# PRINCIPLES OF COMPUTER SCIENCE

## An Invigorating, Hands-on Approach

Joshua Crotts

J.ROSS
PUBLISHING

ISBN-13: 978-1-60427-199-7

Printed and bound in the U.S.A. Printed on acid-free paper.

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data can be found in the WAV section of the publisher's website at jrosspub.com/wav.

This publication contains information obtained from authentic and highly regarded sources. Reprinted material is used with permission, and sources are indicated. Reasonable effort has been made to publish reliable data and information, but the author and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

Direct all inquiries to J. Ross Publishing, Inc., 151 N. Nob Hill Rd., Suite 476, Plantation, FL 33324.

Phone: (954) 727-9333
Fax: (561) 892-0700
Web: www.jrosspub.com

*To my Viola.*

## Acknowledgments

Many were skeptical of my desire to not only write but finish a textbook. Some tried to dissuade me from the idea altogether, claiming it was too much of a time sink. Indeed, while it took over much of my life, I am proud to present the finished product.

Special thanks to the many students and UIs from C211/H211 (Introduction to Computer Science) and C212 (Introduction to Software Systems) at Indiana University for their encouragements and willingness to read the book upon release. Moreover, my friends at IU and UNC Greensboro have constantly provided extensive motivation to continue when times were tough.

I very much appreciate William Orrell, Peter Stratta, and Cliff Mansfield for taking the time to read through partial drafts of the book.

The help and work invested by Gabriel Costa de Oliveira into the first chapter is extremely acknowledged and appreciated.

I thank those who answered the many questions that I posted to the TeX StackExchange regarding LaTeX formatting issues; I hope they are as invaluable to others as they were for myself.

Andrew (Matsurf) Matzureff is always a joy to talk with, and his comments on parts of the book and discussions on continuations were hilariously informative.

In writing *Principles of Computer Science*, I drew inspiration from many authors of other textbooks: including Jeremy Siek, Dan Friedman, Daniel Holden, Robert (Bob) Nystrom, Matthias Felleisen, and others as cited in the bibliography.

Andy Huber's comments, suggestions, and corrections turned this from a textbook into a work of art.

Thanks also to those professors and teachers who taught me everything I know and love about both logic and computer science: Steve Tate, Insa Lawler, Nancy Green, Dan Friedman, Gary Ebbs, Chung-chieh (Ken) Shan, and Sam Tobin-Hochstadt (as well as those not listed). Moreover, Mary Beth Ferrell instilled many lessons on grammar and writing from her AP Literature and Composition course that I utilize to this day (alongside the ever present nightmares of having to complete in-class essays or forgetting to finish an assignment).

I am also sincerely grateful for the inspirational comments from my parents and my wife.

Lastly, I am absolutely and deeply indebted to Tony Smith: my former Advanced Placement Computer Science teacher. Thanks for everything.

# Table of Contents

## List of Figures

# Preface

In July 2021, I had the privilege of meeting Dan Friedman, an esteemed figure in the realms of theoretical computer science and programming languages. Our encounter opened the door for me to delve into the realm of functional programming, particularly in the context of Scheme. I concluded that the best approach to learning Scheme was by implementing it myself. Thus, I embarked on the journey of crafting a simple Scheme interpreter using Java, which, although functional, remained rudimentary. Nevertheless, my affection for the language continued to blossom. Despite being content with the initial version of the interpreter, I made the daring decision to rewrite the entire project in C, an undertaking brimming with both excitement and peril. Drawing inspiration from Dan Holden's remarkable work, *Build Your Own Lisp*, and his exceptional parser combinator library, I undertook the task of reconstructing a Scheme in the C language.

Why should you, as the reader, care about all of this? It led me to a profound realization that certain subjects within theoretical computer science, such as language design and implementation, are often inaccessible to beginners without significant programming experience. I found myself pondering, "How can I make these concepts approachable for those new to this field?" and "What steps can I take to showcase the awe-inspiring nature of computer science to readers?" Undoubtedly, computer science can be intimidating and requires time to master. Some individuals may have the curiosity and drive to embark on this journey, while others may initially fail to recognize or comprehend its allure. This book endeavors to enlighten both these curious audiences and many more, introducing them to a world of design, creativity, and boundless expression.

## Prerequisite Knowledge

The primary target audience for this book is individuals who do not have a background in computer science. We assume that readers possess only high-school level knowledge of algebra, and for the majority of the text, we relax the requirement for trigonometry. While an interest in computing is preferred, it is not strictly necessary, as we aim to cultivate that interest through each successive chapter. The book is structured linearly, intending that readers progress from the beginning to the end in most cases. Those with a background in discrete math or some aspects of computer science, however, have the option to skip the first five chapters should they so choose. Similarly, programmers proficient in C may skip Chapter 5.2. By this point, our goal is that all readers are at least familiar with the topics presented, thereby leveling the playing field, so to speak.

## Exercises

Practice makes perfect in computer science, which is why we supplement our text with code listings and exercises, the latter of which range in difficulty using the following scale:

- Exercises marked with one star (⋆) are "finger exercises", meaning they can be completed in either a few seconds or, at most, a couple of minutes. They often involve writing one or two lines of code or a sentence derivation.
- Exercises marked with two stars (⋆⋆) are slightly more complex, as they may entail defining a function or something that requires a bit more thinking, taking around 5 to 15 minutes to complete.
- Exercises marked with three stars (⋆⋆⋆) are moderately difficult and may take some time to understand and finish. Depending on the reader, this time may be between 15 minutes and an hour. Large function definitions, or complicated code, are common in this type of exercise.
- Exercises marked with four stars (⋆⋆⋆⋆) are the most challenging problems. Significant thought must be invested into these exercises, but once complete, demonstrate extremely high proficiency. Very large programs, multiple function definitions, and more are what lie ahead, taking multiple hours or even (broken up) days to finish.
- Exercises marked with five gold stars (⋆⋆⋆⋆⋆) often introduce new concepts to coincide with the presented task. While we present all the relevant material to complete the problem, some outside references or resources may be necessary to fully understand everything. Gold star problems are even harder than four star problems, but they do not necessarily entail writing *more* code than the latter.

Every exercise in this book is designed to be achievable for any reader and are presented to stimulate creativity. We firmly believe in not writing exercises that are overly (and perhaps unnecessarily) challenging and may demotivate learners. If you find an exercise to be too difficult after making a genuine effort, feel free to skip it and return to it later at your own pace.

## Programming Environment

The majority of the code presented in this text is written in C (do not worry if you are not familiar with C yet). As a result, all C listings are tailored to function on MacOS and Linux operating systems, which regrettably excludes Windows users, who form a significant portion of the audience. One possible workaround for Windows users could be to utilize an online compiler, e.g., `https://replit.com`. In the event that any readers have trouble setting up their coding environment, we provide a setup on `replit` containing all necessary files and packages. Appendix 11.3 describes how to get started with programming in C on MacOS, Linux, and the `replit` sandbox.

## Reading Tips and Tricks

In order to make the most of reading this book, it is important to approach it with care and a deliberate pace. Remember, the words will always be there on the page, and there is no rush to reach the non-existent finish line. It can be beneficial to have a pencil or pen at hand to take notes in the margins. For digital readers, using markup software to annotate the text can serve the same purpose. It is not mandatory to code alongside the text, as doing so may potentially disrupt the reading experience. At the end of each section, however, it is advisable to practice with the code that has been provided, as it offers an opportunity for hands-on learning.

The diction and punctuation throughout this book have been carefully chosen and scrutinized. We introduce commas to break a sentence up into chunks, which aid in conveying either a message or serve as to make the dialog more conversational. Semi-colons, on the other hand, relate concepts together whose central idea would be weakened if separated by a full-stop period. Colons are often accompanied by an explanation, code listing, or exercise; they denote the significance of what immediately follows the colon. Footnotes often provide further information about a topic, reference a popular citation/author, or break up the text monotony. These footnotes should not be ignored, and ought to be read as soon as you arrive at their linking footnote index.

## One Last Remark

Being a PhD student, I acknowledge that my teaching qualifications may be deemed modest and subject to scrutiny. I sincerely hope that the absence of a formal background, for now at least, does not diminish my aspirations for both the audience and myself. In the process of imparting knowledge to my students, I am constantly learning alongside them. Teaching is a deep-rooted passion of mine, yet I firmly believe that learning is a perpetual skill that we can never truly master.

Have a blast!
*Joshua Crotts*

WAV Web Added Value™

This book has free material available for download from the
Web Added Value™ resource center at *www.jrosspub.com*

At J. Ross Publishing we are committed to providing today's professional with practical, hands-on tools that enhance the learning experience and give read-ers an opportunity to apply what they have learned. That is why we offer free ancillary materials available for download on this book and all participating Web Added Value™ publications. These online resources may include interac-tive versions of the material that appears in the book or supplemental templates, worksheets, models, plans, case studies, proposals, spreadsheets and assessment tools, among other things. Whenever you see the WAV™ symbol in any of our publications, it means bonus materials accompany the book and are available from the Web Added Value Download Resource Center at www.jrosspub.com.

Downloads for *Principles of Computer Science* include the Library of Congress Cataloging-in-Publication data and information on a GitHub repository featuring instructional material for classroom use (lecture slides, exercise solutions, etc.).

# 1 A Computing Mindset

*Any sufficiently advanced technology is indistinguishable from magic.*
—Arthur C. Clarke

## 1.1 Computer Heuristics

And magic, it is! Or... is it? Instead of going straight to ones and zeros, let us take several steps back in an attempt to look at these fantastic machines with the critical eyes of a careful spectator in a magic show. As we watch the wondrous tricks and disappearing elephants,[1] we will examine how we got there and how much of that magic can be mastered through many different methods. To this end, we will use a descriptive approach to explore the nature of a computer, its functions, and the manner in which it operates.

### A Filing System

Remember those big-old filing cabinets? Perhaps you are too young for that. You might have seen them in movies at some point. Regardless, imagine for a second you are in the 1950's equipped with no digital computers.[2] Every aspect of your day-to-day life, both significant and otherwise, involves laborious manual accounting for your self-established retail business.

Your company has various employees: sales people, front-desk assistants, general managers, and so on. Every single employee has a file in your main filing system. Each file has information on it such as the employee's name, their address, their general salary, commission rate, etc. We will call that information *data*.

[1] Perhaps "elephant" is an ill-phrased metaphor for some magical phenomena.

[2] The term *computer* was first used in "The Yong Mans Gleanings", where it referred to an individual who performed monotonous mathematical calculations [Braithwaite, 1613]. In the early 20th century, to support the effort towards the United States' involvement in World War II, women were hired as computers for this very task, often overshadowed by their male counterparts [Smith, 2013].

Every month, you must *compute* your general expenses. You gather your accounting team to meticulously examine each file, adding up the general salaries of all employees. For sales personnel, they locate an additional file containing monthly sales information, including the salesperson's name and the amount sold. Accounting then applies the commission rate from their records, multiplying it by the sales figure. Voila! They record each employee's total in a substantial accounting book, calculate the sum, and that becomes your general expenses. The same process is more or less followed for revenue, but you can imagine that part. This repetitive task is carried out month after month.

Now, just picture for a moment that you are presented with a new employee. This individual possesses extraordinary speed—five times faster than your entire accounting team. It seems magical! There is one predicament, however: this employee lacks intelligence. To enable them to adequately perform the job, you must meticulously document every single task they need to carry out.

### An Algorithm

On its first day at work, your faster-than-human employee sits at a desk and finds this on a notecard:

1. Go to the second floor.
2. Open the cabinet that reads: "Employees"
3. For all cards in the "Employees" cabinet, write down, in the accounting book, their name in the first column and their salary in the second column.
4. If the employee is a salesperson, find the monthly sales book.
5. Look up the total sales for that employee and multiply that number by the employees commission rate on their card.
6. Sum the result with their base salary and write down the total value next to their name in the accounting book.
7. When you are done with all the employees in the cabinet, sum all of the payments and write down that number as the last entry in the accounting book.

This, as simplified as it can be, is an *algorithm*: A set of well-established instructions that can be performed potentially repeatedly by your employee. Once written down, it can be interpreted in a way that, in the end, produces a desirable result, assuming it is clearly written.[1]

[1] A "clearly written" algorithm is one that is unambiguous. The unplugged "peanut butter and jelly sandwich" (wherein students write a meticulous algorithm that a computer might use to construct a peanut butter and jelly sandwich) project is an excellent introduction to this idea.

Now, is this not delightful? We not only save precious time, but our employee becomes more cost-effective. But why stop there? Picture this: your staffing agency presents you with an even faster employee. This individual is absolutely astonishing, approximately ten times swifter than your previous superhuman staff member. Quite appealing, right? There is just one small snag: this new addition lacks knowledge of division and multiplication. Their speed, however, is unparalleled–truly remarkable. This employee can retrieve information so rapidly that you are contemplating breaking down your algorithm even further! All you need to do is create lookup tables for multiplication. Even better, you can define multiplication as the consecutive addition of a number to itself. For example, 3 times 5 is simply $5 + 5 + 5$. So, now you proceed to devise a methodology for explaining how multiplication works to your new employee. Perhaps something like what follows:

*To multiply two numbers $a$ and $b$ together, do the following:*

1. Look up the multiplication table for $a$.
2. Find the value where it crosses $b$ in the table.
3. Get result.

This would be pretty easy for the new employee. After all, they are really fast at looking things up. An alternative would be:

*To multiply two numbers $a$ and $b$ together, do the following:*

1. Get the larger number and add to itself as many times as the smaller number.

This is a solution that involves just adding the number, your new employee knows how to add, and does it fast![1] Now, we have simplified this as an example, but the ideal trajectory should be clear.

Our next call from the staffing agent promises the impossible. There is a new guy; one with, dare we say, unrivaled potential. Nearly as fast as the speed of light, things for this magical creature happen basically instantaneously. So you are asking yourself: “What is the catch?” Well, the new employee does not know how to read. In fact, it cannot derive meaning out of anything. All your previous algorithms are no longer useful, but you must, somehow, tap into this employee’s power!

### A Computer Language

The staffing agency informs you that this new individual excels at information retrieval and basic operations, given the right set of instructions. Moreover, they have no need for rest or sustenance, nor do they require a salary. Allow us to explain how this particular employee functions.

[1] We will ignore the complexities of determining the “largest” of two numbers for the time being and assume the new employee understands this concept.

Firstly, they possess the ability to look up information. Secondly, they can store a limited amount of data in their memory. Thirdly, they are capable of performing basic arithmetic. It is important to note, though, that their knowledge is restricted to only eight digits.

At this point, you might express concern regarding the limitations of such a narrow numerical range. How could they possibly accomplish anything with only two eight digits?

Rest assured, the staffing agency states, with utmost confidence, that they have a solution. All you need to do is *codify* your filing system in a way that enables the employee to navigate and identify the desired information. For instance, you can devise a system where the first two digits indicate the floor of a file, the following two digits indicate the room, and the last four digits indicate the cabinet. Admittedly, this system, even for the most astute among us, is not the most optimal 8-bit system. Nevertheless, it allows you to encode your files accordingly. The first two numbers can represent the department to which the file belongs, and so on. You can easily envision where this is leading.

