Modeling Data With Functional Programming In R
Preface

This book is about mathematical reasoning. The goal is to illustrate how to effectively apply it to not only quantitative analysis but also the resulting computer program. The emphasis is on the use of functional programming as the conceptual force that binds the quantitative model with the application model. Most books that cover numerical and/or quantitative methods focus primarily on the mathematics of the model. Once this model is established the computational algorithms are presented without fanfare in an imperative style. While explicitly detailing the steps it is often difficult to reason about the algorithm in a way that can meaningfully leverage the properties of the mathematical model. This is a shame because mathematical models are often quite beautiful and elegant yet when transformed into software become ugly and cumbersome. This unfortunate outcome is exacerbated by the real world, which is messy: data does not always behave as desired; data sources change; computational power is not always as great as we wish; reporting and validation workflows complicate model implementations. Most theoretical books ignore the practical aspects of working in the field. Yet the need for a theoretical and structured bridge from the computational sciences to programming has grown as more practitioners voice their concerns, particularly in the bioinformatics arena [].

Effectively navigating these situations in the context of model development requires a structured approach to quantitative analysis and development. My approach is to re-establish the intimate relationship between mathematics and computer science with the goal of producing computer programs that clearly convey the underlying mathematical model and retain the referential transparency necessary for reasoning about the program. This is possible in R via my package lambda.r that implements a number of advanced features of functional programming languages. Functional programming has a long history within academia and its popularity in industry has recently been trending up. Two prominent reasons for this upswing that are the growing computational needs commensurate with large data sets and the data-centric computational model that is consistent with the analytical model. The superior modularity of functional programs isolates data management from model development, which keeps the model clean and reduces development overhead. Derived from a formal mathematical language, functional programs can be reasoned about with a level of clarity and certainty not possible in other programming paradigms. Hence the same level of rigor applied to the quantitative model can be applied to the software model.
Lambda.r was borne from my personal frustration of seeing the inefficiencies of building quantitative systems using object oriented design principles. The key observation I made was seeing how simple the mathematical models were and the comparative complexity of the corresponding software model. Activities like retrieving and normalizing end of day data sets, using a Taylor series approximation for options prices, calculating historical VaR, and portfolio optimization can all be described mathematically with a handful of equations. Yet converting these mathematical concepts into working software became a Herculean task that was riddled with logical errors difficult to uncover. Worse, the code itself was hard to reason about so the only certainty around the behavior of the system was dictated by our test coverage. This seemed crazy to me given the certainty of the underlying mathematics.

Around the same time I was fortunate enough to be exposed to R. Here was a language that breathed math. With built-in vector semantics, formulas, and of course rock solid statistical models, there was a lot to like. R also has a lineage rooted in Scheme as well as the obvious S heritage, which meant that many important functional programming concepts are built into the language. Examples include first class functions, idioms that leverage higher-order functions, lexical scoping, and lazy evaluation. Yet as I began implementing models and applications in R I discovered that here, too, the conventions for application development were footed in object oriented methods. This wasn’t surprising since R contained not one, but two function dispatching systems to emulate object-oriented programming. The larger systems like Bioconductor and RMetrics are built atop the S4 object-oriented system that provides a class model and static type safety at the expense of a rather complex and counterintuitive interface.

It seemed a shame that a language with a strong functional heritage was lacking the semantics to write large-scale applications in a functional style. While it is certainly possible to write in a purely functional style in R, without more advanced features it is just as cumbersome as developing an application in S4. Some of the features I was looking for included a type system not tied to S4, optional static type safety, multipart function definitions, and pattern matching. Having experience in Erlang and some Haskell, I borrowed the elements that I found useful to create lambda.r. A few years on, lambda.r is now a powerful yet simple functional alternative to S4 and the other object-oriented approaches.

This book begins by establishing the principles of functional programming. The common thread between computer science and quantitative methods has always been mathematics, so I first establish a mathematical formalism for lambda.r, which provides the foundation for discussing and reasoning about functional programs. First is a discussion around the mechanics of functional programming and how to reason about function dispatching. I also prove numerous equivalent forms so that functions can be transformed while main-
taining identical behavior. Afterward is a thorough discussion of the data structures in R and how to reason about them. The fact that vectors are a native type yields interesting and sometimes surprising behaviors that affect program design. I illustrate how to prove various functional programs via symbolic transformation and standard proof techniques. Coupled with this theory is a more practical discussion surrounding the mechanics of functional programming and the mathematical properties provided by lambda.r. Following this treatment of the language and syntax is a discussion on type theory and design. Proper use of types can greatly simplify managing multiple models and also dealing with regime change.

In the second part of the book practical elements of model development are introduced, where I discuss effectively managing data with functional techniques and leveraging the modularity of functional programs to expedite model development. Armed with this structured approach, time spent on tedious plumbing can be minimized. At this point, I also hint at some of the workflows that quants face, such as model research and prototyping, back testing and model validation, visualization and reporting. Understanding these workflows and the data requirements for each is key to designing an effective system architecture. First I present the architecture of computational systems. This is where the principles outlined in the first part are applied. I start by introducing a theory of system architecture that looks at the interaction of data with software in the same way that traditional architects look at the interaction of people with space. This analogy is useful in visualizing the flow of data through a system and how it evolves depending on the workflow and the audience. It is important to distinguish between interfaces for data and interfaces for humans and how this impacts the computational system. Viewing data from this perspective also sheds light on what is necessary for a computation and what is simply baggage that bloats memory and slows down processing. From this perspective, I discuss key workflows relevant to a data scientist, quant, or computational engineer: simulation, visualization, reporting, model validation. Once this framework is established, I take an in-depth look at data management and how to apply the principles discussed in the book to simplify this plumbing. With data out of the way it is then time to look at models and how to maintain modularity. This is essential to being able to swap out models without requiring a complete rewrite of the system. Finally, it is time to dig into computational complexity and how this impacts the system design. Certain problems may technically be intractable, although depending on the purpose, an approximate solution may be acceptable. Hence, by understanding the nature of a problem it is possible to save a lot of agony in the attempt to solve it. If the problem is tractable, then there must be a way to improve the efficiency of the algorithm to meet the processing requirements for the system. Trading applications likely have the greatest demands on computational efficiency, while simulations need to be efficient enough to complete in a reasonable amount of time. A common theme is how the superior modularity of functional programming enables the quant
to easily tune and replace models as conditions and constraints change over time. An equally important benefit is how functional programs with limited side effects are easier to parallelize, which means that not only is the model pluggable but so is the wiring. Packages like foreach and snow can be drop-in implementations that leverage parallelism behind the scenes if done correctly. Similar strategies can be applied for GPU-based computations. When done incorrectly, these enhancements can act as shackles that prevent alternative analytics as the logical flow is stuck in the one path. Proper application design and functional programming techniques simplifies the process of adding these features without disrupting the modeling process.

An interdisciplinary book of this nature runs the risk of alienating all interested parties. To even pick up the book requires more than a passing knowledge and interest of both quantitative fields and computer science. Many introductory concepts are glossed over in order to maintain focus on the core discussion. Knowledge of set theory, abstract algebra, mathematical analysis, basic statistical and numerical methods, machine learning, and programming concepts are all assumed, though I do provide copious references to literature as well as refresher notes in the appendix. When it makes sense, I spend extra time establishing core concepts. For example I expect most quants to know the basics of object-oriented programming but little if anything about its dual, functional programming. Coming from the other side, software engineers transitioning into quantitative development should be comfortable with basic statistics and linear algebra.
Contents

1 Introduction 1
  1.1 Symbolic transformation and notation ...................... 5
  1.2 The lambda calculus .................................. 6
    1.2.1 Reducible expressions .............................. 7
  1.3 Fundamental higher-order functions ....................... 9
    1.3.1 Iteration ........................................ 9
    1.3.2 Conditional Blocks ............................... 10
    1.3.3 Free variables and closures ...................... 12
  1.4 Determinism and referential transparency ................ 15
    1.4.1 Lazy evaluation and promises .................... 17
  1.5 A note on examples .................................... 17
    1.5.1 Financial markets ................................ 18
    1.5.2 Text analysis and NLP ............................ 19
    1.5.3 Marketing analytics ............................... 19
    1.5.4 Civic issues and policy ........................... 20

I The Mathematics of Computation 21

2 Mechanics of Functional Programming 23
  2.1 Function definition and dispatching ..................... 26
    2.1.1 Mechanics of sequence mutation .................... 29
    2.1.2 Optional arguments and default values ............. 32
    2.1.3 The ellipsis argument ............................ 34
    2.1.4 The range operator ............................... 36
    2.1.5 Pattern matching and guard expressions ........... 40
    2.1.6 Parameter compatibility and dispatching .......... 43
  2.2 Types and type constraints ............................. 45
    2.2.1 Type constructors ................................ 47
    2.2.2 Type hierarchies and dispatching ................. 49
    2.2.3 Attributes and orthogonal data structures ......... 51
    2.2.4 Type constraints ................................ 53
    2.2.5 Type variables .................................. 54
    2.2.6 Type constraints and the ellipsis argument ....... 56
    2.2.7 Example: Indicators and signals ................. 57
  2.3 Lexical scope, frames, and state ......................... 57
    2.3.1 Environments and lexical scope ................... 57
2.3.2 Assignment to enclosing scopes .......................... 59
2.4 Discussion: An asset trading system .......................... 60

3 Formal Considerations of Data Structures 63
3.1 Set and sequence mechanics ................................. 63
  3.1.1 Vectors as sequences .................................. 64
  3.1.2 Lists as sets ........................................... 68
  3.1.3 Set operations .......................................... 70
3.2 Mechanics of vectorization ................................. 70
  3.2.1 Recycling of vector arguments ........................... 72
  3.2.2 The \texttt{map} function ................................ 73
  3.2.3 The \texttt{fold} function ................................ 76
  3.2.4 Monte Carlo as a proxy for induction .................... 80
  3.2.5 Transitive aspects of vectorization ..................... 81
3.3 Matrices ....................................................... 83
  3.3.1 Subsets and indexing ................................... 85
  3.3.2 Example: Numerical integration .......................... 89
3.4 Sequences, series, and closures ............................. 91
  3.4.1 Error analysis and the Euler-MacLaurin formula ........... 94
  3.4.2 Example: Taylor series approximation .................... 94

4 Type Theory and Design 99
4.1 Contextualization via types .................................. 101
  4.1.1 Bounded sets ............................................ 101
  4.1.2 Encapsulation of model methodology ..................... 101
  4.1.3 Finite state machines and types ......................... 101
4.2 Revisiting set theory ........................................ 101
  4.2.1 Constructors ............................................ 101
  4.2.2 Types and data structures ................................ 101
  4.2.3 The $ operator ......................................... 102
  4.2.4 The \@ operator .......................................... 102
  4.2.5 Hierarchies and compatibility ............................ 102
4.3 Type constraints .............................................. 102
  4.3.1 Referential transparency and typed functions ............ 102
  4.3.2 Type variables .......................................... 102
  4.3.3 The range operators and types ........................... 102
4.4 Example: Binomial asset pricing model ..................... 102

II Applications of Functional Programming 103

5 Datacentric Architectures 105
  5.0.1 Pipelines and data flows .................................. 106
5.1 Data flows and transformation graphs ....................... 108
  5.1.1 Data-software interface ................................ 108
  5.1.2 Managing multiple time frames ........................... 108
  5.1.3 Example: Daily versus annualized risk ................... 108
5.1.4 Process terraces and precision ................................ 108
5.1.5 Example: Yield curve construction .......................... 108
5.1.6 Simulation ...................................................... 108
5.1.7 Example: Signal generation back testing .................... 108
5.2 Actors and process divergence ................................. 108
  5.2.1 The Human-Computer Interface .......................... 109
  5.2.2 Explanation versus prediction ............................. 109
  5.2.3 Model selection ............................................. 109
  5.2.4 Error analysis ............................................... 109
5.3 Visualization and reporting ...................................... 109
  5.3.1 Static visualization ........................................ 109
  5.3.2 Example: R packages gglplot2, vcd, maps ................. 109
  5.3.3 Interactive visualization .................................. 109
  5.3.4 Example: JavaScript and Rook for interactive web visu-
                 alization ............................................. 109
5.4 Model validation ............................................... 109
  5.4.1 Model risk .................................................. 110
  5.4.2 Internal consistency ....................................... 110
  5.4.3 Sensitivity analysis ....................................... 110
  5.4.4 Simulation and stress tests ............................... 110
  5.4.5 Example: VaR .............................................. 110

6 Data Management .................................................... 113
6.1 Normalization .................................................... 116
  6.1.1 Isolating data management logic .......................... 118
  6.1.2 Data sources and I/O abstraction ........................ 119
  6.1.3 Data formats and versions ................................ 121
  6.1.4 Example: Portfolio construction from a set of signals . 122
  6.1.5 Bad data ..................................................... 122
  6.1.6 Example: Corporate actions ............................... 122
6.2 Model artifacts ................................................... 123
  6.2.1 Model adjustments and new data requirements .......... 123
  6.2.2 Back testing and validation artifacts ..................... 123
  6.2.3 Example: Signal generation and portfolio optimization 124
  6.2.4 Visualization and reporting artifacts ..................... 124
  6.2.5 Example: Saving charts with png ........................ 125
6.3 Data artifacts ..................................................... 125
  6.3.1 Iterative development and data formats .................. 126
  6.3.2 Preventing data dependencies .............................. 126
  6.3.3 Duck typing as a workaround .............................. 126
  6.3.4 Guards as a temporary code stash ......................... 127
6.4 Persistence ........................................................ 127
  6.4.1 Databases .................................................. 128
  6.4.2 Files ........................................................ 128
7 Modeling and Modularity

7.1 Polymorphic models and type hierarchies
7.1.1 Switching models
7.1.2 Parametric models
7.1.3 Stochastic models
7.1.4 Example: Covariance matrix estimation
7.1.5 Example: Optimization and the simplex method

7.2 Closures as model glue
7.2.1 API consistency
7.2.2 Avoiding interface leakage
7.2.3 Example: Unconstrained optimization

7.3 Data source selection
7.3.1 Effects on precision
7.3.2 Effects on performance
7.3.3 Example: Macroeconomic indicators

7.4 Compound models and process flow
7.4.1 Dynamic regime change
7.4.2 Multiple time scales
7.4.3 Example: Trending markets to mean reversion

8 Computational Complexity

8.1 Sources of complexity
8.1.1 NP-completeness
8.1.2 Combinatorial optimization
8.1.3 Example: Modeling matrices and linear algebra

8.2 Measurement and instrumentation
8.2.1 Controlling variables
8.2.2 Performance bottlenecks
8.2.3 Timing
8.2.4 Profiling
8.2.5 Example: Portfolio optimization

8.3 Algorithm optimization
8.3.1 Algorithm complexity
8.3.2 Data compression
8.3.3 Side-effects and shared memory
8.3.4 Example: Multi time frame VaR

8.4 Multicore and multithreading
8.4.1 Native interfaces

Bibliography
## List of Figures

1.1 Fundamental higher-order functions ........................................ 9  
1.2 Windowed versions of `map` and `fold` ................................. 10  
1.3 Functions with declarative control flow ............................... 11  
1.4 Optional evaluation via pattern matching ............................ 12  
1.5 Introducing an external dependency to `maprange` ............... 15  

2.1 Two implementations of Geometric Brownian Motion ............... 24  
2.2 Moving averages and a time series ...................................... 25  
2.3 Myriad ways to define a function ...................................... 27  
2.4 Famous multipart functions .............................................. 28  
2.5 The `samplerange` function ............................................. 34  
2.6 A second clause to the `samplerange` function ..................... 38  
2.7 The behavior of `confine` ............................................... 42  
2.8 Quantizing a time series with 2 and 3 bins .......................... 46  
2.9 The Signal type constructor ............................................. 48  
2.10 Venn diagram of a type hierarchy as subsets ...................... 49  
2.11 Operations on matrices .................................................. 50  
2.12 Generating `Signals` .................................................... 60  
2.13 Constructing a `Portfolio` .............................................. 61  
2.14 Evaluating performance of a `Portfolio` ............................. 61  

3.1 Definition of the factorial function .................................. 80  
3.2 Using Monte Carlo to 'prove' the induction step .................. 81  
3.3 Distribution of trials in Monte Carlo simulation .................. 82  
3.4 Map for 2 dimensions .................................................... 86  
3.5 Fold for 2 dimensions .................................................... 88  
3.6 Approximation of cos(x) about x = 2 ............................... 96  

4.1 Using `apply` as a map between two sets ........................... 100
# List of Tables

1.1 Comparison of syntax for selected operations .......................... 3
2.1 Hierarchy of environments ................................................. 58
3.1 Mapping of logical operators .............................................. 68
3.2 Common functions and their archetypes ................................. 71
Mathematics is generally perceived through the lens of symbolic notation. We think of formulas, variables, and esoteric symbols incomprehensible to mere mortals. Those of us who practice these mystical arts appreciate the power of this notation and why these symbols are necessary to communicate ideas and prove theorems. In the history of mathematics, symbolic notation is a relatively recent invention [9]. Prior to this breakthrough, mathematics had more of an imperative flavor to it. Knuth tells us that the ancient Babylonians had formulas for solving specific classes of equations, which were described in algorithm form or listed out explicitly in tables. Hence a solution to a problem reads like a story. The relatively advanced Babylonian society was already well versed in the ways of finance and described many financial concepts. One documented exercise was to find how long it takes to double an investment given a compounding rate. [14]

One kur (of grain) has been invested; after how many years will the interest be equal to the initial capital?

You should proceed as follows.

Compound the interest for four years.
The combined total (capital+ interest) exceeds 2 kur.
What can the excess of this total over the capital plus interest for three years be multiplied by in order to give the four-year total minus 2?
2,33,20 (months).
From four years, subtract 2,33,20 (months), to obtain the desired number of full years and days.

Knuth’s primary interest is exploring their understanding of algorithmic logic. It is clear that the Babylonians had developed sophisticated algebraic methods, including their famed base 60 number system. Despite their impressive achievements, it is difficult to ignore the clumsiness of this "notation" and how it clouds the underlying relationships between the variables. This is the double edge of algorithms: their explicit nature comes at the cost of hiding meaning. Symbolic notation on the other hand is efficient. Syntactically symbols are far more compact than words. Symbols also bring their own semantics that can be extended across branches of mathematics. In a sense symbolic semantics can be considered homomorphic. Once we understand the
rules governing the manipulation of symbols, insights can quickly be discovered\(^1\).

While mathematics is typically associated with declarative logic and symbolic notation, software is typically associated with imperative logic and algorithms that detail explicit steps. The history of computer science is marked by the progression of algorithms, such as sorting elements in a list. These algorithms were developed in the context of von Neumann machines, themselves being a physical embodiment of Universal Turing Machines [23]. The significance of this is that the behavior of these machines are defined imperatively via explicit expressions and statements that govern the control flow and state of the registers of the underlying CPU. Consequently the evolution of thought was grounded in this particular paradigm, leading to advances in data structures and how to move data from one place to another. Backus made a similar observation, pointing to word-at-a-time von Neumann architectures as the root cause, leading to "unnecessary confusion in the way we think about programs" [5]. Object oriented programming introduces a more structured way of organizing functions and data structures but does nothing to change the fundamental imperative constructs. With object-oriented programming, we are still stuck with "the fundamental problems created by the word-at-a-time von Neumann style of programming, with its primitive use of loops, subscripts, and branching flow of control." [5]

Three decades after Backus’ seminal paper, computational science is struggling with these very issues. The problem is even more acute since these applications are mathematical in nature. Other problem domains begin in the real world such that classes and objects in software are a meaningful representation of physical entities. Analytical models begin with a mathematical model that has no direct analog to the physical world. Attempting to model these concepts using object-oriented programming or imperative syntax in general is akin to the inefficient imperative approach that Babylonians used to codify their equations: Imperative algorithms are problematic in many ways: they are verbose, too specific, and offer few mathematical properties. Imagine expressing vector or matrix notation with algorithms. These simple operations quickly become cumbersome: writing out matrix multiplication becomes tedious after a rank of just a few. Worse, the meaning of the operation is obfuscated by the mechanics of the operation. Consider the generation of random correlated variables after a Cholesky decomposition. Clearly the way to express this is via matrix notation as opposed to writing out explicit operations. In general we favor \( \bar{r}_c = \bar{r} U \) over \( \bar{r}_c = [r_1 U_{1,1} + ... + r_n U_{1,n}, r_1 b_{m,1} + ... + r_n U_{m,n}] \). This simple example also highlights just how explicit algorithms are. When explaining the mechanics of a computation or optimizing the performance of a computation this can be useful, but it is awful for representing a computation.

---

\(^1\)This is a pointed reference to Hamming’s famous quotation that "the purpose of computing is insight, not numbers"
**TABLE 1.1: Comparison of syntax for selected operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Functional (R)</th>
<th>Object-Oriented (Java)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \times B$</td>
<td>$a \times b$</td>
<td><code>a.multiply(b)</code></td>
</tr>
<tr>
<td>det$A$</td>
<td><code>det(a)</code></td>
<td><code>a.det()</code></td>
</tr>
<tr>
<td>$a^2$</td>
<td><code>a^2</code></td>
<td><code>Math.pow(2)</code></td>
</tr>
<tr>
<td>$\sum (x - \bar{x})^2$</td>
<td><code>sum((x - mean(x))^2)</code></td>
<td>double <code>sum = 0;</code>&lt;br&gt;double <code>mean = a.mean();</code>&lt;br&gt;for (int <code>i=0; i&lt;a.length; i++</code>)&lt;br&gt;  <code>sum += Math.pow(a.get(i) - mean, 2);</code></td>
</tr>
</tbody>
</table>

Instead of this von Neumann imperative style, what is necessary is an approach to programming that matches the declarative power of symbolic notation. It turns out that such a model of computation has existed since the 1930s and is known as the lambda calculus, invented by Alonso Church [citation]. The programming model based on the lambda calculus is known as functional programming. The lambda calculus defines lambda expressions and a finite set of reductions that can symbolically transform them. The reductions are known as $\alpha$-conversion, $\beta$-reduction, and $\eta$-conversion. Informally these perform variable substitution, function application, and abstraction simplification, respectively. Creating an abstraction uses the symbol $\lambda$, which is the process of defining functions, and application is applying functions to variables. The significance of the lambda calculus is that computations can be described in an applicative or declarative form retaining referential transparency and thus preserving the mathematical properties of the computation. In turn this means that programs can be reasoned about and proven to be correct, which can even be shown algebraically within the system. Compare this to conventional software testing that relies on an exhaustive set of permutations to establish confidence in an algorithm. While discussing the lambda calculus in depth is beyond the scope of this book, we use it as as a starting point, borrowing certain concepts to show how computational systems can be written in such a way that symbolic transformation is possible between mathematical models and their programming counterparts. Readers wishing to explore this field further are directed to [6].

Functional programming is thus a practical implementation of the lambda calculus. Thanks to its declarative nature and referential transparency, functional programs emphasize what an operation does as opposed to how an operation is implemented. Table 1.1 compares imperative and declarative forms for certain mathematical operations. What this shows us is that the benefits of symbolic notation extend from mathematics to programming languages. This results in programs that closely resemble the syntax of the original mathemat-
ical expression. It is a boon for fields heavy in math to leverage languages that represent computing operations in a similar manner as there is less translation error (what I call a conceptual impedance mismatch) between the model and the code. R was designed for statistical and mathematical analysis, so the idea of minimizing the impedance mismatch between model and code in some ways has been there from the beginning \cite{7}. Supporting many functional programming concepts, R also introduces vectors as a primitive data type. These key features compress programming notation to a level of succinctness that rivals standard mathematical notation. The efficiency of R extends beyond convenient syntax for vectors and matrices and is part of the idioms that drive the development of the language. For example, manipulating any list-like object can be done using set operations. This can be surprising to a newcomer since this is true even for write operations. Suppose you have a vector that contains asset returns and need to find all the six sigma events. Starting with a mathematical statement we use a set comprehension to define the set as \( \{ x : |x| > 6\sigma \} \). Expressing this in R uses a similar declarative syntax
\[
x[\text{abs}(x) > 6*\text{sd}(x)]
\]
while an imperative algorithm looks like
\[
a <- c()
for (xi in x) {
    if (\text{abs}(xi) > 6 * \text{sd}(x)) a <- c(a, xi)
}
\]
Modifying the value of these outliers is even more cumbersome when using standard imperative approaches. Most R users know to do
\[
x[\text{abs}(x) > 6*\text{sd}(x)] <- 6 * \text{sd}(x)
\]
as opposed to explicitly managing the vector index via
\[
for (i in 1:length(x)) {
    if (\text{abs}(x[i]) > 6 * \text{sd}(x)) x[i] <- 6 * \text{sd}(x)
}
\]
Besides being more verbose, these implementations are also less efficient, which requires even more code to improve their performance. Virtually every comparison of declarative and imperative algorithms highlights the efficiency of declarative notation alongside its canny resemblance to the original mathematical notation. When encapsulated within a function, these imperative statements are hidden from view so that code may seem declarative. Wrapping the above statements in functions illustrates that although the function call appears to be declarative, the underlying implementation is still an explicit algorithm that is difficult to reason about. Being able to reason about a program means that the behavior of the program can be deduced and logically proven. This allows you to prove unequivocally that an algorithm does what

\footnote{Originally its predecessor S}
you intend. As expected a logical proof is much stronger than relying on a finite set of examples to prove a theorem. This implies the limited value of unit testing since the set of tests will be finite. The more a program adheres to the rules of mathematics, then the more likely it is able to be proven. At a minimum functions must have referential transparency and be well-defined. Referential transparency means that an expression can be replaced with its value without changing the behavior of a program [citation]. In lambda calculus terms this is essentially a $\beta$-reduction over a lambda expression. While a complete application may not be referentially transparent, this chapter shows how local referential transparency is sufficient to prove certain critical behaviors of an application. Conversely the less functional the system, the harder it is to formally reason about it.

