Logic 2

Today:

- ☐ Review
- ☐ A formal treatment
- ☐ More boolean operators
- ☐ Equivalences
- ☐ Gates and circuits

1 Review

Last time we introduced the boolean domain $\mathbb{B} = \{0, 1\}$ (or $\mathbb{B} = \{\text{F}, \text{T}\}$, if you prefer) and three operators on it: the binary operators AND and OR, and the unary operator NOT. We talked about *truth tables*, which are a way to represent an arbitrary function $F\colon \mathbb{B}^n \to \mathbb{B}$. A potentially long-winded way, as a truth table for an n-input function will have 2^n rows. We spoke of *logical equivalence*, where boolean formulas $\phi(x_1, \ldots, x_n)$ and $\phi'(x_1, \ldots, x_n)$, which might look quite different, are said to be equivalent, written $\phi \equiv \phi'$, if that they have the same truth table (or, equivalently, if they describe the same function). As an example,

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

We showed an important theorem in logic: that $\{\text{AND}, \text{OR}, \text{NOT}\}$ are, taken together, powerful enough to describe *any* boolean function—or, equivalent, they can capture any truth table. We showed this *constructively*: I described how to take a truth table and "read off" a boolean formula that corresponds to it. In fact, that formula, for an n-variable function, will have a special form: it will be the OR of a bunch of terms (one for each "1" in the truth table), each term being the AND of n *literals*, each literal being one of the variables or, instead, its complement. (A formula in that form is said to be in *disjunctive normal form*, or DNF.) For example, from the table below we can read off the DNF formula $AB\overline{C} \vee A\overline{B}C \vee \overline{A}BC$.

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

(I have omitted a technicality in the description above about passing from a truth table to a DNF formula. Did anyone see it? What if the truth-table has *no* 1's at all—what if it's the constant 0?)

The term that I might have introduced last time (I can't remember) is to say that the set of functions $\mathcal{F} = \{\text{AND}, \text{OR}, \text{NOT}\}$ (or $\mathcal{F} = \{\wedge, \vee, \neg\}$) is *functionally complete*. (I might have slipped and said *logically complete*.) In general, a set of boolean functions $\mathcal{F}$ is *functionally complete* if it is possible to write any boolean function on one or more inputs using only functions from $\mathcal{F}$. You get to use each function from $\mathcal{F}$ as many times as you like.

Besides the above, we also talked about representing numbers in different bases, particularly in binary. I wanted you to order the rows of your truth table as though counting in binary. I said that not only can you represent numbers in binary, but it seems as though you can represent essentially *anything* in binary. I made up a term—the "boolean-modeling thesis"—to capture the thesis that everything we are interested in representing can be adequately represented as a finite sequence of bits. I said that *that* was a significant part of what made logic of contemporary interest in computer science.

End of review!!

2 A Formal Treatment

The word "formal" actually has two meanings in math and computer science: one meaning is "rigorous" and the other is "pertaining to the manipulation of symbols." I'd now like to give a formal treatment of boolean formulas—formal in the *second* sense—because that is in fact the pathway we follow if we want to achieve formality in the first sense. Plus, this provide an early excuse for you to see a *recursive definition*, which is an important thing on its own.

Our formal treatment starts with a set of symbols $\mathcal{P}$ that we call the *propositional variables*. These can be just about anything. For example, we might have $\mathcal{P} = \{a, b, \ldots, z\}$ or $\mathcal{P} = \{A, B, \ldots, Z\}$, or an infinite set of propositional symbols like $\mathcal{P} = \{x_1, x_2, \ldots\}$. Then we say the following.

Definition 1 *A **boolean formula** (alternatively termed a **well-formed formula**, or **WFF**) over the propositional variables $\mathcal{P}$ is any of the following:*

1. *Each propositional symbol in $\mathcal{P}$ is a boolean formula.*
2. *If ϕ is a boolean formula then so is $(\neg\phi)$*
3. *If ϕ and ψ are boolean formulas then so is $(\phi \vee \psi)$*
4. *If ϕ and ψ are boolean formulas then so is $(\phi \wedge \psi)$*

I could add in that "nothing else is a boolean formula," but this isn't actually necessary because when I give a definition it is always understood that the *only* thing that qualifies for the definition

is that which has explicitly been thrown into the pot. Note this is only true when you give a definition—not when you're stating a claim.

