighted many of these benefits. The primary theme surrounded the idea that functions can be treated like any another piece of data. Alternatively, data can represent both values and functions. We covered the core concepts of functional programming from first-class functions, to higher-order functions, to closures. This simple toolkit can be applied to virtually any situation offering a clean separation of concerns between model logic, data management logic, and application logic. The end result is a modular program with a clear delineation between reusable pieces of data logic and model-specific ad hoc pieces.

We also explored the mathematical connection with functional programming concepts, which will facilitate model development in subsequent chapters. The brief introduction to the lambda calculus provides a formal framework for understanding function transforms within code, which can simplify model implementation as well as provide insights into the model itself.

2

Vector Mechanics

Much of the elegance of R stems from the interplay between functional programming semantics and native handling of vectors. This unique combination makes R unrivaled in its ease of use for data analysis. We discussed the importance of functional programming in the previous chapter. In this chapter we explore vectors and what it means for vectors to be a fundamental data type. On the surface this simply means that the semantics of vectors are built into the core of the language. This is similar to how objects are fundamental to an object-oriented language, where every variable is an object [**?**], [**?**]. In R, all primitives are vectors, meaning that any scalar value is really a vector of length one. This implies that vector operations can be performed on scalars without issue.

Coming from other languages, where scalars are primitives and arrays are separate data types, such a concept can seem alien. Yet this approach is very powerful in its expressiveness and again can be traced to its mathematical roots. In abstract algebra, the operators + and $*$ are polymorphic over rings, in the sense that they operate consistently over any ring. This is how we naturally think about addition. Regardless of whether the operands are scalars or vectors, we expect addition to operate in a semantically consistent way.

Recall that vector addition is performed when the dimensions of the operands are the same. This rule applies to vectors of arbitrary length. By definition these operations are performed on an element-wise basis. Given $\vec{x} = \begin{bmatrix} 3 \\ 5 \\ 7 \end{bmatrix}$ and $\vec{y} = \begin{bmatrix} 4 \\ 4 \\ 8 \end{bmatrix}$, the sum is naturally $\vec{x} + \vec{y} = \begin{bmatrix} 7 \\ 9 \\ 15 \end{bmatrix}$. When working with matrices, we expect addition to again operate in this semantically consistent manner. R takes this concept and generalizes it to support arbitrary operators and functions.

The challenge faced by model builders is that mathematics is often silent about the implementation of an operation. Mathematical symbols and operators convey meaning regarding what an operator does but not how it does it. Even something as trivial as $\sqrt{x}$ can be articulated simply despite its implementation being difficult to describe. This dichotomy between what an operation does and how to compute it becomes even more pronounced with matrix operations, where concepts like the inverse or eigenvalues are again easy to describe but require much effort to explain how they are computed.

This emphasis on what an operation does but not how is the essence of a declarative programming style.

Definition 2.0.1. *Declarative programming* structures programs based on the logic of a computation without prescribing control flow. [**?**]

Declarative notation offers numerous benefits. By focusing on what an operation does, its fundamental behavior and properties can be communicated concisely. The details of implementation can obfuscate the intention of a proof or algorithm, which detracts from the meaning and ultimately the elegance surrouding an operation. The common theme is that mathematics, functional programming, and native support of vectors all promote a declarative style. In contrast, many programming languages are imperative in nature. Fundamental operators like addition only work for a handful of primitive data types, such as integers, doubles, and sometimes characters. Adding two vectors together therefore requires explicitly looping through both data structures to access each individual primitive element. This makes perfect sense when thinking about data structures from a hardware perspective but less so from a software or mathematical perspective. These programs have an imperative style, which reflects the underlying semantics of a Turing Machine.

Definition 2.0.2. *Imperative programming* structures programs based on a sequence of statements that manipulate program state. [**?**]

Imperative algorithms result in very explicit statements and program structure. This is borne out of necessity since the program state is managed explicitly. Hence a conceptually simple operation like vector addtion all of a sudden requires initializing variables and incrementing indices to access individual elements of vectors. As an example, let `x ←c(3,5,7)` and `y ←c(4,4,8)`. To compute the sum a new vector `z` must be initialized, followed by assignment of each individual element:

```
z ← c()
for (i in 1:length(x))
  z[i] ← x[i] + y[i]
```

By its very nature, data analysis is full of vectors and matrix-like structures. Imagine having to operate on vectors like this any time an ad hoc analysis is desired. The verbosity and tedium required rivals that of manually computing the inverse of a matrix. Surely there is a more efficient way to work with these data structures. Indeed, experienced R users would never sum two variables as shown above. They would instead take advantage of *vectorization* to perform the operation as a single statement:

```
z ← x + y
```

Again, the key is that since all primitive types are vectors in R, the polymorphic properties of mathematical operators are preserved over scalars,

vectors, and matrices. When functions (operators) operate on vectors and adhere to specific properties, they are called *vectorized* functions (operators). In situations where you have to write your own operators and functions, we'll see that higher-order functions provide the necessary semantic foundation that resembles this concept of vectorization. Some approaches for iterating over vectors are so common that they form a canonical set. The canonical higher-order functions include *map*, *fold*, and *filter*,

ch is the essence of Chapters 3 and 4, respectively. We won't discuss the filter operation much since it is functionally equivalent to set comprehensions.

One might argue that this declarative style is nothing more than syntactic sugar. In a language like R that is ultimately built atop C, this holds more than an ounce of truth. Yet through the use of functional programming concepts, the declarative style goes beyond syntax, shaping not only the program structure but also how one thinks about the relationship between the code and the underlying mathematics. This chapter develops the mathematical properties of vectors. The motivation for a formal approach is to leverage the duality between programs and proofs. By focusing explicitly on revealing this duality, something remarkable happens: translating mathematical ideas into algorithms becomes a direct mapping. It becomes almost trivial to implement mathematical ideas, and we see this in much of R already. For instance, the covariance between two random variables x and y is a one line expression: `(x - mean(x))%*%(y - mean(y))/(length(x)-1)`. As an added benefit, it becomes easy to reasson about the program code, which leads to fewer implementation errors. It can also lead to insights into the mathematical transformations. Achieving this nirvana requires codifying the syntactic equivalences between R and mathematical notation. Since functional programming and native vectors facilitate declarative style, such a mapping is natural. The first step in this journey is analyzing the vector.

2.1 Vectors as a polymorphic data type

Until now we've talked about vectors without really defining what they are. As a data structure, vectors can represent numerous mathematical concepts. Beyond simply their mathematical namesake, the vector type can also represent n-tuples (sequences), sets, random variables, etc. Distinguishing between one mathematical type and another is a matter of context and also convention within a program. Irrespective of the mathematical type being modeled, vectors as a data structure are obviously governed by a single set of properties and rules. Properties like atomicity, and rules for concatenation and coercion are all important considerations when modeling a mathematical type with a vector. These implementation details of the data structure are important as they put specific constraints on the mathematical structures being modeled.

For example, elements of a vector must all be of the same atomic type. The set of atomic types is denoted $\mathbb{T}$, and any vectors created with elements not in $\mathbb{T}$ will result in a list. The use of a single data type to represent numerous mathematical entities also affords us a certain flexibility when working with data. At our convenience, it is possible to mentally switch the mathematical context and apply set operations as necessary to a structure that heretofore was considered a sequence. These transformations are legal so long as we understand the implications of the data structure for each mathematical entity.

A natural question arises from this polymorphism: what should the vector data type represent by default? In every day modeling, it is likely unnecessary to make a stance, but for the sake of notational consistency, it is a worthwhile question. Taking a pragmatic approach, an ordered n-tuple, or sequence, is conceptually similar and makes a good candidate. Sequences provide useful semantics for extracting elements from them, which is natural in data analysis. They also abide by the structural rules of the vector data type, as elements of a sequence form a domain T^n, where $T \in \mathbb{T}$. Compare this to sets that do not require type constraints.

Definition 2.1.1. A *vector* is an ordered collection of elements equivalent to a finite sequence. A vector x with n elements is denoted as $c(x_1, x_2, ..., x_n) \equiv \langle x_1, x_2, ..., x_n \rangle$. Vectors are governed by the following rules:

(a) Elements of a vector are indexed by a subset of the positive integers $\mathbb{N}_n = 1, 2, ..., n$.

(b) For vector x and $k \in \mathbb{N}_n$, `x[k]` $\equiv x_k$ is the k-th element of x.

(c) The *length* of a vector is its cardinality and is denoted `length(x)` $\equiv |x|$.

(d) $x_k \in T$, $\forall x_k \in x$ where $T \in \mathbb{T}$.

(e) Two vectors x, y where $|x| = n = |y|$ are *equal* if and only if $x_k = y_k \forall k \in \mathbb{N}_n$.

Example 2.1.1. Definition 2.1.1 provides the foundation for working with vectors. A brief example illustrates how the syntactic equivalences work. Given $x = \langle 3, 5, 7 \rangle$, the first element is $x_1 = 3 =$ `x[1]`. The overall length is $|x| = 3 =$ `length(x)`.

□

Example 2.1.2. As described in Definition 2.1.1e, comparison of vectors is an element-wise operation. This is not limited to equality and is generally true of the relations defined in R. Hence to determine whether two vectors are equal requires comparing their individual elements. The equality operator is vectorized, so testing vector equality results in a logical of the same length as the operands. Let $x = \langle 2, 3, 4 \rangle$ and $y = 2 : 4$. Then

```
> x == y
[1] TRUE TRUE TRUE
```

To determine if the vectors are equal as opposed to the individual elements thus requires the use of another vectorized function `all`, which is equivalent to $\bigwedge$ in logic. The equivalent mathematical expression is $\bigwedge_i x_i = y_i$, which translates to the following in R:

```
> all(x == y)
[1] TRUE
```

□

2.1.1 Vector construction

Like a sequence, a vector is created by a function defined on $\mathbb{N}_n$. The simplest vectors can be created explicitly using the concatenation function `c`.

```
> x ← c(2,3,4,5)
> x
[1] 2 3 4 5
```

Vectors can also be created using the sequence function, which is defined as `seq(i, j, by=m)` $\equiv \langle i, i+m, i+2m, ..., j \rangle$, where $m \in \mathbb{R}$. If the sequence is an integer range, then `i:j` $\equiv$ `seq(i, j, 1)` can be used as a shorthand. As we will see later, this notation is the same as the shorthand for extracting subsequences.

Example 2.1.3. In short, there are three ways to create integer sequences. The least flexible but the most concise is the `:` notation. For static sequences, using the concatenation function explicitly is simplest but can be tedious, while `seq` provides a balance of flexibility and ease of use.

```
> seq(2,5) == 2:5
[1] TRUE TRUE TRUE TRUE

> 2:5 == c(2,3,4,5)
[1] TRUE TRUE TRUE TRUE
```

□

Example 2.1.4. One thing to consider when using the `:` operator is that its precedence is very high. Hence certain constructions will result in behavior that is unexpected. This happens a lot when the end of the sequence is defined by a variable. Suppose we want to specify indices from 2:4 given a vector of length 5.

```
> x ← rnorm(5)
> 2:length(x)-1
[1] 1 2 3 4
```

Due to precedence rules `2:n` is evaluated first and then 1 subtracted from the resulting vector [1]. What we want instead is the sequence `2:4`, which requires explicit parentheses.

```
> 2:(length(x)-1)
[1] 2 3 4
```

□

Traditional mathematical vectors are simply ordered n-tuples. When vectors need to explicitly represent traditional vectors, either tuple or matrix notation is used. For example, $\vec{x} = \left[3,5,7\right] \equiv \begin{bmatrix} 3 \\ 5 \\ 7 \end{bmatrix}$. In either construction, we'll refer to mathematical vectors exclusively via square brackets. These vectors are by default column vectors, which can be confusing to newcomers, since the printed output appears horizontal.

```
> c(3,5,7)
[1] 3 5 7
```

However, performing a transpose on a vector will confirm that the original was indeed a column vector.

```
> t(c(3,5,7))
     [,1] [,2] [,3]
[1,]    3    5    7
```

2.1.2 Scalars

Since all atomic types are vectors, we can surmise that scalars are simply vectors of length one. From this perspective, all vector operations equally apply to scalars.

Definition 2.1.2. A vector x is a *scalar* if and only if $|x| = 1$. It follows that $x = x_1$.

Most properties are semantically conistent when applied to scalars, while some unique properties also emerge. The most noteworthy is that scalars are invariant under indexing. Mathematically this amounts to saying that a set containing a single element is equal to the element itself, which is nonsensical. [2] Yet this is perfectly legal in R and is a valuable contribution to the syntactic efficiency of the language.

Example 2.1.5. It's easier to see the invariance over indexing by plugging some values into the interpreter.

[1] The precedence rules of R can often be suprising, such as `-2^2 == -4`.
[2] Except possibly for an infinite recursive set.

```
> x ← 3
> x == x[1]
[1] TRUE
```

Compare this to a non-scalar, which results in a vector.

```
> y ← c(3,5,7)
> y == y[1]
[1]  TRUE FALSE FALSE
```

In this latter comparison, each element of y is being compared to y_1, which is why the result is a vector.

□

The moral is that scalars in a vectorized language can yield idiosyncractic behavior that must be accounted for. Another such complication involves logical operators. Newcomers are typically confused between `&` (`|`) and `&&` (`||`). The difference is that the single character form is vectorized while the dual version is not.

Example 2.1.6. Let's compare two common sequences. Define $x = \langle 1, 2, 3, 5, 7, 11, 13, 17, 19\rangle$ and $y = \langle 1, 1, 2, 3, 5, 8, 13, 21, 34\rangle$ where $|x| = n = |y|$. When is y greater than x? Set builder notation describes the result as $\{y_i | y_i > x_i, \forall i \in \mathbb{N}_n\}$. In this case, we only care about the comparison, as opposed to the set of elements that satisfy the condition.

```
> y > x
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
```

The result illustrates that the conditional operators are vectorized as claimed earlier. Let's also look at the occurrences when `y` is even.

```
> y %% 2 == 0
[1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
```

Now if we want to find the elements that satisfy both conditions, we use the single `&` operator, since this is the one that is vectorized.

```
> y %% 2 == 0 & y > x
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

If instead we use the `&&` operator, a scalar is returned. This is because the double operator is designed for conditional blocks that traditionally operate on scalar boolean values. When vectors are passed to these operators, only the first result is used, which can be surprising.

```
> y %% 2 == 0 && y > x
[1] FALSE
```

□

Understanding the cardinatlity of operands and the cardinality expectations of an operation is central to writing proper code in R. We'll see in Chapter 3 that a certain class of vectorized functions preserve cardinality while Chapter **??** explores another class that reduces the cardinality to one.

2.1.3 Atomic types

Earlier in the chapter, we casually mentioned the set of atomic types, known as $\mathbb{T}$. Certain types are known as atomic because the data are stored in contiguous cells of memory. This arrangement is for efficiency reasons and results in subtle differences in behavior from the non-atomic types.

Definition 2.1.3. Let $\mathbb{T} = \{logical, integer, double, complex, character, raw\}$ be the set of atomic types. [7]

Example 2.1.7. As stated in Definition 2.1.1, vectors comprise elements of a single atomic type. Some valid constructions of vectors are presented below.

```
> c(3, 5, 8)
[1] 3 5 8

> c("a", "b", "aa")
[1] "a"  "b"  "aa"

> c(TRUE, NA, FALSE, TRUE)
[1]  TRUE    NA FALSE  TRUE
```

□

Note that we are careful in our use of the term atomic versus primitive. The set of primitive types in R contains the atomic types and includes other types such as `environment`, `closure`, and `list`. [3] We will refer to some of these primitive types, but most of the discussion will be centered around the atomic types. Furthermore, notice that `matrix` is not an atomic type, let alone a type at all. The reason is that matrices are simply special vectors with an associated `class` that distinguishes them as matrices instead of vectors. The same is true of factors, which are integers having a class attribute of `factor` and other metadata. Hence, the type of a factor is still `integer`.

Example 2.1.8. Constructing a vector with values that are not of the atomic types will result in a list.

```
> typeof(c(list(1,1), 3, 5, 8))
[1] "list"

> typeof(c(3, 5, function(x) x))
[1] "list"
```

□

2.1.4 Coercion

The previous example showed that concatenation will change its output type depending on the input types. Similarly, if a vector is created with values of

[3] See `?typeof` for more details.

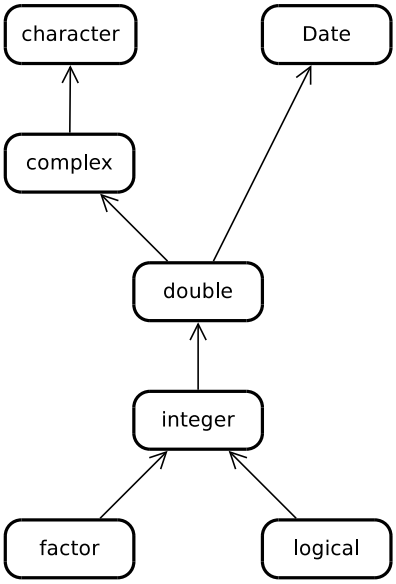


FIGURE 2.1: Partial type coercion hierarchy for concatenation

different types, the values will be coerced into a type that is compatible with all elements. This coercion process ensures that the vector has at most one atomic type. Figure 2.1 displays a partial compatibility hierarchy between types. What this means is that with sufficient variation of types in the input, the output type will likely be character.

Example 2.1.9. Creating a vector from an integer, a double, and a logical results in a double since integers and logicals can be represented as doubles.

```
> c(3, 4.5, FALSE)
[1] 3.0 4.5 0.0
```

However, mixing a number of types with a character results in a character vector.

```
> c(6, "a", TRUE, 5+2i)
[1] "6"    "a"    "TRUE" "5+2i"
```

□

The coercion process occurs even when the input does not appear to be a primitive type. The key is to look at the type and not the class of a variable. A variable's *class* is user-modifiable and determines the representation of a value. On the other hand the type is not user-modifiable and determines the actual storage mode of the variable. Consequently, vectors and matrices support only a single type, whereas lists (and structures derived from them) support one type per element.

Example 2.1.10. A `Date` object is actually a double with a class of `Date`. In this case, the integer is coerced to a `Date` instead of a double.

```
> c(Sys.Date(), 5)
[1] "2014-05-08" "1970-01-06"
```

However, this coercion is determined strictly by the type of the first argument.

```
> c(5, Sys.Date(), Sys.Date()+1)
[1]     5 16198 16199
```

□

Since type coercion is not commutative, it is generally inadvisable to rely on coercion rules to transform data. It is better to epxlicitly convert types as the intention of the operation becomes explicit.

2.1.5 Concatenation

What we've called the vector constructor is actually a special case of the concatenation function. Combining arbitrary vectors together is known as concatenation. When all the elements of `c` were scalars, it was convenient to call this vector construction. However, since scalars are simply vectors of length one, conceatenation operates on arbitrary vectors and is the mechanism for adding elements to a vector. This function differs from languages that provide explicit semantics for arrays again thanks to vectorization. Since each argument to `c` is a vector, new vectors can be created quickly without the need for looping.

Definition 2.1.4. Given vectors $x = \langle x_1, x_2, ..., x_m \rangle$, $y = \langle y_1, y_2, ..., y_n \rangle$ the *concatenation* of x and y is the ordered union of x and y. It is denoted

$$\begin{aligned} c(x, y) &= c(c(x_1, x_2, ..., x_m),\ c(y_1, y_2, ..., y_n)) \\ &= c(x_1, x_2, ..., x_m, y_1, y_2, ..., y_n) \\ &= \langle x_1, x_2, ..., x_m, y_1, y_2, ..., y_n \rangle \end{aligned}$$

Example 2.1.11. Since all primitives are vectors, concatenation operates on any number of vectors with arbitrary length.

```
> c(1:3, 5, 7, c(11,13,17,19))
[1]  1  2  3  5  7 11 13 17 19
```

□

Proposition 2.1.5. *The definition of concatenation implies the following statements* $\forall$ *vectors* x, y *of the same type.*

(a) Concatenation is idempotent: $c^{(m)}(x) = x\ \forall m \in \mathbb{N}$.

(b) Concatenation preserves ordering. It follows that conctenation is not commutative: $c(x, y) \neq c(y, x)$.

(c) Concatenation is associative: $c(x, c(y, z)) = c(c(x, y), z)$. *Furthermore,* $c(x, c(y, z)) = c(x, y, z)$.

(d) Length is linear: $|c(x, y, \cdots)| = |x| + |y| + \cdots$.

Proof. Let $x = \langle x_1, x_2, ..., x_m \rangle$, $y = \langle y_1, y_2, ..., y_n \rangle$, where $m, n \in \mathbb{N}$. Then

(a) For $m = 1$, $c(x) = x$. Assume that $c^{(m)}(x) = x$. Then

$$\begin{aligned} c^{(m+1)}(x) &= c(c^{(m)}(x)) \\ &= c(x) \\ &= x \end{aligned}$$

.

(b) $c(x, y) = \langle x_1, x_2, ..., x_m, y_1, y_2, ..., y_n \rangle$ On the other hand, $c(y, x) = \langle y_1, y_2, ..., y_n, x_1, x_2, ..., x_m \rangle$ Therefore $c(x, y) \neq c(y, x)$.

(c) Let x, y defined as in (a). Let $z = \langle z_1, z_2, ..., z_p \rangle$. Then

$$\begin{aligned} c(x, c(y, z)) &= c(x, c(c(y_1, y_2, ...y_n), c(z_1, z_2, ..., z_p))) \\ &= c(x, c(y_1, y_2, ...y_n, z_1, z_2, ..., z_p)) \\ &= c(c(x_1, x_2, ..., x_m), c(y_1, y_2, ...y_n, z_1, z_2, ..., z_p)) \\ &= c(x_1, x_2, ..., x_m, y_1, y_2, ...y_n, z_1, z_2, ..., z_p) \\ &= \langle x_1, x_2, ..., x_m, y_1, y_2, ..., y_n, z_1, z_2, ..., z_p \rangle \end{aligned}$$

And

$$\begin{aligned} c(c(x, y), z)) &= c(c(c(x_1, x_2, ..., x_m), c(y_1, y_2, ...y_n)), z) \\ &= c(c(x_1, x_2, ..., x_m, y_1, y_2, ...y_n), z) \\ &= c(c(x_1, x_2, ..., x_m, y_1, y_2, ...y_n), c(z_1, z_2, ..., z_p)) \\ &= c(x_1, x_2, ..., x_m, y_1, y_2, ...y_n, z_1, z_2, ..., z_p) \\ &= \langle x_1, x_2, ..., x_m, y_1, y_2, ..., y_n, z_1, z_2, ..., z_p \rangle \qquad = c(x, y, z) \end{aligned}$$

Therefore $c(x, c(y, z)) = c(c(x, y), z)$.

(d) Define vectors $x^1, x^2, \cdots, x^n$. By definition, $|c(x)| = |x|$. Assume that $|c(x^1, x^2, \cdots, x^n) = |x^1| + |x^2| + \cdots |x^n|$. Now let $a = c(x^1, x^2, \cdots, x^n)$ and $b = c(a, x^{n+1})$. Then $|b| = |c(a, x^{n+1})| = |a| + |x^{n+1}|$.

□

Example 2.1.12. Idempotency is an important concept in both mathematics and computer science. Situations may arise where a sequence of concatenations is performed. Knowing the result in advance is useful but also knowing how a result appeared is often critical. For example, `c(c(c(x)))== x`, which is thematically similar to $((x^1)^1)^1 = x$. Rewriting exponentiation as a function, the general form is exposed. Let $f(x, y) = x^y$. Then $f(f(f(x, 1), 1), 1) = x$. The function f is idempotent when $y = 1$. In a similar vein, concatenation on a single argument is idempotent. Therefore, if $c(x, y) = x$, we know that $y = \emptyset$.

□

Example 2.1.13. Commutativity is useful for algebraic operations, but in the case of concatenation it would render it useless. Preserving order is an essential part of the process and provides a reliable way to access vector elements based on their ordinality. The diabetes dataset contains a number of features. We want to conduct a time-series analysis on feature 33 (regular insulin dose), which requires the data to be ordered based on date and time. Assume that `x` contains data related to feature 33 and is already ordered. As new data is collected, it is stored in a new variable `y`. To perform the analysis on the union of `x` and `y`, we want `c(x,y)`, which is clearly different from `c(y,x)`.

□

Example 2.1.14. We can see that associativity is a natural extension of the preservation of ordering. Hence `c(c(c(1:3, 5), 7), 11, 13, 17, 19)== c(1:3, c(5, c(7, c(11, 13, 17, 19))))`.

□

Example 2.1.15. The linearity of length extends to concatenation of an arbitrary number of arguments. Based on example 2.1.14, we can see how this works in practice. Given $c(1:3, c(5, c(7, c(11, 13, 17, 19))))$, the length of the resulting vector is computed as

$$\begin{aligned}|c(1:3, c(5, c(7, c(11, 13, 17, 19))))| &= |c(1:3, 5, 7, c(11, 13, 17, 19)|\\ &= |1:3| + |5| + |7| + |c(11, 13, 17, 19)|\\ &= 3 + 1 + 1 + 4\\ &= 9.\end{aligned}$$

We made use of property 2.1.5c to flatten the nested vector structure, which makes it easier to compute the length. We could have just as easily computed the length moving left-to-right in the evaluation.

$$\begin{aligned}|c(1:3, c(5, c(7, c(11, 13, 17, 19))))| &= |1:3| + |c(5, 7, c(11, 13, 17, 19)|\\ &= 3 + |5| + |c(7, c(11, 13, 17, 19)|\\ &= 3 + 1 + |7| + |c(11, 13, 17, 19)|\\ &= 3 + 1 + 1 + 4\\ &= 9\end{aligned}$$

```
load_diabetes ← function(bad=c(2,27,29,40), base='.') {
  load.one ← function(i) {
    path ← sprintf('%s/data-%02i',base,i)
    flog.info("Loading file %s", path)
    o ← read.delim(path, header=FALSE,
      colClasses=c('character','character','numeric','numeric'
          ))
    colnames(o) ← c('date','time','feature','value')
    o$date ← as.Date(o$date, format='%m-%d-%Y')
    o
  }
  idx ← 1:70
  lapply(idx[-bad], load.one)
}
```

FIGURE 2.2: Loading the diabetes dataset

□

When vectors are concatenated only the ordering is preserved in the resultant vector but not the actual ordinal positions (excepting the first argument). Since length is a linear operator we can deduce the ordinal positions of the resulting vector given the arguments to concatenation. The following proposition tells us how to map indices on pre-concatenated vectors into the corresponding indices of the concatenated result.

Definition 2.1.6. Let x be a vector where $|x| = n$ and let $i \in \mathbb{N}_n$. The *ordinal position* of x_i in x is $ord(x_i) = i$.

Proposition 2.1.7. *Let* $x^1, x^2, \cdots, x^n$ *be vectors of arbitrary length, where* $|n| > 1$. *For* $y = c(x^1, x^2, \cdots, x^n)$, $ord(y_i) = \sum_{j=1}^{k} |x^j| + ord(x_m^{k+1})$, *where* $k = argmax_k \sum_{j=1}^{k} |x^j| < |y|$ *and* $m = |y| - \sum_{j=1}^{k} |x^j|$.

Proof. Pending □

Example 2.1.16. Taking Example 2.1.11 as a cue, the corresponding R code provides an anecdotal confirmation of the proposition.

```
> x ← c(1,2,3)
> y ← c(11, 13, 17)
> a ← c(x, 5, 7, y)
> a[length(x) + 1 + 1 + 2] == y[2]
[1] TRUE
```

□

Example 2.1.17. When data are collected and aggregated, it is difficult to keep track of indices. At times certain records may have been identified as important and their ordinal positions recorded. Parallelization is such a case where it may be desirable to process a dataset in chunks. Other times, data is provided in chunks and must be joined at a later time. This is the case with the `diabetes` dataset **[?]**, which is loaded via `load_diabetes` as defined in Figure 2.2. [4] Let's look at all the values associated with pre-breakfast glucose (feature 58). When conducting the analysis on a single feature, it might be nice to consolidate the values into a single vector.

```
> y ← lapply(diabetes, function(x) x$value[x$feature==58])
> z ← do.call(c, y)
```

Suppose we want to know from which datasets the tails of the distribution come from. We can determine this by using Proposition 2.1.7 to associate the lengths between the vectors. First let's extract the ordinal positions of some outliers.

```
> outliers ← which(z > 400)
```

The file that contains each outlier is simply the first file where the cumulative length is greater than the ordinal of the outlier.

```
> lengths ← cumsum(sapply(y, length))
> sapply(outliers, function(o) which.max(lengths >= o))
[1] 12 12 16 16 20 20 20 20 49 60 64
```

This approach balances ease of use of an ad hoc analysis while simultaneously preserving information necessary to locate outliers in raw data files.

□

In Section 2.1.3 we saw how the type of a vector can change when a vector is constructed with arguments of mixed types. This is an exceptional situation that results in an operation that is not closed with respect to some of the operands. However, the atomic types are closed under concatenation when all arguments of the same type. To see how this works, we need to think of vectors as sets.

Proposition 2.1.8. *Atomic vectors are closed under concatenation up to k.* [5] *Let x, y be vectors where $typeof(x) = typeof(y) = T \in \mathbb{T}$. Then $typeof(c(x, y)) \in T$.*

Proof. Consider x and y as sets containing elements of T. The power set $P(T)$ is the set of all possible sets of type T. Define $a = c(x_1, x_2, \cdots, x_m, y_1, y_2, \cdots, y_n) \in P(T)$. Since $c(x, y) = c(x_1, x_2, \cdots, x_m, y_1, y_2, \cdots, y_n)$, $c(x, y) = a$ and $c(x, y) \in P(T)$. □

[4] Some of these files have bad data and consequently are omitted from the analysis.
[5] The maximum length is due to finite machine limits.

The significance of Proposition 2.1.8 is that it acts as a null hypothesis of sorts. When this property doesn't hold, we know that the set of argument types contained more than one type.

Example 2.1.18. It turns out that the diabetes dataset contains some bad values. In each file, column 4 contains the observed value for each measurement. We expect this vector to be numeric, but in file `data-02` a factor is returned instead.

```
> x ← read.delim('data-02', header=FALSE)
> class(x$value)
[1] "factor"
```

Since concatenation is closed when all elements are of the same type, we know that there must be bad data in the file. Indeed, some records contain a value of `0Hi` or `0Lo`, so R coerced the column first to character and then to factor.

□

Exercise 2.1. Why did the last statement in Example 2.1.10 get coerced to integer and not double?

2.2 Set theory

Most data are readily interpreted as sets. Panel data in particular can easily be partitioned into disjoint sets. Figure 2.3 visualizes this stylistically, where panel data $X = A \cup B \cup C$. In the diabetes dataset each person can represent a panel. It is clear that the panels are disjoint and the union composes the original dataset. Another example is time series data with rolling windows that can be modeled as overlapping sets. Here the sets are not disjoint, so the windows do not partition the original set. Nonetheless, the original set is a subset of the union of rolling windows. We can see that using the language of sets to describe data is natural, and R provides a rich vocabulary to do just this.

2.2.1 The empty set

Befitting a set-theoretic discussion, let's first consider the empty set.

Definition 2.2.1. A vector with 0 elements is called the *empty set* and is denoted as $c() \equiv \emptyset =$ `NULL`. As expected, the cardinality of the empty set is $|c()| = 0$.

One difference between the R version of the empty set and the mathematical empty set, is that the empty set is not preserved in concatenation.

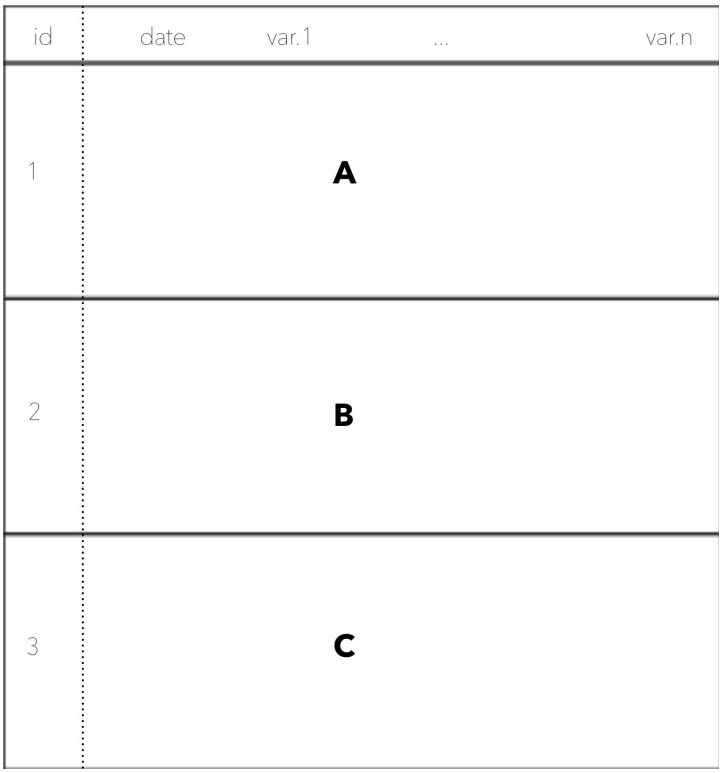


FIGURE 2.3: Panel data as a union of sets

Hence, the empty set acts as the identity for the concatenation function: $c(c(), x) = c(x) = x$ and $c(x, c()) = x$.

This means that $\emptyset$ cannot be an element of a vector and cannot be tested for set membership.

```
> c() %in% c(3,4,c())
logical(0)
```

The empty set affects concatenation in other ways, too. As a consequence of being the identity, the empty set satisfies commutativity over concatenation. This can create a conundrum if one expects the cardinality of the input and output to be consistent. For example, suppose some function f operates on a vector x. If one of the results is $\emptyset$, the output will have a cardinality one less than the input cardinality. Even though the operation preserves ordering, the ordinals are not preserved. This loss of information means that it is not clear how to construct the mapping between input and output vectors.

Example 2.2.1. Suppose a function f operates on a vector, returning the below result. In the output, what is $ord(7)$? We cannot know, since the ordinals were not preserved.

```
> f(c(1,2,3,5,8,13))
[1] 7  3  6  9 11
```

□

Example 2.2.2. Despite this wrinkle caused by $\emptyset$, note that the calculation of ordinal positions is computable if the original ordinal of $\emptyset$ is known. Starting with Example **??**, define `b ←c(x, c(), y)`. The ordinal position of y_2 in b is $|x| + 0 + ord(y_2) = 5$. So `b[5] == y[2]`.

□

2.2.2 Set membership

We generally think of sets as comprising a collection of elements. These elements are said to be *in* the set. Set membership is represented mathematically by the symbol $\in$, whereas in R the `%in%` operator is used. On the surface the semantics may seem the same but there are subtle distinctions between the two. For a given element a and set A, the expression $a \in A$ acts as a statement of fact or condition to be satisfied. Compare this to `a %in% A`, which is a logical test that produces a boolean value. These two forms converge when we use the expression $\forall a \in A$, which describes a set. This latter expression maps more closely to the semantics of the

Example 2.2.3. Let's use the diabetes dataset to illustrate this more clearly. In this dataset, data were collected using two different techniques. While an automatic recording device captured events precisely, paper records only

captured the timestamp of a measurement based on four "logical timel slots [**?**]. To find the records associated with the paper records can be accomplished using set membership.

