Logic and Computation

December 3, 2017

Contents

Chapter 1

Propositional Logic

When you come to any passage you don't understand, read it again: if you still don't understand it, read it again: if you fail, even after three readings, very likely your brain is getting a little tired. In that case, put the book away, and take to other occupations, and next day, when you come to it fresh, you will very likely find that it is quite easy.

—Lewis Carroll, *Symbolic Logic,* 1896.

1.1 You already know what propositional logic is, even if you didn't know what it was called

Much of the power of computer programs stems from their ability to perform conditional execution—that is, to change the order in which statements are executed depending on the current values of program variables. For example, Figure 1.1 shows a Python function that returns `True` or `False` depending on whether or not its argument is a leap year. This is not so simple a matter as checking if the year is divisible by 4; the correct criterion is more complicated: Century years (like 1700 and 1900) are not leap years, unless they are divisible by 400 (like 1600 and 2000).

The code in Figure 1.2 performs the same calculation much more compactly, by returning the value of an expression of *boolean* type (called `bool` in Python). Python, like other programming languages, contains a kind of mini-language for creating complex boolean expressions by combining simpler expressions with the operators `and`, `or` and `not`. The expression building begins with 'atomic' boolean expressions like `year%400==0`, so-called because they cannot be broken down into smaller boolean expressions.

The same mini-language, often with different syntax rules, is present in some form in every programming language, in database query languages, and even within the formula language for Excel spreadsheets (Figure 1.3).

This little language is called *propositional logic.*

```
def leapyear(y):
    if y%4==0:
        if y%100==0:
            if y%400==0:
                return True
            else:
                return False
        else:
                return True
    else:
        return False;
```

Figure 1.1: *A logical calculation in Python, determining if year* `y` *is a leap year*

```
def leapyear(y):
    return (y%4==0) and (not (y%100==0) or (y%400==0))
```

Figure 1.2: *The same calculation, using a complex boolean expression.*

1.2 Propositional Formulas-Syntax

We will shortly give a precise definition of propositional logic, but here, in a nutshell, is the idea: Ordinary algebraic expressions, like

$$4x^2y + 3x - 2$$

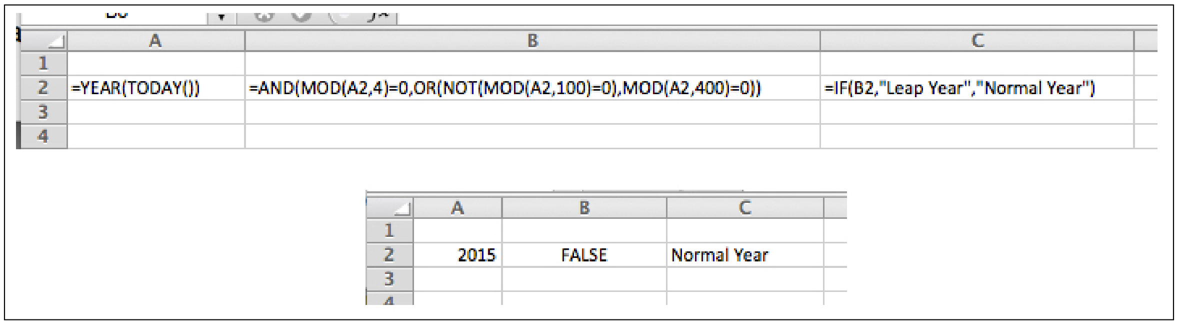
are built from variables (in this case x and y), constants (2, 3 and 4), and operation symbols (for addition, subtraction and multiplication). When you substitute numbers for the variables (say 1 for x and 2 for y) you get a value, which is itself a number (in this example, 3). The formulas of propositional logic are constructed and evaluated in much the same way, however the values are *logical* (**true** and **false**) rather than numerical, and the operation symbols are different. The ability to represent and manipulate logical processes using algebra is the essence of propositional logic.

As you read the definitions, keep these examples in mind, especially the one from Python: We use different symbols ($\wedge$, $\vee$ and $\neg$ in place of `and`, `or` and `not`), but the structure and meaning of formulas of propositional logic are pretty much identical to what you find in Python and other computer languages.

1.2.1 Definition

We first define the *syntax* of propositional logic, describing how formulas are built. The next section is devoted to *semantics*–what formulas *mean.*

A *propositional formula* is a string of symbols having one of the following forms:



	A	B	C
1			
2	=YEAR(TODAY())	=AND(MOD(A2,4)=0,OR(NOT(MOD(A2,100)=0),MOD(A2,400)=0))	=IF(B2,"Leap Year","Normal Year")
3			
4			

	A	B	C
1			
2	2015	FALSE	Normal Year
3			

Figure 1.3: *The same calculation in an Excel spreadsheet. The top view displays the formulas, the bottom is the standard view*

- **T** or **F**
- $p, q, r, p_1, p_2, q_1,$ *etc.* (These symbols are called *variables*.)
- $(\phi \wedge \psi)$ where ϕ and ψ are propositional formulas.
- $(\phi \vee \psi)$ where ϕ and ψ are propositional formulas.
- $\neg\phi$ where ϕ is a propositional formula.