At this point you may starting to convince yourself that maybe this poor fool of a machine can actually be useful. You think, to yourself: “if all I need to do is codify my file system and algorithm, then this can work!”

So you start codifying your file system; that was easy. You then start writing instructions.

1. Go to 00010100
2. Go to 00000101
3. Put value of 00100010 in Memory 0
4. Go to 01001000
5. Go to 00010000
6. Put value of 00100011 in Memory 1
7. Add value in Memory 0 to value in Memory 1.
8. Write result in 00000100

You look at it with a puzzled gaze. You do not remember what any of these 1’s and 0’s mean. You tell yourself: “this will not work; how can any one keep track of any of this? It is impossible!”

Along comes Alice, who is your smartest employee. She’s been working at the company for many years and she is on track to be its newest CEO next year. She informs you that this can work; all you need to do is to make this thing readable and devise a way where the readable bit can be turned into this primitive set of instructions for your new machine.

Alright, fasten your seat-belts because the adventure is about to begin! What you've read so far was just a prelude to the exciting journey that lies ahead. It is not simply about employees racing through a company's building or the basic functions of computers; it is a thrilling exploration of the intricate world of computers and how they shape our lives. Let us make one thing clear: computer science goes beyond the physical machines; we dive deep into programming language design, unraveling its mysteries alongside the underlying mathematics. We embrace the challenges, unearthing the secrets of computation and pushing the boundaries of what is possible. So, get ready to embark on this epic adventure where the foundations of computer science and its theoretical concepts blend harmoniously with practical applications. Brace yourself for a wild ride, as we delve deeper into uncharted territories and satisfy your thirst for knowledge with each turn of the page.

# 2 A Logic Primer

*Science is not a substitute for common sense, but an extension of it.*
—Willard Van Orman Quine

## 2.1 Zeroth-Order Logic

What is logic? A general definition is the use of reasoning to determine the truthfulness of some claim. For instance, it is logical to close the refrigerator door after opening it and retrieving your desired items because if we did not, the cold temperature is no longer encapsulated, leading to some foods or ingredients going bad. We can simply say something to the effect of, "If I do not close the refrigerator door, my food will go bad." This is known as an argument involving deductive reasoning. In other words, we used a form of reasoning which involves premises that lead to, or imply, a conclusion. For this small example, it is easy to understand the reasoning behind the argument. What happens, though, if we create a larger and more complicated argument? Suppose that, in the following example, each itemized bullet point is a proposition to an argument (note that we will examine the argument in greater detail later in the chapter—we introduce it here as motivation).

- Janet goes to university.
- If someone is a mathematician, then they have a lot of experience with using computers.
- Janet does not know how chemicals interact if she is not a chemist.
- Janet is either a chemist or a mathematician if Janet goes to university.
- Janet goes to university as a computer science student.
- Janet is a chemist if and only if she is not a mathematician.
- Therefore, Janet does not know how chemicals interact.

We analyze logical arguments through two lenses: validity and soundness. An argument that is valid has premises that logically imply the conclusion. Though, what is a premise and conclusion, and what in the world does the phrase "logically imply" mean?

### Form of an Argument

Firstly, statements that express a true or false claim/idea are *propositions*. For example, “The sky is blue”, “$2 + 2 = 5$”, “If $(n)^2 = 4$ then $n$ is either $-2$ or 2”. A proposition can be either true or false, but it must be a statement as opposed to an exclamatory, command, or question, as these do not have associated truth values.[1]

*Premises* are propositions that support an argument claim. In other words, a premise expresses truth or falsity. *Conclusions*, similar to premises, also express the truth of a claim, but serve as a judgment to the claim as a whole. A collection of premises either support or reject the judgment asserted by a conclusion. Such support is called a logical implication of a premise to a conclusion.

Before we discuss what it means for one premise to logically imply another, we must describe the notion of truth values. A truth value takes the form of true, i.e., $\top$, or false, i.e., $\bot$. Because of this, any proposition will, by default, have two possible values: $\top$ or $\bot$. As a corollary point, all propositions are either true or they are not true. As an example, “Either $2+2 = 5$ or $2+2 \neq 5$”. In this example, the proposition is “$2 + 2 = 5$”. This is clearly an absurdity, i.e., $2 + 2 = 5$ is $\bot$, but for the purposes of this claim, it does not matter, as it is true that $2 + 2 \neq 5$, and false that $2 + 2 = 5$. This style of argument is otherwise known as the law of excluded middle.

Defining truth in such a bivalent respect is a problem rooted deep in philosophy. Alfred Tarski coined the idea of using *schemata*, or formulae of our logic language, to provide a differentiation between the use of truth as a literal versus truth in the conceptual sense. In a broad sense, we state a claim in our language, then it is provided a truth value from our metalanguage. The classic example is, “Snow is white” is true if and only if snow is white.

In this text, we will represent propositions as lower-case letters, e.g., ‘$p$’, ‘$q$’, ‘$r$’, . . ., ‘$z$’.[2] To symbolize the above claim, let $p$ : “$2 + 2 = 5$”. Now, let us substitute this into our statement: “Either $p$ or $2+2 \neq 5$”. We have run into a small roadblock; how do we represent the negation of a proposition? Negating propositions requires a connective, which we will now explain.

### The Connectives

Every day we use connectives in our speech or text without realizing. *Connectives* link related phrases/words/ideas together with the intent of strengthening or weakening expressed statements.

[1] If this does not make sense so far, think of a command that you might tell someone, e.g., “Go to the store and buy a loaf of bread”. Is it possible to express the truthiness of this command and, if so, what is it? Another example might be to ask someone, “What time is it?”. How can a question, in and of itself, be true or false?

[2] Different authors describe different notation(s), or syntax, to represent the same semantic idea.

### Negation

In most instances, for any proposition '$p$', it is safe to use the phrase, "It is not the case that '$p$' is true", to represent the *logical negation* of '$p$'. This is rather cumbersome to spell out, however, so we symbolically use '$\neg p$' to denote the negation of proposition '$p$'. Now, let us retry our substitution method from earlier: "Either '$p$' or '$\neg p$' is true". Note that '$\neg p$' denotes "It is not the case that $2 + 2 = 5$ is true", which is semantically equivalent to $2 + 2 \neq 5$ (we may read this as "Two plus two is not equal to five"), but even this is still a bit awkward to read as we are intertwining logic and proposition symbols with English statements.

The issue with negation in natural language is that, often times, sentences do not have a straightforward binary conversion between non-negation and negation. For instance, suppose we have the two propositions, "Bob is happy", and "Bob is sad". Could we say that the latter is a mere negation of the former? In this case, we have to analyze what it means to be "not happy". Someone who is not happy may not necessarily be sad; they may be angry, neutral, or many other emotions. All we can express from the proposition that "Bob is happy" is that its negation is, "Bob is not happy", or equivalently as aforementioned, "It is not the case that Bob is happy".

**English Equivalents.** As we stated, negation in natural language is complicated at best and ambiguous at worst. Some equivalents of "negation" in natural language include "not" and "it is not the case that". Continuing with the explanation from the previous paragraph about converting a proposition into its negated counterpart, if we have the phrase "Bob is happy", we cannot prefix the phrase "happy" with "un-", as it does not indicate the negation of "happy". Furthermore, as we will see later on, negating quantified statements is also challenging. Suppose we have the following proposition: "Everyone is happy". Can we negate this by saying "Nobody is happy"? No, we cannot; the negation of "Everyone is happy" would be "It is not the case that everyone is happy", or more concisely, "Everyone is not happy". This suggests that there is someone that is not happy, but it does not suggest that nobody is happy. Negating statements takes practice and an understanding of English semantics.

### Disjunction

"Either… or…" is the form of the first binary, or two-place, connective we will analyze. Symbolically, we represent this with $\vee$, also known as *logical disjunction* or alternation. Other non-symbolic representations of disjunction include, "or". Returning to the prior example, we can now use symbols to fully convert the previous example to symbolic logic: '$p \vee \neg p$'.

**English Equivalents.** Logical disjunction, as we stated, is most often conveyed in the word “or”, e.g., “Jill may have cake or ice cream”. Sometimes, it is implicitly used as part of an enumeration of choices, e.g., “Jill may have one of ice cream, cake, or donuts”. There are other equivalents such as “otherwise” and “alternative”, all of which convey inclusivity. Interestingly enough, in common speech, logical disjunction is often used in the exclusivity sense, i.e., a choice between options is mandatory, and nothing in between is permitted. In logic, we convey exclusive disjunction (also called exclusive or) commonly using “Either $p$ or ‘$q$’ but not both (and not neither)”. In a purely logical interpretation of propositions, however, both ‘$p$’ and ‘$q$’ being true is allowed. Considering one of the examples from before, if Jill has both cake and ice cream, this still expresses a true proposition, because they had at least one of cake and ice cream. If we introduce exclusivity via “Jill may have either ice cream or cake, but not both (and not neither)”, if Jill has both desserts or neither of the desserts, the proposition is false.

### Conjunction

Next, we will discuss conjunction, or the connection of propositions with non-symbolic words such as “and” and “but”. A *logical conjunction* between schemata expresses the idea that each schema individually must be true in order for the conjunction to resolve as true. For example, consider the schemata ‘$p$’ and ‘$q$’. If we know that both ‘$p$’ and ‘$q$’ are individually true, then we can say ‘$p \land q$’ is true. Conversely, if either ‘$p$’ or ‘$q$’ are false, then ‘$p \land q$’ is false. Symbolically, as we just demonstrated, this connective uses the inverted wedge ($\land$) to designate a logical conjunction.

**English Equivalents.** English equivalents of logical conjunction are used frequently in day-to-day speech. For example, “Steve is a cook and a swimmer” is a conjunction between the two propositions of Steve being a cook and Steve being a swimmer. Adverbs such as “additionally” and “moreover”, in general, express the same idea as logical conjunction. So, “Steve is a swimmer. Additionally, he is a cook”, and “Steve is a swimmer. Moreover, he is a cook” express semantically-equivalent propositions.

*Example.* Suppose we want to represent the statement, “John is a computer science major and Billy does not like to fish”. Again, we assign the propositions to letters, e.g., $p$ :“John is a computer science major”, and $q$ :“Billy does not like to fish”. right? Not quite, in this scenario. When negating a schema, we use the $\neg$ symbol, e.g., ‘$\neg q$’. If we assign ‘$q$’ as we did, then there is no negation on the propositional letter. While this is not incorrect, it is correct and widely accepted to always assign the positive version of a proposition to a letter. Thus, instead, $q$ :“Billy likes to fish”, meaning when we attach a negation to the proposition via ‘$\neg q$’ we get the statement, “It is not the case that Billy likes to fish”, one that is semantically equivalent to the original. Finally, the symbolic logic expression is ‘$p \land \neg q$’ since we must conjoin the two sub-schemata.

### Logical Conditional

The *logical conditional* operator is the most complex connective that we will investigate in zeroth-order logic. Its semantic meaning in context is not always obvious. Conditionals are used in sentences where the following phrases exist: "if", "only if", "then", and "implies" (note that, as before, this is a non-exhaustive list of hints). Symbolically, this connective uses an arrow (→) to designate a logical conditional. A point should be made about a logical conditional and its semantics: Logical conditionals are created, as we stated, with the arrow, e.g., '$p \to q$'. *Logical implication*, on the other hand, is a related concept but requires its own description: it ascribes the truth conditions between a collection of schemata and another individual schema. In summary, logical implication explains, in our system, how propositions imply one another, whereas the logical conditional (also sometimes referred to as the conditional with the "logical" dropped) is nothing more than a new syntactic connective.[1] We will use logical conditional in the following example.

**English Equivalents.** Conversion of the logical conditional into speech is more difficult because of its obtuse and confusing meaning. It is still possible, though, to translate using phrases such as those mentioned above. Time-based or action-based adverbs are also sometimes indicators of a conditional. "Whenever Joe is cooking, he smiles" expresses the idea that if Joe cooks, then he smiles. "When", "Given that" and "In the case of" all share the same sentiment of the conditional.

*Example.* Suppose we want to convert the following sentence into symbolic logic: "Katherine is not good at tennis only if she does not practice and she is not motivated to play". Let us first symbolize the propositions: We will assign the proposition $p$ : "Katherine is good at tennis", $q$ : "Katherine practices tennis", and $r$ : "Katherine is motivated to play tennis". Notice how we took some liberties in substituting pronouns for names, as well as adding details to clarify the propositions. This is always acceptable and desired in symbolic logic; as long as it remains semantically equivalent to the original proposition, any translation is valid.

[1] Another way of reasoning about these contrasting ideas is to take two arbitrary schemata '$p$' and '$q$', then stick a conditional in between, i.e., '$p \to q$'. The existence of a logical conditional does not guarantee, or even necessarily suggest, that '$p$' implies '$q$'. Once we get to truth conditions, we will better understand why this is the case.

### Logical Biconditional

*Logical biconditional* is often paraphrased as, "...if and only if...", "just in case", "iff", and symbolically represented as, $\leftrightarrow$ (note that this should not be conflated with "if" and "only if", as there is an important distinction with "if and only if"). For example, "Computers are necessary if and only if they are used by everyone in the world" uses the propositions $p$ : "Computers are necessary" and $q$ : "Computers are used by everyone in the world". Symbolically, '$p \leftrightarrow q$'. The important piece of this is hidden by the syntactic sugaring of $\leftrightarrow$; in actuality, a biconditional is nothing more than the conjunction of two conditionals: '$(p \to q) \land (q \to p)$'. In other words, two propositions must imply one another in order to represent a logical biconditional. Each sub-component of the expression, i.e., '$(p \to q)$' and '$(q \to p)$', corresponds to either the "if" or the "only if". The left-hand side represents "if", whereas the right-hand side represents "only if". We can, of course, convert any "if" expression to an "only if", although it may not be a valid expression at that point.

**English Equivalents.** Compared to other connectives, the biconditional is a bit harder to find variants of, besides the standard "if and only if". We want to use phrases that express equivalence of two ideas. So, "is equivalent to", "exactly when", and "necessary and sufficient" are good replacements for a biconditional in speech. The last example exemplifies the nature of the biconditional. That is, a proposition '$q$' being necessary for '$p$' is expressed by '$p \to q$', whereas a proposition '$q$' being sufficient for '$p$' is expressed by '$q \to p$'. Combining these together with a conjunction forms '$(p \to q) \land (q \to p)$'. A property of the biconditional is its slight ambiguity in mathematical proofs. The phrase "if and only if" may seem to connect directly with the previous schematization, but this is not the case. '$p \to q$' expresses, "$q$ if $p$", or, "$p$ only if $q$". This means that the left-hand side of the conjunction is, in actuality, the "only if" direction of the conditional, and the right-hand side refers to the "if" direction. Another interpretation of this is to say that the "if" refers to, "if $p$, then $q$", which suggests the ordering of the connectives is correct. Either way, because logical conjunction is commutative (i.e., '$p \land q$' is equivalent to '$q \land p$'), it matters not so much.