### 1.1 Symbolic transformation and notation

This idea that programming and math can express the same thing shouldn’t be surprising, but the divergence of computer science from pure math has obscured this relationship. This book is devoted to formalizing this intuition and showing how the notation of one discipline can be symbolically transformed into the other and how this can be used to speed the development of analytical models and prove the correctness of functional programs. Symbolic transformation is a form of direct proof that can explicitly show the equivalence of a mathematical equation and a programmatic function. Operations like replacing one variable for another is a simple example of such a transformation. Nobody would dispute that $f(x, y) = \sqrt{x^2 + y^2}$ is the same as $f(a, b) = \sqrt{a^2 + b^2}$. \(^3\) It is also possible to show the equivalence between two systems of notation. For example there is no difference in denoting a derivative as $f'(x)$ versus $\frac{df}{dx}$. When two systems of notation are equivalent, I use the $\equiv$ symbol to indicate exact equality in meaning between the two notations (typically mathematical and programming). Whenever exact equality appears the two syntactic forms are interchangeable, which will be used to show the equivalence of functional programs with their mathematical counterparts and vice versa. In the derivative example this is codified as $f'(x) \equiv \frac{df}{dx}$. The notion of compatibility is also important, which is denoted as $S \sim W$. This should not be confused with equivalence, which uses the same symbol. When the usage is ambiguous, the meaning will be made explicit.

The natural numbers 1, 2, ... are denoted $\mathbb{N}$, while the whole numbers 0, 1, 2, ... are denoted $\mathbb{W}$. A subsequence 1, 2, 3, ..., $n$ of $\mathbb{N}$ is denoted $\mathbb{N}_n$. Sets are denoted using standard $S = \{x, y, z\}$ notation. The elements in a set are considered unique unless otherwise specified. This implies that the result of

\(^3\)This is known as $\alpha$-conversion in the lambda calculus.
set operations (e.g. union) are similarly taken as unique. Tuples and sequences are treated as equivalent and are denoted \( X = (x_1, x_2, ..., x_n) \). Sequences are assumed countable such that \( X \sim \mathbb{N}, \) with cardinality \( |X| = n. \) The \( k \)-th element of \( X \) is \( x_k \in X \).

Functions are minimally described by their domain and range as \( f : A \rightarrow Z \). A function with multiple arguments is \( f : A \times B \times ... \rightarrow Z \). A function that has a sequence of length \( n \) as an argument is \( f : A^n \rightarrow Z \). From a computability perspective, functions can be recursive.

In terms of syntax, when analyzing a function standard mathematical type-setting is used, to ease the transformation between mathematics and programming. Code listings are displayed in a fixed-width font. Simple variables are denoted with lowercase characters. Notable exceptions are random variables and matrices, which are denoted using a capital letter. Capitalized characters are typically reserved for sets and types. When discussing the lambda calculus, capitalized characters also denote arbitrary lambda terms. Throughout the book I freely interchange R and lambda.r where both refer to the R language extended by the lambda.r package.

1.2 The lambda calculus

One of the key features of the lambda calculus is its ability to define fundamental mathematical concepts like numbers as lambda terms. While it is not necessary for us to define the whole of mathematics, what is necessary 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. In the untyped lambda calculus only variables \( v_1, v_2, \) etc., the symbols \( \lambda \) and \( . \), and parentheses are allowed. The set of all lambda expressions is further defined inductively [6].

**Definition 1.2.1.** The set of all lambda expressions \( \Lambda \) 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. So long as the mathematical constructions we create satisfy this definition, then we can leverage the lambda calculus in our analysis.
1.2.1 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 $\alpha$-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 $\lambda x.x \cdot 2 = \lambda y.y \cdot 2$. 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

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

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 $5! = 5 \cdot 4! = 20 \cdot 3! = 60 \cdot 2! = 120$. 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 $\beta$-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 $\eta$-conversion and is often characterized in terms of extentionality. I tend to think of $\eta$-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 $\eta$-conversion can often lead to greater modularity and simplification of an algorithm.
**Definition 1.2.2.** Given $\lambda x.Mx$ and $N \in \Lambda$, if $x$ is not free in $M$ then the $\eta$-conversion of the lambda abstraction is $(\lambda x.Mx)N = N$.

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.

**Theorem 1.2.3.** $\lambda w.X = \lambda v.(\lambda w.X)[w := v]$

*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 $\lambda w.X = \lambda v.(\lambda w.X)[w := v]$. This theorem can easily be extended to multiple variables, which forms the basis for creating closures. In general the lambda calculus easily handles multivariate functions. Additional arguments are appended after the $\lambda$ symbol as $\lambda xyz.X$. At times I will use the ellipsis $\cdots$ to indicate an arbitrary number of arguments. As a shorthand I will often denote a set of function arguments as a sequence. Hence for $W = (x, y, z), \lambda xyz.X = \lambda W.X$.

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 2.

Standard mathematical notation can be included as lambda expressions, since numbers are variables and operators are functions. Lambda terms can be data or a function. Any lambda expression can be applied to any other lambda expression [6]. In language terms this indifference to the context of the data (whether it is truly data or a function) leads to the concept of a first-class citizen. Obtaining first-class status means space on the heap is allocated for the entity and it can be referenced like any arbitrary variable. In languages like R where functions are first class, they can be used as a variable and passed as arguments to other functions. In the lambda calculus all functions are anonymous and first-class. Anonymous functions are therefore synonymous 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 $f(x) = f = \lambda x$. This equivalence can be extended to any function with an arbitrary number of arguments. For function application we note that $(\lambda x.M)[x := N] = f(N)$, where $f(x) = M$.

In terms of symbolic notation, equality of expressions is denoted by $\equiv$. 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.
1.3 Fundamental higher-order functions

Higher-order functions are a staple of functional programming. In the lambda calculus, higher-order functions provide the means to implement all programming language constructs. For example, in the untyped lambda calculus recursion is implemented using the concept of fixed point combinators. The notion of a fixed point is tied to functions. A fixed point is any value where the domain and range are the same. For scalar functions like $f(x) = x^2$, the values 0 and 1 are fixed points. This concept extends to higher-order functions in the sense that the fixed point is no longer a scalar but a function. An example is the function $f(x) = e^x$ is a fixed point of the derivative. Moving a step further, fixed point combinators are functions that can turn any function into a fixed point. This strange notion is actually the basis for defining recursive functions.

Imperative programming languages have key control flow constructs as part of the language that are combined with data structures to make programs. Functional programming is no different with the exception that theoretically there is no need for explicit control flow semantics. Instead higher-order functions are used for the same purpose. I’m reluctant to use the term emulate, which implies functionality that is missing. Instead what is important is thinking about programming as an extension of mathematical analysis. From this perspective what exactly are control structures? Ultimately they divide into those that iterate over elements in a data structure or execute a block of code based on some condition.

1.3.1 Iteration

Two core functions for iterating over sequences common to functional programming are listed in Figure 1.1. `map` applies a function to every element of
FIGURE 1.2: Windowed versions of map and fold

maprange(x, window, fn, do.pad=FALSE) %as% {
  y <- sapply(window:length(x), function(idx) fn(x[(idx-window+1):idx]))
  onlyif(do.pad, function(z) pad(z, window-1), y)
}

(a) The maprange function

foldrange(x, window, fn, acc=0) %as% {
  sapply(window:length(x), function(idx) fn(acc, x[(idx-window+1):idx]))
}

(b) The foldrange function

a sequence returning a sequence of the same length. \(^4\) fold is similar but is a binary operation that accumulates the results into a scalar. With the popularity of the so-called map-reduce paradigm, these higher-order functions have become more widely known. When working with data, these functions are commonly used to coerce and mold data into a usable state. A simple example of map and fold is computing the expected value of a random variable, which is \(E[X] = \sum P(x) \cdot x, \forall x \in X\). The equivalent implementation is \(\text{fold(' + ', map(\lambda x.P(x) x).)}\).

In addition to map and fold, I introduce additional higher-order functions necessary for modeling data. These include window-ed versions of map and fold, named maprange and foldrange, respectively, which are defined in Figure 1.2. The motivation for these functions is that it is equally useful to apply a function to vectors and not just scalars. Later in the book, two-dimensional versions of these functions will also be introduced.

Example 1.3.1. The Central Limit Theorem tells us that given a population of random variables, the distribution of the mean of the mean converges to the normal distribution. Assume that some vector of random values represents a population. Then a set of random variables drawn from the population can be constructed based on a sliding window over the vector.

```r
x <- rweibull(2000, 1)
y <- maprange(x, 50, mean)
z <- map(1:1000, function(a) mean(sample(y, 50)))
```

1.3.2 Conditional Blocks

The other primary group of control structures are conditional blocks. These are divided into if/else blocks or case statements that check for different val-

\(^4\) In standard R the suite of \(*apply\) functions are analogous to map for a variety of data structures.
FIGURE 1.3: Functions with declarative control flow

```r
pad(x, window, FALSE) %as% x
pad(x, window, TRUE) %as% c(rep(NA, window), x)
```

values of a variable. Theoretically speaking conditional expressions do not have any impact on the referential transparency of a function. Even so they tend to impede reasoning of code due to the increase in cyclomatic complexity [17]. This complexity often makes it difficult to mentally perform symbolic transformation since numerous variable states must be retained to know what value a function will yield. Typically multiple function clauses will reduce the need for nested conditional blocks. In other situations what is needed is a technique to transform a conditional expression into a multipart function. This sort of control flow can be handled by using simple pattern matching in function definitions. These are equivalent because ultimately this sort of logic reduces to checking whether a variable or expression has a specific value. In the if/else situation the value is simply TRUE or FALSE. One example is the pad function that takes a vector and pads it with values to extend its length. Typically some other sequence is generated from a sequence, which has a shorter length. It is often useful, for example with plotting data, to ensure that data are the same length. Hence the decision to extend a sequence is governed by some other function call. This can be accomplished by defining the pad function as in Figure 1.3. Rather than creating a control structure in a calling function, this multipart function accomplishes the same thing without creating the complexity associated with a control structure.

A more general approach is to use a higher-order function that controls the application of a function based on some condition as illustrated in Figure 1.4. This simplifies the logic further by removing a clause from the definition of pad. Now pad is only responsible for prepending vectors with a value, and onlyif is responsible for conditional function application. This approach is used in maprange to optionally pad data so that it is the same length as the input data. Whether data should be padded is clearly situation-specific and therefore cannot be automatically applied in the function. Hence this sort of optional augmentation can be captured in such a block.

In the lambda calculus the higher-order function IF serves a similar purpose. This function is defined as $\lambda p. \lambda a. \lambda b. pab$ returning either $a$ or $b$ depending on the value of the predicate function. This approach functionizes two distinct execution paths, which mirrors the behavior of a traditional if-else control structure. The advantage of this approach is that the use of function application ensures proper isolation between steps in an algorithm paving the way for symbolic reduction.
FIGURE 1.4: Optional evaluation via pattern matching

\begin{verbatim}
onlyif(TRUE, f, x) %as% f(x)
onlyif(FALSE, f, x) %as% x
\end{verbatim}

(a) Conditional application of a function

\begin{verbatim}
pad(x, window, default=NA) %as% c(rep(default, window), x)
\end{verbatim}

(b) Redefinition of pad

1.3.3 Free variables and closures

Lambda expressions can have unbound (or free) variables, meaning that a function references a variable that does not appear in the argument list of a binding lambda expression. For example \( \lambda x.x + y \) has \( x \) bound and \( y \) free. In conventional mathematics this would appear to be non-sensical since \( f(x) = x + y \) is not well-formed. Yet in other cases free variables are consistent with mathematical thinking. The most prominent example is differentiation and integration. For example the integral \( \int x^2 + y^2 \, dx \) has \( x \) bound and \( y \) free. In the lambda calculus and functional programming in general there are few restrictions on this usage, since the free variables can be bound to an argument in the enclosing functions scope. These are typically bound to an argument defined in an outer context or environment. When this happens the inner function is called a closure.

Definition 1.3.1. The free variables of an arbitrary lambda expression is defined recursively [6]. Let \( e_1, e_2, e_3 \in \Lambda \).

\[
FV(v) = \{v\} \\
FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \\
FV(\lambda x.e) = FV(e) \setminus \{x\}
\]

Example 1.3.2. In the simple example of \( \lambda x.x + y, M = x + y \) implies \( W_M = \{x, y\} \). The abstraction \( \lambda x.x + y \) has variables \( V = \{x\} \), so \( FV(M) = \{y\} \).

Definition 1.3.2. Given \( \lambda X.\lambda Y.e \), \( \lambda Y.e \) is a closure if \( FV(\lambda Y.e) \subseteq \{X\} \).

The definition of a closure tells us that closures are created by binding the free variables of a function to an enclosing function’s arguments. It also implies that the closure is the return value of the original function. Taking the previous example a step further, suppose we define another lambda abstraction \( N = \lambda y.\lambda x.x + y \). Then \( N2 = \lambda x.x + 2 \), which is a closure. In terms of R the implementation is rather similar, where the lambda notation is replaced by the keyword function.

\[
> n <- \text{function}(y) \text{ function}(x) x + y
\]
Introduction

> n(2)
function(x) x + y
<environment: 0x7fcb44d208e0>

Higher-order functions take a lambda abstraction as an argument. Typically the behavior of the lambda abstraction is transformed in some way. This transformation happens on a case by case basis and can usually be proven for an arbitrary function. This concept may seem alien, but there exist numerous higher-order functions in conventional mathematics. Examples include the summation and product operators as well as the derivative and integral.

Definition 1.3.3. A higher-order function \([22]\) is a function that either

(a) takes one or more functions as input

(b) outputs a function.

Example 1.3.3. The purpose of a higher-order function is to mediate the application of some arbitrary function. In object-oriented terms one could consider this the functional equivalent to inversion of control. All of the higher-order functions described in Section 1.3 exhibit this behavior where the application of the lambda abstraction is mediated. In map rather than applying the lambda abstraction to a list it is applied to each element in the list.

Theorem 1.3.4. This theorem is the principle of function-operator equivalence. 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 \(f_2 = f_1\).

Example 1.3.4. \(x + y\) vs `+`(x,y)

Any multiple argument function can be transformed into a chain of single argument functions. This transformation is known as Currying \([6]\) and the resultant lambda abstraction is referred to as the curried form of the original multi argument function.

Theorem 1.3.5. Given \(\lambda W.X\) where \(W\) is an argument list, then \(\lambda W.X = \lambda w_1.\lambda w_2.....\lambda w_n.X\).

Proof. Partial application tells us that given \(\lambda W.X\), \((\lambda W.X)[W := S]\) where \(|S| < |W| = \lambda W.X[W_0 := W_1] := S\). Suppose \(W_b = W_n\). Then \((\lambda W.X)[W_n := S_n] = \lambda w_1 w_2...n.X[w_n := s_n]\)

(INCOMPLETE)

Curried functions are evaluated via chained function application. The order of the arguments in the curried form are not restricted to the order of the original argument list. In fact any permutation of the arguments is equivalent.
Corollary 1.3.6. Given $\lambda W. X$ where $W = (w_1, w_2, ..., w_n)$ is an argument list. Let $W' = (w'_1, w'_2, ..., w'_n)$ be a random permutation of $W$. Then $\lambda W. X = \lambda w'_1. \lambda w'_2. .. \lambda w'_n. X \ \forall W'$ permutation of $W$.

Proof. Show via partial application that any argument can be evaluated in any order (INCOMPLETE)

Currying a function has two practical purposes. First it makes matching function signatures trivial. Since the function is an approximation of another function, you would expect that the two functions should be called the same. Second, by binding some variables within a closure, those variables are guaranteed fixed for repeated applications, meaning that the function is guaranteed to be well-defined. We’ll see many other useful examples of closures throughout the book.

Example 1.3.5. At the beginning of this chapter we defined the $\text{fgbm}$ function as a four argument function. Below is an equivalent version in its curried form.

$$
\text{fgbm.curry} \leftarrow \text{function}(n) \ \text{function}(s) \ \text{function}(\text{mean}) \ \text{function}(\text{sd}) \{ \\
\text{cumprod}(\text{c}(s, \exp((\text{mean} - \text{sd}^2/2) / 252 + \text{sd}\ast\text{rnorm}(n)/\text{sqrt}(252)))) \\
\}
$$

Executing the curried version is via chained function application, which yields the same result as the normal three argument version.

$$
\text{fgbm}(100, 10, 1, 2) == \text{fgbm.curry}(100)(10)(1)(2)
$$

Calling $\text{fgbm.curry}(100)$ yields a function that will produce another function. Each chained application of this function will yield a time series with 100 elements.

Chained function application leads to the idea that function application is an iterative process. It is natural to consider halting the process at a specific step. Doing so results in a partially applied function. This partially applied function can then be called later in a program such that the bound function arguments are effectively held constant. In the formal sense partial application occurs when a function is applied with only a subset of its argument list. The return value is a closure whose arguments are the free variables. In some languages partial application occurs automatically. [16] R does not support partial application, but its behavior can be emulated as appropriate.

Example 1.3.6. To make this idea more concrete let’s look at transforming the $\text{fgbm}$ function into a partially applied form.

$$
\text{fgbm.set} \leftarrow \text{function}(n, \text{mean}, \text{sd}) \ \text{function}(s) \ \text{cumprod}(\ldots)
$$

The generalized process is $\mathbb{R}^n \rightarrow \{\mathbb{R}^n, \mathbb{R}^n\} \rightarrow \{-1, 0, 1\}$ and $\mathbb{R}^n \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^n \rightarrow [-1, 1]$. The average of these two signals is then taken to generate the final signal. Calculating each signal is simply function composition, and it is easy to reason about which clauses are being executed based on the characteristics of the data.
1.4 Determinism and referential transparency

Functional programs are generally recognized to be provable. Although there are functions that may not have a normal form (they do not terminate), in general it is straightforward to prove the correctness of a functional algorithm. One of the reasons that these programs are provable is because the data on which they operate have limited scope and cannot affect data in another function. Compare this to object-oriented programming where methods within a class are expected to share and modify the same set of variables. This makes a program harder to reason about because the result of a function is dependent on exogenous factors. When a function changes things beyond its scope it is said to have side effects. Writing to a global variable is a common side effect. It is easy to illustrate how a function cannot be proven to behave a certain way when a function is subject to side effects.

Suppose that the `maprange` function relied on a global variable for controlling padding. Its implementation would look like Figure 1.5. The implementation is identical to the earlier definition with the only change being the removal of `do.pad` from the function signature. Instead this value is accessed in an environment external to the function definition. This means that the caller of `maprange` is no longer in control of the behavior of the function. In a single-threaded environment this may seem inconsequential, but when two or more functions are competing to update the value of `do.pad`, the preservation of the global setting is not guaranteed to be reliable. Unreliable code execution is synonymous with non-deterministic behavior.

As further evidence of the loss of determinism when relying on values external to a function, consider the control of plotting parameters. The convention is to retain the previous settings when updating the plotting parameters. The implementor of a function is responsible for reverting the settings to the previous values once a plot is drawn. However, if the function exits early due to a runtime error, then the configuration is in a modified state that is unknown to other callers of the plotting functions. Hence the state of the plotting parameters is non-deterministic, implying that the only way to know the value is by setting them explicitly. Eliminating the reliance on side-effects like these is thus a prerequisite to having functions with deterministic behavior.
Referential transparency goes hand-in-hand with determinism and is a core requirement of writing deterministic programs. The concept of referential transparency requires thinking about application of functions in time.

**Definition 1.4.1.** Let $t$ be a point in time and let $\phi$ be a function clause. Given parameter list $S$, the application of $\phi$ to $S$ at time $t$ is denoted $\phi^t(S)$.

Hence, for all time, a function call must yield the same result to be referentially transparent. This is the only way that the result of a function can replace the function call.

**Definition 1.4.2.** Let $t_1 \neq t_2$ be two distinct points in time. A function clause $\phi$ is referentially transparent if and only if $\phi^{t_1}(S) = \phi^{t_2}(S)$.

It is difficult to know whether a function is constant over all time. One way to reason about this is to consider the sources of non-determinism. Since R is single-threaded this exercise is relatively easy.

**Theorem 1.4.3.** The function $\lambda W.X$ has referential transparency if and only if $\text{FV}(\lambda W.X) = \emptyset$.

**Proof.** Let $\text{FV}(\lambda W.X) = \emptyset$. Then $\text{FV}(X) = \{W\}$.  \hfill $\Box$

**Theorem 1.4.4.** A function $\Phi$ is referentially transparent if and only if $\phi$ is referentially transparent, $\forall \phi \in \Phi$.

**Proof.** Let $\Phi = \langle \phi_1, \phi_2, \ldots, \phi_n \rangle$, where $\phi_k$ is referentially transparent $\forall \phi_k \in \Phi$. Suppose $\Phi$ is not referentially transparent. Then $\exists \phi_k \in \Phi$ that is not referentially transparent. But by definition there are no such $\phi_k$, so $\Phi$ must be referentially transparent.  \hfill $\Box$

**Theorem 1.4.5.** Let $f : A \rightarrow B$ and $g : B \rightarrow C$. If $f$ and $g$ are referentially transparent, then $g(f(a))$ is referentially transparent.

**Proof.** Let $c = g(b), b \in B$. Now let $b = f(a), a \in A$. Since $f$ is referentially transparent, $b$ can always be replaced by $f(a)$. So $c = g(f(a))$ is referentially transparent.  \hfill $\Box$

This proof illustrates the value of a functional programming approach. If a program can be reduced to a chain of function applications and each function is referentially transparent, then the program is referentially transparent. Even where it might be impractical (particularly when faced with dependencies) for the complete system to be designed this way, understanding this principle can help limit the loss of referential transparency via the design of the application. We will use this insight extensively in the second half of the book.

This theorem can also be extended to functions with multiple arguments.

**Proposition 1.4.6.** Let $\lambda W.X$ be referentially transparent. Then $\text{map}(x, \lambda W.X)$ is referentially transparent.
Introduction

Proof.
\[
\{x, y, fn, FV(fn)\} - \{x, y, fn\} = FV(fn) = \emptyset
\]

Example 1.4.1. As a counter example consider a function that relies on a date argument. If a date parameter is not provided, then a default value is used by calling `Sys.Date()`. This function is not referentially transparent since by definition the return value changes over time. Hence any function that relies on this function will not be referentially transparent.

The example of using `Sys.Date()` raises the important point that it is not necessarily wrong to use functions that are not referentially transparent. What is important is understanding the implications and knowing how to mitigate the undesirable properties (or lack thereof) when referential transparency is lost. On the other hand, functions that model random variables are useful precisely because they are not deterministic. And just like we are conscious and prepared for this randomness, the same should be true of any other function that is not deterministic.

1.4.1 Lazy evaluation and promises

Laziness as a computing concept centers around when an expression is evaluated. Lazy programming languages only evaluate expressions when their values are needed. This property has interesting effects on variable binding since values are not necessarily concrete until some unspecified time in the future. Until they are, the expression is held within a promise, or a function that knows how to evaluate the expression.

In practice we can ignore most implications of laziness, since mathematics can be considered lazy. This interpretation is based on the observation that expressions are not "evaluated" until someone explicitly calculates or writes down an equivalent formal reduction. Where this simplification breaks down is with formulas and parsed expressions. These details, however, are beyond the scope of this book. I mention it primarily for the sake of thoroughness and as a warning that there are practical limits to the theory developed in this book.

1.5 A note on examples

While this book introduces a fair amount of theory, the goal is to demonstrate how to apply this in practice. Too often there is a difficult gap to bridge be-
between theoretical foundations and practical applications. To ease the crossing of this chasm, this book is written with an example-first approach to ground the reader in the practical problems and how a theoretical approach simplifies discovering solutions. What follows are descriptions of the general problem domain in various fields. Each chapter focuses on a specific problem domain such that the end result of the chapter is a working proof of concept within that field. Through this process it is hoped that the reader can appreciate the power of mathematical analysis applied to the domain of computer science.