1.2.2 Examples

- Here is a propositional formula
$$\mathbf{T}$$
- And here is another:
$$(\mathbf{T} \vee (\neg p \vee \neg(q \wedge p))).$$
To see why, observe that the first two rules tell us that $\mathbf{T}, p,$ and q are formulas, the third rule then says that $(q \wedge p)$ is a formula, the fifth rule that $\neg(q \wedge p)$ and $\neg p$ are formulas, and so on.
- According to this definition, neither of the strings
$$\neg p \vee (\neg q \wedge p), ((\neg p) \vee (\neg q \wedge p))$$
is a formula, the first because it lacks a pair of parentheses bracketing the entire formula, and the second because it places a pair of parentheses around $\neg p$. But we will shortly relax our strict requirements and allow them both as acceptable alternatives for $(\neg p \vee (\neg q \wedge p))$. (In this connection, see 1.2.5 as well as the exercises at the end of the chapter, especially Exercises 3 and 7.)

1.2.3 Recursive Nature of the Definition.

The definition at first seems to be circular, since it uses 'propositional formula' in the definition itself. But this circularity is only apparent. The definition is actually *recursive,* much like the functions using recursion that you might have studied in an introductory programming course: It defines propositional formulas in terms of *smaller* propositional formulas, and the recursion bottoms out at the formulas that are not combinations of smaller formulas, *i.e.,* **T**, **F**, p, q, *etc.* These are the *atomic formulas.* The atomic formulas **T** and **F** are *constants,* and the others are *variables.* We will say more about recursive definitions in Chapter 4.

1.2.4 The Definition as a Grammar

Syntax rules like the ones we gave above are often written in a more schematic form:

$$\begin{aligned}
\text{formula} &\longrightarrow \mathbf{T}|\mathbf{F}\\
\text{formula} &\longrightarrow p|q|r\cdots\\
\text{formula} &\longrightarrow (\text{formula}\wedge\text{formula})\\
\text{formula} &\longrightarrow (\text{formula}\vee\text{formula})\\
\text{formula} &\longrightarrow \neg\text{formula}
\end{aligned}$$

A set of rules written in this way is called a *context-free grammar.* The grammar provides a compact scheme for showing the *derivation* of a formula—how it is built from the atoms. Here is a derivation of the formula $(r\vee(\neg p\vee\neg(q\wedge p)))$

$$\begin{aligned}
\text{formula} &\longrightarrow (\text{formula}\vee\text{formula})\\
&\longrightarrow (r\vee\text{formula})\\
&\longrightarrow (r\vee(\text{formula}\vee\text{formula}))\\
&\longrightarrow (r\vee(\neg\text{formula}\vee\text{formula}))\\
&\longrightarrow (r\vee(\neg p\vee\neg\text{formula}))\\
&\longrightarrow (r\vee(\neg p\vee\neg(\text{formula}\wedge\text{formula})))\\
&\longrightarrow (r\vee(\neg p\vee\neg(q\wedge\text{formula})))\\
&\longrightarrow (r\vee(\neg p\vee\neg(q\wedge p)))
\end{aligned}$$

See Exercise 2.

1.2.5 Avoiding parentheses

If you were to formulate similar rules for building algebraic expressions with operations for addition and multiplication, you might write something like, 'if E_1 and E_2 are expressions, then $E_1 + E_2$ and $E_1 \times E_2$ are expressions'. The problem with this is that if you apply the rule twice in succession, you find that there are two different ways to generate expressions like

$$x + y \times z.$$

Does this mean add x and y and multiply by z? Or does it mean add x to the product of y and z? To avoid such ambiguity in our definition of propositional formulas, we require adding a pair of parentheses every time we apply one of the operators $\wedge$ and $\vee$. The result, though, is that formulas get cluttered up with lots of parentheses. We can clean things up a bit by observing that we never need to write the outermost pair of parentheses in a formula, so we can write our propositional formula above as

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

Later we will see that the operator $\vee$, as well as $\wedge$, obeys an associative law, so that there is never any need to distinguish between $\phi \vee (\psi \vee \rho)$ and $(\phi \vee \psi) \vee \rho$. Thus we can write the formula above as

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

What about 'formulas' like

$$p \wedge q \vee r\ ?$$

There is not universal agreement about how to interpret this. Some authors define a precedence rule, much like the rule that multiplication takes precedence over addition in algebraic formulas, and give $\wedge$ a higher precedence than $\vee$. We will take a more conservative approach and require parentheses. So we will allow both $(p \wedge q) \vee r$ and $p \wedge (q \vee r)$, but treat the unparenthesized formula as illegal. (See the Exercises beginning at Exercise 5.)