So, returning to the previous example, we now know what "only if" is symbolically, so we can convert the entire sentence into symbolic logic: '$(\neg q \land \neg p) \land \neg r$'.

### Ambiguous Expressions

We can complicate things slightly by adding *ambiguity* into the equation. For example, let us convert the statement, "Samantha does not play video games or Ryan is a server and Paul is a pianist" into zeroth-order logic. Right away, we find an issue with our translation process: Firstly, there are three propositions instead of two. Thankfully, this does not hinder our progress significantly. What does, on the other hand, is the ambiguity of the expression. Similar to rules for operator precedence and associativity for basic mathematical operations such as addition and subtraction, logical connectives also have precedence values: We evaluate negations first, conjunctions second, disjunctions third, logical conditionals fourth, and biconditionals last. With this guideline in hand, we can now schematize the English sentence.

*Example.* Suppose $p$ : "Samantha plays video games", $q$ : "Ryan is a server", and $r$ : "Paul is a pianist". Symbolically, this is equivalent to, '$\neg p \lor (q \land r)$'. According to our precedence rules, we evaluate the conjunction, i.e., "and" before "or". So, while it is not strictly necessary in this case since we laid out precedence rules, not all sources on symbolic logic provide a precedence list. In these instances, much like mathematical expressions, we insert parentheses to denote that an operator evaluation has higher priority. For example, '$\neg p \lor (q \land r)$'. For completeness, we can insert outer parentheses to group '$\neg p$' and '$q \land r$' with the $\lor$ operator, but it is unnecessary: '$(\neg p \lor (q \land r))$'.

Converting entire arguments, instead of individual sentences containing propositions, is fortunately very simple. The premises of a *valid* argument, when conjoined, imply the consequent. *Sound* arguments are both valid and have only true premises. We will construct a model for arguments. Let $A(P, c)$ denote an argument $A$ with a collection of premises $A_P$ and a conclusion $A_c$. So, suppose we have an argument $A$ such that $A_P = \{a, b\}$ and $A_c = c$.[1] An argument can have zero or more premises, but it must have a conclusion. Thus, $A$ is logically valid if and only if '$(a \land b) \to c$' is true. In a later section, we will investigate what it means for a proposition to be true or false.

We finally have all the tools necessary to convert the argument on page 21 into symbolic logic.

*Example.* Let $p$ : "Janet knows how chemicals interact", $q$ : "Janet is a chemist", $r$ : "Janet is a mathematician", $s$ : "Janet goes to university". We need to identify our premise set and conclusion. The conclusion of an argument is, in many instances, easy to identify if told outright. We can use conclusion word indicators, e.g., therefore, so, hence, thus, and more. In the argument, we are told, "Therefore, Janet does not know how chemicals react". So, our conclusion is, symbolically, $\neg p$. The remaining sentences are, as such, premises. Let us construct each premise piece by piece.

(i) "Janet does not know how chemicals interact if she is not a chemistry student" is represented as '$\neg q \to \neg p$'.

[1] What we allude to, in regards to the representation of $A_P$, is called a *set*, and we will introduce them in the next section of this chapter.

(ii) "Janet is either a chemistry student or a computer science student if Janet goes to university" is represented as '$s \rightarrow (q \lor r)$'.

(iii) "Janet goes to university and is a computer science student" is represented as '$r \land s$'.

(iv) "Janet is a chemistry major if and only if she is not a computer science major" is represented as '$p \leftrightarrow \neg s$'.

Now that we have schematized each premise individually, we can combine them using the conjunction operator, then connect the resulting conjunction to the conclusion with the conditional: '$((((\neg q \rightarrow \neg p) \land (q \lor r)) \land (r \land s)) \land (p \leftrightarrow \neg s)) \rightarrow \neg p$'. Neither the validity nor soundness of this argument are of concern to us for now; we simply wanted to symbolize its premises and conclusion. The form of a symbolic logic formula was also of no concern since we only had small sentences.

### Inductively Defining Zeroth-Order Logic

In order to properly parse and evaluate large ones, to ensure no errors are made, we need to inductively define valid expressions within our symbolic logic language. An inductive definition allows us to build the components of our language, which when combined together provide larger components.

1. All individual proposition letters, i.e., '$p$', '$q$', '$r$', ..., '$z$' are symbolic logic expressions.
2. If $W$ is a symbolic logic expression defined by only a single proposition letter, i.e., (1), then $\neg W$ is a symbolic logic expression.
3. If $W_1$ and $W_2$ are symbolic logic expressions, then
   (a) $(W_1 \lor W_2)$ is a symbolic logic expression,
   (b) $(W_1 \land W_2)$ is a symbolic logic expression,
   (c) $(W_1 \rightarrow W_2)$ is a symbolic logic expression,
   (d) $(W_1 \leftrightarrow W_2)$ is a symbolic logic expression.
4. If $W$ is a symbolic logic expression not defined by only a single proposition letter, i.e., (3), then $\neg W$ is a symbolic logic expression.

With this inductive definition and construction plan, we can also simplify certain expressions with redundant parentheses. For instance, we may omit the outer-most parentheses in an expression. As an example, '$((\neg s \land p) \leftrightarrow \neg(q \rightarrow \neg r))$' becomes '$(\neg s \land p) \leftrightarrow \neg(q \rightarrow \neg r)$' with this omission. Moreover, we often use the terms symbolic logic expression and schema interchangeably, which also holds true for their plurals: symbolic logic expressions and schemata.

### Truth Assignment and Truth Tables

What does it actually mean for two propositions to be joined by a connective? We have seen quite a few examples of translation from English to symbolic logic, but we need to ascribe a meaning to argument validity. We can do this through truth values. Recall when we stated that all propositions are either true or false; verity (truth) and falsity are representable via a construct known as a *truth table*. Suppose we have a proposition '$p$'; its possible truth values are either $\top$ or $\bot$. The left-hand column in the table below demonstrates this, whereas the right-hand column shows the result of an expression.

| $p$ | $p$ |
|---|---|
| $\top$ | $\top$ |
| $\bot$ | $\bot$ |

Figure 2.1: Truth Table of Proposition '$p$'

Obviously, with a single proposition, the right-hand column is rather meaningless. What if we decide to use two propositions with a connective? When given a slightly more complex expression such as '$p \vee q$', how do we know its truth value?

First, when drawing a truth table, it is imperative to draw $n + 1$ columns, where the first $n$ columns correspond to how many distinct propositions are used in the provided expression. Next, we need to have rows for each possible *assignment*, or interpretation, of truth values to each proposition. Because we only have two possible input values for each proposition, this directly correlates to the number of rows in a truth table, namely being $2^n$. Each row corresponds to a different, distinct truth assignment.

First, let us cover the simplest operator: unary negation. A negation flips all truth values of a proposition. Again, we use the '$\neg p$' to negate a proposition '$p$'. Though, if we have more than one proposition to negate, e.g., if we wanted to negate the expression '$(p \vee q)$', we place a large dash on the outside of said expression: '$\neg(p \vee q)$'. This, appropriately, flips all truth values of the inner, nested expression.

| $p$ | $\neg p$ |
|---|---|
| $\top$ | $\bot$ |
| $\bot$ | $\top$ |

Figure 2.2: Truth Table of '$\neg p$'

Now we move into binary operators; or those operators that use two operands. With logical conjunction, the expression is true if and only if both operands are true. The idea is that, when we have two propositions combined with the conjunction connective, we wish to express the idea that '$p$' is true at the same time that '$q$' is true.

| $p$ | $q$ | $p \land q$ |
|---|---|---|
| $\top$ | $\top$ | $\top$ |
| $\top$ | $\bot$ | $\bot$ |
| $\bot$ | $\top$ | $\bot$ |
| $\bot$ | $\bot$ | $\bot$ |

Figure 2.3: Truth Table of '$p \land q$'.

Logical disjunction, or alternation, is true if and only if at least one of its operands is true. The idea is that, when we have two propositions combined with $\lor$, we wish to express the idea that at least '$p$' ought to be true or '$q$' ought to be true. So, if they are both false, then this idea is not expressed.

| $p$ | $q$ | $p \lor q$ |
|---|---|---|
| $\top$ | $\top$ | $\top$ |
| $\top$ | $\bot$ | $\top$ |
| $\bot$ | $\top$ | $\top$ |
| $\bot$ | $\bot$ | $\bot$ |

Figure 2.4: Truth Table of '$p \lor q$'.

Logical conditionals are true if and only if the antecedent, i.e., the left-hand operand, is not true and the consequent is not false. Conditionals are a tricky subject for many people, as the idea behind their meaning is unintuitive at first glance, not to mention the conflation of logical implication versus the conditional. Let us examine the rows which evaluate to true before the false row.

"$\top \rightarrow \top$ evaluates to $\top$" should, hopefully, be self-explanatory. If we state that, when some proposition, i.e., the antecedent, is true, another proposition, i.e., the consequent, is also true, then it is true that if the antecedent is true, then the consequent holds true.

"$\bot \rightarrow \top$ evaluates to $\top$" is a little trickier. Let us examine this with an example using two propositions. "If $2 + 2 = 5$ then $3 + 3 = 6$". Clearly, the antecedent is false whereas the consequent is true. The expression, therefore, evaluates to true because the conditional dictates only what happens when the antecedent is true; not when it is false.

"$\bot \rightarrow \bot$ evaluates to $\top$" is even trickier since both operands are false yet the expression somehow results in truth. If a false antecedent implies a false consequent, this suggests that, even though both expressions are false, it does not falsify the entire conditional since the original antecedent never promised, so to speak, that the consequent is true when the antecedent is false. As we have therefore demonstrated, false implies anything, whether it be true or false, resolves to true.

We may now investigate the only false case of the conditional: "$\top \rightarrow \bot$ evaluates to $\bot$" is the second easiest truth assignment to understand. Our conditional is somewhat akin to a promise, and by the antecedent being true alongside the presence of a false consequent, we are suggesting that the antecedent does not, in fact, imply the consequent. This is an absurdity, which falsifies the implicative.

A phrase to know about the relationship between the conditional and logical implication is, "Implication is the validity of the conditional". That is, a schema implies another schema if the conditional asserts their validity.

| $p$ | $q$ | $p \rightarrow q$ |
|---|---|---|
| $\top$ | $\top$ | $\top$ |
| $\top$ | $\bot$ | $\bot$ |
| $\bot$ | $\top$ | $\top$ |
| $\bot$ | $\bot$ | $\top$ |

Figure 2.5: Truth Table of '$p \rightarrow q$'.

Finally, logical biconditional is also known as logical equivalence. This alternative name provides insight into its results; the expression '$p \leftrightarrow q$' is true if and only if '$p$' and '$q$' are identical in truth value.

| $p$ | $q$ | $p \leftrightarrow q$ |
|---|---|---|
| $\top$ | $\top$ | $\top$ |
| $\top$ | $\bot$ | $\bot$ |
| $\bot$ | $\top$ | $\bot$ |
| $\bot$ | $\bot$ | $\top$ |

Figure 2.6: Truth Table of '$p \leftrightarrow q$'.

Like the conditional, the relationship between equivalence and the biconditional is stated as, "Equivalence is the validity of the biconditional". So, two schemata are equivalent just in case the biconditional expresses valid of those schemata.

After combining these newfound truth tables together, we can evaluate the truth of any expression defined within our logic language. We first, however, need to establish a notion of the "main operator" of an expression.

### Main Operator

The *main operator* of a schema is the last operator that we investigate when performing a full truth analysis of the schema. Another definition is that the main operator is the first-parsed operator when evaluating a schema constructed through the inductive definition. What does this mean for us? Let us walk through an example to see.

Let us determine the main operator of '$\neg r \lor \neg((p \land q) \land (\neg r \leftrightarrow \neg p))$'. According to our rules for evaluation, we evaluate parenthesized expressions first. Inside the right-hand expression, i.e., '$\neg((p \land q) \land (\neg r \leftrightarrow \neg p))$', we see a conjunction of two expressions, one of which is another parenthesized expression, i.e., '$(\neg r \leftrightarrow \neg p)$'. The biconditional is broken down into two negated propositions, which resolve into their non-negated counterparts. If we start to rebuild this expression, we can quickly find the main operator. Combining the biconditional with the chain of conjunctions, i.e., '$p \land q$' gets us '$(p \land q) \land (\neg r \leftrightarrow \neg p)$'. This is then negated as '$\neg((p \land q) \land (\neg r \leftrightarrow \neg p))$'. Finally, we combine this with the disjunction operator where the left-hand operand is '$\neg r$'. Therefore the disjunction operator is the main operator.

### Evaluation Trees

If we view the previous schema as an *evaluation tree*, it is even easier to understand, since each operator is broken into separate components for evaluation. Each branch describes a subexpression which, when combined via traversing up the tree, produces either a valid sub-expression or the original expression.

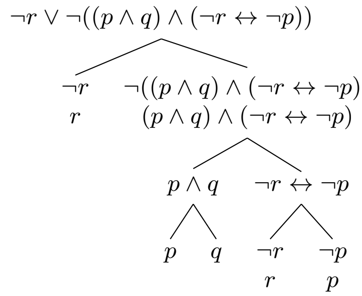
$\neg r \lor \neg((p \land q) \land (\neg r \leftrightarrow \neg p))$
$\neg r$
$\neg((p \land q) \land (\neg r \leftrightarrow \neg p))$
$r$
$(p \land q) \land (\neg r \leftrightarrow \neg p)$
$p \land q$
$\neg r \leftrightarrow \neg p$
$p$
$q$
$\neg r$
$\neg p$
$r$
$p$

Figure 2.7: Evaluation Tree of '$\neg r \lor \neg((p \land q) \land (\neg r \leftrightarrow \neg p))$'

Evaluation trees break down the structure of something into sub-components for further evaluation. In the context of zeroth-order logic schemata, we decompose a schema in one of two ways: if the formula is negated, its subsequent formula is stacked. Otherwise, it produces a branch into two further evaluation trees. In the above example, the main operator is $\lor$, meaning it is split into '$\neg r$' and '$\neg((p \land q) \land (\neg r \leftrightarrow \neg p))$'. The left-hand side decomposes into '$r$' because its main operator is $\neg$, indicating a stacked schema. After this, the left-hand parse tree can no longer decompose, meaning we proceed to the right-hand side. The schema '$\neg((p \land q) \land (\neg r \leftrightarrow \neg p))$', similarly, has a main operator of $\neg$, so we perform a similar operator removal. From there, the main operator is $\land$, indicating a branch with each sub-schemata. This process continues until all branches cannot be further reduced.

We could also describe evaluation trees via parse trees. *Parse trees* are fundamental structures not only in computer science, but also other fields such as (computational) linguistics, where parse trees provide the structure of sentences using parts-of-speech. Furthermore, parse trees express the contextual knowledge of the composition of an input schema. As an example, we might parse an arithmatic expression, e.g., '$2 + 4 \cdot 3$' as follows.[1]

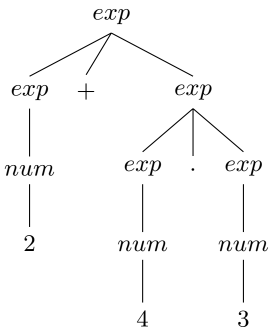

Figure 2.8: Parse Tree of '$2 + 4 \cdot 3$'

Expressions *exp* are comprised of two expressions and an operator in between. An *exp* can lead towards a *num*. In this example, the main operator so happens to be '+', meaning we add its left-hand expression 2 to its right-hand expression $4 \cdot 3$. Its resulting value is, therefore, 14. Let us consider another approach: what is stopping us from interpreting this as an expression whose main operator is multiplication? Nothing at all! Let us see the parse tree for an expression with multiplication as the last-to-evaluate expression instead.

