Lecture 2: Logic

In this lecture, we introduce logic — to be precise, the kind that is also called Boolean logic, propositional logic, or sometimes zeroth-order logic (as opposed to first-order logic that we will look at later).

Imagine a microwave oven, modelled as a system with two components: a magnetron (the thing that produces microwaves) that can be on or off, and a door that can be open or closed. You are responsible for writing a risk assessment and make the following table of all possible states of the two components:

Magnetron	Door	Safe?
off	closed	ok
on	closed	ok
off	open	ok
on	open	no!

From this you can deduce a safety requirement for the designers: *if the door is open, then the magnetron must be off*. Real microwave ovens typically have some kind of interlock to enforce this property.

What if you wanted to reason about safety properties of a much bigger system, such as a processor? The Apple M1 Ultra has over 100 billion transistors, each of which can be on or off. Checking all possible states will not work here, but logic can help you to reason about parts of a larger system and prove safety properties.

Propositions

A *proposition* is an English[1] sentence that can be true or false. 'London is the capital of England.' is a true proposition, 'London is the capital of France.' is a false proposition. 'It is raining.' is also a proposition because, at any one place and time where you check it, it will be true or false, but its truth value obviously depends on when and where you check and how you define rain (does drizzle count?). 'How do you do?', 'Oh my!', and 'Hmmmph' are not propositions, because they do not have a truth value.

Generally, if you can put 'It is true that …' or 'Is it true that ….. ?' before or around an English sentence and it still makes grammatical sense, then that sentence is a proposition.

A simple proposition usually consists of a *subject* (often a noun or a noun-phrase) and a *predicate* which is something that can apply to a subject, often with a form of the verb 'to be'. Thus, 'is the capital of England' or 'is red' are predicates, and 'Roses are red' is a proposition.

A compound proposition is a sentence built up out of one or more simple propositions, connnected with *conjunctions* such as 'and', 'or', 'if …then'. Mathematically, simple propositions are like values, and conjunctions are logical operators.

For example, 'the magnetron is on' is a simple proposition with 'the magnetron' as its subject and 'is on' as its predicate, and 'if the magnetron is on then the door is closed' is a compound proposition.

Propositional logic is about studying relationships between propositions. Individual propositions take the role of variables, and we can prove relationships such that 'if p is true then r is true' and

[1] Of course, logic would work just the same in another language; many of the concepts were originally defined in German.

'if q is true then r is false' imply that p and q cannot both be true at once, even if we do not know which of the three propositions (if any) are true.

To makes this study mathematically precise, conjunctions in logic have mathematical definitions that do not always match everyday English usage. For example, 'Would you like tea or coffee?' normally implies 'one or the other but not both', but in logic, unless stated explicitly otherwise, 'or' always includes 'or both'. Thus, a logician would be mathematically correct in answering 'Yes' to this question unless they disliked both drinks!

Propositional Logic

Mathematically, propositional logic is built on these blocks:

- Instead of numbers, there are only two values called T (true) and F (false). Some books also write them as 1 and 0. This set of values is often written $\mathbb{B}$ for 'Booleans' after the logician George Boole.
- Instead of the usual addition, multiplication etc. there are a new set of logical operations.

Terms in logic work just like terms in arithmetic, although it is convention to call them 'formulas[2]' instead of terms. Formulas are trees with operation nodes that have children, and variable and value nodes that do not have children. There is a syntax and a semantics, and various ways of printing and parsing formulas. However, logical formulas are even easier to work with than arithmetic ones in two ways: first, there is no such thing as dividing, so you cannot divide by 0 by accident — all logical formulas without variables always evaluate to a truth value. Secondly, because there are only two truth values, if a formula does not have too many variables then 'check all possible assignments' is a valid strategy for some calculations.

To avoid confusion, when we have variables for formulas we will never use the letters T, F as variables, as these already stand for the constant values true and false.

