Chapter 1

Logic

> All opinions are not equal. Some are a very great deal more robust, sophisticated and well supported in logic and argument than others.
>
> Douglas Adams

1.1 Introduction

Let's spell out two logic puzzles:

Sam has 1 cow. If Sam has at least 2 cows, then Sam can breed the cows to make one more cow. Assuming Sam has access to infinite resources and time, how many cows can Sam make?

Two givens: knights always tell the truth, and knaves always lie. On the island of knights and knaves, you are approached by two people. The first one says to you, "we are both knaves." What are they actually? (from Popular Mechanic's Riddle of the Week #43: Knights and Knaves, Part 1 [1])

Thinking logically about these puzzles will help you – think about what can and cannot happen; what can and cannot be true. Solving these problems is left as an exercise. There are only two possibilities for Boolean statements – True or False. Here's a formal definition of logic, from Merriam-Webster:

Definition 1.1.1 Logic
A science that deals with the principles and criteria of validity of inference and demonstration : the science of the formal principles of reasoning [**logic**]

This chapter includes a wealth of topics. We will touch on propositional logic and its corollaries, as well as predicate logic and it's applications to the rest of this course. Think-

ing logically should be a natural process, so we hope this section is relatively straightforward.

1.2 Propositional Logic

Propositional logic is a branch of logic that deals with simple propositions and logical connectives. Sometimes propositional logic is referred to as **zeroth-order logic**, as it lays the foundations for *predicate logic*, also known as *first-order logic*.

Definition 1.2.1 Proposition
A statement that is exclusively either true or false. A proposition must *have* a true or false value, and it cannot have *both*. Propositions are usually represented as variables. These variables can represent a single statement, or large compound statements. We will see examples later

Definition 1.2.2 Logical Connective
An operation that connects two propositions. We study these below

The motivation behind propositional logic is that we want to represent basic logical statements as an expression of variables and operators. Propositional logic also lays the groundwork for higher-order logic.

Before we start, let us motivate propositional logic and Boolean algebra by recalling the fundamental ideas of mathematical concepts we are already familiar with. For standard arithmetic, we utilize real numbers $\mathbb{R}$, binary (two inputs) operators $+,-,\times,\div,\cdot\cdot$, and unary (one input) operators $\exp(\cdot)$, $\log(\cdot)$, $\sqrt{\cdot}$, $\text{abs}(\cdot)$, $(-1)(\cdot),\cdot\cdot$. Note that our operators here take elements from our main set, and return an element back inside that set. For example, $4.01+3.99$ gives us 8. We have a way to signal that those two quantities above are the same, namely the equals sign $=$. Finally, we have a way to abstract out elements as *variables*, for example $x + y = z$. And if we fill in some of the variables, we can solve for the other ones. For example $x + 3.99 = 8$ implies $x = 4.01$. In a similar way, this whole mathematical system can be completely abstracted, and we can reason about those abstract structures. Then, any specific example that holds the same inherent properties of our structures will also have any reasoned theorem apply. This is quite an important idea in mathematics – we can abstract concepts and reason about abstract structures that can apply to a variety of specific examples.

$$(\ \_\_\ \ \square\ \ \_\_\ ) \mapsto \_\_$$

Figure 1.1: In abstract mathematics, we care about *operators* that *map* elements from a set into another element in the same set. For addition in the reals, this would look like $(4.01 + 3.99) \mapsto 8$.

Why does this apply here? Well, for propositional logic, we are, essentially, going to *define* a new *structure*. Our new "number set" becomes $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$ (analogous to $\mathbb{R}$). Our new "unary operators" become $\neg$ (analogous to multiplying by -1, etc). Our new "binary operators" become $\wedge, \vee, \Longrightarrow, \cdots$ (analogous to $+, -, \times, \div, \cdots$). Our new "equality" becomes *logical equivalence* ($\equiv$ versus $=$). Our idea of *variables* stays the same. But, interestingly, because our input set $\mathbb{B}$ is *finite* (which is different from the countably infinite $\mathbb{Z}$ and the uncountable $\mathbb{R}$). This gives us nice ways to actually *write out all possibilities* for inputs and outputs of our operators. With this in mind, let us push onward to propositional logic.

1.2.1 Truth Tables and Logical Connectives

Before we dive into the logical connectives, let's study the notion of a truth table. This will help us fully understand the logical connectives.

Definition 1.2.3 Truth Table
A table that shows us all truth-value possibilities. For example, with two propositions p and q:

p	q	*some compound proposition*
F	F	T/F
F	T	T/F
T	F	T/F
T	T	T/F

Now we can begin our study of the logical connectives. The following definitions explain the intuition behind the logical connectives, and present their associated truth tables.

Definition 1.2.4 And $\wedge$
Also known as the *conjunction*. Logical connective that evaluates to true when the propositions that it connects are both true. If either proposition is false, then *and* evaluates to false. To remember: *prop 1* **and** *prop 2* must *both* be true. Truth table:

p	q	$p \wedge q$
F	F	F
F	T	F
T	F	F
T	T	T

Notice the only row that evaluates to true is when both propositions are true

Definition 1.2.5 Or $\vee$
Also known as the *disjunction*. Logical connective that evaluates to true when either of the propositions that it connects are true (at least 1 of the connected propositions is true). If both propositions are false, then *or* evaluates to false. **Note**: if both propositions are true, then *or* still evaluates to true. To remember: either *prop 1* **or** *prop 2* must be true. Truth table:

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

Notice the only row that evaluates to false is when both propositions are false

Definition 1.2.6 Not $\neg$, $\sim$
Also known as the *negation*. Logical connective that flips the truth value of the proposition to which it is connected. Unlike *and* and *or*, *not* only affects 1 proposition. Truth table:

p	$\neg p$
F	T
T	F

Definition 1.2.7 Implication/Conditional $\Rightarrow$, $\rightarrow$
Logical connective that reads as an *if-then* statement. The implication must be false if the first proposition is true and the implied (connected/second) proposition is false. Otherwise it is true. Truth table:

p	q	$p \Rightarrow q$
F	F	T
F	T	T
T	F	F
T	T	T

Note: the direction of an implication can be flipped: $p \Leftarrow q$ is the same as $q \Rightarrow p$

Let's try to understand the truth table for the implication statement before we continue. We present two examples that attempt to form an intuitive analogy to the implication.

