Logic 1

Today:

- ☐ Boolean values and some operations on them
- ☐ Representing natural numbers in binary—or in other bases
- ☐ The boolean-modeling thesis

1 Boolean Values

Propositional logic deals with a universe having just two points:

- $\mathbb{B} = \{0, 1\}$ — the set of *boolean* (or *logical*) values

> ▷ The customary convention is to capitalize the word *Boolean*. But I think it odd that this capital hasn't faded out of existence by now. I am going to decapitalize it. The term honors the English logician George Boole (1816–1864). But it is actually a *greater* honor when something becomes so central in our thought that we no longer associate it to a person, but regard it as a basic conceptual category. That change, it seems to me, is marked by dropping the capital. Congratulations, Prof. Boole!

There are other common names for the points 0 and 1. The point 0 also goes by the name F, False, and false. The point 1 also goes by the name T, True, and true. In one of several fonts, like F instead of F, or `true` instead of true.

> ▷ When you are doing math, differentiating case and font is helpful. It can add to readability. Teach your eyes and mind to see an N, n, N, and n as different things. Really look at what you are seeing on the page. A forward quote is not a backward quote; a short dash is not a long dash; and so on.

The names "false" and "true" for points in $\mathbb{B}$ reflect their intended meaning. But *intent*—not to mention *truth*—is a slippery things. We will imbue the point 1 (or T, True, true) with the character of *truth* only through what we *do* with that point. We will imbue 0 (or F, False, false) with the character of *falsity* only through what we do with *it*. That is, through our choices—our definitions—in how we may manipulate the points in $\mathbb{B}$.

In most of your mathematical career you have dealt with sets that are way bigger and more complicated than our little set $\mathbb{B}$. You know quite well the infinite sets:

- $\mathbb{Z}^+ = \{1, 2, 3, \ldots\}$ — the set of *positive integers*

- $\mathbb{N} = \{0, 1, 2, \ldots\}$ — the set of *natural numbers* — I will always include 0
- $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ — the set of *integers*
- $\mathbb{Q} = \{p/q \colon p, q \in \mathbb{Z},\ q \neq 0\}$ — the set of *rational numbers*
- $\mathbb{R}$ — the set of *real numbers*

But there is a lot of math that deals with *finite* sets. And to a computer scientist, the set of booleans $\mathbb{B}$ feels like the most important finite set of all.

> ▷ It is wonderful thing having a universe of only two points—a way simpler place to live than the natural numbers, the integers, or the reals. I can't for the life of me understand why school children start their studies with the positive integers, $\mathbb{Z}^+$. This is a terribly complicated thing compared to $\mathbb{B}$. Maybe you can go back and teach all the young people in your life to love $\mathbb{B}$, and let them them see it as a great place to play.

Given the importance of boolean values, it might be a strange to have to admit that there isn't really a widely-used symbol for it. I'm calling it $\mathbb{B}$, but that's not so common. Words and notation for basic concepts in discrete math are not well standardized. And I don't want you memorizing lots of words or symbols, which is the antithesis of doing math. Still, I like to comment on it when genuinely basic things go by lots of different names.

And even the name for the topic we are now studying is not exactly standardized! It is variously called *propositional logic*, *propositional calculus*, *sentential logic*, or *sentential calculus*. I think you can also call it *boolean algebra* or *boolean logic*. As far as I can tell, they're all the same thing. Namely, thinking about the two-element sets we're calling $\mathbb{B}$.

It is common to introduce propositional logic by telling the students not that we are studying operations on $\mathbb{B}$, but to tell them that we are representing the veracity of English language propositions—statements like "it is raining today," or "William Shakespeare was born in France." I think this is nonsense. There is a fundamental mismatch, I would claim, between the nature of our subject and what is routinely going on in making and interpreting declarative English statements. But let's come back to that complaint in a little bit. For now, I would prefer that you get comfortable taking an abstract view of mathematical concepts. It is actually much simpler to study abstract operations on $\mathbb{B}$ than to try to ground in mathematics some fragment of English or philosophy.

2 Unary and Binary Operators

Before speaking of operations on $\mathbb{B}$, let's think about operations on other, more familiar sets. In fact, you know many operations on $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$.