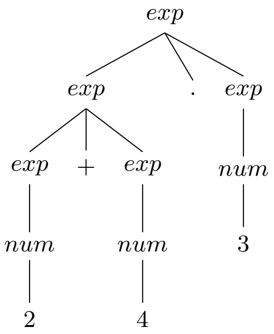

Figure 2.9: Ambiguous Parse Tree of '$2 + 4 \cdot 3$'

[1] The word "parse" stems from the Latin phrase "pars orationis".

From the above figures, we see that, if we change the order of evaluation, we get a completely different answer than the one that is most likely intended.[1] Evaluating the expression as such gives us a value of 24. In due time, parse trees will reappear, so if their purpose is not as clear at the moment as we hope, do not fret.

[1] "Intended" on the grounds that the standard order of operations, i.e., where multiplicatives are evaluated before additives, apply.

## 2.2 First-Order Logic

In the previous section, we created propositions that represented a collective idea or claim. We will quickly understand, however, that zeroth-order logic is rather weak in comparison to first-order logic. As a motivating example, suppose we want to represent the sentence, “All students go to college or school” in zeroth-order logic. There is really only one way to do so; because the disjunction connective exists, we can let $p$ : “All students go to college”, and $q$ : “All students go to school”. Symbolically, the schema is ‘$p \vee q$’. While this is feasible in our zeroth-order logic system, there is no possible way of adequately representing a *quantification* of variables, e.g., “All students”, and therefore, we lose a large chunk of the semantics behind our statement. First-order logic rectifies this with the introduction of predicates, as well as three new connectives: the universal quantifier, the existential quantifier, and the identity symbol.

A *predicate* is a statement that provides context to some input variable(s). For example, we can let ‘$S(x)$’ represent the idea that some ‘$x$’ is a student. Similarly, let ‘$C(x)$’ represent the idea that some $x$ go to college. Finally, let ‘$D(x)$’ represent the idea that some ‘$x$’ goes to school. Using predicates in this fashion, i.e., with only one input variable is known as monadic first-order logic. Later, we will analyze polyadic first-order logic, i.e., first-order logic where predicates use more than one input variable.

### The Quantifiers

*Quantifiers* are necessary in first-order logic for one reason: as their name suggests, they quantify, or provide numeric amounts to, some entity. There are two powerful quantifier connectives in first-order logic: the universal and existential quantifiers.

### Universal

If we want to say, “All math majors are smart”, then we are quantifying the proposition “math majors are smart” via “All”. In other words, every *thing* that is a math major has the property of “smart”. Additionally, we may claim that “No person who is not smart lives on Earth”, which quantifies the proposition “smart people exist” via “No”. In other words, anyone that lives on Earth is smart. We classify phrases and keywords as universal because they apply to an entire domain—nothing is excluded. In mathematical and logic contexts, the phrase, “For all”, or “For every” is often utilized as an English representation of the *universal quantifier*. As an example, “For all integers $x$ greater than one, $x^2$ is greater than $2x$”. If we wanted to instead use symbols, this may be represented as “$\forall x \in \mathbb{Z}$ such that $x > 1$, $x^2 > 2x$”.[1] This symbolage is not reserved only for the phrase “For all”; it is equivalently suited for “No”, “Every”, “None”, etc. One key rule-of-thumb is that, when quantifying with a universal over some domain, use an implication to indicate a relationship. We will show an example of where this falls apart soon. This may seem obvious to some, but while the universal quantifier is rather straightforward, it quickly descends into madness after introducing its existential counterpart. Before doing so, let us write a few examples of translating natural language universal quantifiers into schemata.

Suppose we want to translate, “All intelligent beings are either from Earth or from Mars”. We need three predicates: $I(x)$ meaning $x$ is an intelligent being, $E(x)$ meaning $x$ is from Earth, and $M(x)$ meaning $x$ is from Mars. We need to express that, anything we choose from our domain, i.e., what we quantify over, if that thing is an intelligent being, then they are either from Earth or Mars. Thus we translate the sentence into the schema ‘$\forall x(I(x) \rightarrow (E(x) \lor M(x)))$’.

Let us consider an example of where not using an implication gets us into trouble. Suppose we want to translate, “All cats and dogs can fly.” We need three predicates: $C(x)$ means $x$ is a cat, $D(x)$ means $x$ is a dog, and $F(x)$ means $F$ can fly. Attempting to translate this proposition using only a conjunction connective gets us the schema ‘$\forall x(C(x) \land D(x) \land F(x))$’, which expresses an incorrect proposition; namely that all things are cats and dogs and they fly. We, instead, should use the disjunction operator with an implication to express that, if something is either a cat or a dog, then it flies: ‘$\forall x((C(x) \lor D(x)) \rightarrow F(x))$’.

[1] The $\mathbb{Z}$ symbol denotes any integer. So, $\forall x \in \mathbb{Z}$ says, “for any integer” or equivalently “for every integer”.

### Existential

Unlike the universal quantifier whose scope is, as its name says, universal, the *existential quantifier* is simultaneously more and less specific. Take, for instance, "Not all cows have spots". It is important to understand that this claim does not say, "No cow has spots", nor something akin; it merely states that there exists a cow that does not have spots. What is more interesting about the existential quantifier is its power in representing propositions. Take, for instance, there exists an integer $x$ such that $x^2$ is greater than $2x$. Symbolically, $\exists x \in \mathbb{Z}$ such that $x^2 > 2x$. Existentials do not express as strong of a statement as those expressed by its universal counterpart; indeed, an existential '$\exists x P(x)$' states the existence of at least one $x$ that satisfies the predicate $P$. It does not, however, express that there is *only* one $x$, or exactly one $x$; the existential quantifier does not, in and of itself, denote an exact quantity of objects that satisfy a predicate. Therefore if we wanted to translate the proposition, "Most people are smart", we would use an existential quantifier via '$\exists x(P(x) \wedge S(x))$', where $S(x)$ means $x$ is smart and $P(x)$ means $x$ is a person. The determiners "Most", "Many", "Some", "A lot", "A", "An", "Any", all refer to the existential quantifier. "Any", however, becomes confusing, as does "every", when combined into a conditional. Let us see what this means via two examples.

First, suppose we want to schematize, "If Kate can solve any math problem, then she can solve every math problem". Of course, let $k$ represent Kate, $S(x, y)$ to mean $x$ solves $y$, and $M(x)$ to mean $x$ is a math problem. One may believe that the use of "any" refers to a universal quantifier, resulting in '$\forall x(M(x) \wedge S(k, x)) \rightarrow \forall x(M(x) \rightarrow S(k, x))$'. This, however, is a tautological statement, because when the antecedent is true, the consequent is never false. We could also equivalently schematize this as '$\forall x((M(x) \wedge S(k, x)) \rightarrow (M(x) \rightarrow S(k, x)))$', since '$\forall x(P(x) \rightarrow Q(x))$' is logically equivalent to '$(P(x) \rightarrow \forall x Q(x))$'. We should, instead, treat this use of "any" as an existential quantifier, meaning that the proposition would unambiguously express that, "If Kate can solve at least one math problem, then she can solve every math problem". Schematizing such a proposition results in an existential quantifier in the antecedent: '$\exists x(M(x) \wedge S(k, x)) \rightarrow \forall y(M(y) \rightarrow S(k, y))$'. Note that the use of "any" in the antecedent suggests an existential quantifier, but what happens if we place it in the consequent?

Suppose we want to schematize, "If Kate can solve every math problem, then she can solve any math problem." Using the same schematization of predicates, we translate this as '$\forall x(M(x) \rightarrow S(k, x)) \rightarrow \exists x(M(x) \rightarrow S(k, x))$', right? If we apply the same idea insofar as using an existential quantifier for "any", we express, yet again, a tautology, but not for the same reason as before. It is true and hopefully apparent that the schema '$\forall x P(x) \rightarrow \exists x P(x)$' is tautological, because if everything satisfies $P$, then by definition, there is something that satisfies $P$. So, using an existential in the consequent leads us to a weaker assertion than our original desired proposition. We want to express that, because Kate can solve every math problem, then she can solve *any* math problem, where *any* means that, given an arbitrary math problem, Kate solves that problem. It should be clear that this is expresses the same proposition as the antecedent, leading us to instead use a universal quantifier in the consequent: "$\forall x(M(x) \wedge S(k, x)) \rightarrow \forall x(M(x) \wedge S(k, x))$".

### Identity

Suppose we have two constants in our domain representing the individuals Alan Turing and Katherine Johnson as $a$ and $k$ respectively. We can definitively state that $a$ and $k$ do not refer to the same entity, and symbolically, we say '$\neg(a = k)$'. Perhaps we want to represent the North Star as $n$ and Polaris as $p$, and because we know that the North Star is the same thing as Polaris, we write '$n = p$'.

*Identity* is a two-place predicate amongst variables and constants, as opposed to symbolic logic expressions like the other (unary and binary) connectives. Let us use identity inside a schema to show how it works: imagine we want to express the proposition that anyone who is the best computer scientist, is Katherine Johnson. To do so, we need a predicate $C(x)$ to indicate that $x$ is a computer scientist, a predicate $B(x, y)$, denoting that $x$ is better than $y$.[1] Our schema is, therefore, '$\forall x(C(x) \to \forall y((C(y) \land B(x, y)) \to x = k))$'.

Identity is also useful for determining uniqueness amongst objects in our domain. For instance, to say that there is at most one computer scientist, then we write the following schema: '$\forall x \forall y((C(x) \land C(y)) \to x = y)$', which suggests that any computer scientists that we choose from our domain must refer to the same computer scientist.

To say that there are at least two computer scientists, then we need to instead use existential quantification: '$\exists x \exists y((C(x) \land C(y)) \land \neg(x = y))$', which suggests that there are at least two computer scientists that we can poll from our domain that do not refer to the same computer scientist.

Combining these ideas together allows us to represent *exactness*; suppose we wish to express that there are exactly two computer scientists. We might use the following schema: '$\exists x \exists y((C(x) \land C(y)) \land \neg(x = y)) \land \forall z(C(z) \to (x = z \lor y = z))$', which first states that there are at least two computer scientists, but follows this with a restriction stating that any computer scientist chosen from our domain must be (identical to) either $x$ or $y$.

Finally, let us translate the proposition "Alan Turing is a computer scientist who is smarter than Alonzo Church, but Katherine Johnson is the smartest" using predicates $S(x, y)$ to represent that $x$ is smarter than $y$, and $c$ to represent Alonzo Church. We schematize this proposition as '$(S(a, c) \land C(a)) \land \forall x(\neg(x = k) \to S(k, x))$' (note that we do not express the propositions that Alonzo Church or Katherine Johnson are computer scientists).

### Inductively Defining First-Order Logic

Much like our definition for zeroth-order logic, we can inductively define first-order logic schemata. Note that our first-order logic system uses constants and variables, whereas some textbooks and introductions to the material omit constants altogether.

[1] There are, perhaps, arguments against our interpretation of someone being better than everyone else to mean they are the best.

1. Variables are defined as $t, u, v, \ldots, z, t', \ldots, z'$, but are themselves not symbolic logic expressions.

2. Constants are defined as $a, b, c, \ldots, s, a', \ldots, s'$, but are themselves not symbolic logic expressions.

3. All predicates, i.e., $A, B, C, \ldots, Z, A', \ldots, Z'$, followed by one or more variables or constants wrapped in parentheses and commas are symbolic logic expressions.

4. If $W$ is a symbolic logic expression, then $\neg W$ is a symbolic logic expression.

5. If $W$ is a symbolic logic expression and $\omega$ is a variable, then

   (a) '$\forall \omega W$' is a symbolic logic expression.

   (b) '$\exists \omega W$' is a symbolic logic expression.

6. If $W_1$ and $W_2$ are symbolic logic expressions, then

   (a) $(W_1 \vee W_2)$ is a symbolic logic expression.

   (b) $(W_1 \wedge W_2)$ is a symbolic logic expression.

   (c) $(W_1 \rightarrow W_2)$ is a symbolic logic expression.

   (d) $(W_1 \leftrightarrow W_2)$ is a symbolic logic expression.

7. If $w_1$ and $w_2$ are either variables or constants as defined in either (1) or (2), then $(w_1 = w_2)$ is a symbolic logic expression.

Once again, as we did with zeroth-order logic, we may drop the outer-most parentheses of an expression given that it does not ambiguate said expression. For instance, the schema '$(\forall x(\forall y(P(x) \rightarrow \neg Q(y))) \vee \exists z Q(z))$' becomes '$\forall x(\forall y(P(x) \rightarrow \neg Q(y))) \vee \exists z Q(z)$'. The scope of a quantifier extends over the schema to its immediate right, meaning that the $x$ bound by '$\forall x$' has scope over the sub-schema '$\forall y(P(x) \rightarrow \neg Q(y))$'. Consequently, we could remove the parentheses surrounding the '$\forall y$' quantifier, producing a semantically-equivalent schema: '$\forall x \forall y(P(x) \rightarrow \neg Q(y)) \vee \exists z Q(z)$'.

## 2.3 Sets

*Sets* are a fundamental construct in computer science, mathematics, and many other fields. A *set* is a collection of objects, or elements, with arbitrary ordering.

Sets are delimited by braces, i.e., $\{\ldots\}$. Objects within a set are delimited by commas, i.e., $\{\ldots,\ \ldots\}$. For example, $A = \{9, 8, 1, 4\}$, $B = \{a, 2, 54, bd, e\}$, $C = \{\{7, 8, 9\}, -6, \{1, a, q\}, 9\}$ are all *proper sets* because they do not contain duplicate elements. $D = \{12, 12, -6, 80\}$ is what we will denote as an *improper set*, because it has duplicates. A set that contains duplicates is equivalent to the same set with all duplicates removed. For instance, $\{9, 8, 12, 12, 4, 9, 10, 12\}$ is equivalent to $\{9, 8, 12, 4, 10\}$.

We say that an element $x \in S$ if $x$ is a member of $S$. With the preceding example, $9 \in A$, $\{12, abcd, qrst\} \in C$, but $10 \notin B$.

The size, or *cardinality*, of a set $S$ is denoted as $|S|$. For example, in the preceding examples, $|A| = 4$, $|B| = 5$, and $|C| = 4$. Elements within a nested set of a set, as exemplified with set $C$, do not count towards the cardinality of a set. Duplicate elements also do not affect a set's cardinality; the set $E = \{a, a, a, a, a\}$ has $|E| = 1$ because there is only one distinct element, namely $a$, in $E$. Accordingly we also categorize $E$ as improper.