Example: Think of the implication as a vending machine. p is the statement *we put money into the vending machine*, and q is the statement *we received a snack from the vending machine*. Notice that the statements do not necessarily depend on each other. We examine the four cases and see when we are *unhappy*:

1. p is **false** and q is **false** – we did not put in money, and we did not get a snack, so we remain happy (normal operations)
2. p is **false** and q is **true** – we did not put in money, and we did get a snack, so we are very very happy (free snack!)
3. p is **true** and q is **false** – we did put in money, and we did not get a snack, so we are very very unhappy (we got robbed!)
4. p is **true** and q is **true** – we did put in money, and we did get a snack, so we are happy (normal operations)

When we are unhappy, then the implication statement is false. Otherwise it is true (we are not *unhappy*).

Example: Think of the implication in the lens of a program. You want to evaluate whether your program *makes sense*. Here is the example program from the statement $p \Rightarrow q$:

```
(...)
if (p is true) {
        <run body code if q is true>
}
(...)
```

The body code is run only if q is true. Now let's examine the 4 cases and see whether the program makes sense:

1. p is **false** and q is **false** – the program does not go into the body of the if-statement and hence makes sense
2. p is **false** and q is **true** – the program again does not go into the body of the if-statement and hence makes sense (regardless of the value of q)
3. p is **true** and q is **false** – the program goes into the body of the if-statement but since q is false the program does not evaluate the body code. This does not make sense
4. p is **true** and q is **true** – the program goes into the body of the if-statement and evaluates the body code. This makes sense

When the code evaluator makes sense, then the implication statement is true.

Definition 1.2.8 Bi-conditional $\Leftrightarrow$, $\leftrightarrow$
Logical connective that reads as an *if and only if* statement. This means that both propositions must imply each other. For the bi-conditional to be true, both propositions must either be true or false. Truth table:

p	q	$p \Leftrightarrow q$
F	F	T
F	T	F
T	F	F
T	T	T

Some more complicated ones:

Definition 1.2.9 Exclusive Or (*Xor*) $\oplus$
Logical connective that evaluates to true when *only* one of the two propositions that it connects is true. If both propositions are true or false, then *xor* evaluates to false. The exclusive part means we *exclude* the *or* case when both propositions are true. Truth table:

p	q	$p \oplus q$
F	F	F
F	T	T
T	F	T
T	T	F

Definition 1.2.10 Exclusive Nor (*Xnor*) $\otimes$
Logical connective that negates the *xor*. It is just an *xor* connective appended with a *not* connective. Truth table:

p	q	$\neg(p \oplus q) \equiv (p \otimes q)$
F	F	T
F	T	F
T	F	F
T	T	T

Now an example:

Example: Translate the following statement into a propositional logic statement: *exclusively either the weather rains or students wear rain jackets*.

Solution: Let r be the proposition *the weather rains* and j be the proposition *students wear rain jackets*. Then the statement becomes $r \oplus j$

1.2.2 Boolean Algebra

The motivation behind Boolean algebra is that we want to take complicated compound propositional statements and simplify them. If we notice that a variable does not affect the final output, then getting rid of that variable cuts the amount of truth-value possibilities (truth-table rows) in half.

Definition 1.2.11 Boolean Statement
A statement that is exclusively either true or false

Some handy notations:

Definition 1.2.12 Equivalence $\equiv$
Logical equivalence says that the two connected statements are logically the same. You can think of this notation as the *equals* sign. Equality is poorly defined for Boolean expressions, so we use the equivalence notation instead

Definition 1.2.13 Tautology t *or* **T**
A proposition that is always true

Definition 1.2.14 Contradiction c *or* **F**
A proposition that is always false

We provide a handful of helpful theorems to aid in your Boolean algebra simplifications. You do not need to memorize these theorems – they will be given to you as a table.

See appendix **??**

Theorem 1.2.1 Commutativity
*For any propositions p and q the **and** and **or** operations are commutative:*

$$p \vee q \equiv q \vee p$$

$$p \wedge q \equiv q \wedge p$$

Theorem 1.2.2 Associativity
*For any propositions p, q, and r the **and** and **or** operations are associative:*

$$(p \vee q) \vee r \equiv p \vee (q \vee r)$$

$$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$$

Theorem 1.2.3 Distributivity
*For any propositions p, q, and r the **and** and **or** operations are distributive:*

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

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

Theorem 1.2.4 Identity
For any proposition p the following hold:

$$p \vee c \equiv p$$

$$p \wedge t \equiv p$$

Theorem 1.2.5 Negation
For any proposition p the following hold:

$$p \vee \neg p \equiv t$$

$$p \wedge \neg p \equiv c$$

Theorem 1.2.6 Double Negation
For any proposition p the following holds:

$$\neg(\neg p) \equiv p$$

Theorem 1.2.7 Idempotence
For any proposition p the following hold:

$$p \vee p = p$$

$$p \wedge p = p$$

Theorem 1.2.8 De Morgan's
For any propositions p and q the following hold:

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

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

Theorem 1.2.9 Universal Bound
For any proposition p the following hold:

$$p \vee t \equiv t$$

$$p \wedge c \equiv c$$

Theorem 1.2.10 Absorption
For any propositions p and q the following hold:

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

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

Theorem 1.2.11 Negation of Tautology and Contradiction
The following hold:

$$\neg t \equiv c$$

$$\neg c \equiv t$$

Here are two important theorems that you will use throughout your proofs in this course. The first theorem says that for a bi-conditional both propositions must imply each other. The second theorem gives an equivalence between the implication and *or* connective. The proofs of these theorems are left as an exercise to the reader.