Implication: if . . . then

Our first logical operation that we study is logical implication, which we write $a \to b$ and pronounce 'if a then b'. We can list all possible values of this operation:

$$T \to T = T \quad T \to F = F \quad F \to T = T \quad F \to F = T$$

Or, the same thing again as a table, which is the usual way to list logical operations:

Definition 1 (logical implication $\to$)**.** The operation $a \to b$ is defined as:

a	b	$a \to b$
F	F	T
F	T	T
T	F	F
T	T	T

[2] Or 'formulae', which is the correct Latin plural; in English, both versions are acceptable.

The intuition here is that in the microwave example, if a means 'the magnetron is on' and b means 'the door is closed', then $a \to b$ models 'if the magnetron is on, then the door is closed' which is true in 3 out of 4 possible states, and false in the state where this safety property is violated.

The meaning of 'if . . . then' in logic is always that given by the table above, even where this conflicts with normal English usage. For example, 'if it is sunny then I will go to the park' in normal English carries the connotation 'and if it rains, then I won't' but in the microwave example, there is no connotation of 'and if the door is closed then the magnetron is on'. Indeed, most microwaves spend most of their lives with the door closed and the magnetron turned off.

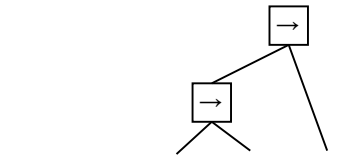
Looking again at the table for $a \to b$, we see that $a \to b$ is true if either of the following two hold (or both): a is false, or b is true. Therefore, in logic, an implication whose left-hand side is false is always true: 'if Paris is in England, then the Earth is flat' is logically true because Paris is not in England, and therefore the proposition says nothing about the Earth, just like seeing a microwave oven with a closed door says nothing about whether its magnetron is on or off. Similarly, you can deduce that 'if Nyon is in France, then the Earth goes around the Sun' is logically true if you know that the Earth goes around the Sun, without knowing anything about the geography of Nyon (it is actually in Switzerland).

Truth Tables

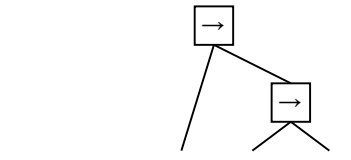
A truth table, for a logical formula with variables, is a table with one row per possible state of these variables (that is, a possible environment in which to evaluate this formula). For example, a logical statement with 2 variables will have 4 rows (not counting the header row).

Two logical formulas are semantically equivalent ($U \equiv V$) if they evaluate to the same value in every possible environment (logical formulas cannot evaluate to $\perp$ so we do not need a special rule for this case anymore). This means that to check if two logical formulas are semantically equivalent, we can create truth tables for them: if the tables end up with the same values for the formula's column, then the formulas are equivalent; if the tables differ anywhere in the formula's column, then they are not equivalent.

We can create a truth table for any logical formula by evaluating it step by step. For example, suppose we want to know if logical implication is associative, that is if $(a \to b) \to c$ is semantically equivalent to $a \to (b \to c)$.



a	b	c	a	b		c
F	F	F	F	T	F	F	F
F	F	T	F	T	F	T	T
F	T	F	F	T	T	F	F
F	T	T	F	T	T	T	T
T	F	F	T	F	F	T	F
T	F	T	T	F	F	T	T
T	T	F	T	T	T	F	F
T	T	T	T	T	T	T	T



a	b	c	a		b	c
F	F	F	F	T	F	T	F
F	F	T	F	T	F	T	T
F	T	F	F	T	T	F	F
F	T	T	F	T	T	T	T
T	F	F	T	T	F	T	F
T	F	T	T	T	F	T	T
T	T	F	T	F	T	F	F
T	T	T	T	T	T	T	T

For the table on the left representing $(a \to b) \to c$, first we note there are three variables so there are 8 possible environments (if you replace F with 0 and T with 1, the environments are counting from 000 to 111 in binary). We can copy the a, b, c columns over to the right-hand side of the left table for the three value formulas. The second column of the five on the right, representing $a \to b$, can then be filled in by applying the operation to the first and third columns; similarly the fourth column can be filled in by doing the $\to$ operation on the second and fifth columns. For example, in the first row, the second column is T, the fifth column is F, and $T \to F = F$ so we put a F in this row. In the second row, the second column is T and the fifth column is T, and $T \to T = T$ so we put a T in this row, and so on.