As a first example, given a natural number n, there is an operator (or an *operation* or a *function*—I will use the words synonymously) that gives its *successor*. Call this function S. It's probably

the *simplest* thing you can do to a natural number. It takes a natural number n and returns the natural number $S(n)$ that is one more. For example, $S(13) = 14$ and $S(S(0)) = 2$.

As a second example, given a pair of natural numbers a and b, there is an operator that gives their sum, which is another natural number. Denoting this operator $+$, it takes in two natural numbers, a and b, and returns a third, $a + b$. For example, $3 + 4 = 7$.

The successor function is a *unary* operator. We could write its *signature* as

$$S : \mathbb{N} \to \mathbb{N} .$$

We will be exploring functions later in the term, but we can explain the notation now. First comes the name of the function, followed by a colon. Functions deserve names. Then comes the *domain* of the function, followed by a right arrow. The domain is the *set* of all the values that you can feed to the function. Following that is the *codomain* of the function, which is a set big enough to contain all the possible return values from the function. Thus here we have a function named S that takes in ("maps") a point in the domain $\mathbb{N}$ to a point that is again in $\mathbb{N}$. Every point in $\mathbb{N}$ must get mapped to something; no point can be left out. We call the successor function unary because it takes in only *one* input.

On the other hand, the addition operator, which we denoted "+", would have a signature that looks like

$$+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N} .$$

What this means is that $+$ is a function (that is its name) that takes in a pair of integers (a, b). That's the meaning of $\mathbb{N} \times \mathbb{N}$; it's a new set whose elements are *pairs* of points from $\mathbb{N}$. Such pairs are the domain of our function $+$. Given such a pair (a, b), our operator $+$ will return a natural number $a + b$. We call $+$ a *binary* operator because it takes in two points rather than one. The addition function returns a single number, just as the successor function did.

We could keep going; we could have operators (functions) that take in 3, 4, or more inputs. The number of inputs to an operator can be anything.

You'll notice some major notational differences between S and $+$. First, we wrote the value that S operated on *after* we wrote its name, and we surrounded that value with parentheses: $S(\text{something}) = \text{something-else}$. Because we wrote S before the thing we were operating on, we could call it *prefix* notation.

In contrast, with addition, we didn't write $+(3, 5)$, but $3 + 5$. Why? There is no real reason; it is just tradition. This notation is called *infix*, where you put the operator *between* the two arguments it is operating on.

Infix notation works fine for binary operators, but if you had three or more inputs, it wouldn't be clear what to do. (Well, can you suggest something?)

> ▷ If the domain of $+$ is pairs of numbers (a, b), and if we elect to write our $+$ operator in prefix, then ought not the notation have been $+((a, b))$, not $+(a, b)$? The outer parentheses would signify an argument to the named function; the inner parentheses would be used in denoting the pair. But nobody does this. In writing $F(a, b)$, for some function F, it is understood that the value being operated on by the function F is the pair (a, b).

There is a second difference between our notations involving for S and $+$: one used a letter of English and one used a special symbol. In fact, mathematicians and computer scientists use a wide variety of letters, words, and symbols to denote functions and operators. They write them in functional notation (like $F(a, b)$), prefix notation (like $-a$), infix notation (like $a + b$), suffix notation (like $n!$), and in weirder ways still. Don't allow the notational peculiarities to hide the mathematical substance. Notation is something to read "through," a sort of necessary evil to support communication. Maybe it changes so much because we are forever trying to push it towards invisibility.

Examples of binary operators on integers and reals include addition ($a + b$), subtraction ($a - b$), and multiplication ($a \cdot b$). Examples of unary operators on integers and reals include negation ($-x$, confusingly employing the same symbol as subtraction), absolute value (with the exceptional notation $|a|$), and squaring (with the exceptional notation a^2).

3 Some Operators on Booleans

Let's return now to our boolean domain $\mathbb{B}$. It supports a collection of unary and binary operators, too. If we had plenty of time, and if you were school children never before exposed to this stuff, I'd let you *invent* all these operators on your own, and decide for yourself which were interesting, and why.

The first binary operator I'll name is the OR operator. If p and q are boolean variables ($p, q \in \mathbb{B}$) then $p \vee q$ is true if either p is true or q is true, while it is false if both p and q are false. We can represent this with a *truth table*:

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

Another way to write $p \vee q$ is p or q. Or you could write it as p OR q. Or even $p + q$. In the context or boolean logic, they all mean the same thing.