1.3 Propositional Formulas-Semantics

1.3.1 The value of a propositional formula.

In a boolean expression in Python, an atom, like `year%4 == 0` in our example, is a *proposition*: it asserts something that is either true or false, and the truth or falsehood varies, depending on the value of the integer variable `year`. The entire expression is also a proposition, whose truth depends in turn on the truth or falsehood of the atomic boolean formulas it contains. This is the situation in general in propositional formulas: The variables p, q, *etc.,* in the formula denote propositions that can be assigned the *values* **true** and **false** arbitrarily. Given such an assignment, the value of the entire formula is determined by the following recursive rules:

- **T** has the value **true**, and **F** has the value **false**.
- $\neg\phi$ has the value **true** if ϕ has the value **false,** and vice-versa.

- $(\phi \wedge \psi)$ has the value **true** if both ϕ and ψ do; otherwise it has the value **false**.
- $(\phi \vee \psi)$ has the value **false** if both ϕ and ψ do; otherwise it has the value **true**.

These are, of course, the same rules as for `and`, `or` and `not` in Python.

For example, consider again the formula

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

(We'll now allow ourselves the luxury of leaving off the outermost pair of parentheses.)

What is the value of this formula if the variables p, q, r are assigned, respectively, the values **false**, **false**, **true**? Since p is **false**, $\neg p$ is **true**, Consequently both $\neg p \vee \neg(q \wedge p)$ and the complete formula $r \vee (\neg p \vee \neg(q \wedge p))$ have the value **true**.

What if the assignment is $p \mapsto$ **true**, $q \mapsto$ **true**, and $r \mapsto$ **false**? Then $q \wedge p$ is **true**, $\neg(q \wedge p)$ is **false**, $\neg p$ is **false**, and thus $\neg p \vee \neg(q \wedge p)$ is **false,** and consequently $r \vee (\neg p \vee \neg(q \wedge p))$ gets the value **false**.

Can you find another assignment of values to p, q and r that makes the formula **false**? (See Exercise 9.)

We will take another look at the semantics of propositional formulas in Chapter 3, when we discuss functions.

1.3.2 Example: Modeling ordinary language by propositional formulas.

Your eating habits at lunch are subject to several constraints of personal taste and the rules of the cafeteria:

- You can order either soup or salad.
- You can't order both soup and salad.
- While you are fond of soup, you only eat it on cold days, so it has to be less than 32 degrees outside for you to order it.

We will model these rules with a formula of propositional logic, much as we described the rule for determining if a year is a leap year with a boolean expression in Python. We proceed by identifying the underlying atomic propositions in the rule, and associate these with the variables p, q, r:

- p: You order the soup.
- q: You order the salad.
- r: It is less than 32 degrees outside.

The three constraints can then be encoded by the propositional formulas:

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

In fact, there are many different ways to write these constraints as propositional formulas. Look especially at our encoding of the last constraint, which we are in essence rephrasing as 'Either it's freezing or you order the salad.' We could also have phrased it as, 'You can't eat soup on a warm day', which we would write as $\neg(p \wedge \neg r)$.

We can encode all three constraints by the single propositional formula

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

When this formula has the value **true**, the values assigned to the variables are consistent with the constraints above; when it's **false**, the constraints are not all satisfied.

1.3.3 Truth tables

A *truth table* shows how the values of a complex formula are computed from the values assigned to the variables. Below are truth tables for the formulas $\neg p$, $p \vee q$ and $p \wedge q$.

p	$\neg p$
T	F
F	T

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

We can use such tables to compute the truth value of any formula, given assignments of truth values to the variables. Below is the truth table for our soup and salad rule. We begin in the left-hand columns by tabulating all the possible assignments of truth values to the variables. In subsequent columns we compute the values of the various subformulas entering into the formula. The last column displays the values of the formula itself.

The rows in which a **T** appears in the last column correspond to the three assignments of values to the variables p, q, r that make the whole formula true: It's freezing and you order the soup but not the salad; it's freezing and you order the salad but not the soup; it's not freezing and you order the salad but not the soup.

p	q	r	$\phi_1 = p \vee q$	$\phi_2 = \neg(p \wedge q)$	$\phi_3 = r \vee q$	$\phi_1 \wedge \phi_2 \wedge \phi_3$
T	T	T	T	F	T	F
T	T	F	T	F	T	F
T	F	T	T	T	T	T
T	F	F	T	T	F	F
F	T	T	T	T	T	T
F	T	F	T	T	T	T
F	F	T	F	T	T	F
F	F	F	F	T	F	F

Table 1.1: Truth table for the soup-or-salad rule

1.3.4 Satisfiability and tautology

A formula ϕ is *satisfiable* (also called *consistent*) if there is at least one assignment of truth values to the variables that makes the formula true. Another way to say this is that the last column of the truth table for ϕ contains at least one occurrence of **T**. For example, our soup-or-salad formula is satisfiable. In fact, there are several satisfying assignments, two in which r is assigned the value **true** and one in which r is assigned **false**—a good thing, since it means that you can always find something to eat, whatever the weather.