The column for $(a \to b) \to c$ is the fourth column of the right-hand part of the left table (highlighted in bold text) and the column for $a \to (b \to c)$ is the bolded second column of the right-hand side of the right table. We can see that these two columns are not the same, so we have

$$(a \to b) \to c \not\equiv a \to (b \to c)$$

because they differ for example in the environment $\{a = F, b = F, c = F\}$ when the left-hand formula is F and the right-hand formula is T. Therefore, $\to$ is not associative.

Two-sided Implication

We stated that 'if a, then b' in logic does not say anything about the case when a is false. Logic has a separate operation for an 'if' that works in both directions: it is written $a \leftrightarrow b$ and pronounced 'a if and only if b'. Some mathematics books write this with the made-up word 'iff' with a doubled letter f, but when reading this, you still read 'if and only if'. The truth table for this operation:

Definition 2 (two-sided implication)**.** The operation $a \leftrightarrow b$ is defined as:

a	b	$a \leftrightarrow b$
F	F	T
F	T	F
T	F	F
T	T	T

Unlike $\to$, this operation is associative, as you could check with a truth table.

And, Or, Not

Here are some more standard logical operations. The logical not operation, written $\neg$, is unary and reverses its input:

Definition 3 (logical negation $\neg$)**.** The unary operation $\neg a$ is defined as:

a	$\neg a$
F	T
T	F

Logical and, written $a \wedge b$, is true when both inputs are true and otherwise false; logical or, written $a \vee b$, is false when both inputs are false and otherwise true. This is an inclusive or in the sense that 'a or b' means 'a or b or both'; the exclusive or operation $a \oplus b$ (sometimes also written and pronounced 'xor') is true if and only if exactly one of its two inputs is true.

Definition 4 (logical $\wedge$, $\vee$ and $\oplus$)**.** The logical operations $\wedge$, $\vee$ and $\oplus$ are:

a	b	$a \wedge b$	$a \vee b$	$a \oplus b$
F	F	F	F	F
F	T	F	T	T
T	F	F	T	T
T	T	T	T	F

There is a convention for precedence[3] of logical operators, just like for arithmetic ones. In the arithmetic term $1 + 2 \times 3$, we say the multiplication has higher precedence because it is performed first by convention, so the result is 7 and not 9. Similarly, in logic,

Definition 5 (precedence of logical operators)**.** The precedence of logical operators is:

- Negation ($\neg$) has the highest precedence.
- And ($\wedge$) is one precedence level lower.
- Or ($\vee$) is one precedence level lower again.
- Not all books agree on, or even define, precedence for other operators, but we will state that $\rightarrow$ has lower precedence than $\vee$, and $\leftrightarrow$ has lower precedence yet again.

Binary operators further bind to the right, unlike arithmetic operators, that is $a \wedge b \wedge c$ means $a \wedge (b \wedge c)$ etc.

The highest precedence of negation means that it always applies to the smallest possible formula following it, so $\neg a \vee b$ means $(\neg a) \vee b$ and not $\neg(a \vee b)$, if you want the latter formula, or any other change to the conventions, then you need to use brackets. For more examples, $a \vee \neg b \wedge c$ means $a \vee ((\neg b) \wedge c)$ and $a \rightarrow b \leftrightarrow \neg a \vee b$ means $(a \rightarrow b) \leftrightarrow ((\neg a) \vee b)$.

Note that precedence rules are rules for the infix notation of formula strings, that express how they are converted to formulas. Terms themselves need neither brackets nor precedence rules since they are abstract mathematical objects whose 'tree structure' defines how they are to be interpreted.

Similarly, we define that logical implication binds to the right, that is $a \rightarrow b \rightarrow c$ means $a \rightarrow (b \rightarrow c)$ and not the other way round. The same holds for all other binary operators, but $\wedge, \vee$ and $\leftrightarrow$ are all associative so semantically, this does not make a difference. It is a fact of logic itself that $\rightarrow$ is not associative and so $a \rightarrow (b \rightarrow c)$ and $(a \rightarrow b) \rightarrow c$ are not equivalent; it is purely a convention that writing the formula without brackets at all means the former and not the latter. If you were to write logical formulas some other way, such as S-expressions or in RPN, then you would not need any bracketing conventions, but the laws of logic would still be the same.