A set can be empty, as denoted by the *empty set*, i.e., $\varnothing$. In other words, there does not exist an element $x$ that is a member of $\varnothing$. Symbolically, $\forall x,\ x \notin \varnothing$.

$S'$ is a *subset* of some set $S$ if and only if every element of $S'$ is a member of $S$. Symbolically, we represent subset as $S' \subseteq S$, and we can formulate our definition as '$\forall S \forall S'(S' \subseteq S \leftrightarrow \forall x((x \in S') \wedge (x \in S)))$'. $S''$ is a *proper subset* of $S$ if and only if every element of $S''$ is a member of $S$, but $S'' \neq S$. In other words, $S''$ is not the same set as $S$. Symbolically, we represent this idea as '$\forall S \forall S''(S'' \subset S \leftrightarrow \forall x(x \in S'') \wedge (x \in S) \wedge (S'' \neq S))$'. A way of remembering the difference comes through the appearance of the symbol. In algebra, we represent, "$x$ is less than or equal to $y$" as $x \leq y$ with the bar underneath. The same idea holds for subsets; we represent "$S'$ is a proper subset or equal to $S$" as "$S' \subseteq S$".

Two sets are *equivalent* if and only if they share the same elements. Note that this definition does not account for element ordering/positioning and duplicate elements, because these properties are irrelevant when working with sets. For instance, with the previous example, $D$ is equivalent to the set $D' = \{12, -6, 80\}$, which is equivalent to the set $D'' = \{80, 12, -6\}$. We symbolize this as, '$\forall A \forall B(\forall x(x \in A \wedge x \in B) \rightarrow A = B)$'. To clarify, this says, for any two sets $A$ and $B$, $A$ is equal to $B$ if every element $x$ is both a member of $A$ and $B$. Interestingly, we can create another definition using subsets to define equality as follows: '$\forall A \forall B((A \subseteq B) \wedge (B \subseteq A) \rightarrow A = B)$'.

We say $S$ is the *union* of two sets $A$ and $B$ if and only if $S$ contains all elements that are members of $A$ or members of $B$. We use the "cup" to represent set union, i.e., $A \cup B$. Formalizing the definition, '$\forall A \forall B \forall S(\forall x((x \in S) \leftrightarrow (x \in A \vee x \in B)) \rightarrow S = A \cup B)$'. For example, let $A = \{5, 6, 7, 8, 9\}$ and $B = \{3, 4, 5, 6, 7\}$. Thus, $A \cup B = \{3, 4, 5, 6, 7, 8, 9\}$. Note that it is impossible for the union of two sets to be the empty set as long as $|A| + |B| > 0$.

We say $S$ is the *intersection* of two sets $A$ and $B$ if and only if $S$ contains all elements that are members of both $A$ and $B$. We use the "cap" to represent set intersection, i.e., $A \cap B$. Formalizing the definition, '$\forall A \forall B \forall S (\forall x((x \in S) \leftrightarrow (x \in A) \land (x \in B)) \to S = A \cap B)$'. For example, let $A = \{a, b, c, d, e, f\}$, and $B = \{q, r, s, c, d, t, u\}$. Thus, $A \cap B = \{c, d\}$. Another example is, let $C = \{12, 340, \{q, r, s\}, \{\{900\}\}\}$, and $D = \{\{q, s, r\}, \{\{900\}\}, 341\}$. Thus, $C \cap D = \{\{900\}\}$. Some may find it confusing that the nested set $\{q, r, s\}$ is not part of the intersection. Recall that the definition is that $\{q, r, s\}$ from $C$ must be a member of $D$, which it is not, as $\{q, s, r\} \in D$. Finally, one more example is let $E = \{a, b, c\}$, and $F = \{d, e\}$. Thus, $E \cap F = \varnothing$. It should, hopefully, be apparent that the intersection of any set with the empty set always results in the empty set.

We say $S$ is the *difference* of two sets $A$ and $B$ if and only if $S$ contains all elements that are in $A$ but not in $B$. We use the backslash, i.e., $\setminus$, to represent set difference, i.e., $A \setminus B$. Formalizing the definition, '$\forall A \forall B \forall S (\forall x((x \in S) \leftrightarrow (x \in A) \land (x \notin B)) \to S = A \setminus B)$'. For example, let $A = \{a, b, c, d\}$, and $B = \{b, c, d, e\}$. Thus, $A - B = \{a\}$. Another example is, let $C = \{q, r\}$, and $D = \{q, r, s, t, u, v\}$. Thus, $C - D = \varnothing$. Finally, one more example is, let $E = \{1, 2, \{3, 4\}, 5\}$, and $F = \{\{4, 3\}, 2, 6\}$. Thus, $E - F = \{1, \{3, 4\}, 5\}$.

## Common Mathematical Sets

There are several popularized sets in mathematics, each being referenced by a specific special symbol.

### Integers

The set of *integers*, i.e., $\mathbb{Z}$, has all positive and negative whole numbers, including zero. Thus $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. The set of integers has no limit on either side and continues forever.

### Natural Numbers

The set of *natural numbers*, i.e., $\mathbb{N}$, contains all positive integers and zero. In computer science, we generally define $\mathbb{N}$ as $\mathbb{N} = \{0, 1, 2, \ldots\}$. Defining the natural numbers like this is controversial, since some mathematicians consider zero to not be a natural number. A way around this approach is to consider the set of non-zero positive integers, i.e., $\mathbb{N}^+$ or even $\mathbb{Z}^+$.[1]

### Rational Numbers

The set of *rational numbers*, i.e., $\mathbb{Q}$, contains all numbers that are representable as a ratio $\frac{p}{q}$ for some integers $p$ and $q$.

[1] It is provable that mathematicians can never be pleased no matter the compromise.

### Real Numbers

The set of *real numbers*, i.e., $\mathbb{R}$, contains all numbers that we know, including $\pi$, $e$, and is the only set, out of the four that we have described, which is *uncountably infinite*.[1] In essence, this means that we cannot find an end to the representation of a number. To exemplify this idea, imagine we have the real number 0.01. By the definition of $\mathbb{R}$, we also have 0.00000000000001 as a real number. But we also have $\frac{1}{10^{9999999999}}$ as a real number. We can always add a zero to the decimal representation making the number smaller than it was previously. The general idea of *countably infinite* sets versus uncountably infinite sets is to create a pairing between a set and the natural numbers. For instance, we can create a mapping between $\mathbb{Z}$ and $\mathbb{N}$ by mapping all positive integers in $\mathbb{Z}$ to odd natural numbers in $\mathbb{N}$, and map all negative integers in $\mathbb{Z}$ to even natural numbers in $\mathbb{N}$. Because we can create this correspondence, we can formally prove that the set of integers is countably infinite. On the contrary, we cannot do this with the set of real numbers, because no matter what possible mapping we attempt, even if we mapped each added zero to $0.00\ldots01$ to a natural number, we would never even get past this "first" real number. The notion of creating a correspondence, or map, between sets is described in further detail in the next section.

[1] By the phrase "that we know", we mean to exclude the set of imaginary numbers.

## 2.4 Functions

Functions are often introduced to students in middle or high school. Though, it is rare that curricula at this level discuss the theory behind what general function actually does. As an example, we can denote the square root of a number $x$ via a function $\mathsf{Sqrt}$.

A *function*, as an informal reintroduction, is a construct that receives some input, performs some operation, and returns an output. For example, we can write a function $f(x) = x^2$ to square a number $x$. In this definition, $f$ is the function name, $x$ is the input argument, and the body, i.e., the operation to-be performed is $x^2$. The data returned is the result after applying, or unwrapping, the value of $x$ inside the operation. We can proceed through this derivation step-by-step to understand it better.

$$f(x) = x^2 \tag{2.1}$$

$$f(3) = x^2 \tag{2.2}$$

$$f(3) = 3^2 \tag{2.3}$$

$$f(3) = 9 \tag{2.4}$$

Line (2.1) simply repeats our function name, input variable name, and function body for convenience. Line (2.2) invokes a call to $f$ with an input of 3. Line (2.3) shows the process of applying the values of the input variable $x$ to every occurrence of $x$ in the function body. Lastly, in (2.4), we compute the result of the expression.

### Domain, Codomain, and Range

Functions are maps over sets. We *map* an element $x$ from a set $A$ to an element $y$ of a set $B$ if $f(a) = b$ for a function $f$. Let us narrow our set scope for a minute to get a better picture of function properties. The *domain* $D$ of a function $f$ is a set of its possible inputs. The *codomain* $D'$ of a function $f$ is a set of its possible outputs. The *range* $R$ of a function $f$ is the set of mapped values from $D$ to $D'$. That is, if there exists $x$ and $y$ such that $f(x) = y$, then $y \in D'$. Let us see an example before continuing further.

*Example.* Let $D = \{a, b, c\}$ and $D' = \{1, 2, 3, 4\}$. Further suppose that a function $g$ maps values from $D$ to $D'$, namely as $g(a) = 2$, $g(b) = 4$, $g(c) = 2$. The range of $g$ is therefore $\{2, 4\}$.

### Image and Pre-image

The *image* of a function $f$, denoted as $f(I)$, is a set such that, when given a set $I \subseteq D$, we get all elements mapped to those in $I$. The *pre-image* of a function $f$, denoted as $f^{-1}(I)$, is a set such that, when given a set $I \subseteq D'$, we get all mapping values from $D$. Again, we present an example.

*Example.* Using the definitions of $D$ and $D'$ from earlier, we can deduce multiple images and pre-images.

*Possible images:*

$$\begin{aligned} f(\{a\}) &= \{2\} \\ f(\{b, c\}) &= \{4, 2\} \\ f(\{d\}) &= \varnothing \end{aligned}$$

*Possible pre-images:*

$$\begin{aligned} f^{-1}(\{1, 2, 3\}) &= \{a, c\} \\ f^{-1}(\{1, 4\}) &= b \\ f^{-1}(\{1, 3\}) &= \varnothing \end{aligned}$$

A good rule-of-thumb for images is to ask, "For these values of $D$, what values do I get from $D'$?", and for pre-images we ask, "For these values of $D'$, what values map them from $D$?".

Importantly, functions must map one value to only one other value. Therefore a function $f$ cannot map an input $x$ to distinct outputs $y$ and $y'$.

For a "square" function, its domain might be $\mathbb{Z}$ with its codomain as $\mathbb{N}$. The "square root" function, on the other hand, is a function only if we limit its codomain to the set of real numbers greater than or equal to zero because $\sqrt{x}$ maps input elements to multiple output elements in that its result can be either positive or negative. E.g., $\sqrt{25} = \pm\, 5$. Note that its domain might be the set of real numbers greater than or equal to zero; negative real numbers are undefined for the square root function.

The most commonly-presented functions throughout primary/secondary education are unary functions, i.e., functions of one input such as $f(x)$. Nothing is stopping us from defining a function that has multiple inputs. As an example, let $h(a, b) = 5$, where $a, b \in D$ and $5 \in D'$. Functions of two arguments are binary, and use ordered pairs to denote their inclusion in the domain. The set of possible inputs is defined as the cross product $D \times D \to D'$. We use binary functions almost, if not, daily without realizing. Suppose we define addition as a binary function $\mathsf{Add}$ over the set of natural numbers. We know that $\mathbb{N} \times \mathbb{N} = \{(0, 0), (0, 1), (0, 2), \ldots\}$, and can define $\mathsf{Add}(m, n) = m + n$. Of course, we rely on the definition of addition from the $+$ symbol, but this simplification helps our discussion, since we can conclude that $m + n \in \mathbb{N}$. Ternary functions are also possible, requiring three inputs rather than one or two. We could go on, but any non-negative number of inputs to a function is definable.

### Operations and Properties

Functions are sometimes called operations when they exhibit certain properties. Addition, or $+$, is an associative and commutative binary operation over the set of real numbers. An operation $\circ$ is *associative* over a set $S$ if, for all elements $x, y, z \in S$, $x \circ (y \circ z)$ is equal to $(x \circ y) \circ z$. For instance, if $x, y, z \in \mathbb{Z}$, $(x + y) + z$ is equal to $(x + (y + z))$. An operation $\circ$ is *commutative* over a set $S$ if, for all elements $x, y \in S$, $x \circ y$ is equal to $y \circ x$. For instance, if $x, y \in \mathbb{Z}$, $x + y$ is equal to $y + x$. Another example of an associative and commutative operation is multiplication over the set of real numbers. Subtraction, on the other hand, is neither commutative nor associative over the set real numbers. It is, however, both associative and commutative over the set $\{x, x, x, \ldots\}$ for any real number $x$. We can change the properties of an operation by modifying the set it is over. Classifying properties of operations in this way is not as interesting, since we most often care about larger sets such as the natural numbers, integers, or reals. We can categorize an operation $\circ$ and a set $S$ as a *group* if $\circ$ is associative, there exists an element $e$ such that $e \circ x = e$ for all $x \in S$, and there is a (not necessarily distinct) $y$ such that $x \circ y = e$ for all $x \in S$. As an example, we can form a group $G$ as $+$ over $\mathbb{Z}$ because addition is associative, any integer added to zero gets us that integer, and every integer has a negative counterpart we can add to get zero. Group theory and abstract algebra present these topics in far greater detail, so we will end the discussion here.

### Recursive Functions

Recursive functions, and the idea behind recursion confuses many students and beginners to computer science, but a proper understanding is fundamental to programming and mathematics. A *recursive function* is a function $f$ that calls itself. For example, the simplest recursive function may be defined as $f() = f()$, which states that $f$ is a function that receives no arguments and calls itself. When we call a recursive function, we substitute the call with the function body. This is a bit difficult to visualize with a function of no arguments, so why not introduce a function $g(x)$ that receives some argument $x$ whose definition adds $x$ to a recursive call to $g$, i.e., $g(x) = x + g(x)$. When we evaluate the recursion, we see this resolves to $g(x) = x + x + x + \cdots + x$, for an infinite number of $x$'s. Though, what is interesting about recursion is that we can define primitive operations, such as addition, via recursion. This form of recursion will be over the set of natural numbers. Hence we refer to recursive functions over $\mathbb{N}$ as *naturally-recursive* functions.

When we add two numbers, say, 3 and 2, we certainly know that their sum is 5. Though, what if we did not know how to add two numbers? Suppose that all we were given are two functions $\mathsf{add1}(x)$ and $\mathsf{sub1}(x)$, where $\mathsf{add1}(x)$ returns the next natural number after $x$, namely $x + 1$. $\mathsf{sub1}(x)$, on the contrary, returns the previous natural number before $x$, namely $x - 1$.[1] Assuming that these are the only two possible ways we can add numbers, we can write the function, $x + y$, using recursion.

[1] The range of the $\mathsf{sub1}$ function is only positive integers; meaning $\forall n \leq 1$, $\mathsf{sub1}(n) = 0$.

Because of our prior knowledge on elementary arithmetic, we can say, with absolute certainty, that $3+2$ is equal to $4+1$ which is equal to $5+0$. Conveniently enough, we know that $x + 0 = x$ for any number $x$. We can use this to determine a *terminating condition*, i.e., when to stop recursing. That is, when $y = 0$, we stop recursively adding values, since $x + 0$ is just $x$. Let us walk through the example step-by-step.