```
> timeslots ← c('8:00','12:00','18:00','22:00')
> subset(diabetes[[1]], time %in% timeslots)
          date  time feature value
21  1991-04-24 12:00       33     4
53  1991-04-29 12:00       65     0
66  1991-04-30 22:00       65     0
71  1991-05-01 12:00       65     0
95  1991-05-04 12:00       33     5
...
```

□

You may have noticed that from a language perspective there isn't much of a difference between an element of a set and a set. This is clearly due to the fact that all atomic types are vectors. Like other operators in R, `%in%` is vectorized. The implication is that multiple elements can be tested for membership in a single operation. Testing for set containment is simply aggregating over set membership of each element in question.

Definition 2.2.2. Given sets A and B, $A \subseteq B$ iff $\forall a \in A, a \in B$.

Example 2.2.4. Let $x = c(5, 6, 7)$ and $A = 1 : 10$. The set x is a subset of A iff $x_i \in A \; \forall x_i \in x$.

```
> all(x %in% A)
[1] TRUE
```

Since each operand to `%in%` is a vector, it may be tempting to reverse the operands. However, set membership is not commutative.

```
> all(A %in% x)
[1] FALSE
```

□

2.2.3 Set comprehensions and logic operations

A powerful method for constructing sets is the set comprehension. The general syntax is $\{x|\Phi(x)\}$, which is read "the set of values x such that $\Phi(x)$ is true". For example, the natural numbers can be extracted from the integers using the following set comprehension: $\{x|x \in \mathbb{Z} \wedge x > 0\}$. The notation in R is similar but is constructed from a specific domain, which indicates that the set membership clause is implied.

```
z ← -1000:1000
z[z > 0]
```

Operator	Logic	R	Length
negation	$\neg$	!	n
conjunction	$\wedge$	&	n
disjunction	$\vee$	—	n
all	$\bigwedge_k$	all()—	1
any	$\bigvee_k$	any()—	1

TABLE 2.1: Logical operators given input(s) of length n

Like set notation, the R syntax supports logical operations to further specify the set membership rules. The standard set of boolean operators are valid in addition to the aggregation operators, as shown in Table 2.1. Since the goal is to construct a set from another set, the vectorized version of the boolean operators must be used. Constructing set comprehensions like this is useful for removing extreme values from data or selecting a specific slice of data. While the `all` and `any` operators are vectorized, notice that they return scalars instead of vectors. These functions are useful for detecting a specific condition within a set, such as an outlier or an unwanted value.

Example 2.2.5. Define `y` $\leftarrow$`x[x > 3]`. If $|y| > 0$, then $any(x > 3)$ is true.

□

Example 2.2.6. In one of the panels of the diabetes dataset, suppose we want to examine data for feature 69 (typical exercise activity) but only for times before lunch. This requires two logical conditions to be met, which is described as $\{x|x\$feature = 69 \wedge x\$time < 12:00\}$. The corresponding R expression is `x[x$feature==69 &hour(x$time)< 12,]`.

□

Note that an expression that returns the empty set yields a vector of length 0. This behavior implies that in R the empty set is not unique! While a length 0 vector is conceptually equivalent to `c()`, they are technically different. The reason is that `c()` has no type information, while a 0-length vector preserves type.

Example 2.2.7. Suppose we want to extract all samples associated with an non-existent feature. This results in a 0-length integer vector.

```
> diabetes$value[diabetes$feature==37]
integer(0)
```

□

2.2.4 Set complements

In Example 2.2.3 we identified the records associated with the paper records. Suppose we want to find the complement, which are all the records associated with the automatic recording device. Suppose we want to only use the panels without these outliers. What we want is the complement, or negation, of the original set of this set, which is $\forall a \notin A$. The `!` (bang) operator is used for this purpose when using the `%in%` operator. The mechanical operation is unary logical negation, and when applied to `%in%` becomes set negation. It is now possible to show that the theorem $A \cup \bar{A} = X$, where $A \subseteq X$ holds with the R operators.

To simplify the analysis we'll only look at the time component of one of the panels, so `x ←diabetes[[1]]$time`. Now we can construct the set *a*, which contains the times assouated with the paper records and *ac* as its complement.

```
a ← x[x %in% timeslots]
ac ← x[! x %in% timeslots]
```

The union is simply the concatenation of these two subsets.

```
y ← c(a, ac)
```

The only task remaining is to verify that the two sets are equal. Unfortunately since vectors are ordered tuples, this ordering adds some complexity. Or does it? Set theory actually tells us how to test for set equality and the same approach is valid in R! Recall that two sets are equal if they are subsets of each other. Hence, $X = Y$ iff $X \subseteq Y$ and $Y \subseteq X$. In R, we have

```
all(x %in% y) && all(y %in% x)
```

Another form of set negation is based on a negative index. The negative index removes the given ordinal positions from a set, which can also be viewed as set subtraction. Consider a dataset `x` whose elements correspond to $X = \mathbb{N}_n$. We can construct a subset $Y \subset X$ such that $\bar{Y} = X - Y$. Since Y can similarly be described by its index, notice the congruence between set subtraction and the set complement.

Definition 2.2.3. A negative index on a vector x is equivalent to set negation and is denoted $x[-i] \equiv \langle x_k | k \neq i \rangle_{k \in \mathbb{N}}$ where index order is preserved. In other words $x[-i] = c(x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n)$. Due to the duality of scalars and vectors of unit length, this definition extends to any vectors in $-\mathbb{N}_k$, where $k \leq |x|$. Given a vector $y \subset \mathbb{N}_k$, $x[-y] \equiv x - \{x_i\}_{i \in y}$.

This definition shows us that a vector of negative indices negates this subset within the context of the original set.

Example 2.2.8. In Figure 2.2 we remarked that the diabetes dataset contains panels with bad data. Instead of removing them in the `load_diabetes` function, suppose we loaded everything and now need to remove the panels prior

to performing our analysis. To remove these rows from the dataset let's define `y ←c(2,27,29,40)`. Then the filtered set is simply `diabetes[-y,]`.

□

Proposition 2.2.4. *Let x be a vector of length n and $k \in \mathbb{N}_n$. Then $c(x[k], x[-k]) \neq x$.*

Proof. This follows from the definition.

$$\begin{aligned} c(x[k], x[-k]) &= c(x_k, c(c(x_1, x_2, ..., x_{k-1}), c(x_{k+1}, ..., x_n))) \\ &= c(x_k, x_1, x_2, ..., x_{k-1}, x_{k+1}, ..., x_n) \end{aligned}$$

But $x = c(x_1, x_2, ..., x_n)$, so $c(x[k], x[-k]) \neq x$ □

This proposition underscores the fact that the use of negative indices is not invertible. In set thoery a set can be reconstructed after removal of elements: $(A - B) + B = A$, which is possible because sets are unordered. With ordered sets, reconstructing the original set is cumbersome and requires splitting the dataset according to the original ordinal positions.

Negative indices preserve ordering in the sense that the resultant set is the same irrespective of the order of the negative indices.

```
y1 ← sample(y,4)
all(x[-y] == x[-y1])
```

Example 2.2.9. In Example 2.2.8 we removed the bad panels from the diabetes dataset. Suppose for a moment that a vector `x` contains the panel indices for the complete dataset. In other words, `x ←1:70`. Another vector tracks the panels with bad data: `bad ←c(2,27,29,40)`. Let's create a new vector that has the bad indices removed, such as `y ←x[-c(2,27,29,40)]`. How can we recreate the original vector at some arbitrary point in the future? Reconstructing the original ordinals requires shifting `y` at the four given index positions. One way of doing this involves creating a 2D structure where each row represents the segments of the vector.

```
segments ← cbind(c(1,bad), c(bad-1,length(x)))
x.new ← lapply(1:length(bad),
  function(idx) c(x[segments[i,1]], x[segments[i,2]], bad(idx)
      ))
x.new ← do.call(c, x.new)
```

□

The complexity surrounding the reconstruction of the data is due to two factors. First is information loss. If the original ordinals were preserved, then the reconstruction would be far simpler. Suppose we recorded the original ordinals in the names attribute of `x`, as

```
names(x) ← 1:length(x)
```

Reconstructing the original vector is then a matter of sorting:

```
x.new ← c(y, bad)
x.new[order(names(x.new))]
```

We'll discuss this technique in more detail in Section 2.3.1. A more fundamental issue is that our attempt at reconstruction was based on a map process. For this situation what is needed is a fold process, which provides a means for iteratively reconstructing the vector instead of via segmentation. The distinctions between these two vectorization processes are discussed in Chapters 3 and 4.

2.3 Indexing and subsequences

The previous section treated vectors as sets and showed how to extract subsets from a set. In this section, we look at the mechanism of indexing and how it can be used to extract any subsequence from a vector. The indexing operator performs this function and is applied to an *index vector* **[?]** that represents the ordinal positions of the elements contained in the vector. As such the indexing operator supports many extraction methods determined by the type of the indexing operand. The most basic subsequence is created by specifying an index vector as a set of integers representing the ordinal positions of the target vector.

Definition 2.3.1. Let x be a vector, where $|x| = n$. Let $y \subset \mathbb{N}_n$, where $|y| = m$. A *subsequence* of x is $x[y] \equiv \langle x_{y1}, x_{y2}, ...x_{ym} \rangle$. Given $x = \langle x_k \rangle_{k \in \mathbb{N}}$, then

(a) a subsequence from the range $i : j$ is $x[i : j] = \langle x_k \rangle_{k=i}^{j}$;

(b) if $i = j$ then $\langle x_k \rangle_{k=i}^{i} = \langle x_i \rangle = x_i$.

For notational convenience, I use a shorthand notation to represent subsequences where $\langle x_k \rangle_{k=i}^{j} \equiv x_{i:j}$. In this notation the subscript k is assumed and may be referenced in certain expansions.

Example 2.3.1. Let x be a vector $c(1, 1, 2, 3, 5, 8, 13, 21)$ and $y = c(2, 3, 6)$. Then $x[y] = c(1, 2, 8)$.

□

2.3.1 Named indices

We saw in Example 2.2.9 that the `names` attribute can be used to preserve the original ordinals of a vector. In addition to ordinals, indexing also supports character names. These names provide a mapping between unique characters

and ordinal positions. Under the hood, named indices are implemented as a character vector of unique values that has the same cardinality as the target vector.

Definition 2.3.2. Let x be a vector. A character vector a is a *named index* of x if $|a| = |x|$ and $a_i \neq a_j \forall i, j \in \mathbb{N}_{|a|}$, where $i \neq j$. Ordering is preserved between a vector and its named index, so elements of x can be extracted using a such that $x[a_i] = x_i$.

A named index is associated with a vector using the `names` function. Using x as defined in 2.3.1, let's associate the alphabet to this vector.

```
> names(x) ← letters[1:length(x)]
```

We can now extract elements of x using the associated names.

```
> x[c('b','c','f')]
b c f
1 2 8
```

The `names` function is peculiar insomuch that it is not vectorized and is also more liberal than the mathematical definition. For example, assigning only three names to the above vector yields a named index with compatible length where the remaining elements are `NA`.

```
> names(x) ← NULL
> names(x) ← letters[1:3]
> x
   a    b    c <NA> <NA> <NA> <NA> <NA>
   1    1    2    3    5    8   13   21
```

Another curiosity is that names do not technically need to be unique during assignment, but extracting elements by name will only return the first encountered element!

```
> names(x) ← c(rep('a',4), rep('b',4))
> x[c('a','b')]
a b
1 5
```

Example 2.3.2. In Example **??** we saw how it is difficult to reconstruct an ordered set after set subtraction. Using named indices, reconstruction becomes far simpler.

```
> names(x) ← letters[1:length(x)]
> ns ← names(x)
> z ← c(x[-c(3,5,6)], x[c(3,5,6)])
> z
 a  b  d  g  h  c  e  f
 1  1  3 13 21  2  5  8
> z[ns]
 a  b  c  d  e  f  g  h
 1  1  2  3  5  8 13 21
```

This works because names are preserved over concatenation and subsetting operations. Hence the original ordinal structure can be retrieved from a saved copy of the original named index.

□

Since the `names` attribute is character-based, assigning the names with a vector of another type will automatically coerce them to character. The consequence is that comparison operators will be character-based, which may result in unexpected behavior. For instance, numeric indices will compare lexically instead of numerically.

```
> x ← letters
> names(x) ← 1:length(x)
> x[names(x) < 13]
  1  10  11  12
"a" "j" "k" "l"
```

This doesn't necessarily need to be an impediment so long as the lexical order is consistent with the ordering desired. A common trick to force lexical ordering of integers to match their natural ordering is to pad them with leading zeroes.

```
> names(x) ← sprintf('%02i',1:length(x))
> x[names(x) < 13]
 01  02  03  04  05  06  07  08  09  10  11  12
"a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
```

Exploiting this property means that objects like `Date`s can be compared without issue. Otherwise, only the `%in%` operator would be available, which can complicate the set comprehension.

Named indices can be used to emulate a hash data structure. This is not a true hash, since lookup has $O(n)$ complexity. [6] In essence, it is possible to create key-value pairs using a vector and a corresponding named index.

Example 2.3.3. In Example 2.2.3, we extracted all of the paper records using a vector that described the fictitious times. Suppose we want to find only those records associated with lunch. Rather than using the time, we can create a key-value lookup so that the times can be accessed by word.

```
> timeslots ← c('8:00','12:00','18:00','22:00')
> names(timeslots) ← c('breakfast','lunch','dinner','bedtime')
> subset(diabetes[[1]], time %in% timeslots['lunch'])
          date  time feature value
21  1991-04-24 12:00      33     4
53  1991-04-29 12:00      65     0
71  1991-05-01 12:00      65     0
95  1991-05-04 12:00      33     5
...
```

[6] For $O(1)$ complexity, an `environment` object must be used.

□

2.3.2 Logical indexing

Subsequences can also be created from logical vectors. Reasoning about this process requires the help of the inclusion function, which is inspired by the Kronecker delta.

Definition 2.3.3. Let a be a logical scalar and x a scalar. The *inclusion function* is defined $\delta_I(a, x) = \begin{cases} x, & \text{if } a \text{ is true} \\ \emptyset, & \text{if } a \text{ is false} \end{cases}$

This function produces a value whenever the predicate a is true. In fact, the behavior of logical indices is prediated on order isomorphism insomuch that the logical vector must preserve the order of the original vector. Since these operations are vectorized, this assumption usually holds. However, when manipulating indices directly, this assumption may not necessarily hold.

The subsequence of a vector given a logical index vector can now be defined in terms of the inclusion function.

Definition 2.3.4. Let x be a vector where $|x| = n$, and let a be a logical vector such that $|a| = |x|$. Then $x[a] = \langle \delta_I(a_k, x_k) \rangle_{k=1}^{n} \equiv \delta_I(a, x)$.

Example 2.3.4. When using a logical vector to extract a subset the length must be compatible with the target vector. Let's use the Fibonacci sequence to illustrate the behavior of the inclusion function. Define `x ←c (1,1,2,3,5,8,13,21,34)`. How do we extract the elements that are even? In a procedural language it is a matter of iterating over the vector, testing each value, and adding the matches to another vector.

```
y ← c()
for (i in x) {
  if (i %% 2 == 0) y ← c(y, i)
}
```

The idiomatic approach is to construct a set comprehension with a logical test to extract only those elements that resolve to `TRUE`.

```
x[x %% 2 == 0]
```

Let's break it down into two pieces: the logical index and the subsetting operation. The logical index is simply a vector with the result of the modulus test for each element. Like other native operators, modulus is vectorized and returns a vector with the same length as its argument. The expression `x %% 2 == 0` yields the logical vector `c(FALSE, FALSE, TRUE, ...)`. When the subsetting operation is applied to a logical vector with compatible length, its behavior is governed by the inclusion function. Hence `even ←x %%2 == 0` implies $x[even] = \delta_I(even, x)$.

□

Proposition 2.3.5. *Let x be a vector where $|x| = n$, and let a be a logical vector such that $|a| = |x|$. Then $0 \leq |x[a]| \leq n$.*

Proof. From the definition of δ_I, $|\delta_I| = \begin{cases} 1, & \text{if } a \text{ is true} \\ 0, & \text{if } a \text{ is false} \end{cases}$.
The lower bound on $|x[a]|$ occurs when a_k =false $\forall k \in \mathbb{N}_n$. Then $x[a] = \emptyset$ and $|x[a]| = 0$. Similarly the upper bound is reached when a_k =true $\forall k \in \mathbb{N}_n$ and $x[a] = x$ and $|x[a]| = n$. □

The fact that subsetting is a bounded operation implies that the ordinals of a subset are generally compatible with the original set. The significance of Proposition 2.3.5 is in helping to prove the following proposition.

Proposition 2.3.6. *Let x be a sequence and a be a sequence of logical values, such that $|a| = |x|$. Then $\{x[a], x[-a]\}$ is a partition of x.*

Proof. Pending □

Example 2.3.5. The `ifelse` function exploits this property to construct a conditional vector. At 0, the sinc function must be defined explicitly, since division by 0 is not allowed. This can be expressed by `ifelse(x == 0, 1, sin(x)/x`. With this function, a vector is created for each alternative. The values selected are based on the properties of the partition, which are $x[a] \cap x[-a] = \emptyset$ and $x[a] \cup x[-a] = x$. Since these are sequences and not sets, note that the union preserves the ordering.

□

2.3.3 Ordinal mappings

With consistent ordering, we've seen how a set of ordinals can be used to associate elements between multiple sets. Set operations yield the elements satisfying the given expression, while the raw expression yields a logical vector. How then do you obtain the actual ordinal values associated with the logical vector? Enter the `which` function. The behavior of this function can be described using the inclusion function defined in Definition 2.3.2.

Definition 2.3.7. Let a be a logical vector with length n. Then $which(a) = \delta_I(a, \mathbb{N}_n)$.

Example 2.3.6. In the diabetes dataset, suppose we want to know which pre-breakfast glucose measurements are greater than 2σ from the mean.

```
> with(subset(diabetes[[1]], feature==58),
+   which(abs(value - mean(value)) > 2 * sd(value)))
[1]  36  42 111
```

□

It becomes clear that `which` is simply returning the ordinals associated with true values. Interestingly, this operation is context-free, meaning that ordinals constructed this way may not be appropriate for a given vector. As an extreme example, let's extract the ordinals of a random logical vector via `which(sample(c(TRUE,FALSE), 20))`. Now do these ordinals have any significance?

This line of questioning brings up another aspect of ordinal mappings, which is that a set of ordinals can be applied to multiple vectors if the vectors share the same context, such as with multivariate data. More generally, any two-dimensional dataset possesses this quality across columns (rows), meaning that a single set of ordinals applies consistently to any column (row).

Example 2.3.7. When extracting a specific feature from the diabetes dataset, this presevation of ordinals is used to drive the indexing. Let `x` ←`diabetes [[1]]`. The values associated with feature 58 is obtained via

```
x$value[x$feature == 58]
```

Since each row contains a specific sample of data, the expectation is that the logical vector produced by the conditional expression has ordinals that map to the vector `x$value`. Hence, to obtain the associated time measurements, the same logical vector can be used, such as `x$time[x$feature == 58]`.

□

2.3.4 Sorting

In other languages sorting is done as a direct operation on a vector so that the result is the sorted vector. In R sorting relies on indexing, so the process is indirect. Basic sorting is performed by the `order` function. Instead of returning a modified version of the input vector `order` returns a vector of the corresponding ordinals. The second step is to extract a subset according to the new ordinal sequence. This two-step dance is a constant source of confusion for newcomers as it takes a non-traditional approach to a common computing problem. When viewed from the perspective of indexing, sorting becomes a natural extension to a fundamental semantic construct of the language.

Example 2.3.8. Suppose we want to order the observations for feature 58 according to time of day. We accomplish this by calling `order(x$time)`. The output is a set of ordinals where $y_i \in \mathbb{N}_{|x\$time|} \forall y_i \in y$. The second step is applying this ordering to the vector of actual values, which is `x$value[y]`.

□

Now why might this scenic route be preferred over the more direct route? It is debatable whether this extended syntax provides any visual pleasure

so there must be another reason. Consider the process of data analysis. It is common to have numerous variables representing a set of features. In this situation all variables have the same length and have a consistent ordering such that the indices correspond to the same sample. By separating the ordinals from the actual mutation of a data structure means that the same ordered index can be applied to multiple vectors. Another advantage is that by unaltering the original vector, multiple orderings can be created quickly. Indeed, this is the basis for the `sample` function.

Exercise 2.2. Given $x = \langle NA, 1, 0, NA, -1, -1 \rangle$,
$a = \langle TRUE, FALSE, FALSE, TRUE, FALSE, FALSE \rangle$, reduce $x[a]$ based on the inclusion function.

Exercise 2.3. Given x as defined in Exercise 2.2, $P(x) = \begin{cases} true, & if x = 0 \\ false, & otherwise' \end{cases}$
reduce the set comprehension $\langle x|P(x) \rangle$ mathematically.

Exercise 2.4. Show $x[a]$, where $a \subset -\mathbb{N}_k$, $|x| = k$.

2.4 Recycling

When two operands do not have the same length, recycling provides a mechanism for extending the shorter vector to the length of the longer vector. If the length of the shorter vector divides the longer vector, then the shorter vector is repeated until the lengths are equal. Recycling greatly enhances the power of vectorization. It takes a mathematical perspective to see why this concept is so powerful. Consider the addition of a scalar with a vector. The rule is to add the scalar value to each element of the vector, irrespective of the length of the vector. Hence, the scalar value is repeated until the length of the resulting vector is equal to the other operand. We typically don't consider the mechanical aspect of this operation, although a close examination reveals its nature:

$$a + \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} a + x_1 \\ a + x_2 \\ a + x_3 \end{bmatrix}$$

Given a scalar `a` and vector `x`, the notation is the same in R: `a + x`. For a vector operand with arbitrary length n, linear algebra thus gives rules for operands of length 1 and length n, both of which divide n. R simply takes this concept a step further and applies the recycling rule to any operand whose length divides the length of the longest vector.

The behavior of recycling can be codified with the `rep` function.

Definition 2.4.1. Let $f : X^m \times Y^n \to Z^p$. If $m \geq n$ and n divides m, then

```
hour ← function(x) {
  parts ← strsplit(x, ':', fixed=TRUE)
  sapply(parts,
    function(p) as.numeric(p[1]) + as.numeric(p[2])/60)
}
```

FIGURE 2.4: Convert a time into a decimal hour

$y \to rep(y, \frac{m}{n})$ and $p = m$. Similarly if $n > m$ and m divides n, then $x \to rep(x, \frac{n}{m})$ and $p = n$.

Example 2.4.1. Let `x ←3:6` and `y ←8:19`. Then `x + y == y + x`.

Vector lengths are not always compatible. If m does not divide n, then the operation will not succeed and an error will be thrown:

```
Warning message:
In y + 1:5 :
  longer object length is not a multiple of shorter object
      length
```

Note that this also happens when a vector has 0 length, which can result from a bad calculation. In chapter **??**, we'll see how recylcing affects vectorized operations.

Not limited to addition, recycling rules are honored for most arithmetic and algebraic operators by default. In addition to these operators, concatenation and assignment operators also support recycling. This is used most often within the context of two-dimensional data structures. For example, adding a scalar as a column to a matrix is done via

```
m ← matrix(rnorm(12), nrow=3)
cbind(m, 1)
```

A column can also be changed by assigning a scalar to it, which of course is recycled to the same dimension as a column in m.

```
m[,2] ← 0
```

Recycling also applies to `data.frame`, which makes it easy to assign values to panel data. For example, suppose we want to add a column that specifies the the closest meal associated with a measurement. We'll compute this based on a decimalized hour as given by Figure 2.4.

```
lapply(diabetes, function(d) {
  d$meal ← NA
  d$meal[hour(d$time) < 11] ← "breakfast"
  d$meal[hour(d$time) >= 11 & d$time < 16] ← "lunch"
  d$meal[hour(d$time) >= 16] ← "dinner"
  d
})
```

Using this approach, data can be quickly added or updated in a manner that is clear and concise. This is the power of a declarative style.

Example 2.4.2. Suppose we want to find all outliers in the iris dataset. We construct a set comprehension to select only those records greater than six standard deviations. The first argument *abs* is map-vectorized so $|abs(x)| = |x|$. On the other side of the comparison operator, $|6| = 1$ and $|sd(x)| = 1$. Hence $|6 * sd(x)| = 1$. Now due to recycling, $|abs(x) > 6 * sd(x)| = |x|$.

□

2.5 Exercises

Exercise 2.5. Let $x = rnorm(10)$. Determine the length of the operation $(x - \overline{x})^2$.

3

Map Vectorization

No matter what you do in R, you're bound to come across a *map* operation. Since R is a vector language, *map* operations are nearly ubiquitous. The reason is that most functions are vectorized, and their arguments are typically vectors. Recall that whenever a function is not natively vectorized, *map* or *fold* can emulate that behavior. In this chapter, we'll look at the *map* paradigm more closely. We'll start by developing an intuition around *map*, using mathematical analysis as a motivation. With this foundation, we'll then work through a case study highlighting numerous uses of *map* to parse ebola situation reports from the health ministries of Liberia and Sierra Leone. The latter part of the chapter will examine *map* more closely and discuss its properties and how to effectively leverage these properties to aid in modeling data.

3.1 A motivation for *map*

Higher order functions are a staple of functional programming, and *map* might be the most common of the three canonical functions. Definition 3.1.1 shows that the essence of *map* can be distilled to iterated function application over a set of values.

Definition 3.1.1. Let $f : A \to B$ be a univariate scalar function. Given $X \subseteq A$, $map(f, x) \equiv \langle f(x) \mid x \in X \rangle$.

The motivation for a higher-order function like this is revealed in real analysis. Functions are defined in terms of a domain and range, where some input in the domain yields some corresponding output in the range. A function is visually represented as mapping the domain to the range, as depicted in Figure 3.1. If we apply only a subset X of the domain to a function f, the result is called the graph of f over X. Within this context it is understood that f is a scalar function and to produce the graph of f, f must be applied to every element $x \in X$.

In a programming language, we cannot rely on implied understanding to perform a calculation. To obtain the graph of f over X we actually have to apply f to each element in X! The first step is putting all these elements of X into a data structure. Suppose we assign each element of X with an index. It

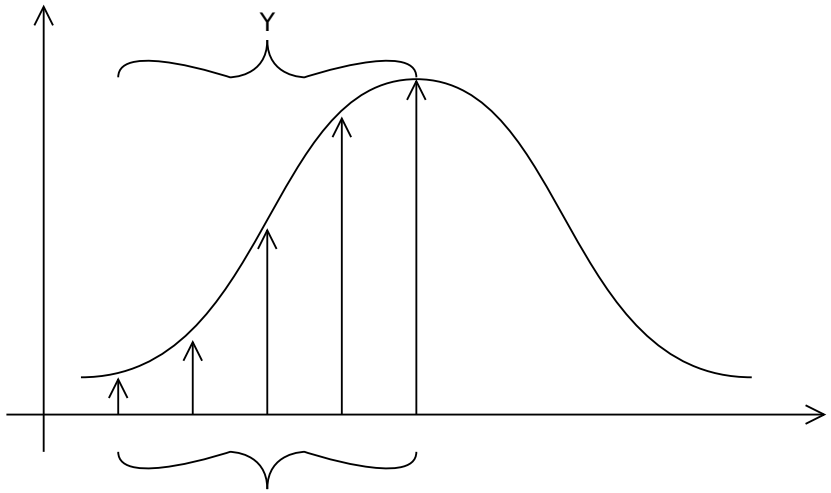


FIGURE 3.1: The graph of f over a set X

is only natural to assign the same index used in X for the output in Y. In other words, $y_i = f(x_i)$ for all elements in X. Now that we've created an *ordering*, we can put these elements into a vector that represents X. When we apply f to each element in X, we want the result Y to be in the same order as X. If not, it makes it difficult to use the result. Algorithm 3.1.1 outlines a naive approach that implements this rationale in a loop, assigning each output in the order of the input.

Algorithm 3.1.1: GRAPHOF(f, X)

for each $x \in X$
  do $Y[\text{LENGTH}(Y) + 1] \leftarrow \text{F}(x)$

Algorithm 3.1.1 is a standard imperative approach to this simple problem, and the telltale sign is that indices must be managed explicitly. That means we need to remember that R is 1-based and not 0-based, otherwise the algorithm will fail. On the other hand what does a functional approach look like? Recall that mathematically it is understood that f is applied to each element $x \in X$ with order preserved. The *map* function codifies this understanding (see Definition 3.1.1) so that any application of *map* on a function and a vector structurally behaves the same. Therefore, computing the graph is merely a call to *map*, implying that Algorithm 3.1.2 isn't even worthy of being called an algorithm!

Algorithm 3.1.2: GRAPHOF(f, X)

$Y \leftarrow$ MAP(f, X)

It may sound perverse, but it is actually desirable to disqualify this operation as an algorithm. The beauty is that *map* takes care of the low-level mechanics of computing a graph. This allows us to focus on the more important work of implementing the functions or transforms used in our analysis. A hallmark of functional programming is the generalization of common data transformations that can be used repeatedly. Once the vocabulary of higher order functions is mastered, algorithms take on a new form, one that is efficient and free of distraction.

3.1.1 Map implementations

In data analysis, we compute sub-graphs of a function all the time. Whenever we transform data, whether it's a log-linear transformation, a linear transformation, change of coordinates or base, the data is transformed as a set. This means that computationally these transformations are simply *map* operations. Algebraic and trigonometric operations are already vectorized, so we don't immediately think of these as *map* operations. However, if we think about how these operations transform a vector, the operation is semantically the same. We'll see in Section **??** how these operations are in fact equivalent. Whenever a function is natively equivalent to a *map* operation, we call that function as being *map – vectorized*, which distinguishes it from other forms of vectorization. For a language that has so many *map* operations, it may seem surprising that there is no explicit `map` function. While R traces some of its ancestry to Scheme, and therefore has numerous functional programming capabilities, it isn't always obvious how to use these features. There is no *map* function per se, but there is a suite of similar functions ending in `apply`. [1] Each version has slightly different semantics. Despite these differences, the syntax is more or less the same for the whole family. The general signature is `apply(x, fn, ...)`, which has reversed operands when compared to the canonical *map* function.

The eponymous function `apply` actually has a different signature since it does not operate on raw vectors. Instead, it operates on arrays, matrices, and data.frames. Hence, we will ignore it for now and discuss it again when exploring two dimensional data types. Similarly, `mapply` and `tapply` are designed to operate on two-dimensional data and will be ignored for now. This leaves `lapply` and `sapply`, but `lapply` only returns lists, which we haven't

[1] A more faithful implementation of *map* appears in the package `lambda.tools`. This implementation uses the author's `lambda.r` package, which implements a function disptaching system and type system using functional programming principles. The syntax and mechanics of these packages are out of scope for this book.

New Ebola Cases and Deaths Summarised by County

County	Alive[1]			Alive[2]	Deaths[1]			Deaths	
	Suspected & probable				Suspected & probable				
	Total	Susp.	Prob.	Confirmed	Total	Susp.	Prob.	Confirmed deaths in community[2]	Deaths in confirmed patients[3]
Bomi	2	0	2	0	0	0	0	0	0
Bong	1	1	0	0	0	0	0	0	0
Gbarpolu	1	1	0	0	0	0	0	0	0
Grand Bassa	5	3	2	0	0	0	0		0
Grand Cape Mount	0	0	0	0	7	6	1	0	0
Grand Gedeh	0	0	0	0	0	0	0	0	0
Grand Kru	0	0	0	0	0	0	0	0	0
Lofa	0	0	0	0	0	0	0	0	0
Margibi	1	0	1	1	0	0	0	0	0
Maryland	0	0	0	0	0	0	0	0	0
Montserrado	19	3	16	2	3	3	0	1	3
Nimba	5	5	0	0	1	1	0	0	0
River Gee	0	0	0	0	0	0	0	0	0
River Cess	0	0	0	0	0	0	0	0	0
Sinoe	0	0	0	0	0	0	0	0	0
NATIONAL	34	13	21	3	11	10	1	1	3

[1] From daily email county reports of aggregated data for that day
[2] Based on laboratory tests of suspects and probable cases identified during the preceding days
[3] Refers to cases confirmed while alive that later died in ETU, or elsewhere
■ County did not report

FIGURE 3.2: Ebola situation report table

discussed yet. Therefore, `sapply` will generally be used in this chapter since it both operates on and (mostly) returns vectors. When talking about these operations in the general sense, or at the algorithm level, we'll describe them as *map* processes or *map* operations. Specific R functions will be referenced when discussing the implementation of an algorithm.

3.2 Standardized transformations

Around the turn of the 20th century, Edison coined the saying "genius is 1% inspiration and 99% perspiration". Presumably Edison was referring to the process of invention, but the truth of this statement applies equally well to data science. Here, the 99% sweat is affectionately known as data munging or data wrangling. These tasks aren't so much about modeling as they are about making data usable. This includes getting data from remote sources, cleaning data, converting data into compatible formats, merging data, and storing it locally. Without this preparation, data are unusable and inappropriate for analysis. Enterprise software folks have been doing this for years, referring

```
<div id="page">
<div id="content">

 <li> <a href="/documents/Sitrep 233 Jan 3rd 2014.pdf" target="_blank">Situation Report on the EBOLA
Virus disease epidemic in Liberia as of March to 3rd January, 2015.</a> | Posted on   Jan 03, 2015</li>
   <li> <a href="/documents/Sitrep 232 Jan 2nd 2014.pdf" target="_blank">Situation Report on the EBOLA
Virus disease epidemic in Liberia as of March to 2nd January, 2015.</a> | Posted on   Jan 02, 2015</li>
   <li> <a href="/documents/Sitrep 231 Jan 1st 2014.pdf" target="_blank">Situation Report on the EBOLA
Virus disease epidemic in Liberia as of March to 1st January, 2015.</a> | Posted on   Jan 01, 2015</li>
   <li> <a href="/documents/Sitrep 230 Dec 31st 2014.pdf" target="_blank">Situation Report on the EBOLA
Virus disease epidemic in Liberia as of March to 31th December, 2014.</a> | Posted on   Dec 31, 2014</li>
   <li> <a href="/documents/SITRep 228 Dec 29 2014 final.pdf" target="_blank">Situation Report on the
EBOLA Virus disease epidemic in Liberia as of March to 29th December, 2014.</a> | Posted on   Dec 29,
2014</li>
   <li> <a href="/documents/SITRep 197 Nov 28th 2014.pdf" target="_blank">Situation Report on the EBOLA
Virus disease epidemic in Liberia as of March to 28th November, 2014.</a> | Posted on   Dec 28, 2014</li>
   <li> <a href="/documents/SITRep 196 Nov 27th 2014.pdf" target="_blank">Situation Report on the EBOLA
Virus disease epidemic in Liberia as of March to 27th November, 2014.</a> | Posted on   Dec 27, 2014</li>
   <li> <a href="/documents/SITRep 225 Dec 26 2014.pdf" target="_blank">Situation Report on the EBOLA
Virus disease epidemic in Liberia as of March to 26th December, 2014.</a> | Posted on   Dec 26, 2014</li>
   <li> <a href="/documents/SITRep 224 Dec 25 2014 (1).pdf" target="_blank">Situation Report on the EBOLA
Virus disease epidemic in Liberia as of March to 25th December, 2014.</a> | Posted on   Dec 25, 2014</li>
   <li> <a href="/documents/SITRep 223 Dec 24 2014 presentation.pdf" target="_blank">Situation Report on
the EBOLA Virus disease epidemic in Liberia as of March to 24th December, 2014.</a> | Posted on   Dec 24,
2014</li>
   <li> <a href="/documents/SITRep 222 Dec 23 2014 (2).pdf" target="_blank">Situation Report on the EBOLA
Virus disease epidemic in Liberia as of March to 23th December, 2014.</a> | Posted on   Dec 23, 2014</li>
   <li> <a href="/documents/SITRep 221 Dec 22 2014 Final Presentation.pdf" target="_blank">Situation
Report on the EBOLA Virus disease epidemic in Liberia as of March to 22th December, 2014.</a> | Posted on
```

FIGURE 3.3: Portion of the Liberia Ministry of Health situation report web page

to the general process as extraction, transformation, and loading, or ETL for short.

An example ETL process is consolidating ebola situation reports from West African health ministries. These reports provide numerous statistics on incidence, prevalence, and death rates across the various counties and provinces in each country. Figure 3.2 shows one of the tables published by the Liberian Ministry of Health in their daily situation report. Both the Liberia and Sierra Leone ministries publish data on more or less a daily schedule on their web sites. The reliability of the schedule is poor, so numerous days can elapse without any updates. While not an ideal situation, the health ministries should nonetheless be lauded for their efforts at disemminating information in a timely manner. The situation reports themselves are published as PDF files and are not immediately machine readable. Using this data in an analysis therefore requires downloading numerous documents, parsing each one, and finally merging them into a single dataset. In terms of an algorithm, if all documents are formatted the same, the parse process will be exactly the same for each document. This observation indicates that the parse process can be managed by a *map* operation.

3.2.1 Data extraction

Let's start by downloading some data. We'll use the `scrapeR` package to download and parse web pages. The first source is the Liberia Ministry of Health, so we'll create constants for that and also a destination directory. [2]

[2] In the actual package, these constants are private and named `.LR_SOURCE` and `.LR_DEST`.