A common theme in these analyses is the feature extraction process and the commensurate modeling that follows. What we want to avoid is the trap of data mining that searches for specific patterns without any sort of underlying theoretical model backing the analysis. There are numerous pitfalls in this approach. Not only can this yield over-fitting that performs poorly in the wild, it also impedes reasoning about the characteristics and limits of the model. Identifying features is also suboptimal since without a model the search for features is essentially random. These same pitfalls exist in the software model and must be avoided.

1.5.1 Financial markets

Asset management has long been in the purview of quantitative methods. Whether for predicting market trends, pricing assets, estimating risk, or optimizing portfolios, there are innumerable applications of numerical methods in finance. For the purposes of this book I focus on a simple stock trading system that generates signals based on certain indicators. Typically various data ranging from direct market data to economic indicators to indicators of sentiment are used to generate trading signals. In the framework presented in this book, raw input data is used to produce indicators. Functions that produce these indicators are real-valued and designed to either oscillate or be bounded in a meaningful way. Multiple indicators are then combined to form a signal, which is bounded to the narrower range of \([-1, 1]\]. These signals are asset specific and when combined in aggregate dictate what assets will be held in the portfolio. A value of 1 indicates a long position while a value of -1 indicates a short position. A 0 means no position. Values in between can either be forced into these values or interpreted as a confidence measure such that a 1 means absolute confidence in a long position and so on.

The examples begin by simulating data that represent a time series of asset prices. We will then create some basic indicators and develop a simple trading strategy based on the indicators. From an initial universe of assets, we will generate a portfolio based on the result of the signals. Finally we will use a Monte Carlo simulation to estimate the risk, return, and evaluate the general performance of the trading strategy.
1.5.2 Text analysis and NLP

Online content generated by users on a daily basis tops terabytes. Quantitative methods are used to extract meaningful information from all this data. At the aggregate level, data is used to determine behavioral trends and to group people into similar profiles. Numerous techniques exist for not only predicting what content a user might like but also which content (and users) are similar. One way to do this is by using the content produced by a user as a proxy for the user. The goal then is to find documents that are similar to find users that are similar. Document clustering can also remove redundant information by grouping similar documents together. I map out a simple approach to clustering documents using well-known collocation theory. Collocations are phrases that have semantic meaning that is greater than the individual words that form the collocation. Proper nouns and place names fall into this category as do idiomatic expressions. The general idea is that similar documents will share collocations to a degree that cannot be explained by randomness alone.

The process of identifying collocations begins by cleaning textual data and then generating n-grams, or sequences of adjacent words. Collocations are typically limited to bigrams or trigrams, though there is no theoretical limit to their length. For our purposes we look strictly at bigrams, which are limited by the Justeson & Katz part-of-speech filter [12] to remove unlikely candidates and improve the overall signal-to-noise ratio.

1.5.3 Marketing analytics

Understanding the needs and wants of customers is a key component to building a successful business. All businesses are able to ask their customers directly with respect to their preferences and interests. However, an online business/presence has richer methods at its disposal, since hypotheses can be quickly tested and validated based on behavior alone without the need of direct interaction. This can be as simple as an A/B test to determine which visual design induces people to click a button or link. Going beyond usability are predictions of which products a consumer will purchase, such as movies, articles, restaurants, tweets, or more traditional consumer products. Going another step further, models of consumer behavior can predict which users will be the most loyal or which users are most likely to promote a product or brand.

Another area that gets a lot of attention is on identifying useful content. With so much user-generated content produced online it is difficult to extract meaningful content from retweets, shouts, flames, etc. While techniques like crowdsourcing can help to a certain extent, the ability to predict which pieces of content are valuable is obviously more compelling. Besides the obvious benefit of scale, algorithmic predictions can also be produced nearly instantaneously as opposed to the crowdsourced approach, which requires a non-trivial critical mass before results can be considered useful. As an example analysis we look at user reviews from Yelp to predict which reviews are most likely
to receive useful ratings. This feeds on the NLP analysis built in Section 3.2. In this context we want to group documents into clusters and then identify the useful features within each cluster. The idea is to then determine whether what is useful across document classes has some identifiable pattern, which is our starting hypothesis.

1.5.4 Civic issues and policy

Policy, whether educational, health, welfare, etc, is fast becoming an area that is not only seeing a multitude of data sets coming online but also a commensurate level of interest in exploring and analyzing this data. These data sets typically contain geographic and sometimes demographic information that can be examined for patterns. Some of the goals for analyzing this data is to understand patterns of distribution and utilization of resources, evaluate the effects of policy decisions, identify areas suffering from acute issues. More recently people are looking at how to create models that can simulate the effects of policies a priori. Data that affects policy will likely be on a longer time horizon and will utilize visualization in the end product more regularly.

This analytical workflow is explored in the specific case of the Open Data movement. The challenge with government data is largely a usability problem insomuch that data sets are typically raw exports from a data store and do not have standard data formats to join with other data sets. This is due in large part to the lack of standards around traditional data modeling. Hence each data set has an element of uniqueness to it. Common fields like zip code can have multiple formats and labels, such as ”ZipCode” or ”zip_code”, which then requires bespoke normalization of formats. Due to all these variations transformation logic can get rather involved just to make the data usable. As a case study I use the model used by my open index project, Odessa [3], which provides a standard format for common primary keys as well as tools to quickly and easily join disparate data sets together.
Part I

The Mathematics of Computation
This and the following chapter aim to provide a foundation for the remainder of the book. The marriage of mathematical analysis and computer programming is a novel idea that requires a firm footing. I begin by detailing the mechanics of lambda.r and the mathematical properties that certain programming language constructs possess. These definitions and rules form the basis for reasoning about programs written in lambda.r. On its own R provides numerous features related to functional programming yet is lacking in a few fundamental areas. Lambda.r is an extension to the R language that implements many functional programming concepts, most notably a framework for defining and dispatching functions using a declarative notation along with a well-defined and powerful type system. Both of these frameworks are consistent with functional programming principles. As R is descended from S and borrows concepts from Scheme [11], lambda.r similarly borrows ideas from other languages, most notably Erlang and Haskell. These ideas are integrated into the idioms of R with the goals of being easy to use and reason about, while also preserving the mathematical properties necessary to reason about programs.

While developing the ideas in this chapter, in parallel I will develop a framework for simulating asset prices and generating trading signals based on them. Equity asset prices are typically modeled using Geometric Brownian Motion, which is a stochastic process described by $dS = \mu S dt + \sigma S dZ$, where $dZ = \phi \sqrt{dt}$, $\phi \sim N(0,1)$. The first term of this equation is known as the drift while the second one is the stochastic term. The solution to this stochastic equation is $S(t) = S(0) \exp\left(\frac{\mu - \sigma^2}{2} t + \sigma (Z(t) - Z(0))\right)$, where $Z(t) - Z(0) \sim N(0, t)$ [10]. This solution is based on $S_0$ and is typically transformed to be constructed solely on the value of the previous time step, which is $S(t + 1) = S(t) \exp\left(\frac{\mu - \sigma^2}{2} t + \sigma Z(t)\right)$. Conveniently in this model the value of each time step is dependent solely on the value of the previous time step. This behavior is codified in the assumption of the Efficient Market Hypothesis, which assumes that all information is reflected in the most recent price of an asset. Two implementations are shown in Figure 2.1, followed by a sample time series generated by the \texttt{fgbm} function. The difference in the two implementations is the approach, whether explicitly recursive or embedded within a higher-order function. While the implementations may appear to be different, they are in fact equivalent. This point and its corresponding analysis will be explored further in this chapter and the following one.
FIGURE 2.1: Two implementations of Geometric Brownian Motion

```r
rgbm(n, s=10, mean=0.01, sd=0.03) %as% {
  rgbm(n-1, s, mean, sd, s)
}
rgbm(0, s, mean, sd, acc) %as% acc
rgbm(n, s, mean, sd, acc) %as% {
  s1 <- s * exp((mean - sd^2/2) / 252 + sd * rnorm(1) / sqrt(252))
  rgbm(n-1, s1, mean, sd, c(acc, s1))
}
```

(a) A recursive definition of GBM

```r
fgbm(n, s=10, mean=0.01, sd=0.03) %as% {
  cumprod(c(s, exp((mean - sd^2/2) / 252 + sd * rnorm(n) / sqrt(252))))
}
```

(b) An implementation of GBM using fold

The approach for the framework is to simulate a number of asset prices within a single market and then find the assets that have a positive trend. Once these assets have been identified, a portfolio is constructed, which can then be analyzed to determine the risk and return associated with the strategy. A basic way of determining trend is using a moving average (MA) crossover, which consists of two moving averages, one slow and one fast. The basic premise is that whichever direction the fast MA crosses the slow MA dictates the trend. Figure 2.2 shows a generated time series along with a 10-day and 20-day moving average. In this simple model the moving averages are indicators that are fed into a signal that produces one of three values: \{-1, 0, 1\}. Since we are interested in modeling, we will start with simulated data. Conceptually, the sequence of transformations is \textit{GBM} \rightarrow \textit{Indicator} \rightarrow \textit{Signal} \rightarrow \textit{Portfolio}. In one instance of this process data is transformed from \(\mathbb{R} \rightarrow \{\text{false, true}\} \rightarrow \{-1, 0, 1\} \rightarrow \{\text{assets, 0}\}\).

A second trading model also begins as a trend following strategy. Instead of using a moving average crossover as a proxy for the trend, the slope of the 10-day moving average is used to indicate the trend. I use the moving average to reduce the noise of the raw time series. The processing flow is similar to the crossover model except that only one indicator is used to generate the signal.
FIGURE 2.2: Moving averages and a time series

...
2.1 Function definition and dispatching

In mathematical analysis functions are introduced as a mapping from a set \( A \) into another set \( B \), which is denoted \( f : A \to B \) [20]. Beyond this not much is said regarding the mechanics of function application since in pure math functions are executed via mental computation. When this computation moves into a machine the rules must become explicit so that the behavior of the machine is consistent with our mental model. Outside of a machine it is understood by context that certain operations only apply to certain sets. It is also understood that depending on the input a different operation will result. As an example consider the multiplication of real numbers versus complex numbers versus matrices. The multiplication operator has the same semantic meaning for each of these sets, but the mechanics of the operation are different for each. Based on the type of the set, we naturally select the correct operation. To translate this into a computation it is necessary to define explicit rules on how to dispatch between the three definitions of multiplication given the operands.

The standard way to define functions in R is with the function key word. I reserve this notation for anonymous functions and transient closures. More permanent functions use the extended syntax of lambda.r, which defines named functions in a more traditional mathematical style. What follows are the formal rules for defining functions and how they are executed.¹ Programming language syntax is typically defined using extended Backus-Naur form. This formalism is very explicit and good for parser generators but difficult for humans to digest easily. Instead I continue to use mathematical notation to define functions and programming concepts. The benefits of this approach will become clear as we progress through these chapters.

Definition 2.1.1. Let \( W = (w_1, w_2, \ldots, w_n) \) be an ordered sequence of variables. When bound to a function name \( W \) is called an argument list. Hence \( W \) has the standard properties of a sequence:

(a) \( W \) is countable, i.e. \( W \sim \mathbb{N}_n \)

(b) \( |W| = n \)

Definition 2.1.2. A function signature is a tuple \( (f, W) \) consisting of the literal name of a function and an argument list.

Definition 2.1.3. A function clause \( \phi \) is defined by a function signature and a body. For signature \( (f, W) \) and block \( X \) the four forms of function definition in Figure 2.3 are equivalent.

¹
FIGURE 2.3: Myriad ways to define a function

<table>
<thead>
<tr>
<th>symbolic notation</th>
<th>permanent function</th>
<th>temporary function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(w_1, w_2, \ldots, w_n) = X$</td>
<td>$f = \lambda w_1, w_2, \ldots, w_n.X$</td>
<td>$f &lt;- \text{function}(w_1, w_2, \ldots, w_n) X$</td>
</tr>
</tbody>
</table>

The two constructions in the left column are for defining permanent named functions. The first version is standard mathematical notation, which is followed by the notation of lambda.r. The two expressions in the right column show notation for defining temporary first class functions. Of these the first is the notation of the lambda calculus while the latter is standard R syntax. Each format lists arguments explicitly. At times $f(w_1, w_2, \ldots, w_n)$ will be denoted $f(W), W = \langle w_1, w_2, \ldots, w_n \rangle$ as a shorthand. It will be clear from the context that $W$ is an argument list and not a block nor a lambda term.

A function clause $\phi$ of Definition 2.1.3 conforms to the standard mathematical definition of a function. From mathematical analysis, a function can be described by its domain and range. For example a function $f : A \rightarrow B$ has the function signature $(f, a), a \in A$. This notation generalizes to a multivariate function $g : D \times E \times \cdots \times Y \rightarrow Z$, translating to a function signature $(g, (d, e, \cdots, y)), d \in D, e \in E, y \in Y$. Function signatures reference arguments as variables within a set and should not be confused for being the set itself. In Section 2.2 we’ll see how the description of a function in terms of sets translates to type constraints.

**Example 2.1.1.** Starting with the GBM equation at the beginning of the chapter, let’s examine the components of this function. The function signature is $fgbm(n, s, mu, sigma)$, which returns the next value in the sequence. The argument list for this function is $W = \langle n, s, mu, sigma \rangle$ and the function clause is $\phi = \langle fgbm, W \rangle$.

The notation of mathematics is rich with constructions to express innumerable concepts. Functions do not always have simple definitions that can be described by a single expression. Such functions possess multiple parts, where each part corresponds to a distinct expression. Examples include famous functions like the Heaviside step function, the Kronecker delta, and even absolute value as shown in Figure 2.4. Evaluating a given expression in these functions is predicated on some condition being satisfied. From this perspective a function is simply a collection of function clauses with rules governing the selection of a function clause based on the parameters specified when calling the function. These function clauses are typically grouped together for conceptual clarity. Behind the scenes all function clauses sharing the same name are bound to the same function object. When calling a function with multiple clauses, the first matching clause will be executed. The concept of compatibility will be introduced later in the chapter, which provides a formal definition for the idea of a matching clause. One way of looking at a function is that it is
FIGURE 2.4: Famous multipart functions

\[ H[n] = \begin{cases} 0, & n < 0 \\ 1, & n \geq 0 \end{cases} \]
\[ \delta_{i,j} = \begin{cases} 0, & \text{if } i \neq j \\ 1, & \text{if } i = j \end{cases} \]
\[ |x| = \begin{cases} -x, & x < 0 \\ x, & x \geq 0 \end{cases} \]

a chained operation consisting of a mapping from a function space to a single function clause and then applying a set of parameters to the function clause.

**Definition 2.1.4.** A function \( f \) is uniquely identified by the sequence \( \Phi = (\phi_1, \phi_2, \cdots, \phi_n) \), where \( \phi_k \in \Phi \) is a function clause \( (f, W_k) \), \( W_k \) an argument list. As with an argument list, \( \Phi \) is countable, so \( \Phi \sim \mathbb{N}_n \) and \( |\Phi| = n \).

What does this definition imply regarding the domain and range of the function? To answer this we need to look at the process of function application, otherwise known as calling a function with a given set of parameters. When applying a function to a set of values, it is important to consider the process for which these values are bound to the argument list. This set of values is known as the parameter list. ²

**Definition 2.1.5.** A parameter list is a sequence of values \( S \) applied to a function \( f(W) \).

The act of function application is said to bind the parameter list to the argument list. For named functions the syntax for function application is what you would expect, namely \( f(S) \). Less common is the construction for an anonymous function. In the lambda calculus this is denoted \( (\lambda W.X)[W := S] \) and in R as \( (\text{function}(\bar{W}) X)(S) \). Functions are typically called based on the position of parameters. These parameters are then bound to the arguments having the same index. This syntax is the standard way of denoting function application in mathematics.

**Definition 2.1.6.** Let \( \lambda W.X \) be a function. Given parameter list \( S \) where \( |W| = |S| \) the application of the function \( \lambda W.X \) to \( S \), denoted as \( (\lambda W.X)[W := S] \), binds \( S \) to \( W \) such that \( w_k = s_k \).

**Example 2.1.2.** Suppose we want to generate a GBM time series with an arbitrary number of points \( n \). Calling this function as \( \text{fgbm}(100, 10, 0.02, 0.06) \) binds 100 to \( n \), 10 to \( s \), and so on.

Function application is not limited to positional parameters. In the name of convenience and flexibility R offers numerous variations on this process, which introduces some complexity around the basic concept of function application. The first extension is applying a function to a named parameter list. This

²In the R language reference the argument list is typically referred to as the formal arguments of a function. The values passed to a function are simply called arguments. To avoid confusion I refer to them as the argument list and the parameter list, respectively.
makes sense since function signatures name their arguments. Many languages specify calling functions based strictly on the position of arguments within the argument list. With named arguments it is possible to call a function with arbitrary argument position so long as arguments are assigned a name in the function call. The syntax for application of lambda abstractions supports assignment of named arguments using the bracket notation.

**Definition 2.1.7.** Let $U$ be a parameter list or an argument list. Then the name operator $N[U]$ yields the names bound to $U$. $N[U] = \{N[u_k] \forall u_k \in U,$ where $N[u_k] = \begin{cases} \text{literal name,} & \text{if a name exists} \\ \emptyset, & \text{otherwise} \end{cases}$.

In $\beta$-reduction parameters are bound to the argument list of a lambda abstraction. This process is generally name-aware in the sense that parameters are bound explicitly to arguments using the := operator. At times the symbol representing a variable must be distinguished from the variable itself. In order to keep this clear, I extend the definition of :=.

**Definition 2.1.8.** Given argument list $W$ let $\lambda W.X$ be a lambda abstraction and $S$ be a parameter list where $|S| \leq |W|$. Then $(\lambda W.X)[W := S] \Rightarrow W := (s_k|N[w_j] = N[s_k])$.

**Example 2.1.3.** The use of named parameters means that there are numerous permutations of a parameter list that all bind to the same argument list. $\text{fgbm}(10, \text{mu}=5, \text{sigma}=1, s=50) \Rightarrow (10, 50, 5, 1)$

Named parameters offer more than a syntactic convenience. This explicit binding process enables the use of a greedy argument (the ellipsis) for unmatched parameters. Named parameters also pave the way for default argument values, which are used when a parameter list omits certain values.

### 2.1.1 Mechanics of sequence mutation

In reasoning about the binding of parameters to arguments we need a concept that describes this binding process. Standard lambda calculus doesn’t discuss optional arguments or concepts like the ellipsis argument. Traditional set theory has operations for adding and removing elements from a set but no operations for replacing elements. To fill this gap, I introduce the set mutation operators $+_{o}$ and $-_{o}$ that provide a grammar for reasoning about these sorts of operations.

**Definition 2.1.9.** Let $x, y$ be scalars. **Additive mutation** is defined as

$$x +_{o} \emptyset = x$$

$$\emptyset +_{o} y = y$$

$$x +_{o} y = x$$
Definition 2.1.10. The empty set has special properties in this scheme, in-so-much that it acts as an indexing placeholder that can be extended infinitely.

\[ \emptyset = \{ \emptyset_k | k \in \mathbb{N} \} \]

\[ X +_o \emptyset = \langle x_1, x_2, \ldots, x_n, \emptyset, \emptyset \} \]

In Chapter 3, we’ll see that this definition is consistent with vectorized variables.

Definition 2.1.11. A overlay sequence \( X \) is a sequence where any indexed element can be \( \emptyset \). Since the \( \emptyset \) can be repeated an infinite number of times, the length of a overlay sequence is defined to be \( |X| = \text{argmax}_k x_k \neq \emptyset \).

As a shorthand, an indexed scalar is taken to be \( x_k = \langle x_k \rangle = \langle 1, 2, \ldots, k-1, k \rangle \).

Example 2.1.4. The following examples help to develop an intuition around overlay sequences. Let \( x_1, x_3 \neq \emptyset \).

\[ |\langle x_1, \emptyset, x_3 \rangle| = 3 \]

\[ |\langle x_1, \emptyset, x_3, \emptyset \rangle| = 3 \]

\[ \langle x_1, \emptyset, x_3 \rangle = \langle x_1, \emptyset, x_3, \emptyset \rangle \]

\[ |\emptyset| = 0 \]

\[ |\emptyset, \emptyset| = 0 \]

Binary operations on overlay sequences act in a pairwise manner.

Definition 2.1.12. Let \( X \) and \( Y \) be overlay sequences. \( X +_o Y = \langle x_k +_o y_k | x_k \in X, y_k \in Y \rangle \)

Theorem 2.1.13. Additive mutation has the following properties.

(a) Associativity: \( (X +_o Y) +_o Z = X +_o (Y +_o Z) \)

(b) Length: \( |X +_o Y| = \text{max}(|X|, |Y|) \)

Proof. Let \( X = \langle x_1, x_2, \ldots, x_n \rangle \), \( Y = \langle y_1, y_2, \ldots, y_n \rangle \), \( Z = \langle z_1, z_2, \ldots, z_n \rangle \).

(a) Since \( X +_o Y +_o Z \) is a pairwise operation we only need to show that \( x_k +_o y_k +_o z_k \) is associative. Let \( x, y, z \neq \emptyset \). Then

\[ (x + y) + z = x + z = x \]

\[ x + (y + z) = x + y = x \]
The above relationship holds regardless of the values of \( x \) and \( y \) so long as \( x \neq \emptyset \). Let \( x = \emptyset \). Then
\[
(\emptyset + y) + z = y + z = y
\]
\[
\emptyset + (y + z) = \emptyset + y = y
\]

Let \( x, y = \emptyset \). Then
\[
(\emptyset + \emptyset) + z = \emptyset + z = z
\]
\[
\emptyset + (\emptyset + z) = \emptyset + z = z
\]

We can see that overlay addition has left-precedence, meaning that the left most non-empty term will be the final value in a chain of operations.

(b) Let \( X, Y \) be overlay sequences. Consider \( k = \max(|X|, |Y|) \). Let \( x_k = \emptyset \), \( y_k \neq \emptyset \). Then \( x_k +_o y_k = y_k \Rightarrow |X +_o Y| = |Y| > |X| \). Now let \( x_k \neq \emptyset \), \( y_k = \emptyset \). Then \( x_k +_o y_k = x_k \Rightarrow |X +_o Y| = |X| > |Y| \). Finally let \( x_k \neq \emptyset \), \( y_k \neq \emptyset \). Then \( x_k +_o y_k = x_k \Rightarrow |X +_o Y| = |X| = |Y| \).

Example 2.1.5. Here are some simple examples illustrating how the overlay addition operator works in practice with two sequences.

(a)
\[
\langle x_1, \emptyset, x_3 \rangle
\]
\[
+\langle s_1, s_2, \emptyset, s_4 \rangle
\]
\[
\overline{\langle x_1, s_2, x_3, s_4 \rangle}
\]

(b)
\[
\langle w_1, w_2, w_3, \emptyset, \emptyset, w_6 \rangle
\]
\[
+\langle \emptyset, \emptyset, s_3, \emptyset, s_5 \rangle
\]
\[
\overline{\langle w_1, w_2, w_3, \emptyset, s_5, w_6 \rangle}
\]

Definition 2.1.14. Let \( x, y \) be scalars. Then overlay subtraction is defined as
\[
\emptyset -_o x = \emptyset
\]
\[
x -_o \emptyset = x
\]
\[
x -_o y = x
\]
\[
x -_o x = \emptyset
\]
Definition 2.1.15. Let $X$ and $Y$ be overlay sequences, where $X \cap Y \neq \emptyset$. Then $X -_o Y \equiv \langle x_k -_o y_k | x_k \in X, y_k \in Y \rangle$.

Theorem 2.1.16. Subtractive mutation has the following properties. Let $X$, $Y$, $Z$ be overlay sequences such that $X \cap Y \cap Z \neq \emptyset$.

(a) $|X -_o Y| = \begin{cases} |X|, & |X| > |Y| \\ \text{argmax}_k x_k -_o y_k \neq \emptyset, \text{ otherwise} \end{cases}$

(b) $(X +_o Y) -_o Y = X$

Proof. (a) (INCOMPLETE)

(b) (INCOMPLETE)

Example 2.1.6. (a)

\[
\frac{\langle w_1, w_2, \emptyset, w_4 \rangle}{\langle w_1, \emptyset, \emptyset, w_4 \rangle}
\]

(b)

\[
\frac{\langle w_1, w_2, w_3, w_4, w_5 \rangle}{\langle w_1, \emptyset, w_3, \emptyset, \emptyset \rangle}
\]

Exercise 2.1. Show that if $|X| \neq |Y| \neq |Z|$ that Theorem 2.1.13 (b) still holds.

Exercise 2.2. Show that overlay addition is not commutative.

Exercise 2.3. Let $X = \langle x_1, x_2, x_3, \emptyset, x_5 \rangle$, $Y = \langle \emptyset, \emptyset, y_3, y_4, y_5 \rangle$. Find $X +_o Y$.

Exercise 2.4. Show that $X = x_1 +_o x_2 +_o \ldots +_o x_n$ and $|X| = n$.

2.1.2 Optional arguments and default values

A common feature of R functions is to provide default values in function definitions. Essentially this amounts to functional polymorphism in a compact form. In languages that don’t support this feature it is necessary to define a second function that is called with a full parameter list, passing in the default values. Thinking about this explicit operation provides insight into how to model the behavior of optional values.