(i) Given $3+2$, we know that, with $x = 3$ and $y = 2$, $y$ is not zero, so we can add one to $x$ and subtract one from $y$. Namely, $\mathsf{add1}(3) + \mathsf{sub1}(2) = 4 + 1$.

(ii) We now have $4 + 1$, we know that, with $x = 4$ and $y = 2$, $y$ is not zero, so we can add one to $x$ and subtract one from $y$. Namely, $\mathsf{add1}(4) + \mathsf{sub1}(1) = 5 + 0$.

(iii) We now have $5 + 0$, we know that, with $x = 5$ and $y = 0$, $y$ is equal to zero, so we simply return $x$, which is 5. Therefore, the correct result is obtained.

So, we continuously subtract one from $y$ and continuously add one to $x$ until $y$ is zero. We can define this as a piece-wise function (recall the definition of such functions from the previous sections) where $\mathsf{add}(x, y)$ corresponds to $x + y$.

$$\mathsf{add}(x,\, y) = \begin{cases} x & \text{if } y = 0 \\ \mathsf{add}(\mathsf{add1}(x), \mathsf{sub1}(y)) & \text{if } y > 0 \end{cases}$$

As another example, we will add 10 and 4 using this recursive algorithm. Though, we will be a bit less verbose and not explicitly mention the values for $x$ and $y$.

(i) We have $10 + 4$. Clearly, $y$ is not zero, so we, instead, compute $\mathsf{add1}(10) + \mathsf{sub1}(4) = 11 + 3$.

(ii) We now have $11+3$. Clearly, $y$ is not zero, so we compute $\mathsf{add1}(11) + \mathsf{sub1}(3) = 12 + 2$.

(iii) We have $12 + 2$. Clearly, $y$ is not zero, so we compute $\mathsf{add1}(12) + \mathsf{sub1}(2) = 13 + 1$.

(iv) We now have $13+1$. Clearly, $y$ is not zero, so we compute $\mathsf{add1}(13) + \mathsf{sub1}(1) = 14 + 0$.

(v) We now have $14 + 0$. Clearly, $y$ is zero, so we return $x$, which is 14.

With addition out of the way, let us implement monus. No, that is not a spelling mistake—monus is a subtraction-like operation defined only for natural numbers. That is, $x \dot{-} y$ is defined only for those $x$ that are greater than or equal to $y$. Thus, $x \dot{-} y \geq 0$. We need this notion of monus, and not minus, in order to write a recursive algorithm because again, we need a terminating condition. We know that $x \dot{-} 0 = x$, so we can use this as our terminating condition. Namely, when $y = 0$, return $x$. Otherwise, call the function recursively. How can we do this using only $\mathsf{add1}$ and $\mathsf{sub1}$? Well, firstly, we need only to use $\mathsf{sub1}$ because we know that, for instance, $5 \dot{-} 3$ is equal to $4 \dot{-} 2$ which is equal to $3 \dot{-} 1$, which is equal to $2 \dot{-} 0$. So, instead of adding one to $x$ as part of the recursive step, we subtract one from both operands and only stop once $y$ reaches zero. Let us walk through the example step-by-step.

(i) Given $5 \dot{-} 3$, we know that $y$ is not zero, so we can subtract one from $x$ and subtract one from $y$. Namely, $\mathsf{sub1}(5) \dot{-} \mathsf{sub1}(3) = 4 \dot{-} 2$.

(ii) We now have $4 \dot{-} 2$. $y$ is not zero, so we can subtract one from both $x$ and $y$. Namely, $\mathsf{sub1}(4) \dot{-} \mathsf{sub1}(2) = 3 \dot{-} 1$.

(iii) We now have $3 \dot{-} 1$. $y$ is not zero, so we can subtract one from both $x$ and $y$. Namely, $\mathsf{sub1}(3) \dot{-} \mathsf{sub1}(1) = 2 \dot{-} 0$.

(iv) We now have $2 \dot{-} 0$. $y$ is zero, so we can return 2.

This completes the example. Let us define monus as a recursive piecewise function similar to our approach to addition. We will use $\mathsf{sub}(x\ y)$ to represent $x \dot{-} y$.

$$\mathsf{sub}(x, y) = \begin{cases} x & \text{if } y = 0 \\ \mathsf{sub}(\mathsf{sub1}(x), \mathsf{sub1}(y)) & \text{if } y > 0 \end{cases}$$

What about multiplication over natural numbers? We can actually use our definition of addition in a definition for a recursive multiplication function. Let us take an example. Once again, because of our prior knowledge, we know that $5 \cdot 3$ is equal to 15. Though, we can represent this as follows. $5 \cdot 3 = 5 + (5 \cdot 2)$ which is equal to $5 + 5 + (5 \cdot 1)$ which is equal to $5 + 5 + 5$. So, we see that multiplication is nothing more than repeated addition. Instead of zero, however, we use one as our terminating condition, because $x \cdot 1 = x$. Thus, when $y = 1$ in $x \cdot y$, we return $x$. Let us walk through the example step-by-step.

(i) Given $5 \cdot 3$, we know that $y$ is not one, so let us add $x$ to the result of multiplying $x$ by $\mathsf{sub1}(y)$. Namely, $5 + (5 \cdot \mathsf{sub1}(3)) = 5 + (5 \cdot 2)$.

(ii) We now must compute $5 \cdot 2$. $y$ is not one, so let us add $x$ to the recursive function call. Namely, $5 + (5 \cdot \mathsf{sub1}(2)) = 5 + (5 \cdot 1)$

(iii) We now must compute $5 \cdot 1$. $y$ is clearly one, so we return 5.

At this point, we start a process called recursion unwinding. That is, we have the result of the base case, but we need to substitute these values in for the previous recursive calls. We evaluate these "recursive unwinds" from bottom-to-top.

(i) 5 is the base case for $5 \cdot 1$. We substitute 5 in for this expression in $5 + (5 \cdot 1)$ to get $5 + 5 = 10$, which is the value of $(5 \cdot 2)$.

(ii) We can substitute 10 in for the expression $5 + (5 \cdot 2)$ to get $5 + 10 = 15$, which is the value of $(5 \cdot 3)$, which is our answer.

With this example, we can write our recursive piece-wise definition for multiplication in terms of addition. We will use $\mathsf{mult}(x,\, y)$ to represent $x \cdot y$, or $xy$ without the explicit symbol:

$$\mathsf{mult}(x, y) = \begin{cases} x & \text{if } y = 1 \\ \mathsf{add}(x, \mathsf{mult}(x, \mathsf{sub1}(y))) & \text{if } y > 1 \end{cases}$$

Now that we have explained recursion for simple arithmetic functions, we can move on to slightly harder concepts found in subsequent chapters. We will revisit and reintroduce natural recursion in Chapter 5 with the added benefit of implementing these rules in a programming language!

## 2.5 Proofs

What is a proof? A *proof* is a sequence of logical deductions from premises to a conclusion. When we prove something, we state facts in an attempt to convince ourselves, or others, that the conclusion is true. There are several methods of proof, and we will go through many examples.

When writing proofs, it is important to be diligent and careful with explanations. Similarly, stating things that are perhaps obvious to some, is always a good idea, e.g., definitions of even/odd numbers. In addition, all proofs should begin with *Proof:* and conclude with "QED", or a square □. QED, or the square, symbolizes the Latin phrase, "Quod erat demonstrandum", which translates to, "which was to be stated". In other words, it designates the end of a proof.[1]

### Direct Proofs

A direct proof takes the form of an implication, namely, "If $p$ then $q$". When we prove such statements directly, we assume the antecedent $p$, and with this, we attempt to show that the consequent $q$ cannot be false.

*Example.* "If $x$ is even, then $(x)^2$ is even".

*Proof.* First, we assume the antecedent, namely "$x$ is even", is true. Thus, $x$ is an even number of the form $2k$ for some integer $k$. Now, we need to show that $(x)^2$ is also even. Let us plug in $2k$ for $x$ to get $(2k)^2$. Expanding this out, we get $(2k \cdot 2k)$, which after factoring, we get $2(k + k)$. If we let $l = (k + k)$, we can substitute the parenthesized expression for $l$ to get $2l$, which by definition is an even number. Therefore, if $x$ is even, then $(x)^2$ is even. QED.

*Example.* "If $x$ and $y$ are odd integers, then $x + y$ is even".

*Proof.* Assume that $x$ and $y$ are odd, meaning they are of the form $x = 2k + 1$ and $y = 2m + 1$ for some integers $k$ and $m$. Substituting these in for $x + y$ gets us $(2k + 1) + (2m + 1) = 2k + 2m + 2$. Factoring gets us $2(k + m + 1)$. If we let $l = k + m + 1$, we can substitute the parenthesized expression for $l$ to get $2l$, which by definition is an even number. Therefore, if $x$ and $y$ are odd integers, then $x + y$ is even. QED.

*Example.* "The square of an odd integer is always odd".

*Proof.* Let us turn this statement into a conditional: "If $x$ is an odd integer, then $(x)^2$ is odd". Now, assume $x$ is odd, meaning $x = 2k + 1$ for some integer $k$. Thus, $(x)^2 = (2k+1) \cdot (2k+1) = 4k+4+1$. Factoring this result gets us $2(2k+2)+1$. Let $l = 2k + 2$, which after substituting gets us $2l + 1$, the definition of an odd integer. Thus, if $x$ is an odd integer, then $(x)^2$ is odd. QED.

[1] Formatting a proof in this fashion is largely a stylistic choice; as long as the argument and reasoning are clear, any variation is acceptable.

*Example.* "If $a \mid b$ and $b \mid c$, then $a \mid c$".

Recall the definition of $a \mid b$: $a \mid b$ means that $ax = b$ for some integer $x$. In other words, we can evenly divide $b$ by $a$ to get some integer $x$. Example: $3 \mid 6$ because $3(2) = 6$.

This proof asks us to prove the transitive property of division: if we can divide some number $b$ by $a$, and we can further divide some number $c$ by $b$, then we can divide $c$ by $a$, i.e., $a$ is a multiple of both $b$ and $c$.

*Proof.* Assume that the antecedent is true, indicating $a \mid b$ and $b \mid c$. This claim means that $ax = b$ and $by = c$ for some integers $x$ and $y$. We can substitute the value of $b$ in $by = c$ with the former equation, i.e., $ax = b$, as follows: $(ax)y = c$. Now, because multiplication is associative, $(ax)y = a(xy)$. Thus, we get the form $a(xy) = c$, which, if we let $l = xy$, we get $al = c$. An equivalent representation is $a \mid c$. Therefore, if $a \mid b$ and $b \mid c$, then $a \mid c$. QED.

*Example.* "If $x \mid y$ and $y$ is odd, then $x$ is odd".

*Proof.* We will prove that the two conditions $x \mid y$ and $y$ imply that $x$ is odd. First, assume the antecedent, which means that $xm = 2k + 1$ for some integers $m$ and $k$. This implies that the product of $x$ and $m$ results in an odd integer. The product of two integers is odd only when one of its operands is odd. Therefore $x$ must be odd. Hence, if $x \mid y$ and $y$ are odd, then $x$ is odd. QED.

### Proof by Contraposition

A conditional may be easier to prove if we use its contrapositive. That is, recall that, '$p \to q$' is equivalent to '$\neg q \to \neg p$'. This equivalence is particularly useful when the antecedent of an implication is complex and full of schemata.

*Example.* 'If $x$ and $y$ are integers and $x + y$ is even, then $x$ and $y$ have the same parity".

We will prove this by contraposition. Namely, the contrapositive of the statement is, "If $x$ and $y$ do not have the same parity, then $x + y$ is odd" (note that we do not negate the piece of the premise that states that $x$ and $y$ are integers).

*Proof.* Assume, by contraposition, $x$ and $y$ do not have the same parity. That is, one of the values is even and the other is odd. Without loss of generality, we can assume that if the consequent holds for when $x$ is even and $y$ is odd, we can conclude that it holds for when $x$ is odd and $y$ is even. So, assume $x$ is even and $y$ is odd, meaning $x = 2k$ and $y = 2l + 1$ for some integers $k$ and $l$. Let us now substitute in these values for $x$ and $y$ in $x + y$: $(2k) + (2l + 1) = 2k + 2l + 1$. Factoring out 2 gets us $2(k + l) + 1$. If we let $m = k + l$, we get $2m + 1$, which is the definition of an odd integer. Therefore, by contraposition, if $x$ and $y$ are integers and $x + y$ is even, then $x$ and $y$ have the same parity. QED.

*Example.* "If $x$ and $y$ are real numbers where $xy$ is irrational, then either $x$ or $y$ must be irrational".

*Proof.* Assume, by contraposition, that neither $x$ nor $y$ are irrational. This means that we can write $x = p/q$ and $y = r/s$ where the fractions $p/q$ and $r/s$ are rational numbers written in their lowest terms. We wish to show that $xy$ is rational. We can substitute in our fractions for $xy$ to get $(p/q)(r/s) = pq/rs$ where $q \neq 0$ and $s \neq 0$. Thus, we can represent the product of $x$ and $y$ as a fraction, meaning it is rational. Therefore, by contraposition, if $x$ and $y$ are real numbers where $xy$ is irrational, then either $x$ or $y$ must be irrational. QED.

### Proof by Contradiction

A proof by contradiction, as the name implies, is an attempt to show that two claims cannot exist at the same time and, by extension, the original statement is true. For instance, a contradiction is saying that an integer $x$ is both even and odd. A proof by contradiction is also known as an indirect proof.

Proving a direct statement '$p$' by contradiction assumes that '$\neg p$' is true and attempts to derive a contradiction via this assumption. We know that this schema is a logical contradiction because '$p \land \neg p$' is always false for any truth value of '$p$'.

Proving an implication of the form '$p \to q$' by contradiction assumes that '$p$' is true and '$\neg q$' is true. For the implication to hold, a contradiction must arise, because an implication of the form '$\top \to \bot$' is false. The reason we make the aforesaid assumptions is because the goal is to prove that '$-(p \to q)$' does not hold true. Pushing the negation inward and rewriting the implication into a disjunction gives us the schema '$(p \land \neg q)$'. Accordingly, if we demonstrate that '$\neg p$' or '$q$' hold true, then this contradicts with the true value derived from the conjunction.

*Example.* "If $(x)^3$ is odd, then $x$ is odd".

*Proof.* We will prove that, if $(x)^3$ is odd and $x$ is even, then we arrive at a logical contradiction. Since $x$ is even, we can write it as $2m$ for some integer $k$. Plugging this into $(x)^3$ gets us $(2m)^3 = (2m \cdot 2m \cdot 2m)$. We can factor 2 out to get $2(4mmm)$. If we let $l = 4mmm$, we can substitute this in to get $2l$, the definition of an even integer. But we assumed that $x^3$ is odd. Therefore, by contradiction, if $x^3$ is odd, then $x$ is odd. QED.

*Example.* "There does not exist a largest integer".

*Proof.* To the contrary, we assume that there is a largest integer, which we call $N$. Thus, for all $n$, $N > n$. Now, suppose $m = N + 1$. $m$ is an integer because it is the sum of two integers. However, $m > N$, which contradicts our claim that $N$ is the largest integer. Therefore, by contradiction, there does not exist a largest integer. QED.