```
url ← 'http://www.mohsw.gov.lr/content_display.php?sub=report2'
base ← './data/lr'
```

This source web page lists the historical situation reports, linking to each individual daily PDF report. A relevant snippet of the HTML source is in Figure 3.3. To download each situation report, we need to extract each report link and format it into a proper URL. Since the web page is XML it is easy to find all relevant URLs by using an XPath query [4] to represent the location of each URL in the document: `//div[@id='content']/li/a`. Reading from right to left, this expression simply says to match all `<a>` tags that have `<li>` as a parent, which have `<div>` as a parent. The `<div>` tag must have an `id` attribute equal to 'content'. While this narrows down the set of links, it can still contain links we aren't interested in, such as `Final_NHSW_P_P_(low_resolution).pdf`. Even though the situation reports do not have a standard naming scheme, they do share a common prefix, which is different from the artifact. We'll use this observation to construct a pattern to filter the result set and remove unwanted artifacts. We'll also convert white space to URL-encoded spaces to save some work later on. The resulting Algorithm 3.2.1 is encapsulated into the function `find_urls` and is applied to each matching node of the XPath query.

Algorithm 3.2.1: FINDURLS(*AnchorNode*, *Pattern*)

Link ← GETATTRIBUTE(*AnchorNode*, "href")
if *Link* contains *Pattern*
  then Replace ' ' with '%20' in *Link*
  else ∅

Armed with this function, it is simply a matter of applying it to every matching node. The `xpathSApply` function in the `XML` package does just this. The function name hints at its heritage as a descendant of `sapply`. This means that we can expect `xpathSApply` to behave like a *map* process. Indeed, this function applies the given function to each node that matches the XPath query. Figure 3.4 visualizes this process including the arguments passed to each function. Putting it all together we have

```
page ← scrape(url)
xpath ← "//div[@id='content']/li/a"

find_urls ← function(x, pattern) {
  link ← xmlAttrs(x)['href']
  link ← grep(pattern,link, value=TRUE)
  gsub(' ','%20',link, fixed=TRUE)
}

links ← xpathSApply(page[[1]], xpath,
  function(x) find_urls(x,'SITRep|Sitrep')).
```

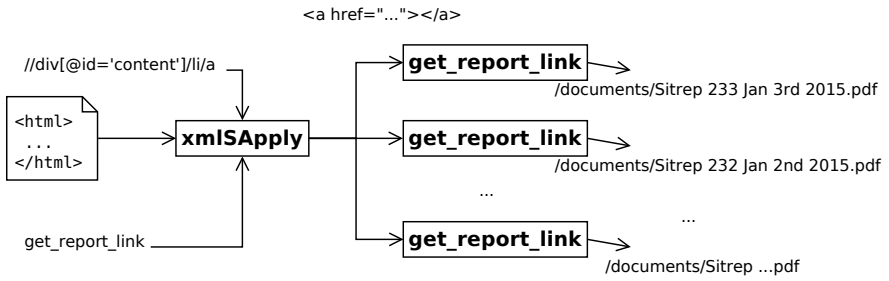


FIGURE 3.4: How `xpathSApply` transforms an XML document. Boxes represent functions, and arrows represent input/output.

Now we have a character vector of all the paths to situation reports. This isn't sufficient for downloading the files since we still need complete URLs. We can construct these by parsing the original URL (via the `httr` package) and extracting the various components of the URL that we need. Can you spot the map-vectorized call in the listing?

```
url.parts ← parse_url(url)
urls ← sprintf("%s://%s/%s",
  url.parts$scheme, url.parts$hostname, links)
```

In addition to the URLs, we also need the destination file names to write to once we download each URL. Normally we would use the source document name directly, but they contain URL-encoded white space (`%20`), which adds unnecessary complexity (i.e. headache) to UNIX-like systems. It is generally easier to work with documents when they have more convenient names. Therefore, we'll replace these characters with _s. Algorithm 3.2.2 shows the steps for a single URL, which is then applied to each document path via `sapply`.

Algorithm 3.2.2: GetFileName(*DocumentPathParts*)

$X \leftarrow$ *DocumentPathParts*[*length*(X)]
Replace '%20' with '_' in X

The document paths are absolute, so the actual name of the file is contained in the last path component. The corresponding R code is

```
get_file_name ← function(x) gsub('%20', '_', x[length(x)])
files ← sapply(strsplit(links,'/'), get_file_name).
```

Notice that Algorithm 3.2.2 is implemented completely in the first line, followed by the *map* operation over all valid links extracted from the HTML

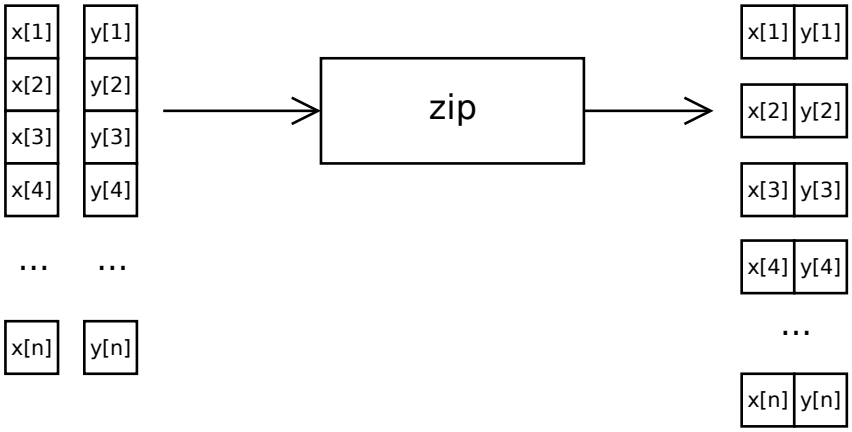


FIGURE 3.5: Zip converts column-major data into a row-major structure. Column-major data structures give fast sequential access along column indices. A row-major structure optimizes for row access.

page. Finally, we can pass each input URL and corresponding output file destination to a function that does the actual work of downloading and writing the file. This function uses `httr` to download the PDF as raw binary data:

```
extract_pdf ← function(url, file, base) {
  flog.info("GET %s",url)
  response ← GET(url)
  pdf ← content(response,'raw')
  pdf.dest ← paste(base,file,sep='/')
  writeBin(pdf, pdf.dest)
}.
```

Calling this function requires a bit of glue that we'll discuss in the next section.

3.2.2 Multivariate *map*

Since *map* specifies as its argument a univariate function, how do you pass the second argument to `extract_pdf` via *map*? If only one argument varies, the standard R way to do this is to use the ellipsis argument. Had we implemented `extract_pdf` to construct the destination file name within the body, the call would look like `sapply(links, extract_pdf, base)`. While convenient, this is an idiom specific to R and not portable. Functional languages solve this by using a closure to conform the two function interfaces. Using this approach the call looks like

```
sapply(links, function(link) extract_pdf(link, base)).
```

When only a single argument varies over *map*, this approach works because all other arguments are treated as constants.

Example 3.2.1. Calculating the Z-score for a vector is an example of a function with multiple arguments, though only one varies.

```
xs ← rnorm(100)
x.mean ← mean(xs)
x.sd ← sd(xs)
z_score ← function(x, m,s) (x-m)/s
```

We can convince ourselves that the two computations are the same with a quick test of equality.

```
all(
  sapply(xs, function(x) z_score(x,x.mean,x.sd))
  ==
  sapply(xs, z_score, x.mean, x.sd)
)
```

Note that the idiomatic way of computing the Z-score uses the native *map* vectorization of the arithmetic operators. Hence, the following one-liner is also equivalent.

```
(xs - mean(xs)) / sd(xs)
```

□

In cases where arguments co-vary, the above approach will not work. Instead, the pairs of arguments must be passed to the first-class function together. This process is known as *zip* and works by effectively converting a column-major data structure into a row-major data structure (see Figure 3.5). The distinction between these two formats is how a two-dimensional array is organized. Column-major structures preserve vectors along columns, whereas row-major structures preserve vectors along rows. [3] The choice depends on how data is accessed. R doesn't provide a native `zip` function, nor does it support tuple notation, making emulation tricky. The function `cbind` can emulate *zip*, by creating a matrix from a set of vectors. [4] However, since Rstores data in a column-major format [5], it can be more consistent to use `rbind`, despite the confusion in terminology. This choice becomes more apparent when working two-dimensional data, as discussed in Chapter **??**. Algorithm 3.2.3 illustrates this pattern for a function of two variables. The same approach is valid for any arbitrary multivariate function.

[3] R stores data in column-major format.
[4] As with *map*, we'll use the canonical name in algorithms but actual functions in code listings.
[5] Clearly, $cbind(x, y, z) = rbind(x, y, z)^T$.

Algorithm 3.2.3: MULTIVARIATEMAP(x, y, fn)

block ← ZIP(x, y)
glue ←λ row . FN(row[1], row[2])
MAP(*glue*, *block*)

Applying this algorithm to `extract_pdf`, the implementation is nearly verbatim and looks like

```
apply(rbind(docs,files), 2,
  function(u) extract_pdf(u[1], u[2])).
```

One thing to notice is the output of `rbind`, which is a character matrix. We'll see in Chapter **??** and 6 how the choice of call depends on the types of the arguments. In some cases it is necessary to use a data.frame or list, respectively, to preserve types. Putting it all together, we now have a function that downloads and saves locally all available situation reports published by the Liberian government.

```
extract_lr ← function(url=.LR_SITE, base=.LR_DIR) {
  page ← scrape(url)
  xpath ← "//div[@id='content']/li/a"
  get_report_link ← function(x)
    grep('SITRep|SitRep', xmlAttrs(x)['href'], value=TRUE)
  links ← xpathSApply(page[[1]], xpath, get_report_link)

  get_file_name ← function(x) gsub('%20', '_', x[length(x)])
  files ← sapply(strsplit(links,'/'), get_file_name)

  url.parts ← parse_url(url)
  urls ← sprintf("%s://%s/%s",
    url.parts$scheme, url.parts$hostname, curlEscape(links))

  apply(cbind(docs,files), 1,
    function(u) extract_pdf(u[1], u[2]))
}
```

This is only the first step in obtaining data and eventually doing an analysis on it. Before working with the dataset in earnest, it is necessary to parse each PDF. This isn't something that R can do natively, so we'll have to make a system call. We may as well do it in our `extract_pdf` since that's its job. System calls are not vectorized, but that doesn't matter here since `extract_pdf` operates on a single document only. In general, it's a good idea to note which functions are vectorized and which ones are not. While it doesn't affect the behavior of a function, it can impact performance and readability. We use the function `pdftohtml` to convert the PDF into an HTML representation. Doing this preserves some document structure, which can then be exploited when calling `sed` to further process the file.

```
extract_pdf ← function(url, file, base) {
```

```
  flog.info("GET %s",url)
  r ← GET(url)
  pdf ← content(r,'raw')
  pdf.dest ← paste(base,file,sep='/')
  writeBin(pdf, pdf.dest)

  txt.dest ← replace_ext(pdf.dest,'txt')
  sed ← paste("sed",
    "-e 's/<page number=\"\\([0-9]*\\)\"[^>]*>/PAGE-\\1/'",
    "-e 's/<[^>]*>//g'",
    "-e 's/[^[:space:]0-9a-zA-Z-]//g'")
  cmd ← "pdftohtml"
  args ← sprintf("-i -xml -stdout '%s' | %s", pdf.dest,sed)
  flog.debug("Call `%s %s`",cmd,args)
  system2(cmd, args, stdout=txt.dest)
  txt.dest
}
```

The first five lines are the same as before. But now we apply two transformations that are connected via a UNIX pipe. One of the replacements that `sed` applies is creating explicit page markers for each page of the PDF. This will simplify the extraction of tables from the text file, since each page is demarcated explicitly. We'll see later that the output is still not very easy to use, but it is better than binary data. There are also a few observations that will simplify the parsing of the file.

For sake of completeness, let's also define `replace_ext`. This function simply replaces the extension of the file name with the one provided. The rationale is that the file name should be preserved for the resulting text file where only the file extension is replaced.

```
replace_ext ← function(file, ext) {
  parts ← strsplit(file, ".", fixed=TRUE)
  sapply(parts, function(x) paste(c(x[-length(x)], ext),
      collapse='.'))
}
```

Since we defined `extract_pdf` as a first-class function, by design it is self contained and easy to move about. Assuming that we don't change the output, we can even modify the function without worrying about how it affects the surrounding code. Although this is true of any function, functional programming's emphasis on functions amplifies its benefit. Echoing the UNIX philosophy, functions typically do one thing well and are implemented with only a few lines of code. This is usually not the case with imperative code and loops, where behavior tends to spread across multiple lines in a single shared scope.

3.2.3 Data normalization

The easiest data to work with is structured, but as any data scientist knows, not all data comes nicely packaged. A data scientist is expected to be self-sufficient, meaning that it is up to her to tame unstructured data into a consistent, normalized structure. Challenges arise when data come from multiple sources, or the format of unstructured data changes. It's therefore essential to have a structured approach to data normalization in order to account for future changes. Normalizing data into a common format is necessarily dependent on the input data, and the delineation between the two roles can be fuzzy. For our purposes we draw the line at the creation of a machine-readable document that can be parsed automatically. For the normalization process to be efficient and scalable, it cannot depend on manual intervention, meaning that all transformations must be compatible with the complete set of files.

Ebola situation reports have a high degree of variability in their structure and content since the reports are created manually. Although we want a plan to parse these files in advance, it can be difficult to understand all the quirks of the raw data prior to actually parsing them. Therefore, it is out of necessity that the data scientist must parse each file incrementally. Over time (and over many iterations), we can expect each desired table in the report to be parsed correctly. What happens to the code during this process? If the parsing logic relies on a lot of conditional blocks, we can expect that their complexity will increase. As each parse error is fixed another conditional block must be added, often embedded within yet another conditional block. The whole thing eventually becomes unmanageable as it becomes exceedingly difficult to understand and update the logic within the hierarchy. Let's think about why this happens. From a structural perspective, if-else blocks form a tree structure, as illustrated in Figure 3.6. In order to determine the rules necessary to parse a particular document, the document 'flows' through the conditional branches until it arrives at some terminal node, which is effectively the aggregate parse configuration. To reduce code, bits of logic are applied at each node, implying that the aggregate configuration is the union of the nodes that the document passes through. In Figure 3.6, the configuration can be described as $config = A \cup C \cup G$.

Trees are useful for organizing data for fast searches, but they are less beneficial when organizing logic. When data is added or removed from a tree, the algorithm governing the behavior of the tree is optimized to do this correctly and efficiently. When writing software, we don't have the luxury of letting a computer execute an algorithm to reorder the conditional blocks. Hence, our poor brains are faced with the task. In an environment where code is built up incrementally, there is a high cost associated with changing a tree structure. Imagine if a document assigned to G doesn't parse correctly. To correct the problem, suppose a new conditional is added to C, labeled J. How do you verify that the new logic is correct? You either have to mentally evaluate $A \cup C \cup J$ and compare it to $A \cup C \cup G$ or run the code against all

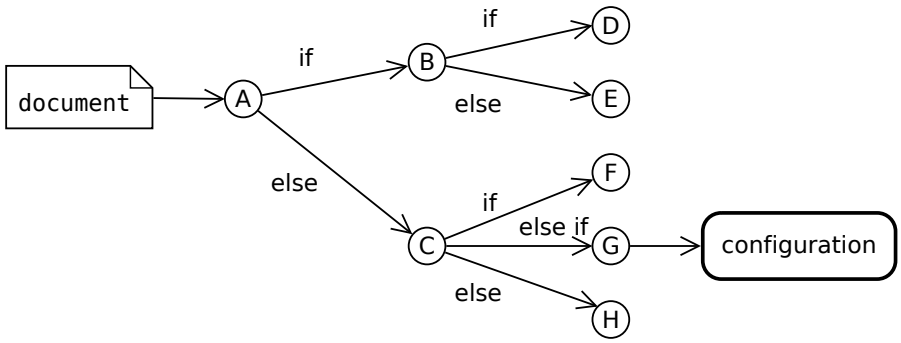


FIGURE 3.6: Modeling conditional blocks as trees

the documents to ensure they end up in the correct terminal condition. It's easy to imagine how this can add unnecessary complexity and work to an otherwise simple problem.

An alternative to a monolithic decision block is to think about the data and configuration as sets. With this approach, normalization is simply a mapping between a document and configuration space to the normalized data space. What do we mean by a configuration space? It's simply a set of variables that fully determines how to parse a table from a document (i.e. the text file). This is similar in concept to how μ and σ^2 fully determine a normal distribution, except we're talking about parsing documents instead of probability distributions. A simple way of achieving this is to partition the set and assign a configuration to each set as in Figure 3.7. Ideally the partitions would be defined prior to parsing the dataset, but realistically this process is incremental. Therefore, it's perfectly reasonable to map all documents to a single configuration space. Then as the variability in situation reports increases, the size of the configuration space increases until it reaches the size of the document space. Understanding this behavior is key in planning out a method for managing the configurations. Notice that on either extremes of this range, the implementation is rather obvious. When all reports have the same configuration, then a single static configuration is sufficient. On the other extreme when there is a one-to-one mapping, it is easy to map each file name to a specific configuration and process accordingly. What happens in between is different. Now we need to encode the mapping between files to configurations and do that in such a way that future configurations can be added to the mapping. Much of this machinery is in the domain of filters that create the actual partition, although an important part of this process is applying the same configuration to a set of reports. This differs from earlier applications of *map* where one function applied to every element of a set.

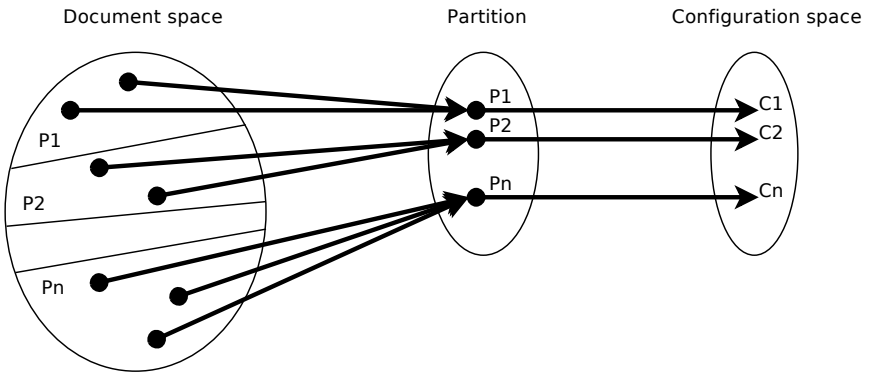


FIGURE 3.7: Mapping a partition to a configuration space simplifies transformations

Now a different function applies to each subset of the partition, where each function returns a valid normalized dataset.

Let's use the ebola data to illustrate how this concept works in practice. The transformed text files are still in a raw state, as seen in Figure 3.8. These files are simply the raw text extracted from the PDF and are the epitome of unstructured data. However, since the PDFs comprise mostly tablular data, the formatting is mostly regular and can be parsed according to rules. These files can be downloaded locally by calling `lr.files ←extract_lr()(*,*)`, defined in Section 3.2.2, which returns the file names of all extracted text files. The files themselves are written to `data/lr`.

Examining file SITRep_204_Dec_5th__2014.pdf, we see that the table on page 3 (seen in Figure 3.2) starts with the unique string 'New Ebola Cases and Deaths Summarized by County', which acts as a marker for this table. The text representation of the table in Figure 3.8 is spread across multiple lines, where each line contains exactly one table cell. Since tables form a grid, we can simply collect each set of lines together to reconstruct a table row. Knowing this about the text file, how can we parse this table without a lot of ceremony? A typical imperative approach would iterate through the file line-by-line, accumulating each row cell-by-cell. This approach may start off simple but will eventually grow out-of-control. A declarative approach treats the file as a set and extracts the relevant subsets. In this case we want all the lines following a county to become the columns associated with the given

```
med
2
dea
ths
 in

communi
ty
De
at
hs
 in

con
fir
med
pa
tie
n
ts
3
Tot
al
Susp

Pr
ob
Tot
al
Susp

Pr
ob
Bomi
2
0
2
0
0
0
0
0
0
Bong
1
1
0
0
```

FIGURE 3.8: Raw extract of PDF situation report. Numerous words are split across lines due to formatting in the source PDF.

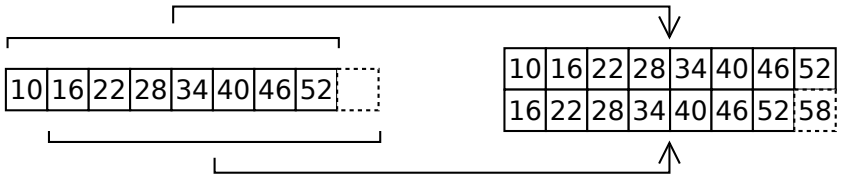


FIGURE 3.9: Constructing table boundaries by shifting a vector. The first $n - 1$ indices represent the start index, while the last n-1 indices are used as corresponding stop indices.

county. This hints at one of the elements of the configuration space, which are names of each county. It also means that the next county acts as a boundary marker. For example, the row containing `Bomi` spans 10 lines (including itself), ending at the line containing `Bong`. Translating this observation into a function requires a modified use of the *zip* paradigm we discussed earlier. The idea is to construct the bounds for each table row based on the line numbers in the file. We can do this easily by zipping a shifted version of the indices with itself, as shown in Figure 3.9. The result is a matrix where each row represents an interval defining the start and end lines for each county. The construction of the matrix is performed using either `cbind` or `rbind`.

Given a vector `x`, the general approach is `cbind(x[-length(x)], x[-1])` . Since the county markers only represent the starting points, this approach isn't quite right. To identify the last line of a table row, we need to know the line number marking the start of the next row. This works fine until the last row. Not all is lost, though. As tabular data, we can assume that each section has the same number of lines, so the final end marker can be appended prior to constructing the matrix of bounds.

Now that the basic strategy is in place, we can codify it in Algorithm 3.2.4. Starting with a subset of the text file that represents a single page, the algorithm only has a few steps. In addition to the raw lines, the function also takes a vector of markers that indicate the beginning of a row and also a vector of column labels for the resultant data.frame.

Algorithm 3.2.4: GetRows(*lines*, *markers*, *labels*)

$x \leftarrow$ GetIndex(*lines*, *markers*)
$x[length(x) + 1] \leftarrow x[2] - x[1]$
$bounds \leftarrow$ zip$(x[-1], x[2 : length(x)])$
$df \leftarrow$ map$(\lambda$ b . GetSeries(lines,b[1],b[2]), bounds)
$colnames(df) \leftarrow labels$
return (df)

The algorithm calls two other functions that we'll define shortly. First, consider this approach. Up to now, we've discussed how declarative code (typically) avoids the messy process of working with indices, yet here we are doing just that. This is a hint that this algorithm can be improved. Manipulating the indices is actually symptomatic of something worse. What's unsavory about this approach is that we're manually modifying data generated by some other process. Other processes might not know that we've appended a pseudo index to the vector, which can cause subtle problems later in the processing. These sorts of issues arise when a text file is modified after the indices are computed.

An alternative approach makes use of a simple observation. We know in advance the names of the columns, so we already know how many lines to grab for each table row. With this approach we don't need the index of the stop marker and can instead pass a constant `size` parameter. Doing so returns us to familiar land where only a single variable varies.

```
fn ← function(lines, markers, labels) {
  size ← length(labels)
  indices ← get_index(lines, markers)
  rows ← lapply(indices, function(i) get_series(lines,i,size))
  df ← as.data.frame(do.call(rbind, rows))
  colnames(df) ← labels
  df
}
sapply(markers, function(m) fn(lines,m,columns))
```

There's no hard and fast rule about when to use one approach versus another. That said, direct manipulation of indices generally has a higher cost than methods that don't. Either way we've managed to parse a PDF table into an Rdata.frame.

Before moving to the next step, let's visit the `get_index` and `get_series` functions. The first function, `get_index`, returns the line numbers associated with the set of markers, which are the county names. The `do.call` trick is used since there's no guarantee that every row exists in a table.

```
get_index ← function(raw.table, markers) {
  section.idx ← lapply(markers,
    function(x) grep(sprintf('^%s$',x), raw.table))
  do.call(c, section.idx)
}
```

The `get_series` function extracts the actual data series from the vector of lines, converting each value into numeric data. This is where we use the size argument to select the actual range of the data. Note also that we skip the row containing the marker since we already have it.

```
get_series ← function(raw.table, index, size) {
  chunk ← raw.table[(index+1):(index+size)]
  as.numeric(gsub('[^0-9]','', chunk))
}
```

The next step involves parsing this table from each document. We know that depending on the document, there are different markers that must be used. Examining the various situation reports it turns out that a specific table can appear on different pages, and the number of columns may change over time. Each table must therefore be separated from the other tables, or the data will be corrupt. This tells us that the structure of the configuration has two parts: first are the markers to delineate each section, and second are the markers for each table. The requirement is that each section marker must be unique in the document. Within a section, the row markers must be unique but can reappear in other sections.

Given the above analysis, we conclude that our configuration space consists of four variables per section. These are the section start and stop markers, the table row markers, and the column names. Within a specific partition by definition the configuration space is constant. This implies that to parse this table from multiple situation reports, we simply need to pass the appropriate set of text to the function. There are two approaches to this. First up is Algorithm 3.2.5, where we can *map* over a vector of filenames, read the file, extract the lines, then call `get_rows`.

Algorithm 3.2.5: GetTable(f)

($start$, end, $markers$, $labels$) = GetConfiguration(f)
GetRows(GetChunk(ReadLines(f), $start$, end), $markers$, $labels$)

This is a compact approach and can seem efficient, but it's important not to conflate these two properties. Consider an incremental development process where a number of iterations is required to get the markers just right. This happens often during testing and debugging. If it's necessary to run the code numerous times, the compact syntax is actually a detriment because the source data must be downloaded each time the parse process is executed. This adds unnecessary latency to an otherwise efficient process. An alternative approach is to first *map* over the file names to extract the lines (Algorithm 3.2.6), then *map* over the set of lines (Algorithm 3.2.7, calling `get_rows` for each vector of lines.

Algorithm 3.2.6: GetRawLines(f)

($start$, end, $markers$, $labels$) = GetConfiguration(f)
GetChunk(ReadLines(f), $start$, end)

Algorithm 3.2.7: GetTable(ls, f)

($start$, end, $markers$, $labels$) = GetConfiguration(()f)
GetRows(ls, $markers$, $labels$)

The benefit of the second approach is separation of concerns. Not only

	county	alive.total	alive.suspect	dead.total
209	Bomi	2	0	0
210	Bong	1	1	0
211	Gbarpolu	1	1	0
212	Grand Bassa	5	3	0
213	Grand Cape Mount	0	0	7
214	Grand Gedeh	0	0	0
215	Grand Kru	0	0	0
216	Lofa	0	0	0
217	Margibi	1	0	0
218	Maryland	0	0	0
219	Montserrado	19	3	3
220	National	34	13	11
221	Nimba	5	5	1
222	River Cess	0	0	0
223	River Gee	0	0	0
224	Sinoe	0	0	0

FIGURE 3.10: Parsed data.frame from unstructured text

does this make each function easier to understand, it can also speed up development and save computing/network resources. Although reading from a file is relatively cheap, reading from a database or web service can be significantly more costly. By separating the extraction logic with the parse logic, it's possible to minimize calls to external resources despite multiple calls to the parse logic. During development and troubleshooting, the ability to repeatedly call local functions without having to wait for data transfer can be extremely useful.

The result of this parse process is a data.frame containing a date, plus all the parsed columns of the table. Figure 3.10 shows a subset of the result from parsing the New Ebola Cases and Deaths Summarized by County table (shown in Figure 3.2). Reflecting back on the code used to generate this data.frame, not a single loop nor conditional block was used. Hence the control flow is strictly linear, making the code very easy to understand as well as increasing the usability of the constituent functions.

Configuration spaces are appropriate for any process where the number of configurations is between a one-to-one mapping and a constant configuration. In Chapter 5 well discuss in detail how to construct the partition that maps to the configuration space using filters and set operations.

3.3 Validation

After parsing the situation reports and extracting each table, it's time to start analyzing data, right? Well, actually, no. Even though we've parsed the data, we have no guarantee that the data was parsed properly. If we were cavalier we might throw caution to the wind and start our analysis. But as data *scientists* we know to first verify that the data is accurate before using it. Indeed, at this point there are two distinct sources of error that can be present. In addition to errors from the parse process, the source data itself could contain errors. The ideal case is to be able to check for both types of errors.

3.3.1 Internal consistency

While the most thorough validation is comparing the parsed output with the source PDF, it's too manual (and itself error prone) to be practical. Instead, we need an automated approach to validating the data. In general it's difficult to automate the parse process in absolute terms, since the point of parsing is to produce a machine-readable document. One appoach is to have a second, independent parse process to compare against the first. In some contexts this can be justified, but it's a costly exercise that doesn't guarantee perfect results. Another approach is to focus on the internal consistency of the data. These types of validations rely solely on the parsed data. The ebola data is actually easy to test in this regard since each table includes a row for the national totals. This means that for each column of a table, the sum of the individual counties must equal the national value. If not, then either the source data is bad, or more realistically, the parse process has an error.

Given a table of data, how do we implement the above consistency check? For each column, the check is simply $\sum_i measure_i = national,\ i \in counties$. To validate the whole table, we can iterate across each column with a *map* process. The implementation follows the same two-step procedure we've seen throughout this chapter: 1) define a first-class function that implements the logic for a single case, and 2) perform a *map* operation over all columns.

```
check_total ← function(df, col, nation='National') {
  national ← df[df$county==nation,col])
  counties ← sum(df[df$county != nation,col])
  national == counties
}
sapply(colnames(df), function(col) check_total(df, col))
```

While internal consistency is a good approach to validating the correctness of data, it is not fool-proof. There are numerous ways to have consistent data that passes the validation but is incorrect. Here are two simple scenarios. First suppose that you decide to use a zero whenever a parse error is encountered.

Ebola Case and Death Summary by County

County	DAILY EMAIL REPORT New[1] suspected and probable cases (Alive and Dead)			Laboratory[2] Confirmed Cases (Alive and Dead)	VHF DATABASE Cumulative[3] cases 23 May-December 5th 2014				Cumulative deaths[2]
	Total	Suspect	Probable		Total	Suspect	Probable	Confirmed	
Bomi	2	0	2	0	293	83	75	135	158
Bong	1	1	0	0	557	390	32	135	124
Gbarpolu	1	1	0	0	38	25	3	10	10
Grand Bassa↑	5	3	2	0	156	41	75	40	65
Grand Cape Mount↑↑	7	6	1	0	153	47	47	59	76
Grand Gedeh	0	0	0	0	10	8	0	2	5
Grand Kru	0	0	0	0	36	14	18	4	27
Lofa	0	0	0	0	645	171	148	326	385
Margibi	1	0	1	1	1254	415	474	365	550
Maryland	0	0	0	0	21	15	2	4	17
Montserrado↑	22	6	16	3	4149	1733	786	1630	1688
Nimba↑↑	6	6	0	0	322	80	128	114	53
River Gee	0	0	0	0	19	7	5	7	8
River Cess	0	0	0	0	41	9	12	20	26
Sinoe	0	0	0	0	37	18	3	16	14
NATIONAL	45	23	22	4	7731	3056	1808	2867	3206

[1] From daily email county reports of aggregated data for that day
[2]Laboratory confirmed cases of suspects and probable cases identified during the preceding days
[3]From individual-level data from the Case Investigation form; cases may be reclassified according to surveillance case definitions
↑Increase in daily reported cases by county

FIGURE 3.11: A table in the Liberia Situation Report containing cumulative counts

What happens when the whole column fails to parse? Then you have all zeros, which satisfies internal consistency but is wrong. Another common example is that due to a parse error the data is shifted so that columns are misaligned with their labels. Again for a given column the data are internally consistent but is still wrong. These scenarios underscore the importance of combining internal consistency with other types of checks when working with parsed data.

In addition to raw counts, the situation reports also contain cumulative counts. These data points give us another axis for evaluating internal consistency. The observation here is that cumulative counts are necessarily monotonic. Therefore, across multiple situation reports any cumulative value must be greater than (or equal to) the value in a prior date. If both the raw and cumulative counts are available then the consistency of each value can be tested from day-to-day, assuming all reports exist.

Let's see how this works. The column `cum.dead.total` lists the cumulative number of deaths due to ebola, which is the last column in Figure 3.11. Since the column represents cumulative counts, each successive date must have a value greater than (or equal to) all previous dates. If not, then either the source data is wrong or the parse process has errors. For this validation

we need both a column and a county. Here we assume that the data.frame representing the case and death summary table contains multiple dates. First we'll test that the final column value (in other words, the most recent datum) is greater than (or equal to) all others. For a sequence x we can express this as $\bigcap_i^{n-1} x_i \leq x_n$, where $|x| = n$. Here we can take advantage of built-in vectorization of logical operators and write this as `all(x[-length(x)] < x[length(x)])`. We can apply this technique to our data.frame verbatim so long as the sequence is properly ordered, as in `x ←df[order(df$date),col]`.

The next step is applying this to all columns containing cumulative data, which is left as an exercise for the reader.

3.3.2 Spot checks

At times it is necessary to explicitly verify the value of a table cell. These spot checks can add a degree of comfort to parsed data since values are visually confirmed by a human to be the same. The cost of human comfort is of course its manual nature, which is both time consuming and fallible. What would be nice is to take advantage of spot checks but without human involvement. For this to be effective we need to think about what makes a spot check useful. Clearly they need to offer some benefit that we can't get from internal consistency. One such benefit is verifying column alignment. We already know that this is a weakness of internal consistency checks, so using spot checks for this purpose seems like a good idea. Does it matter which column to use? The answer depends on the structure of the raw data. In this case, it matters since cells are read along rows, one cell at a time. Performing a spot check on a column in the upper left of the table is less effective than a cell in the lower right. The reasoning is that any non-fatal parse errors will propagate through the table as the parse process progresses. Therefore, a shift in a column will affect all columms to its right as well. It's also important to use unique values that don't appear elsewhere in the table to avoid false negatives. For example, confirming that the number of probable cases in Sinoe is zero is a poor choice, since any number of parse errors could result in a zero. However, checking the number of probable cases in Montserrado is a good choice, since 16 doesn't appear anywhere else in the table.

The structure of a spot check is fairly simple and contains similar information as a unit test. The primary difference is that spot checks have a regular structure meaning that we can organize the data in a data.frame. Each row represents a spot check and is described by 1) a county, 2) a measure, 3) the expected value. Following the strategy above we can add a few rows that describe what we expect from the summary of cases and deaths.