Theorem 1.2.12 Bi-conditional to Implication
For any propositions p and q the following hold:

$$p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$$

Theorem 1.2.13 Implication to Disjunction
For any propositions p and q the following holds:

$$p \Rightarrow q \equiv \neg p \vee q$$

Now some examples of Boolean statement simplification.

Example: Simplify the following expression: $(p \Rightarrow q) \Rightarrow r$

Solution:

$$
\begin{aligned}
(p \Rightarrow q) \Rightarrow r &\equiv \big(\neg(p \Rightarrow q)\big) \vee r && \text{Implication to Disjunction} \\
&\equiv \big(\neg((\neg p) \vee q)\big) \vee r && \text{Implication to Disjunction} \\
&\equiv ((\neg(\neg p)) \wedge (\neg q)) \vee r && \text{De Morgan's} \\
&\equiv (p \wedge (\neg q)) \vee r && \text{Double Negation}
\end{aligned}
$$

Notice how we reference a theorem in each step. This allows us to fully explain our equivalence, keeps us from making mistakes, and ensures our equivalence is valid.

Example: Simplify the following expression: $(p \otimes q) \wedge p$

Solution:

$$
\begin{aligned}
(p \otimes q) \wedge p &\equiv \neg(p \oplus q) \wedge p && \text{XNOR equivalence} \\
&\equiv \neg\big((p \wedge (\neg q)) \vee ((\neg p) \wedge q)\big) \wedge p && \text{XOR equivalence} \\
&\equiv \big(\neg(p \wedge (\neg q)) \wedge \neg((\neg p) \wedge q)\big) \wedge p && \text{De Morgan's} \\
&\equiv \big(((\neg p) \vee \neg(\neg q)) \wedge (\neg(\neg p) \vee (\neg q))\big) \wedge p && \text{De Morgan's} \\
&\equiv \big(((\neg p) \vee q) \wedge (p \vee (\neg q))\big) \wedge p && \text{Double Negation} \\
&\equiv ((\neg p) \vee q) \wedge \big((p \vee (\neg q)) \wedge p\big) && \text{Associativity} \\
&\equiv ((\neg p) \vee q) \wedge \big(p \wedge (p \vee (\neg q))\big) && \text{Commutativity} \\
&\equiv ((\neg p) \vee q) \wedge p && \text{Absorption} \\
&\equiv p \wedge ((\neg p) \vee q) && \text{Commutativity} \\
&\equiv (p \wedge (\neg p)) \vee (p \wedge q) && \text{Distributivity} \\
&\equiv c \vee (p \wedge q) && \text{Negation} \\
&\equiv p \wedge q && \text{Identity}
\end{aligned}
$$

Note in the preceding example that we used equivalences between the *xnor* to the *xor*, and the *xor* to the *or*. We will discuss later exactly how we deduced these equivalences.

Now that we understand some equivalences, we can motivate some definitions relating to the implication.

Definition 1.2.15 Converse
The converse of $p \Rightarrow q$ is $q \Rightarrow p$. Obtain this by reversing the arrow direction.

Definition 1.2.16 Inverse

The inverse of $p \implies q$ is $(\neg p) \implies (\neg q)$. Obtain this by negating both propositions.

Definition 1.2.17 Contrapositive
The contrapositive of $p \implies q$ is $(\neg q) \implies (\neg p)$. Obtain this by reversing the arrow direction, and negating both propositions. Or, take both the converse and inverse.

Definition 1.2.18 Negation
The negation of $p \implies q$ is $\neg(p \implies q)$. Obtain this by negating the entire implication.

Now we can use our equivalencies to show some important facts about these definitions.

Theorem 1.2.14 Contraposition Equivalence

$$p \implies q \equiv (\neg q) \implies (\neg p)$$

We leave the proof as an exercise. This theorem will serve us well in our study of Number Theory. The contrapositive indirect proof technique relies on this fact.

Theorem 1.2.15 The Converse Error

$$(p \implies q) \not\equiv (q \implies p)$$

Proof. We want to show that if you have $p \implies q$, then you do not necessarily have $q \implies p$. There are many ways to show this, but we will start by simply examining cases of p and q.
Consider $p \equiv \mathbf{F}$ and $q \equiv \mathbf{T}$. Then $p \implies q \equiv \mathbf{F} \implies \mathbf{T} \equiv \mathbf{T}$. But then $q \implies p \equiv \mathbf{T} \implies \mathbf{F} \equiv \mathbf{F}$. Then overall we have $(p \implies q) \implies (q \implies p) \equiv \mathbf{T} \implies \mathbf{F} \equiv \mathbf{F}$. So we've shown that if you have $p \implies q$, then you do not necessarily have $q \implies p$. □

Theorem 1.2.16 The Inverse Error

$$(p \implies q) \not\equiv ((\neg p) \implies (\neg q))$$

Remark 1.2.19
A similar proof can be made for this theorem. Find one truth-value pairings for p and q such that $p \implies q$ is true, but $(\neg p) \implies (\neg q)$ is false.

1.2.3 Circuits

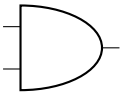
Propositions have two possible values: true and false. If we set true to mean *on* and false to mean *off*, then we can translate our Boolean statements into logical circuits. To do this, think of each logical connective as a *gate*. Similar to the logical connectives, a gate takes 2 (or more) inputs and returns some output. The inputs and outputs are all 1s and 0s (*on*s and *off*s – true and false). Circuits are the bare-bones to computers, so it is necessary you understand the basics. For computer engineers, you must know circuits by heart.