If *all* the entries in the final column are **T**, the formula is called a *tautology* (also *valid*). Such a formula has the value **true** regardless of the values of the variables. Another way to say this is that ϕ is a tautology if and only if $\neg\phi$ is not satisfiable. Every tautology is satisfiable, but not every satisfiable formula is a tautology, as our soup-and-salad example shows. A simple example of a tautology is the formula $p \vee \neg p$. An even simpler example is **T**.

A formula is a *contradiction* (also *inconsistent*) if it is not satisfiable, that is, all the entries in the last column of the truth table are **F**. Another way to say this is that ϕ is a contradiction if and only if $\neg\phi$ is a tautology. A simple example is $p \wedge \neg p$. An even simpler example is **F**.

1.4 New connectives: conditional, biconditional, exclusive or

Our logical operators $\vee$, $\wedge$, $\neg$ are often called *connectives*. Here we will describe three new connectives. In a sense, these connectives are not new at all: As we will explain at length, anything that we can say with these new symbols can also be expressed using the three original ones. What we gain by adding them is a more succinct, and in some respects, more natural way to write logical propositions.

1.4.1 Conditional

We could rephrase the last constraint in the soup-and-salad example by 'if you order the soup, then it has to be freezing outside'. (But *not* by, 'if it's freezing outside then you order the soup', which is false, since you might order salad on a cold day.) We introduce a new connective $\rightarrow$, and write $\phi \rightarrow \psi$, to mean 'if ϕ then ψ'. So, using the variable symbols from that example, we can write the last constraint in our example as $p \rightarrow r$. The official definition of the semantics of this connective is given by the truth table Table 1.2.

ϕ	ψ	$\phi \rightarrow \psi$
T	T	T
T	F	F
F	T	T
F	F	T

Table 1.2: *Truth table defining the connective* $\rightarrow$

It seems strange to say that 'if ϕ then ψ' is true when both ϕ and ψ are false! To make some sense of this, imagine that you go to the cafeteria every day: Some days it is freezing and you order the soup, some days it is freezing and you order the salad, and some days it is warmer and you order the salad (so that p and r are both false) . But it is *never* the case that you order the soup on a warmer day. The *only* way $p \rightarrow r$ can be false is for p to be true and r false. That is *all* that $\rightarrow$ means. [1]

1.4.2 Biconditional

We write $\phi \leftrightarrow \psi$ to mean 'if ϕ then ψ and if ψ then ϕ', so you can think of $\phi \leftrightarrow \psi$ as an abbreviation for $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$. Another way to say this is 'ϕ if and only if ψ'. For example: A different cafeteria customer in our soup-and-salad story, one with more rigid habits, might always order soup on freezing days and the salad on all other days. We can represent this constraint by $p \leftrightarrow r$.

Table 1.3 is the truth table for the biconditional.

[1] Equivalent formulations of $\phi \rightarrow \psi$ in English are 'ϕ implies ψ', 'ψ is a necessary condition for ϕ', 'ϕ is a sufficient condition for ψ', and others. Implicit in some of these formulations is a notion of dependent and independent conditions. Saying 'cold weather is a necessary condition for ordering soup' is entirely reasonable, but 'ordering soup is a sufficient condition for cold weather' has a peculiar ring to it, as it suggests that your lunch choice influenced the weather. It is important to keep in mind that the formal definition of the connective $\rightarrow$ involves no such notion of dependence or causality.

ϕ	ψ	$\phi \leftrightarrow \psi$
T	T	T
T	F	F
F	T	F
F	F	T

Table 1.3: *Truth table defining the connective* $\leftrightarrow$.

1.4.3 Exclusive-or

The symbol $\vee$ does not mean 'or' in the exclusive sense it often has in English. When a restaurant waiter tells you, 'you can get the soup or the salad with your dinner', he certainly does not mean that you can order *both*, but the logical 'or' would allow this. We define a new connective $\phi \oplus \psi$ to mean just what the waiter intends here: 'ϕ or ψ but not both'. For instance, our cafeteria's soup-or-salad rule is precisely $p \oplus q$. The exact definition is given by the truth table Table 1.4.

ϕ	ψ	$\phi \oplus \psi$
T	T	F
T	F	T
F	T	T
F	F	F

Table 1.4: *Truth table defining the connective* $\oplus$.

With these new symbols we can express the soup-and-salad example much more succinctly:

$$(p \oplus q) \wedge (p \rightarrow r).$$

1.5 Equivalent formulas

As we observed above, the only way for $p \rightarrow q$ to have the value **false** is for p to be **true** and q **false**. This means that the formulas $p \rightarrow q$ and $\neg(p \wedge \neg q)$ have the same truth value regardless of what values are assigned to the variables. We say that the two formulas are *equivalent*. Similarly, you may have noticed that for each assignment, $p \oplus q$ has the opposite value from $p \leftrightarrow q$, so that $p \oplus q$ is equivalent to $\neg(p \leftrightarrow q)$. Equivalent formulas say exactly the same thing, but with different notation.