Definition 2.1.17. Let $W$ be an argument list and $X$ a lambda term. The argument $w_k \in W$ is an optional argument if $\exists v_k$ such that $\lambda W.X = \lambda W'.(\lambda W.X)[W := W' +_o v_k]$, where $W' = W -_o \langle w_k \rangle$. The value $v_k$ is called the default value of $w_k$. 
From the definition it is clear that the use of optional arguments is a concise way to emulate multiple function clause definitions.

**Example 2.1.7.** The definition of \( \text{fgbm} \) uses default values to supplement missing parameters. Let \( V \) be a overlay sequence of default values. Suppose \( \text{fgbm} \) is called as \( \text{fgbm}(100, \mu=2) \). Then

\[
W := S + V
\]
\[
:= (100, \emptyset, 2, \emptyset)
\]
\[
+ o (\emptyset, 10, 0.01, 0.05)
\]
\[
:= (100, 10, 2, 0.05)
\]

We can see that the peculiar mechanics of overlay addition provide a mechanism for applying default values to a list of parameters. The use of defaults clearly simplifies the function definition while also adding functionality since all permutations of arguments are handled in a single definition, which is codified in Theorem 2.1.18. This is a reflection of the less-is-more principle applied to function design.

**Theorem 2.1.18.** Let \( \Phi = (\phi_1, \phi_2) \). \( K \) be a overlay sequence of constants, and \( V \) be a overlay sequence of default values. \( W_{\phi_1} \subset W_{\phi_2} \Rightarrow \lambda W_{\phi_1}.(\lambda W_{\phi_2}.X)[W_{\phi_2} := W_{\phi_1} + o K] = \lambda W_{\phi_1}.(\lambda W_{\phi_2}.X)[W_{\phi_2} := W_{\phi_1} + o V] \)

**Proof.** Let \( K = V \).

The result of the above theorem may seem trivial, but the pragmatic implications are rather powerful. The equivalence between multiple function clauses and default values indicates that functions can be written in a compact form without losing polymorphic traits. Not only does this equivalence simplify function definitions, it also allows us to ignore certain arguments in function calls and replace them automatically with default values.

**Example 2.1.8.** Without Theorem 2.1.18, the \( \text{fgbm} \) implementation would require at a minimum four separate function clauses. In languages that don’t support default values in arguments (e.g. Java), this is precisely what happens.

\[
\text{fgbm}(n) \%as\% \text{fgbm}(n, 10)
\]
\[
\text{fgbm}(n, s) \%as\% \text{fgbm}(n, s, 0.01)
\]
\[
\text{fgbm}(n, s, \text{mean}) \%as\% \text{fgbm}(n, s, \text{mean}, 0.03)
\]
\[
\text{fgbm}(n, s, \text{mean}, \text{sd}) \%as\% \{\text{cumprod}(c(s, \exp((\mu - \text{sd}^2/2) / 252 + \text{sd} \cdot \text{rnorm}(n) / \sqrt{252})))\}
\]
FIGURE 2.5: The samplerange function

samplerange(x, size, window, ...) %when% {
  is.null(dim(x))
  window < length(x)
} %as% {
  count <- length(x) - window + 1
  samples <- sample.int(count, size, ...)
  t(sapply(samples, function(s) x[s:(s+window -1)]))
}

2.1.3 The ellipsis argument

When defining a function it is possible to group unreferenced arguments together into a special variable known as the ellipsis argument. When specified in the function signature any unbound arguments will be bound to the ellipsis argument. In turn this argument can be transformed into a list and accessed like a regular variable or passed on to another function unevaluated.

Definition 2.1.19. Let $\epsilon$ denote the ellipsis argument. Let $W$ be an argument list where $\epsilon \in W$, and let $S$ be a parameter list where $|S| > |W| - 1$. If $\exists S' \subset S$ where $S'$ is bound to $W' \subset W$ then $W := S \Rightarrow \epsilon = W -_{o} W' := S -_{o} S'$.

The idea of a greedy argument may seem strange but is actually quite useful. Suppose a function $f(V)$ depends on another function $g$ with a large argument list $W$. If this argument list needs to be accessible, rather than mirroring the arguments the ellipsis can be used to cleanly pass these parameters along. This situation arises when a function embeds plotting functions and those parameters should be exposed in the signature of the outer function. It also happens regularly when a function uses an arbitrary numerical method that may have tuning parameters that need to be exposed to an end user.

Example 2.1.9. The samplerange function in Figure 2.5 uses the ellipsis to pass arguments to the underlying sample.int function. The most significant of these are replace and prob, which dictate whether resampling should be used and the probability distribution for choosing samples, respectively. Exposing these parameters in samplerange via the ellipsis simultaneously increases the usability of the function while also maintaining a streamlined interface. This interface is codified by the argument list $W = (x, size, window, \epsilon)$. The function can be applied to the compatible parameter list $S = (rnorm(100), 50, 10)$, which binds $\emptyset$ to $\epsilon$. If instead $S = (rnorm(100), 50, 10, replace = TRUE)$, then $\epsilon := TRUE$.

\[ ^3 \text{In general I denote the ellipsis argument as $\epsilon$ to avoid confusion with the notation for describing arbitrarily long sequences.} \]
The process of binding parameters to arguments affects the notion of length. In particular, a bound argument list is no longer a overlap sequence (it is simply a sequence), so the conventional properties of sequences apply. The following proposition makes use of the binding idea insomuch that $\epsilon$ is bound to $\emptyset$. The consequence is that the unbound argument list (which is a overlap sequence) has length $|W|$ while the bound argument list has length $|W| - 1$.

**Proposition 2.1.20.** If $|S| = |W| - 1$ then $\epsilon = \emptyset$

**Proof.** Let $|S| = |W| - 1$ and $S' = S$. Then $\epsilon = W - \alpha W' := S - \alpha S' = \emptyset$. □

**Example 2.1.10.** Calling `samplerange` as `samplerange(rnorm(100), 50, 20)` results in $\epsilon$ being bound to $\emptyset$. This conclusion can be verified in the debugger.

```
> debug.lr(samplerange)
> samplerange(rnorm(100), 50, 20)
Browse[2]> epsilon <- list(...)
Browse[2]> length(epsilon)
[1] 0
```

Any explicit set of arguments can be replaced with the ellipsis argument when passed to another function. The significance is that high-level APIs do not need to repeat portions of lower level function signatures, since the parameters can be captured in the ellipsis argument.

**Theorem 2.1.21.** Let $W, V$ be argument lists. Given $W' \subseteq W$ and parameter constants $K' \in W$, $W' \cap K' = \emptyset$, also $W' \subseteq V$, then $(\lambda V.(\lambda W.X)[W := W' \cup K'])S = (\lambda V - W' + \alpha \epsilon.(\lambda W.X)[W := \epsilon + \alpha K']).$

**Proof.** From Definition 2.1.19 $\epsilon$ binds to any unbound parameters. Let $S' \subseteq S$ bound to $W'$. Then $(\lambda V.(\lambda W.X)[W := W' + \alpha K'])S \rightarrow (\lambda W.X)[W := S' + \alpha K']$. Also by Definition 2.1.19 $V - W' + \epsilon = V \Rightarrow \epsilon = W'$. Then $(\lambda V.(W') + \alpha \epsilon.(\lambda W.X)[W := \epsilon + \alpha K'])S \rightarrow (\lambda W.X)[W := S' + \alpha K']$. □

**Corollary 2.1.22.** The use of an ellipsis to pass arguments to a lambda term (closure) is equivalent to an infinite number of function calls.

**Proof.** Given $\lambda \epsilon.(\lambda W.X)[W := \epsilon], (\lambda \epsilon.(\lambda W.X)[W := \epsilon])S \Rightarrow \epsilon$ bound to $S$. Since $|S|$ is not bounded from above, $|\epsilon|$ is similarly unbounded. □

**Example 2.1.11.** Referring back to the `samplerange` function, any number of parameters can be captured in the ellipsis. Use of the ellipsis avoids having to define separate and distinct functions to pass different sets of arguments. Without the ellipsis, `samplerange` would need to define the arguments passed to the underlying `sample.int` function. The problem with this approach is that the arguments become required in the signature of the `samplerange` signature and is now tightly coupled with its specific implementation, such as `samplerange(x, size, window, replace, prob)`. Even
with default values, as in `samplerange(x, size, window, replace=TRUE, prob=NULL)`, the mirroring of the underlying implementation is sub-optimal since if the `sample.int` function were to be replaced, then there is a question of what to do with any new arguments that may be introduced by its replacement. Hence the ellipsis argument facilitates decoupling of the two function interfaces.

Another benefit of the ellipsis is that it acts as a simplification of a closure with equivalent behavior.

**Theorem 2.1.23.** Let $K$ be a overlay sequence of parameter constants. $\lambda V \cup \epsilon.X[W := K +\epsilon] = \lambda U.\lambda V.X[W := K + U]$.

**Proof.** Let $\epsilon = U$.

$$
\lambda V \cup \epsilon.X[W := K +\epsilon] = \lambda V \cup U.X[W := K + U] = \lambda U.\lambda V.X[W := K + U]
$$

\[\square\]

**Example 2.1.12.** Instead of always using the `sample.int` function within `samplerange` function, suppose we’d rather have the function passed as an argument. This can be accomplished equivalently by using a closure or the ellipsis. Calling `samplerange(rnorm(100), 50, 20, replace=TRUE)` is equivalent to calling `samplerange(rnorm(100), 20, function() sample.int(81, 50, replace=TRUE))` with the definition

```r
samplerange(x, window, f) %as% {
  t(sapply(f(window, size), function(s) x[s:(s+window -1)]))
}
```

In turn, this is equivalent to calling `samplerange(rnorm(100), 20, function() sample.int(81, 50, ...), replace=TRUE)` where `samplerange` is defined as

```r
samplerange(x, window, f, ...) %as% {
  t(sapply(f(window, size, ...), function(s) x[s:(s+window -1)]))
}
```

**Exercise 2.5.** Implement a function $\lambda window size.X$ that uses `sample.int` as the underlying implementation.

**Exercise 2.6.** Implement the function $\lambda window size \epsilon.X$.

### 2.1.4 The range operator

With so many ways to define and call functions, we need a device to think about how parameters are bound to arguments. The range operator helps us
reason about the number of parameters a given function clause will accept. With the introduction of optional arguments and the ellipsis argument, function clauses can accept more or less parameters than specified in the argument list. The lower bound of this range, \( \inf R[\phi] \), is the minimum number of parameters necessary to dispatch to \( \phi \). The upper bound, \( \sup R[\phi] \), is the maximum number of parameters allowed.

Before formally introducing the range operator, it is first necessary to introduce the set combinator \( \oplus \). This operator essentially sums the ordered pairs in the cartesian product.

**Definition 2.1.24.** Let \( X \) be a sequence and \( y \) a scalar. Then \( X \oplus y = X + y \), which is simply vector-scalar addition.

**Definition 2.1.25.** Let \( X, Y \) be sets. \( X \oplus Y = \{ x + y | x \in X \land y \in Y \} \). Unlike the cartesian product, \( \oplus \) is commutative.

**Theorem 2.1.26.** Given \( X, Y \) contiguous integer sequences, \( X \oplus Y = [\inf X + \inf Y, \sup X + \sup Y] \)

**Proof.** Let \( X = \langle x_1, x_2, \ldots, x_m \rangle \) and \( Y = \langle y_1, y_2, \ldots, y_n \rangle \), such that \( X, Y \subset \mathbb{N}_k, k \geq n, m \). Then \( \forall y_k \in Y \),

\[
X \oplus Y = \bigcup_{k} X \oplus y_k = X \oplus y_1 \cup X \oplus y_2 \cup \cdots \cup X \oplus y_n = X \oplus y_1 \cup x_m + y_2 \cup \cdots \cup x_m + y_n
\]

**Example 2.1.13.** Let \( X = [1, 4], Y = [3, 5] \).

\[
X \oplus Y = \{1, 2, 3, 4\} \oplus \{3, 4, 5\} = \{4, 5, 6, 7\} \cup \{5, 6, 7, 8\} \cup \{6, 7, 8, 9\} = \{4, 5, 6, 7, 8, 9\} = [4, 9]
\]

**Definition 2.1.27.** Every argument \( w \in W \) has a range that specifies the number of parameters that can be bound to it. The range operator is defined as

\[
R[w] = \begin{cases} 
[0, 1], & \text{\( w \) an optional argument with a default value} \\
[0, \infty), & \text{\( w = \epsilon \)} \\
1, & \text{otherwise}, 
\end{cases}
\]

and \( R[w] \subset \mathbb{N} \).

\(^4\)Standard \( R \) is even less restrictive insomuch that a function can be executed even if an argument with no default is missing.
FIGURE 2.6: A second clause to the samplerange function

```r
samplerange(x, size, window, ...) \%when\% {
  window < length(x)
} \%as\% {
  count <- nrow(x) - window + 1
  samples <- sample.int(count, size, ...)
  lapply(samples, function(s) x[s:(s+window -1),])
}
```

**Definition 2.1.28.** The range of arguments in a function clause with argument list \( W \) is \( R[\phi] = R[W] = R[w_1] \oplus R[w_2] \oplus \cdots \oplus R[w_n] \).

**Theorem 2.1.29.** \( R[W] = [\sum \inf w_k, \sum \sup w_k] \)

**Proof.** When \( |W| = 1 \), then by definition \( R[W] = [\inf w_1, \sup w_1] \). Assume true for \( |W| = n \). Then for \( |W| = n + 1 \), let \( W' = W \cup k \).

\[
R[W \cup k] = R[W] \oplus R[k] \\
= [\sum \inf w_1, \sum \sup w_1] \oplus R[k] \\
= [\sum \inf w_i + \inf k, \sum \sup w_i + \sup k]
\]

\[\square\]

**Example 2.1.14.** The range of \( \text{samplerange} \) is

\[
R[\text{samplerange}] = R[x] \oplus R[\text{size}] \oplus R[\text{window}] \oplus \epsilon \\
= 1 + 1 + 1 + [0, \infty) \\
= [3, \infty)
\]

**Corollary 2.1.30.** The range of an argument list is the difference between two sub sequences in \( \mathbb{N} \). Let \( j = \inf R[W] \), \( k = \sup R[W] \). Then \( R[W] = \begin{cases} k, & k = j \\ \mathbb{N}_k - \mathbb{N}_j, & j < k \end{cases} \)

These properties are used internally to select the correct function clause given a parameter list. It is therefore convenient to reason about function dispatching using the same mechanism.

**Example 2.1.15.** The signature of \( \text{fgbm} \) has a range of

\[
\inf R[\text{fgbm}] = \sum_k \inf R[w_k] \\
= \inf R[n] + \inf R[s] + \inf R[\text{mean}] + \inf R[\text{sd}] \\
= 1 + 0 + 0 + 0 \\
= 1
\]
\[ \sup R[\text{fgbm}] = \sum_k \sup R[\mu_k] \]
\[ = \sup R[n] + \sup R[s] + \sup R[\text{mean}] + \sup R[\text{sd}] \]
\[ = 1 + 1 + 1 + 1 = 4 \]

Therefore, \( R[\text{fgbm}] = [1, 4] \).

The range operator is sufficient for reasoning about functions when required arguments are listed at the beginning of a function definition. If optional arguments are specified before required arguments, then these arguments are effectively required unless called with explicit names.

While it is possible to create a formalism sophisticated enough to accommodate arbitrary signatures the complexity erodes the value of the model. The goal of this book is to develop just enough rigor to make the theory useful, so it is unnecessary to develop an exhaustive formalism. Hence as a general rule, required arguments should always precede optional arguments and the ellipsis. On the other hand, the ellipsis can be placed arbitrarily within a sequence of optional arguments. This is an important point that not only underscores the model of the formal system but also the models built using the system.

**Theorem 2.1.31.** Let \( f \) be a function composed of function clauses \( \Phi \). Then \( R[\Phi] = \bigcup R[\phi_k], \forall \phi_k \in \Phi \).

**Proof.** Let \( \Phi = \phi \). Then \( R[\Phi] = R[\phi] \). Now let \( \Phi = \langle \phi_1, \phi_2, \ldots, \phi_n \rangle \). Assume that \( R[\Phi] = \bigcup_k R[\phi_k] \). Then
\[
R[\Phi + \langle \phi_{n+1} \rangle] = R[\Phi] \cup R[\phi_{n+1}]
\]
\[
= \bigcup_k R[\phi_k] \cup R[\phi_{n+1}]
\]
\[
= \bigcup_{k=1}^{n+1} R[\phi_k]
\]

While a function’s range is also a subset of the natural numbers, there is no guarantee that the \( R[\Phi] \) maps onto \( \mathbb{N}_k \). Notice that the range operator doesn’t say anything about the order of the function clauses and is therefore commutative, whereas actual function application is dependent on the order of \( \phi_k \in \Phi \).

**Example 2.1.16.** Unlike \( \text{fgbm} \) the \( \text{rgbm} \) function has multiple function clauses, so the range of the function is the combined range of the clauses. Let \( \Phi = \text{rgbm} \).
\[ R[\phi_1] = [1, 4] \]
\[ R[\phi_2] = 5 \]
\[ R[\phi_3] = 5 \]

Therefore \( R[\Phi] = [1, 5] \).

### 2.1.5 Pattern matching and guard expressions

We already know that functions can be defined with multiple clauses. Differentiating between function clauses is easy when each clause has a different range. However, what happens when function clauses have overlapping ranges? Under this scenario, function clauses can be differentiated by using pattern matching or explicit guard expressions. This concept is a feature of both mathematical notation and lambda.r that, among other things, makes it easy to define boundary conditions. In the next few sections we’ll see different approaches to differentiating function clauses that have the same signature and how to leverage this in programs. In mathematics pattern matching is used in equations to establish initial conditions and for defining recursive functions. With lambda.r a similar construction can be used. Examples include recursive sequences like the Fibonacci or Bernoulli numbers. From a programming perspective pattern matching also provides a mechanism for managing certain types of control flow that retain mathematical properties. This works the same way that multipart mathematical definitions implicitly control flow by creating a declarative structure that defines the individual logical cases.

The syntax for pattern matching is the same as a normal function definition except a scalar value is used instead of a variable. This implies that within the function body the argument can only be referenced by the literal scalar value instead of by name. Note that if this restriction becomes burdensome, usually it indicates that the technique is being abused.

**Example 2.1.17.** The implementation of `rgbm` in Figure 2.1 uses a recursive definition. The reduction process starts by executing the second function clause. When \( n \) becomes 0 the function application matches the first function clause and returns the result.

\[
gbm(10) = gbm(10, 10, 1, c())
\rightarrow gbm(9, 10, 1, c(x_1))
\rightarrow \cdots
\rightarrow gbm(0, 10, 1, c(x_1, x_2, \cdots, x_{10}))
= c(x_1, x_2, \cdots, x_{10})
\]

Aside from concisely handling recursive definitions pattern matching is also
useful for encapsulating conditional expressions. Code with nested conditional blocks impede reasoning and should be avoided. In section 1.3.2 we saw how to use a function to achieve the same result.

Pattern matching is a convenient feature, but it is limited in expressive power. When more complicated conditions need to be defined a guard expression must be used instead. The intuition for guard expressions derives again from multipart functions.

**Example 2.1.18.** The definition for the `confine` function is an example of a multipart function definition. This function is used to ensure that signals are well formed.

\[
\text{confine}(x) = \begin{cases} 
-1, & x < -1 \\
1, & x > 1 \\
x, & \text{otherwise}
\end{cases}
\]

Suppose we want to use the slope of a moving average as a Signal. We can use standard polynomial interpolation followed by differentiation to find the slope. Obviously the slope of a function can take on any real value. To limit its output the `confine` function can be used to bound the function so it can be used as a Signal. Figure 2.7 shows a generated time series, and its bounded version following a transformation by `confine`.  

A guard is simply the condition associated with a given case in a multipart definition. Consequently any conditional expression is valid such that a true result will evaluate the given function clause. The same definition in lambda.r is

\[
\text{confine}(x) \%\text{when}\% \{ x < -1 \} \%\text{as}\% -1 \\
\text{confine}(x) \%\text{when}\% \{ x > 1 \} \%\text{as}\% 1 \\
\text{confine}(x) \%\text{as}\% x
\]

A guard expression can reference any function argument for the given clause. Any arbitrary expression is allowed given that the result is deterministic. In practice, guards should be kept computationally simple to avoid introducing performance bottlenecks. Multiple expressions are collected together in a guard block. When multiple expressions exist then each must evaluate to `TRUE`. The function clause is then considered compatible with the given parameter list and is executed.

Guards are defined using a similar construction as lambda abstractions.

**Definition 2.1.32.** Given \(\lambda W.X, \gamma W.\bigwedge p_k(w_j)\) is the corresponding guard block, where \(p_k\) is a predicate function that represents a guard expression returning either true or false and \(w_j \in W\).

A function clause can only be applied to an argument list if all predicates within a guard block resolve to `TRUE`. In the next section we will see that this is defined as guard compatibility.

---

5In signal processing this is called clipping and happens automatically when a signal exceeds the dynamic range of the ADC.
FIGURE 2.7: The behavior of confine
Example 2.1.19. Suppose we want to ensure that our `confine` function rejects any values that are not numeric. The guard on the first clause is expressed as $\gamma x.\text{is.numeric}(x) \land x < -1$. In `lambda.r` this is

```r
confine(x) %when% { 
  is.numeric(x)
  x <- 1
} %as% -1
```

where each guard expression is placed on its own line.

It should be clear that pattern matching is simply a special type of guard. Given our understanding of guards, it is possible to define pattern matching in terms of guard expressions.

Definition 2.1.33. Given an argument list $W$, a literal value $\nu_k \in W \Rightarrow \gamma W.\nu_k = \nu_k$.

Example 2.1.20. The recursive definition of the GBM generator uses literal values to control the function dispatching. The second clause is equivalent to

```r
rgbm(n, s, mean, sd, acc) %when% { 
  n == 0
} %as% acc
```

This guard is expressed as $\gamma W.n = 0$.

There are three special patterns that can be used in function signatures. The first two mirror the special values `NULL` and `NA` and are written as such. Since in R these cannot be matched using equality, they are automatically transformed into equivalent guard expressions by `lambda.r`. The `EMPTY` keyword is similar. When used in a function signature it is transformed into a guard expression that tests for a 0 length vector.

Definition 2.1.34. Given an argument list $W$, if $\exists \nu_k = \text{EMPTY}$, then there is a corresponding guard $\gamma W.\text{length}(\nu_k) = 0$.

Exercise 2.7. Generalize `confine` so that it can use any value as the bound.

Exercise 2.8. Write the guards for `onlyif` in Figure 1.4

2.1.6 Parameter compatibility and dispatching

Multipart mathematical functions are silent about the mechanics of how one clause is executed versus another. In the corresponding computer program, it is necessary to know which function clause will be executed. Providing a formalism around this it is possible to then symbolically transform a function into its specific function clause body. The base behavior of R is to dispatch to a function based on the name alone. This means that a function signature does not need to match a parameter list, which seems illogical. Since R is a lazy language, parameters are not bound to arguments until the arguments are referenced. This results in some behaviors that may seem surprising.
> f <- function(x, y) x
> f(2)
[1] 2

In lambda.r dispatching is dictated by the mechanics of the range operator, so scenarios like this are not allowed. Prohibiting such function calls is necessary to properly implement multipart functions and maintain referential transparency. When calling a function with multiple clauses the language provides rules for deciding which function clause to execute. This process is known as dispatching and is defined in terms of parameter-argument compatibility. Dispatching is essentially an indexing function that maps a parameter list to an index \( k \) representing the function clause to execute.

Functional dispatching in lambda.r has three distinct attributes that must be considered in order to know which particular function clause will be dispatched. The first consideration is whether the parameter list is within the range of the function clauses. Guard expressions must all evaluate to true. Finally the types must match between the argument list and the parameter list. If a function clause is selected based on a given parameter list \( S \), then it is said that \( S \) is compatible with \( W \), which is denoted \( S \sim W \).

**Definition 2.1.35.** A parameter list \( S \) is signature-compatible with a function clause \( \phi = (f, W) \), denoted \( S \sim W \) if and only if \( |S| \in R[\phi] \) and \( N[S] \subseteq N[W] \).

**Example 2.1.21.** Since \( R[\text{confine}] = 1 \), \( \text{confine}(5) \) is compatible but \( \text{confine}(5,6) \) is not compatible. Alternatively \( \text{confine}(g=5) \) is also not compatible since \( N[W] = \langle x \rangle \neq \langle g \rangle \).

**Definition 2.1.36.** \( S \sim \phi \) if \( (\gamma W.p(W))[W := S] \) is true. Under this condition \( S \) is said to be guard-compatible with \( \phi \).

**Example 2.1.22.** When passing a character parameter \( \text{confine}("a") \sim W \) but \( \text{confine}("a") \not\sim W \).