Definition 1.2.20 Circuit
Representations of Boolean statements into electronic components

Boolean variables can be exclusively either true or false; this is analogous to electric wires being exclusively either on or off. We let a tautology be equivalent to a *power source*, which is a wire that is always **on**. Similarly we let a contradiction be equivalent to **off**, or a wire receiving no power.

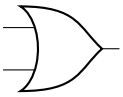
The following gates are exactly equivalent to their logic counterparts. We thus only include the gate picture.

Definition 1.2.21 And Gate – *conjunction*
Picture:

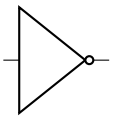


To remember the *and* gate, note that the picture looks like a **D**, which corresponds to the D in AN**D**

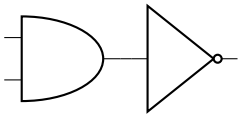
Definition 1.2.22 Or Gate – *Disjunction*
Picture:



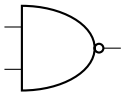
Definition 1.2.23 Not Gate – *Negation*
Picture:



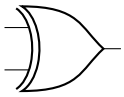
Remark 1.2.24
When we have a gate that has a *not* after it, we can simplify the gate by just adding a circle to the output. Example (the *nand* gate is the negation of the *and* gate):



becomes



Definition 1.2.25 Xor Gate (*eXclusive Or gate*)
Picture:



We can think of truth tables in an equivalent fashion, where 1 is true and 0 is false.

Example: Write the truth table for the *nand* gate.

Solution:

p	q	$\neg(p \wedge q)$
0	0	1
0	1	1
1	0	1
1	1	0

You can use 1/0 or T/F, whichever you prefer. If an assignment specifies you use a specific key, then use the one specified. Later when we discuss different number systems, you will notice a correlation between binary numbers and truth tables with 1/0s. This makes it easy to construct a truth table very quickly with a given amount of inputs.

Circuit Addition

We can also create circuits that add numbers. For a discussion of different number bases, see section **??**.

We can build a circuit that adds two single-digit binary numbers? There are only 4 possibilities for adding single-digit numbers, so let's examine what a truth table for this process might look like (all numbers are in base-2):

input a	input b	output $a + b$
0	0	0
0	1	1
1	0	1
1	1	10

In the output, we can always append zeros to the beginning of the binary number without changing the actual number:

input a	input b	output $a + b$
0	0	00
0	1	01
1	0	01
1	1	10

Now let's separate our output column into two columns – the sum-bit (right) and the carry-bit (left):

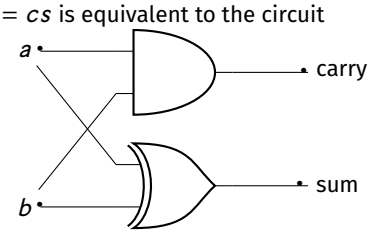
input a	input b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

We know how to generate Boolean expressions for the output columns:

$$\text{sum} \equiv a \oplus b$$
$$\text{carry} \equiv a \wedge b$$

Which gives us the **half adder**:

Adding two bits $a + b = cs$ is equivalent to the circuit



Now we can add two 1-bit numbers together. What if we want to add multiple-bit numbers together? Consider adding two 2-bit numbers:

$$\begin{array}{ccc} & c & \\ & x & y \\ + & z & w \\ \hline o_2 & o_1 & s \end{array}$$

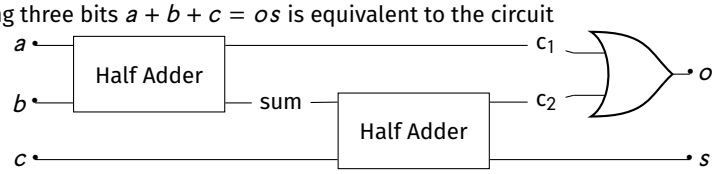
We see here that after adding $y + w$ we are left with a carry bit c for our addition $x + z$. How do we add three bits $c + x + z$? Well, we can separate it into two 1-bit additions: $c + x$ which, via a half-adder, yields a sum bit s_1 and carry bit c_1, then $s_1 + z$ which, via another half-adder, yields a sum bit s_2 and carry bit c_2. Then, we can let $o_1 = s_2$. Unfortunately this leaves us two carry bits that somehow need to be combined into a final carry bit.

Consider now the truth-table for adding three bits:

c	x	z	$c + x + z$	carry$(c + x)$	sum$(c + x) = s_1$	carry$(s_1 + z)$
0	0	0	00	0	0	0
0	0	1	01	0	0	0
0	1	0	01	0	1	0
0	1	1	10	0	1	1
1	0	0	01	0	1	0
1	0	1	10	0	1	1
1	1	0	10	1	0	0
1	1	1	11	1	0	0

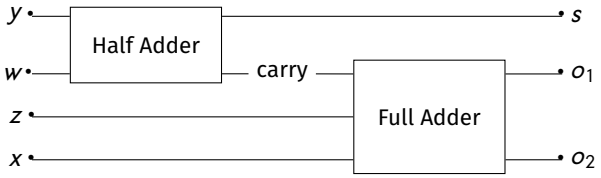
Now notice the carry bit in the output column $c + x + z$ is the same as OR-ing the two carry bits from $c + x$ and $s_1 + z$. This completes our circuit for adding three bits. We call this the **full adder**:

Adding three bits $a + b + c = os$ is equivalent to the circuit



Putting these two structures, the half adder and full adder, together, we can construct a 2-bit adder, which solves our previous problem of doing $xy + zw = o_2o_1s$:

Adding three bits $a + b + c = os$ is equivalent to the circuit



This solves our 2-bit addition problem. What if we want to add 3-bit numbers? 4-bit numbers? n-bit numbers? Well, after the first bit column (adding two bits) we must add three bits. After this second column, we get a sum bit and a carry bit. If we tack on another column, which would make a 3-bit adder, then we add in the previous carry to the two new bits. This same process repeats for all further columns. So, tacking on another bit solely entails tacking on another full adder! *Picture omitted*.