```
county ← c('Montserrado', 'Grand Cape Mount', 'National')
measure ← c('alive.probable', 'dead.total', 'alive.total')
expected ← c(16, 7, 34)
```

```
spot.checks ← data.frame(county, measure, expected)
```

A simple approach to test this is to map along the rows of the data.frame. The signature of the function applied to each element looks like $fn(county, measure, expected) \rightarrow logical$ and can be implemented as a set operation:

```
fn ← function(county, measure, expected) {
  df[df$county == county, measure] == expected
}.
```

While this is functionally correct, the trouble with this approach is that the actual value in the table is opaque. If there are errors (and there will be!), it's difficult to quickly determine the source of the error when the actual value is not returned. It's better to split the procedure into two steps: collect actual table values first, and then compare. The first step is simply the first half of the previous function, namely

```
df[df$county == county, measure].
```

Now we can collect the values and attach it to the spot check data.frame:

```
fn ← function(row) df[df$county==row[1], row[2]]
spot.checks$actual ← apply(spot.checks,1, fn).
```

Finally, the actual verification uses a vectorized equality test, `spot.checks$expected == spot.checks$actual`, to compare each individual spot check.

In this section we've shown how data validation can be implemented as a two step process. First is defining the validation rules for a given column, or data series. Second is identifying the broader set of data series for which the validation is applicable. Applying the validation to each series is simply a call to *map*.

3.4 Preservation of cardinality

When manipulating or analyzing data it is generally useful to know how large a particular set is. This quantity is known as cardinality and specifies the number of elements within a set. For higher order functions we're interested in the relationship between the cardinality of the input and the output. Knowing what happens to the cardinality of the data is important and allows us to deduce certain properties of the data. *map* operations are special in the sense that cardinality is always preserved under the operation (Proposition 3.4.1). Hence, if a vector x has length $|x| = 10$, then for any scalar function f, $map\, f\, x$ also has length 10.

Proposition 3.4.1. *Let $f : A \rightarrow B$ be a scalar function. Given $X \subseteq A$, where $|X| = n$, $|map\, f\, x| = n$.*

In order to satisfy this requirement, there are certain consequences to the way *map* implementationas work. Specifically, `sapply` changes its output data structure depending on the result of the scalar function passed to it. To the novice, the behavior may seem inconsistent, but it can be fully explained by the need to preserve cardinality. Why is preservatin of cardinality so important? The answer requires thinking about functions as relations and reconciling the difference between mathematical functions and programming functions.

3.4.1 Functions as relations

Classical mathematics defines functions as a black box taking some input and returning some output. It doesn't really matter what the input or output are, so long as they both exist. Functions are also only allowed to return one value per unique input. This simple model of a function can be generalized as a *relation* between two values: the input and the output. For example, the function $f(x) = x^2$ is a relation that maps $\mathbb{R}$ to $\mathbb{R}^+$. The pair $(2, 4)$ is in f but $(3, 0)$ is not. A consequence of this definition is that a function must always return a value for a given input. This is subtly different from above, where the concern is typically limiting the upper bound on the number of values a function can provide. The so-called vertical line test is often used for this purpose. This test operates on the graph of a function, so we're already assuming that a value exists. But what if no output value exists? As a relation, there must be *exactly* one value returned per input. If there is no valid result, then the function is just not defined for the given input. Indeed, this is how division by zero is treated: the operation is simply not defined.

From a programming language perspective, the mathematical constraint on functions doesn't exist, and it's perfectly legal to define a function with no return value. This happens often with I/O operations, such as writing data to a database, a file, or a display. Now data science is at the nexus of math and programming, so how do we reconcile these two views of a function? If we take a formal stance we would simply say that such a function is not compatible with mathematics and should be ignored. However in the real world these functions do exist and we cannot categorically ignore them. To arrive at a more palatable answer, let's consider what it means for a function to not have a return value. For many programming languages, there is a specific keyword indicating the value to use as the function return value. If no return value is specfied, then the function returns the equivalent of a `NULL` value. In R, functions return the value of the last expression, but there is a way to define a body-less function.

```
> f ← function(x) { }
> f()
NULL
```

This interpretation may seem slight, but there is value to this approach. It's

reasonable to interpret no return value as being equivalent to returning the empty set. This has interesting implications starting with the idea that a function with no return value is indistinguishable from a constant function that maps all values to $\emptyset$. Mathematically, the function can be described as $f : A \to \emptyset$, and the pairs defining the relation look like $(a, \emptyset)$, where $a \in A$. While this may seem strange, it's perfectly legal, since the only requirement mathematically is to return *some* value, even if that value is the empty set!

The situation is not as dire as we thought, now that we've successfully shown that programming functions can be represented mathematically as relations. This means that for any input set, we can compute the graph of the input and the output will have the same length. Another way of saying this is that the cardinality of the input is preserved in the output. Since *map* provides the machinery to compute the graph of a function, then it's natural to expect *map* to preserve cardinality. An obvious example of this property is linear algebra, where vector addition preserves cardinality. We also saw this property throughout the chapter as we transformed text data into numeric values. Knowing that *map*-vectorized operations preserve cardinality, how does it affect our understanding of *map* operations?

3.4.2 Demystifying `sapply`

One important finding is that preservation of cardinality explains the behavior of the *map* implementation `sapply`. For newcomers `sapply` can appear to have inconsistent behavior, sometimes returning a vector, sometimes a list, and other times a matrix. Why does it do this, and is it possible to deduce the type of the result? Let's see if our toy function $f(x) = \emptyset$ can shed light on the matter. Suppose we want to compute the graph of f over $(1, 2, 3)$. What is the result? Based on our earlier analysis, the answer is $(\emptyset, \emptyset, \emptyset)$. What is the answer in R? Technically this function is natively vectorized so it can be called as `f(1:3)`, which results in `NULL`. What happened? Shouldn't the result be `c(NULL, NULL, NULL)`? We know from Chapter 2 that vectors treat the empty set different from other values. No matter how many `NULL` we try to concatenate, the result will always be a single `NULL`. This can be problematic if we *map* over a function that might return a `NULL` value. To preserve the `NULL`s requires returning a data type that supports holding multiple `NULL` values, such as a list. How does this reasoning compare with what `sapply` does? Let's see:

```
> f ← function(x) c()
> sapply(1:3, f)
[[1]]
NULL

[[2]]
NULL
```

```
[[3]]
NULL
```

Lo, the result is a list of `NULL`s! Therefore, when using `sapply`, cardinality is in fact preserved. This implies that whenever the result contains a `NULL` value, the result type will be a list and not a vector. [6] This same logic explains why a matrix is returned for some functions.

Example 3.4.1. In addition to `NULL`s, the same behavior holds for typed vectors of zero length, such as `character()`. Conceptually these can be treated as $\emptyset$ with type information included. An empty character vector is different from an empty string. The difference is that an empty string is a proper value and thus survives concatenation. For example,

```
> c("text", character(), "")
[1] "text" ""
```

Let's revisit algorithm 3.2.1 and consider the effect of a function that returns `character()`. To preserve the output cardinality of the `xpathSApply` call, we know that the result cannot be a vector. Therefore a list must be returned in order to preserve the cardinality and the `character()` values. We can verify this claim by printing a portion of the variable `links`.

```
> links[14:16]
[[1]]
                                                                    href

"http://mohsw.gov.lr/documents/SITRep%20143%20Oct%205th,
    %202014.pdf"

[[2]]
named character(0)

[[3]]
                                                                    href

"http://mohsw.gov.lr/documents/SITRep%20142%20Oct%204th,
    %202014.pdf"
```

Calling `do.call(c,links)` removes the empty character values leaving a character vector of valid web paths.

□

3.4.3 Computing cardinality

Knowing that *map* processes preserve cardinality, is it possible to deduce the cardinality over a sequence of operations? Combining our knowledge of

[6] The documentation for `sapply` says that if the result cannot be simplified, then it will remain a list.

recycling with preservation of cardinality, it's not only possible but easy to deduce this by simply looking at the code. The benefit is that the data scientist can establish a priori the expectation for the output of a (deterministic) function, which can then drive tests in the code. Furthermore, development can be streamlined since it is now trivial to ensure cardinality is preserved where you want it.

Univariate functions are the easiest to analyze in terms of cardinality. Ultimately there are those functions that preserve cardinality and those that don't. *Map*-vectorized functions preserve cardinality, while $fold$ (Chapter 4) and $filter$ operations (Chapter 5) do not. What about scalar functions? By definition scalar functions accept and return scalar values, so cardinality is preserved. Again, it's important to remember that we mean this in the formal sense. The function `f(x)= rnorm(x, 5)` is not a scalar function, so these rules do not apply.

For binary operations, the cardinality of the result is dictated by the cardinality of both its operands.

Definition 3.4.2. Let $\circ$ be a binary operation and let x, y be vectors with length $|x|$ and $|y|$, respectively. The cardinality of $x \circ y$ is defined as

$$|f \circ g| \equiv |f| \times |g| = \begin{cases} |x|, & \text{when } |y| = 1 \\ |x|, & \text{when } |y| \mid |x| \\ undefined, & \text{when } |y| \nmid |x| \end{cases}.$$

To improve readability, the $\times$ operator is defined to be left-associative. Hence, $|a| \times |b| \times |c| = (|a| \times |b|) \times |c|$.

For the most part, this definition combines the behavior of native vectorized functions and recycling [**?**]. When the lengths of two vector operands are not equal but the length of the longer vector is an integer multiple of the shorter vector, the elements of the shorter vector are reused until the two lengths are the same. This recycling procedure happens naturally in linear algebra.

Example 3.4.2. The cardinality rules for binary operations can be generalized to specific classes of functions, such as polynomials. Let $f(x) = x^3 + 2x^2 - 5$. What is the cardinality of f?

$$\begin{aligned} |f| &= |x^3 + 2x^2 - 5| \\ &= |((x^3) + (2 * (x^2))) - 5| \\ &= ((|x| \times |3|) \times (|2| \times (|x| \times |2|))) \times |5| \\ &= (|x| \times (1 \times |x|)) \times 1 \\ &= |x| \times |x| \times 1 \\ &= |x| \end{aligned}$$

□

Example 3.4.3. Returning to the Z-score example, given a vector x, the final cardinality is $|x|$. By breaking down the operation, we can prove that this is true.

$$\begin{aligned} |Z| &= |(xs - mean(xs))/sd(xs)| \\ &= |xs| \times |mean(xs)| \times |sd(xs)| \\ &= |x| \times 1 \times 1 \\ &= |x| \end{aligned}$$

□

These rules can generally be extended to multivariate functions with an arbitrary argument list. However, as we'll see in Section **??**, some function arguments do not contribute to cardinality. It's up to the data scientist to determine which arguments need to be analyzed.

3.4.4 Idempotency of vectorized functions

Since Ris vectorized, many functions are natively vectorized. Similarly, functions built from vectorized functions using vectorized operations are themselves vectorized. These functions clearly do not need the help of *map* to transform them into vectorized functions. What happens if we apply *map* to the function anyway? Let's start by analyzing the polynomial function in Example 3.4.2. What is *map f x* for $x = (2, 3, 4)$? Evaluating `sapply(2:4,f)` yields $(11, 40, 91)$, which is no different from `f(2:4)`. This result holds for a broad range of functions.

Proposition 3.4.3. *Let $f : A \to B$ be a map-vectorized function. If $\emptyset \notin B$, then map f x = f x for $x \subseteq A$. In other words, f is idempotent under map.*

This property may seem inconsequential, but this is precisely what makes it useful. When building a model or writing an application, it's easy to lose track of which functions are vectorized and which ones are not. This property assures us that there is no harm in applying *map* to a vectorized function. During development, this can occur when evaluating one model versus another or one implementation versus another. If a function f is vectorized while another g isn't, it can be a hassle switching between `f(x)` and `sapply(x,g)`. Since f is idempotent under *map*, there's no need to change the call syntax. Instead `sapply` can be used for both functions.

While applying map to a vectorized function is inconsequential semantically, there are other costs associated with this practice. From a performance perspective there is a non-trivial difference. In general, natively vectorized operations are going to be faster than those performed by a *map* operation.

An informal comparison can be made by using `system.time` to measure how long it takes to execute the function. For a more representative measurement, we'll want to do this a number of times. Thus, even this instrumentation process can be implemented as a *map* operation. Note that in this case we *map* along a sequence of indices. These have no meaning in the function, but as with output values, there must be an input value passed to the function.

```
> t1 ← t(sapply(1:1000, function(i) system.time(f(-9999:10000)
    )))
> colMeans(t1)
 user.self   sys.self    elapsed user.child  sys.child
  0.001027   0.000011   0.001042   0.000000   0.000000

> t2 ← t(sapply(1:1000, function(i) system.time(sapply
    (-9999:10000, f))))
> colMeans(t2)
 user.self   sys.self    elapsed user.child  sys.child
  0.068611   0.000430   0.069740   0.000000   0.000000
```

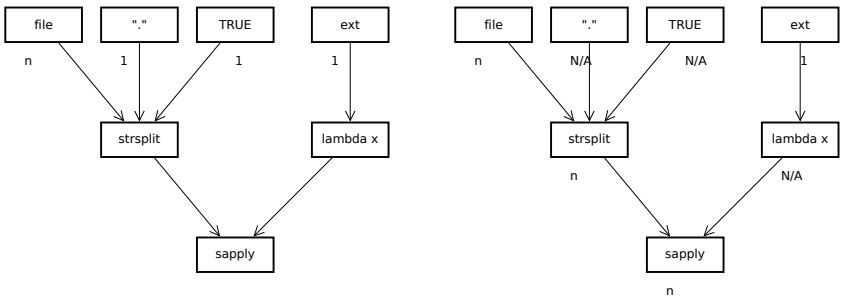
This result shows just how much slower a *map* operation is. During development convenience is often more important than performance, so this isn't a big concern. As the code transitions to production and the size of the datasets increase, then it is necessary to apply these sorts of optimizations as appropriate.

3.4.5 Identifying *map* operations

As a codebase grows, it's not always immediately obvious which functions are vectorized and which ones are not. Since vectorized functions are idempotent under *map*, we already know that one trick is to go ahead and apply *map* to these functions. Falling back on this behavior is fine for development, but at some point we actually need to know what the behavior of the functions we depend on is. Its more than good housekeeping. If we want to reason about the behavior of the system, we need to understand this aspect of our functions. For example, can vectors be passed to `replace_ext`? For convenience, the definition is provided once more.

```
replace_ext ← function(file, ext) {
  parts ← strsplit(file, ".", fixed=TRUE)
  sapply(parts, function(x) paste(c(x[-length(x)], ext),
      collapse='.'))
}
```

Looking at this function it's not clear what will happen if a vector is passed into this function. What's the best way to determine if the function is vectorized? We can take advantage of the same method we used to compute cardinality to determine if a function is vectorized. To do this we need to be careful of how we think about cardinality. A complication arises when dealing with structures that can contain elements that have cardinality or are matrices.



(a) Initial model of `replace_ext` as a graph. Arrows represent flow of arguments and is reversed from a dependency graph.

(b) Graph with cardinality updated based on arguments that make meaningful contributions.

The `strsplit` function is an example of this since the result is a list and each element of the list is a vector. What this implies is that it's necessary to keep track of the cardinality of each variable to determine the final cardinality. For complex functions this can become challenging, but keep in mind that functions should typically be short. Furthermore, when functions are written without loops and conditional branches, functions are essentially one long sequence of function composition. This makes it easy to trace the cardinality of the function through the body of the code. The `replace_ext` function is designed this way and can be rewritten as a chain of function compositions by removing the assignment of the intermediate variable `parts`.

The first step is to specify the cardinality of the function arguments, which is essentially the function's inital condition. In this case, $|file| = n$, and $|ext| = 1$. Next is to determine the cardinality of `strsplit`. The documentation tells us that it takes a character vector and returns a list of results []. So `strsplit` is vectorized, meaning that its cardinality is $|strsplit(file, ".", fixed = TRUE)| = n$. The next line is just a call to `sapply`, and we know that the cardinality is preserved, so $|replace\_ext| = n$.

Notice that we didn't even bother checking the cardinality of the closure passed to `sapply`. This function is actually equivalent to a *fold* operation, but since it's being called by *map*, we already know that the cardinality of the output will be the same as the input. What we don't know is what data structure the output will have. For this we need to consider the return value. In this case, the result is simply a scalar, so `sapply` will return a character vector. We'll see in Chapters 6 and **??** how this can change depending on the return type of the closure passed to `sapply`.

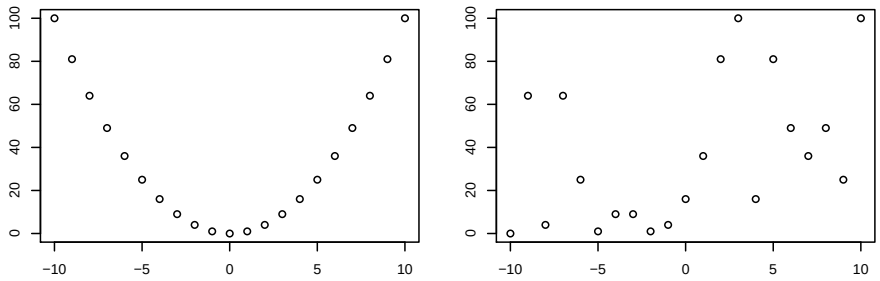
We mentioned that `replace_ext` is essentially an application of function compositions. Using symbolic notation, `replace_ext` can be rewritten as $sapply \circ strsplit$. These chains can be arbitrarily long, representing a complex computation. If a function can be described in this way, it can be modeled as a

directed graph. Visualizing the function calls as a graph can aid in analyzing the cardinality of the function. The first step is plotting each dependency and annotating it with its cardinality. Figure **??** illustrates such a graph using the `replace_ext` example. It's okay if the cardinality is not exactly known, as the values can remain unbound using $|x|$ notation. We still need to know how a function uses each argument, since not all arguments are treated the same. This is certainly the case with `strsplit` where the second and third arguments do not affect the cardinality of the result. These arguments can be excised from or marked as impotent in the graph. At this point it's simply a matter of applying the cardinality rules to the operations. The final result is the cardinality of the output.

3.5 Order invariance

Preserving cardinality on its own doesn't seem particularly special, so why bother? Indeed, for functions like `sapply` it's annoying worrying about an output that changes and appears inconsistent. The value of cardinality preservation is realized when coupled with order invariance. Not only is cardinality preserved, but the output will be in the same order as the input. We actually hinted at this in Section 3.4.1 when discussing the input/output pairs of a relation. When thinking about sets, order doesn't matter, but for vectors, tuples, and sequences, order is critical. In other areas of mathematics, order is implied and taken for granted. Order is important when creating the graph of a function. Points are plotted in input/output pairs as dictated by the relation. The axes are similarly ordered, with values plotted on the graph according to the value of the independent variable. These input/output pairs must be kept together, otherwise the resultant graph would be garbage. To speak of breaking the ordered pairs of a relation is mathematical heresy, but in programming terms there is no such religion. If we desired, it's perfectly legal to return the values of a vectorized function in any order we want. At first glance this may seem ridiculous, but traditionally there are a class of data structures that have *no defined order*. These data structures are variously known as dictionaries, hash tables/maps, or in R, environments. Again we are presented with two views of the world that need to be reconciled. Figure 3.13 compares the two situations for our simple transformation: $f(x) = x^2$. Clearly Figure 3.13b has less information content than **??**, which underscores the importance of maintaining order across a computation. Most mathematical functions thus have an implicit pairing of values that preserves order through a computation. We call this property order invariance.

Definition 3.5.1. Let $f : X \to Y$ and $A \subseteq x$, $B \subseteq Y$. If $f[A] \to B$ such that $|B| = |A|$ and $b_i = f(a_i)$ for all $a_i \in A, b_i \in B$, then f is *order invariant*.



(a) Plotting a graph assumes a consistent ordering between the input and output values

(b) Decoupling the pairs of the relation results in a random graph

FIGURE 3.13: Comparing two 'graphs' of the same function

Order invariance is the result of mapping an ordered set of values to a sequence. This sequence is better known as the index of the vector. By making this process explicit, the ordinals can actually be reused. This property is useful when working with various sets that all have the same ordering. This property is exploited whenever a set operation is performed, which we'll discuss at length in Chapter 5. It also becomes handy when appending vectors to data frames (Chapter **??**) since each column is assumed to have the same ordering. We used both of these techniques in Section **??** when automating a spot check on the parsed data.

Comparing Definition 3.5.1 with the definition of *map*, we can see that by definition, *map* operations are order invariant. Hence, the order of the output will be the same as its input.

Proposition 3.5.2. *The higher order function map is order invariant.*

Returning to the spot checks in Section **??**, the final operation was

```
spot.checks$actual ← apply(spot.checks,1, fn).
```

Each row represents a specific test with an expected value. The *map* operation appends a new column containing the actual value in the specified cell. Hence the ordering is defined by the rows of `df`, whatever they are. Calling `apply` preserves the ordering and returns the value in the same order. Notice that we aren't concerned with the actual order of the records. What matters is that the order is invariant under *map*. We can convince ourselves of this by starting with a different order and then applying *map*.

```
> spot.checks.a ← spot.checks[sample(1:nrow(spot.checks),nrow(
    spot.checks)),]
> spot.checks.a$actual ← apply(spot.checks.a,1, fn).
> spot.checks.a
            county        measure expected actual
```

```
2 Grand Cape Mount     dead.total        7      7
1      Montserrado alive.probable       16     16
3         National    alive.total       34     34
```

What happens to the ordinality if cardinality is not preserved? To understand the effect on ordinality, consider a cartoon function

$$f(x) = \begin{cases} x^2 & if x is even \\ \emptyset & otherwise' \end{cases}$$

which behaves like $f(x) = x^2$, except that all odd values are not computed. Suppose we want to compute the graph of f over $x = (1,2,3,4,5,6)$. Hypothetically, we expect the result to be $y = (4,16,36)$, since the rules of concatenation will remove all instances of $\emptyset$. Consequently, the ordering is lost as x_2 now maps to y_1, x_4 maps to y_2, and so forth, as illustrated in Figure **??**. [7] Mathematically the effect of this function is to map values out of the range completely! If we make further calculations and want to compare back to x, it becomes a burden to recover the mapping between the two sets of indices.

A more practical example can be found in the ebola data. When parsing the rows of a table, what happens if there are fewer columns than expected? Novice Rusers coming from other languages might be tempted to return a `NULL` value. Using this approach can result in a loss of cardinality, and consequently, a loss of ordinality. In Section 3.2.3 we defined the function `get_series` that parses a row of data. The last line of this function is

```
as.numeric(gsub(   [ 0 -9]   ,      , chunk)).
```

Our implementation removes all non-numeric characters prior to parsing a numeric value. If we were cavalier and didn't include this, what would be the result of `as.numeric` if it tried parsing non-numeric data? Suppose that the chunk looks like `c('23','34','12a')`.

```
> as.numeric(chunk)
[1] 23 34 NA
Warning message:
NAs introduced by coercion
```

So `as.numeric` preserves cardinality by using `NA` as a placeholder for unparseable values. Most functions that ship with Rbehave this way. User-defined code might not be so reliable. Let's see what happens if we try to parse the function returns `NULL`s instead of `NA`s. This can happen if we check the value of cells first and carelessly choose to return a `NULL` value as a default value. It's quite plausible that we use the `do.call` trick to indiscriminately collapse the output.

```
> fn ← function(x) if (grepl('[a-zA-Z]', x)) NULL else
    as.numeric(x)
```

[7] Note that the ordering is still monotonic but is not complete since elements are removed. In Chapter 5 we'll revisit this behavior.

```
> do.call(c, lapply(chunk, fn))
[1] 23 34
```

The result is as expected. The output cardinality is different from the input cardinality. When cardinality isn't preserved, there's no way for us to know the ordering of the result. In other words, we cannot identify the field containing the bad parse value without manually comparing the input and the output. A good rule of thumb, therefore, is to return `NA` if it's necessary to preserve order and cardinality.

The lesson here is that ordering is a key property that we often take for granted. We've seen how ordering plays an important role when working with data structures. In Chapter **??**, we'll see that *fold* operations are not guaranteed to be order invariant, which makes it difficult to know what effect specific inputs have on the end result. These two properties are what utimately makes *map* so powerful. By using *map*, functions that go beyond built-in operations can retain declarative vector semantics.

3.6 Function composition

Our discussion of *map* properties has so far been limited to single functions, such as $f(x) = x^2$. What can we say about these properties for more complex structures? From our discussion on computing cardinality, it's clear that function composition of vectorized operations preserves cardinality, and therefore order. The relationship between *map* and function composition runs deeper, with function composition itself being preserved under *map*.

Proposition 3.6.1. *Let $f : B \to C$, $g : A \to B$ be functions. Then $map(f \circ g, x) = (map\ f \circ map\ g)x$, for $x \subseteq A$.*

Given another function $g(x) = log\ x$, let's see how to apply this proposition to actual functions. It can be helpful to convert the composition symbol into a more familiar form. The definition isn't so helpful, though, since expanding $(f \circ g)(x) \to f(g(x))$ is no longer a function reference. A quick way around this is to use lambda notation to preserve the function reference: $map(f \circ g, x) = map(\lambda a.f\ g\ a, x)$. This form is actually the same approach necessary to implement the function composition in R: `sapply(x, function(a)f(g(a)))`. Proposition 3.6.1 is saying that this is equivalent to $map\ f \circ map\ g$. What does this notation mean though? Here again it's useful to interpret this from a lambda calculus perspective. The term $map\ f$ is a partially applied function, with only the first-class function argument bound. This makes sense since $\circ$ takes functions as arguments. Expanding this out we get $(map\ f \circ map\ g)x = map\ f\ (map\ g\ x)$. In R this translates to `sapply(sapply(x, g), f)`.

To make this clearer, let's use some actual values. Suppose $x = (1, 2, 3, 4, 5)$. Then $map(f \circ g)$ is

```
> f ← function(x) x^2
> g ← function(x) log(x)
> x ← 1:5
> sapply(x, function(a) f(g(a)))
[1] 0.000000 0.480453 1.206949 1.921812 2.590290.
```

Alternatively, *map* $f \circ$ *map* g is

```
> sapply(sapply(x, g), f)
[1] 0.000000 0.480453 1.206949 1.921812 2.590290.
```

One application of this equivalence is that it's possible to improve performance by restructuring a calculation. Portions of code can be parallelized differently based on the computational needs of each function. In other languages, a single *map* operation will generally be faster than two. However, in R this heuristic is not so cut and dry. The reason is that natively vectorized functions can be orders of magnitude faster than their non-native counterparts. Obtaining the best performance thus requires that functions are composed according to which functions are natively vectroized. Obviously this needs to be tested on a case-by-case basis. It's also possible to move around bits of computation according to the overall system design without having to worry about whether the calculation will change.

Not surprisingly, both cardinality and ordering is preserved under function composition. This property also gives us assurance in that both *map* and function composition do not change the nature of a calculation.

Proposition 3.6.2. *Let* $f : A \to B$ *and* $g : B \to C$ *be scalar functions. For vector* x *with length* n*, if* $\emptyset \notin B$ *and* $\emptyset \notin C$*, then* $|map(f \circ g, x)| = n$*.*

Example 3.6.1. In Section 3.2.3 we discussed a function that had two successive applications of *map*. It's so natural to expect that cardinality will be preserved that we didn't notice its significance. The call was

```
indices ← get_index(lines, markers)
rows ← lapply(indices, function(i) get_series(lines,i,size)).
```

Proposition 3.6.2 tells us that the final cardinality will be $|lines|$. Or does it? We're only guaranteed preservation of cardinality when composing *map* operations. To take advantage of Proposition 3.6.2 we need to first show that `get_index` is *map*-vectorized. Upon inspection, it seems to be, since it contains a call to `lapply`. However, since the second line is of the form `do.call(c, x)`, cardinality is not guaranteed. Therefore, this function is technically not *map*-vectorized, so we can't take advantage of this proposition.

Note that according to Proposition 3.6.1, this call can also be written as $\textsc{map}(\lambda x.\textsc{GetSeries}(lines, \textsc{GetIndex}(x, markers), size), lines)$. Even though this transformation is legal, we still cannot make claims about the final cardinality of the operation.

□

3.6.1 *Map* as a linear transform

A related property to function composition is linearity. Recall that a function or transformation L is linear if the following relationships are satisfied:

(a) $L(u + v) = Lu + Lv$

(b) $L(au) = aLu$,

where a is a scalar. In linear algebra, u and v are typically vectors, but this is not a requirement. It depends on the argument types of the transformation. For higher order functions, their arguments are functions. Higher order functions like the differential and the integral are both linear operators. This property can yield simplifications and other transformations to aid in problem solving. Linearity is not limited to the above functions and is also satisfied by map.

Proposition 3.6.3. *The map function preserves linearity. Let vector $x \in X^n$ and $a \in X$. If f and g are linear, then the function map is linear.*

(a) $map(y, f + g) = map(y, f) + map(y, g)$

(b) $map(y, a\ f) = a\ map(y, f)$

The value of this property is similar to that of function composition. Sometimes performance can be improved by rearranging a calculation. The most obvious example is if f and g are both computationally expensive and the final value comprises their sum. In this situation, f can be computed on one machine while g is computed on another machine. The sum can be performed wherever since it's computationally cheap.

3.7 Exercises

Exercise 3.1. Pending

4

Fold Vectorization

Whereas *map* operations iteratively apply the same function to a list of values, *fold* models iterated function application. [1] In this structure, a function is applied to each element of a list, coupled with the result of the previous application of the function. Iterated function application can also be considered n-fold function composition. Functions passed to *fold* are binary operators, and like *map*, this isn't so much a limitation as it is a protocol for defining the operation. In this chapter, we'll explore the mechanics of *fold* and the different types of operations that can be computed using it. These range from algebraic operations to operations on sequences to all sorts of state-based systems. We'll start the chapter by establishing the motivation for *fold* as a higher order function, work through some examples using the ebola data, and finally look at applying *fold* for numerical analysis. Additional examples appear in Chapter 8 that demonstrate using these techniques for modeling state-based systems.

4.1 A motivation for *fold*

In Chapter 3, we saw that many computations can be modeled as *map* operations. Broadly speaking, though, *map*'s repertoire is actually somewhat limited. At its core, *map* is designed to operate on element-wise data whose operations are independent from each other. This is why *map* can implement simple vector and matrix arithmetic but not an operation like summation. On the other hand, *fold* can be considered functionally complete since all other iterative operations can be implemented using *fold*, including *map*. This universality is similar to how NAND gates can implement all other logical operations. It stems from the fact that *fold* is a binary operation whose iterations are sequentially dependent on a changing accumulator.

Definition 4.1.1. Let $f : X \times Y \to Y$ and $x \in X^n$. The higher-order function *fold* is defined $fold(f, x, I_f) \equiv f(x_n, f(x_{n-1}, \cdots, f(x_1, I_f)))$. If $Y = X$, then I_f is the identity over f. Otherwise, I_f is the initial value.

The first-class function argument specifies two distinct sets X and Y as

[1] The terms *fold* and *reduce* are used interchangeably to describe this higher order function.

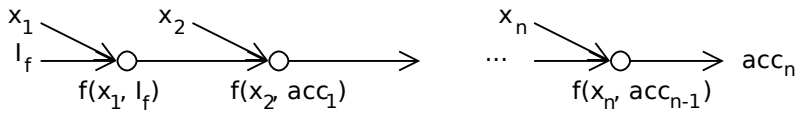


FIGURE 4.1: How *fold* operates on a vector input x. The result of each application of f becomes an operand in the subsequent application

arguments. The first argument, with elements in X, is the sequence to iterate over. The second argument, $y \in Y$ is an accumulator, which reflects the result of the previous iteration. At the first iteration, there is no previous value, so *fold* requires an initial value to seed the operation. Using two distinct domains accounts for operations that use state as the accumulated value. One such example is a Markov Chain, which we'll explore in Section **??**. A more prosaic use of *fold* is to collapse a vector into a single value based on some repeated operation, such as addition. This process is visually depicted in Figure 4.1 and illustrates how the result of each application of the function becomes an operand to the next iteration.

Example 4.1.1. A canonical example of a *fold* operation is to implement the summation operator. The sum of a sequence of values is merely addition repeated numerous times. The addition operator is the function passed to *fold*, along with an initial value of zero. In other words, $\sum_i x_i = fold(+, x, 0)$, for all $x_i \in x$. The implementation in R is essentially the same. [2]