In general, if ϕ and ψ are formulas, we say that ϕ and ψ are *equivalent,* and write $\phi \equiv \psi$, if for every assignment of truth values to the variables occurring in ϕ and ψ, the resulting

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

Table 1.5: *Truth table establishing the identity* $\neg(p \vee q) \equiv \neg p \wedge \neg q$.

values of ϕ and ψ are the same. Another way to say this is, $\phi \leftrightarrow \psi$ is a tautology. It's worth pointing out a little subtlety here: The symbol $\equiv$ is *not* another connective of propositional logic and never occurs inside formulas; it is a symbol we use when we talk *about* propositional formulas.

Here is another example. Table 1.5 displays the truth tables for the formulas $\neg(p \vee q)$ and $\neg p \wedge \neg q$. (We have combined the tables for these two formulas into a single array, giving the value of the first formula in the next-to-last column, and of the second formula in the last column.)

Since the last two columns are the same, we conclude that the two formulas are equivalent:

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

We really didn't need to go to all the trouble of constructing truth tables to show the equivalence. In cases like this it is just as easy to talk our way through the problem: the only way $\neg(p \vee q)$ can be true is for both p and q to be false, that is, for both $\neg p$ and $\neg q$ to be true.

1.5.1 Identities

Equivalences between propositional formulas work much like equations between algebraic expressions that hold regardless of the values of the variables they contain. An example of this is the distributive law for multiplication and addition of numbers.

$$x(y + z) = xy + xz.$$

Such equations are called *identities*. (In contrast, something like $xy = z$ is an equation, but not an identity, as there are values of x, y and z for which the two sides are unequal.) As with any equation, you can do the same thing to both sides of the identity—multiply them by the same value, add the same value to them, *etc.*—and the result will still be an identity. Moreover, if you replace each of the variables occurring in the identity by an arbitrary expression, the result is still an identity. For instance, if we replaced x by $u + 2v$, y by w^2 and z by $5uv$, we would get the equation

$$(u + 2v)(w^2 + 5uv) = (u + 2v)w^2 + (u + 2v) \cdot 5uv,$$

which is still an identity, since it holds for all values of the variables u, v, w.

Equivalences between propositional formulas are also called identities, and work much the same way. Starting from the identity

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

we can apply the same operation to both sides of the identity to obtain a new identity:

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

And we can substitute any formulas for the variables in an identity and obtain further identities:

$$(s \wedge t) \wedge \neg\neg((s \vee t) \vee (s \wedge t)) \equiv (s \wedge t) \wedge \neg(\neg(s \vee t) \wedge \neg(s \wedge t)).$$

There are a number of standard propositional identities. These are analogous to (and in some instances closely resemble) algebraic identities like the distributive and associative laws. We have tabulated about half of the most basic ones in Table 1.6 below. Most of these are nearly obvious at a glance. Some, like the distributive law in the last row, require a little thought. But all can easily be proved, either by construction of truth tables, or by talking your way through, just as we proved the identity $\neg(p \vee q) \equiv \neg p \wedge \neg q$, which appears in the second-to-last line of the table.

$p \wedge \mathbf{T}$	$\equiv$	p
$p \vee \mathbf{T}$	$\equiv$	$\mathbf{T}$
$p \vee \neg p$	$\equiv$	$\mathbf{T}$
$p \vee p$	$\equiv$	p
$p \vee (p \wedge q)$	$\equiv$	p
$\neg\neg p$	$\equiv$	p
$p \vee q$	$\equiv$	$q \vee p$
$p \vee (q \vee r)$	$\equiv$	$(p \vee q) \vee r$
$\neg(p \vee q)$	$\equiv$	$\neg p \wedge \neg q$
$p \vee (q \wedge r)$	$\equiv$	$(p \vee q) \wedge (p \vee r)$

Table 1.6: Some basic propositional identities

1.5.2 DeMorgan's Laws and Duality

The identity

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

is called *DeMorgan's Law.* Let's play with it a bit: We can perform the same operations on both sides of an identity and the result will still be an identity, so let's apply $\neg$ to both sides. This gives us

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

We can also substitute arbitrary formulas for the variables in an identity and still have an identity, so let's substitute $\neg p$ for p and $\neg q$ for q. Now we have

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

The identity in the sixth line of the table allows us to erase any occurrence of $\neg\neg$ (the two successive negations cancel one another), so we wind up with

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

This is *also* called DeMorgan's Law, and it is identical to the original version of DeMorgan's Law, except that we have changed the $\vee$ to $\wedge$ and the $\wedge$ to $\vee$.