**Definition 2.1.37.** Given \( W' \subseteq W \), \( S \sim \phi \) if \( T[S] = T[W'] \), \( N[S] = N[W'] \). Under this condition \( S \) is said to be type-compatible with \( \phi \).

**Definition 2.1.38.** \( S \sim \phi \) if and only if \( S \sim \phi \land S \not\sim \phi \land S \not\sim \phi \) is true. We say that \( S \) is compatible with \( \phi \).

Function clauses are evaluated in the order they are defined. Hence the first compatible function clause will be used when applying a parameter list to the function.

**Definition 2.1.39.** Let \( \langle \phi_k, W_k \rangle \in \Phi \). Then \( \Phi(S) = \text{argmin}_k \phi_k(S) \), such that \( S \sim W_k \). Given \( S \), we say that \( \Phi \) dispatches to \( \phi_k \).

\footnote{In S3 dispatching a function name is constructed from the type of the first parameter, which is appended to the end of the base name. For example calling \( f(x,y) \) where \( x : X \) results in calling function \( f.X(x,y) \).}
Adding new function clauses does not change the compatibility or precedence of existing dispatching. If $\Phi(S) \rightarrow \phi_k$, then $\Phi(S) \rightarrow \hat{\phi}_k$, where $\hat{\Phi}(S) = \langle \phi_m \rangle \cup \langle \hat{\phi} \rangle$, $\phi_m \in \Phi$. This is an important point as our analysis is stateless, yet within the interpreted environment function are built incrementally. Hence at two points in time a function $\Phi_t$ is not necessarily equal to $\Phi_{t-1}$.

If $|S \sim \Phi| = 1$, then $\Phi'(S) \rightarrow \phi_k$ when $\Phi(S) \rightarrow \phi_k$. However, if $|S \sim \Phi| > 1$, no such guarantee can be made.

Example 2.1.23. Sometimes it is necessary to confine values to a fixed range, while other times it is necessary to quantize values to a smaller set of values. In some ways this can be considered a form of numerical classification. Suppose we want to map the real numbers to the set $\{-1, 0, 1\}$. The quantize function maps values in $\mathbb{R}$ to any finite set based on the specified distance metric.

```r
quantize(x, bins=c(-1,0,1), metric=function(a,b) abs(a-b)) %as% {
  ds <- sapply(bins, function(b) metric(x,b))
  apply(ds,1, function(d) item(bins, which.min(d)))
}
```

2.2 Types and type constraints

Mathematics is littered with sets that contain specific types of stuff. Set theory is a branch of mathematics that provides a grammar to describe and study collections of like things. As the restrictions on (and properties of) sets increase, different names are attributed to them. In abstract algebra a group is a set plus an operator that satisfies four fundamental properties: closure over the operator, associativity, identity, and has an inverse [2]. Rings are more specialized and are defined over two operators (addition and multiplication). The ring must be a group over addition and also possess left and right distributivity and multiplicative associativity [4]. These sets can be transformed from one set to another, and depending on the properties that are preserved the transformation has a different name (eg isomorphism vs homomorphism).

In computer science the notion of a set translates to that of a type or class. Members of a mathematical set such as a ring have strict properties, and we expect the same to hold for types. It is natural to expect that operations valid for one element of a type are valid for all other elements of that type. Mathematical operations are applicable over many sets, and the same is true in programming. The notion that the same operation is relevant over different sets is called polymorphism. We take polymorphism for granted when

---

7The word class is generally reserved for object-oriented programming and is used when discussing S3 and S4 object systems.
FIGURE 2.8: Quantizing a time series with 2 and 3 bins
thinking about simple sets like the natural numbers, integers, polynomials, or even matrices. Operations like addition are well defined for these sets and conceptually perform equivalent operations regardless of the set. This generality is in fact built into the definition of a group. Where this becomes less straightforward is with more complex operations like exponentiation. Although the concept of a power transfers without a hitch, the definition of a root is more complicated for matrices. Furthermore division only exists as multiplication by an inverse, which only exists if the matrix is singular. Conceptually we can make the operation consistent, but the exact mechanics of the operation is determined by the set over which the operation is defined. So types provide a context for operations and by extension functions. Depending on the type of an argument, we can control the mechanics of the function for the specific type ensuring it behaves as we expect.

2.2.1 Type constructors

When thinking about numbers we usually don’t consider the need to define how they are created. A 1 is simply a 1 and doesn’t require any special rules to create it since numbers are literal values. For more complex mathematical objects type constructors are common, but we generally don’t characterize them as such. Moving beyond numbers, creating a sequence of numbers has explicit construction rules. The well known Fibonacci sequence is defined by an equation and two initial conditions. Similarly, a matrix can be constructed numerous ways. Each method has a set of rules that must be satisfied to produce a well-formed matrix. The standard approach is to use the special bracket notation to create the matrix, such as

\[
A = \begin{bmatrix}
x_{11} & x_{12} & x_{13} \\
x_{21} & x_{22} & x_{23} \\
x_{31} & x_{32} & x_{33}
\end{bmatrix}.
\]

This notation is really a type constructor from the perspective of initializing the matrix elements explicitly. The same is true of other matrix construction techniques such as with a set of column vectors: \( A = [\vec{x}\vec{y}\vec{z}] \) or row vectors:

\[
A = \begin{bmatrix}
\vec{x}^T \\
\vec{y}^T \\
\vec{z}^T
\end{bmatrix}.
\]

Clearly matrices can be constructed numerous ways. These type constructors translate to programming languages directly except that the bracket notation is replaced with a function call. For example, given vectors \( x, y, z, A = \text{matrix}(x, y, z) \). Despite the syntactic difference, the operation is semantically the same. We will exploit this observation throughout the book.

**Definition 2.2.1.** A type \( X \) is the range of a function \( \Phi_X : A \rightarrow X \)

(a) The function \( \Phi_X \) is called the type constructor of \( X \)

(b) A typed variable \( x \in X \) is denoted \( x : X \)

(c) The name of the type constructor is the name of the type
FIGURE 2.9: The Signal type constructor

\[
\begin{align*}
\text{Signal}(x) & \%::\% \text{ numeric} : \text{ numeric} \\
\text{Signal}(x) & \%as\% \text{ quantize}(x) \\
\text{Signal}(x) & \%::\% \text{ logical} : \text{ numeric} \\
\text{Signal}(x) & \%as\% \text{ ifelse}(x, 1, -1)
\end{align*}
\]

From the definition we can see that a typed object is nothing more than a named data structure, where the rules of membership are governed by the type constructors. Syntactically types in lambda.r are denoted using PascalCase while variables are lower_case.

Since type constructors are functions they can have multiple clauses like any other function. The implication is that a single clause does not necessarily map the input onto \(X\).

**Proposition 2.2.2.** Let \(\Phi = \langle \phi_k \rangle\) be a type constructor. Then \(\forall \phi_k \in \Phi\)

(a) \(\bigcup \phi_k(W)\) maps onto \(X\)

(b) \(\phi_k(W) \subseteq X\)

**Proof.**

(a) By definition \(\Phi\) maps \(W\) onto \(X\). Since \(\Phi = \langle \phi_k \rangle | k \in J_n, |\Phi| = n\).

(b) Let \(\Phi = \langle \phi_1 \rangle\). Then \(\phi_1\) maps onto \(X\). Let \(X_1 \neq X_2 \subseteq X\) and \(\Phi = \langle \phi_1, \phi_2 \rangle\), where \(\phi_1 \to X_1\) and \(\phi_2 \to X_2\).

\(\square\)

**Example 2.2.1.** The \texttt{Signal} type defined in Figure 2.9 is simply an integer in the set \([-1, 0, 1]\). Hence \(X = \{-1, 0, 1\}\). Let \(\Phi = \langle \phi_1, \phi_2 \rangle\) be the \texttt{Signal} type constructor. The domain of \(\Phi\) is the union of the domain of each function clause \(\phi_k \in \Phi\). Hence \(A = \mathbb{R} \cup \{\text{true}, \text{false}\}\).

Programming languages have a set of primitive types and custom types that may be defined by a user of the language. Primitives generally do not have explicit type constructors since they are created using literal values. Primitive types in R are logical, numeric, complex, character, or raw [21]. I denote this group of types (technically modes) \(^8\) as \(T\). For our purposes there is no distinction between built-in primitives and bespoke types, so long as the rules of construction are explicit.\(^9\) A full treatment of types including principles of their design appears in Chapter 4.

\(^8\)In R there are multiple attributes that appear to define a type: mode, type, class. For this book usually only the class is relevant. In lambda.r we say type to distinguish from object-oriented semantics, but the underlying mechanics are based on the S3 class system. The reasoning is that existing R functions are compatible with lambda.r since they use the same underlying structure.

\(^9\)This isn’t completely true as vectors are atomic and can only support the primitive types listed above.
An object is typically defined as an instance of a type. In R typing is dynamic, meaning that it is not declared explicitly. For example, when creating a sequence of integers the returned vector has a defined type even though it is not specified explicitly. Hence this vector is an instance of the numeric type.

Exercise 2.9. Show that Proposition 2.2.2 holds for the Heaviside step function.

2.2.2 Type hierarchies and dispatching

The relationship between a subset and its parent set can be considered a type hierarchy. Suppose that there are two types $A$ and $B$, where $B$ is a sub-type of $A$. Then any element of $B$ is also an element of $A$, which means that $B \subseteq A$.

I use the same notation to indicate that a type descends from another type.

Theorem 2.2.3. Let $A, B$ be types such that $B \subseteq A$. Define an argument list $W$ and parameter list $S$, where $s_k : A \in S$. If $S \sim W$, then for $S' = (\emptyset, \ldots, s_k : B, \ldots, \emptyset) \circ_o S$, $S' \sim W$.

Proof. Since $B \subseteq A$, $s_k : B \in A \Rightarrow s_k : A$. Hence $S' \sim W$. \qed

Proposition 2.2.4. Below are some properties of type hierarchies.

(a) Let $C \subseteq B$ and $B \subseteq A$. Then $C \subseteq A$. 
FIGURE 2.11: Operations on matrices

<table>
<thead>
<tr>
<th>Type</th>
<th>Property</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix</td>
<td>Has rows, columns</td>
<td>$AB, A + B, A^T$</td>
</tr>
<tr>
<td>SquareMatrix</td>
<td>row = column</td>
<td>$\text{tr}(A), \det A$</td>
</tr>
<tr>
<td>SymmetricMatrix</td>
<td>$A^T = A$</td>
<td>$D = Q^T AQ$</td>
</tr>
<tr>
<td>SingularMatrix</td>
<td>$\det A = 0$</td>
<td>$A = UDV^T$</td>
</tr>
<tr>
<td>PositiveDefiniteMatrix</td>
<td>$z^T A z &gt; 0$</td>
<td>$A^{-1}$</td>
</tr>
</tbody>
</table>

(b) $B \subseteq A$ and $A \subseteq B$ if and only if $B = A$.

All types in lambda.r make use of this basic principle since bespoke types extend built-in types.

**Example 2.2.2.** We’ve seen that the Signal type consists of just three elements: $\{-1, 0, 1\}$. Since the values returned by the type constructor are simply integers, the Signal type formally extends numeric.

To construct a portfolio, we need the result of multiple Signals, one per asset. Suppose we want to know whether the portfolio is net long or short. This is simply $\sum(\text{signals})^{10}$. Since $\text{Signal} \subseteq \text{numeric}$, $\text{Signal} \sim \sum$. It is easy to simulate a set of signals using the sample function, such as

```r
s <- sapply(sample(c(-1, 0, 1), 100, replace=TRUE), Signal)
sum(s)
```

Returning to linear algebra there are many types of matrices to the chagrin of students. Some are square matrices, others are symmetric or positive definite, each with their own set of properties and allowable operations. As the properties on matrices increase so do the operations and inferences that can be performed based on its type. Type hierarchies automatically provide type inheritance so functions defined for a base type are applicable to a subtype. Table 2.11 illustrates the operations that can be performed on matrices depending on the properties they possess.

Type inheritance is directed and non-symmetric. Sub types are specializations of parent types, so it is only natural that operators specific to a sub type do not necessarily apply to the parent type.

**Proposition 2.2.5.** Let $\Phi_A$ be the set of functions compatible with type $A$. Given $T[W] \in A \forall W \in U_B$ and $B \subseteq A$ then $B \sim \Phi_A$ but $A \not\sim \Phi_B$.

**Proof.** Show that $A \not\in B$. (INCOMPLETE)

**Example 2.2.3.** When building our portfolio only values of type Signal are meaningful. Although derived from numeric, values outside of the range

$10$Note that the standard $\text{sum}$ function is described by $\text{sum} : \text{numeric} \times \text{logical} \rightarrow \text{numeric}$. 
[-1, 1] are not defined. Meaningless calls like \texttt{Signal(2)} can be handled by either an intermediate call to \texttt{confine} or \texttt{quantize}, depending on whether values should be continuous or discrete, respectively.

**Exercise 2.10.** Create a type to encapsulate a time series generated by \texttt{fgbm}. What conditions must be met to create values of this type?

**Exercise 2.11.** Instead of GBM, create a type that represents asset prices. Are the criteria for the type different than for GBM?

### 2.2.3 Attributes and orthogonal data structures

Custom types are useful but most functionality in the language is built around the core data types of R. Creating lots of custom types, while seemingly attractive from an organizational perspective, results in a lot of extra tedious data management to coerce these bespoke types to be compatible with existing functions. What would be more useful is a technique where existing data types can be used yet somehow a distinction can be made between different objects. The attribute mechanism in R does just this by allowing orthogonal data structures to be attached to an object of a given type. In many cases there is no need to create an elaborate data structure and in fact predefined data structures can typically be used with the judicious use of attributes to manage metadata.

**Definition 2.2.6.** A \textit{variable} is a data structure where \texttt{variable} \Rightarrow (\texttt{value}, \texttt{attributes}). The value \texttt{attributes} is of type list. Dereferencing a variable yields its value.

**Definition 2.2.7.** The \texttt{attribute} operator \texttt{A[variable]} accesses attributes of a variable, such that \texttt{A[variable] = attributes}.

Operations on a variable with attributes have idiosyncratic preservation rules.

**Axiom 2.2.8.** Let \texttt{A[a] \neq \emptyset}, \texttt{A[b] \neq \emptyset}.

(a) \texttt{A[a op b] = A[a] \cup A[b] \neq A[a], op \in \{+, -, *, /, \%, \%^\}}

(b) \texttt{A[c(a, b)] = \emptyset}

(c) \texttt{A[a_k] = \emptyset}

(d) \texttt{A[list(a, b)] = list(A[a], A[b])}

Any list can be transformed into a simple built-in type \textsuperscript{11} with attributes holding the remaining elements of the list. This equivalence between attributes and lists can eliminate the need for complex and restrictive type hierarchies.

\textsuperscript{11}This is more than \texttt{T} and includes \texttt{data.frames}, \texttt{matrices}, \texttt{arrays}, etc.
Theorem 2.2.9. \( \text{list}(V) \sim v_k, \text{such that } A[v_k] = \text{list}(V - v_k). \)

Proof. By Definition 2.2.7 an arbitrary list can be transformed into a primitive type with attributes.

Similarly, any type can be transformed into a primitive with attributes attached to it.

Theorem 2.2.10. Let \( T[a] = A : \text{list}. \text{If } \exists a \in T \in a, \text{then } a \rightarrow a_k, \ A[a_k] = a - a_k. \)

Proof. By Proposition 2.2.5

Example 2.2.4. Using our GBM generator we produce a stochastic time series. The resultant series says nothing about the parameters that created it. Rather than creating a custom data structure to encapsulate and carry this meta-data around, it is simpler to add attributes describing the underlying process.

\[
\text{fgbm}(n, s, \text{mean}, \text{sd}) \%as\% \{
    y <- \text{cumprod}(c(s0,}
    \exp((\text{mu} - \text{sigma}^2/2) / 252 + \text{sigma}\ast\text{rnorm(n-1)} / \text{sqrt}(252))))
    y@\text{mean} <- \text{mean}
    y@\text{sd} <- \text{sd}
    y
\}
\]

The alternative is a list or type that stores all this data explicitly.

\[
\text{fgbm}(n, s, \text{mean}, \text{sd}) \%as\% \{
    y <- \text{cumprod}(c(s0,}
    \exp((\text{mu} - \text{sigma}^2/2) / 252 + \text{sigma}\ast\text{rnorm(n-1)} / \text{sqrt}(252))))
    \text{list(\text{series}=y, \text{mean}=\text{mean}, \text{sd}=\text{sd})}
\}
\]

This second approach makes it more cumbersome to work with the data and changes how we reason about the data. In the first version, the function’s range is \( \mathbb{R} \). In the second version it is now \( \text{list} \), which reduces the properties that can be inferred from the function. Use of the data structure necessarily requires transformations just to extract the underlying value in \( \mathbb{R} \).

\[
x <- \text{fgbm}(100, 10)
x@\text{series}
\]

There are no hard and fast rules about choosing attributes over types for a data model. Like most engineering problems, it depends on the specific circumstances of the situation. Ultimately the choice is governed by which one is easier to reason about and simpler to use.
2.2.4 Type constraints

How does lambda.r know how to dispatch between one typed function versus another? I hinted at the answer when I mentioned type compatibility in Section 2.1.6. A type constraint dictates whether a parameter list is type compatible with a function clause. Syntactically, type constraints are a special declaration that tell lambda.r what types are allowed for each argument of the function along with its return type. The syntax has a form that is equivalent to describing the domain and range of a function.

\[
f : A_1 \times A_2 \times \cdots : A_n \rightarrow B
\]

\[
f(w_1, w_2, \cdots, w_n) \%::\% A_1 : A_2 : \cdots : A_n : B
\]

This declaration has the effect of enforcing strong typing on the function clause such that the types of the parameter list must match the types of the argument list in order to dispatch to the function clause.

**Definition 2.2.11.** Let \( \phi = (f, W) \). \( T[\phi] = T[W] = \langle T[w_1], T[w_2], \cdots, T[w_n] \rangle \).

**Example 2.2.5.** We can formally define a type that represents a portfolio based on a set of signals. For the portfolio to be meaningful only **Signals** are allowed as input arguments to the **Portfolio** type constructor. The type constraint is \( \text{Portfolio}(\ldots) \%::\% \text{Signal} \ldots : \text{list} \).

**Definition 2.2.12.** \( S \sim \phi \) if and only if \( \bigcap T[s_k] \subset T[w_k], \forall s_k \in S, w_k \in W \).

This definition is consistent with type hierarchies. The return type is not included in the definition for type compatibility. Once a function is executed the return type is validated. If it does not match the type specified in the type constraint then the function call will fail. This usually only happens if a function implementation is incorrect.

**Proposition 2.2.13.** If \( B \subset A \) then \( B \in T[a], a \in A \).

**Proof.** See Proposition 2.2.5.

**Example 2.2.6.** The **crossover** function can use a type constraint to ensure that input is well-behaved. A time series is considered to cross another when its magnitude changes from being less than to greater than the other time series. Our intuition is that this function is most applicable to real-valued time series where each pair of scalars are compared. However, if we consider magnitude as the norm, then the idea of a crossover can apply to \( n \)-dimensional time series as well as matrices. This would lead to multiple type constraints, where the simplest is for numeric vectors.

> crossover(a, b) \%::\% numeric : numeric : Signal
> crossover(a, b) %as% Signal(a > b)

> crossover(rnorm(10), rnorm(10))

[1] -1 -1 -1 1 1 1 -1 -1 -1 1
Attempting to apply the function to non-numeric arguments results in an error.

```r
> crossover(rnorm(10), c(TRUE,FALSE))
Error in UseFunction(crossover, "crossover", ...) :
  No valid function for 'crossover(c(-1.96616163687446, 
  -0.736753030351807,0.388075382585652,0.373931253748484, 
  ..., logical)'
```

Example 2.2.7. It can be beneficial to define a Price type that extends numeric. This type is essentially \( \mathbb{R}_+ \), where any values not in \( \mathbb{R}_+ \) are assigned the value NA. Here it is easy to see how a sub type is equivalent to a subset, so clearly elements of Price will be compatible with the type constraint for crossover.

```r
> crossover(Price(rnorm(10, mean=5)), Price(rnorm(10, mean=6)))
[1]  1 -1 -1 -1 -1 -1 -1 -1  1 -1
```

Type constraints are greedy in that they apply to all successive function clauses that are compatible. This is a syntactic convenience to reduce clutter and typing when multiple function clauses are defined.

Example 2.2.8. The confine function has three function clauses but only requires a single type constraint.

```r
c confine(x) %::% numeric : numeric
c confine(x) %when% { x < -1 } %as% -1
c confine(x) %when% { x > 1 } %as% 1
c confine(x) %as% x
```


Exercise 2.13. Define a Price type that is compatible with Example 2.2.7.

Exercise 2.14. Show how to use Price in the implementation to generate a moving average. What is the return type of the type constraint?

Exercise 2.15. Write the definition of the Heaviside function using a type constraint. How many type constraints are needed?

2.2.5 Type variables

Sometimes explicit types can be too restrictive. R is dynamically typed so the flexibility of functional polymorphism appears readily in the language. As with arithmetic operators it is beneficial to have a single set of common operators that behave consistently across various types. Statically declaring explicit types in a constraint can have the opposite effect, by overly restricting the function arguments. Consistently using the base types \( \mathbb{T} \) removes some of the limitations, since type inheritance can be leveraged. But sometimes
operations are valid over wildly different classes of data. Type variables serve as a compromise to preserve this sort of polymorphism while also providing type safety by focusing on describe the relationship between types as opposed to the specifying the exact names of the types. Syntactically, a type variable is a single lower case letter in a type constraint. It indicates that an argument is some unspecified arbitrary type. Referencing a type variable multiple times in the same type constraint has the effect of requiring that each argument associated with the given type variable must be of the same type. Inclusion of a second type variable states that arguments associated with the first type variable have a type that is different from the type of the second type variable.

Much like how arguments are bound to parameters during function application, a type variable is bound to the type of a parameter.

**Definition 2.2.14.** Let \( \tau = T[w_k] \) be a type variable. Given \((\lambda W.X)[W := S], \tau := T[s_k]\).

Type variables are fully compatible with explicit types insomuch that a type constraint can contain both type variables and concrete types. The uniqueness constraint only applies to the type variables such that a bound type variable can be the same as an explicitly declared type in the constraint.

**Example 2.2.9.** The implementation of `crossover` works equally well for arguments of type numeric, integer, and logical. What is important is that each argument has the same type, which can be codified using a single type variable.

```latex
crossover(a, b) :: d : d : Signal
```

Any single lowercase character is valid as a type variable, so for clarity’s sake I chose one that does not appear in the argument list.

**Proposition 2.2.15.** Let \( \tau \) be a type variable. If \( T[w_k] = \tau \) and \( T[w_j] = \tau \), then \( T[s_k] = T[s_j] \).

**Proof.** (INCOMPLETE) \( \square \)

**Example 2.2.10.** Specifying multiple type variables indicates that the types of arguments are distinctly different. This is useful when you know that the possible types for the two arguments are disjoint. It is possible (and limiting) to add such a constraint to `map`, which would look like

```latex
map(x, fn, y) ::: a : Function : b : b
```

Compare this to specifying the same type variable, which requires the values to have the same type.

```latex
map(x, fn, y) ::: a : Function : a : a
```

**Proposition 2.2.16.** Let \( \tau \) be a type variable. If \( T[w_k] = \tau_a \) and \( T[w_j] = \tau_b \), then \( T[s_k] \neq T[s_j] \).
Proof. (INCOMPLETE) □

Example 2.2.11. Suppose we want to extend the Signal type to allow a consumer of the type to specify the conversion function. So long as the function returns a value in the set \{-1, 0, 1\}, it is valid. Since the type of the function argument is unknown, a type variable can be used. One way to do this is by defining

\[
\text{Signal}(x, f) \%a\% \text{ Function : Signal} \\
\text{Signal}(x, f) \%a\% \text{ Signal}(f(x))
\]

The use of the Signal constructor in the body guarantees that \(f(x) \in \{-1, 0, 1\}\).

2.2.6 Type constraints and the ellipsis argument

The ellipsis argument has special treatment in type constraints. It may seem counterintuitive for a dynamic argument to be allowed in static type constraints, but seen from the perspective of type variables it is a consistent construct. Specifying the ellipsis in a type constraint acts as a dynamic type variable so that any concrete types can be specified in the remainder of the type constraint.

Definition 2.2.17. Given argument list \(W\), \(w_k \in W\) and parameter list \(S\), \(s_k \in S\). Then \(w_k = \varepsilon\) if and only if \(T[s_k] = \varepsilon\). \(R[T[\varepsilon]] = [0, \infty)\).

(INCOMPLETE)

Example 2.2.12. The samplerange function passes unadulterated arguments to the underlying sample.int function. A suitable type constraint for samplerange looks like

\[
\text{samplerange}(x, \text{size}, \text{window}, \ldots) \%::\% \text{ a : numeric : numeric : ... : a}
\]

For greater control it is possible to assign a type to the ellipsis as though it were a vector. By appending a literal ellipsis to the end of any type will require that the values bound to the ellipsis all have the specified type.