Your zyBook calls a boolean formula a *compound proposition*. For some reason that term sounds silly to me. A well-formed formula is actually the standard term.

When I write $(\neg\phi)$, $(\phi \vee \psi)$, or $(\phi \wedge \psi)$ in the definition above, these are just sequence of characters (symbols). We call a finite sequence of characters , each taken from some allowed set of characters called the *alphabet*, we call that a *string*. A boolean formula, in this point of view, is nothing but a string—a list of characters.

The propositional symbols shouldn't include $\neg$, $\wedge$, $\vee$, (, or). Those would make terrible propositional symbols because they would get confused with the syntactic stuff we added.

Our definition of a boolean formula is an example of a *recursive definition.* In our definition we have defined boolean formulas in terms of ... boolean formulas. This sounds a little like saying, I don't know, "a *circle* is a circle." But it's not the same because we made sure to eventually "bottoms out" with the "base case"—which is condition (1).

It is understood that you only get to apply the rules of a recursive definition a *finite* (but unbounded) number of times.

But wait! By this definition, something like $P \wedge Q$ isn't a Boolean formula, because its missing parentheses. The way that mathematicians usually do things it so imagine that a WFF is fully parenthesized, and then allow you, when you are being *in*formal, to omit writing parentheses if it's clear where they should be. And they establish conventions to indicate where omitted parentheses are assumed to fall. For example, $\neg P \vee Q \wedge R \wedge S$, or even $\overline{P} \vee QRS$, are understood as shorthand for $((\neg P) \vee ((Q \wedge R) \wedge S))$. Because of the NOT then AND then OR precedence, with a further understanding that, within a level, grouping is left-to-right.

The syntax of a boolean formula that we have selected was carefully chosen so that if you look at any boolean formula there will be one and only one way to realize it as a sequence of applications of these rules. That isn't obvious—it is something that requires proof. We won't prove it, but see if you can convince yourself that it is true. See also how it would *not* have been true if we had, say, omitted the parentheses in rule (3).

I noticed that the Wikipedia article on well-formed formulas omits the parentheses in rule (2). I found that choice interesting. One would again need check that, with that convention, there is once again a unique way to realize each WFF as a sequence of applications of the (now modified) rules.

One thing I omitted from the definition of a boolean formula was the constants 0 and 1. We certainly could have included them. Alternatively, we can think of 0 as a shorthand for $(p \wedge (\neg p))$ and 1 as a shorthand for $(p \vee (\neg p))$, where p is an arbitrary propositional symbol from $\mathcal{P}$.

But why bother with *any* of this? Because it is how we give rigorous *meaning* to our formulas! The goal is to interpret each WFF ϕ—which, so far, is just a string—as something more interesting—a function.

We do this with the help of a new concept: a *truth assignment*.

Definition 2 *A **truth assignment** for a set of propositional symbol $\mathcal{P}$ is a function $t : \mathcal{P} \to \mathbb{B}$.*

That is, a truth assignment t tells you, for each symbol in $\mathcal{P}$, if, under t, that symbol is true or is false.

For example, going back to $\phi(a, b, c) = (ab \vee bc \vee ac \vee \overline{abc})$, we might have a truth assignment t that says that $t(a) = t(b) = t(c) = 0$. Or it could say that $t(a) = t(b) = 1$ but $t(c) = 0$. There are eight possible truth assignments t for the propositional symbols $\{a, b, c\}$ that appear in ϕ.

We use the recursive definition of a boolean formula and the notion of a truth assignment to give *meaning* to our formulas as follows:

Definition 3 *Fix a set of propositional symbols $\mathcal{P}$. We define a function M that takes in two inputs: an arbitrary truth assignment $t : \mathcal{P} \to \mathbb{B}$ and an arbitrary boolean formula ϕ over the propositional symbols in $\mathcal{P}$. The function returns a boolean value $M_t(\phi)$ that is defined as follows:*

1. *$M_t(x) = t(x)$ for each $x \in \mathcal{P}$.*
2. *$M_t((\neg\phi)) = \neg(M_t(\phi))$.*
3. *$M_t((\phi \vee \psi)) = M_t(\phi) \vee M_t(\psi)$.*
4. *$M_t((\phi \wedge \psi)) = M_t(\phi) \wedge M_t(\psi)$.*

In (2)–(4), the $\neg$, $\vee$, and $\wedge$ symbols that appear to the left of the equal signs are just formal symbols. The $\neg$, $\vee$, and $\wedge$ symbols that appear to the right of the equal signs are the boolean functions whose meaning we have defined. They look the same—but the meaning differs.