*Example.* "$\sqrt{2}$ is irrational".

*Proof.* Aiming for a contradiction, we assume that $\sqrt{2}$ is rational. This means we can write $\sqrt{2}$ as a rational number $p/q$ in its lowest terms. Namely, $\sqrt{2} = p/q$. Squaring both sides gets us $(\sqrt{2})^2 = (p/q)^2 = 2 = p^2/q^2$. We can rewrite this in terms of $p^2$ to get $p^2 = 2q^2$. From our first direct proof example, we know that when an integer $x$ is even, $x^2$ is also even. Thus, $p$ must be even, meaning we can represent it as $p = 2k$ for some integer $k$. Let us substitute this back into the equation: $2 = (2k)^2/q^2$, and simplifying gets us $2 = 4k^2/q^2$. Rewriting this to isolate $4k^2$ results in $2q^2 = 4k^2$. Finally, dividing both sides by 2 gives $q^2{=}2k^2$. Again, since $q^2$ is even (notice the $2k^2$), $q$ must be even. Since $p$ and $q$ are even, then we know $2 \mid p$ and $2 \mid q$. This contradicts our original assumption that $p/q$ is written in its lowest terms. Therefore, by contradiction, $\sqrt{2}$ is irrational. QED.

### Proof by Cases

Sometimes, it is necessary to prove multiple parts, or sub-pieces, of a conditional. This usually occurs when there are obvious cases in which a condition only may occur. The idea is to assume the premise within the case is true, then derive that the consequent of the original statement must also be true. When we encounter such a situation, we preface each case, or scenario, with **Case #n**.

*Example.* "If $y$ is odd and $x$ is an integer, then $x(x + y)$ is even".

*Proof.* Assume that the antecedent is true, namely that $y$ is odd and $x$ is an integer, where $y = 2k + 1$ for some integer $k$. There are two cases: one in which $x$ is even, and one in which $x$ is odd. We will prove that the consequent holds true for both cases:

**Case 1:** Assume that $x$ is even, where $x = 2l$ for some integer $l$. Then, we can substitute in our values for $x$ and $y$ into the consequent expression: $2l(2l + 2k + 1)$. If $m = 2l + 2k + 1$, we get $2m$, which is an even integer.

**Case 2:** Assume that $x$ is odd, where $x = 2l + 1$ for some integer $l$. Then, we can substitute in our values for $x$ and $y$ into the consequent expression: $(2l + 1)(2l + 1 + 2k + 1) = (2l + 1)(2l + 2k + 2) = 4l^2 + 2lk + 6l + 2k + 2$. Factoring out 2 gets us $2(2l^2 + 3l/2 + k + 1)$. Let $m = 2l^2 + 3l/2 + k + 1$. Thus, we get the form $2m$, which by definition is an even integer (note that the fractional $3l/2$ cancels out when multiplying the expression by two).

This covers both cases where $x$ is an integer. Therefore, if $y$ is odd and $x$ is an integer, then $x(x + y)$ is even. QED.

### If and Only If Proofs

If and only if proofs consist of two core components: proving the "if" direction, then proving the "only if" direction. Because an if and only if is comprised of two conjoined implications, namely $p \leftrightarrow q$ is equivalent to $(p \rightarrow q) \wedge (q \rightarrow p)$, we need to write proofs for two implications. When writing the subproofs, it is important to distinguish between the two. Therefore, writing **If:** or $\rightarrow$, and **Only if:** or $\leftarrow$. Remember that the rules for proving these implications remain the same, meaning we can prove the contrapositive of one implication but not the other and still achieve a correct and conclusive result. Note that for the following first example, we explicitly state the sub-implications. In future examples, we will only use the arrows to indicate the direction of the implication used in the subproof.

*Example.* "The sum of two integers $x$ and $y$ is odd if and only if exactly one of $x$ or $y$ is odd".

*Proof.*
($\rightarrow$) We will first prove the implication, "If $x + y$ is odd, then exactly one of $x$ or $y$ is odd". We will prove this by contraposition. Assume that it is not the case that exactly one of $x$ or $y$ is odd. This means that there are two cases:

**Case 1:** Assume that both $x$ and $y$ are odd. This means that $x = 2k + 1$ and $y = 2m + 1$ for some integers $k$ and $m$. $x + y$ is, therefore, $(2k + 1) + (2m + 1) = 2k + 2m + 2 = 2(k + m + 1)$, which is by definition an even integer. Thus, $x + y$ is even.

**Case 2:** Assume that both $x$ and $y$ are even. This means that $x = 2k$ and $y = 2m$ for some integers $k$ and $m$. $x + y$ is, therefore, $(2k) + (2m) = 2k + 2m = 2(k + m)$, which is by definition an even integer, meaning $x + y$ is even.

So, by cases and contraposition, if $x + y$ is odd, then exactly one of $x$ or $y$ is odd.

($\leftarrow$) We will now prove the converse, "If exactly one of $x$ or $y$ is odd, then $x + y$ is odd". Assume that exactly one of $x$ or $y$ is odd. Without loss of generality, we can assume that $x$ is the odd integer. Thus, $x = 2k + 1$ for some integer $k$, meaning $y = 2m$ for some integer $m$ (because $y$ cannot be odd). So, $x + y = (2k + 1) + 2m = 2k + 2m + 1 = 2(k + m) + 1$, which is the definition of an odd integer. Thus, $x + y$ is odd.

We have proved both implications. Therefore, the sum of two integers $x$ and $y$ is odd if and only if exactly one of $x$ or $y$ is odd. QED.

*Example.* "$x^3 - y^3$ is even if and only if $x - y$ is even".

*Proof.*
($\rightarrow$) We will prove this by contraposition, and assume that $x - y$ is odd. The only way that $x - y$ can be odd is if exactly one of $x$ or $y$ is odd. So, without loss of generality, we can assume that $x = 2k + 1$ and $y = 2m$ for some integers $k$ and $m$. Plugging these into $x - y$ shows that the result is odd. Substituting these into $x^3 - y^3$ gets us $(2k+1)^3 - (2m)^3$. Expanding this out results in $(8k^3 + 12k^2 + 6k + 1) - (2m \cdot 2m \cdot 2m)$ which simplifies to $2(4k^3 + 6k^2 + 3k) - 2(4mmm) + 1$. If we let $l = (4k^3 + 6k^2 + 3k) - 2(4mmm)$, our result is of the form $2l + 1$, which is the definition of an odd integer. Therefore, by contraposition, $x^3 - y^3$ is odd.

($\leftarrow$) Assume that $x - y$ is even. This occurs when $x$ and $y$ have the same parity. That is, either both $x$ and $y$ are even, or they are both odd. We can, therefore, use case analysis:

**Case 1:** Assume that $x$ and $y$ are even, in that $x = 2k$ and $y = 2l$ for any integers $k$ and $l$. Thus, to reaffirm our assumption, $(2k) - (2l) = 2(k - l)$, an even integer. Moreover, $x^3 - y^3 = (2k)^3 - (2l)^3 = 2(4k^3 - 4l^3)$, an even integer. Therefore, $x^3 - y^3$ is even.

**Case 2:** Assume that $x$ and $y$ are odd, in that $x = 2k + 1$ and $y = 2l + 1$ for any integers $k$ and $l$. Thus, to reaffirm our assumption, $(2k + 1) - (2l + 1) = 2(k - l)$, an even integer. Moreover, $x^3 - y^3 = (2k + 1)^3 - (2l + 1)^3 = 2(4k^3 - 4l^3 + 6k^2 - 6l^2 + 3k - 3l)$, an even integer.

So, by cases, $x^3 - y^3$ is even.

We have proved both implications. Therefore, $x^3 - y^3$ is even if and only if $x - y$ is even. QED.

*Example.* Prove that $x + y$ is even if and only if $x - y$ is even for all integers $x$ and $y$.

*Proof.*
($\rightarrow$) We will prove this directly and assume that $x + y$ is even. This means that $x + y = 2k$ where $k$ is some integer. Let us write an equation in an attempt to substitute $2k$ into $x - y$:

$$\begin{aligned} x + y &= 2k \\ x &= 2k - y \\ x - y &= (2k - y) - y \\ x - y &= (2k - 2y) \\ x - y &= 2(k - y) \end{aligned}$$

Let $m = k - y$, which gets us $2m$ where $m$ is some integer. Thus, if $x + y$ is even, then $x - y$ is even.

($\leftarrow$) We will prove this by contraposition, in which we have the statement if $x+y$ is odd, then $x-y$ is odd. Let us assume that $x+y$ is odd, meaning that $x+y=2k+1$ where $k$ is some integer. Let us again write an equation in an attempt to substitute $2k+1$ into $x-y$:

$$\begin{aligned} x+y &= 2k+1 \\ x &= 2k+1-y \\ x-y &= (2k+1-y)-y \\ x-y &= (2k+1-2y) \\ x-y &= 2(k-y)+1 \end{aligned}$$

Let $m = k-y$, which gets us $2m+1$ where $m$ is some integer. Thus, if $x+y$ is odd, then $x-y$ is odd.

We have proved both conditionals. Therefore, $x+y$ is even if and only if $x-y$ is even. QED.

## 2.6 Natural Deduction

Most proofs are completed via a sequence, or chain, of deductive steps to reach a logical conclusion. One such approach for proving statements is via *natural deduction*. While this method of proof has different names in slightly different contexts, the idea is to apply predefined rules and axioms to a set of premises to arrive at the desired conclusion. A benefit to using natural deduction is that its rules, in general, flow from what makes sense intuitively, hence the "natural" qualifier. For our purposes, we will apply natural deduction to both zeroth and first-order logic problems, starting with the former.

In a natural deduction proof, we have an argument $A(P, c)$, where $P$ may be extended with premises and derivations towards the conclusion. Let us do a ton of examples to present this technique and its axioms. As we said, however, natural deduction is taught in numerous ways, applicable to many contexts; our technique will be similar with the exception of adding and removing axioms when necessary out of simplicity.

*Example.* $A(\{p \to q, p\}, q)$. When we begin a natural deduction style proof, we write down each premise one after another on separate lines with annotations to the right that state that they are, in fact, premises. Subsequent lines are derivations from these premises or other presumed assumptions (more on this later).

$$\begin{array}{r|ll} 1 & p \to q & \text{P} \\ 2 & p & \text{P} \\ \cline{2-2} \end{array}$$

Where do we go from here? We want to prove '$q$', and we know that '$p \to q$' and '$p$' are true by the assertion that they are premises. Recall from its truth table that an implication is true if and only if it is not the case that the antecedent is true and the consequent is false. Therefore, because the antecedent, namely '$p$', is true, it must be the case that '$q$' is true. Intuitively, this idea should also make sense. So, we conclude that '$q$' is true via *modus ponens*, or →-elimination (abbreviated as $\to_{\text{elim}}$). In a natural deduction proof, after a derivation step, we specify the rule as well as the lines used in that rule.

| | | |
|---|---|---|
| 1 | $p \to q$ | P |
| 2 | $p$ | P |
| 3 | $q$ | $\to_{\text{elim}}$, 1, 2 |

Because we reached the conclusion, this completes the proof. QED.

*Example.* $A(\{p \to q, q \to r\}, p \to r)$

| | | |
|---|---|---|
| 1 | $p \to q$ | P |
| 2 | $q \to r$ | P |

This proof is slightly more complicated because it brings rise to a new rule, namely →-introduction (abbreviated as $\to_{\text{intro}}$). We want to show that, if '$p \to q$' is true and '$q \to r$' is true, then it holds that '$p \to r$' is true. Some may view this as a transitive implication, and that line of thought is exactly correct. Furthermore, this style of argumentation is, in fact, an axiom in many natural deduction systems called hypothetical syllogism. For us, however, we will prove the implication holds true through the notion of sub-proofs and $\to_{\text{intro}}$. The $\to_{\text{intro}}$ rule is defined as follows: if, by assuming $\varphi$ we can derive $\psi$, then we can derive '$\varphi \to \psi$'. So, the first part of the $\to_{\text{intro}}$ rule is to assume the antecedent of the implication is true. Because we are trying to deduce that '$r$' is true under the assumption that '$p$' is true, we will indent this as a sub-proof.

| | | | |
|---|---|---|---|
| 1 | $p \to q$ | | P |
| 2 | $q \to r$ | | P |
| 3 | | $p$ | Ass. |

Now we show that '$q$' is true by $\to_{\text{elim}}$, and from this, show that $r$ is true by the same logic.

| | | | |
|---|---|---|---|
| 1 | $p \to q$ | | P |
| 2 | $q \to r$ | | P |
| 3 | | $p$ | Ass. |
| 4 | | $q$ | $\to_{\text{elim}}$, 1, 3 |
| 5 | | $r$ | $\to_{\text{elim}}$, 2, 4 |

We have shown that, if we assume '$p$', we can deduce '$r$'. Hence, we may conclude that '$p \to r$'. Because this completes the sub-proof, we un-indent its conclusion.

$$\begin{array}{r|lll}
1 & p \to q & & \text{P}\\
2 & q \to r & & \text{P}\\ \hline
3 & & \multicolumn{1}{|l}{p} & \text{Ass.}\\ \cline{3-3}
4 & & \multicolumn{1}{|l}{q} & \to_{\text{elim}},\ 1,\ 3\\
5 & & \multicolumn{1}{|l}{r} & \to_{\text{elim}},\ 2,\ 4\\
6 & p \to r & & \to_{\text{intro}},\ 3\text{—}5
\end{array}$$

This, of course, completes the collective proof. QED.

*Example.* $A(\{p, q, p \wedge r\}, p \wedge (q \wedge r))$

$$\begin{array}{r|ll}
1 & p & \text{P}\\
2 & q & \text{P}\\ \cline{2-2}
\end{array}$$

This next proof is incredibly simple and showcases two rules involving conjunction: $\wedge_{\text{intro}}$ and $\wedge_{\text{elim}}$. The former allows us to conjoin any two schemata that are currently "active". On the other hand, the latter rule allows us to prove either proposition of a conjunction. Recall that a conjunction is true if and only if both operands are true. So, if we want to create our desired conclusion, we should split '$p \wedge r$' to get '$r$', then use the introduction rule twice.

$$\begin{array}{r|ll}
1 & p & \text{P}\\
2 & q & \text{P}\\
3 & p \wedge r & \text{P}\\ \cline{2-2}
4 & r & \wedge_{\text{elim}},\ 3\\
5 & q \wedge r & \wedge_{\text{intro}},\ 2,\ 4\\
6 & p \wedge (q \wedge r) & \wedge_{\text{intro}},\ 1,\ 5
\end{array}$$

This completes the proof. QED.