Example 2.2.13. A portfolio is constructed based on a collection of Signals. There is no restriction to the number of Signals that can be provided, but each parameter must be a Signal. For the underlying data structure to implement a Portfolio, we’ll use a data.frame.

\[
\text{Portfolio}(\ldots) \%::\% \text{ Signal\ldots : data.frame}
\]

If the only requirement is that all parameters bound to the ellipsis must be of the same type, then a type variable appended with the ellipsis can be used.
Definition 2.2.18. Let $a$ be a type variable. If $S \can W$, then $w_k = \epsilon$ and $T[\epsilon := s] = T[a]$.

(INCOMPLETE)

Example 2.2.14. Like a Signal we can provide a Portfolio type constructor that accepts a function parameter to transform into a compatible type.

$\text{Portfolio}(f, \ldots) \%::\% \text{Function : a... : data.frame}$

2.2.7 Example: Indicators and signals

Multiple signals can be generated for a single asset class. To create a Portfolio, these Signals need to be collapsed into a single Signal per asset. The integrate_signals function has no way of knowing a priori how many Signals exist for a given asset, yet we still want type safety around the function.

$\text{integrate_signals}(\ldots, \text{fn}) \%::\% \text{Signal... : Function : Signal}$

$\text{integrate_signals}(\ldots, \text{fn=mean}) \%as\% \text{fn(do.call(c, list(\ldots)))}$

(INCOMPLETE)

2.3 Lexical scope, frames, and state

Scope describes the visibility of variables. Each language has its own rules that govern scope. R adheres to what is known as lexical, or static, scoping that allows variables higher in the call stack to be visible within a function. While this may seem to violate the idea of no side effects there are two points to consider. First in many functional languages variables are immutable and can only be assigned to once. This means that a referenced variable that is guaranteed to have a constant value. The only situation where this would fail is if the compiler or interpreter doesn’t detect that the variable has not been bound. This is the default behavior in R insomuch that variables declared out of the function scope are read-only. The second point is that referencing variables in a higher call stack is no different than currying. Functions can be defined with unbound variables. The function won’t fully resolve until applied to another function that binds those variables. The curried version of $\text{fgbm}$ in Example ?? illustrated this process. In order for a value to be produced, four distinct function applications must be performed.

2.3.1 Environments and lexical scope

An environment is a data structure that holds objects along with a pointer to a parent environment. [?] Since all non-primitive functions are closures,
TABLE 2.1: Hierarchy of environments

<table>
<thead>
<tr>
<th>R Environment</th>
<th>Lambda Calculus</th>
<th>Visible Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>.GlobalEnv</td>
<td>(\lambda x.)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>function(n)</td>
<td>(\lambda n.)</td>
<td>(n)</td>
</tr>
<tr>
<td>function(s)</td>
<td>(\lambda s.)</td>
<td>({n, s})</td>
</tr>
<tr>
<td>function(mean)</td>
<td>(\lambda mean.)</td>
<td>({n, s, mean})</td>
</tr>
<tr>
<td>function(sd)</td>
<td>(\lambda sd.X)</td>
<td>({n, s, mean, sd})</td>
</tr>
</tbody>
</table>

An environment is automatically bound to a function. This means that any variables referenced in the function that are outside the function scope are bound in its corresponding environment. Note that these references point to the same underlying object.

Lexical scope is so called because the scope is bound at compile or parse time based on the syntactic structure of the code. References to variables are governed by the lexical hierarchy of blocks within the program such that variables defined in enclosing environments are accessible to a given environment. However, siblings and descendants are not accessible. At the top of this hierarchy is the .GlobalEnv, which acts as the runtime container. Variables and functions added to this environment are accessible by functions defined within this environment. In turn, variables and functions created within one of these functions can access any variables and functions defined there.

**Example 2.3.1.** Referring again the `fgbm.curry` definition, Table 2.1 shows the relationship between the different functions that comprise the definition according to their environment hierarchy. Any variable not defined in the immediate environment will be searched for along this hierarchy. The behavior can be verified by calling `ls` on a function’s environment, such as

```r
> f <- fgbm.curry(10)(100)(0.02)
> ls(environment(f))
[1] "mean"
> ls(parent.env(environment(f)))
[1] "s"
```

If a variable with the same name is defined in a descendant environment, then it will take precedence over variables defined earlier. When assigning to a variable that is out of scope, a new variable is created within the current scope. The following two examples illustrate this behavior.

**Example 2.3.2.** The inner `x` in this example is clearly different from the `x` being passed to `map`.

```r
> f <- function(x) map(x, function(x) x^2)
```

---

12 On practice this is true, but any object is technically accessible using the reflection mechanisms available in R.
Example 2.3.3. In this example the extent of masking is more obvious in terms of the value that y resolves to.

```r
f <- function(x) {
  y <- x
  z <- map(x, function(x) { y <- -x; y^2 })
  z / y
}
> f(1:5)
[1] 1 2 3 4 5
```

As discussed in Theorem 2.1.21, the ellipsis can simplify function calls since it is equivalent to constructing a closure. The stock `apply` functions take advantage of this equivalence, yet it is not always desirable. The reason is that a closure is more explicit, which avoids potential naming collisions in argument names, in addition to incompatible parameter alignment with arguments.

Example 2.3.4. `sapply(rnorm(10), fgbm, n=100, mean=x, sd=y)`

```
sapply(rnorm(10), function(s) fgbm(100,s,x,y))
```

Properties: call by value semantics, variable assignment
Creation of closures (discuss the enclosing environment)

Exercise 2.16. A function for extracting the asset name from a `Signal` is defined as `asset.name(x,f=toupper) %as% f(x@ticker)`. Identify which objects are visible when calling `asset.name`

Exercise 2.17. Suppose this function is used inside a `Portfolio` to construct the list of assets in the portfolio.

```
Portfolio(...) %as% {
  n <- sapply(list(...), asset.name)
}
```

What is the type and value of x in f?

Exercise 2.18. Writing to a variable

### 2.3.2 Assignment to enclosing scopes

Since R is not a pure functional language it is legal to reassign the value of a variable. Local assignment is performed by the standard `<-` operator. While the standard assignment operator disallows writing to variables in external
environments, there is a separate assignment operator that allows precisely this. Assignment in an enclosing (i.e. parent) scope is done via the \texttt{<-} operator. This operator searches for the top-most declaration and writes to that. If no such variable is found, then the value is written to the global environment.

As a general rule the use of this operator is rather dangerous as it can destroy the referential transparency we have worked so hard to establish. Of course, as with any rule, there are exceptions to the rule that I discuss below.

Combined with closures, external assignment enables scoped state management. In a controlled situation like this it is possible to implement shared state that is limited in scope to an explicit and finite set of functions. This leads to simple and elegant implementations of common idioms, including counters, generators, finite state machines, and even Markov Chains.

**Example 2.3.5.** Counter example  
(INCOMPLETE)

**Example 2.3.6.** Generator example  
(INCOMPLETE)

**Example 2.3.7.** Finite state machine  
(INCOMPLETE)

**Example 2.3.8.** Markov Chain  
(INCOMPLETE)

### 2.4 Discussion: An asset trading system

To conclude this chapter we will piece together all the functions presented in this chapter to finish our simple trading system. The goal is to generate a portfolio and evaluate its performance. We will conduct a Monte Carlo simulation to understand the behavior of the strategy over numerous potential scenarios. Conceptually the program can be divided into three distinct components: one for generating signals, another for creating portfolios, and the last one to evaluate performance.

(Incomplete)
FIGURE 2.13: Constructing a Portfolio

FIGURE 2.14: Evaluating performance of a Portfolio
Picking up from the discussion at the end of the last chapter, how do we know whether these strategies are winners? For the ones that are winners, how do we know how reliable they are? Essentially we can treat each signal as a random variable, where we would like to know the expected value of the strategy along with characteristics of its distribution. If the strategy has a positive expected value and is uncorrelated with the market, then we can conclude that the strategy has alpha. We can test this hypothesis by comparing the strategy distribution using a random time series versus an actual time series. For pedagogical purposes, we’ll assume that the process driving the strategy is continuous. Computing expected value is then a matter of integrating the probability density function.

The previous chapter provided the foundation surrounding the semantics of functions. Concepts like higher-order functions and closures are key tools frequently leveraged in functional programs. Just as important are the data structures used in their application. We saw that functions in R have some properties that are uncommon in other languages. The same is true of data structures, which have some unique properties of their own. Getting to grips with the formal aspect of the core data structures will allow us to begin formally proving various functions and programs. The goal is to simplify the process of transforming mathematical ideas into code and establish a level of certainty in our programs. To this end numerous proofs are presented that seamlessly move between R syntax and standard mathematical notation. The purpose is for the reader to develop and intuition surrounding this type of reasoning and analysis.

3.1 Set and sequence mechanics

Set theory is a foundational field that provides a grammar to describe collections of things. Although elements of sets do not need to be similar, typically we think of elements of sets as having something in common. For example the set of integers, rational numbers, etc. are useful precisely because the members of each set are governed by the same underlying properties. For our purposes, sets are useful in two ways. First they provide a rigorous frame-
work for describing and manipulating collections of things. Second they form
the basis for other mathematical entities like sequences and series. Since vec-
tors are a primitive type in R, establishing a formal relationship between these
mathematical concepts and their programmatic counterparts is essential. This
conceptual bridge is what allows us to reason about our functional programs.

3.1.1 Vectors as sequences

Vectors in R can be treated as either n-tuples or sequences. At times they
will also be treated as sets insomuch that values are assumed or forced to be
unique. To distinguish between ordered n-tuples and sets, either the ⟨x⟩ or {x}
notation is used, respectively. Generally the ̸x notation is eschewed although
at times this notation will be used to clarify that an object is intended to be
treated as a vector.

Definition 3.1.1. A vector is an ordered collection of elements equivalent to
a finite sequence. A vector x with n elements is denoted as c(x₁, x₂, ..., xₙ) =
⟨x₁, x₂, ..., xₙ⟩.

(a) Elements of a vector are indexed by a subset of the positive integers
Nₙ = 1, 2, ..., n.

(b) For vector x and k ∈ Nₙ, x[k] ≡ xₖ is the k-th element of x.

(c) The length of a vector is its cardinality and is denoted length(x) ≡ |x|.

(d) xₖ ∈ T, ∀xₖ ∈ x where T ∈ T.

Definition 3.1.2. Let x be a vector. If |x| = 1 then x is a scalar, which has
the following properties:

(a) c(x) = c(x₁)

(b) x = x₁

Like a sequence, a vector is created by a function defined on Nₙ. The
simplest vectors can be created explicitly using the concatenation function,
which has already been introduced as the vector constructor.

x <- c(1, 5, 10, 10, 5, 1)

Vectors can also be created using the sequence function, which is defined as
seq(i, j, by = m) ≡ ⟨i, i + m, i + 2m, ..., j⟩, where m ∈ R. If the sequence is an
integer range, then i : j ≡ seq(i, j, 1) can be used as a shorthand notation. As
we will see later, this notation is the same as the shorthand for indicating sub
sequences.

Definition 3.1.3. Given vectors x = ⟨x₁, x₂, ..., xₘ⟩, y = ⟨y₁, y₂, ..., yₙ⟩ the
**Proof.** statements

**Proposition 3.1.4.** The definition of concatenation implies the following statements ∀ vectors x, y of the same type.

(a) Concatenation is not commutative: \( c(x, y) \neq c(y, x) \).

(b) Concatenation is associative \( c(x, c(y, z)) = c(c(x, y), z) \).

(c) \( \text{length}(c(x, y)) = \text{length}(x) + \text{length}(y) \).

(d) If a is a scalar then \( c(a, x) = \langle a, x, x, ..., x \rangle \).

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

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

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

And

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

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

(c) This is trivial based on the definition.

(d) This is also trivial based on the definition.
Given the definition of a scalar, it is obvious that concatenation takes any number of vector and scalar arguments returning a single vector.

**Definition 3.1.5.** A vector with 0 elements is called the *empty set* and is denoted as \( c() \equiv \emptyset \).

(a) The empty set is the identity for the concatenation function: \( c(c(), x) = c(x) = x \) and \( c(x, c()) = x \).

(b) \( |c()| = 0 \).

Comparison of vectors is an element-wise operation. This is true of ordering operators in addition to equality. Hence to determine whether two vectors are equal requires comparing their individual elements.

**Definition 3.1.6.** Two vectors \( x, y \) where \( |x| = n, |y| = m \) are *equal* if and only if \( x_k = y_k \forall k \in \mathbb{N}_n \).

Extracting sub sequences from a vector is a common and important operation, which is performed by the indexing operator applied to an *index vector* [18]. As such the indexing operator supports many extraction methods determined by the characteristics of the index operands. The most basic sub sequence is created by specifying an index vector as a set of integers representing the indices of the target vector.

**Definition 3.1.7.** Let \( x \) be a vector where \( |x| = n \). Let \( y \in \mathbb{N}_n \), where \( |y| = m \). A sub sequence of \( x \) is \( x[y] = (x_{y_1}, x_{y_2}, \ldots x_{y_m}) \). Given \( x = (x_k)_{k \in \mathbb{N}} \), then

(a) a sub sequence from the range \( i : j \) is \( x[i : j] = (x_k)_{k = i}^{j} \); 

(b) if \( i = j \) then \( (x_k)_{k = i}^{j} = (x_i) = x_i \).

For notational convenience, I use a shorthand notation to represent sub sequences where \( (x_k)_{k = i}^{j} = x_{i:j} \). In this notation the subscript \( k \) is assumed and may be referenced in certain expansions.

**Example 3.1.1.** Let \( x \) be a vector \( c(1, 1, 2, 3, 5, 8, 13, 21) \) and \( y = c(2, 3, 6) \). Then \( x[y] = c(1, 2, 8) \).

Sub sequences can also be created from logical vectors. This definition of the indexing operator requires use of the inclusion function, which is inspired by the Dirac delta function.

**Definition 3.1.8.** Let \( a \) be a logical scalar and \( x \) a scalar. The *inclusion function* is defined \( \delta_I(a, x) = \begin{cases} x, & \text{if } a \text{ is true} \\ \emptyset, & \text{if } a \text{ is false} \end{cases} \).
The subsequence of a vector given a logical index vector can now be defined in terms of the inclusion function.

**Definition 3.1.9.** Let \( x \) be a vector where \(|x| = n\), and let \( a \) be a logical vector such that \(|a| = |x|\). Then \( x[a] = \langle \delta_I(a_k, x_k) \rangle_{k=1}^n \).

**Proposition 3.1.10.** 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 \( \delta_I \), \(|\delta_I| = \begin{cases} 1, & \text{if } a \text{ is true} \\ 0, & \text{if } a \text{ is false} \end{cases} \). The lower bound on \(|x[a]|\) occurs when \( a_k = \text{false} \ \forall k \in \mathbb{N}_n \). Then \( x[a] = \emptyset \) and \(|x[a]| = 0\). Similarly the upper bound is reached when \( a_k = \text{true} \ \forall k \in \mathbb{N}_n \) and \( x[a] = x \) and \(|x[a]| = n\).

Having defined the inclusion function it is now possible to define set comprehensions, also known as set-builder notation, using this function. Informally set comprehensions are a syntactic construct for defining sets from other sets. The definition of a set comprehension also requires the introduction of predicates.

**Definition 3.1.11.** A predicate is a function \( P : X \to \{true, false\} \), where \( X \) is an arbitrary domain.

Predicate functions act like typical logical expressions and can be operated on by the usual logical operators. For our purposes, the logical connectives are negation \((\neg)\), conjunction \((\land)\), and disjunction \((\lor)\). Collectively we define \( \Omega = \{\neg, \land, \lor\} \) and a sequence of repeated application of operators in \( \Omega \) as \( \Omega_k \).

**Definition 3.1.12.** Let \( x \) be a vector. A set comprehension is a subset of \( x \) satisfying an arbitrary number of predicates \( P(x) \) denoted \( \{x|\Omega P_k(x)\} \equiv x[\Omega \delta_I(P_k(x), x)] \).

**Example 3.1.2.** Our crossover signal generates values in the set \( \{-1, 0, 1\} \). Suppose we generate 100 signals using a random time series. If \( x \) is a vector of signals, we can find all the long and short positions via \( x[x==1 \mid x==1] \).

The mapping between mathematical logic operators and their programmatic counterpart is in Table 3.1. In terms of data this sort of set comprehension is useful for finding extreme values or values within a range. Since order is preserved this notation is equally valid for defining sub sequences in addition to subsets. Under this situation it is written as \( \langle x|\Omega_k P_k(x)\rangle \).

**Definition 3.1.13.** A negative index on a vector \( x \) is equivalent to set negation and is denoted \( x[-i] = \langle x_k|k \neq i\rangle_{k=1}^n \) where index order is preserved. In other words \( x[-i] = c(x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n) \).
TABLE 3.1: Mapping of logical operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Logic</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td>¬</td>
<td>!</td>
</tr>
<tr>
<td>conjunction</td>
<td>∧</td>
<td>&amp;</td>
</tr>
<tr>
<td>disjunction</td>
<td>∨</td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>( \bigwedge_k )</td>
<td>all() —</td>
</tr>
<tr>
<td>any</td>
<td>( \bigvee_k )</td>
<td>any() —</td>
</tr>
</tbody>
</table>

The fact that negative indices preserve vector order is used extensively when iteratively manipulating a sequence. In set terms, a negative index acts as the relative complement of \( x_i \) in \( x \), or \( x[-i] = \{ x_k \}_{k=0}^i \setminus \{ x_i \} \). Due to the duality of scalars and vectors of unit length, it is easy to show that this definition extends to any vectors in \(-\mathbb{N}\).

**Proposition 3.1.14.** 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.

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

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

**Example 3.1.3.** Suppose \( x = \langle -1, 0, 0, NA, 0, -1, -1, 1 \rangle \). Identifying the NA values can be done via \( x[!is.na(x)] \).

**Exercise 3.1.** 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 3.2.** Given \( x \) as defined in Exercise 3.1, \( P(x) = \begin{cases} true, & \text{if } x = 0 \\ false, & \text{otherwise} \end{cases} \), reduce the set comprehension \( \langle x | P(x) \rangle \) mathematically.

**Exercise 3.3.** Show \( x[a] \), where \( a \subset -\mathbb{N}_k, |x| = k \).

### 3.1.2 Lists as sets

While lists are ordered like a sequence, they are more general than vectors. Lists are more like sets in that they may contain any type of element. Vectors are also restricted to primitive types, but lists can hold any data structure including functions, lists, and other complex data structures\(^1\).

\(^1\)This generality also forms the basis for data.frames.
Definition 3.1.15. A list is an ordered collection of elements equivalent to a finite set. A list $x$ with $n$ elements is denoted as $\text{list}(x_1, x_2, ..., x_n) \equiv (x_1, x_2, ..., x_n)$.

(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 $\text{length}(x) \equiv |x|$.

Concatenation for lists work the same as for vectors with the added distinction of the two indexing operators. Hence elements can be added to lists by using $c()$ in addition to the list type constructor. These two operations are not equivalent though, particularly when mixing types. When creating a vector where one of the elements is a list, then the result is a list.

Definition 3.1.16. Let $y = \{y_1, y_2, ..., y_n\}$, $\forall y_k \in y$, $\langle y_k \rangle = \text{list}(y)$ if $\exists y_k \in \text{list}$.

Note that only under this circumstance is the concatenation operator equivalent to the list type constructor: $c(y_1, y_2, ..., y_n) = \text{list}(y_1, y_2, ..., y_n)$.

Proposition 3.1.17. Let $x$ be a vector and $a$ a scalar of the same type. Then

(a) $c(x[k], a) = c(x_k, a)$

(b) $c(x[k], a) = \text{list}(x_k, a)$

Proof.  

(a) By definition $x[k] = x_k$ is a vector. Hence $c(x[k], a) = c(x_k, a)$.

(b) $x[k] = \text{list}(x_k)$. Hence $c(x[k], a) = c(\text{list}(x_k), a) = \text{list}(x_k, a)$ by axiom 3.1.16.

Definition 3.1.18. Let $x$ be a list where $|x| = n$ and $a$ be an integer vector with $|a| = k, k \leq n$. Then

(a) $x[a] = \text{list}(x_{a_1}, x_{a_2}, ..., x_{a_k})$

(b) $x[a] = c(x_{a_1}, x_{a_2}, ..., x_{a_k})$

Definition 3.1.19. Given a list $x$ and $i, j \in \mathbb{N}_n$, a sub sequence of list $x$ from $i$ to $j$ inclusive is $x[i : j] \equiv \langle x_k \rangle_{k=i}^j$, where $i \leq j$.

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

(b) $x[i : j] \equiv \langle x_k \rangle_{k=i}^j : \text{vector}$
Note that the major difference between lists and vectors is the fact that lists make a distinction between indexing to retrieve an element from the list and to retrieve a sub-list.

**Exercise 3.4.** Under what conditions does vector collapse fail? In other words, when does \( x[[a]] \neq c(x_{a1}, x_{a2}, \ldots) \)?

**Exercise 3.5.** Given \( x = (3, 4, 5) \), \( y = \text{list}(8, 30) \), what is \( c(x, y) \)?

### 3.1.3 Set operations

Set membership can be tested using the infix operator \( \%\in\% \). The \( \%\in\% \) operator is defined for both vector indices as well as for values. Whenever the values are integers then the operation is applied to indices.

**Definition 3.1.20.** Given vector \( x \) and variable \( a \), \( a \%\in\% x \) is true if and only if \( \exists x_k \in x \) such that \( x_k = a \).

**Definition 3.1.21.** The union of two vectors \( x, y \) is denoted as \( \text{union}(x, y) \equiv \text{unique}(x) \cup \text{unique}(y) \).

Since sets do not allow duplicate values, the union function discards any duplicate values in the vector arguments. Typical applications use concatenation to emulate set union. The difference is that concatenation is order-preserving and does not require uniqueness of values. Set intersection is defined similar to union.

**Definition 3.1.22.** The intersection of two vectors \( x, y \) is denoted as \( \text{intersect}(x, y) \equiv \text{unique}(x) \cap \text{unique}(y) \).

### 3.2 Mechanics of vectorization

Vectorization yields a number of interesting properties, such as the fact that scalars are exactly vectors of length one. Functions that are vectorized operate across elements for each vector. In other languages this normally occurs via a zip function that creates a list of tuples from a set of vectors [19], [1]. The concept of recycling plays an important role in vectorization as it normalizes vectors so that they have compatible lengths. This transformation can have interesting implications, which we will explore a bit in this section. There are also two higher order functions that will be discussed from the perspective of vectorization: map and fold, which were introduced in Chapter 1. In general these functions play a significant role in functional programming languages. Maps are prominent in abstract algebra and set theory and can also be used to generate sequences. Fold has its own counterparts and can be used to construct
sequences, series, and even certain classes of operators. With vectors as a primitive type, many R functions implicitly perform a map or fold operation on their argument. When a function behaves as though it were called via map or fold we call the function map-vectorized or fold-vectorized, respectively. Table 3.2 highlights some common functions whose behavior can be classified as being either map-vectorized or fold-vectorized. In fact, any function can be vectorized if it satisfies certain conditions. In layman’s terms, any vector arguments to a function must have compatible lengths. The output of the function must also have a length equal to the longest vector argument.

**Definition 3.2.1.** Given a function $\lambda W.X$ with arguments $W = \langle w_1, w_2, ..., w_n \rangle$. A function is vectorized if the following conditions are satisfied.

(a) $\exists p \neq 1 \in \mathbb{N}$ such that $|w_k| = p$ for some $w_k : \text{vector}$ and $k \in \mathbb{N}_n$

(b) $|w_k|$ divides $p \forall w_k : \text{vector}$

(c) $|(\lambda W.X)[W := S]| = p$

**Example 3.2.1.** Based on this definition, it is possible to show that the ifelse function is vectorized. Suppose we want to cap a series of returns to six standard deviations.

ifelse(abs(x) > 6 * sd(x), sign(x) * 6 * sd(x), x)

This function takes three vector arguments, where the first vector is of logical type. The length of each vector is $|x|$, so $p = |x|$ and $|x|$ clearly divides $p$. The output vector also has length $|x| = p$ so ifelse is vectorized.

For a formal proof, we would need to examine the implementation of ifelse assuming it has a functional implementation. Otherwise as we will see it is necessary to perform a Monte Carlo analysis to set an expectation regarding its behavior.
Exercise 3.6. Show whether \texttt{fgbm} is vectorized.

Exercise 3.7. Show whether \texttt{quantize} is vectorized.

Exercise 3.8. Show whether \texttt{sprintf} is vectorized.

3.2.1 Recycling of vector arguments

Recycling is an important property of vectorization, where elements of a vector are reused in an operation when the length of the operands are whole number multiples of each other \cite{citation}. Hence when two operands have differing lengths, the values in the shorter vector will be recycled until the length is the same as the longer vector. Via recycling the power of vectorization is greatly enhanced.