[3] Some books talk about *priority* instead, which is the opposite of *precedence*. In arithmetic, multiplication has higher precedence than addition, which is the same thing as lower priority (that is, it appears lower down the term tree).

Notation and Types

We need to talk about two related problems around notation in logic. The first problem is that different books use different notation for the same thing. Depending on whether you are a mathematician, an electrical engineer, or a computer scientist, you might see the same logical formula written as $a \wedge b \vee c$, $a \,\&\&\, b \| c$ or $ab+c$ (some people use addition and multiplication signs for logical operations too). This gets even worse when you notice that some people use $\rightarrow$ for implication, some use $\Rightarrow$ and others use $\rightarrow$ for implication and $\Rightarrow$ for a different concept in the same text. Similarly the use of $=$ and $\equiv$ (and sometimes also $\Leftrightarrow$) is not consistent.

The second problem is that different kinds of strings parse to mathematical objects of different types. Going back to arithmetic for a moment, we know that 2 and $1+x$ are terms (or to be precise, term strings), but $1+x=2$ is not a term: it is an equation (string). Thus, $+$ is an operation that takes two values and returns a value, but if you want to talk about $=$ as an operation then it takes two terms and returns something of a different type, namely an equation. You can do different kinds of things with equations such as subtracting 1 on both sides, or more generally solving an equation; you cannot solve a term like $1+x$.

In our notation for logic, $\wedge, \vee, \oplus, \neg, \rightarrow, \leftrightarrow$ are all operations that take one or two logical values (true or false), and so we can use them to build logical formulas. However, $=, \widehat{=}, \equiv$ are not logical operations. They can be used to combine two logical formulas into another type of object, but they can never appear inside a logical formula.

The reason that this is more of a problem in logic than arithmetic is that there are actually two levels of logic going on here. In arithmetic, no-one would say that for example '$2(x+1)$ is true' but one can very much say that '$2(x+1) = 2x+2$ is true': truth values can be applied to arithmetic equations, but not to arithmetic terms[4]. When we study arithmetic and say that a certain rule is true, what we are really doing is using logic as a tool to study the subject of arithmetic. But when we study logic, we are using logic as a tool to study the subject of logic, so one can both say that the formula $T \vee F$ is true, and that $T \vee F \equiv T$ is true. But $T \vee F \equiv T$ is not a logical formula, it is a logical equivalence (something a bit like an equation). As long as you are clear what objects are logical formulas and what objects are equivalences, you can apply truth values to both of them.

So, to summarise, the following are two different types of mathematical objects:

1. For logical values a, b, the operation $a \leftrightarrow b$ produces another logical value c that is true if a, b are the same (both true, or both false) and false if a, b are different. For logical formulas A, B, one can build a new formula $A \leftrightarrow B$ which evaluates to true if both A, B evaluate to the same value, and evaluates to false if A, B evaluate to different values.
2. For logical formulas A, B, the statement $A \equiv B$, written out 'A is semantically equivalent to B', means that A and B both have the same truth value. This statement itself is not a logical formula, and $\equiv$ is not a logical operation.

Of course, there is a relationship between the two, namely $A \equiv B$ is equivalent to saying that $A \leftrightarrow B$ is true (that is, $A \leftrightarrow B \equiv T$). But $A \leftrightarrow B$ is of type 'formula' and so can be combined with other formulas to make bigger formulas, whereas $A \equiv B$ is, if you had to give it a type, a 'statement about formulas' that is not, itself, a formula.

[4] The C programming lanugage lets you branch on the truth value of an integer value with `if (x) {...}`: here 0 counts as false, everything else counts as true. But this is a convention that we do not follow in mathematics or logic. In fact, many other programming languages disallow this because it is a source of bugs; if you want to do the same thing you have to write `if (x != 0) {...}` or something like that which produces a boolean rather than an integer value.