We started with a function t and we *extended* it to a function M_t. It's called an extension because M_t agrees with t on t's domain—but M_t has a bigger domain. You can apply M_t not just to propositional symbols, but to *any* boolean formula. And M tells us what a formula ϕ means. Because, for any truth assignment t—that is, for any *row* in a truth—$M_t(\phi)$ is a value—the bit to fill in for that row. Thus $M(\phi)$ is, in essence, the *meaning* that is indicated by ϕ, the *truth table* for ϕ.

I know this is all very formal—formal in the sense of pushing symbols around—but I wanted to develop things this way because this formalistic approach is *how* we invest a formula with meaning. It is our path to making things rigorous. And if we're not adequately rigorous in logic, things blow up in our faces.

You have probably seen hints of this, like the liar's paradox. You know, if I say that everything Harry says is a lie, and then Harry asserts that he is lying, https://bit.ly/3qfJ1Ns. These sorts of paradoxes aren't just silly—they can point to fundamental issues in what we are trying to do.

Anyways, using the recursive definition above, every WFF ϕ (a string) now has a boolean function, $M(\phi)$, associated to it. We often don't even make the distinction: writing ϕ when we really mean $M(\phi)$. Is $(\neg(P \wedge Q))$ a string that is eight character long, or is it a function from two boolean bits to one? It is either, depending on the context. is $\overline{PQ}$ shorthand for the string $(\neg(P \wedge Q))$ or is it the boolean function $M((\neg(P \wedge Q)))$? The context decides.

3 More Booleans Operators

So far we've met three boolean operators: AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). But there are 16 possible binary operators. (Why?) Some of them are important. In fact, I think 14 of the 16 have convenient names. Let's meet some more boolean operators.

Implies We write $p \to q$ if p is false or q is true. That is, $p \to q$ is true unless p is true yet q is false. That truth table thus looks like this:

p	q	$p \to q$
0	0	1
0	1	1
1	0	0
1	1	1

Other ways to write $p \to q$ are: p implies q, p IMPLIES q, if p then q, $\Rightarrow q$, $p \longrightarrow q$, $p \Longrightarrow q$.

Note that the p and q play different roles in $p \to q$; the operator is not *symmetric*. Your book calls p the *hypothesis* and q the *conclusion*. It's not true that $p \to q$ implies $q \to p$.

The implication operator plays a central role in mathematical reasoning. It can be a strategy for showing some proposition Q. Namely, one establishes that P implies Q, and then one establishes P. The proposition Q then follows. This form of reasoning is so central that it has a name: *modus ponens*. It hasn't quite made it into English, so best to retain the italics if you use this phrase.

Paired with *modus ponens* is the method of reasoning that shows the hypothesis P to be false by establishing the implication $P \to Q$ and, also, $\neg Q$. This form of reasoning goes by the name *modus tollens* It too is a common strategy for proofs.

XOR This is every cryptographer's favorite binary operator. We write $p \oplus q$ to mean that p is true and q is false, or, alternatively, q is true and p is false. This is the same as saying that $p \vee q$ is true, but pq is false. And it's the same as saying that p and q have opposite truth values.

p	q	$p \oplus q$
0	0	0
0	1	1
1	0	1
1	1	0

Other ways to write $p \oplus q$ are: p xor q, p XOR q.

If we XOR a whole bunch of variables together, $x_1 \oplus \cdots \oplus x_n$, when is the result true? You should think about this! The answer is that it is true if an *odd* number of the values we are XORing are true. So XOR is essentially a computation of *parity*—the evenness or oddness of something.

What happens if you multiple a whole bunch of -1 values together in the realm of integers, and how does that relate to XOR?

Biconditionals We write $p \leftrightarrow q$ to mean that p and q have the same truth value. That's the exact opposite of XOR; $p \leftrightarrow q \equiv \overline{p \oplus q}$. Here's the truth table:

p	q	$p \leftrightarrow q$
0	0	1
0	1	0
1	0	0
1	1	1

Other ways to write $p \leftrightarrow q$ are: p iff q, p IFF q, $p \Leftrightarrow q$, $p \iff q$. In English, we usually read it as p if and only if q.