**Axiom 3.2.2.** Let $f : X^m \times Y^n \rightarrow Z^p$. If $m \geq n$ and $n$ divides $m$, then $y \rightarrow \text{rep}(y, \frac{m}{n})$ and $p = m$. Similarly if $n > m$ and $m$ divides $n$, then $x \rightarrow \text{rep}(x, \frac{n}{m})$ and $p = n$.

Recycling is another one of those concepts that initially seems strange but is actually fairly common in mathematics. For example the rules of linear algebra utilize the principle of recycling when an operation has both scalars and vectors as operands.

$$
\begin{bmatrix}
a + x \\
a + y \\
a + z
\end{bmatrix}
$$

Most arithmetic and algebraic operators support recycling 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

```r
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 $m$.

```r
m[,2] <- 0
```

We can now reason more formally about the vector length of the \texttt{ifelse} example. The first argument \texttt{abs} is map-vectorized so $|abs(x)| = |x|$. On the other side of the comparison operator, $|6| = 1$ and $|sd(x)| = 1$. Hence $|6sd(x)| = 1$.

Exercise 3.9. Let $x = \text{rnorm}(10)$. Determine the length of the operation $(x - \bar{x})^2$.

Exercise 3.10. Show that pattern matching is not vectorized.

Exercise 3.11. Determine the length of \texttt{fgbm}(50, 20, 0.03, 0.05).
3.2.2 The map function

From a set perspective, map provides the computational machinery to perform a mapping from one domain to another, or map : X^n → Y^n. \(^2\) In practice, a subset of a domain represented as a vector is transformed. The difference between a map operation and a conventional function call is that this subset is transformed in a single function call. This happens because map is a higher-order function that takes a function f : X → Y as an argument and applying f to each element in the sequence. Many built-in R functions are map-vectorized.

Based on the definition of map, we can prove that its behavior is constant over any arbitrary function and vector input.

Theorem 3.2.3. Let f : X → Y. The function map operating on f and vector x, |x| = n is map(f, x) = \(\langle f(x_1), f(x_2), ..., f(x_n)\rangle\) ∀x ∈ X^n.

Proof. The proof is done in two parts. First we show via induction that each iteration of the function has the same behavior. Second we show that the operation halts.

We begin by transforming the programmatic definition into hybrid mathematical notation.

\[
\text{map}(f, x, y) = \begin{cases} 
  y, & \text{if } x = \emptyset \\
  \text{map}(f, x[-1], c(y, f(x[1]))) & \text{otherwise}
\end{cases}
\]

When x ≠ ∅, map(f, x, y) = map(f, x[-1], c(y, f(x[1]))) . We want to show that for some intermediate case that map(f, x_{k:n}, y_{1:k−1}) = map(f, x_{k+1:n}, y_{1:k}), where y_{i:j} = \(\langle f(x_i), ..., f(x_j)\rangle\). The case for k = 1 is below.

\[
\text{map}(f, x_{1:n}, y = c()) = \text{map}(f, x[-1], c(y, f(x[1]))) = \text{map}(f, x_{2:n}, c(c(), x_1)) = \text{map}(f, x_{2:n}, x_{1:1})
\]

We assume that the case for k is true and show that it holds for k + 1.

\[
\text{map}(f, x_{k+1:n}, x_{1:(k+1)-1}) = \text{map}(f, x[-1], c(x_{1:(k+1)-1}, f(x[1]))) = \text{map}(f, x_{k+2:n}, c(x_{1:k}, x_{k+1})) = \text{map}(f, x_{(k+1)+1:n}, x_{1:k+1})
\]

The second half of the proof simply requires showing that the map function terminates. Initially x = x_{1:n} and |x| = n. In general for step k, x = x_{k:n} and |x| = n − k + 1. Suppose k = n. Then x = x_n and the next iteration will yield x_n[-1] = c(x_n)[-1] = c().

Example 3.2.2. Quantize uses map in its implementation. Based on our understanding of map, we know that the first line will generate a value in R^n.

\(^2\)In R the suite of apply functions serves the same purpose, but for conceptual clarity I refer to map for these operations. In Part II of the book, the discussion will transition to using these built-in functions to take advantage of their optimized execution.
Note that in the definition for \textit{map} there are only two arguments, while the proof shows three arguments. This notational sleight of hand exists because a typed accumulator must be provided to the R implementation of \textit{map}, which safely defaults to \textit{c()}. However, the implementation is sufficiently general that a \textit{list} could be used as well.

Higher order functions like the difference operator, differential, and the integral are linear operators. This property can yield simplifications and other transformations to aid in solving a problem. Linearity is not limited to the above functions and is also a property of \textit{map}.

\textbf{Theorem 3.2.4.} The \textit{map} function preserves linearity. Let vector $x \in X^n$ and $a \in X$. If $f$ and $g$ are linear, then the function \textit{map} is linear.

(a) $\text{map}(y, f + g) = \text{map}(y, f) + \text{map}(y, g)$

(b) $\text{map}(y, af) = a \text{map}(y, f)$

\textbf{Proof.} The proofs are straightforward and begin by expanding simplified notation into the lambda calculus.

(a) $\text{map}(y, f + g) = \text{map}(y, \lambda x. f(x) + g(x))$

$\quad = \langle f(y_k) + g(y_k) \rangle_{k \in \mathbb{N}^n}$

$\quad = \langle f(y_k) \rangle_{k \in \mathbb{N}^n} + \langle g(y_k) \rangle_{k \in \mathbb{N}^n}$

$\quad = \text{map}(y, f) + \text{map}(y, g)$

(b) $\text{map}(y, af) = \text{map}(y, \lambda x. a * f(x))$

$\quad = \langle a f(y_k) \rangle_{k \in \mathbb{N}^n}$

$\quad = a \langle f(y_k) \rangle_{k \in \mathbb{N}^n}$

$\quad = a \text{map}(y, f)$

\textbf{Example 3.2.3.} Consider a function that adds the quantized and confined version of a vector together. Then $\text{map}(y, \lambda x. \text{confine}(x) + \text{quantize}(x)) = \text{map}(y, \text{confine}) + \text{map}(y, \text{quantize})$

Since vectors and list-like structures are prevalent in R it convenient to define the concept of map-equivalence as well, which provides a notational shorthand for functions. Map-equivalence follows directly from the definition of \textit{map} and allows us to denote any non-vectorized single argument function as a vectorized function.
Definition 3.2.5. Let \( f : X \to Y \) be a function. Then the map-equivalent function of \( f \) is \( f_{\text{map}} : X^n \to Y^n = \text{map}(f, x) \), where \( x \in X^n \).

In order to use this definition we need to guarantee that the simplification is consistent across multiple applications of the same function. In other words the function \( f \) must be well-defined \( \forall x \in X \).

Theorem 3.2.6. Every non-vectorized single argument function has exactly one map-equivalent counterpart.

Proof. We use a proof by contradiction to show this is true. Let \( f : X \to Y \). Suppose \( f \) has two distinct map-equivalent representations \( f_{\text{map}_1} \) and \( f_{\text{map}_2} \) such that \( f_{\text{map}_1} \neq f_{\text{map}_2} \). Then \( f_{\text{map}_1} = (f(x_1), f(x_2), \ldots, f(x_n)) \) but \( f_{\text{map}_2} = (f(x_1), f(x_2), \ldots, f(x_n)) = f_{\text{map}_1} \), which contradicts the original assumption. Hence \( f \) has exactly one map-equivalent representation.

One seeming limitation to \text{map} is its requirement that its function argument must be a function of a single parameter. Obviously functions exist with more than one argument, so it would be nice to extend \text{map} to support these functions. Rather than defining a new \text{map} function, it is far simpler to use a closure that reduces the argument count to one. The following theorem shows that it is possible to construct such a function where the return value of the function is a single-argument closure compatible with \text{map}.

Theorem 3.2.7. Given a function \( M \) with arguments \( W = (w_1, w_2, \ldots, w_n) \) and \( \text{FV}(M) = 0 \), \( \exists V. \lambda W. \lambda W - V.M \) where \( V \subset W \) and \( |V| = |W| - 1 \).

Proof. Let \( M = \lambda W.X \), where \( X \in \lambda \). Choose some \( w_k \in W \). Now define \( \lambda w_k.M \), such that \( \text{FV}(\lambda w_k.M) = W - \{w_k\} \). Let \( V = W - \{w_k\} = \{w_1, w_2, \ldots, w_{k-1}\} \cup \{w_{k+1}, \ldots, w_n\} \). By definition \( V \subset W \) and \( |V| = |W| - 1 \). Finally define \( \lambda V. \lambda w_k.M = \lambda V. \lambda W - V.M \).

This theorem shows that \text{map} can be applied to any arbitrary function by using an intermediate closure to conform the argument list. However, \text{map} can only vectorize a single dimension, which means that all other variables are effectively held constant. Depending on the operations within the function it may or may be a limitation. This can have the effect of vectorizing variables unintentionally.

Example 3.2.4. Suppose you are calculating the \( \chi^2 \) for a number of sample sets. Here the expected value is the same while the actual value varies. Calculating the \( \chi^2 \) for each set can be performed via this transformation.

\[
\begin{align*}
f & \leftarrow \text{function}(y) \text{ function}(x) \chi.\text{sq}(x, y) \\
\text{map}(xs, f(ys))
\end{align*}
\]

For each sample set, the same function \( f(ys) \) can be used, which can simplify the operation by ensuring the value of \( ys \) is held constant across each error calculation.

Under real-world situations it is more common to use a 2-dimensional version of \text{map} that operates on a set of variables per iteration.
map(function(xy) chi.sq(xy[1], xy[2]), xys)

Knowing whether a function is vectorized or not is important in reasoning about a program. The idea that a function is map-vectorized implies that certain properties hold for these functions.

**Definition 3.2.8.** Given \( x = (x_1, x_2, \ldots, x_n) \), a function \( f \) is *map-vectorized* if the following relationship holds:

\[
    f(x) = (f(x_1), f(x_2), \ldots, f(x_n)).
\]

Comparing the definition of map-vectorization with map-equivalence, we see that the map-equivalent function of a scalar function is map-vectorized. The direct way of showing that a function is map-vectorized is by analyzing the function definition. With native or built-in functions this can be buried within the source code of R itself. Instead structural induction can be used to prove the equivalence between the two forms of the function. The basic approach is to show the basis case is equivalent and then show the equivalence for the recursive step [8]. Let’s use the \( \log \) function as a case study for how this works.

**Proposition 3.2.9.** The function \( \log \) is map-vectorized.

**Proof.** Let \( a \in \mathbb{R}_+ \). Then \( \log(a) = (\log(a)) \) by scalar-vector equivalence. Assume \( \log(x) = (\log(x_1), \log(x_2), \ldots, \log(x_n)) \). Show that \( \log(c(x, a)) = (\log(x_1), \log(x_2), \ldots, \log(x_n), \log(a)) \). Define \( f_{\log}(a) = \log(a) \).

\[
    \log(c(x, a)) = \text{map}(f_{\log}, c(x, a))
    \rightarrow \text{map}(f_{\log}, a, c(f_{\log}(x_1), f_{\log}(x_2), \ldots, f_{\log}(x_n))
    \rightarrow \text{map}(f_{\log}, c(), c(c(f_{\log}(x_1), f_{\log}(x_2), \ldots, f_{\log}(x_n)), f_{\log}(a))
    \rightarrow c(f_{\log}(x_1), f_{\log}(x_2), \ldots, f_{\log}(x_n), f_{\log}(a))
    = (\log(x_1), \log(x_2), \ldots, \log(x_n), \log(a))
\]

\( \square \)

**Proposition 3.2.10.** Any polynomial function is map-vectorized.

**Proof.** (INCOMPLETE)

This theorem tells us that applying a polynomial function to a vector of length \( m \) will always yield a value of length \( m \). Collectively these theorems give us a framework for preserving the properties of vectorization in our functions.

### 3.2.3 The fold function

The behavior of \( \texttt{fold} \) is rooted in combinatory operators. In this model the operator is transformed from \( f : X \times X \rightarrow X \) to behave like \( f : X^n \rightarrow X \). This curious transformation is commonplace in mathematics, although the semantics are usually hidden in notation. For example the summation operator...
\(\sum^n\) provides this transformation for addition while the product operator \(\prod^n\) does the same for multiplication. Hence \textit{fold} is an inductive process that successively applies a binary operation to a sequence of elements such that the result of the previous operation becomes an operand in the current operation.

**Theorem 3.2.11.** Let \(f : X \times X \rightarrow X\) and \(x \in X^n\). Then \(\text{fold}(f, x, I_f) = f(x_n, f(x_{n-1}, ..., f(x_1, I_f)))\), where \(I_f\) is the identity over \(f\).

\textit{Proof.} The implementation of \textit{fold} can be written in hybrid syntax as we did with \textit{map}.

\[
\text{fold}(f, x, y) = \begin{cases} 
  y, & \text{if } x = \emptyset \\
  \text{fold}(f, x[-1], f(x[[1]], y)), & \text{otherwise}
\end{cases}
\]

Following the same general approach, we prove this in two parts. Assume the case for \(x_k\) is true. Then \(\text{fold}(f, x_{k:n}, y_k) = \text{fold}(f, x_{k+1:n}, f(x_k, y_k))\). We now show that this holds for \(x_{k+1}\).

\[
\text{fold}(f, x_{(k+1):n}, y_{k+1}) = \text{fold}(f, x[-1], f(x[[1]], y_{k+1})) \\
= \text{fold}(f, x_{(k+1)+1:n}, f(x_{k+1}, y_{k+1}))
\]

The second half of the proof shows that \textit{fold} halts. Assume we are at the penultimate step.

\[
\text{fold}(f, x{n:n}, y_{n-1}) = \text{fold}(f, x[-1], f(x[[1]], y_{n-1})) \\
= \text{fold}(f, c(), f(x, y_{n-1})) \\
\rightarrow y_n
\]

\(\square\)

This definition of \textit{fold} is actually a specific case of a more general view of \textit{fold}. The broader view is that \textit{fold} takes a sequence and aggregates it to some arbitrary data structure. Here the operation is \(f : X^n \rightarrow Y\). In practice this version is useful for solving certain types of problems but it is difficult to say anything concrete about the form. Hence unless otherwise specified \textit{fold} will be used in the manner first discussed.

**Corollary 3.2.12.** \(\text{fold}(f, x, I_f) = \text{fold}(f, x[-1], x[[1]])\)

\textit{Proof.}

\[
\text{fold}(f, x, I_f) = \text{fold}(f, x[-1], f(x[[1]], I_f)) \\
= \text{fold}(f, x[-1], x[[1]])
\]

\(\square\)

Similar to \textit{map}, functions can have a fold-equivalent representation. For a function to be fold-equivalent it must satisfy more restrictions than for map-equivalence.
Definition 3.2.13. Let \( f : X \times X \to X \) be a function. Then the fold-equivalent function of \( f \) is \( f_{\text{fold},I_f} : X^n \to X = \text{fold}(f, x, I_f) \), where \( x \in X^n \).

As mentioned at the beginning of the section, the sum and product operators can be generalized as applications of \texttt{fold}. In this model the addition and multiplication operators are expressed as functions using the principle of function-operator equivalence.

\[
\text{plus}(x, y) \%as\% \{ x + y \}
\]

Using a functional definition of addition allows us to define summation as an application of \texttt{fold}. Hence the fold-equivalent counterpart to \texttt{plus} is the summation operator \( \Sigma_n \). In standard R this is the \texttt{sum} function, to be equivalent to its \texttt{fold} counterpart. To do so we must take as an axiom that \( \text{sum}(x) = \Sigma^n x \).

Proposition 3.2.14. \( \text{sum}(x) = \text{fold}(\text{plus}, x, 0) \)

Proof.

\[
\text{fold}(\text{plus}, x, 0) = \text{plus}(x_n, \text{plus}(x_{n-1}, \ldots, \text{plus}(x_2, x_1)))
\]

\[
= x_n + (x_{n-1} + \ldots + (x_2 + x_1))
\]

As with \texttt{map}, fold-equivalence gives us formal semantics to transform binary functions into a function that operates over a sequence of values. While these properties hold for functions defined over a fixed domain, it is important that \texttt{fold} can yield other interesting results outside of this constraint.

Definition 3.2.15. A function is fold-vectorized if the following relationship holds.

\( f(x) = f(x_n, \ldots, f(x_2, f(x_1))) \)

Example 3.2.5. The addition operator can be used directly in a fold procedure.

\[
x_n + (x_{n-1} + (x_{n-2} + \ldots + (x_2 + x_1))
\]

which generalizes to any arbitrary operator \( \circ \) as

\[
x_n \circ (x_{n-1} \circ (x_{n-2} \circ \ldots \circ (x_2 \circ x_1))
\]

Using function syntax this is equivalent to

\[
f(x_n, f(x_{n-1}, f(x_{n-2}, \ldots, f(x_2, x_1)))))
\]

Again with induction we can show that \texttt{fold} operates correctly \( \forall f \in \text{binary operator} \) and \( x \in X^n, n \in \mathbb{N} \).

Proposition 3.2.16. Next let’s examine the \texttt{min} function. We do not know how \texttt{min} is implemented, but we do know its documented behavior. The base case is when \( |x| = 2 \). We want to show that the function is fold-vectorized, in other words \( \text{min}(x) = \text{fold}(f_{\text{min}}, x) \). This is a function that has no corresponding identity element, which implies that the second form of \texttt{fold} must be used.

The function \texttt{min} is fold-vectorized: \( \text{min}(x) = \text{fold}(f_{\text{min}}, x) \).

\[3\]Technically all operators in R are functions, so this is unnecessary. For clarity of argument, it is useful to make this distinction.
Formal Considerations of Data Structures

Proof. Define $f_{\min}(a, b) = \min(a, b)$, where $a, b$ are scalars. When $x \in \mathbb{N}_2$,

\[
\begin{align*}
    \text{fold}(f_{\min}, x) &= \text{fold}(f_{\min}, x[-1], x[[1]]) \\
    &= \text{fold}(f_{\min}, x_2, x_1) \\
    &\rightarrow \text{fold}(f_{\min}, x[-1], \text{fold}(x_1, x[[1]])) \\
    &= \text{fold}(f_{\min}, c(), \text{fold}(x_1, x_2)) \\
    &\rightarrow \text{fold}(x_1, x_2) \\
    &= \min(x_1, x_2)
\end{align*}
\]

Assume that for $x \in \mathbb{N}_n$, $n > 2$, $\min(x) = \text{fold}(f_{\min}, x)$. Show that this holds for $x \in \mathbb{N}_{n+1}$.

Let $a$ be a scalar. Then $c(a, x) \in \mathbb{N}_{n+1}$.

\[
\begin{align*}
    &= \text{fold}(f_{\min}, c(a, x)) \\
    &= \text{fold}(f_{\min}, x, a) \\
    &= \text{fold}(f_{\min}, x[[1]], \ldots) \\
    &= \min(c(a, x))
\end{align*}
\]

\[
\text{Proposition 3.2.17. The factorial function } \text{fac} \text{ is map-vectorized. In other words } \text{fac}(x) = (\text{fac}(x_1), \text{fac}(x_2), \ldots, \text{fac}(x_n)).
\]

Proof. Since $\text{fac}$ uses $\text{map}$ in its implementation then $\text{fac}$ is vectorized if the function passed to $\text{map}$ is well-defined. We also have to show that the length of the result of $\text{map}$ is equal to the length of $x$. The lambda expression to $\text{map}$ calls $\text{fold}$, so a similar process must be followed here. By Definition 3.2.13 we know that the application of $\text{fold}$ to a function $f$ is equivalent to a fold-vectorized function. The strategy here is to work inside out to transform the function into an equivalent mathematical expression.

\[
\begin{align*}
    \text{fac}(x) &= \text{map}(\text{function}(x_i) \text{fold}(\text{product}, 1 : x_i, 1), x) \\
    &= \text{map}(\lambda x_i. (x_i * (x_{i-1} * \ldots * (2 * 1))), x) \\
    &= \text{map}(\lambda x_i. x_i! , x) \\
    &= (x_1!, x_2!, \ldots, x_n!)
\end{align*}
\]

\[
\text{Example 3.2.6. For native R functions this process is much simpler. As an example for } \text{map} \text{ this is the proof for the factorial function } \text{fac} \text{ defined in Figure 3.1.}
\]
fac(0) %as% 1
fac(x) %as% map(function(xn) fold(product, 1:xn, 1), x)

Exercise 3.12. Define the map-equivalent form of `confine`.

Exercise 3.13. Write a version of `mean` using `fold`. What structure does the accumulator have?

### 3.2.4 Monte Carlo as a proxy for induction

If the behavior of a function is not well understood then a weaker alternative approach can be used as a last resort. In this situation a Monte Carlo simulation can be used to show that the algorithm behaves correctly within a given margin of error [15]. This approach requires choosing an input distribution that is consistent with the domain of the function being tested. With this technique the base case is proven using the same approach. However rather than proving absolutely that an equivalence holds for the inductive step it is possible to show that this step is correct over a finite number of iterations.

As an example let’s take another look at the `ifelse` function. The definition of this function uses conditional expressions, which cannot be reasoned about easily. To show that `ifelse` is vectorized we can run a Monte Carlo simulation to show that the induction step holds regardless of the length `n` of the input vector. In essence we are making the generality of the induction hypothesis concrete and attempting to satisfy it over as broad a range of `n` as is practical. For each trial the goal is to show that given `n`, `f(c(xs, a)) = c(f(xs), f(a))` for any vector `xs` and scalar `a`. To verify that the induction test is correct and non-random, we can construct a binomial test where the null hypothesis is that the result is deterministic, in other words the probability of a correct value is 1. Doing so gives us the following result:

```r
> x <- o[,2]
> s <- length(x[x])
> f <- if(is.empty(x[!x])) { 0 } else { length(x[!x]) }
> binom.test(c(s,f), p=1)

Exact binomial test

data:  c(s, f)
number of successes = 1000, number of trials = 1000, p-value = TRUE
alternative hypothesis: true probability of success is not equal to 1
95 percent confidence interval:
  0.9963179 1.0000000
sample estimates:
  probability of success
```

Formal Considerations of Data Structures

FIGURE 3.2: Using Monte Carlo to 'prove' the induction step