Example: How many half adders are required for an n-bit adder?

Solution: Note, here we implicitly assumed $n > 0$.
We need 1 half adder to start, and we need $n - 1$ full adders which each contain 2 half adders. This totals to $2(n - 1) + 1 = 2n - 1$ half adders.

1.2.4 Translations

You must know how to translate between circuits, Boolean statements, and truth tables.

We start by motivating the translation between truth tables and Boolean statements. First, notice that we have already discussed how to make a truth table from a Boolean statement – simply draw the truth table and fill in each row. Now we can focus on the reverse.

We attempt to first motivate a process for this translation by showing a few examples.

Example: First recall the truth table for the conjunction:

p	q	$p \wedge q$
F	F	F
F	T	F
T	F	F
T	T	T

Recall that we can extend the conjunction to take multiple inputs – in this case, *each* input *must* be **True** for the output to be **True**.
Now consider the following unknown table:

p	q	unknown
F	F	F
F	T	T
T	F	F
T	T	F

This table is similar to the conjunction table in that it has *one row* that outputs **True**. In the conjunction case, the assignments to p and q were both **True**. What are the assignments to the previous truth table? Well, the p input is **False** and the q input is **True**, as per the table. If we apply the conjunction to this row we have, we get the following table:

p	$\neg p$	q	$(\neg p) \wedge q$
F	T	F	F
F	T	T	T
T	F	F	F
T	F	T	F

In the previous table, if we look at the middle two columns, we recover the conjunction with the p variable negated.

This is only half the story though.

Example: First recall the truth table for the disjunction:

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

For this example, we only care about the middle two rows where *either* variable is **True**.

These bring light to a simple 3-step process to translate from truth tables to Boolean statements:

1. Collect the rows whose *output* is **True**
2. For each of those rows
    (a) Look at the truth value assigned to the *input*

(b) Construct a Boolean statement where, for each input

- If the assignment is **True** then use the variable itself
- If the assignment is **False** then use the *negation* of the variable

(c) Chain each input with **And** ($\wedge$) connectives

3. Chain each row's statement with **Or** ($\vee$) connectives

Example: Find the unknown Boolean formula corresponding to the following truth table:

p	q	unknown
F	F	T
F	T	F
T	F	T
T	T	F

Solution: We follow the algorithm as before. The first and third rows return true in the output. In the first row, we see neither input is true, so we AND the negation of each input: $(\neg p) \wedge (\neg q)$. In the third row, we see the first input is true, while the second input is false, thus we get: $p \wedge (\neg q)$. OR-ing each statement together thus yields our unknown formula:

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

Remark 1.2.26

In our Boolean formula algorithm, we only focus on the true rows. If we wanted to also include the false rows, then we would need the statement in each false row to return false. We know how to get a statement that returns true, so we can simply take that statement and negate it! What happens, though, if we include the false rows then?

Example: Include the false rows in the Boolean formula from the previous example.

Solution: We negate the second and fourth row returned by the algorithm: $\neg((\neg p) \wedge q)$ and $\neg(p \wedge q)$
So our entire statement is now

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

Let's simplify this statement (rules omitted):

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

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

Nice, we have recovered the original statement!

Remark 1.2.27
Including each false row in the Boolean statement generated by the algorithm – so long as their true versions are negated – does not logically change the Boolean statement

We have a name for the type of statement generated from our algorithm above:

Definition 1.2.28 Disjunctive Normal Form
Describes a Boolean statement that is a conjunction of disjunctions – abbreviated DNF

If we flip each gate (AND goes to OR, OR goes to AND), then we get another important type of statement:

Definition 1.2.29 Conjunctive Normal Form
Describes a Boolean statement that is a disjunction of conjunctions – abbreviated CNF

Interestingly, any Boolean statement can be translated to an equivalent statement in CNF. Namely, statements in DNF, which are easy to generate, can be translated into CNF. This is important for computational complexity theory – specifically NP-completeness. There exists a problem in computer science which entails finding a set of truth-value assignments for n different Boolean variables which makes a Boolean statement in CNF return true (or, become *satisfiable*).

1.2.5 Reasoning/Deductions

Propositional logic also allows us to *reason* about things.

Definition 1.2.30 Knowledge Base
A group of information that you know is true

Definition 1.2.31 Reasoning
The process of deriving new information from a given knowledge base

See the introduction of this chapter for an example.

Classical Rules of Deduction

We may refer to *deductions* as *inferences*. They are the same.

Definition 1.2.32 Deductions
Using previously-known knowledge in your knowledge base to obtain/create new knowledge

We have a whole list of useful deductions that are provably valid. As with our Boolean algebra theorems, you do not need to memorize these theorems – they will be given to you as a table.

Theorem 1.2.17 Modus Ponens

$$\begin{array}{l} p \\ \underline{p \Rightarrow q} \\ \therefore q \end{array}$$

Theorem 1.2.18 Modus Tollens

$$\begin{array}{l} \neg q \\ \underline{p \Rightarrow q} \\ \therefore \neg p \end{array}$$

Theorem 1.2.19 Disjunctive Addition

$$\begin{array}{l} \underline{p} \\ \therefore p \vee q \end{array}$$

Theorem 1.2.20 Conjunctive Addition

$$\begin{array}{l} \underline{p, q} \\ \therefore p \wedge q \end{array}$$

Theorem 1.2.21 Conjunctive Simplification

$$\begin{array}{l} \underline{p \wedge q} \\ \therefore p, q \end{array}$$

Theorem 1.2.22 Disjunctive Syllogism

$$\begin{array}{l} p \vee q \\ \underline{\neg p} \\ \therefore q \end{array}$$

Theorem 1.2.23 Hypothetical Syllogism

$$\begin{array}{l} p \Rightarrow q \\ \underline{q \Rightarrow r} \\ \therefore p \Rightarrow r \end{array}$$