NAND If you negate the output of the AND function you get the NAND function. I've seen it written $\overline{\wedge}$.

p	q	$p \overline{\wedge} q$
0	0	1
0	1	1
1	0	1
1	1	0

Other ways to write $p \overline{\wedge} q$ are: p nand q, p NAND q, $p \uparrow q$.

The significance of NAND is that, all by itself, it is functionally complete: you can write any boolean function with just this one building block. We already know that $\{\text{AND}, \text{OR}, \text{NOT}\}$ is logically complete, so if we could write just those three functions using NAND we would have established our claim. Well, it is easy to write NOT, because $\overline{p} \equiv p \overline{\wedge} p$. And it is easy to write AND, because we've just said how to do negation, and AND is the negation of NAND. But disjunction needs a little trick. The trick is called *De Morgan's law*, named after the 19th century British logician Augustus De Morgan. It says that

$$p \vee q \ \equiv \neg(\neg p \ \wedge \neg q).$$

Equivalently,

$$\neg(p \vee q) \ \equiv \neg p \ \wedge \neg q.$$

Make up an English-language example to convince yourself that it is plausible—and verify that it's true with a truth table. Using De Morgan's law, we can write OR using NOT and AND. And we already said we can write NOT and AND using NAND. So we can write any boolean formula with one or more inputs using nothing but NAND.

I should mention that there's a second version of De Morgan's law, the version saying how to write AND using NOT and OR:

$$p \wedge q \equiv \neg(\neg p \vee \neg q)$$

or, equivalently,

$$\neg(p \wedge q) \equiv \neg p \vee \neg q.$$

NOR If you negate the output of the OR function you get the NOR function. I've seen it written $\overline{\vee}$.

p	q	$p \overline{\vee} q$
0	0	0
0	1	1
1	0	1
1	1	1

Other ways to write $p \overline{\vee} q$ are: p nor q, p NOR q, $p \downarrow q$.

Just like the NAND function, NOR, all by itself, is logically complete. Use De Morgan's law—the other one—to show this.

Constant functions Don't forget the constant function 0 that always returns false; and the constant function 1 that always returns true. It's a bit of an abuse of notation to write 0 and 1 for those functions, because 0 and 1 are boolean constants, not functions. But that's the most natural nomenclature, seems to me.

Additional functions I believe that leaves only two boolean functions unnamed. Find the buggers! Then, try to figure out if they are logically complete.

4 Equivalences

We have mentioned De Morgan's law. There are a bunch of these "laws"; see Figure 1.

One important law that is not listed in the table above is the fact that

$$p \to q \equiv \neg q \to \neg p.$$

This one comes up so often as to have a name: the *contrapositive* of $p \to q$ is $\neg q \to \neg p$. An implication and it's contrapositive are equivalent. What is more, showing an implication by showing its contrapositive is sometimes a good proof strategy.

Another useful law, mentioned already, is that

$$p \leftrightarrow q \equiv (p \to q) \wedge (q \to p).$$

One can take this as a tactic for showing a biconditional.

5 Gates and Circuits

There is a different viewpoint as to what our boolean operators are: logic gates. Instead of thinking of a world $\mathbb{B}$ with two values, these values abstractly manipulated by operators, we think of signals flowing through a digital circuit, these signals manipulated by gates. Typically a low voltage (e.g., 0V–1.2V) is used as a physical representation of 0; a high voltage (e.g., 3.3V–5V ground) is used as a physical representation of 1; and we design our circuits so as to

Table 1.6.1: Laws of propositional logic.

Idempotent laws:	$p \vee p \equiv p$	$p \wedge p \equiv p$
Associative laws:	$(p \vee q) \vee r \equiv p \vee (q \vee r)$	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
Commutative laws:	$p \vee q \equiv q \vee p$	$p \wedge q \equiv q \wedge p$
Distributive laws:	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
Identity laws:	$p \vee F \equiv p$	$p \wedge T \equiv p$
Domination laws:	$p \wedge F \equiv F$	$p \vee T \equiv T$
Double negation law:	$\neg\neg p \equiv p$	
Complement laws:	$p \wedge \neg p \equiv F$ $\neg T \equiv F$	$p \vee \neg p \equiv T$ $\neg F \equiv T$
De Morgan's laws:	$\neg(p \vee q) \equiv \neg p \wedge \neg q$	$\neg(p \wedge q) \equiv \neg p \vee \neg q$
Absorption laws:	$p \vee (p \wedge q) \equiv p$	$p \wedge (p \vee q) \equiv p$
Conditional identities:	$p \rightarrow q \equiv \neg p \vee q$	$p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$