```
> x ← 1:10
> fold(x, `+`, 0)
[1] 55
```

This works because the second operand to addition is the accumulated value from each preceding sum. To make it clearer, let's rewrite the operation using the notation of *fold*.

$$\begin{aligned}\sum_i^n x_i &= x_1 + x_2 + \cdots + x_{n-1} + x_n \\ &= ((((I_f + x_1) + x_2) + \cdots) + x_{n-1}) + x_n \\ &= x_n + (x_{n-1} + (\cdots + (x_2 + (x_1 + I_f)))) \\ &= f(x_n, f(x_{n-1}, f(\cdots, f(x_2, f(x_1, I_f)))),\end{aligned}$$

where $f(a, b) = a + b$ and $I_f = 0$. By making associativity explicit, it's easy to

[2] Once again our formal definition has a different order for the operands than the actual implementation. Although this presents some syntactic inconsistency, the author believes it's best for the mathematical formulation to maintain consistency with the abstract form, while the implementation remain consistent with R idioms.

see how the result of one addition becomes an operand for the next addition.

□

4.1.1 Initial values and the identity

In Example 4.1.1 we snuck in the I_f term that appears in the definition. What is the importance of this initial value I_f, and why should it be an identity? To answer these questions, let's compare $\sum$ with $\prod$. The implementation of $\sum$ used zero as the initial value. What is appropriate for $\prod$? The product operator is like summation except the terms are multiplied together. Clearly zero cannot be the initial value, since the result would also be zero. The only value that makes sense is one, since it doesn't materially change the value of the operation. The reason this works is because these values are the mathematical identities for their respective operations. That's why `fold(1:10, '*', 1)` gives the correct answer but `fold(1:10, '*', 0)` doesn't.

Other implementations of *fold*, such as [9] and [1], tend to use the first element of the sequence to seed the accumulator. For the algebraic case, using the first element of the sequence is equivalent to using the identity as the initial value. To convince yourself that this is true, consider some binary operation $\circ$ and corresponding identity I. The first iteration of the operation is $x_1 \circ I = x_1$ by definition of the identity over the operation. The second iteration is then $x_2 \circ x_1$, which is the same as simply passing x_1 as the initial value. That's a nice property, so why not do it all the time? Unfortunately it only works for certain cases where the accumulated value is the same type as the input sequence. In other words, the function argument to *fold* must have the more restrictive signature of $f : X \times X \to X$. This is why in our definition of *fold* we specify two distinct domains X and Y. When the accumulator $y \in Y$ is of a different type than the sequence, using this default action results in undefined behavior since the domains are incompatible. Hence, it's better to be explicit about the initial value than to save a few keystrokes with a default value that is only sometimes applicable.

Example 4.1.2. The argument to $\sum$ is typically some algebraic operation or other expression, such as $\sum_i x_i^2$. How can *fold* represent this type of calculation? The base function within the operator is $f(x) = x^2$, which maps $\mathbb{R}$ to $\mathbb{R}^+$. This seems incompatible with our current implementation of summation since it already uses addition as the binary function argument. To see if this

is the case, let's expand the operation.

$$\begin{aligned}\sum_i^n x_i^2 &= x_1^2 + x_2^2 + x_3^2 + \cdots + x_n^2 \\ &= (((x_1^2 + x_2^2) + x_3^2) + \cdots) + x_n^2 \\ &= (((f(x_1) + f(x_2)) + f(x_3)) + \cdots) + f(x_n)\end{aligned}$$

Now define $g(a, b) = f(a) + b$. Then

$$\begin{aligned}\sum_i^n x_i^2 &= f(x_n) + (f(x_{n-1}) + (\cdots + (f(x_2) + (f(x_1) + I_f)))) \\ &= g(x_n, g(x_{n-1}, \cdots, g(x_2, g(x_1, I_f))))\end{aligned}$$

So with a bit of rewriting, we can actually define a new function that subsumes the raw addition operation implicit in $\sum$. The end result is an n-fold function composition compatible with $fold$.

□

Let's pause to think about the significance of this transformation. We're saying that any repeated binary operation can be transformed into an n-fold function composition. It may seem limiting to use a single representation for numerous computations, but this generalizability is what makes $fold$ so powerful. Whole classes of equations from polynomials to sequences can be implemented using a single representation. For example, the factorial is a $fold$ process and is discussed in Section 4.4. The Fibonacci sequence, which is left as an exercise for the reader, is also a classic $fold$ operation.

Example 4.1.3. There is another way to represent $\sum_i x_i^2$ that isolates the two operations of addition and exponentiation. This approach composes $fold$ with *map* resulting in $fold(+, map(\lambda a.a^2, x), 0)$. Notice that this is equivalent to the idiomatic solution, which uses the natively vectorized versions of each operation, namely $sum(x^2)$. When is it appropriate to use one construction versus the other? One area where the formulations differ is in the algorithm complexity. In the $fold$ approach the operation is $O(n)$, whereas the combined *map* plus $fold$ approach is $O(2n)$. In this case, the difference isn't so great that it matters, but for more complex calculations the decision can be governed by which approach has less complexity. This construction is also the hallmark of the so-called map-reduce approach, since *map* operations can be easily parallelized. Complex *map* functions can thus be distributed across multiple compute nodes to reduce the overall "wall-clock" time of a computation. Another factor is related to function reuse. If certain functions already exist, then it's simplest to accommodate the approach implied by the function. For example, it's better to leverage native vectorization within a function since it

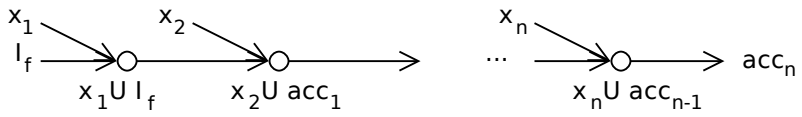


FIGURE 4.2: Iterated application of *union* over X

will be very fast. The reduction in big O complexity that comes with a scalar approach will likely be offset by the native vectorization.

□

Example 4.1.4. Set operations can be applied iteratively over an arbitrary set of values, similar to the behavior of the summation operator. But unlike summation, the values are themselves sets and not numbers. The union of multiple sets is $\bigcup_i X_i$, where each X_i is a set. This operation is depicted as a *fold* process in Figure 4.2 and is structurally equivalent to $\sum$. In R, `sum` is vectorized, but `union` is different. Since the arguments are vectors, you might consider it to be a vectorized function. But these vectors are semantically sets, which are treated as discrete entities. From this perspective `union` is strictly scalar. For sake of illustration, let's look at two approaches for implementing the vectorized $\bigcup$ operator. Suppose we want to take the union of four sets w, x, y, z, which we'll store in a list named `sets`. An imperative implementation has the typical set up of initializing an accumulator and then looping through each item. It looks like

```
x ← c()
for (s in sets) x ← union(x,s).
```

A declarative implementation is similar, except that the wiring for the loop and accumulator are encapsulated within the *fold* function:

```
fold(sets, union, c()).
```

Notice that in this construction the empty set is used as the initial value. This is consistent with our claim that the initial value should be the identity since $X \cup \emptyset = X$ and $\emptyset \cup X = X$ for any set X under $\cup$.

□

These examples show that *fold* semantics are universal irrespective of the function argument. Once the core pattern of iterated functions is understood, it's easy to deduce the behavior of any *fold* operation. It also becomes trivial to implement other operations, like $\bigcap$, or iterated set intersection: all we do is

replace the function argument to `intersect` from `union`. In a simple example such as this, the payoff in using a declarative abstraction may seem slight. At its core, though, the use of *fold* simply extends what we do naturally, which is to abstract and encapsulate operations via functions. Functional programming takes it a step further by abstracting the mechanics of iteration and function application itself.

Example 4.1.5. These initial examples treat the second operand to the first-class function as the incremental result of an iterated function. A more general perspective is to treat this variable as a total accumulation of the function output built up over time. In this representation the operand is not a scalar but its own explicit type. The cumulative sum and product are examples whose output are vectors. These operations produce a vector that grows with each iteration. The appended value for the cumulative sum operation represents the sum of the input up to that point in the sequence, i.e. $y_j = \sum_{i=1}^{j} x_i$ for all $j \in \mathbb{N}_n$. For instance, the cumulative sum of $(1, 2, 3, 4)$ is $(1, 3, 6, 10)$. The second cumulative sum is $y_2 = \sum_{i=1}^{2} x_i = 3$, while $y_3 = \sum_{i=1}^{3} x_i = 6$. Summation uses 0 as an initial value, but this won't work for the cumulative sum since the operands have two distinct types. Instead, the first element of the vector must be used as the initial value. The first pass of the function is thus $c(1, 2 + 1) = c(1, 3)$. This is slightly more interesting than summation as the result of the calculation is appended to the previous result. So in this case, the output cardinality is equal to the input. Our version of `cumsum` can thus be defined

```
mycumsum ← function(x)
  fold(x[-1], function(a,b) c(b, a+b[length(b)]), x[1]).
```

□

Accumulated values are a simplification of more general state-based systems. These are time-dependent processes that rely on previous state to generate the current state. Deterministic examples include moving averages, finite state machines, and cellular automata, while probabilistic examples include Markov Chains and stochastic processes. The additional complexity of these mathematical systems can require explicit manipulation of vector indices. Despite the added complexity of managing indices, this opens up a number of classes of calculation that are inaccessible with *map*.

Example 4.1.6. A simple example of a time-dependent system is calculating the position of a moving object. Recall that $s(t) = s(t-1)+vt$, when acceleration is zero. Even though this is a univariate function dependent on time, $s(t)$ also depends on the previous position. We can thus model the function in two variables. Hence, the two operands in *fold* represent different entities: the iterated vector is a sequence of time steps, while the second operand is $s(t-1)$. The function passed to *fold* is basically the equation with a slight

change of variable names: `function(t, s0)s0 + v*t`, where v is constant. Implementing this as a *fold* operation is therefore

```
fold(rep(1,n), function(t,s0) s0 + v*t, 0).
```

Now suppose we want the value at each time step? We can use the same technique as when deriving the cumulative sum, which yields

```
fold(rep(1,n-1), function(t,s) c(s, s[length(s)] + v*t, v).
```

As with the cumulative sum, the two operands to the first-class function argument are of different types. Hence, the initial value is seeded by a data structure containing the first result of the operation. This is not always required. The deciding factor is whether the first-class function can support multiple types. We could just as easily add logic to the function and simplify the *fold* call:

```
fold(rep(1,n), function(t,s) {
    if (length(s) == 0) return(v*t)
    c(s, s[length(s)] + v*t)
  }, c()).
```

Now the initial value is simply the empty set, but the drawback is that the lambda expression is more complicated. As with the other examples, choosing one approach over another is situation-specific.

□

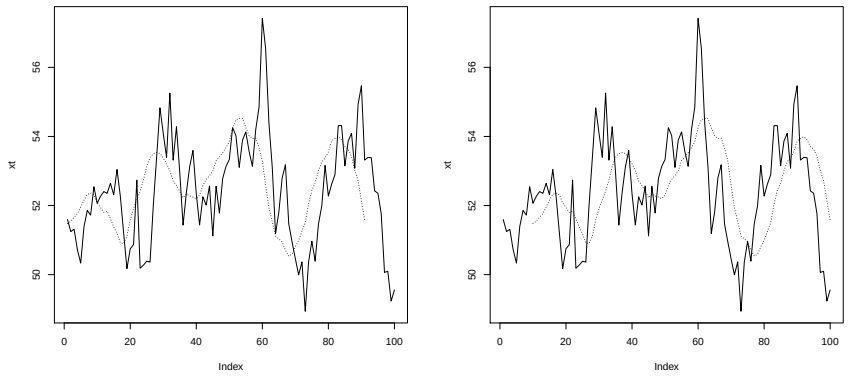
4.1.2 *Fold* implementations

Despite the plethora of native *map* implementations in R, there is a conspicuous dearth of *fold* implementations. The `funprog` package included in the base collection provides the analogous `Reduce` function [8]. At 65 lines, its implementation is imperative in nature and rather complex, which contradicts the simplicity argument made in this book. Consequently, we'll use the author's implementation of *fold* in his `lambda.tools` package [10], which is implemented in two lines. [3]

```
fold ← function(x, fn, acc, ...) {
  sapply(x, function(xi) acc ↞ fn(xi, acc), ...)
  acc
}
```

Interestingly, *fold* is a more fundamental operation than *map*, insomuch that *map* is typically derived from *fold*. For performance reasons, the author inverts this relationship so that his implementation is based on `sapply`. One aspect of this design choice is that `fold` is meant to be idiomatically consistent with `sapply`. Hence, results are automatically converted to vectors when

[3] To be fair, `Reduce` provides "left" and "right" variants, while `fold` only provides "left". To achieve the "right" version requires reversing the vector prior to calling `fold`.



(a) The SMA is not properly aligned when the cardinalities don't match

(b) Padding the SMA with NAs restores cardinality and alignment

FIGURE 4.3: Comparing the alignment of a derived time series

possible, and options to `sapply` work the same as in `fold`. In Sections **??** and **??**, we'll see the value of this approach.

Unlike with *map*, where the result data types are relatively few (and can be limited strictly to list if we so choose), there is no such limitation on *fold*. The output of a *fold* operation will typically be either a scalar type (which can be complex) or a vector. If the output is a vector, then there is likely 1) dependence between the terms, and 2) the cardinality will be equal between the input and output. This isn't a hard rule though, and as we'll see in Section 4.2, it can be appropriate to return a vector of arbitrary length. The following example shows the opposite case, where an arbitrary output length is undesirable.

Example 4.1.7. Most of the time, the output cardinality of a *fold* operation is either 1 or n, where n is the length of the input. Though common and useful, this is not a requirement dictated by *fold*. Operations like a moving average can have an arbitrary fixed length. Suppose we want to calculate a simple 10 day moving average of a time series. At any point in time, a simple moving average gives the mean of the last m values of a time series. We'll use a simple random walk to create a time series. Since this is a simulation, we can generate the time series in a single operation and calculate the SMA afterward, which is given by

```
xt ← 50 + cumsum(rnorm(100)).
```

This approach takes advantage of native vectorization but means we need to manage the vector indices. The implementation passes a truncated set of indices to *fold*, which acts as one of the endpoints of a window. The mean is calculated for each window and collected in a result vector.

```
sma ← function(xt, window=10) {
  fn ← function(i,acc) c(acc, mean(xt[i:(i+window-1)]))
  fold(1:(length(xt)-window+1), fn, c())
}
```

Applying this function to our time series results in a vector of length $n - window + 1$. While correct, the fact that the cardinality is different from the input means that we've lost information about how this result aligns with the original time series. Naive plotting of the SMA yields a time series pinned to the left side of the window. This shifting makes the graph appear to use future information as illustrated in Figure 4.3a. The correct graph aligns the time series to the right bound of the window (Figure 4.3b). Achieving this graph requires padding the SMA with `NA`s.

```
sma ← function(xt, window=10) {
  fn ← function(i,acc) c(acc, mean(xt[i:(i+window-1)]))
  fold(1:(length(xt)-window+1), fn, rep(NA,window-1))
}
```

The cardinality of the input vector dictates the number of `NA`s to use, which is the number required to equalize the two cardinalities. In this case $window - 1$ `NA`s must be used, since these first points have no associated mean. This example tells us that even though it's not required to preserve cardinality, oftentimes the result is more useful when it is preserved.

□

4.2 Cleaning data

Now that we have an understanding of *fold* and how it works, let's continue the analysis of the ebola situation report parsing process. We saw in Chapter 3 that the raw ebola data isn't so convenient to work with. It took a number of steps to parse a single table, and we didn't even address all the issues with the raw data. Numerous typos and PDF formatting artifacts exist that need to be resolved prior to parsing the actual tables. This is what's required for a single table, and each situation report contains multiple tables. With a few transformations the data can become more convenient to work with. One such transformation is correcting syntactic errors in county names so that table rows are associated with the right county. This is especially important since we'll want to merge all of the tables within a single situation report together. The effect is to produce a single table that contains all the columns we care about from the report. It's reasonable to consolidate the columns into a single data frame because each table represents observations associated with the same group of subjects (the counties). If we don't normalize the county names, we'll end up with a bunch of data associated with invalid counties

Pattern	Replacement
Garpolu	Gbarpolu
Grand Cape Mount	Grand Cape Mount
River Cees	River Cess
Rivercess	River Cess

FIGURE 4.4: Common syntax errors in Liberian situation reports

resulting in a corrupted data frame. Section 4.2.1 discusses the mechanics of this procedure.

The end goal of the parsing process is to return a single data frame that contains the desired measures from each situation report over a set of dates. Each table row is uniquely identified by a date and a county name, where the individual measures span the columns of the data frame. Since each table in a report represents just a single day, having a consolidated data structure makes it easier to conduct time series analyses. Reaching this goal requires that each table has the same column structure. While this may seem a given, measures change over time, so we cannot assume that the set of measures are constant over the history. Instead, we need to fill missing columns with some value (e.g. `NA`), which is discussed in Section 4.2.2. At a minimum, filling data regularizes the structure of the data. Whether we need to do further processing or manipulation of the data will be dictated by a specific analysis. Once a consistent structure is enforced across all data frames, the final step of data preparation simply consolidates all those data frames together into a single one. That step is covered in Section 4.3. The overall procedure is summarized in Algorithm 4.2.1, which also acts as the high-level entry point into the `ebola.sitrep` package.

Algorithm 4.2.1: ParseNation(fs)

$reports \leftarrow$ map($ParseFile, fs$)
$columns \leftarrow \bigcup_i$ colnames($reports_i$)
$reports \leftarrow$ map(λx.FillMissingData($x, columns$), $reports$)
return ($\bigcup_i reports_i$)

The basic idea is to take a set of file names and parse each file, returning a list of `reports`. These `reports` are then normalized over the set of all columns creating a consistent structure. Finally, the union of all the `reports` is returned. The implementation of this algorithm is postponed until Section 4.3.2, after we have discussed each individual line.

4.2.1 Fixing syntactic and typographical errors

Recall Figure 3.2 containing a sample page from a Liberian situation report. Constructing a digital version of this table is conceptually easy since the county names are constant over the different tables. However, the unruly nature of unstructured data complicates things. As we saw in Chapter 3, the parsing process is exacting and expects a regular structure. Even something as minor as an extra line break will corrupt the parsed output. Since county names are used as row indices, spelling errors can result in a bad join later on. Figure 4.4 lists common errors and their corrections in the reports. For instance, some of the tables list "Garpolu" as a county when it should be "Gbarpolu". Other syntax issues arise from the PDF output, leading to inconsistent identification of counties. One example is "Grand Cape Mount", which often spans two lines but sometimes doesn't, leading to "Grand Cape", "Mount", and "Grand Cape Mount" all appearing as counties. The extra line also means that data will be incorrectly shifted leading to "Mount" becoming a cell value if the row is picked up at all. Incorrect county names, therefore, result in missing data later on that can impede an analysis.

Properly cleaned data is clearly central to producing a quality dataset. In the ebola parsing process, county names are cleaned within the `get_chunk` function, [4] defined as

```
get_chunk ← function(lines, start.marker, stop.marker)
{
  start.idx ← grep(start.marker,lines)
  stop.idx ← grep(stop.marker,lines[-(1:start.idx)])[1]

  chunk ← lines[start.idx:(start.idx+stop.idx)]
  grep('^$',chunk, invert=TRUE, value=TRUE)
}.
```

A chunk represents a group of table cells where each cell is on a single line. Each chunk is later transformed into a row of a data frame. Cleaning the cell entries within this function ensures that syntax errors do not propagate through data structures built on the raw parsed data. Hence, we'll introduce a cleaning operation after the assignment of the `chunk` variable.

What's the best way to implement this cleaning algorithm? A cavalier approach to fixing these errors is to bundle the corrections with the parsing logic. But like a time series modulated by seasonal effects, it's harder to understand each individual behavior when they are intermingled. It is also harder to reuse bits of logic. For illustrative purposes, let's suppose that the cleaning logic is written in-line in an imperative development style. The core of the implementation is contained within a loop, preceded by variable initialization. It's easy to imagine a solution that looks like

```
clean.cells ← c()
```

[4] Some logic around edge cases have been removed to simplify the discussion.

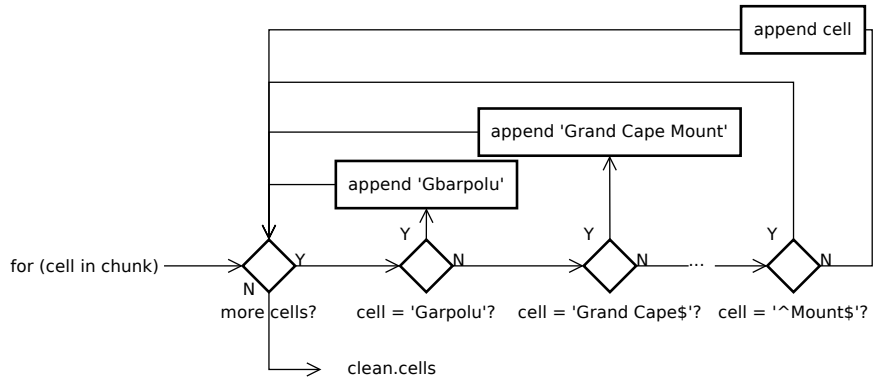


FIGURE 4.5: The control flow of a sequence of if-else blocks

```
for (cell in chunk) {
  if (cell == "Garpolu")
    clean.cells ← c(clean.cells, "Gbarpolu")
  else if (cell == "Grand Cape")
    clean.cells ← c(clean.cells, "Grand Cape Mount")
  else
    clean.cells ← c(clean.cells, cell)
}.
```

The county names are one of many data cells within a chunk. Cells are conditionally replaced when a known mistake is encountered. Each correction becomes a conditional branch in the loop, as depicted in the flowchart of Figure 4.5. If we're not familiar with vectorized operations, our inclination is to add another conditional branch to correct each new typo or other issue in the situation reports. Adding more conditional branches adds to the complexity of the code but offers little semantic value. Even visually we can see that the mechanics of a strictly imperative approach tend to overshadow the intention of the algorithm.

The problem is that a naive procedural approach intermingles many operations within the same scope. Adding the conditional block in-line not only overshadows the logic of the function, but it tightly couples the function to Liberia-specific data. Separating the tasks thus makes it easier to understand the logic and also replace and/or reuse individual functions. A better approach begins by encapsulating this operation within a function, such as `clean_chunk` and calling it via `chunk ←clean_chunk(chunk)`. Even though this would isolate the table extraction logic, the imperative approach still requires adding another conditional expression for each new syntax error encountered. Worse, the function is still Liberia-specific, which means it can't be reused for Sierra Leone's situation reports. It's easy to imagine that the

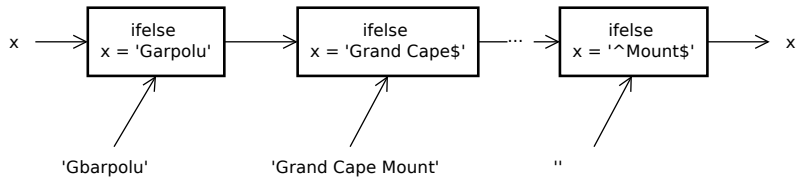


FIGURE 4.6: Using `ifelse` to simplify control flow

control flow illustrated in Figure 4.5 will continue to spawn new code paths that add to the complexity of the code.

An incremental improvement is to vectorize the conditional expressions and apply them as an iterated function using `ifelse`. Conceptually, this is iterated function application but mechanically it's still cumbersome to work with:

```
clean_chunk ← function(x) {
  ifelse(x == 'Mount', '',
    ifelse(x == 'Grand Cape', 'Grand Cape Mount',
      ifelse(x == 'Garpolu', 'Gbarpolu', x)))
}.
```

In this construction, the vectorization logic is built into the `ifelse` function. The transformation logic is syntactically isolated from the vectorization logic and is represented by two distinct vector operands, which is depicted in Figure 4.6. The effect is that the transformation now looks like a straight pipeline. In mathematical terms, a single code path generalizes the form for any sequence of transforms with length 1 to ∞ (up to machine limits). Through generalizations like this, the simplicity and elegance of mathematics is visible through the code. Despite these conveniences, the transformation specification is tightly coupled to the application of function composition. The end result is a long line of code that can be visually difficult to parse. It's also difficult to understand if the behavior of `ifelse` is not well understood. More problematic is dealing with a transformation specification that needs updating. Iterated unary functions are easy to understand, but additional in-line arguments can obfuscate the logic. The practical aspect of changing explicit function iteration means moving around a lot of commas and parentheses in code. While seemingly benign, changing these structures can introduce subtle bugs that are difficult to track down. In general we want to avoid constructions like this as they impede the path to implementing our ideas.

That said, we're on the right track by viewing mulitple replacements as function iteration. As with `ifelse`, to capture the effect of all replacement operations, the result of one replacement must be passed as input to the next one. However, the manual function composition is not a joy to work

with. Perhaps another vectorized function is more appropriate? Consider `sub`, which takes a scalar pattern and replacement along with a character vector. In this situation, `sub` has the same effect as a call to `ifelse` since the operation is idempotent when the pattern is not found in a string. Using `sub` in place of `ifelse` looks like

```
sub('^Mount$', '',
  sub('Grand Cape$', 'Grand Cape Mount',
    sub('Garpolu', 'Gbarpolu', x))),
```

which is a little better since we've removed the multiple references to `x`. A naive simplification is to assign the result to a temporary variable repeatedly, manually passing the output of one call into the next call, such as

```
o ← sub('Garpolu','Gbarpolu', x)
o ← sub('Grand Cape$','Grand Cape Mount', o)
o ← sub('^Mount$','', o).
```

Structuring the transformations like this reveals the repetition embedded within the operation. Consequently, we see that the arguments to `sub` have a regular pattern as well. This implies that with a simple rewriting of the function, it's possible to construct a binary iterated function compatible with *fold*. Doing so involves creating a data structure that represents the transformations as a single argument (since the other argument is the accumulator). One such structure is

```
xforms ← list(
  c('Garpolu', 'Gbarpolu'),
  c('Grand Cape$', 'Grand Cape Mount'),
  c('^Mount$', '')),
```

where each list element is a tuple containing the pattern and corresponding replacement arguments to `sub`. We call this structure a transformation specification, which is a narrow type of "business rule", or domain-specific logic. In this case it refers to rules like "replace Garpolu with Gbarpolu". These rules are typically specific to a dataset or instance of an application. Applying these rules to the sequence of cells is just a matter of using *fold* to manage the function iteration:

```
fold(xforms, function(r,ch) sub(r[[1]],r[[2]],ch), chunk),
```

which collapses the conditional logic into a single code path. Algorithm 4.2.2 codifies this approach to cleaning the county names.

Algorithm 4.2.2: CLEANCHUNK(*chunk*, *xforms*)

return (FOLD(λ *x ch.ch* $\sim s/x_1/x_2/$, *xforms*, *chunk*))

This is a lot easier to understand, but where does the logic of the branches go? The answer is that it's split between `sub` and the `list` data structure

instead of within explicit control flow. What's interesting is that we've completely separated the rules of the transformation from the mechanics of applying the transformation. In the same way that *fold* is purely mechanical, the replacement operation becomes purely mechanical via abstraction. Why is this useful? One reason is that the parsing process is now generalized. Ebola data comes from multiple countries, each of which publish their own situation report. The process of cleaning county names is applicable to each country. By separating the Liberia-specific replacement rules from the mechanical process of cleaning county names, the cleaning code can be reused verbatim. Separating application logic, which focuses on the processing pipeline and data management, from the transformation specification is what allows us to quickly deploy a new version of the parsing process for a new country. To use it for another country simply requires changing the transformation specification. For the `ebola.sitrep` package, there are separate transformation rules for Liberia and Sierra Leone, but the code path is the same. Working through this thought process demonstrates how functional programming naturally leads to modular code whose logic is easy to follow and reuse.

4.2.2 Identifying and filling missing data

It's no secret that the real world is a messy place. Mathematics on the other hand is a pure, ideal world that behaves consistently. Part of the role of data science is to coerce the real world into the pretty box of mathematics. Missing data is one of those messy circumstances that must be handled prior to entering the pristine world of math. Many analytic methods require a regularized data structure, and the gaps created by missing data can hinder an analysis. For example, missing data for a variable can invalidate a given cross section in a linear regression reducing the total number of observations available. Many time series methods also require values at a regular interval. With field data, this requirement might not always be met, so a strategy must be defined for handling missing data.

In Chapter 3 we showed how to validate the data and ensure that it's internally consistent. However, we didn't discuss what to do if missing or bad data was found. Ebola situation reports contain two types of data: raw counts and cumulative counts. Raw counts include newly reported cases or deaths, along with follow-ups with infected people. Another measure is the number of lost follow-ups, meaning that it was not possible to follow up with a given subject. An example appears in the county Nimba, where an `NA` appears in the `lost.followup` measure. [5]

```
> data.lr[data.lr$county=='Nimba', 'lost.followup']
 [1]  0 NA  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
[19]  0  0  0  0  0  0  0  0  0  0  0 35  0  0  0  0  0
```

[5]This dataset can be loaded directly by executing `data.lr ←read.csv('data/sitrep_lr.csv')`.

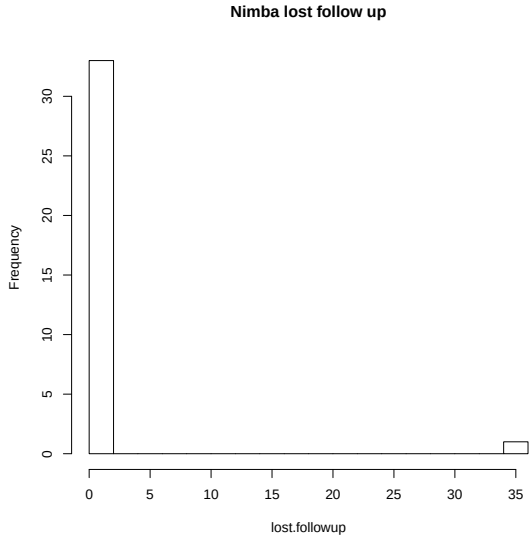


FIGURE 4.7: Histogram of patients lost in follow up in Nimba county

It's unclear why this `NA` exists, and a thorough investigation is warranted to determine whether it's a parser error versus an omission from the source report. For our purposes, we'll assume that the `NA` is legitimate and not caused by incorrect parsing.

After determining the source of a missing raw count, the next step is to decide how to handle `NA`s. We can either throw them all away and carry on with a complete-case analysis or impute the missing values from other values in the data set. Choosing the right imputation method is out of scope for this book, so we'll use a simple visual approach to decide what to do. It can be useful to plot a histogram of values to see the distribution more clearly, as in Figure 4.7, which clearly shows that most days the value is zero. Filling the `NA` with zero seems like an appropriate strategy. All that's needed to replace the `NA` is to use a set operation to select the cells in question:

```
data.lr[is.na(data.lr$lost.followup),'lost.followup'] ← 0.
```

This is simple enough, but what if we want to apply different replacements for each column? We would encounter this situation if we wanted to use a different imputation method for each column. Moving from a generic one-off transformation against all data to individual transformations is a good time to plan out a general approach to applying these transformations. Since we want to apply a transformation to each column in a data frame, we already have two iterations (once along columns, and along elemnts of each column) that can be managed by either *map* or *fold*. The simplest approach is to treat each column as being independent as well as each column vector element. If this is

the case, then a *map* operation can be applied twice, once along columns and then per column vector. Now suppose each column vector is transformed via a *map*-vectorized function. How do we apply the correct function to a given column? The simplest and most procedural form is to implement it as a double loop. Since this is R, the inner loop is almost always replaced with a vectorized operation. In our case we'll set a subset of a column to the same scalar value, which simplifies the loop. However, the code is already has a cumbersome structure with a massive conditional block:

```
for (column in colnames(data.lr)) {
  if (column == 'new.contacts') {
    data.lr[is.na(data.lr[,column]), column] ← 1
  } else if (column == 'under.followup') {
    data.lr[is.na(data.lr[,column]), column] ← 2
  } else if (column == 'seen.on.day') {
    data.lr[is.na(data.lr[,column]), column] ← 3
  } else if (column == 'completed.21d.followup') {
    data.lr[is.na(data.lr[,column]), column] ← 4
  } else if (column == 'lost.followup') {
    data.lr[is.na(data.lr[,column]), column] ← 5
  }
}.
```

After adding a few conditionals that have the same format, we might decide to clean it up by specifying only the value in each conditional and setting the value at the end. This only works if a constant value is appropriate for each column. Otherwise, it's back to column-specific code in each conditional.

Can we find a better way using *map* or *fold*? Recall in Section 3.5 the invariant nature of ordinals in a *map* operation. By creating a data structure whose ordinals are congruent with the column ordinals, it's possible to apply *map* to the shared set of ordinals. For illustrative purposes, let's assume that the data frame only contains the columns associated with the Contact Investigation Summary table: new.contacts, under.followup, seen.on.day, completed.21d.followup, and lost.followup. For each column, we'll follow the approach of filling in `NA`s based on the other values. A single higher-order function can provide the machinery for this, such as

```
get_fill_fn ← function(value)
  function(x) { x[is.na(x)] ← value; x }.
```

Calling this function returns a function that fills all `NA`s in the target vector with the `value` specified. [6] We need a data structure to hold each closure so that a distinct function is applied to each unique column in the data frame. For this to work, the closures must be added to a list in the same order as the columns:

[6] Our choice of fill value is purely for illustrative purposes to assist in distinguishing between the different closures.

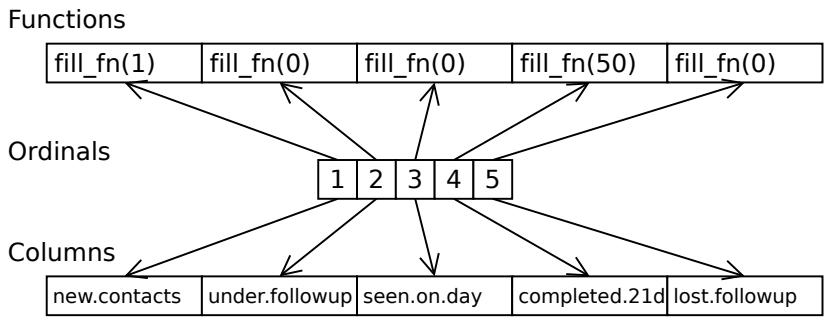


FIGURE 4.8: Using ordinals to map functions to columns

```
xforms ← list(fill_fn(1), fill_fn(2), fill_fn(3), fill_fn(4),
    fill_fn(5)).
```

Iteration can now be applied over these ordinals, mapping each function to the appropriate column:

```
out ← as.data.frame(lapply(1:nrow(data.lr),
  function(idx) { f ← xforms[[idx]]; f(data.lr[,idx]) }
)).
```

What we've done is used order invariance of the *map* process as a fixed mapping between a function and a column vector. A visual depiction of this relationship appears in Figure 4.8. Whenever the ordinals of two data structures are aligned, it's possible to use this technique to iterate over both data structures in tandem. Note that this is a short cut to using `zip`, which explicitly sets the ordering via a tuple.

This functional approach differs from the earlier procedural approach because it enforces structure and modularity in the solution. Separate functions exist for the transform, the transformation specification, and the mechanics of applying the transformation. This approach again promotes reuse, since only the transformation specfication need change to accommodate another country.

4.2.3 Ordinal maps

One advantage of the procedural solution is that columns can be in any order. The functional approach also works for any permutation of the ordinals but can get complicated. Suppose we created our transformation specification using a different ordering, such as

```
xforms ← list(fill_fn(3), fill_fn(1), fill_fn(5), fill_fn(2),
    fill_fn(4)).
```

The assumption is that the same parameter to `fill_fn` is used for the same column as above. To map these functions to the correct column now requires the use of a secondary ordinal mapping, which can be defined

```
my.order ← 1:5
names(my.order) ← c(3,1,5,2,4).
```

This mapping is dereferenced to obtain the correct closure:

```
out ← as.data.frame(lapply(1:nrow(data.lr),
  function(idx) {
    f ← xforms[[my.order[as.character(idx)]]]
    f(data.lr[,idx])
  }
)).
```

It's equally valid to invert the map, which means that it would be used to dereference the column ordinal and not the function ordinal. Either way, the complexity of the code begins to outweigh the flexibility gained by the approach.

Using a vector as an ordinal map is common practice in R. Keep in mind that performance is $O(n)$ and many times the map can be eliminated by being judicious in how the data structure is constructed. One trick is to pre-sort data structures to guarantee the same ordering. Another reason for using an ordinal map is when only a subset of a data structure needs processing. We started this section by assuming only five columns were in the data frame. Since these five columns are only a subset of the total available columns, it's difficult to use the implied ordinals. Using the implied ordinals necessarily defines the cardinality as well. To gain control of the cardinality of a set of ordinals requires an ordinal map of arbitrary length. This can even be a sub-sequence of the original ordinals.

Data structures are named in R, so we can use column names to simplify creating the ordinal map. The benefit is that it's easy to define a subset of columns to process, not to mention it removes a dependency on the order of the columns. The only change to the implementation centers around the naming of the set of transformations to correspond with a column of the data frame.

```
columns ← c('new.contacts','under.followup','seen.on.day',
  'completed.21d.followup','lost.followup')
names(xforms) ← columns

out ← as.data.frame(lapply(columns,
  function(col) { f ← xforms[[col]]; f(data.lr[,col]) }
))
```

4.2.4 Data structure preservation

So far this whole discussion has focused on *map*. It seems like a reasonable approach, so what is the benefit of using *fold* to mediate the transformation? In the *map* implementation, notice that the `lapply` call is wrapped up in a call to `as.data.frame`. The independent operations within *map* implies that the structure of a data frame is lost. Since *fold* returns accumulated state throughout the iteration, another way of thinking about *fold* is that it preserves a data structure through a sequence of transformations. Hence any time that it's necessary to preserve a data structure over an iteration, it likely makes sense to use *fold*. For this example, the idea is to fold along the specified columns as determined by a validation.