Now we can use DeMorgan's Laws to show that this trick works for *every* identity. Let's take, for example, the distributive identity at the end of the table:

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

We negate both sides,

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

apply DeMorgan's Law to each side,

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

and apply DeMorgan's Law *again* on both sides,

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

substitute $\neg p$ for p and $\neg q$ for q,

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

and clean up the double negations:

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

This is *another* distributive law, this time with $\wedge$ distributing over $\vee$. If ϕ is a formula built from variables with the connectives $\vee$, $\wedge$, $\neg$ then its *dual* ϕ' is obtained by replacing every occurrence $\wedge$ by $\vee$ and every occurrence of $\vee$ by $\wedge$. The general principle is that if $\phi \equiv \psi$ is an identity, then so is $\phi' \equiv \psi'$.

If we rush ahead and try to apply this to the first identity in the table

$$p \wedge \mathbf{T} \equiv p,$$

we would get $p \vee \mathbf{T} \equiv p$. This is not an identity, as you can tell by assigning p the value **false**. The problem is that we overreached in forming the dual of a formula that contains constants like $\mathbf{T}$ as well as variables. But the solution is simple: to form the dual of such a formula we change $\vee$ to $\wedge$, $\wedge$ to $\vee$, $\mathbf{T}$ to $\mathbf{F}$ and $\mathbf{F}$ to $\mathbf{T}$. Then everything works, and the new equation we get is $p \vee \mathbf{F} \equiv p$, which is indeed an identity.

Here is an 'official' statement of the general principle that we have just established:

Theorem 1.5.1. *(Duality Principle) Let ϕ and ψ be propositional formulas consisting of variables, constants, and the connectives $\wedge$, $\vee$, $\neg$. Let ϕ' be the formula obtained from ϕ by changing all occurrences of $\vee$ to $\wedge$, and vice-versa, and all occurrences of* **T** *to* **F**, *and vice-versa. Let ψ' be obtained from ψ in the same way. If $\phi \equiv \psi$, then $\phi' \equiv \psi'$.*

Thanks to Theorem 1.5.1, we can produce a new table of identities from Table 1.6:

$p \vee \mathbf{F}$	$\equiv$	p
$p \wedge \mathbf{F}$	$\equiv$	$\mathbf{F}$
$p \wedge \neg p$	$\equiv$	$\mathbf{F}$
$p \wedge p$	$\equiv$	p
$p \wedge (p \vee q)$	$\equiv$	p
$p \wedge q$	$\equiv$	$q \wedge p$
$p \wedge (q \wedge r)$	$\equiv$	$(p \wedge q) \wedge r$
$\neg(p \wedge q)$	$\equiv$	$\neg p \vee \neg q$
$p \wedge (q \vee r)$	$\equiv$	$(p \wedge q) \vee (p \wedge r)$

Table 1.7: Duals of the identities in Table 1.6

You might notice that we have left off the dual of one identity that appeared in the original table: this is because $\neg\neg p \equiv p$ is its own dual, so we chose not to write it twice.

The identities have a few obvious extensions, which should be familiar from the way we deal with standard algebraic expressions. Because of the associative law

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

we can just write these formulas as $p \vee q \vee r$; it doesn't matter how we insert parentheses into this expression, because the result will be the same either way.

By applying the associative law several times in succession, we get

$$\begin{aligned}(p_1 \vee (p_2 \vee p_3)) \vee p_4 &\equiv ((p_1 \vee p_2) \vee p_3) \vee p_4 \\ &\equiv (p_1 \vee p_2) \vee (p_3 \vee p_4) \\ &\equiv p_1 \vee (p_2 \vee (p_3 \vee p_4)) \\ &\equiv p_1 \vee ((p_2 \vee p_3) \vee p_4)\end{aligned}$$

so, again, no matter how we (legally) introduce parentheses into

$$p_1 \vee p_2 \vee p_3 \vee p_4,$$

the result is the same, in the sense that any two formulas we produce this way are equivalent. As you might expect, the same holds for any number of terms[2], so we can write

$$p_1 \vee p_2 \vee \cdots \vee p_k$$

[2]This might not be completely obvious (see the discussion below about 'hand-waving'). Later in the text, you will be asked to provide a careful proof of this fact.

without ambiguity.

We can similarly extend the distributive identity to any number of terms:

$$q \wedge (p_1 \vee \cdots \vee p_k) \equiv (q \wedge p_1) \vee \cdots \vee (q \wedge p_k).$$

Of course the same applies to the duals of all these formulas.

There is a more compact notation for the disjunction and conjunction of many terms. We write

$$p_1 \vee \cdots \vee p_k$$

as

$$\bigvee_{i=1}^{k} p_i$$

and

$$p_1 \wedge \cdots \wedge p_k$$

as

$$\bigwedge_{i=1}^{k} p_i.$$

With this notation the extended distributive law becomes

$$q \wedge \bigvee_{i=1}^{k} p_i \equiv \bigvee_{i=1}^{k} (q \wedge p_i).$$

1.5.3 Did we just prove something?

In the preceding paragraphs we made a number of claims about equivalent formulas and gave some arguments to support those claims. We even went so far as to call one of them a *theorem.* In mathematics, theorems have *proofs.* Did we actually prove anything?