The four rows of our truth table exhaust the four possible settings of truth values to p and q. The first two columns indicate which setting of truth values we are considering. The corresponding third column then indicates the result.

You could just as well write the truth table above with one of our other ways to denote the points in $\mathbb{B}$, or with one of the alternative symbols for the *disjunction* operation—a fancy word for *or*.

p	q	$p \vee q$
F	F	F
F	T	T
T	F	T
T	T	T

The table above represents the exact same information as the previous one. I'm writing it in a different way for no reason beyond that fact that I want you to try to *see through* notation. To see into the meaning. I myself struggle with this; my mind wants to attend to all those glyphs on the page. There is a captivating aesthetic in them, when they are chosen well, when they are elegant and minimal. And if those glyphs are ugly or incorrectly used, then I myself have no chance at seeing through them, to their intended meaning.

Another binary operator on booleans is AND. If $p, q \in \mathbb{B}$ then $p \wedge q$ is true when both p and q are true, but is otherwise false:

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

Another way to write $p \wedge q$ is p and q, or as p AND q. And you can write it as just $p\,q$. That is, when no explicit operator is written between the boolean variables, one assumes that logical *conjunction*—AND—is what's meant. This is exactly analogous to ab meaning $a \cdot b$ in the world of integer or real multiplication; no written operator implicitly means to multiply.

The boolean AND and integer multiply have a lot of commonalities. As does boolean OR and integer addition. Can you find parallels on your own, pushing this analogy as far as makes sense?

An important *unary* operator is NOT, or *negation*. If $p \in \mathbb{B}$ then $\neg p$ is the true when p is false, and false when p is true:

p	$\neg\, p$
0	1
1	0

Alternative ways to write $\neg p$ are: not p, NOT p, and $\overline{p}$. The overline notation is quite convenient for negation, saving on parenthesis and enhancing readability. The length of the overline indicates the scope of what is negated. For example, you can write $\overline{pq}$ instead of $\neg(pq)$ or $\neg(p \wedge q)$; they all mean the same thing.

In all of the above I've been using lower case letters p, q for boolean variables. Other choices look fine, too, like $P \vee \neg P$, $A(B \vee C)$, $(X_1 \vee \overline{X_2} \vee X_4)(\overline{X_1} \vee X_4 \vee X_5)$.

You can make long boolean formulas using ANDs, ORs, and NOTs. For example, the formula $S(a, b, c) = ca \vee \overline{c}b$ involves two ANDs, one OR, and one negation. What does the formula *do*? Or the formula $T(a, b, c) = (ab \vee bc \vee ac) \wedge \overline{abc}$. What does *it* do?

Truth tables aren't limited to two input variables and four resulting rows; they can describe boolean function with a finite number of boolean inputs. Here's a truth table for the function function $T(a, b, c)$ that I just named:

a	b	c	$T(a,b,c)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

To describe a boolean function $P(x_1, \ldots, x_n)$ you will need a truth table with 2^n rows, one row for each possible setting of its arguments.

How should we list the rows of a truth table? I'd like to make an important aside for that. Our aside will actually help motivate why boolean values are so cool.

4 Representing Numbers in Binary and Other Bases

There is a natural order for enumerating the rows of a truth table or, equivalently, for enumerating all the possible n-tuples of binary values $(x_{n-1}, \ldots, x_1, x_0)$, $x_i \in \{0, 1\}$. The ordering of which I speak is the one where the 2^n rows of the truth table represent the numbers $0, 1, \ldots, 2^n - 1$, written as n-bit binary *strings* in the way that we now describe.