```
out ← fold(columns,
  function(col, df) {
    f ← xforms[[col]]
    df[,col] ← f(df[,col])
    df
  }, data.lr).
```

With *fold*, the data frame is preserved, so there is no need to call `as.data.frame` to recreate the original data structure. We derive a similar advantage from the basic procedural approach, where in-line modifications do not affect the top-level data structure.

4.2.5 Identifying and correcting bad data

Missing data can often be handled independent of other data, but bad data often requires cross-checking with other columns. We discussed the idea of internal consistency in Section 3.3.1 that ensures individual county data sums to reported national data. Another internal consistency metric is to check whether a cumulative column y is monotonically increasing and also that each value y_t is equal to $y_{t-1} + x_t$, where x is the corresponding raw count. This is a case where one column is dependent on another, such as cum.death and dead.total. Figure 4.9 lists the values of the two columns over a two month span. It's clear that numerous errors exist in the data. The cum.death column not only contains `NA`s but also is not monotonic. It's also clear that the cum.death column is inconsistent with dead.total. How can we programmatically identify these issues? The answer requires first deciding on an output format for identifying these issues. We'll keep it simple by implementing each check separately.

A monotonic function f has the property that the output is either always increasing in value or decreasing in value. Given two scalars a and b, where $a \leq b$, f is monotonically increasing if $f(a) \leq f(b)$ and monotonically decreasing if $f(a) \geq f(b)$. The `cumsum` function is monotonically increasing when its input is in Z^*. A simple test of monotonicity is to check if the preceding value is less than the current value. The result of our test should thus be a logical

```
> data.lr[data.lr$county=='Sinoe', c('date','dead.total','
    cum.death')]
          date dead.total cum.death
16  2014-11-06          0        11
32  2014-11-07          0        11
48  2014-11-08          0        11
64  2014-11-14          0        11
80  2014-11-15          0        11
96  2014-11-19          2        11
112 2014-11-20          0        11
128 2014-11-21          0        11
144 2014-11-23          0        11
160 2014-11-24          0        22
176 2014-11-26          0        14
192 2014-11-27          0        14
208 2014-11-28          0        14
224 2014-12-05          0        14
240 2014-12-06          0        14
256 2014-12-07          0        14
272 2014-12-08          1        14
288 2014-12-09          0        14
304 2014-12-10          0        14
320 2014-12-11          0        14
336 2014-12-12          0        NA
352 2014-12-13          0        NA
368 2014-12-14          0        14
384 2014-12-16          1        17
400 2014-12-17          0        17
416 2014-12-18          0        17
432 2014-12-19          0        17
448 2014-12-20          0        17
464 2014-12-21          0        17
480 2014-12-22          1        17
496 2014-12-23          1        17
512 2014-12-24          0        17
528 2014-12-25          0        17
544 2014-12-26          1        17
560 2014-12-29          1        18
```

FIGURE 4.9: Cumulative death values are inconsistent with daily dead totals and need to be fixed.

vector, where the first value is NA. To simplify the discussion, we'll extract the cum.death column for Sinoe county and assign to a variable x:

```
xs ← data.lr[data.lr$county=='Sinoe',]
```

Now let's define a function that determines if a vector is monotonic or not.

```
is_monotonic ← function(x) {
  fold(2:length(x), function(i,acc) c(acc,x[i] >= x[i-1]), NA)
}
```

Applying this function to Sinoe county yields our desired logical vector.

```
> is_monotonic(xs$cum.death)
 [1]    NA  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[10]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[19]  TRUE  TRUE    NA    NA    NA  TRUE  TRUE  TRUE  TRUE
[28]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

We also need to take care around the NAs in the cum.death series, which will result in additional NAs. Indeed, our implementation of is_monotonic actually introduces a bug. The NA at index 23 is incorrect as the correct value is TRUE. Based on our logic we cannot simply check the previous value but need to check the last valid value. Our function therefore needs to change to take this into account. Instead of using a previous value of x[i-1] we have to be smarter and use

```
prev ← xs$cum.death[which.max(!is.na(xs$cum.death[1:(i-1)]))]
```

When $i = 3$, the previous value should be 14 at element 21. For readability, it's convenient to wrap this up in a closure and call it within *fold*, like

```
is_monotonic ← function(x) {
  prev ← function(i) x[max(which(!is.na(x[1:(i-1)])))]
  fold(2:length(x),
    function(i,acc) c(acc,x[i] >= prev(i)), NA)
}.
```

This gives us the expected result of

```
> is_monotonic(xs$cum.death)
 [1]    NA  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[10]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[19]  TRUE  TRUE    NA    NA  TRUE  TRUE  TRUE  TRUE  TRUE
[28]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE.
```

Why did we seed the operation with NA instead of starting with an empty vector and the first index? Monotonicity compares two values, so by definition the lower bound of a range cannot be compared with anything. Hence, it is appropriate to use NA for the initial value.

Testing for monotonicity is the easier of the two tasks. Verifying that the cumulative column is internally consistent is a bit more involved due to the dependence on another column. Once again, it's useful to consider what our

result should look like before getting started. Given that our target is the cumulative column, it's reasonable to return the same type of logical vector where the first value is `NA`. We'll call this function `is_consistent`. The input for each step of the iteration comprises three pieces of information: the current values for cum.death and dead.total plus the previous value of cum.death. Collectively, though, this information is contained within two vector arguments which we'll call x for the raw counts and y for the cumulative values. One implementation is to fold along the ordinals and use the same manipulation of indices to get the previous value. We saw how that required a bit of work in the previous example, so perhaps there's a simpler alternative.

Let's assume that cumulative counts are correct in the data. Instead of taking the cumulative sum of the raw counts x, now we want to add each x value to the preceding y value, such as

```
is_consistent ← function(x, y) x + lag(ts(y)) == ts(y)
```

and produces

```
> is_consistent(xs$dead.total, xs$cum.death)
Time Series:
Start = 1
End = 34
Frequency = 1
 [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE
[10] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
[19]  TRUE    NA    NA    NA FALSE  TRUE  TRUE  TRUE  TRUE
[28]  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE.
```

Suppose instead we assume that the daily raw counts in dead.total are correct. This is a better assumption since the cum.death column contains numerous obvious errors. Under this assumption, the validation takes the form of calculating the cumulative sum from the raw counts and comparing it to the parsed values. Since the dataset does not start at the beginning of the series (i.e. the first reported ebola case), there is already a non-zero cumulative value established for cum.death. Hence, we need to assume that the initial cumulative value is correct and subtract the first raw count to emulate the accumulated raw counts up until this point in time. In other words,

```
is_consistent ← function(x, y) y[1] - x[1] + cumsum(x) == y,
```

which yields

```
> is_consistent(xs$dead.total, xs$cum.death)
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
[10] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
[19]  TRUE  TRUE    NA    NA  TRUE FALSE FALSE FALSE FALSE
[28] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE.
```

There are a number of differences between these two implementations. Most striking is that the cardinalities are not equal. In the first implementation, comparing the lagged series against the original produces a shorter time

series. The cardinality of the result is $|y| - lag$, where y is the argument to `is_consistent`. Consequently, we are unable to compare the last value in the series. The second approach, though, preserves cardinality. By avoiding a lag and instead constructing a cumulative series, we preserve all data points. It also guarantees monotonicity, which means at least one of the series behaves as desired. In the first example, neither series is monotonic, which raises issues concerning the validity of the approach. This explains why the truth values are different between the two implementations.

A vectorized approach is compact and fairly easy to understand. Since this chapter is about using *fold*, why might we use *fold* instead of a *map*-vectorized approach? Aside from the usual case of using *fold* when a function is not vectorized, at times the code can be easier to understand. Since the closures passed to *fold* compute a single iteration at a time, it can aid in working through the logic of the process. This is a primary argument for using procedural code, but then you lose the encapsulation and isolated scopes provided by the functional approach. Another reason for using *fold* is to handle edge cases and errors more effectively. Using natively vectorized code will either produce `NA`s or fail completely depending on the problem encountered. For computationally heavy processes, it can be annoying and time-consuming for a calculation to fail requiring a complete restart after addressing an issue. Having access to each step of a calculation enables numerous debugging scenarios, from inline calls to `browse()`, logging statements, to conditional imputation. In this version of the function, we include a conditional log statement whenever the current cumulative death value is less than the previous one.

```
is_consistent ← function(x, y) {
  f ← function(i, acc) {
    cum.death ← x[i] + acc$cum.death
    if (cum.death < acc$cum.death)
      flog.warn("cum.death of %s is less than %s", cum.death,
          acc$cum.death)
    list(out=c(acc$out, y[i] == cum.death), cum.death=
        cum.death)
  }
  out ← fold(1:length(x), f, list(out=c(), cum.death=y[1] - x
      [1]))
  out$out
}
```

This implementation takes advantage of the accumulated state that *fold* operations carry. Rather than extracting the previous value from cum.death (which could be incorrect), we pass this value as part of the state. Instead of a vector result, a list with two components is used where the first element is the result of the consistency test and the second the "true" cumulative sum based on manually accumulating dead.total.

Each approach offers its own benefits and drawbacks. Simpler rules can

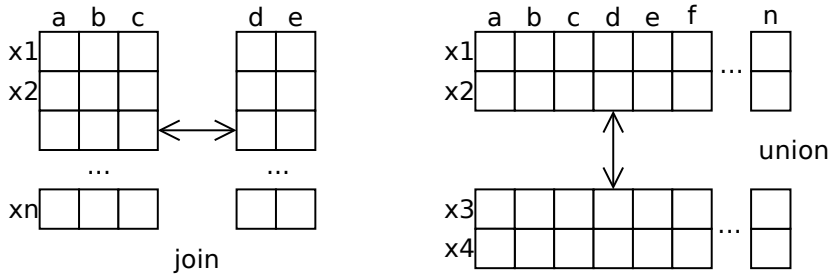


FIGURE 4.10: Two approaches to combining tabular data together. Adding new features to existing samples is a join, while increasing the sample for the same features is a union.

usually be implemented using native vectorization, while more complex scenarios require explicit calls to *map* or *fold*.

4.3 Merging data frames

Suppose you have two tabular datasets A and B that you want to combine. The way you go about this is dictated by the datasets themselves. If the datasets represent the same underlying samples, then the solution is to merge the columns of B with A. The solution involves identifying the desired columns from each data frame and merging the respective subsets together on a common index. In SQL terms this is known as a *join*. In R, this task is the purview of the `merge` function, which ensures referential integrity over the join operation. On the other hand, if the datasets represent two sets of samples that you want to treat as a single population, then the solution is to take the union of A and B.[7] Another way to look at a join operation is that the keys are treated as being constant while columns vary between the two tables. In contrast, a union operation holds columns constant while rows are added from different tables. The difference between joins and unions is depicted in Figure 4.10. In this section we'll look at effective techniques for managing either scenario.

4.3.1 Column-based merges

Combining tables that share row indices is known as a table join. How tables are combined in R depends on whether row indices or column indices have a

[7] In algorithms, we'll use matrix notation to denote join operations, such as $[AB]$. On the other hand, a union is denoted using set notation, such as $A \cup B$. In either case, the rows/columns of A and B are expected to be compatible.

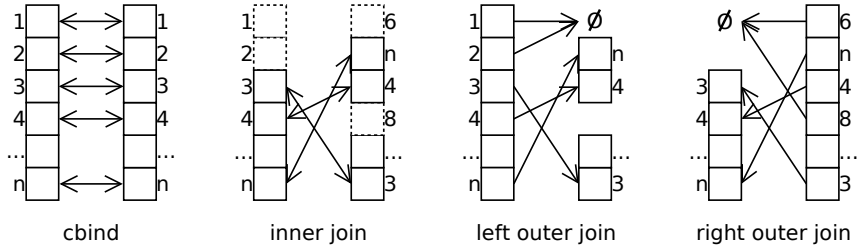


FIGURE 4.11: A set-theoretic interpretation of join operations based on table indices. A full join (not shown) is a combination of both a left and right outer join.

constant ordering. In the simplest case where row indices have a one-to-one correspondence across the tables, a basic `cbind` is sufficient to combine the tables. This is the same requirement that vectorized binary operations have, where the cardinality and ordinals are identical between the operands. If these conditions are not satisfied, then the tables must instead be combined via `merge`, which is analogous to a SQL join. SQL defines four join types depending on which keys the output set must contain [5]. If the output keys contain only the intersection of two tables A and B, then this is an inner join. On the other hand, if the output set must have all the keys in A, then this is a left outer join. Similarly if all the keys of B must be returned, this is a right outer join. A fourth join type is a full outer join, which requires that all keys in both tables must be returned. These operations are illustrated in Figure 4.11. When a table is missing a key in an outer join operation, the columns associated with the table are filled in with `NULL`. Maintaining a well-formed tabular data structure is useful when making future transformations since ordering and cardinality are consistent over the rows and columns of the table. In the situation reports, counties can be completely missing from a table, so the set of unique keys (the county names) can vary from table to table. If we want to join these tables, the missing values must be filled in with `NA`, which preserves the regularity of the data structure like the SQL `NULL`. It is up to the data scientist, however, to decide how best to deal with the `NA` value.

The `ebola.sitrep` package uses the `merge` function to join the individual tables together. This is a standard approach to this problem, though it's instructive to consider the desired behavior of the operation independent of `merge`. To simplify the discussion we'll focus on merging the Contact Investigation Summary table with the Healthcare Worker Cases and Deaths table. Explicitly adding `NA`s to the data requires taking the set difference between the set of all counties U and the set of counties C_i for each table i. We can explicitly specify the set of counties in advance, although they can equally be

inferred from the data itself. In other words we can define $U = \bigcup_i C_i$, which we implemented in Example 4.1.4. The nice thing about using inference is that there is less configuration for the parsing process, which makes the tool easier to use. [8] Adding `NA` rows to a data frame is conceptually simple but somewhat involved as Algorithm 4.3.1 attests. The complexity stems from the creation of a compatible table structure. Both the rows and columns to add must be inferred from the data frame argument and have the proper column names. Identifying the `NA` columns involves taking the set difference of all column names in the data frame and the assigned key column. The key column is populated with the set difference of the universe of keys and the keys in the given data frame. Finally, the `setNames` function binds a set of column names to a vector or list.

Algorithm 4.3.1: FillTable($x, keycol, keys$)

$cols \leftarrow$ SetNames(NA, colnames(x) $-$ $keycol$)
$shim \leftarrow$ SetNames($keys - x_{*,keycol}, keycol$)
$table \leftarrow x \cup [shim\ cols]$
return (sort($table, keycol$))

The corresponding implementation is essentially the same, although there is a little extra work to ensure type compatibility. Notice that lists are used as generic containers and converted to data frames only when needed. Chapter 6 discusses some of these strategies along with transformations between lists and data frames plus their inverses.

```
fill_table ← function(x, keycol, keys) {
  raw.cols ← setdiff(colnames(x), keycol)
  cols ← setNames(as.list(rep(NA,length(raw.cols))), raw.cols)
  shim ← setNames(list(setdiff(keys,x[,keycol])),keycol)
  table ← rbind(x, as.data.frame(c(shim,cols)))
  table[order(table[,keycol]),]
}
```

This function ensures that a row exists for each specified value of the primary key. Rows are also sorted according to the primary key, which makes it easy to bind the table with another table in the future.

Example 4.3.1. A short example illustrates how `fill_table` works. Let's create a data frame y with four rows with column a as the primary key. We want y to span $a = [1, 7]$, filling any missing rows with `NA`s.

```
> y ← data.frame(a=1:4, b=11:14, c=21:24)
> fill_table(y, 'a', 1:7)
  a  b  c
```

[8] Technically we cannot know the complete set of names by inference alone. However, the union behaves like the supremum of the set insomuch that its power set is the smallest set that contains all other power sets of known county names.

```
1 1 11 21
2 2 12 22
3 3 13 23
4 4 14 24
5 5 NA NA
6 6 NA NA
7 7 NA NA
```

As expected, the returned table contains three more rows filled with `NA`s except in the specified key column.

□

The `fill_table` function does a lot of work in just a few lines. In fact, most of the work to merge the data frames is contained in the filling process as this function must accommodate any dimensions. Algorithm 4.3.2 focuses on the final step of joining the two tables, which reduces to a column-based union operation.

Algorithm 4.3.2: FULLJOIN(a, b, key)

$keys \leftarrow a_{*,key} \cup b_{*,key}$
return ([FILLTABLE($a, key, keys$) FILLTABLE($b, key, keys$)])

This implementation is virtually verbatim to the algorithm since only set operations are being performed.

```
full_join ← function(a,b,key) {
  keys ← union(a[,key], b[,key])
  cbind(fill_table(a,key,keys), fill_table(b,key,keys))
}
```

Now we see why it's useful to pre-sort the rows based on the primary key. If we didn't do this, we would instead have to do it in this function. The cost of pre-sorting is delivered as extra overhead when sorting is not needed. This incurs an extra $O(n)$, which typically is insignificant.

Example 4.3.2. Continuing from the previous example, suppose we want to join data frame z with y. Both tables share column a as the primary key but contain some different elements. The `full_join` of the two tables works as expected and matches the rows according to the key column.

```
> z ← data.frame(a=3:6, d=31:34, e=41:44)
> full_join(y,z,'a')
  a  b  c a  d  e
1 1 11 21 1 NA NA
2 2 12 22 2 NA NA
3 3 13 23 3 31 41
4 4 14 24 4 32 42
5 5 NA NA 5 33 43
6 6 NA NA 6 34 44
```

□

For practical problems the `merge` function is used to join two data frames together. This function supports the four common join operations (inner, left outer, right outer, full). A full join is performed by specifying `all=TRUE`.

Example 4.3.3. Let's see if `merge` gives us the same answer as `full_join`.

```
> merge(y,z, by='a', all=TRUE)
  a  b  c  d  e
1 1 11 21 NA NA
2 2 12 22 NA NA
3 3 13 23 31 41
4 4 14 24 32 42
5 5 NA NA 33 43
6 6 NA NA 34 44
```

It turns out that the output is not exactly the same. The difference is that `merge` removed the redundant column for us. Changing `full_join` to produce the same output as `merge` is left as an exercise for the reader.

□

The `merge` function works well for two datasets, but each situation report has multiple tables that need to be joined together. At this point it should be clear that it is unnecessary to explicitly call `merge` multiple times. This is desirable as n tables require $n - 1$ calls to `merge`, which can quickly become tedious and error prone. When individual tables are removed or added to the parsing process, the pain related to using hard-coded steps becomes acute. A better approach is to pass a list of individual reports to *fold* and call merge that way.

```
fold(tables, function(a,b) merge(a,b,by='',all=TRUE),
  c(), simplfiy=FALSE)
```

Now it doesn't matter how many tables exist since the same code path is valid for all cases. The `parse.lr` function follows this approach, but as we'll see in Chapter 6, different situation reports have different sections and tables. Hence, each one needs to be parsed slightly differently. Since additional work must be done, this logic is embedded within the parse configuration.

4.3.2 Row-based merges

Folding all tables of an individual situation report gives us one data frame per day. The second combination strategy occurs when merging each of these days together. Unfortunately, tables within situation reports are not constant. Data were collected in real-time as the outbreak was occurring, so the measures were not stable over the collection period. It's unknown the precise reason for the changing data series, but we can surmise that it was to be consistent with

data sets from other organizations (e.g. WHO or the Sierra Leone MoH), was difficult to record consistently, or perhaps had little reporting value. At any rate we need to ensure that they share a consistent column structure prior to combining the daily data frames. The procedure is similar to what we did in Algorithm 4.3.1, but now we want to fill columns and not rows. Since the data available in situation reports varies, we can't blindly join multiple reports together. Doing so causes an unrecoverable error because their structures are incompatible. Using a toy example let's see what happens when two data frames have only one column in common.

```
> x ← data.frame(a=1:3, b=11:13)
> y ← data.frame(b=14:16, c=21:23)
> rbind(x,y)
Error in match.names(clabs, names(xi)) :
  names do not match previous names
```

It goes without saying that we cannot simply join these two data frames together. For these data frames to be compatible they need to have the same column structure. [9] We can guarantee universal conformity of the column structure by identifying the set of all columns over all reports. Knowing the elements of this set means that we can fill in the missing columns in individual situation reports. This is sufficient to normalize the column structure. It's easy to verify this behavior by simply assigning those columns with `NA` and trying again.

```
> x ← data.frame(a=1:3, b=11:13, c=NA)
> y ← data.frame(a=NA, b=14:16, c=21:23)
> rbind(x,y)
   a  b  c
1  1 11 NA
2  2 12 NA
3  3 13 NA
4 NA 14 21
5 NA 15 22
6 NA 16 23
```

Filling columns is straightforward and only requires knowing which columns to add. In this toy example we are defining the columns explicitly, so filling the missing columns with `NA` is a static exercise. Parse processes are different though, because we don't always know a priori what columns exist in each individual report. Previewed earlier in the chapter, Algorithm 4.2.1 highlights this issue. The actual implementation of `parse_nation` is

```
parse_nation ← function(fs) {
  o ← lapply(fs, parse_file)
  cols ← fold(o, function(i, acc) union(acc,colnames(i)), c())
```

[9] Although data frames require the same column structure, there is no restriction on the ordering.

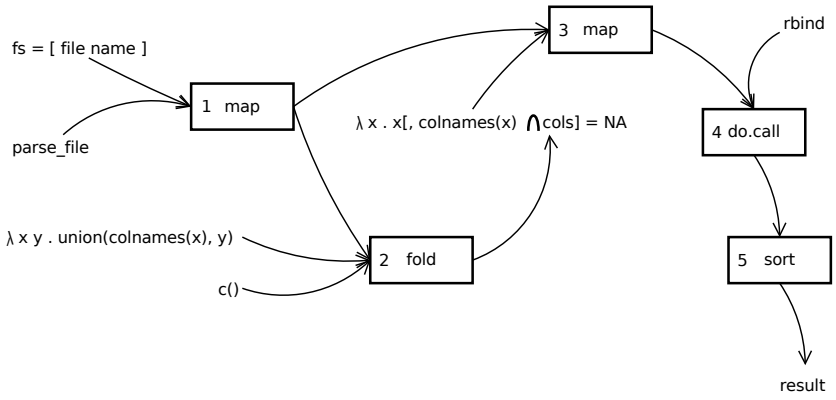


FIGURE 4.12: The `parse_nation` function modeled as a graph.

```
  o ← lapply(o, function(x) { x[,setdiff(cols,colnames(x))] ←
      NA; x })
  o ← do.call(rbind, o)
  o[order(o$date,o$county),]
}.
```

Let's work through each of the five lines to understand what's happening. The first line is simply the *map* operation that processes every situation report. The variable `o` is a temporary object, which is a list containing each parsed situation report. The next line uses *fold* to construct the union of column names across all parsed situation reports. This is followed by a *map* operation to fill missing columns in situation reports with `NA`. The final union of all data frames is via `rbind`, which is mediated by `do.call`. This acts as another form of vectorization, the full mechanics of which are discussed in Chapter 6.

The small bit of code in `parse_nation` packs quite a punch. Each line represents a standalone transformation that is clear in its intent. Furthermore, the whole body can be modeled as a sequence of function compositions (shown in Figure 4.12). Another way of saying this is that the whole transformation can be modeled as a depedency graph. The benefits of modeling computations as graphs are well understood [2]. There is a direct correlation between the complexity of the computation graph and the difficulty of understanding the corresponding code. Function composition is one of the easiest ideas to understand since it simply models a sequential chain of transformations.

4.4 Sequences, series, and closures

We've looked at numerous uses of *fold* to iterate over sequences of values. Using the ebola data we've seen how *fold* easily operates on data with serial dependencies, such as cumulative counts. It's now time to turn our attention to mathematical sequences and techniques to manipulate them using both *map* and *fold*. Sequences and series are indispensable structures in math. They are equally important in R given the emphasis on vector operations. Conventional iterative approaches to building sequence-like structures are often eschewed in favor of one-shot vector approaches. This implies that the result of a series operation is typically formed after creating a sequence of the terms and then applying the appropriate operator across the elements (usually addition). A sequence is essentially an ordered set (the natural numbers) coupled with a rule for constructing the sequence. A very simple sequence is the set of factorials, which maps a natural number n to $\prod_{i=1}^{n} i$. Creating the sequence of factorials $\{k!\}_{k\in\mathbb{N}}$ is an example of vectorization and can be quickly constructed by calling *factorial*(k), where $k \in \mathbb{N}^n, n \in \mathbb{N}$. A concrete example is

```
> factorial(1:5)
[1]   1   2   6  24 120,
```

where the vector $k = [1, 5]$ yields the corresponding factorials for each element of k.

4.4.1 Constructing the Maclaurin series

Real-world sequences are typically more involved than the sequence of factorials. As an example let's look at approximating the function e^x about 0 using its Maclaurin series. The coefficients of this power series expansion comprise a sequence given by $\left\{\frac{x^n}{n!}\right\}$, where $x \in \mathbb{R}, n \in \mathbb{N}$. This expands to $e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} ... \frac{x^n}{n!}$, where each term in the series can be codified as `function(x, n)x^n /factorial(n)`. Note that the arguments to the function include both x and n, where x is a scalar and n is a vector. Instead of producing the sequence of coefficients suppose we want the series so that we can use the approximation. This requires nothing more than wrapping the sequence term in the *fold*-vectorized *sum* function, which transforms the sequence into a scalar value.

```
maclaurin ← function(x, n) 1 + sum(x^n / factorial(n))
```

It's also useful to plot this approximation against the actual equation over an interval of x. We have to be careful here as standard vector notation will cause unexpected results. It is not possible to simply pass x as a vector since R does not know how the values are supposed to vary in relation to each

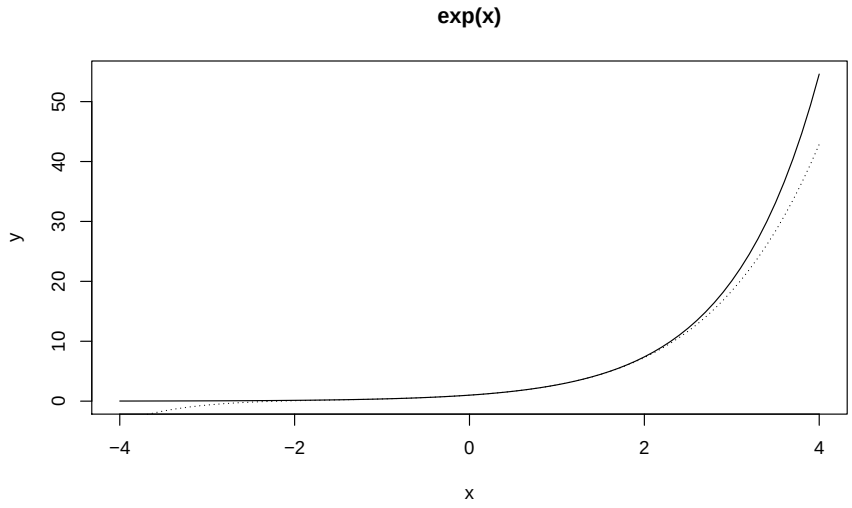


FIGURE 4.13: The Maclaurin series approximation of e^x about 0.

other. This has to do with the nature of the variables and the fact that they are independent. This makes sense, since e^x is defined for $\mathbb{R}$ and not $\mathbb{R}^n$. Hence x can only be specified a single value at a time. In general a function can only support vectorized arguments for a single dimension. This means that the higher order function *map* must be used to manage varying the second argument. The end result is the mapping $X^n \to Y^n$. The resultant function definition becomes

```
maclaurin ← function(n)
  function(x) 1 + sum(x^n / factorial(n))
```

which is the curried form of the original function. This can be passed to *map* and executed over any interval, such as

```
sapply(seq(-4,4,by=0.1), maclaurin(1:5)).
```

This is a practical example of making two function interfaces compatible. In the series approximation of e^x, the original two argument function is transformed into a chain of one argument functions so that it is compatible with the function interface map expects. The inner function is the closure since it references the variable n, which is defined in the outer function f. Recall that the same result is returned whether the function is defined with two arguments and called as $f(0.1, 0:5)$ or is curried into a chain of functions and is called as $f(0:5)(0.1)$.

This *map* implementation of the approximation is very clear and concise. At times, however, it may be more desirable to take an alternative approach and use the higher order function $fold$. Given the definition of the Maclaurin

series, we know that each term a_n in the sequence can be constructed from the preceding term by

$$\begin{aligned} a_n &= \frac{x^n}{n!} \\ &= \frac{x^{n-1}x}{(n-1)!n} \\ &= a_{n-1}\frac{x}{n} \end{aligned}$$

Although somewhat unconventional, this transformation expresses each coefficient as a reccurrence relationship, which is well suited to a *fold* operation. Since each successive element can be accumulated in the accumulator, it is efficient to build the next term of the sequence in this manner. We will use the built-in *cumprod* function, which implements this operation more efficiently than a direct use of *fold*. [10]

```
ex ← function(n) function(x) sum(1,cumprod(x / n))
sapply(seq(-1,1,by=0.1), ex(0:5))
```

This changes the operation to have a linear complexity instead of $O(n^2/2)$. Improving computational performance based on mathematical transformations is an important concept and goes hand-in-hand with functional programming transformations that improve the modularity and readability of the code.

4.4.2 Multiplication of power series

Continuing with our power series example suppose we multiply two power series together.

$$\begin{aligned} f(x)g(x) &= \sum_{n=0}^{\infty} a_n(x-c)^n \sum_{n=0}^{\infty} b_n(x-c)^n \\ &= \sum_{n=0}^{\infty} z_n(x-c)^n \end{aligned}$$

where $z_n = \sum_{i=0}^{n} a_i b_{n-i}$. This problem is more challenging since it is difficult to take advantage of the recurrence relationship between terms in the sequence. With some manipulation, it is easy to see that each term can be efficiently computed based on subsequences of the coefficients of the two power series.

[10] In general the use of an internal function will be far faster than a native R implementation.

$$\begin{aligned} z_n &= \sum_{i=0}^{n} a_i b_{n-i} \\ &= \sum_{i=0}^{n} a_i b'_i \\ &= a \cdot b' \end{aligned}$$

where $b' = \langle b_n, b_{n-1}, ..., b_1 \rangle \ \forall b \in \mathbb{R}^n$. In fact each coefficient of the product is essentially a dot product of the coefficients of the operands, where one sequence is reversed. With this simplification, it is easy to see the way forward. Since the summation is fixed for each iteration at length n it is possible to create two functions: one to manage the subsequences and the other to perform the modified dot product.

```
mod.dot ← function(x, y) x %*% rev(y)
series.product ← function(a, b) {
  sapply(1:length(a), function(n) mod.dot(a[1:n],b[1:n]))
}
```

Algorithmic brute force is often used to implement equations, yet it is possible and advantageous to preserve the duality with the underlying mathematical model. The resulting simplicity clearly captures the intent of an expression in often just a few lines of code. In cases like the multiplication of power series the internal summation term is a function of two arguments while *map* takes just one. An unnamed closure is passed directly to *map* that takes the index value n and calls the dot product with length-appropriate sub sequences. Using closures to make two function signatures compatible is a common theme, as is managing subsequences.

4.4.3 Taylor series approximations

This section concludes with a final example. Building on the power series expansion, let's examine the general case of a Taylor polynomial. This example is slightly different from what we've discussed so far since the form of the coefficients is more complex. The Taylor series approximation of a function is defined as

$$f(x) = \sum_{n=0}^{k=\infty} \frac{1}{n!}(x-a)^n f^{(n)}(a)|_{x=a}$$

This is an infinite series where each term is different but has the same form. In other words the inner term of the summation operation can be represented as a function, as we've done earlier in this section. Evaluating this function for a specific case gives insights into how to transform the equation into an executable function. For practical purposes, a Taylor polynomial is used with a finite k to approximate a function. This procedure involves choosing

the number of terms k and the point a in addition to the function being approximated. Until these variables are selected, the general equation for the infinite series cannot be evaluated. Suppose $k = 7$, $a = 2$, and the function to approximate is $cos\, x$. Once these variables are bound, the equation reduces to a case-specific form

$$cos\, x = \sum_{n=0}^{7} \frac{1}{n!}(x-2)^n cos^{(n)}(2)|_{x=2}$$

that can then be evaulated for a given x. The lambda.r version of the equation begins with a function that describes the term of the summation for each n

```
function(n) (x-a)^n / factorial(n) * eval(d(f,n))
```

which mirrors the term within the summation equation. Notice that all the variables except n are *free*, meaning they must be bound in the enclosing scope. Turning this into a complete function is a matter of applying this term function over each index of the summation operator and adding the results.

```
taylor ← function(f, a, k) {
  function(x) sum(map(0:k,
    function(n) (x-a)^n / factorial(n) * eval(d(f,n))))
}
```

In the outer function the free variables are defined as arguments to this function. For completeness, we define the derivative function d as

```
d ← function(expr, order, name='a') {
  if (order == 0) return(expr)
  fold(order:1, function(junk,ex) as.expression(D(ex,name)),
      expr)
}
```

By returning a function of x, the approximation function can be used in place of the original function for any x. This is common sense since the approximation function should match the signature of the original function. The principle is to separate the specification from the application, which means distinguishing between the arguments required to produce a function and the arguments required to execute a function. Understanding this distinction is key to efficient design of closures. Hence the Taylor polynomial is specified by the variables f, a, and k, but its application only requires x. An example approximation is of $cos(x)$ about 2, plotted in Figure 4.14. This graph is constructed by first creating the Taylor function, using $a = 2$ and $k = 7$, like

```
cos2.7 ← taylor(expression(cos(a)), 2, 7).
```

This produces the closure `cos2.7` that can then be iterated over a range of x values defined by `xs ←seq(-3,5, by=0.1)`. The approximation applies this closure to the x values, `ys7 ←sapply(xs, cos2.7)`.

As in the earlier case, this implementation is inefficient for large n. Presumably the same transformation should apply for the general form, but

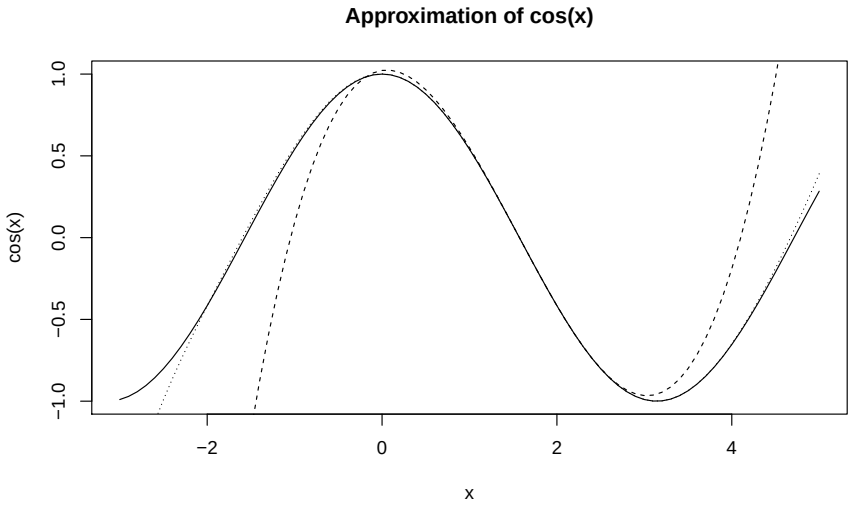


FIGURE 4.14: Approximation of $cos(x)$ about $x = 2$

what happens with the derviative term?

$$\begin{aligned} a_n &= \frac{(x-a)^n}{n!} f^{(n)}(a) \\ &= \frac{(x-a)^{n-1}(x-a)}{(n-1)!n} f^{(1)} \circ f^{(n-1)}(a) \\ &= \tilde{a}_{n-1} \frac{x-a}{n} f^{(1)} \circ f^{(n-1)}(a) \end{aligned}$$

where $\tilde{a}_{n-1} = \frac{(x-a)^{n-1}}{(n-1)!}$. For the generalized Taylor polynomial, it is not possible to simplify the derivative term since function composition is used. All is not lost, though. Since we are dealing with sequences, each sequence can be constructed independently and the terms multipled together afterward. This is particularly easy since the derivative is simply a higher-order function, so it is possible to successively apply the derivative to a function via programmatic composition. Doing this allows us to use $fold$ to perform the operation. Developing an intuition around the functional implementation requires thinking about the term construction as a function. This grounds us in the choice of arguments, which will be beneficial when designing any closures in the body.

```
function(f,a,k) {
  es ← fold(function(i, df) c(df, d(df[[length(df)]],1)), 1:k,
       list(f))
  sapply(es, function(e) eval(e))
}
```

The extra machinery in this function is related to type handling of expression objects. The actual composition is simply taking the derivative of the last element in the sequence. The term for the $\tilde{a}$ is simpler and follows a similar pattern as when implementing the Maclaurin series.

```
function(a,n) function(x) sum(1,cumprod((x-a) / n))
```

When we assemble the code into a new function, it is not necessary to include the function definitions as provided above. The value of that exercise is that the argument sets are explicit. It is also apparent that to make the argument lists a formal subset of each other (instead of a union), `n` can be constructed from `k` as `n ←0:k` [11].

```
taylor.f ← function(f, a, k) {
  es ← fold(function(i, df) c(df, d(df[[length(df)]],1)), 1:k,
       list(f))
  f.tilde ← sapply(es, function(e) eval(e))
  function(x) sum(c(1, cumprod((x-a) / (1:k))) * f.tilde)
}
```

This new implementation highlights an important tradeoff in function design: efficiency for complexity. This is a fundamental concern of programming as codified by Kernighan: "Controlling complexity is the essence of computer programming" [6]. In our case we improved the computational efficiency of the code with a complex transformation. While the transformation is grounded in a mathematical transformation the implementation begins to diverge. This can be dangerous and efficiency transformations should be tested and verified that they actually provide a tangible gain for the majority of use cases.