We really did prove the first version of DeMorgan's Law, by carefully tabulating the values of the two sides of the identity on all four assignments of truth values to the variables. And we did show, step by step, how the dual version of the law could be derived from its original formulation. In each of these cases, we were dealing with only a single formula. But to justify the general duality principle, we gave just the example of just a single identity, and claimed that the same procedure would work in the same way for every identity. As if to underscore the problems with this approach, we immediately followed this with an identity for which the method *didn't* work, and so we patched up the definition of dual formula, said 'ok, *now* it works', and called the result a theorem.

This is a classic instance of the 'hand-waving' argument, in which the prover shoves aside a lot of difficulties that he is either too lazy or too uncertain to address, often punctuating his argument with 'obviously's and 'it is clear that's. Later in this text, we will take up the question of proofs, and give the need for careful proof the attention that it deserves. We will even give, as an example, a real proof of Theorem 1.5.1. But that will have to wait. In the next few sections we will continue with the cavalier approach, and establish a few more general principles of propositional logic, with a wave of the hand.

1.5.4 Normal Forms

Disjunctive Normal Form. Look again at the soup-or-salad example, and in particular at its truth table Table 1.1. When we first displayed this table, we noted that exactly three rows of the table gave the result **T** in the last column. The three rows correspond to the three assignments

$$\begin{array}{lll} p \mapsto \textbf{true} & q \mapsto \textbf{false} & r \mapsto \textbf{true} \\ p \mapsto \textbf{false} & q \mapsto \textbf{true} & r \mapsto \textbf{true} \\ p \mapsto \textbf{false} & q \mapsto \textbf{true} & r \mapsto \textbf{false} \end{array}$$

We can express the property that one of these assignments is made by the single formula:

$$(p \wedge \neg q \wedge r) \vee (\neg p \wedge q \wedge r) \vee (\neg p \wedge q \wedge \neg r). \tag{1.5.1}$$

This formula has the value **true** for the three assignments above, and the value **false** otherwise. In other words, this formula is equivalent to the one we started from to make the truth table.

Formulas like (1.5.1) are said to be in *disjunctive normal form,* or DNF. In general, a formula is said to have this form if it is a disjunction

$$\phi_1 \vee \cdots \vee \phi_k,$$

where each subformula ϕ_i is a conjunction

$$L_{i1} \wedge L_{i2} \wedge \cdots \wedge L_{is_i},$$

and each L_{ij} is a *literal*: either a variable itself or the negation of a variable.[3] The procedure we described above works in general: Starting from the truth table for a formula, you can extract an equivalent formula in DNF by looking at the rows in which the formula has the value **true**. Here is a formal statement of the result:

Theorem 1.5.2. *Every propositional formula is equivalent to a formula in disjunctive normal form.*

Universality of propositional formulas. There is another conclusion we can draw from the reasoning above. We didn't need to start from a formula; we could have begun with a table that gives all possible assignments to the variables, and *any* sequence of truth values in the last column. Following the procedure outlined above, we could have still produced a DNF formula that has the given table as its truth table. So propositional formulas are a kind of universal language: Any rule that we can describe with sufficient precision to be represented as a truth table can be described by a propositional formula. We will return to this very important point in the next chapter.

[3]The indices k and s_i in the above formulas can be equal to 1. That is, a formula in DNF could be just a single conjunction of literals, and a conjunction of literals can be just a single literal. Thus $p \vee \neg q$, $p \wedge q \wedge r$, and even p, are all in DNF.

Conjunctive Normal Form. Theorem 1.5.2 has a dual version. Let us begin with a formula ϕ. It has a dual ϕ', and by Theorem 1.5.2, $\phi' \equiv \delta$, where δ is a DNF formula. Since $\phi = (\phi')'$, our Theorem 1.5.1 tells us that $\phi \equiv \delta'$, where δ' is the dual of δ. What does the dual of a DNF formula look like? It is the *conjunction*

$$\phi_1 \wedge \cdots \wedge \phi_k,$$

where each ϕ_i is in turn a disjunction

$$L_{i1} \vee L_{i2} \vee \cdots \vee L_{is_i},$$

of literals. Such a formula is said, naturally enough, to be in *conjunctive normal form*, or CNF. We conclude:

Theorem 1.5.3. *Every propositional formula is equivalent to a formula in conjunctive normal form.*

Example. We will compute a CNF representation of the soup-and-salad condition. We begin again with the truth table, and this time select the five lines that contain **F** in the last column. Writing the corresponding assignments in DNF, and negating the whole thing, gives us an equivalent formula

$$\neg\big[(p \wedge q \wedge r) \vee (p \wedge q \wedge \neg r) \vee (p \wedge \neg q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge \neg q \wedge \neg r)\big].$$