Theorem 1.2.24 Resolution

$$\begin{array}{l} p \vee q \\ \underline{(\neg q) \vee r} \\ \therefore p \vee r \end{array}$$

Theorem 1.2.25 Division Into Cases

$$\begin{array}{l} p \vee q \\ p \Rightarrow r \\ \underline{q \Rightarrow r} \\ \therefore r \end{array}$$

Theorem 1.2.26 Law of Contradiction

$$\begin{array}{l} \underline{(\neg p) \Rightarrow \mathbf{c}} \\ \therefore p \end{array}$$

You may be wondering how to prove these deductions are *valid*. We have two equivalent methods:

1. Tautological implication
2. Critical-row identification

Consider an arbitrary deduction:

$$\begin{array}{l} P_1 \\ P_2 \\ \cdots\cdots\cdots \\ \underline{P_n} \\ \therefore Q \end{array}$$

To prove it valid,

Tautological implication

1. Construct a new proposition $A \Rightarrow B$ where A is a **conjunction of the premises** and B is the conclusion. For our arbitrary deduction, we would have

$$\left(P_1 \wedge P_2 \wedge \cdots \wedge P_n\right) \Rightarrow Q$$

2. Inspect this proposition in a truth-table. If the proposition is a tautology, then the deduction is valid. Otherwise, the deduction is invalid. So, to be valid we must have

$$(A \Rightarrow B) \equiv \big(P_1 \wedge P_2 \wedge \cdots \wedge P_n\big) \Rightarrow Q \equiv t$$

Critical-row identification

1. Construct a truth-table with columns for each proposition and for the conclusion. *You may need extra columns, that is fine.* For our arbitrary deduction, we would have

P_1	P_2	$\cdots$	P_n	Q
F/T	F/T	$\cdots$	F/T	F/T
		$\vdots$		
T/T	F/T	$\cdots$	F/T	F/T

2. Identify the rows in which each **premise** is true. We call these rows **critical-rows**.
3. For each critical-row, inspect the conclusion. If the conclusion is true in **every** critical-row, then the deduction is valid. Otherwise, the deduction is invalid.

Examples following the above steps.

Example: Show that Modus Ponens is a valid rule of inference.

Solution: *method 1 – tautological implication*
Our proposition we care about is $(p \wedge (p \Rightarrow q)) \Rightarrow q$, so we build the following truth-table with extraneous rows:

p	q	p	$p \Rightarrow q$	$p \wedge (p \Rightarrow q)$	q	$(p \wedge (p \Rightarrow q)) \Rightarrow q$
F	F	F	T	F	F	T
F	T	F	T	F	T	T
T	F	T	F	F	F	T
T	T	T	T	T	T	T

Inspect the last column.

p	q	p	$p \Rightarrow q$	$p \wedge (p \Rightarrow q)$	q	$(p \wedge (p \Rightarrow q)) \Rightarrow q$
F	F	F	T	F	F	T
F	T	F	T	F	T	T
T	F	T	F	F	F	T
T	T	T	T	T	T	T

In the case of Modus Ponens, the final column is a tautology, hence the deduction is valid.

Example: Show that Modus Ponens is a valid rule of inference.

Solution: *method 2 – critical-row identification*
Construct a truth table with each premise and conclusion:

p	q	p	$p \Rightarrow q$	q
F	F	F	T	F
F	T	F	T	T
T	F	T	F	F
T	T	T	T	T

Identify the critical-rows.

p	q	p	$p \Rightarrow q$	q
F	F	F	T	F
F	T	F	T	T
T	F	T	F	F
T	T	**T**	**T**	T

Inspect the conclusion in each critical-row.

p	q	p	$p \Rightarrow q$	q
F	F	F	T	F
F	T	F	T	T
T	F	T	F	F
T	T	**T**	**T**	T

In the case of Modus Ponens, the conclusion is true in each critical row, hence the deduction is valid.

We leave it to the reader to understand why the two methods are equivalent.

Deducing Things

In a later section, we will see that this order of logic is not powerful enough to prove mathematical statements. For now, we can still do interesting things with a given knowledge base.

Example: Given the following knowledge base, deduce as much new information as possible using the following rules of inference: Modus Ponens, Modus Tollens, Hypothetical Syllogism, and Disjunctive Syllogism.

$$a \Rightarrow b \qquad b \Rightarrow (\neg d) \qquad e \qquad d \vee (\neg e)$$

Solution:

By Hypothetical Syllogism $a \Rightarrow b$, $b \Rightarrow (\neg d)$, $\therefore a \Rightarrow (\neg d)$
By Disjunctive Syllogism $d \vee (\neg e)$, e, $\therefore d$
By Modus Tollens d, $a \Rightarrow (\neg d)$, $\therefore \neg a$
By Modus Tollens d, $b \Rightarrow (\neg d)$, $\therefore \neg b$

Remark 1.2.33
From the above example, we restricted the rules you could have used. We did this mainly because Disjunctive Addition can allow you to generate any new knowledge you like – so long as you have one thing that is true, then you can add in a disjunction infinitely-many times.

Remark 1.2.34
In the above example, we could have translated $d \vee (\neg e) \equiv e \implies d$ and concluded d by Modus Ponens. This somewhat tells you that Disjunctive Syllogism and Modus Ponens are equivalent.

Remark 1.2.35
From an inconsistent database, anything follows.

This is due to the law of contradiction. An *inconsistent database* is one that contains a contradiction. Recall from the law of contradiction that $(\neg p) \implies c \equiv (\neg(\neg p)) \vee c \equiv p$ using the Identity Boolean algebra theorem. Using Disjunctive Addition, we have the contradiction c, $\therefore A \vee c$, and by Identity, $\therefore A$. A can be *Anything*.