```r
mc.induction(f, tries=1000, rng=rnorm) %as% {
  fn <- function(junk) {
    n <- round(runif(1, min=2, max=1000000))
    xs <- rng(n)
    a <- rng(1)
    pass <- all(f(c(xs, a)) == c(f(xs), f(a)))
    c(n=n, success=pass)
  }
  t(sapply(1:tries, fn))
}

o <- mc.induction(function(x) ifelse(x > 0, x, abs(x)))
```

Obviously with this approach the induction hypothesis cannot be proven in absolute terms. Whether this is acceptable depends on the application.

In general this approach can be used for testing functionality as well. The key is to determine what the behavior would be if the process were random and use the simulation to reject the null hypothesis.

Exercise 3.14. Show that \( \min \) is fold-vectorized using the Monte Carlo approach.

Exercise 3.15. Create a Monte Carlo simulation for \( \log \) to show that it is vectorized. What is an appropriate distribution to use?

### 3.2.5 Transitive aspects of vectorization

As functions are built from smaller functions it is useful to know when vectorization is preserved. When a property is preserved over function composition, then the function is compositionally transitive.

**Theorem 3.2.18.** If \( a, b \) are map-vectorized, then \( f(x) = a(b(x)) \) is map-vectorized.

**Proof.** This proof basically follows from the definition.

\[
\begin{align*}
f(x) &= a(b(x)) \\
&= a(b(x_1), b(x_2), ..., b(x_n)) \\
&= (a(b(x_1)), a(b(x_2)), ..., a(b(x_n))) \\
&= (f(x_1), f(x_2), ..., f(x_n))
\end{align*}
\]

\( \square \)
FIGURE 3.3: Distribution of trials in Monte Carlo simulation
Vectorization is also preserved over Currying. This means that vectorization is transitive over chained function application.

**Theorem 3.2.19.** If \( f(x, y) \) is vectorized then \( f(x)(y) \) is vectorized.

**Proof.** This follows from the property of Currying. Given \( \lambda W.X \) vectorized, where \( W \) is an argument list and \( X \in \Lambda \) with no free variables. Let \( p = \max|w_k| \). Then \( \forall w_k \in W, |w_k| \) divides \( p \). Currying tells us that \( \lambda W.X = \lambda w_1.\lambda w_2....X \). Since no additional terms of computation are introduced we know that \( w_k \) is constant in the function application. \( \square \)

In general we treat variables within function bodies as immutable such that these properties hold. Vectorization is transitive over partial application as well. Note that the same cannot be said generally of closures. The reason is that closures can introduce arbitrary expressions that can change a computation while the rules of construction for partial application and Currying are strict.

**Theorem 3.2.20.** Given function \( \lambda W.X \), \( W \) an argument list and \( X \in \Lambda \). Let \( \lambda V.\lambda U.X \) be the partial application of \( \lambda W.X \). If \( \lambda W.X \) is vectorized then \( \lambda V.\lambda U.X \) is vectorized.

**Proof.** Let \( p = \max|w_k| \). Since \( \lambda W.X \) is vectorized we know that \( |w_k| \) divides \( p, \forall w_k \in W \). Since \( V \subseteq W \), \( |v_k| \) divides \( p \). Similarly since \( U \subseteq W \), \( |u_k| \) divides \( p \). Since the body \( X \) is unchanged we know that \( |\lambda V.\lambda U.X| = p \). Therefore \( \lambda V.\lambda U.X \) is vectorized. \( \square \)

**Exercise 3.16.** Use Theorem 3.2.19 to show that \( \text{map}(\text{crossover}(x), \text{Signal}) \) is vectorized.

**Exercise 3.17.** Given \( f(x, \lambda) = (\sum_{i=1}^{n} x_i) \log \lambda - n \lambda \) where \( |x| > 1 \), what is \( |f(x, \lambda)| \)?

### 3.3 Matrices

Like their mathematical counterparts, matrices are vectors in two dimensions. Matrices have numerous type constructors depending on the type of the inputs. The interface for creating a matrix is the same for any type of number, whether integer, real, complex. For example, constructing a \( 3 \times 4 \) matrix is done like

\[
A = \begin{bmatrix}
-0.3865880 & 2.36008580 & -0.03091161 & 0.1965394 \\
-0.3453409 & 0.05575077 & -0.32740135 & 2.4701649 \\
-1.1195847 & 0.54923611 & -0.29193250 & -0.8851350
\end{bmatrix}.
\]

The rules of construction are visually represented in the notation but are also explicitly stated insomuch that each row (column) must have the same
number of elements. Programatically this means that the number of rows times the number of columns must equal the total number of input elements, which is how a matrix is created in R. When the input types are vectors the notation looks more or less the same: \( A = [x\, y\, z] \) where \( x, y, z \) are vectors of the same length. Matrices can even be constructed from other matrices given that the matrices conform to certain rules governing their dimensions.

In programming terms we’ve identified three distinct type constructors each with their own set of rules that describe how to create a matrix type. Mathematical types and their constructors are not limited to linear algebra and can be found in numerous branches of mathematics. Similarly the analytical system consists of numerous types, where the efficacy of the system is partially determined by the choice of types. Regarding matrix constructors, it is telling that they do not mix and match the input types. For example there is no constructor that creates a matrix from a sequence of numbers and a set of vectors. Instead you would apply the vector constructor to create a vector from the sequence of numbers and pass in a set of vectors. It is surprising how quickly these common sense principles can get lost when writing software. In Chapter 4 we will explore in more detail how mathematical principles of notation inform data management and type design.

**Definition 3.3.1.** Let \( v \) be a vector of length \( n \). A matrix \( x \in X^{p\times q} \) is a two-dimensional data structure that can be constructed in the following ways.

(a) \( \text{matrix}(v, nrow = p) \equiv [v_{i,j}]_{p\times q} \), where \( pq = n \).
(b) \( \text{matrix}(v, ncol = q) \equiv [v_{i,j}]_{p\times q} \), where \( pq = n \).

Concatenation is more involved for matrices than vectors since the structure of the matrix must be considered. The intuition on how to construct a matrix from an existing matrix is revealed in mathematical notation. Given an \( m \times n \) matrix \( A \) and another matrix \( B \) with dimensions \( p \times n \), a new matrix \( P \) can be constructed by \( [A \, B] \). Similarly given \( m \times q \) matrix \( C \) the matrix \( Q \) can be constructed as \( [AC] \). When using this notation it is clear that concatenation is native to matrices but must be performed as a row-based or a column-based operation.

**Definition 3.3.2.** Let \( x \) be a \( m \times n \) matrix and \( W \) be an argument list where \( w_k \in W \) is a \( p \times n \) matrix, \( p \in \mathbb{N} \). Then

\[
\text{rbind}(x, W) = \text{rbind}(x, w_1, w_2, \ldots)
\]

\[
\equiv \begin{bmatrix}
x \\
w_1 \\
w_2 \\
\vdots
\end{bmatrix}
\]

The definition for \( \text{cbind} \) is similar but appends columns instead of rows.
**Definition 3.3.3.** Let \( x \) be a \( m \times n \) matrix and \( W \) be an argument list where \( w_k \in W \) is a \( p \times n \) matrix, \( p \in \mathbb{N} \). Then

\[
\text{cbind}(x, W) = \text{cbind}(x, w_1, w_2, ...)
\]

\[
\equiv [xw_1w_2...].
\]

Similar to vector concatenation, the empty set is the identity for both \text{rbind} and \text{cbind}.

### 3.3.1 Subsets and indexing

Since matrices are ultimately just data structures, their behavior is not restricted to one mathematical model. This principle is the same for vectors, where they can be viewed as vectors, sequences, \( n \)-tuples, or sets. Consequently set comprehensions work equally well on matrices. This operation effectively provides a syntax for describing a sub matrix declaratively. In pure mathematics this is an uncommon operation, but for computational systems it is useful.

When extracting elements from two-dimensional objects, it is helpful to think about matrix notation and row vectors versus column vectors. Suppose there is a matrix \( A \) composed as

\[
A = \begin{bmatrix}
-0.3865880 & 2.36008580 & -0.03091161 & 0.1965394 \\
-0.3453409 & 0.05575077 & -0.32740135 & 2.4701649 \\
-1.195847 & 0.54923611 & -0.29193250 & -0.8851350
\end{bmatrix}
\]

Clearly \( A \) can be defined as a column of row vectors and also as a row of column vectors. In the below representation, I use an unconventional notation that treats vectors as partially indexed matrix elements:

\[
A = \begin{bmatrix}
\tilde{a}_{1,*} \\
\tilde{a}_{2,*} \\
\tilde{a}_{3,*}
\end{bmatrix} = [a_{1,*}a_{2,*}a_{3,*}].
\]

Extracting the second row vector \( \tilde{a}_{2,*} \) is then \( a[2,*] \) while extracting the third column vector \( \tilde{a}_{*,3} \) is \( a[*,3] \). It is also possible to extract a subset of the matrix as either row vectors or column vectors by exploiting the duality with set comprehensions. The notation is similar to the one dimensional case where the logical expression is located in the row position or the column position. For our toy matrix \( A \) suppose we want to find the row vectors whose mean is less than 0, or \( \{x_{i,*} | \bar{x}_{i,*} < 0\} \) for \( i \in \{1,2,3\} \). The same operation in R is \( a[\text{rowMeans}(a) < 0, \] \), which illustrates that the choice of row or column vector is determined by the position of the expression within the indexing operation.

**Definition 3.3.4.** Let \( x \in \mathbb{R}^{p \times q} \) be a matrix and \( i \in \mathbb{N}_p, j \in \mathbb{N}_q \). Then

(a) An element of \( X \) is \( x[i,j] \equiv x_{i,j} \)

(b) A row vector is \( x[i,*] \equiv x_{i,*} \)

(c) A column vector is \( x[*,j] \equiv x_{*,j} \).
FIGURE 3.4: Map for 2 dimensions

\[
\begin{align*}
\text{rmap}(\text{fn}, \text{EMPTY}, y) & \%as\% y \\
\text{rmap}(\text{fn}, x, y=c()) & \%as\% \text{rmap}(\text{fn}, x[-1,], \text{rbind}(y, \text{fn}(x[1,]))) \\
\end{align*}
\]
(a) A row-based map operation

\[
\begin{align*}
\text{cmap}(\text{fn}, \text{EMPTY}, y) & \%as\% y \\
\text{cmap}(\text{fn}, x, y=c()) & \%as\% \text{cmap}(\text{fn}, x[-1,], \text{cbind}(y, \text{fn}(x[1,]))) \\
\end{align*}
\]
(b) A column-based map operation

Two important consequences arise from this definition. The first is that if \( i = N_p \) or \( j = N_q \), then \( x[i,j] \) is a column or row vector, respectively. Second, the indexing operator accommodates vectors of arbitrary length, which can be easily proven given definition 3.1.2. Vectors produced by indexing are by default column vectors. Hence even extracting a row vector results in a column vector. Typically this isn’t an issue since many operations are explicitly defined as working on either rows or columns.

For more complex transformations on matrices, it would be nice to use the map and fold concepts we developed earlier. However, as with indexing and concatenation, it is necessary to specify whether you are operating on row vectors or column vectors. Definitions for rmap and cmap are presented in Figure 3.4, while the corresponding rfold and cfold are presented in Figure 3.5.

Recall that functions operated under map simply maps a scalar to another scalar. In the two-dimensional case, arguments passed to the mapping function are vectors. Here the function does not need to result in a vector of the same length so any well-formed output is valid.

**Proposition 3.3.5.** Let \( f : X^n \rightarrow Y^p \) and \( x \in X^{m \times n} \). The function rmap operating on \( f \) results in

\[
\text{rmap}(f, x) = \begin{bmatrix} f(x_{1,*}) \\ f(x_{2,*}) \\ \vdots \\ f(x_{m,*}) \end{bmatrix},
\]

\( \forall x \in X^{m \times n} \).
Proof.

$$\text{rmap}(f, x) = \text{rmap}(f, x[-1], \text{rbind}(\epsilon(), f([x[1]])))$$
$$= \text{rmap}(f, x_{2:m,}, f(x_{1,}))$$
$$\to \text{rmap}(f, x_{3:m,}, \text{rbind}(f(x_{1,}), f([x[1]])))$$
$$= \text{rmap}(f, x_{3:m,}, \begin{bmatrix} f(x_{1,}) \\ f(x_{2,}) \end{bmatrix})$$
$$\to \ldots$$
$$= \text{rmap}(f, x_{m,}, \text{rbind}(\begin{bmatrix} f(x_{1,}) \\ \vdots \\ f(x_{m-1,}) \end{bmatrix}, f([x[1]])))$$
$$= \text{rmap}(f, x_{m,}, \begin{bmatrix} f(x_{1,}) \\ \vdots \\ f(x_{m,}) \end{bmatrix})$$
$$\to \text{rmap}(f, \epsilon(), \begin{bmatrix} f(x_{1,}) \\ \vdots \\ f(x_{m,}) \end{bmatrix})$$
$$= \begin{bmatrix} f(x_{1,}) \\ \vdots \\ f(x_{m,}) \end{bmatrix} \quad \square$$

**Corollary 3.3.6.** Given a function $f : X^n \to Y^p$ and $x \in X^{m \times n}$. If $y = \text{rmap}(f, x)$ then $y \in Y^{m \times p}$.

**Proof.** From ?? $\text{rmap}(f, y) = \ldots$ (INCOMPLETE)

Each application of $f$ yeilds a vector in $Y^p$. So $\text{rmap}(f, x) \in Y^{m \times p}$. \quad \square

Note that the same approach can be used to prove the $\text{cmap}$ version as well. This is left as an exercise for the reader.

**Proposition 3.3.7.** Let $f : X^n \to Y^p$ and $x \in X^{m \times n}$. The function $\text{cmap}$ operating on $f$ results in $\text{cmap}(f, x) = [f(x_{1,})f(x_{2,})...f(x_{m,})]$, $\forall x \in X^{m \times n}$.

**Example 3.3.1.** Earlier we showed subsetting with the rowMeans function. The same operation can be performed using $\text{rmap}$. The function $\text{mean} : \mathbb{R}^n \to$
FIGURE 3.5: Fold for 2 dimensions

\[
\text{rfold}(f, \text{EMPTY}, \text{acc}) \%as\% \text{acc} \\
\text{rfold}(f, x, \text{acc}) \%as\% \{ \text{rfold}(f, x[\cdot,1], f(x[\cdot,1], \text{acc})) \}
\]

(a) A row-based fold operation

\[
\text{cfold}(f, \text{EMPTY}, \text{acc}) \%as\% \text{acc} \\
\text{cfold}(f, x, \text{acc}) \%as\% \{ \text{cfold}(f, x[,-1], f(x[,-1], \text{acc})) \}
\]

(b) A column-based fold operation

**R** is fold-vectorized, which is compatible with \( \text{rmap} \). Given \( x \in \mathbb{R}^{m \times n} \), we have

\[
\text{rmap}(\text{mean}, x) = \begin{bmatrix}
\text{mean}(x_{1,*}) \\
\text{mean}(x_{2,*}) \\
\vdots \\
\text{mean}(x_{m,*})
\end{bmatrix} = \text{rowMeans}(x)
\]

Using Corollary 3.2 we know that \( \text{rmap}(\text{mean}, x) \in \mathbb{R}^m \).

This next theorem is the principle of argument aggregation, which shows that a function with a single matrix argument is equivalent to a function with a set of vectors as arguments.

**Theorem 3.3.8.** Let \( |w_k| = p, \forall k \in \mathbb{N}_n \). Then \( \exists n! \) equivalent functions \( \lambda U.(\lambda W.X)[W := U] \). Equivalently \( \lambda w_1w_2...w_n.X = \lambda U.\lambda w_1w_2...w_n.X[w_1 := u_{1,1}, w_2 := u_{1,2}, ..., w_n := u_{1,p}] \), where \( U \) an \( n \times p \) matrix.

**Proof.** By construction and relying on Theorem 3.3.7, it is possible to construct a matrix \( U = [w_1, w_2, ..., w_n] \). Then the function \( \lambda U.\lambda W.X[W := U] = \lambda U.\lambda w_1w_2...w_n.X[w_1 := u_{1,1}, w_2 := u_{1,2}, ..., w_n := u_{1,p}] \)

Since the matrix \( U \) is composed of \( n \) column vectors, the possible constructions of \( U \) is \( n! \).

In general two-dimensional data structures can be considered sequences of vectors. From this perspective multi argument functions can be collapsed into single argument functions where the operand is a two-dimensional data structure. This premise is what motivates functions like segment.

**Exercise 3.18.** Add a clause to map to perform the same function as cmap.

**Exercise 3.19.** Prove Proposition 3.3.7.

**Exercise 3.20.** Show what the range of \( \text{maprange}(x, 30, \text{mean}) \) is.
3.3.2 Example: Numerical integration

Sometimes the translation of mathematical expressions into a programming language runs into difficulties. Defining an approximation to an integral using the trapezoid rule is easy to express, but transforming the equation into an algorithm is not trivial. The reason is that the mathematical equation is silent about the actual implementation. To ease this process implementations can and do diverge from the underlying mathematics. The goal is to retain the ability to reason about the program by ensuring that the transformations are formally sound.

Numerical quadrature conceptually operates on a sequence of $x$-values called nodes. When the spacing between nodes is regular, then it is possible to use an expression that operates directly over the sequence. However with non-uniform spacing, this simplification does not hold. Applying the trapezoid rule over a non-uniform grid requires calculating the width between each adjacent set of nodes explicitly. Since we are interested in calculating the area of each segment it would be simpler to provide the endpoints of each segment to the trapezoid function. We can define the segment function

$$\text{segment} : \mathbb{R}^n \rightarrow \mathbb{R}^{n-1 \times 2}$$

that takes a sequence of nodes and returns a sequence of segments. Using the properties discussed in the section, it is simple enough to write an implementation that is equivalent to this construction.

$$\text{segment}(x) \equiv \text{cbind}(x[1:(\text{length}(x)-1)], x[2: \text{length}(x)])$$

Proposition 3.3.9.

$$\text{segment}(\vec{x}) = \begin{bmatrix} x_1 & x_2 \\ x_2 & x_3 \\ \vdots \\ x_{n-1} & x_n \end{bmatrix} = [x_{1:n-1} \ x_{2:n}]$$

Proof. Let $n = \text{length}(x)$.

$$\text{segment}(x) = \text{cbind}(x[1: (\text{length}(x) - 1)], x[2: \text{length}(x)]) = [x[1: (\text{length}(x) - 1)] \ x[2: \text{length}(x)]) = [x[1: (n - 1)] \ x[2: n)] = [x_{1:n-1} \ x_{2:n}]$$

Approximating a definite integral can be performed using the trapezoid rule, which instead of using rectangles uses trapezoids. For an interval $[a, b]$ the area of each trapezoid is $(b - a) \frac{f(b) - f(a)}{2}$. Hence the full integral is the sum over the specified domain. Since this sum iterates over a sequence of $x$ values
we can think of the iteration in one of two ways. The first is as a rolling window of length two where each window acts as the bounds of the current interval. We could use the `maprange` to accomplish this, which creates a matrix with two columns where one column represents the lower bound and the other is the upper bound for each interval. We already defined such a function and have proven that the transformation is correct. This means that we can use this function to transform our vector sequence into a sequence of bounds to pass into our trapezoid rule.

```r
area(f, a, b) %as% {
  (b-a) \frac{f(b) - f(a)}{2}
}

integral.definite(f, a, b, step.size) %as% {
  xs <- seq(a, b, by=step.size)
  segments <- segments(xs)
  sum(rmap(function(bs) area(f, bs[1], bs[2]), segments))
}
```

We can now show that this function is equivalent to the mathematical definition of numerical quadrature. This is less trivial than it appears because we are using our custom segment function to modify the original sequence of nodes. The explicit use of variables is also a new addition, although its effect is innocuous. Each iteration of `rmap` produces

\[
\text{segment area} = (\lambda b.\text{area}(f, b[1], b[2])) \text{segments}_{k,*} \\
= \text{area}(f, x_k, x_{k+1}) \\
= (x_{k+1} - x_k) \frac{f(x_{k+1}) - f(x_k)}{2}
\]

Hence by vectorization,

\[
\text{integral} = \text{sum}(\text{rmap}(\text{segments}, \lambda b.\text{area}(f, b[1], b[2]))) \\
= \text{sum}((\frac{(x_2 - x_1)f(x_2) - f(x_1)}{2}, ..., (x_n - x_{n-1})f(x_n) - f(x_{n-1}))) \\
= \sum_{k=1}^{n-1} (x_{k+1} - x_k) \frac{f(x_{k+1}) - f(x_k)}{2}
\]

**Exercise 3.21.** Implement integration using `maprange` and prove that it is equivalent to the mathematical equation.
3.4 Sequences, series, and closures

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 coupled with a rule for constructing the sequence. A very simple sequence are the factorials defined as \( \{k!\}_{k \in \mathbb{N}} \).

In terms of sets, a sequence typically maps an index to the set of real numbers. Creating the sequence of factorials is an example of vectorization and can be quickly constructed by calling `factorial(k)`, where \( k \in \mathbb{N}, n \in \mathbb{N} \). This simplified form is only possible because the underlying implementation is vectorized so an implicit `map` operation is performed on the input.

It is unlikely that a real-world sequence can be defined so easily. 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 \( \{\frac{x^n}{n!}\} \), where \( x \in \mathbb{R}, n \in \mathbb{N} \). This expands to \( e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} \), where each term in the series can be codified as

\[
\text{function}(x, n) \ x^n / \text{factorial}(n)
\]

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.

\[
f <- \text{function}(x, n) 1 + \text{sum}(x^n / \text{factorial}(n))
\]

Suppose we want 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 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 \( R \) and not \( R^n \). 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 \( X^n \rightarrow Y^n \). The resultant function definition becomes

\[
m <- \text{function}(n) \ \text{function}(x) \ \text{sum}(x^n / \text{factorial}(n))
\]

which is the curried form of the original function. This can be passed to `map` and executed over any interval.
map(m(0:5), seq(-4, 4, by=0.1))

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 \( \text{map} \) implementation of the approximation is very clear and concise. At times, however, it may be more desirable to take an alternative alternative approach and use the higher order function \( \text{fold} \). 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

\[
a_n = \frac{x^n}{n!} = \frac{x^{n-1}x}{(n-1)!n} = a_{n-1} \frac{x}{n}
\]

Although somewhat unconventional, this transformation expresses each coefficient as a recurrence 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 \( \text{cumprod} \) function, which implements this operation more efficiently than a direct use of \( \text{fold}^4 \).

\[
\text{ex} \leftarrow \text{function}(n) \text{ function}(x) \text{ sum}(1, \text{cumprod}(x / n))
\]

map(ex(0:5), seq(-1, 1, by=0.1))

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, which will be discussed in more detail in Chapter 8.

Continuing with our power series example suppose we multiply two power series together.

\[
f(x)g(x) = \sum_{n=0}^{\infty} a_n(x-c)^n \sum_{n=0}^{\infty} b_n(x-c)^n = \sum_{n=0}^{\infty} z_n(x-c)^n
\]

\(^4\)In general the use of an internal function will be far faster than a native R implementation.
where $z_n = \sum_{i=0}^{n} a_ib_{n-i}$. 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.

$$z_n = \sum_{i=0}^{n} a_ib_{n-i}$$

$$= \sum_{i=0}^{n} a_ib'_i$$

$$= a \cdot b'$$

where $b' = (b_n, b_{n-1}, \ldots, b_1) \forall b \in \mathbb{R}^n$. 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.

```r
mod.dot(x, y) %as% { x %*% rev(y) }
series.product(a, b) %as% {
    map(function(n) mod.dot(a[1:n], b[1:n]), 1:length(a))
}
```

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.

Showing that the Maclaurin series expansion is correct can take advantage of the proof for `map`. It is only necessary to show that the approximation function is correct for a single value of $x$. Once this has been proven, then by virtue of the proof for `map` it is valid $\forall x$ and is guaranteed to produce a sequence via `map`. As a first step, it is easier to consider the uncurried version
of the function
\[
e^x \sim f(x, n) = 1 + \text{sum}(x^n / \text{factorial}(n)) \\
= 1 + \text{sum}(x^{n_1, k} / \text{factorial}(n_{1,k})) \\
= 1 + \text{sum}(c(x^{n_1}, ..., x^{n_k}) / c(n_1!, ..., n_k!)) \\
= 1 + \sum_{i=1}^{k} \frac{x^{n_i}}{n_i!} \\
= \sum_{n=0}^{k} \frac{x^n}{n!}
\]

where \( n \in \mathbb{N} \).

Recursive functions behave similarly except that instead of an initial condition we have a halting criterion. This implies that the sequence is being generated in reverse. Remarkably all functions can be expressed recursively as this class of functions is the same as that for the lambda calculus \([\]. The advantage here is that recursive functions can be easily proven to be correct. Knowing how a function is transformed into a recursive function can facilitate showing that a function behaves as expected.

### 3.4.1 Error analysis and the Euler-MacLaurin formula

By definition numerical methods are not exact, so any self-respecting text must also include a discussion on errors. The below discussion is focused on translating mathematical models to code, while a full treatment of error analysis within the context of analytical systems is reserved for Chapter ??.

In Example ?? we looked at implementing numerical quadrature using the trapezoid rule. It would be useful to know how close our approximation is to the actual integral. The Euler-Maclaurin formula tells us that there is an analytic solution for smooth functions.

\[
\sum_{k=2}^{p} \frac{B_k}{k!} (f^{(k-1)}(n) - f^{(k-1)}(m)) + R,
\]

where \( B_k \) are Bernoulli numbers and \( R \) is the error term.

MORE

### 3.4.2 Example: Taylor series approximation

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

\[ \cos x = \sum_{n=0}^{7} \frac{1}{n!} (x - 2)^n \cos^{(n)}(2) \mid_{x=2} \]

that can then be evaluated 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

\[
\text{function(n)} \ (x-a)^n / \text{factorial(n)} * \text{eval(d(f,n))}
\]

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.

\[
\text{taylor}(f, a, k) \ %as\%
\{
  \text{function(x)} \ \text{sum(map}(0:k, \ 
  \text{function(n)} \ (x-a)^n / \text{factorial(n)} * \text{eval(d(f,n))))}
\}
\]

In the outer function the free variables are defined as arguments to this function. For completeness, we define the derivative function d as

\[
\text{d(expr, 0, name='a')} \ %as\% \ expr
\text{d(expr, 1, name='a')} \ %as\% \ D(expr, name)
\text{d(expr, order, name='a')} \ %as\% \ d(as.expression(D(expr, name)), order - 1, name)
\]

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.

As in the earlier case, this implementation is inefficient for large n. Presumably the same transformation should apply for the general form, but what
FIGURE 3.6: Approximation of $\cos(x)$ about $x = 2$

(a) Plot of $\cos(x)$ and approximations

```
xs <- seq(-3, 5, by=0.1)
cos2.3 <- taylor(expression(cos(a)), 2, 3)
cos2.7 <- taylor(expression(cos(a)), 2, 7)
ys3 <- map(xs, cos2.3)
ys7 <- map(xs, cos2.7)
```

(b) Code listing
happens with the derivative term?

\[ a_n = \frac{(x - a)^n}{n!} f^{(n)}(a) \]

\[ = \frac{(x - a)^{n-1}(x - a)}{(n-1)!} f^{(1)} \circ f^{(n-1)}(a) \]

\[ = \tilde{a}_{n-1} \frac{x - a}{n} f^{(1)} \circ f^{(n-1)}(a) \]

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 multiplied 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 \texttt{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.

```r
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.

```r
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 \( n \leftarrow 0:k \).\(^5\)

```r
taylor.f(f, a, k) %as%
{
  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

\(^5\)The exception is that 0! cannot be deconstructed recursively, so the zero case is provided explicitly. The consequence is that \( n \leftarrow 0:k \).
as codified by Kernighan: "Controlling complexity is the essence of computer programming." [13] 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.