Figure 1: Table 1.6.1 from your zyBook. Most of these "laws" are straightforward, and all can be verified by making a truth table. Many have analogs in the world of integers, with operations of plus and times in place of OR and AND.

never employ intermediate voltages. See Figure 2 for the symbols routinely used to represent 1- and 2-input logic gates.

If you are trying to design or understand a complex piece of logic, the circuit representation can be a convenient visualization. We may do an example of this next time. But it is no more or less expressive than writing formulas. It is easy to go from one representation of a function to another.

Not every circuit you could draw is meaningful: it is interesting to think about what happens if the output of a gate is used as an input to the same gate or, more generally, if there is a "path" from the output of a gate back to its input. If you are expressing a boolean formula as a circuit you will never get any "cycles."

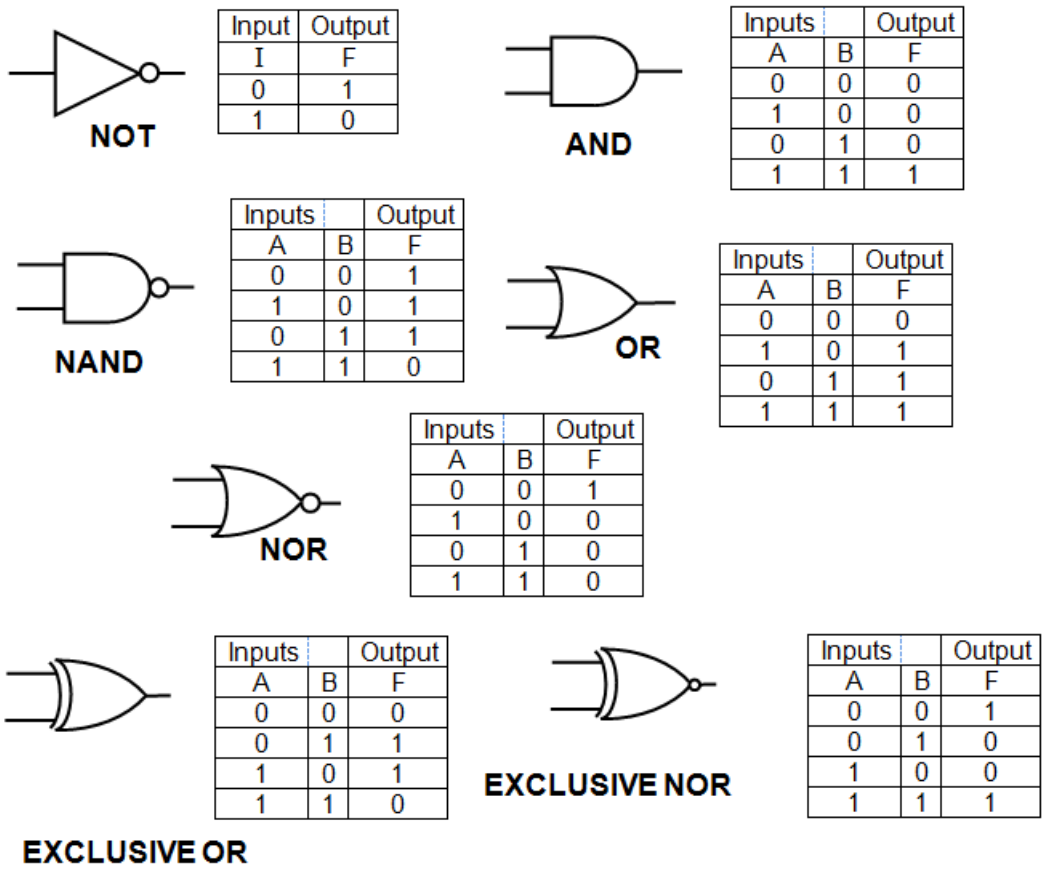

Figure 2: Symbols for logic gates. While less conventional, it would be quite clear to draw a rectangle and put the word AND, OR, NOT, NAND, NOR, XOR inside.