In Artificial Intelligence, there exists an algorithm called **The Resolution Algorithm**. Essentially, it says to take a given knowledge base, translate each statement into disjunctive normal form, then apply the Resolution rule of inference as many times as possible.

1.3 Predicate Logic

Sometimes basic propositions are not enough to do what you want. In programming we can have functions that return true or false. We can do the same thing with logic – we call this *first-order* logic, or *predicate* logic. Predicate logic includes all of propositional logic, however it adds predicates and quantifiers.

Definition 1.3.1 Predicate
A property that the subject of a statement can have. In logic, we represent this sort-of like a function. A predicate takes, as input, some element, and returns whether the inputted element has the specific property.

Example: We could use the predicate $EVEN(x)$ to mean x is an even number. In this case, the predicate is $EVEN(\cdot)$

Example: We could use the predicate $P(y)$ to mean y is an integer multiple of 3. In this case, the predicate is $P(\cdot)$

Remark 1.3.2
Predicates take **elements**. They do **not** take in other predicates. This is because predicates

say whether the input element *has* the property specified by the predicate – true and false cannot have properties.

In terms of programming, you can think of a predicate as a program method. For example, the $EVEN(x)$ predicate might be implemented as follows:

```
func EVEN(Entity x) -> bool {
        if IS_INTEGER(x) {
                return x % 2 == 0
        }
        return false
}
```

In this case, entities are *objects* and true/false are *Boolean primitives* (or, propositional statements, which can only be true or false). In contrast to, say, Java, a compiler for this code would not allow *true/false* to be an object.

A better example,

```
class Foo extends Entity {
        bool isInteger
        bool isOdd

        Foo(Integer i) {
                this.isInteger = true
                this.isOdd = i % 2 == 1
        }
}

func ODD(Entity x) -> bool {
        if x has type Foo {
                return x.isOdd
        }
        if IS_INTEGER(x) {
                return x % 2 == 1
        }
        return false
}
```

Definition 1.3.3 Quantifier
A way to select a specific range of elements that get inputted to a predicate. We have two quantifiers:

- The Universal quantifier $\forall$
- The Existential quantifier $\exists$

The universal quantifier says to select **all** elements, and the existential quantifier says to select **at least one** element.

Definition 1.3.4 Quantified Statement
A logical statement involving predicates and quantifiers. Syntax:

$$(\text{quantifier } var \in D)[\text{statement involving predicates}]$$

And now we can define:

Definition 1.3.5 Predicate Logic
Also called *first-order logic*, is a logic made up of quantified statements.

Example: Translate the following statements to predicate logic:

1. All people are mortal
2. Even integers exist
3. If an integer is prime then it is not even

Solution:

1. Denote P as the domain of people, and the predicate $M(x)$ to mean x is mortal. Then the statement translates to
$$(\forall p \in P)[M(p)]$$
2. Denote $\mathbb{Z}$ as the domain of integers, and the predicate $EVEN(x)$ to mean x is even. Then the statement translates to
$$(\exists x \in \mathbb{Z})[EVEN(x)]$$
3. Denote the predicate $PRIME(y)$ to mean y is prime. Then the statement translates to
$$(\forall a \in \mathbb{Z})[PRIME(a) \Rightarrow \neg EVEN(a)]$$

1.3.1 Negating Quantified Statements

One may find useful to negate a given quantified statement. We present here how to do this, first with an English example, followed by a quantified example, followed by an algorithm.

Example: The following statement

There is no student who has taken calculus.

is equivalent to

All students have not taken calculus.

Example: The following statement

Not all students have taken calculus.

is equivalent to

There is a student who has not taken calculus.

Example: The following statement

$$\neg(\exists x \in D)[C(x)]$$

is equivalent to

$$(\forall x \in D)[\neg C(x)]$$

Example: The following statement

$$\neg(\forall x \in D)[C(x)]$$

is equivalent to

$$(\exists x \in D)[\neg C(x)]$$

The generic algorithm for pushing the negation into a quantified statement:

1. Flip each quantifier $\forall \longrightarrow \exists$ and $\exists \longrightarrow \forall$
2. Apply the negation to the propositional part of the quantified statement, and simplify
   (a) If the inside contains another quantified statement, then recursively apply this algorithm

Remark 1.3.6
The domain and variable attached to any quantifier are **not** changed.

Example: Push the negation in as far as possible:

$$\neg(\forall x, y \in \mathbb{Z})[(x < y) \Rightarrow (\exists m \in \mathbb{Q})[x < m < y]]$$

Solution:

$$\begin{aligned}
&\neg(\forall x, y \in \mathbb{Z})[(x < y) \Rightarrow (\exists m \in \mathbb{Q})[x < m < y]] \\
\equiv&(\exists x, y \in \mathbb{Z})\neg[(x < y) \Rightarrow (\exists m \in \mathbb{Q})[x < m < y]] \\
\equiv&(\exists x, y \in \mathbb{Z})\neg[\neg(x < y) \vee (\exists m \in \mathbb{Q})[x < m < y]] \\
\equiv&(\exists x, y \in \mathbb{Z})[\neg\neg(x < y) \wedge \neg(\exists m \in \mathbb{Q})[x < m < y]] \\
\equiv&(\exists x, y \in \mathbb{Z})[(x < y) \wedge (\forall m \in \mathbb{Q})\neg[x < m < y]] \\
\equiv&(\exists x, y \in \mathbb{Z})[(x < y) \wedge (\forall m \in \mathbb{Q})\neg[x < m \wedge m < y]]
\end{aligned}$$

$$\equiv (\exists x, y \in \mathbb{Z})[(x < y) \wedge (\forall m \in \mathbb{Q})[x \geq m \vee m \geq y]]$$

Remark 1.3.7
We typically expect the final statement to contain no $\neg$ operators.