You all know the usual way to write a number a in decimal notation, writing a sequence of digits $a_{k-1}a_{k-2}\ldots a_2a_1a_0$, each digit being a character (a formal symbol) from $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Each of those ten characters represents a corresponding number. The sequence of characters above represents the number

$$a_{k-1} \cdot 10^{k-1} + a_{k-2} \cdot 10^{k-2} + \cdots + a_2 \cdot 10^2 + a_1 \cdot 10^1 + a_0 \cdot 10^0.$$

For example, $385 = 3 \cdot 10^2 + 8 \cdot 10^1 + 5 \cdot 10^0 = 300 + 80 + 5$. We call this way of writing numbers the *decimal* (or *base-10* or *radix-10*) representation of the number because we are using 10 different symbols to represent the number, with each position, as you move to the left from the right-hand edge, representing the next higher power of 10. To emphasize that we are writing a number in decimal we can write the base as a subscript of the parenthesized value, as in $385 = (385)_{10}$.

The *binary* (or *base-2* or *radix-2*) representation of a number works in the exactly analogous way. Now the digits are drawn from $\{0, 1\}$ and these symbols, called *bits*, represent the numbers 0 and 1. A sequence of bits represents a numbers where each position you move to the left represents the next higher power of 2. Thus 10110 in base-2, which we might write as $(10101)_2$, would represent the number $1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 4 + 2 = 22 = (22)_{10}$.

> ▷ What's the difference between a boolean value and a bit? None. They're the same thing.

▷ Powers of 2 come up so often in computer science that you might want to learn by heart at least the first ten: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, $1024 = 2^{10}$. Also, it's good to keep in mind that 2^{20} is about a million (10^6), while 2^{30} is about a billion (10^9).

▷ Try to distinguish between a thing and a *representation of the thing.* Is 385 a number or a sequence of digits that represent a number? It is both, and they are different. We are explaining different ways to represent numbers as sequences of symbols.

Most things of interest will have lots of representations—maybe infinitely many. For example, I might choose to represent 385 as $5 \cdot 7 \cdot 11$, or as the binary number 110000001. It's the same number regardless of representation. On the other hand, the sequence of symbols "385" will represent the integer 901 if I tell you to interpret the sequence of digits in "hexadecimal." In math you get to declare or invent the context, which will determine what things mean.

I have seen students assume that "decimal notation" has something to do with writing a decimal point—the period symbol in 3.0 or 3.14. This is not the intended meaning. When we speak of binary or decimal notation we usually have in mind that we are representing natural numbers, so there are no decimal points in sight. That said, what rational number do you think that the binary number 101.1101 *should* represent?

You can represent numbers in any base $b \geq 2$ you like. If you were working base-3 (I have no idea why), the digits would be $\{0, 1, 2\}$ and we'd have, for example, $(20121)_3 = 2 \cdot 3^4 + 0 \cdot 3^3 + 1 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0 = 2 \cdot 81 + 9 + 2 \cdot 3 + 1 = 162 + 9 + 6 + 1 = 178$.

There is a difficulty with representing numbers in a radix exceeding 10: what symbols should you use for digits exceeding 9? There is a standard answer for the next six digits: we use A (or a) for 10, B (or b) for 11, C (or c) for 12, D (or d) for 13, E (or e) for 14, and F (or f) for 15. So $(3\text{B}9\text{F})_{16} = 3 \cdot 16^3 + 11 \cdot 16^2 + 9 \cdot 16^1 + 15 = (15263)_{10}$.

For making truth tables with the rows pleasantly ordered, count in binary. How do you count in binary? When you are incrementing a number that ends with a run of ones and then a zero, or nothing at all, just to their left, then you change all of those ones to zeros, make the next digit to the left a one, and leave alone any further digits to the left. For example, the successor of 1000111 is 1001000; the successor of 1111 is 10000. This is the natural generalization of incrementing a number represented in decimal. Make sure you can do it, understand it, and understand how to generalize the idea to other bases.

▷ Practice counting. In binary: 0, 1, 10, 11, 100, 101, 110, ... In radix-3: 0, 1, 2, 10, 11, 12, 20, 21, 22, 100, 101, ...