*Example.* $A(\{t, (r \wedge \neg s) \to p, \neg s, t \to r\}, p \vee q)$

Let us put the rules that we have learned so far to the test and introduce a new rule: $\vee_{\text{intro}}$. $\vee_{\text{intro}}$ allows us to take any schema and affix any other arbitrary schema with disjunction. Recall that a disjunction is true if and only if at least one of its operands are true. So, if we know that an arbitrary schema $\varphi$ is true, then it must be the case that '$\varphi \vee \psi$' is true for any schema $\psi$. In this proof, we must isolate '$p$' so we may apply $\vee_{\text{intro}}$.

$$\begin{array}{r|ll}
1 & t & \text{P}\\
2 & (r \wedge \neg s) \to p & \text{P}\\
3 & \neg s & \text{P}\\
4 & t \to r & \text{P}\\ \cline{2-2}
5 & r & \to_{\text{elim}},\ 1,\ 4\\
6 & r \wedge \neg s & \wedge_{\text{intro}},\ 6,\ 3\\
7 & p & \to_{\text{elim}},\ 2,\ 6\\
8 & p \vee q & \vee_{\text{intro}},\ 7
\end{array}$$

This completes the proof. QED.

*Example.* $A(\{p \vee q, p \rightarrow r, q \rightarrow r\}, r)$

| | | |
|---|---|---|
| 1 | $p \vee q$ | P |
| 2 | $p \rightarrow r$ | P |
| 3 | $q \rightarrow r$ | P |

A natural deduction proof technique that we have omitted until now is otherwise called an indirect proof or a proof by contradiction. We saw how this works for formal proofs with numbers and other statements, but we can generalize it with schemata. That is, assume '$\neg A$' for some schema $A$. Then, show that, by assuming '$\neg A$', we reach a contradiction. We are then allowed to conclude that '$\neg A$' must be true. Recall that we reach a contraction when, for any premise '$\neg A$', we deduce '$A \wedge \neg A$'. Let us try this out by proving another form of $\rightarrow_{\text{elim}}$, namely modus tollens: the law of contraposition. In addition, we will make use of two new rules: double negation elimination (DNE), which allows us to conclude a proposition $A$ from '$\neg\neg A$', as well as double negation introduction (DNI), which allows us to conclude '$\neg\neg A$' from a proposition $A$.

*Example.* $A(\{p \rightarrow q, \neg q\}, \neg p)$

| | | | |
|---|---|---|---|
| 1 | $p \rightarrow q$ | | P |
| 2 | $\neg q$ | | P |
| 3 | | $\neg\neg p$ | Ass. |
| 4 | | $p$ | DNE, 3 |
| 5 | | $q$ | $\rightarrow_{\text{elim}}$, 1, 4 |
| 6 | | $q \wedge \neg q$ | $\wedge_{\text{intro}}$ |
| 7 | | $\bot$ | False, 6 |
| 8 | $\neg p$ | | IP, 3–7 |

This completes the proof. QED.

*Example.* $A(\{p \vee q, \neg p\}, q)$

Let us prove disjunctive syllogism (DS). Though, there is one extra rule that we must introduce in order to complete this proof: the explosion principle. In summary, it says that from a contradiction, we may infer any premise at all. Note that this is different from an indirect proof/proof by contradiction where we explicitly assume '$\neg A$' to derive a contradiction that produces $A$. If we find a contradiction without any previous assumptions that aim toward a contradiction, any formula is provable (including the conclusion itself!). The proof of disjunctive syllogism may look a little funky because we nest a proof by cases inside; we need to show that by assuming '$p$' and assuming '$q$' separately, we deduce '$q$'. The latter case is obvious, but the former requires the explosion principle.

| | | |
|---|---|---|
| 1 | $p \lor q$ | P |
| 2 | $\neg p$ | P |
| 3 | $p$ | Ass. |
| 4 | $p \land \neg p$ | $\land_{\text{intro}}$ |
| 5 | $\bot$ | False, 4 |
| 6 | $q$ | Explode, 5 |
| 7 | $q$ | Ass. |
| 8 | $q$ | Rep, 7 |
| 9 | $q$ | Cases, 1–2, 3–6, 7–8 |

This completes the proof. QED.

*Example.* $A(\{p \to q, r \to s, p \lor r\}, q \lor s)$

Let us try another argument form: constructive dilemma. We will show that by assuming either antecedent of the provided implications, we can always derive the argument conclusion.

| | | |
|---|---|---|
| 1 | $p \to q$ | P |
| 2 | $r \to s$ | P |
| 3 | $p \lor r$ | P |
| 4 | $p$ | Ass. |
| 5 | $q$ | $\to_{\text{elim}}$, 1, 4 |
| 6 | $q \lor s$ | $\lor_{\text{intro}}$, 5 |
| 7 | $p \to (q \lor s)$ | $\to_{\text{intro}}$, 4–6 |
| 8 | $r$ | Ass. |
| 9 | $s$ | $\to_{\text{elim}}$, 2, 8 |
| 10 | $q \lor s$ | $\lor_{\text{intro}}$, 9 |
| 11 | $r \to (q \lor s)$ | $\to_{\text{intro}}$, 8–10 |
| 12 | $q \lor s$ | Cases, 1–3, 7, 11 |

This completes the proof. QED.

*Example.* $A(\{p \to q, r \to s, \neg q \lor \neg s\}, \neg p \lor \neg r)$

Up next, we shall prove destructive dilemma: the dual to constructive dilemma.

| | | | |
|---|---|---|---|
| 1 | $p \to q$ | | P |
| 2 | $r \to s$ | | P |
| 3 | $\neg q \vee \neg s$ | | P |
| 4 | | $p$ | Ass. |
| 5 | | $q$ | $\to_{\text{elim}}$, 1, 4 |
| 6 | | $\neg\neg q$ | DNI, 5 |
| 7 | | $\neg s$ | DS, 3, 6 |
| 8 | | $\neg r$ | MT, 2, 7 |
| 9 | | $\neg p \vee \neg r$ | $\vee_{\text{intro}}$ |
| 10 | $p \to \neg p \vee \neg r$ | | $\to_{\text{intro}}$, 4–9 |
| 11 | | $r$ | Ass. |
| 12 | | $s$ | $\to_{\text{elim}}$, 2, 11 |
| 13 | | $\neg\neg s$ | DNI, 12 |
| 14 | | $\neg q$ | DS, 3, 13 |
| 15 | | $\neg p$ | MT, 1, 14 |
| 16 | | $\neg p \vee \neg r$ | $\vee_{\text{intro}}$ |
| 17 | $r \to \neg p \vee \neg r$ | | $\to_{\text{intro}}$, 11–16 |
| 18 | $\neg p \vee \neg r$ | | Cases 1-3, 10, 16 |

This completes the proof. QED.

## 2.7 Numbering Systems

### Binary

We, as the readers, use base ten numbering in our daily lives.[1] That is, we have ten digits ranging from 0 to 9, which form base ten numbers. After 9, we wrap around to 10, proceeding to 19, and rolling right into 20. This continues until we run out of "tens" decimal places at 99, hence we roll over to 100, repeating ad nauseam. Base ten is convenient because we can represent a very large number with only a few digit positions. Compare this to the base two system that computers use, otherwise called *binary*. In base ten, or decimal, we have digits, whereas in binary we have bits, or binary digits. Each bit is either 0 or 1, hence the base two name. Similar to base ten, we wrap around and introduce a new bit once we exhaust all possible values. Let us take a look at a table to better visualize the concept.

| Base 10 | Base 2 |
|:-:|--:|
| 0 | 0 |
| 1 | 1 |
| 2 | 10 |
| 3 | 11 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |

Figure 2.10: Base Ten and Base Two Equivalents from Zero to Seven

Hopefully, the pattern is evident. We will also note of a type of notation used to distinguish between bases: we use a subscript $x_{10}$ to indicate that $x$ is written in base ten, whereas $y_2$ is used to indicate a binary base. In general, we adopt the style $z_b$ to designate that the number $z$ is in base $b$. Some may beg the question as to why computers do not work with base ten, since it is the system we, as humans, rely on daily. The answer boils down to the fact that computers use electricity and circuits; electricity in a system either flows or it does not. This binary representation is ideal for designing logic gates since we only work with two possibilities. Of course, this raises another point: how do we convert from base ten to base two, or vice-versa? Let us walk through this by a few examples, and then we can devise an algorithm.

*Example.* Convert $17_{10}$ into base two.

To convert $17_{10}$ into base two, we will continuously divide the number by 2, taking the remainder and pushing it in an output space. We continue to divide our number until its quotient is zero.

[1] This claim is made under the assumption that we do not switch numbering systems, or a new species introduces us to something superior.

| Input | Quotient | Remainder | Output |
|---|---|---|---|
| 17 | 8 | 1 | 1 |
| 8 | 4 | 0 | 01 |
| 4 | 2 | 0 | 001 |
| 2 | 1 | 0 | 0001 |
| 1 | 0 | 1 | 10001 |

At each step, we compute the quotient, its remainder, and push the resulting remainder to the output. $17_{10} = 10001_2$. There is a simple verification method to check our work, but we will save this until we get to the section on converting from base two to base ten.

*Example.* Convert $63_{10}$ into base two.

| Input | Quotient | Remainder | Output |
|---|---|---|---|
| 63 | 31 | 1 | 1 |
| 31 | 15 | 1 | 11 |
| 15 | 7 | 1 | 111 |
| 7 | 3 | 1 | 1111 |
| 3 | 1 | 1 | 11111 |
| 1 | 0 | 1 | 111111 |

$63_{10} = 111111_2$.

*Example.* Convert $101_{10}$ into base two.

| Input | Quotient | Remainder | Output |
|---|---|---|---|
| 101 | 50 | 1 | 1 |
| 50 | 25 | 0 | 01 |
| 25 | 12 | 1 | 101 |
| 12 | 6 | 0 | 0101 |
| 6 | 3 | 0 | 00101 |
| 3 | 1 | 1 | 100101 |
| 1 | 0 | 1 | 1100101 |

$101_{10} = 1100101_2$.

*Example.* Convert $724_{10}$ into base two.

| Input | Quotient | Remainder | Output |
|---|---|---|---|
| 724 | 362 | 0 | 0 |
| 362 | 281 | 0 | 00 |
| 181 | 90 | 1 | 100 |
| 90 | 45 | 0 | 0100 |
| 45 | 22 | 1 | 10100 |
| 22 | 11 | 0 | 010100 |
| 11 | 5 | 1 | 1010100 |
| 5 | 2 | 1 | 11010100 |
| 2 | 1 | 0 | 011010100 |
| 1 | 0 | 1 | 1011010100 |

$724_{10} = 1011010100_2$.

In each of these examples, we perform the same steps to reach the desired outcome. Consequently, we can write an algorithm to convert between base ten to base two as a procedure:

```
(1) Store input into $N$.
(2) Store N/A into Result.
(3) If $N$ is greater than zero GoTo (4), Else GoTo (7)
(4) Append ($N$ mod 2) to front of Result.
(5) Set $N$ to the quotient of $N$ and 2.
(6) GoTo (3)
(7) Display Result.
```

Converting from base two to base ten is even simpler than the other way around. First, let us reason about what a decimal number is, at its core, and how its digit positions interplay. For instance, consider the number $342_{10}$. If we didn't know that $342_{10}$ *is* 342 in base ten, we could check by multiplying each digit by its position as a power of ten, then summing the result. Thus, our answer is

$$\begin{aligned} &= 3 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0 \\ &= 3 \cdot 100 + 4 \cdot 10 + 2 \cdot 1 \\ &= 300 + 40 + 2 \\ &= 342 \end{aligned}$$

This exact principle is shared among any base and not exclusive to base ten. To demonstrate, let us convert the base two values from the previous exercises back into base ten.

*Example.* Convert $10001_2$ to base ten.

Like we said, we can use the principle that each bit corresponds to a power of two, multiply them by the bit at that position, and sum the results.

$$\begin{aligned} &= 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 \\ &= 16 + 0 + 0 + 0 + 1 \\ &= 17 \end{aligned}$$

So, $10001_2 = 17_{10}$.

*Example.* Convert $111111_2$ to base ten.

$$\begin{aligned}
&= 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
&= 1 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 \\
&= 32 + 16 + 8 + 4 + 2 + 1 \\
&= 63
\end{aligned}$$

So, $111111_2 = 63_{10}$.

*Example.* Convert $1100101_2$ to base ten.

$$\begin{aligned}
&= 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\
&= 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 \\
&= 64 + 32 + 0 + 0 + 4 + 0 + 1 \\
&= 101
\end{aligned}$$

So, $1100101_2 = 101_{10}$.

*Example.* Convert $1011010100_2$ to base ten.

$$\begin{aligned}
&= 1 \cdot 2^9 + 0 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 \\
&= 1 \cdot 512 + 0 \cdot 256 + 1 \cdot 128 + 1 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 0 \cdot 1 \\
&= 512 + 0 + 128 + 64 + 0 + 16 + 8 + 4 + 0 + 0 \\
&= 724
\end{aligned}$$

So, $1011010100_2 = 724_{10}$.

### Hexadecimal

The last non-decimal numbering system that is important to us as computer scientists (that we mention, at least) is hexadecimal. Using context clues, we can infer that the prefix '*hexa*' means six, and the prefix '*deci*' means ten. Therefore hexadecimal has sixteen possible values, ranging from 0 to 9, then A to F, representing 10 to 15 respectively. Converting between binary and hexadecimal is amazingly trivial. Because this is the case, we will not describe the approach to converting directly to and from base ten to hexadecimal since it is more cumbersome and largely resembles the conversion from decimal to binary, just with base 16 powers rather than base two. Hexadecimal has interesting uses in computer programming, including a compact representation for colors, and memory address layout numbering.

*Example.* Convert $8A_{16}$ into base two.

To convert $8A_{16}$, we split the number character-by-character, and convert each hexadecimal character individually. $8_{16}$ is $1000_2$, and $A_{16}$ is $10_{10}$, which is 10102. From there we conjoin the two binary numbers to get $8A_{16} = 10001010_2$.

*Example.* Convert $94C0B53_{16}$ into base two.

Although this number is rather large, the process is the same as before, just more laborious. $9_{16} = 1001_2$, $4_{16} = 0100_2$, $C_{16} = 1100_2$, $0_{16} = 0000_2$, $B_{16} = 1011_2$, $5_{16} = 0101_2$, $3_{16} = 0011_2$. Conjoining each sub-binary number gets us $94C0B53_{16} = 1001010011000000101101010011_2$.

Let us go the other direction and convert a binary number into hexadecimal. Fortunately the process brings nothing new to the table, although it begins to feel mundane.

*Example.* Convert $100011_2$ to hexadecimal.

Converting $100011_2$ into hexadecimal may seem scary at first because the number of bits is not divisible by four. Worry not, though, because we can just pad leading zeroes to the front of the number if we so choose without losing any semantic detail. So, $0010_2 = 2_{16}$ and $0011_2 = 3_{16}$, meaning $100011_2 = 23_{16}$.

*Example.* Convert $1100010111010001101_2$ to hexadecimal.

Again, we split the binary number into groups of four bits as follows: $0110_2 = 6_{16}$, $0010_2 = 2_{16}$, $1110_2 = \mathrm{E}_{16}$, $1000_2 = 8_{16}$, and $1101_2 = \mathrm{D}_{16}$. Thus we arrive at the result $1100010111010001101_2 = \mathrm{62E8D}_{16}$.