Syntax and Semantics of Propositional Logic

Remember from last lecture that the *syntax* of a mathematical system is a list of rules for building terms, and a list of rules for transforming terms; whereas the *semantics* of a system is a function for evaluating terms to produce values (with the help of an environment, if variables are involved). Two terms are syntactically equivalent ($\widehat{=}$) if you can turn one into the other and back again with the syntax rules, and two terms are semantically equivalent ($\equiv$) if they both evaluate to the same value in any environment that defines all the variables involved.

In propositional logic, unlike arithmetic, one can actually give a complete and sound list of syntax rules so that $\widehat{=}$ and $\equiv$ become equivalent, and one can even prove this. One reason this works is that there are only finitely many values (to be precise, two) rather than the infinite number of natural numbers.

Thus, the following laws of logic can be read in two ways: as syntax rules that we can apply in both directions, or as statements about the semantics. In either case, the proof that each of these rules is sound is simply to create a truth table for both sides and check that the columns for both formulas are identical.

You might wonder why we need syntax rules in propositional logic if we can just check all possibilities. As soon as you have a logical formula with say 200 variables, this would mean checking 2^{200} possible environments, which is simply not possible. However, a syntax rule with for example two variables lets you change parts of a larger formula, so for example using $A \wedge B \vdash B \wedge A$ you can change $(A \wedge B) \wedge C$ for $(B \wedge A) \wedge C$ even if A, B, C contain 200 variables each; as long as you have checked that $A \wedge B \equiv B \wedge A$ which takes only 4 rows of truth table, this will not change the semantics of the larger formula.

Laws of Logic

In a general sense, any statement $A \equiv B$ that is true is a law of logic, but some laws are useful enough that they get their own names.

The laws given here are based around the three operations $\wedge, \vee, \neg$ as we will see later that these are enough to express all possible logical operations.

- The *identity laws* are $a \wedge T \equiv a$ and $a \vee F \equiv a$.
  These are the counterpart to $a + 0 = a$ in arithmetic.
- The *domination laws* are $a \vee T \equiv T$ and $a \wedge F \equiv F$.
  These are roughly the counterpart to $0 \times a = 0$ in arithmetic.
- The idempotent laws are $a \wedge a \equiv a$ and $a \vee a \equiv a$.
- The double negation law is $\neg(\neg a) \equiv a$.
- The commutative laws are $a \vee b \equiv b \vee a$ and $a \wedge b \equiv b \wedge a$.
  These are the counterpart to $a + b = b + a$ in arithmetic.
- The associative laws are $(a \vee b) \vee c = a \vee (b \vee c)$ and the same for $\wedge$.
  These are the counterpart to $(a + b) + c = a + (b + c)$ in arithmetic.
- The distributive laws are $a \vee (b \wedge c) \equiv (a \vee b) \wedge (a \vee c)$ and $a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c)$.
  These are the counterpart to $a(b + c) = ab + ac$ in arithmetic, but note that in logic, both of

$\wedge, \vee$ distribute over each other whereas in arithmetic, if we put the operations the other way round, $a + bc \neq (a + b)(a + c)$.

- DeMorgan's laws are $\neg(p \wedge q) \equiv \neg q \vee \neg p$ and $\neg(p \vee q) \equiv \neg p \wedge \neg q$.
  In words, 'pulling a $\neg$ inside or outside a bracket flips the other operators'.

Law of the Excluded Middle

The *Law of the Excluded Middle*, often also called by its Latin name *tertium non datur* (literally 'there is no third [option]') says that any logical statement is either true or false, but never both.

Definition 6 (Law of the Excluded Middle)**.** For any logical formula A, we have $A \vee \neg A \equiv T$ and $A \wedge \neg A \equiv F$.

This law is special enough to get its own section, because there are other logics where it does not hold. For example, the moment a formula can evaluate to $\perp$ like in arithmetic when you try and divide by zero, this law would no longer hold. Logics without this law are used in computing both in databases (where values can be true, false, or null) and in hardware design (where a wire can be on, off, or disconnected) among other places.