1.3.2 Quantified Rules of Inference

Again, *deductions* and *inferences* are the same. We present a handful of important *quantified* rules of inference.

Theorem 1.3.1 Universal Instantiation
For a predicate $P(\cdot)$ and some domain D with $c \in D$,

$$\frac{(\forall x \in D)[P(x)]}{\therefore P(c)}$$

As an example, if our domain consists of all dogs and Fido is a dog, then the above rule can be read as

"All dogs are cuddly"

"Therefore Fido is cuddly"

Theorem 1.3.2 Universal Generalization
For a predicate $P(\cdot)$ and some domain D for an arbitrary $c \in D$,

$$\frac{P(c)}{\therefore (\forall x \in D)[P(x)]}$$

This is most-often used in mathematics. As an example, if our domain consists of all dogs, then the above rule can be read as

"An arbitrary dog is cuddly" (which in-turn applies to all dogs)

"Therefore all dogs are cuddly"

Theorem 1.3.3 Existential Instantiation
For a predicate $P(\cdot)$ and some domain D for some element $c \in D$,

$$\frac{(\exists x \in D)[P(x)]}{\therefore P(c)}$$

As an example, if our domain consists of all dogs, then the above rule can be read as

"There is a dog who is cuddly"

"Let's call that dog c, and so c is cuddly"

Theorem 1.3.4 Existential Generalization
For a predicate $P(\cdot)$ and some domain D for some element $c \in D$,

$$\frac{P(c)}{\therefore (\exists x \in D)[P(x)]}$$

As an example, if our domain consists of all dogs and Fido is a dog, then the above rule can be read as

"Fido is cuddly"

"Therefore there is a dog who is cuddly"

Theorem 1.3.5 Universal Modus Ponens
For two predicates $P(\cdot)$ and $Q(\cdot)$, and some domain D with $a \in D$,

$$\begin{array}{l} P(a) \\ \underline{(\forall x \in D)[P(x) \Rightarrow Q(x)]} \\ \therefore Q(a) \end{array}$$

Theorem 1.3.6 Universal Modus Tollens
For two predicates $P(\cdot)$ and $Q(\cdot)$, and some domain D with $a \in D$,

$$\begin{array}{l} \neg Q(a) \\ \underline{(\forall x \in D)[P(x) \Rightarrow Q(x)]} \\ \therefore \neg P(a) \end{array}$$

1.3.3 Proving Things

Our familiar rules of inference are not strong enough to prove abstract mathematical statements. Typically we want our proof to apply to a whole *set* of things (numbers). Now that we know about *predicate logic*, we can apply our more powerful *quantified* rules of inference to prove real mathematical statements.

Example: Using Universal Modus Ponens, verify the validity of the following proof:

Proof. Let $m, n \in \mathbb{Z}$, and let m be even. Then $m = 2p$ for some integer p.[(1)] Now,

$$\begin{aligned} m \cdot n &= (2p)n && \text{by substitution} \\ &= 2(pn)^{(2)} && \text{by associativity} \end{aligned}$$

Now, $pn \in \mathbb{Z}$,[(3)] so by definition of even $2(pn)$ is even.[(4)] Thus mn is even. □

Solution:

(1) If an integer is even, then it equals twice some integer.
m is a particular integer, and it is even.
$\therefore$ m equals twice some integer p.

(2) If a quantity is an integer, then it is a real number.
p and n are both particular integers.
$\therefore$ p and n are both real numbers.

For all a, b, c, if $a, b, c \in \mathbb{R}$ then $(ab)c = a(bc)$.
2, p, and n are all particular real numbers.
$\therefore$ $(2p)n = 2(pn)$.

(3) For all u, v, if $u, v \in \mathbb{Z}$ then $uv \in \mathbb{Z}$.
p and n are both particular integers.
$\therefore$ $pn \in \mathbb{Z}$.

(4) If a number equals twice some integer, then that number is even.
$2(pn)$ equals twice the integer pn.
$\therefore$ $2(pn)$ is even.

Of course, we would never do a mathematical proof like this. In reality, you do this in your head automatically. Seeing this form, however, allows you to easily verify the **validity** of the proof.

1.4 Summary

- Propositional logic contains the entirety of Boolean algebra and logic connectives, with True and False as the only inputs/outputs
- Predicate logic contains the entirety of propositional logic and uses functions along with entities
- Deriving knowledge from familiar rules entails mathematical proof

1.5 Practice

1. Answer the two logic puzzles presented in the introduction of this chapter.
2. Translate the following statement into propositional logic: *turn right then turn left*.
3. Translate the following statement into propositional logic: *if it is raining then everyone has an umbrella*.
4. How can you quickly construct a truth table with all row possibilities? Use your technique to construct a truth table with 4 variables.
5. How many rows does a truth table with n variables have?
6. Prove theorem 1.2.12.

7. Prove theorem 1.2.13.

8. Draw the circuit representation of theorem 1.2.12.

9. Prove the following rule valid or invalid:

$$\begin{array}{l} (a \wedge d) \Rightarrow b \\ e \\ b \Rightarrow (\neg e) \\ (\neg a) \Rightarrow f \\ \underline{(\neg d) \Rightarrow f} \\ \therefore f \end{array}$$

10. Prove the following rule valid or invalid:

$$\begin{array}{l} (a \wedge d) \Rightarrow b \\ e \\ b \Rightarrow (\neg e) \\ (\neg a) \Rightarrow f \\ \underline{(\neg d) \Rightarrow f} \\ \therefore b \Rightarrow e \end{array}$$

11. Push the negation inside the following statement as far as possible:

$$\neg(\forall x \in \mathbb{R})(\exists m \in \mathbb{Z})[(0 \leq x - m < 1) \Leftrightarrow (m = \lfloor x \rfloor)]$$