Let's return now to making a truth table for an arbitrary function F with three boolean inputs. The eight rows are counting from 0 to $7 = 2^3 - 1$ in binary, but where we now pad on the left

with leading zeros so as to make each binary value three bits long. As with decimal numbers, prefixing a binary number with an arbitrary number of zeros doesn't change its value.

a	b	c	$F(a,b,c)$
0	0	0	$F(0,0,0)$
0	0	1	$F(0,0,1)$
0	1	0	$F(0,1,0)$
0	1	1	$F(0,1,1)$
1	0	0	$F(1,0,0)$
1	0	1	$F(1,0,1)$
1	1	0	$F(1,1,0)$
1	1	1	$F(1,1,1)$

5 Logical Completeness

Can you express any boolean-valued function you like with just AND, OR, and NOTs? Suppose, say,you have the following boolean-valued function:

a	b	c	$f(a,b,c)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Can you see a way to "read off" a boolean formula $f(a,b,c)$ that has the truth table above? How about

$$f(a,b,c) = \overline{a}bc \vee a\overline{b}c \vee ab\overline{c}.$$

In general, suppose we have a truth for n variables and where there are $c \geq 1$ rows where the function value takes on 1. Then we can write this formula as the OR of c "terms," where each term is the AND of n literals, and each literal is either a variable or its complement. It means that AND, OR, NOT, taken together, are rich enough to encode *anything*.

Well, be careful about the truth table that is identically 0—the constant function that is everywhere false. How can you write it as a formula with only ANDs, OR, and NOTs.

The fact that you can represent *any* truth table with just AND, OR, and NOT says something interesting about those three functionalities, taken together. That they're super powerful. That you don't need anything else. We say that, taken together, they are *functionally complete*. A set of operators is functionally complete if they can be used to express arbitrary logical functionality.

We can try to find other functionally complete sets of operators, if we want.

6 Equivalence

Recall the function I named earlier, $T(a, b, c) = (ab \vee bc \vee ac) \wedge \overline{abc}$. If you make a truth table for it, you will find that it is *identical* to the truth table for f above. Said differently, the two formulas are *equivalent*: they capture the exact same logical condition. You can write it like this:

$$\overline{a}bc \vee a\overline{b}c \vee ab\overline{c} \;\equiv\; (ab \vee bc \vee ac) \wedge \overline{abc}.$$

In general, if $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_m)$ are boolean formulas, then we say that f and g are equivalent, written $f \equiv g$, if $n = m$ and $f(x_1, \ldots, x_n) = g(x_1, \ldots, x_n)$ for all $x_1, \ldots, x_n \in \mathbb{B}$. Which is the exact same thing as saying that f and g have identical truth tables.

A host of interesting questions present themselves. If you are given two formulas, can you test if they are logically equivalent? How long will this take? Can you find the *shortest* formula ϕ that is logically equivalent to some arbitrary formula ϕ? All wonderful questions.

7 Representing Almost Anything

We have seen how, using nothing but bits (boolean values), you can represent any natural number N. It's pretty efficient, too; try to convince yourself that the representation of a number N in binary will never take more than four times the number of symbols (bits) as representing the same number in decimal.

If we had used *unary* notation, writing N 1's for the number N, this wouldn't have been true: the unary representation would be more than a multiplicative factor longer than the decimal representation. Unary representation is exponentially more verbose, you could say, than using any other radix.

And if you can represent numbers, it seems like you can represent virtually anything. What I type into my computer is a sequence of *characters*, each character given a one-bit (8-bit) code according to a convention known as ASCII. Every text file I make is such. Everything in my computer. Everything in all the computers of the world.

Where does our real interest in logic come? Not so much from trying to translate arguments from English to logic,

> All men are mortal.
> Socrates is a man.
> Therefore, Socrates is mortal.

That's not what anyone cares about. The interest in logic springs more from the viewpoint that a fragment like the one above is but 560 boolean values, each of which is happy to be bent to human will. *Homo logicus*, we increasingly imagine ourselves to be; man as the manipulator of a gazillion passive, subservient bits.

I think there is actually an implicit *thesis* that has taken hold of our collective imagination. It is a belief so widely held that I have never seen mention of it made in any paper or book. Like the water in which the fish swims, that which is total and environmental may go unnoticed. So let me explicitly say what this thesis I see is:

> **Boolean-modeling thesis.** Everything we are interested in can be adequately represented as a finite sequence of bits. Once so represented, the bits can be logically manipulated, using our machines and our ingenuity, to achieve desirable ends.

A "thesis" is something that one believes—perhaps—but cannot prove. In this case, I would call myself a skeptic. Yet I think the boolean-modeling thesis has come to hold a hegemonic grip over our collective imagination. Beware: a thesis might be widely held, yet fundamentally wrong.