4.5 Exercises

Exercise 4.1. In Example 4.3.2, the `full_join` function duplicated the primary key column. Update the function to remove the redundant column.

Exercise 4.2. Update `full_join` to support left and right outer join

Exercise 4.3. Implement the factorial function as a fold process. Can you add make the function more efficient by prioritizing certain arguments over others?

Exercise 4.4. Use a transformation specification to to implement the `is_consistent` test for all cumulative columns in `data.lr`.

[11] The exception is that 0! cannot be deconstructed recursively, so the zero case is provided explicitly. The consequence is that `n ←1:k`.

Exercise 4.5. Use an ordinal map to apply the spelling corrections on county names.

5

Filter

PENDING

6

Lists

Vectors are an obvious choice when all data have the same type, such as for a time series or a collection of samples from the same population. Real-world data often comprise numerous data series, each with their own distinct type. In other situations, a collection of elements of varying type need to be grouped together. These cases warrant the use of the `list` type, which is like a tuple and can hold values of arbitrary type. From the perspective of functional programming, lists provide both a generic container for iteration and accumulation as well as a mechanism for dynamic function calls via argument packing. Many list operations are consistent with vectors, although there are a few important semantic differences. Section 6.1 investigates the basic properties and behaviors of lists. Afterwards, we consider what it means to compare lists in Section 6.2. Relations are the formal structures describing comparisons and form a rich body of work on their own. Relations can be used to order complex structures as well as provide machinery to transform these structures into metric spaces.

Once we establish this foundation, we are ready to use lists as both an iterable container and as a result container in *map* and *fold* operations, which appear in Sections 6.3 and 6.4, respectively. Lists can also be used to pack function arguments, providing a mechanism for dynamically constructing and calling functions via `do.call`. The mechanics and benefits of this process are described in Section 6.5.

The primary difference between lists and vectors is that lists can contain heterogeneous data types. Vectors are also restricted to primitive types, while lists can hold any data structure including functions, data frames, and other complex data structures[1]. Lists can even contain other lists, providing a mechanism for creating hierarchical data structures. With JSON being a popular serialization format for structured data, deep list structures are becoming more commonplace. It is important to remember that although a list can emulate these data structures they are not the same as these data steuctures. This is most important when thinking about lists as dictionaries. Dictionaries typically have $O(1)$ lookup and insert cost but the order of keys/values is undefined. On the other hand, a list has variable lookup cost (depending on how elements are accessed) and $O(n)$ insert cost. However, ordering is pre-

[1] This generality also forms the basis for `data.frames`.

served. [2] Section 6.6 details the considerations for emulating data structures with lists.

6.1 Primitive operations

To take advantage of lists we need to know how to create and manipulate them. As with a vector, let's begin by formally defining the list data type.

Definition 6.1.1. A *list* is an ordered collection of elements equivalent to a finite tuple. A list x with n elements is denoted as $list(x_1, x_2, ..., x_n) \equiv \langle x_1, x_2, ..., x_n \rangle$.

(a) Elements of a list are indexed by a subset of the positive integers $\mathbb{N}_n = 1, 2, ..., n$.

(b) For list x, $x[[k]] \equiv x_k$ is the k-th element of x.

(c) For list x, $x[k] \equiv \langle x_k \rangle$ is the list containing the k-th element of x.

(d) The *length* of a list is its cardinality and is denoted $length(x) \equiv |x|$.

Throughout this section, we'll see how to apply this definition to effectively use lists.

6.1.1 The list constructor

Scalars are implicitly vectors. It is trivial to create an arbitrary vector of length one simply by specifying a literal value. As discussed in Chapter 2, there are numerous ways to create longer vectors. One such approach is to use the concatenation operator `c`, which colloquially acts as the vector constructor. This syntactic convenience is not available to lists, so they must be explicitly created using the list constructor. An empty list is simply `list()`. Like vectors, a list can have both named or unnamed elements. The ebola configuration contains both styles:

```
config.a ← list(min.version=175, max.version=196,
  sections=2,
  start=list('Ebola Case and Death Summary by County',
    'Healthcare Worker HCW Cases')
  ).
```

The elements of this list are all named, but the list assigned to `start` is unnamed. Later in this chapter we'll see how this affects the choices for element access in the list.

[2] For hash-like behavior in R, users can use environments.

Most list operations are consistent with vectors. However, subtle differences can arise given the distinct constructor and concatenation functions. One such difference is that the list constructor is not idempotent. Hence, calling the constructor multiple times will result in a deep list.

Example 6.1.1. A simple list is `a ←list('a', 3)`. Its structure is

```
> str(a)
List of 2
 $ : chr "a"
 $ : num 3
```

Repeated application of the list constructor produces a deep list with `a` at the terminal node. This list has a different structure.

```
> str(list(list(list(a))))
List of 1
 $ :List of 1
  ..$ :List of 1
  .. ..$ :List of 2
  .. .. ..$ : chr "a"
  .. .. ..$ : num 3
```

□

It may seem inconvenient that lists are not idempotent, yet the lack of this property contributes to the list's utility. Section 6.6 discusses emulating the tree data structure with lists precisely because the list constructor is not idempotent.

Another departure from vectors is that an empty list is not equivalent to `NULL`. This distinction can be problematic for code that checks for `NULL` return values via `is.null`. We'll see in Section 6.1.6 that this inconsistency can be problematic when the result of a function can be either a vector or a list. Functions often return `NULL` instead of throwing an error, and it becomes almost natural to check for a `NULL` result. Since `NULL` is defined to have cardinality of 0, a more consistent approach is to check for a non-zero length. That way, a single test can capture either result.

Example 6.1.2. To illustrate the properties of a `NULL` value, let's look at a function that returns a whole number or `NULL` otherwise, such as

```
whole ← function(x) if (x < 0) return(NULL) else x.
```

For a negative integer, this function returns `NULL`.

```
> is.null(whole(-1))
[1] TRUE
> length(whole(-1))
[1] 0
```

We can redefine the function to return an empty list instead:

`whole ← function(x) if (x < 0) return(list()) else x.`

The result still has length 0 but is now non-null.

```
> is.null(whole(-1))
[1] FALSE
> length(whole(-1))
[1] 0
```

□

Speaking of `NULL`s, although vectors cannot contain `NULL` values, the same is not true of lists. When creating a list, any element can be defined as `NULL`, such as `z ←list(3, NULL, 5)`. This property is particularly useful in *map* operations since it provides a way to preserve cardinality, which is discussed further in Section 6.3.2. While a useful feature, working with `NULL`s can be tricky. We'll see in Section 6.1.6 some idiosyncracies related to working with `NULL` elements.

6.1.2 Raw element access

Raw element access is governed by the double bracket operator, defined in Definition 6.1.1(b). This operator supports both positional and named arguments. Positional arguments are the most basic and work like vector indexing except double brackets are used instead of single brackets. When elements of a list are named, a string can be used inside the double bracket operator as well.

Example 6.1.3. Using the previously defined list `config.a`, we can show that named indexing is equivalent to positional indexing.

```
> config.a[["version"]] == config.a[[2]]
[1] TRUE
```

□

A more convenient syntax for single element access uses the `$` operator. This notation is equivalent to using a character string within double brackets.

Example 6.1.4. The `$` operator yields the same value as a character index.

```
> config.a$version == config.a[["version"]]
[1] TRUE
```

□

Example 6.1.5. An index that extends beyond the length of the list will result in an error.

```
> config.a[[5]]
Error in config.a[[5]] : subscript out of bounds
```

Note that this is not true of a non-existent named index. In this case, the result is `NULL`, so no error is raised.

```
> config.a[['country']]
NULL
```

This is the case irrespective of whether double brackets or the `$` operator is used.

□

The double bracket operator enforces a single result, so an input with length greater than one has behaves differently from subset selection. Using a strictly vector argument (i.e. cardinality greater than 1) with raw element access results in recursive indexing. This means that the second element of the index vector will be used as the index to the result of the first indexing operation, and so on.

Example 6.1.6. What happens if we try to access multiple elements using double brackets?

```
> x ← list(1:2, list(3,4,5), 6:7)
> x[[c(2,3)]]
[1] 5
```

The resulting value is the third element of the list at the second index of the original list `x`. Note that recursive indexing works for both list and vector elements.

```
> y ← list(1:2, c(3,4,5), 6:7)
> y[[c(2,3)]]
[1] 5
```

□

Example 6.1.7. Recursive indexing works with named lists as well. Using the same list as in Example 6.1.6, let's recreate the list with names.

```
> x ← list(a=1:2, b=list(x=3,y=4,z=5), c=6:7)
> x[[c('b','z')]]
[1] 5
```

□

Example 6.1.8. Recursive indexing only works when the list has a structure compatible with the index. If the list is not compatible, an error is raised. Suppose we attempt to extract the fourth element from the list at `x[[2]]`.

```
> x[[c(2,4)]]
Error in x[[c(2, 4)]] : subscript out of bounds
```

□

6.1.3 Selecting subsets of a list

We've seen how double brackets are used to access a single underlying value in a list. The result is the raw value with no embellishments. On the other hand, when selecting multiple elements from the list, the result cannot simply be the raw elements. Some container must hold these values and the obvious choices are a vector or a list. If more than one element is accessed, it's not always possible to collapse multiple list elements into a vector. Since lists can contain heterogenous types a separate mechanism must be used that necessarily returns a list. This is different from vectors, where both operations return vectors. It's convenient to think about the single bracket operator of Definition 6.1.1(c) as extracting a subset from a list. This terminology implies that a list is a set, and the result of subsetting is another list. Subsetting operations work the same as vector subsetting. In other words, numeric, character, and logical vectors all specify a subset. Even if the subset has length one or zero, a list is always returned.

Example 6.1.9. We know that double brackets are used for raw element access. What happens when a single selector is specified with sublist notation? The result is naturally a list of length one.

```
> str(a[2])
List of 1
 $ : num 3
```

Compare this to the double bracket operator, which returns the underlying scalar.

```
> str(a[[2]])
 num 3
```

□

Selecting non-trivial sublists requires using vectors with length greater than one. If the vector has a logical type, then the cardinality must match that of the list[3], just like with vectors. On the other hand, integer and character vectors can be of arbitrary length.

Example 6.1.10. A subset of the English alphabet can be created by specifying a set of integer indices. First we convert the built-in `letters` variable to a list.

```
> list.letters ← as.list(letters)
> list.letters[sample.int(length(letters), 3)]
[[1]]
[1] "h"

[[2]]
[1] "d"
```

[3] with the exception of recycling rules

```
[[3]]
[1] "e"
```

□

Note how the printed output displays indices surrounded by double brackets. This indicates that the subsequent value is the actual value (in this case a vector) as opposed to being wrapped in a list.

Example 6.1.11. As with vectors, numeric indices provide a means for ordering lists. The alphabet can be reversed by reversing the indices of the original list.

```
> head(list.letters[26:1], 3)
[[1]]
[1] "z"

[[2]]
[1] "y"

[[3]]
[1] "x"
```

□

6.1.4 Replacing elements in a list

Over the course of an analysis, a list might change for a number of reasons. For example, if a list is holding state, then as the state changes, the list must change to reflect that. Updating a single element in a list follows the semantics of double bracket notation. In other words, the general form is `x[[i]] ←y`. While the indexing rules are the same as selection, typically a scalar integer or string is used as opposed to a full logical vector.

Example 6.1.12. Returning to the ebola configuration object defined in Section 6.1.1, we can update the version by assigning the selection to a value.

```
config.a$min.version ← 170
```

□

Sublist notation is also used to replace a subset of list elements with new values. This operation can be considered a form of element assignment for sets. The syntax follows the syntax for sublist selection except that we assign a value to the selection instead of returning it. For this to work, both the selection and the replacement must have compatible cardinality.

Example 6.1.13. Suppose we want to add a third section to our parse configuration shown in Section 6.1.1. We can replace multiple elements so long as the cardinality of the replacement is compatible with the cardinality of the selection.

```
config.a[c('sections','start')] ← list(3, list(
  'Ebola Case and Death Summary by County',
  'Healthcare Worker HCW Cases', '^Newly'))
```

Notice that the replacement list does not need to be named since the ordering has already been defined by the selection.

□

6.1.5 Removing elements from a list

Variables in R are deleted using the `rm` function. However, this doesn't work for elements within a data structure. In this case, an element is removed by assigning its value to `NULL`. The curious reader may wonder whether single or double bracket notation should be used. Happily, the answer is "yes". In other words, both operations have the same behavior.

Example 6.1.14. What happens to indexes after deleting an element? The values are shifted so that the integer indices are contiguous.

```
> list.letters[1] ← NULL
> head(list.letters, 3)
[[1]]
[1] "b"

[[2]]
[1] "c"

[[3]]
[1] "d"
```

We see that the indices have been shifted so that there are no gaps. This behavior is expected and has known implications on algorithm complexity. The change in ordinal structure also has implications if there is an implicit mapping between the list's ordinals and another object. It's important to consider whether ordinal correspondence needs to be preserved across objects. If so, then a `NULL` can be inserted into the list as opposed to removing the element from the list.

□

6.1.6 Lists and NULLs

Earlier in the chapter, we hinted at some of the idiosyncracies associated with `NULL`. The `NULL` is truly a curious entity in R. In some ways it is like a value

but not in others. We saw in Chapter 2 that an empty vector is equivalent to `NULL`. While assigning `NULL` to a variable will create the variable, doing the same to a list element will remove that element from the list. Yet, when creating a list, `NULL`s are legal and increase the cardinality of the list. What if you want to add a new element with a value of `NULL`? Using the element access form does not work since the effect is to remove the not-yet defined element. Instead we must leverage our understanding of list accessors. Since we can create a list with `NULL` elements, the trick is to assign a single element sublist to a list with a single `NULL` element. By wrapping the `NULL` in a list, it is preserved and the rules of subset assignment apply.

Example 6.1.15. Assigning a `NULL` to a list element removes the element from the list. To replace a list element with `NULL` requires subset notation.

```
> letters.list[2] ← list(NULL)
> head(letters.list)
[[1]]
[1] "a"

[[2]]
NULL

[[3]]
[1] "c"
```

□

6.1.7 Concatenation

Concatenation for lists works the same as for vectors, so elements can be added to lists using `c()`. Both lists and vectors can be combined in concatenation to produce a single list. Any number of lists will concatenate into a single list.

```
> c(list(1,2), list(3,4))
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

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

List concatenation is a greedy operation, which means that the result of `c()` will be a list if any of the operands is a list. Under this circumstance the concatenation operator acts like the list type constructor:

$c(y_1, ..., y_{k-1}, x, y_{m+1}, ..., y_n) = list(y_1, y_2, ..., y_n)$, where $x = list(y_k, ..., y_m)$. Concatenating vectors with a list also results in a single list. Vector elements are treated like lists since each element of a vector becomes an element of the resulting list.

```
> c(1, list(2), 3:4)
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

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

However, list concatenation preserves list structure, so deep lists will not be flattened.

```
> c(list(1,2, list(3)), 4)
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[[3]][[1]]
[1] 3


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

The preceding operations show that concatenation behaves the same as vector concatenation when there are no deep lists. This behavior extends to lists containing complex types like data frames.

6.2 Comparing lists

Since lists can hold arbitrary objects, there is no universal relation that applies to all lists. In other words, lists are not generally comparable. Attempting to do so results in an error. This is not to say that specific lists cannot be compared. Indeed, for specific lists it is easy to compare them. First we need to decide what we are comparing. Do we only care about equality or do we need some

ordering for the list? Second, it's important to know the types involved. Lists containing only numbers are easy to compare since they can be treated as metric spaces. If this is not the case, then non-numeric data must either be encoded as numbers or thrown away in a transformation process.

6.2.1 Equality

Testing for equality is the simplest comparison operation since it only requires two conditions. First, the list structures must be the same. Not only do the lengths need to match, but list elements also need to have compatible cardinality. Similar to the difference between permutations and combinations, equivalence can either ignore or account for order. We can also talk about two types of ordering. When ordering does not matter at all, equality reduces to set equality. When ordering does matter, typically we think of this in terms of the ordinals of each set. This is classic element-wise comparison. A middle ground is relative ordering, where the order within a set does not matter, but each set must have the same ordering. This may seem cryptic but in fact is the situation when lists are named. For comparison purposes it doesn't matter in which order the keys are specified. Yet once this ordering is fixed, each list will be compared element-wise based on the given ordering.

Example 6.2.1. List elements do not need to be named. When this is the case, lists can only be compared if an explicit ordering is provided. Are lists `a` and `b` equal or not?

```
> a ← list(211, max.version=215)
> b ← list(max.version=215, 211)
```

□

Example 6.2.2. Even if all elements are named, what if the names have different orders? For example, are lists `a` and `b` equal? The answer depends on how the ordering is defined.

```
> a ← list(min.version=211, max.version=215)
> b ← list(max.version=215, min.version=211)
```

If we ignore order, then we can consider the lists to be the same. However if we care about order, then they are not considered equal.

```
> apply(rbind(a,b), 2, function(x) x[[1]] == x[[2]])
min.version max.version
      FALSE       FALSE
```

□

Example 6.2.3. A simple way to test set equality is to order the values and then do an element-wise comparison. This can be implemented most simply as a vector operation. For named lists, this just means iterating over the labels and testing equality of each corresponding element.

```
> sapply(names(a), function(i) a[[i]]==b[[i]])
min.version max.version
       TRUE        TRUE
```

Alternatively, the second list can be sorted based on the labels of the first and equality tested that way.

```
> b = b[names(a)]
> apply(rbind(a,b), 2, function(x) x[[1]] == x[[2]])
min.version max.version
       TRUE        TRUE
```

□

6.2.2 Orderings

Aside from equality, ordering relations themselves are important. An ordering is what tells us that 59 km/hour is faster than 47 km/hour. For numeric quantities, measures like magnitude or length are often used to order elements. Orderings are also closely related to a ranking, which provides a means of indicating priority. Orderings are obvious for numbers, but what about structures? For example, is $(5, 3)$ bigger or smaller than $(4, 4)$? Again, it depends on how we define the ordering. The astute reader may recognize these tuples as a Euclidean space, in which case, the L^2 norm can be used to define an ordering. But we are not beholden to this particular ordering. In other situations, this type of ordering is not possible. Consider the telephone book. Our two tuples now have the opposite order if we use alphanumeric ordering.

If an ordering is not fixed, how can we define the ordering? When the list structure is known to be constant, the ordering can be defined explicitly. One example is a simple social network graph, where the ordering is dependent on a node's relationships within the graph.

Example 6.2.4. A social network is characterized by a graph where nodes represent users and edges represent connections. A very simple model for the network represents each user as a tuple $(id, friends)$, where $friends$ is itself a tuple of user ids. With this structure, it is possible to calculate the degree centrality of each user **[?]**, which can be used to order the tuples.

```
set.seed(112514)
n ← 50
graph ← lapply(1:n,
  function(i) list(id=i, friends=sample(n, sample.int(n))))
```

The normalized degree centrality for a given user is $\frac{|friends|}{N-1}$, where N is the number of nodes in the network. We can define a transformation that encodes the ordering relation.

```
degree ← function(x) length(x$friends)
d ← sapply(graph, degree)
```

To apply the ordering, we use the built-in `order` function.

```
graph ← graph[order(d)]
```

We can verify that the graph is now ordered from smallest to largest degree centrality by applying the `degree` function once more.

```
> sapply(graph, degree)
 [1] 0.08163265 0.10204082 0.10204082 0.12244898 0.16326531
 [6] 0.16326531 0.18367347 0.22448980 0.24489796 0.32653061
[11] 0.32653061 0.34693878 0.36734694 0.38775510 0.40816327
[16] 0.40816327 0.42857143 0.44897959 0.44897959 0.46938776
[21] 0.46938776 0.53061224 0.53061224 0.59183673 0.59183673
[26] 0.61224490 0.63265306 0.65306122 0.67346939 0.71428571
[31] 0.71428571 0.71428571 0.73469388 0.73469388 0.73469388
[36] 0.75510204 0.75510204 0.85714286 0.85714286 0.87755102
[41] 0.87755102 0.91836735 0.91836735 0.93877551 0.97959184
[46] 0.97959184 1.00000000 1.00000000 1.00000000 1.02040816
```

□

6.2.3 Metric spaces, distances, and similarity

In the previous section we mentioned using magnitude as a way to order elements. Magnitude is measured from some fixed origin. If we replace that fixed origin with another element, magnitude starts to look like distance. This in turn can be considered a measure of how similar two elements are. The formal concept of distance comes from real analysis. A metric space is a set that has a distance measure defined for all elements in the set [**?**]. Euclidean space is the canonical metric space, where physical distance is defined between points in the real world. This concept can be extended to arbitrary data structures insomuch that values can be encoded as numbers. Treating a data structure as a metric space is a cheap approach to defining the distance between two elements within the set. From a formal perspective a distance measure must satisfy four properties as listed in Definition 6.2.1. By constructing some function d to satisfy these requirements, distance can be quantified for non-numeric data.

Definition 6.2.1. A metric (distance) d on a set X is a function $d : X \times X \to [0, \infty)$ that satisfies the following conditions.

(a) $d(a, b) \geq 0$

(b) $d(a, b) = 0$ iff $a = b$

(c) $d(a, b) = d(b, a)$

(d) $d(a, c) \leq d(a, b) + d(b, c)$

Distance is useful because it can act as a proxy for similarity. Unlike the concept of size, distance accounts for position and tells us how close two objects are. We call this notion of closeness *similarity*, which is conceptually the inverse of distance.

Example 6.2.5. In document analysis, documents are often encoded as vectors of word counts, or term frequencies. For example, we can analyze different stanzas in the poem *A Tree For Me* [**?**], shown in Figure 6.1. We'll consider each stanza as a document. To encode the stanzas, we first need to remove all punctuation and capitalization. As an example, the second stanza will become the string `one owl nesting golly gee no room for me in this ol tree`. To transform this into a vector of word counts, we can split the string on whitespace and then use the `table` function to count up the word occurrences. To get started we create a vector where each element represents a stanza.

```
poem ← c('All around the hill where the brook runs free, I
    look, look, look for a tree for me. Big one, small one,
    skinny one, tall one, old one, fat one, I choose that one!
    ',
  "One owl nesting, golly gee! No room for me in this ol'
     tree.",
  'All along the brook, frogs peep, "Chip-chee!" as I look,
     look, look for a tree for me. Big one, small one, skinny
      one, tall one, old one, fat one, I choose that one.',
  "Two possums dangling, golly gee! No room for me in this ol'
      tree.")
```

Next we create a function that removes formatting and punctuation, leaving lower case words only.

```
prepare ←
  function(s) strsplit(gsub("[\"',.!]","", tolower(s)), " ")
```

After normalizing the poem, the next step is to count the term frequency of each stanza.

```
words ← prepare(poem)
counts ← lapply(words, function(x) table(x))
```

The term frequencies are independent of each other, but the analysis requires a uniform set of vectors. To construct the document matrix, we begin by initializing a matrix with all zeroes. Each stanza is represented by a column of the matrix. We use matrix selection to assign the appropriate terms with their counts.

```
all.words ← unique(names(do.call(c, counts)))
ncol ← length(counts)
```

All around the hill where the brook runs free,
I look, look, look for a tree for me.
Big one, small one, skinny one, tall one,
old one, fat one, I choose that one!

One owl nesting,
golly gee!
No room for me
in this ol' tree.

All along the brook, frogs peep, Chip-chee!
as I look, look, look for a tree for me.
Big one, small one, skinny one, tall one,
old one, fat one, I choose that one.

Two possums dangling,
golly gee!
No room for me
in this ol' tree.

FIGURE 6.1: Excerpt from *A Tree For Me*

```
mat ← matrix(rep(0, length(all.words)*ncol), ncol=ncol)
rownames(mat) ← all.words
lapply(1:ncol,
  function(i) mat[names(counts[[i]]),i] ↞ counts[[i]])
```

The resulting vectors can be treated as being in an n-dimensional Euclidean space, which means we can calculate the distance between two stanzas within the poem.

```
distance ← function(a,b) sqrt(sum((a-b)^2))
```

Anecdotally, we expect the first and third stanzas to be similar and the second and fourth to be similar. Indeed, the distance for $(1, 2)$ is greater than both $(1, 3)$ and $(2, 4)$. This implies that we've successfully created a useful metric for the stanzas of the poem.

```
> distance(mat[,1], mat[,2])
[1] 8.888194
> distance(mat[,1], mat[,3])
[1] 3.316625
> distance(mat[,2], mat[,4])
[1] 2.44949
```

While Euclidean distance is convenient to compute, it is also unbounded. Hence, it is difficult to know how similar one document is to another. This is why cosine similarity [**?**] is often used as an alternative, since the codomain is $[-1, 1]$, where 1 means exactly the same and -1 means exactly opposite. The

cosine similarity is simply the cosine of the angle separating the two vectors: $cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$. The equivalent code in R is

```
similarity ←
  function(a,b) (a %*% b) / (distance(a,0) * distance(b,0)).
```

Applying the similarity measure on the same stanza pairs yields different values.

```
> similarity(mat[,1], mat[,2])
           [,1]
[1,] 0.3252218
> similarity(mat[,1], mat[,3])
          [,1]
[1,] 0.936557
> similarity(mat[,2], mat[,4])
           [,1]
[1,] 0.7692308
```

Despite the differences, the cosine similarity confirms the closeness of $(1,3)$ and $(2,4)$. However, it tells us that $(1,3)$ is more similar than $(2,4)$, which contradicts the Euclidean distance. The reason is that the cosine ignores the magnitude of the vectors, which affects the interpretation of similarity. Returning to document space, this means that stanzas with many word repetitions are considered similar to those without. These subtleties are something to be aware of when encoding a space.

□

6.2.4 Comparing other spaces

Even when your data cannot be treated as a metric space, the concept of a distance is still useful. In many domains this is the bridge between non-numeric data and orderings. We're talking about orderings based on a relation, as an absolute ordering based on magnitude might not exist. To make this concrete, let's see how we can construct a distance measure for sets. Suppose you are at the grocery store and encounter a friend. Upon looking at her basket, you exclaim that you two have similar shopping habits. How do you know that your two baskets are similar? One approach is to count the number of overlapping items, like apples and oranges versus the total number of distinct items in both carts. Mathematically, this is expressed as $JS(A,B) = \frac{\|A \cap B\|}{\|A \cup B\|}$ and is known as Jaccard similarity [**?**]. This measure is not a distance, since taking the Jaccard similarity of two identical baskets does not equal 0. It can easily be converted to a distance measure with the transformation $JD = 1 - JS$. It's important to remember that like cosine similarity, this measure is a relation, so a standalone ordering based on magnitude is not possible.

Example 6.2.6. Using the graph produced in Example 6.2.4, let's find the

two most similar users in the network. To do this we need to calculate the similarity across all pairs of users.

```
jaccard ← function(a,b)
  length(intersect(a,b)) / length(union(a,b))
pairs ← combn(1:n,2)
sim ← apply(pairs, 2, function(x)
  jaccard(graph[[x[1]]]$friends, graph[[x[2]]]$friends))
```

As with degree centrality, the result is a vector that tells us the similarity of each pair of users.

□

6.3 Map operations on lists

Like vectors, the *map* operator can be applied to lists. We've already seen numerous examples of using *map* with lists, and this section will explore the mechanics of the operation. Typically `lapply` is a starting point as it both accepts and returns lists. Given the choice, when is it useful to iterate over a list instead of a vector? The lazy answer is "when the iterable has heterogeneous elements". This is only half the story though. The real value of `lapply` is the ability to both operate on complex data structures as well as return complex data structures. Complex data structures can be hierarchical, but they can also be sequences of non-atomic data, such as functions or data frames. This flexibility means that `lapply` can be used in all sorts of applications.

Continuing our discussion on the Jaccard similarity (Example 6.2.6), how did we apply the measure to every pair of elements in our set? In other words, how do `combn` and `apply` work together to produce the similarities? All that's really happening is that index pairs are constructed and then used to extract a pair of users from the `graph` object. calling the `jaccard` function. The Jaccard similarity is thus applied to each pair of users in the graph. This only works when the ordinals are fixed. Once the pairs are constructed, a *map* operation is used to compute the similarity on the given pair. In a non-functional approach, the indices and the call would normally be commingled in a single loop. This interplay between *map* operations and ordinals appears regularly as discussed in Chapter 2 and plays a prominent role when ranking users based on similarity.

Example 6.3.1. The initial order of the vector of similarities is based on the pairs of user indices produced by `combn`. How can we rank the similarities and also tie back to the underlying pairs of users? If we take the same sorting approach as we did in the cosine similarity example, we lose this relationship. Preserving the ordinal correspondence between the user pairs and the

similarities is thus paramount. Let's split the operations and assign the result of `order` to a new variable, say `ranking ←order(sim, decreasing=TRUE)`. The user pair with the highest Jaccard similarity is the first element of this new vector, which corresponds to the 1222th column of the pairs matrix.

```
> ranking[1]
[1] 1222
> pairs[,ranking[1]]
[1] 47 50
```

It appears that users 47 and 50 have the highest Jaccard similarity, with a value of

```
> sim[ranking[1]]
[1] 0.98.
```

Both the ranking process and the actual ordering process are *map* operations. In the first case, ordinals representing the ranking are given as input, and the output is a new set of ordinals. For the ordering, a set of ordinals are the input with actual values as the output. This is equivalent to `map(ranking, function(i)sim[i])`, which is obscured by the vectorized indexing operation.

□

6.3.1 Applying multiple functions to the same data

Typically we think of *map* as applying the same function to a set of values. What about applying a set of functions to the same value? This may seem strange, but it's actually quite common. An example of this is calculating summary statistics on data. The data are constant while different functions are applied to get the statistics of choice. Similarly, when evaluating the performance of a model, multiple metrics are used against the same set of predicted and actual values. The same concept can be applied to model selection as well, where variations on model parameters can be captured in closures and then run on the same dataset. A good reason for doing this is that it's easy to parallelize, since each model specification is independent.

Example 6.3.2. Suppose we want basic statistics such as the mininum, maximum, mean, median, and standard deviation. A somewhat naive approach is to explicitly calculate each of these statistics and manually add them to a list.

```
stats = list()
stats$min = min(x)
stats$max = max(x)
```

An alternative is to use a list to hold all the functions and apply each function to the vector. To make this work we take advantage of the fact that functions are first class. So each iteration of *map* is passing a function reference to the

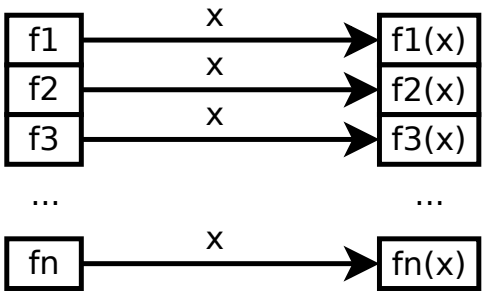


FIGURE 6.2: Using a list, multiple functions can be applied to the same object via `lapply`

function argument of *map*. This function in turn is responsible for applying the given function to the vector argument.

```
fs = list(min, max, mean, median, sd)
lapply(fs, function(f) f(x))
```

What's nice about this approach is that it's so concise. It is also obvious how to change the set of summary statistics used.

□

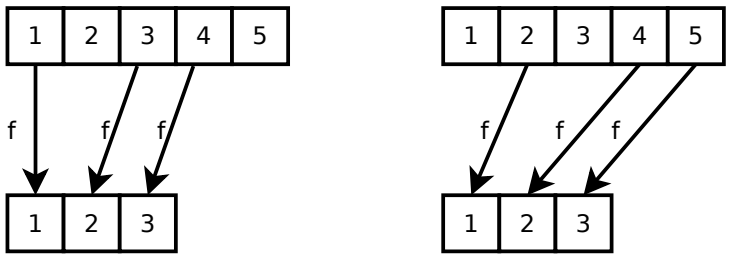
6.3.2 Cardinality and null values

Some of the stock map-like functions appear non-deterministic in their output. At times the expected vector or matrix result ends up as a list. This happens most often with `sapply` as it is designed to "simplify" the output. It does this by using heuristics to reduce lists to vectors and matrices where possible. Due to the modality rules of the core data structures, a list can be returned in order to preserve indices. The irony is that by changing the output type to preserve determinism, the result is that `sapply` appears non-deterministic. Let's see how this happens.

```
> sapply(-1:2,
+   function(x) { if (x <= 0) return(NULL); log(x) })
[[1]]
NULL

[[2]]
NULL

[[3]]
[1] 0
```



(a) A mapping from one set to another (b) Or is this the correct mapping?

FIGURE 6.3: When cardinality is lost, the ordinals are also lost

```
[[4]]
[1] 0.6931472
```

Why is a list the final result? As the most generic (excepting environments) data structure lists can accommodate any value (or non-value) as an element. If the result of a function is `NULL`, a list will happily store this as an element. If lists weren't able to do this, then we wouldn't know how the valid values map to the input values.

Suppose we're obstinate and want to use a vector instead. What would the result look like? Based on the rules of vector concatenation, we know that $c(x, NULL) = x$, which means the value is not stored. When iteratively processing data, how do you know which index produced the `NULL`? This problem is illustrated in Figure 6.3, where a hypothetical function f takes a vector of length five and returns a vector of length three. From a mathematical perspective, f is discontinuous since output is undefined at certain points. Here's the conundrum: which of the two mappings is the correct mapping of the input vector to the output vector? Sadly, without the ordinal mapping, we cannot know. The only way to discover the undefined point is to evaluate (virtually) all input elements and discover which values yield `NULL`. This is one motivation for the special `NA` value, which acts as a placeholder to preserve both the cardinality and ordinality over a vectorized operation.

The fact that lists can hold `NULL`s guarantees ordinal correspondence between the input and the output. The lesson is that whenever you require ordinals to be preserved, it's imperative that a list is returned, which can be guaranteed by `lapply`.

6.3.3 Hierarchical data structures

Although far from perfect, the CSV format has long been the king of file formats for data analysis. Its regular structure is well suited to many types of analysis, and its ubiquity means any programming language can read CSVs.

A more recent newcomer is JSON, which has effectively replaced XML. A natural question is whether JSON will replace CSV as well? The answer is no, because they are two different beasts. Unlike CSV, JSON is used to represent hierarchical data structures. While tables can be represented this way, it is not as efficient as CSV. On the other hand, when a dataset contains arbitrary embedded structures, it is difficult to represent this in a table. In short, JSON is useful for representing complex data structures, but it's less useful for data analysis. This implies there's work to be done to transform JSON structures into a vector or data frame. In this section, we construct a vector based on multiple fields within a JSON structure. In Chapter 7, we'll extract bits and pieces of a JSON object to construct a data frame.

Most data coming out of a web service is represented as JSON. The reason is that the hierarchy provides context. For example, social media data from Twitter and message data from Slack are represented as JSON.

Example 6.3.3. Let's investigate the voting behavior of the US legislature. This data is available via the `govtrack.us` API, which returns results as JSON. In R this JSON object is converted into a hierarchical list structure. This hierarchical structure is not useful for analysis, but it can be convenient to keep the structure around to access different fields ad hoc. A simple operation is to extract the vote count for a given bill. A (small) portion of this data structure appears in Figure 6.4. Each entry in the data structure is an object that contains multiple fields. Some of these fields are objects themselves.

An arbitrary bill is H.R. 37, known as the "Promoting Job Creation and Reducing Small Business Burdens Act". What is the vote tally for this bill? The answer is obtained by performing a *map* operation on the list structure. Since each JSON element is named, it's quite easy to access embedded list elements, such as the `value` field from the `option` object.

```
library(httr)
uri ← 'https://www.govtrack.us/api/v2/vote_voter/?vote=116111&
    limit=500'
json ← content(GET(uri))
tally ← sapply(json$objects, function(o) o$option$value)
```

The `tally` consists of the votes for each member of the House.

```
> table(tally)
tally
       Nay Not Voting        Yea
       146          5        276
```

A follow-up question is how each member voted on the bill. Using `sapply` once more, we need to return a vector result. This is no big deal, since both values are characters. In other cases, it's necessary to return a data frame, which we know will result in a list container.

```
votes ← t(sapply(json$objects,
  function(o) c(name=o$person$name, vote=o$option$value)))
```

```
{
 "meta": {
  "limit": 500,
  "offset": 0,
  "total_count": 415
 },
 "objects": [
  {
   "created": "2015-01-06T18:31:00",
   "id": 30546177,
   "option": {
    "id": 444962,
    "key": "+",
    "value": "Yea",
    "vote": 116108
   },
   "person": {
    "bioguideid": "A000055",
    "birthday": "1965-07-22",
    "cspanid": 45516,
    "firstname": "Robert",
    "gender": "male",
    "gender_label": "Male",
    "id": 400004,
    "lastname": "Aderholt",
    "link": "https://www.govtrack.us/congress/members/robert_
       aderholt/400004",
    "middlename": "B.",
    "name": "Rep. Robert Aderholt [R-AL4]",
    "namemod": "",
    "nickname": "",
    "osid": "N00003028",
    "pvsid": "441",
    "sortname": "Aderholt, Robert (Rep.) [R-AL4]",
    "twitterid": "Robert_Aderholt",
    "youtubeid": "RobertAderholt"
   },
   "person_role": {
    "caucus": null,
    "congress_numbers": [
     114
    ],
    "current": true,
    "description": "Representative for Alabama's 4th
       congressional district",
    "district": 4,
    "enddate": "2017-01-03",
    "extra": {
     "address": "235 Cannon HOB; Washington DC 20515-0104",
     "contact_form": "http://aderholt.house.gov/email-me2/",
```


FIGURE 6.4: An excerpt of a JSON structure from the `govtrack.us` API

While the output of `sapply` is indeed simplified, the result has dimensions 2×415. This is due to vector results being appended as columns. A transpose operation makes the result slightly more readable.

```
> head(votes)
     name                              vote
[1,] "Rep. Robert Aderholt [R-AL4]"  "Yea"
[2,] "Rep. Joe Barton [R-TX6]"       "Yea"
[3,] "Rep. Xavier Becerra [D-CA34]"  "Nay"
[4,] "Rep. Rob Bishop [R-UT1]"       "Yea"
[5,] "Rep. Sanford Bishop [D-GA2]"   "Yea"
[6,] "Rep. Marsha Blackburn [R-TN7]" "Yea"
```

□

6.4 Fold operations on lists

Any operation that requires repeated application of a function or that consolidates values into a single structure can be implemented as a *fold* process. One such case is discussed in Section 4.2.1 where multiple regular expressions are applied in succession to a line of text. *Fold* operations can also iteratively construct a data structure column by column.

6.4.1 Abstraction of function composition

Function composition is a higher order operation since it takes both functions as input and returns a function. As a refresher, given two functions f and g, the operator is defined $(fg)(x) \equiv f(g(x))$ for x in the domain of g. Mathematically, composition is interesting since certain properties are preserved through the operation. From a programming perspective, function composition can seem a bit mundane as it is simply calling functions in succession. What makes composition interesting is when the sequence of functions is determined dynamically. Instead of hard-coding a chain of function applications, suppose you wanted that chain determined based on the context of data. A simple example is data normalization based on the type of data being normalized. In finance, conceptually identical data can come from many sources. To compare series together involves normalizing the data into a standard format. While separate functions can be written to handle each input source, it is often cleaner to use a configuration. This way the data processing pipeline is independent of the domain-specific business rules. Let's first pretend that we're naive and don't know about *fold*. Given some dataset `ds`, what happens if we try to use the *map* strategy to apply multiple functions to the same dataset?

```
fs ← list(f, g, h)
lapply(fs, function(f) f(ds))
```

What's wrong with this approach? Most obvious is that the object is not properly scoped. Also, the output of one function is not passed to the next function. This can be resolved manually using the global assignment operator, but this isn't particularly safe. Better to use *fold*.

```
fold (fs, function(f,o) f(o), ds)
```

We know from Chapter 4 that the typical use of *fold* is applying function composition incrementally over some input. The same principle applies to list inputs.

6.4.2 Merging data

Complex data structures are not always shrink wrapped and delivered to you. Sometimes these structures need to be constructed based on some parsing process. This is what happens with the ebola data, where tables are parsed iteratively and joined together. Merging cross-sectional tables typically involves the use of `cbind`. However, if the data are not normalized, there is no guarantee that the merge process will be successful. In this case `merge` is better. When there are more than two tables to merge, *fold* can mediate the process.

Example 6.4.1. In the `ebola.sitrep` package, multiple tables are merged together using *fold*. The call to *fold* takes an argument `fn` that parses each table and merges it with the accumulated data frame. First a `handler` is obtained that can parse a table into a data frame. Then the data frame is merged with the accumulated data frame.

```
config ← get_config(file.name)
init ← data.frame(county=markers, date=config$date)
fn ← function(i,acc) {
  start ← config[[sprintf('start.%s',i)]]
  label ← config[[sprintf('label.%s',i)]]
  handler ← 'get_rows'
  if (! is.null(config[[sprintf('handler.%s',i)]]))
    handler ← config[[sprintf('handler.%s',i)]]

  o ← do.call(handler, list(lines, start,'PAGE',config$markers
      ,label,regex))
  merge(acc,o, by='county', all=TRUE)
}
out ← fold(1:config$sections, fn, init, simplify=FALSE)
```

□

6.5 Function application with do.call

Function application is often dictated by language syntax and is specified directly in the code. In certain cases it can be useful to call a function dynamically. Rather than explictly specifying a function in code, It's possible to dynamically construct the function name and apply the function to an argument list. With dynamic application, a mechanism is needed so that a function can be applied to an arbitrary list of parameters. More formally, we need a transformation where $f(g, x)$ becomes $g(x_1, x_2, \cdots x_n)$ for x_i in x. In many languages the `eval` keyword is used for this purpose. In R, the function `do.call` is used, where x is a list. This function takes a character function name or a function reference along with a list of arguments. The application of `do.call(fn, list(a,b)` is equivalent to `fn(a,b)`.

Calling a function based on a particular context is one of the primary uses of `do.call`. Example 6.4.1 where a `handler` is called to parse a table. A default handler of `get_rows` is set, which is replaced with a specific handler if it exists. Lists can act as packed function arguments and operate in two paradigms. The first is unmarshalling the ellipsis into a packed argument list. The second is calling a specific function using a packed argument list. In cases like the ebola situation report parsing, it is not known when writing the parser how many tables will be parsed. By collecting these tables in a list, a single code path works for any arbitrary set of parsed tables. This pattern works whenever argument lists are unknown up until run-time.

Example 6.5.1. The function `parse_file` in the `ebola.sitrep` package dynamically calls a country-specific parser based on a file name. This function creates an associative array that stores a mapping from country code to a regular expression. The regular expression matches the file name used for the situation reports in Liberia and Sierra Leone. Based on which type is matched, the parse function is constructed and executed via `do.call`.

```
parse_file ← function(file.name) {
  types ← c(lr='^SITRep',sl='^Ebola-Situation-Report')
  type ← names(types)[sapply(types, function(x) grepl(x,
      file.name))]
  if (! type %in% c('lr','sl')) stop("Incorrect type")

  flog.info("[%s] Process %s", type, file.name)
  cmd ← sprintf('parse_%s',type)
  do.call(cmd, list(file.name))
}
```

The packed arguments must be passed as a list. Since the file name is a character vector, we simply wrap it in a list. The equivalent call is `cmd(file.name)`.

□

Example 6.5.2. The simplest example of using lists to pack an argument list is dynamically calling the concatenation function. Earlier we saw how lists preserve cardinality and therefore the ordinals as well. As discussed earlier this allows us to detect errors in processing. However, we may still want to continue processing despite the error. An example of this is when there is a parse failure due to a missing section. We can easily get rid of any nulls by executing `do.call(c,x)`. Note that this call will also transform the list into a vector, so care must be taken to ensure that types are appropriate.

□

Example 6.5.3. In Example 6.2.6 we explicitly used the indices to specify the two elements of each list. Since the argument is a list and the cardinality is fixed, we can also use `do.call` to apply the distance measure.

```
> apply(pairs[,1:5], 2, function(x) {
+   fn ← function(a,b) jaccard(a$friends, b$friends)
+   do.call(fn, graph[x])
+ })
[1] 0.12500000 0.00000000 0.00000000 0.09090909 0.09090909
```

□

In certain cases `do.call` acts as a short cut of *fold*. Consider `cbind` used to join columns together into a matrix or data frame. Suppose the columns are dynamically created by some function f. When the number of columns are known in advance, the de facto approach is to explicitly specify each and create the matrix via `cbind(a, b, c, d)`. What if the number of columns are dependent on some condition and are not known a priori? Collecting a dynamic set of columns necessarily requires a list container. In an iterative process a data frame is constructed via successive calls to `cbind`.

```
x ← data.frame()
for (col in cols) x ← cbind(x, f(col))
```

Alternatively *fold* can be used to accomplish the same task.

```
x ← fold(cols,
  function(col,df) cbind(df,f(col)), data.frame())
```

Even simpler is to take advantage of `do.call` to pass the transformed columns as arguments to `cbind`.

```
do.call(cbind, lapply(cols, function(col,df) cbind(df,f(col)))
```

Example 6.5.4. Not all data can be efficiently represented in a tabular structure. A common approach is to use a hierarchical data structure, such as a tree. Similar to how a CSV represents a tabular data structure and is converted to a data frame, serialized formats like JSON and XML represent arbitrary tree structures. While trees can be efficient for storing and looking up certain types

of data, they can be cumbersome for data analysis. One approach to working with JSON data is to extract specific elements. The result is still a hierarchical structure, although it is limited to required information. To produce a more convenient data structure typically involves flattening the structure into a flat list.

```
flatten ← function(x) {
  if (length(x) < 2) return(x)
  do.call(c, lapply(x, flatten))
}
```

As before, the same operation can be implemented with *fold*.

```
flatten ← function(x) {
  fn ← function(a,acc) {
    if (length(x) < 2) return(c(acc,x))
    c(acc, flatten(a))
  }
  fold(x, fn, c())
}
```

The value of `flatten` will be illustrated in the following section.

□

6.6 Emulating trees and graphs with lists

Programming languages all have a core set of data structures native to the language. This set of data structures is dependent on the language. With R being a language by and for statisticians, this core set differs from other languages. One obvious difference is the lack of a public hash or map structure. [4] Another is the inclusion of an `NA` primitive. For data structures that R lacks, lists can be used to emulate these structures. We are careful to emphasize *emulate*, as behaviors can be copied, but there is no guarantee that the algorithmic complexity is the same. This is particularly true when a list is used to emulate a hash table.

As an initial example, consider the tuple. Many functional languages provide a tuple type. What differentiates a tuple from a list is that tuples have fixed length, like arrays but contain heterogeneous elements. R doesn't provide native tuples so instead a list can be used to emulate its behavior. Similarly in many function languages, a list is implemented as a linked list. While the behavior appears the same as an R list, the performance is governed by the underlying data structure. Trees are another data structure not native

[4] The environment is a native hash strucutre but is not intended for use as a general purpose map.

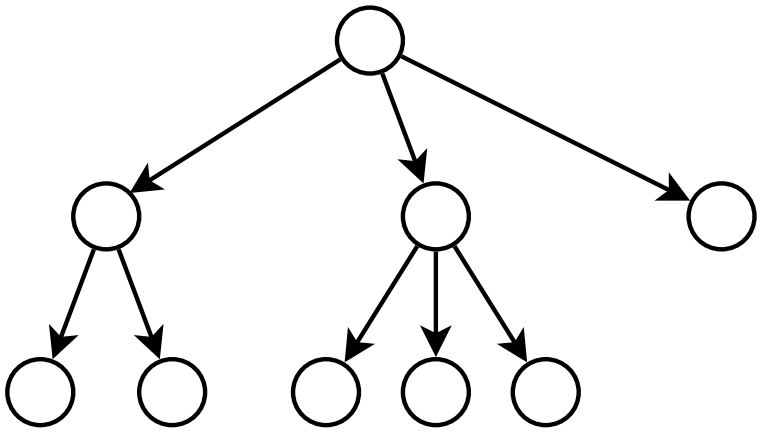


FIGURE 6.5: A cartoon tree

to R but easily emulated using lists. Trees are similar to hierarchical data structures like JSON. What sets them apart is that their structure is more constrained. A tree is basically a directed acyclic graph where each node has a value and a collection of child nodes. In data analysis trees can be used for text analysis [5] as well as for modeling probabilities. For example, a tree can represent the time evolution of a discrete process, like the binomial option pricing model, discussed in Section 6.6.2.

The simplest tree can be modeled as a list that contains two elements: a value and a list of children. Our implementation is a bit more general and supports an arbitrary number of node values. It will become clear in subsequent sections why we design it this way. Notice that we set the class to "Tree" since that is the name of the function. Some readers may notice that what we've created is a type constructor.

```
Tree ← function(..., children=NULL) {
  o ← list(...)
  if (! 'depth' %in% names(o)) o$depth ← 1
  o$children ← children
  class(o) ← c("tree", class(o))
  o
}
```

On its own our new type is not that useful. We need to add some functions that know how to manipulate the tree. Two common functions are to apply a transformation to each node in the tree or to only the terminal nodes. Tree operations are typically recursive in nature since each subtree is a valid tree. This is the case with our implementation, which first applies the given

[5] and their variant tries

function to the current node and then calls itself for each child tree. This second operation is a *map* process and is implemented with `lapply`. Doing so guarantees that the operation works correctly for any number of child nodes.

```
tree_apply ← function(tree, fn) {
  tree$value ← fn(tree)
  if (! is.null(tree$children))
    tree$children ← lapply(tree$children,
      function(x) tree_apply(x,fn))
  tree
}
```

Given the recursive nature of the function, what is happening to the tree? Each child node gets assigned a value based on the current tree state. By manipulating the values in child nodes, it's possible to create conditional dependencies between layers in the tree.

Another common operation is to apply a function only on the terminal nodes, or leaves. This function assumes that a tree has already been constructed and applies `fn` to each leaf.

```
terminal_apply ← function(tree, fn) {
  if (is.null(tree$children)) {
    return(fn(tree))
  }
  lapply(tree$children, function(x) terminal_apply(x,fn))
}
```

To get the value of the terminal nodes requires a function that knows how to distinguish between a regular node and a leaf.

```
node_value ← function(tree, id, depth=0) {
  if (is.null(tree$children)) return(tree[[id]])
  lapply(tree$children, function(x) node_value(x,id,depth+1))
}
```

The output of this function is another tree. Preserving structure is a useful property, but it can be difficult to digest the result. This is where the `flatten` function becomes useful, since it can transform a tree of terminal values into a vector.

6.6.1 Modeling random forests with trees

Random forests is a statistical learning technique that can be modeled using trees. The basis of a decision tree is not surprisingly a tree, so it is rather obvious how to implement it. The trick is related to constructing the tree based on the input data. The input is a vector of values. Each node in the tree contains an index and a predicate that operates on the element at the given index. We can use our earlier tree data structure to hold each decision tree in the random forests. The learning algorithm is shown in Algorithm 6.6.1. Our goal is not to replicate random forests verbatim, so we'll make a few

simplifications. Nonetheless, our naive 30 line implementation achieves 80% accuracy on the `fgl` dataset, included with the `MASS` library.

Algorithm 6.6.1: RANDOMFORESTS(X, Y)

Random forests algorithm

The random forests algorithm requires a new tree operation, where the result of a function is added as child nodes to terminal nodes. Notice that the function must return two sub-trees, one to represent each half of the split. As with most tree operations, this function is recursive. Notice that the recursion takes place for each child node, which is why the recursive call is mediated by `lapply`. The function passed to `terminal_split` must return a list of trees that represents the children. While our tree is binary, there is no such limitation in the function, which means other sorts of tree structures can be created using the same function.

```
terminal_split ← function(tree, fn) {
  if (is.null(tree$children)) {
    tree$children ← fn(tree)
    return(tree)
  }
  tree$children ← lapply(tree$children,
    function(x) terminal_split(x, fn))
  tree
}
```

Before going into the details of tree creation and training, let's take in the view. At a high level, the `train_rf` function has two operations. First it creates a forest by calling `make_tree` *ntree* times. Trees are "trained" at creation, and most of the details reside in `make_tree`. Second, the trees cast a vote on each input element to produce a single classification. Remember that this is in-sample, so the performance is optimistic.

```
train_rf ← function(x, y,
ntree=500, mtry=sqrt(ncol(x)), depth=5) {
  forest ← lapply(1:ntree,
    function(i) make_tree(x,y, mtry,depth)
  out ← sapply(1:nrow(x),
    function(i) forest_vote(x[i,], forest))
  attr(forest, "out") ← out
  forest
}
```

To understand how the tree is created, it's easier to first look at how trees vote on a particular element. The `forest_vote` function uses lapply to iterate over each tree and calculates the winner. When it comes to democracy, random forests is rather primitive and follows a winner-takes-all scheme. The class with the most tree votes is the output of the forest.

```
forest_vote ← function(x, forest) {
  votes ← lapply(forest,
    function(tree) tree_vote(x, tree))
  winner(votes)
}

winner ← function(x) {
  if (length(x) < 1) return(NA)
  o ← names(which.max(table(as.character(x))))
  if (length(o) < 1 || is.na(o)) browser()
  o
}
```

Now let's look at how an individual tree votes. During the tree creation process, each node in a tree (excepting the root) is assigned a predicate. This function determines which child is returned. Hence, the function recursively descends down the tree until there are no more children. At this point, it returns the value of the winner element for the given node. The winner is computed the same as before except that it is happening on subsets of the data. As the tree is created, the data are partitioned into successively smaller sets based on the predicate. Each node therefore has a winner assigned, which can be useful in understanding how the behavior of the tree changes as the depth increases.

```
tree_vote ← function(x, tree) {
  if (is.null(tree$children)) return(tree$winner)
  if (tree$children[[1]]$predicate(x))
    node ← tree$children[[1]]
  else
    node ← tree$children[[2]]
  tree_vote(x, node)
}
```

With a general understanding of how the tree structure is used, it's time to consider the big kahuna, `make_tree`, which is given in full in Listing 6.1. This function is itself composed of multiple functions, each responsible for a specific operation. The first function selects a random pivot point for a variable in the input. This pivot is used in the predicate to decide which branch a vector should follow. The function understands both continuous and categorical data and is responsible for choosing a valid value from the variable's domain. A set of variables and their pivot points are passed to the second function, `get_split`, which selects which of the variable/pivot pairs to use. Those familiar with random forests will notice that the number of variable/pivot pairs used is governed by the mtry argument. Hence, this function is responsible for choosing the best variable/pivot pair, and it does this by partitioning the data based on the pivot and calculating the information gain of each partition. The gain function is simply taking the difference of the entropy of the original set and the weighted sum of the entropies of each partition. In other words,

when two partitions have a lower overall entropy vis a vis the combined set, it is said to have positive information gain.

```
gain ← function(x, xs) {
  entropy(x) - sum(sapply(xs,
    function(xi) length(xi)/length(x) * entropy(xi)))
}
```

Information gain is dependent on entropy. Multiple measures of entropy are valid. We use the Shannon entropy, which is considered standard for random forests. This form of entropy is simply the negative sum of the probability mass function (PMF) times the log of the PMF.

```
entropy ← function(x) {
  pmf ← table(as.character(x)) / length(x)
  -sum(pmf * log(pmf))
}
```

The final closure brings everything together. This function takes a node and copies the data into its local context. The effect is that each successive node gets a smaller subset of the data based on the partitioning up to that point. With this subset, it evaluates *mtry* variable/pivot pairs and chooses the actual split pair. This pair is used in the predicate, which is called when the tree votes on a particular vector. The output of this function is a list comprising the two child nodes. Each tree includes the winning class for the node as well as the two partitions based on the selected variable split.

The final operation applies all the functions to a tree root via fold. Why is fold used here? Consider the output of each iteration. The `terminal_split` function will add two children to each terminal node. This new tree is then passed to the next fold iteration, so the result is that each later of the tree is grown for each fold iteration. To construct an actual forest, we'll use the `fgl` dataset, which has 214 samples and 9 variables. The dependent variable is `type`.

```
set.seed(12358)
forest ← train_rf()
```

Using our implementation, we can test the in-sample performance by comparing the winning votes to the actual classes. This results in 80.4% accuracy. Compare this with the `randomForest` package, which also produces 80.4% accuracy (though out-of-bag).

6.6.2 Modeling the binomial asset pricing model using trees

Another problem that can be easily modeled as a tree stems from finance. Valuation of financial instruments is a broad field in quantitative finance. The binomial asset pricing model is a staple for developing an intuition surrounding stochastic approaches for options pricing [**?**]. The basic idea is to model the value of an option as the present value of the option at

```
make_tree ← function(x,y, mtry, depth) {
  nvar ← ncol(x)
  ids ← sample.int(length(y), replace=TRUE)
  x ← x[ids,]
  y ← y[ids]

  get_threshold ← function(x, idx) {
    if (class(x[,idx]) == "factor")
      sample(levels(x[,idx]),1)
    else
      runif(1) * diff(range(x[,idx])) + range(x[,idx])[1]
  }

  get_split ← function(x,y, idx, threshold) {
    if (length(y) < 1) return(NULL)
    do_split ← function(i,t) {
      b ← x[,i] <= t
      list(y[b], y[!b])
    }
    best ← which.max(sapply(1:length(idx),
      function(i) gain(y, do_split(idx[i],threshold[i]))))
    if (length(best) < 1) browser()
    list(idx=idx[best], threshold=threshold[best])
  }

  add_node ← function(node) {
    x ← node$x; node$x ← NULL
    y ← node$y; node$y ← NULL
    idx ← sample.int(nvar,mtry)
    threshold ← sapply(idx, function(i) get_threshold(x,i))

    split ← get_split(x,y, idx, threshold)
    if (is.null(split)) return(NULL)

    predicate ← function(v) v[,split$idx] <= split$threshold
    b ← predicate(x)

    list(Tree(x=x[b,], y=y[b],
           winner=winner(y[b]), index=split$idx, predicate=
               predicate),
         Tree(x=x[!b,],y=y[!b],
           winner=winner(y[!b]),index=split$idx, predicate=
               function(x) !predicate(x)))
  }

  tree ← Tree(x=x,y=y)
  fold(1:depth, function(i,acc) {
    #flog.info("Constructing level %s of tree",i)
    terminal_split(acc, add_node)
  }, tree)
}
```

Listing 6.1: The `make_tree` function for a random forests implementation

exercise. At each time step, all possible prices of the underlier is computed, which produces a number of possible paths. As the name suggests, the binary options pricing model only has two outcomes, up or down, with a probability attached to each one. A tree in this model is simpler than in random forest and only contains the initial price of the underlier and an accounting variable for the depth, which corresponds to the time step.

```
make_binomial_tree ← function(x,s,u,d=1/u, depth=4) {
  add_node ← function(tree) {
    list(Tree(depth=tree$depth+1, s0=s, mult=tree$mult * u),
         Tree(depth=tree$depth+1, s0=s, mult=tree$mult * d))
  }

  root ← Tree(x=x,mult=1)
  fold(1:depth, function(i,acc) terminal_split(acc, add_node),
       root)
}
```

For simplicity we will only deal with put options, which gives the option holder the right to sell an equity at a given strike price. The option only has value if the current price is below the strike price; otherwise it's worthless. This calculation is known as the payoff of the option and is simply $max(X - S_t, 0)$, where X is the strike price and S_t is the price of the underlier. In our model, S_t is a terminal node and is computed using the multiplier and the initial price of the underlier.

```
payoff_put ← function(node,x) max(x - node$s0*node$mult, 0)
```

To get all of the payoffs at time t is just a call to `terminal_apply`.

```
payoffs_put ← function(tree) {
  terminal_apply(tree, function(node) payoff_put(node,tree$x))
}
```

Based on the set of payoffs, how do we calculate the current value of the option? The technique is called backwards induction and amounts to solving for the price of the option at the previous time step. Many approaches exist for this, and a standard methodology is due to Cox, Ross, and Rubinstein [**?**]. In this method, the givens include the strike price of the option, the initial price of the underlier, the standard deviation of the underlier, and the risk-free rate. A closed form solution exists for the percentage gain u and its probability p by defining the loss as $d = 1/u$.

$$p = \frac{e^{r\Delta t} - d}{u - d}$$
$$u = e^{\sigma\sqrt{\Delta t}}$$
$$d = e^{-\sigma\sqrt{\Delta t}}$$

These relationships are used to both create the tree of possible paths and also calculate the present value of the option based on a discount function.

Since the discounting works from the terminal nodes back to the root, a new tree function needs to be implemented. This new function needs to start at the terminal nodes, apply a function and recurse backwards up the tree. Let's call this function `reverse_apply`. It is almost the same as `tree_apply`, except that the application of the function occurs after the recursion. This means that a function is applied to a node after all its children have been applied.

```
price_put ← function(x, s0, sd, rf, dt, steps) {
  u ← exp(sd * sqrt(dt))
  d ← 1/u
  p ← (exp(rf * sqrt(dt)) - d) / (u - d)
  tree ← make_binomial_tree(x, s0, u,d, steps)

  vfn ← function(sn, vu, vd) max(x-sn, exp(-rf*dt) * (p*vu +
      (1-p)*vd))
  discount(tree, vfn, payoff_put)
}

reverse_apply ← function(tree, fn) {
  if (is.null(tree$children)) return(tree)
  tree$children ← lapply(tree$children, function(x) reverse_
      apply(x,fn))
  fn(tree)
}
```

The discount function is responsible for using `reverse_apply` in conjunction with a valuation function. First the payoff is computed for the given time step and then the value of the option is computed based on the current time step t and the two nodes at the subsequent time step $t + 1$.

```
discount ← function(tree, vfn, payoff) {
  fn ← function(node) {
    if (is.null(node$children[[1]]$price)) {
      node$children[[1]]$price ← payoff(node$children[[1]],
          tree$x)
      node$children[[2]]$price ← payoff(node$children[[2]],
          tree$x)
    }
    node$price ← vfn(node$s0*node$mult,
      node$children[[1]]$price, node$children[[2]]$price)
    node
  }
  out ← reverse_apply(tree, fn)
  out$price
}
```

Putting it all together, we can price an option provided a small set of initial values. Assuming that the underlying asset can increase by 20% in each year, we set `u ←1.2` and the timestep as `dt ←1` in annual terms. We also assume a risk-free rate of 5% and expiry in two years. For a strike price of 52 and a current price of 50, the price of the option is computed as

```
> sd ← log(u) / sqrt(dt)
> price_put(52, 50, sd, 0.05, dt, 2)
[1] 4.423332.
```

6.7 Configuration management

Let's return to the `ebola.sitrep` data and discuss how to apply these principles to configuration management. While configuration management is an essential aspect of software development, it is often overlooked in data analysis. There is good reason for this. An analysis can often be ad hoc and thrown away after it's done. In these cases, the time spent creating a structure for configuration management is not justified. As an analysis transitions into an operational process for reporting or decision support, configuration becomes more important. One reason is that the same analysis may be applied to multiple scenarios, each with their own specific configuration. Alternatively, input data may vary across data sources or time, thus requiring an easy way to specify differences in format. The ebola situation reports behave in this manner and benefit from abstracting the parse configuration into a separate function. This approach keeps all transformation logic together and isolated from analysis logic, making it easier to change both.

It's not necessary to use configuration *files* that are separate from code. Lists are plenty sufficient for managing configuration data. They also have the advantage of being quick to construct and integrate. For the ebola data, each list element represents a distinct set of rules to parse a set of situation reports. As a reminder, the format of the situation report changes over time. It would be inefficient to specify a configuration for each report. But we're getting ahead of ourselves. Recall that we're parsing a plain text representation of a PDF document. The parsing workhorse is the function `parse_lr` for Liberia. A similar function `parse_sl` exists for Sierra Leone. We saw in Example 6.5.1 how they are called conditionally based on the name of the file. A single configuration tells the parser how many and which tables to parse. It also provides information on when a table starts, what rows to expect, and which columns to extract. An initial parse configuration has the following structure:

```
config.a ← list(min.version=175, max.version=196,
  sections=3, markers=markers,
  start.1=start.case.a,  label.1=label.case.a,
  start.2=start.hcw,     label.2=label.hcw,
  start.3=start.contact, label.3=label.contact)
```

The first two elements define the range of report version (one per day) for which it's valid. This first configuration applies to version 175 through version 196. Next, the `sections` variable states that there are three tables of interest.

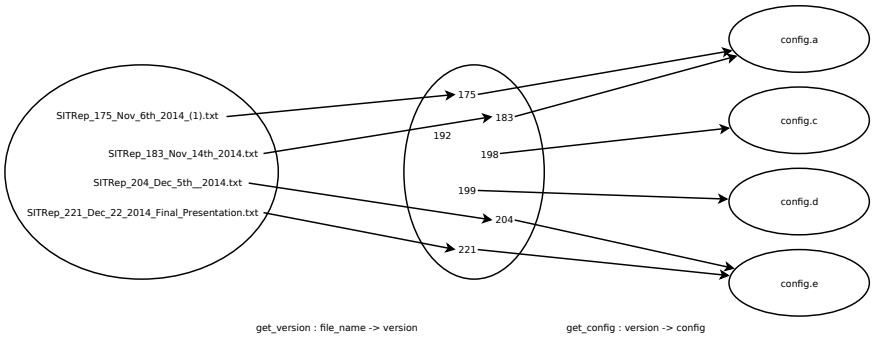


FIGURE 6.6: A transformation chain mapping file names to configurations

The parser uses this information to look for list elements named `start.`n and `label.`n, where $n \in [1, \textit{sections}]$. The start element is a unique regular expression used to indicate the beginning of the given table. The first value, `start.case.a` is the string `'Ebola Case and Death Summary by County'`. The labels represent the columns of interest in the given table. For example, `label.1` is assigned the vector

```
label.case.a ← c(
  'alive.total', 'alive.suspect', 'alive.probable',
  'dead.total', 'cum.total', 'cum.suspect', 'cum.probable',
  'cum.confirm', 'cum.death').
```

The final element in `config.a` is `markers`, which are the row markers for the tables. These happen to be the county/province names since that's how the tables are structured. Hence, for Liberia, they include

```
markers ← c('Bomi', 'Bong', 'Gbarpolu', 'Grand Bassa',
  'Grand Cape Mount', 'Grand Gedeh', 'Grand Kru', 'Lofa',
  'Margibi', 'Maryland', 'Montserrado', 'Nimba',
  'River Gee','River Cess','Sinoe','NATIONAL'),
```

while for Sierra Leone they are

```
markers ← c('Kailahun', 'Kenema', 'Kono', 'Kambia',
  'Koinadugu', 'Bombali', 'Tonkolili', 'Port Loko',
  'Pujehun', 'Bo', 'Moyamba', 'Bonthe',
  'Western area urban', 'Western area rural', 'National').
```

Notice that each vector contains a row for the national totals. This is an artifact of the tables, which we leveraged in the validation process.

This single configuration covers reports in the range [175, 196]. For other report versions, we need additional configurations. The Liberia parser contains five such configurations, and they are collected into a single list

```
config ← list(
  config.a, config.b, config.c, config.d, config.e).
```

How is a configuration chosen for a given file? We can think of this process as a function that maps file names into a configuration space. To do this, a file name is first mapped into version space, which is then mapped to the configuration space as depicted in Figure 6.6. This sequence of transformations is the algorithm and is realized by

```
parts ← strsplit(file.name, '_')[[1]]
version ← as.numeric(parts[2])

truth ← sapply(config, function(cfg)
  version >= cfg$min.version && version <= cfg$max.version)
config[truth][[1]].
```

The first two lines represent the `get_version` function in the figure, while the remaining three lines represent `get_config`. We use the predicate approach discussed in Chapter 5 to extract the correct configuration based on version number.

An explicit configuration system thus introduces a structured approach for managing variations in data and computation. The tradeoff is that the code is more abstract but is much easier to maintain. What if no configuration strategy is taken? The result is a teetering mass of procedural code with conditional blocks specifying how to handle one case or another. This logic can be spread across multiple files and functions, making it difficult to debug and update.

6.8 Exercises

Exercise 6.1. Can `combn` be modeled as a *map* operation? Why or why not?

Exercise 6.2. For sets that can be treated as metric spaces, a preliminary transformation can be applied to convert the set into a metric space. Then a metric can be applied, which is useful if other analyses will be conducted in the metric space. This is the approach we took in Example 6.2.5, where each stanza was converted to a term frequency vector first. By separating the two steps, the same encoding can be used with a different similarity measure. We discussed two possible measures in that example. What if we want to calculate these measures for each possible pair? Implement this procedure as a *map* operation by expanding all possible pairs in the set and iterating over the pairs.

Exercise 6.3. Create an HtmlTree data structure based on Tree. Add an extra element called "attributes" that contain HTML attributes. Create supporting functions that can access attributes for a given element specified by /x/y/b@att

Exercise 6.4. do.call

Exercise 6.5. Implement an ordering where the two lists of Example xx are equal

Exercise 6.6. Compute the similarity of each pair of stanzas using the `combn` function. Which stanzas are most similar? Which stanzas are least similar?

7

Data frames

PENDING

8

State-based Systems

PENDING

9

Alternate functional paradigms

PENDING

Bibliography

[1] ECMAScript Language Specification (Standard ECMA-262). Technical report.

[2] Alfred V Aho and Jeffrey D Ullman. *Foundations of computer science*, volume 2. Computer Science Press New York, 1992.

[3] Hendrik Pieter Barendregt. *The lambda calculus: Its syntax and semantics*, volume 103. North Holland, 1985.

[4] World Wide Web Consortium. Technical report, 2010.

[5] Andrew Eisenberg, Jim Melton, Krishna Kulkarni, Jan-Eike Michels, and Fred Zemke. Sql:2003 has been published. *SIGMOD Rec.*, 33(1):119–126, March 2004.

[6] B Kernighan and P. J. Plauger. *Software Tools*, page 319, 1976.

[7] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[8] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. See ?Reduce.

[9] Guido Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[10] Brian Lee Yung Rowe. *lambda.tools*. Zato Novo, LLC, 2012.