We now apply DeMorgan's Law: We move the negation inside the outermost brackets, changing the $\vee$'s to $\wedge$'s. Then we apply DeMorgan's Law again, bringing negation symbols inside the inner brackets, changing $\wedge$'s to $\vee$'s and negating each literal. The resulting formula is in conjunctive normal form, and is equivalent to (1.5.1):

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

Formulas do not have unique representations in these normal forms. In particular, it is possible to give a simpler CNF formula that is equivalent to the one we just derived. The first two conjuncts of the above formula can be combined with the help of some of our basic identities:

$$\begin{aligned}(\neg p \vee \neg q \vee \neg r) \wedge (\neg p \vee \neg q \vee r) &\equiv (\neg p \vee \neg q) \vee (\neg r \wedge r)\\ &\equiv (\neg p \vee \neg q) \vee \mathbf{F}\\ &\equiv \neg p \vee \neg q.\end{aligned}$$

Similarly, the last two conjuncts are equivalent to $p \vee q$, so we get the simplified CNF formula

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

See Exercise 26.

1.6 Complete Sets of Connectives

As we noted above, any truth table can be realized by a propositional formula, and we even have some control over the form of the formula (*e.g.,* it can be in DNF or CNF). We have already mentioned that the 'new' connectives $\to$, $\leftrightarrow$ and $\oplus$ introduced in Section 1.4 are not, strictly speaking, necessary: They can all be expressed in terms of $\vee$, $\wedge$ and $\neg$. For example, $p \to q$ is equivalent to $\neg p \vee q$, and thus for any formulas ϕ and ψ, $\phi \to \psi$ is equivalent to $\neg\phi \vee \psi$.

Are all three of the original connectives necessary? By DeMorgan's Law (negating both sides),

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

This means that we don't need $\vee$ either: you can use this identity to replace every occurrence of $\vee$ by combinations of $\neg$ and $\wedge$. Thus every propositional formula is equivalent to one using only the two connectives $\neg$ and $\wedge$: These two form a *complete* set of connectives.

In precisely the same fashion, we can use the dual version of DeMorgan's Law to write $\phi \wedge \psi$ in terms of $\neg$ and $\vee$, and thus these two as well form a complete set of connectives.

Can we do better? Can we get things down to just a single connective? Using $\wedge$ by itself would not suffice, because then every formula could be put in the form.

$$p_1 \wedge \cdots \wedge p_k.$$

Such a formula could not be equivalent to $\neg p$. For the same reason, $\vee$ by itself does not suffice. And if all you had was $\neg$, you could not write a formula that involved more than one variable. So we seem to have come to the end of the line as far as reducing the number of connectives necessary in our logic.

Or have we? Let's define a new connective, called NAND (for 'not-and'). Its definition is

$$p \text{ NAND } q \equiv \neg(p \wedge q).$$

We then have

$$\neg p \equiv p \text{ NAND } p,$$

and

$$\begin{aligned} p \wedge q &\equiv \neg(p \text{ NAND } q) \\ &\equiv (p \text{ NAND } q) \text{ NAND } (p \text{ NAND } q). \end{aligned}$$

We already know that we can replace every propositional formula by an equivalent one that uses only the connectives $\wedge$ and $\neg$, so the foregoing equations show that we need only the single connective NAND.

Example. To illustrate this, we will show how to express the connective $\oplus$ using only NAND. It will be convenient to have a more compact notation for the NAND operator, so we will write $\phi|\psi$ in place of ϕ NAND ψ.

As we saw earlier,

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

We provisionally write $p \vee q$ as ϕ, so that the formula above is equivalent to

$$\begin{aligned} \phi \wedge (p|q) &\equiv \neg\neg(\phi \wedge (p|q)) \\ &\equiv \neg(\phi|(p|q)) \\ &\equiv (\phi|(p|q))|(\phi|(p|q)). \end{aligned}$$

We can, in turn, rewrite $\phi = p \vee q$ as

$$\begin{aligned} p \vee q &\equiv \neg(\neg p \wedge \neg q) \\ &\equiv (\neg p|\neg q) \\ &\equiv (p|p)|(q|q) \end{aligned}$$

Substituting this for ϕ in the previous formula gives

$$p \oplus q \equiv (((p|p)|(q|q))|(p|q))|(((p|p)|(q|q))|(p|q)).$$

Obviously, there is a big tradeoff here: reducing the number of connectives makes the resulting formulas quite long and difficult to read. Maybe expressing everything with just a single connective is not such a good idea after all!

Nonetheless, this is more than a mere mathematical curiosity, and even has some practical importance, as we'll see in the next chapter.

1.7 Historical Notes

The systematic study of logic was initiated by the Greek philosopher Aristotle (*circa* 350 BC), who undertook a classification of syllogistic reasoning, the kind contained in arguments like 'No dog is green. Rover is green. Therefore Rover is not a dog.' Aristotle used letters A, B, Γ, to denote the properties in the premises and conclusions of the syllogisms, and thus would write them as 'no B is Γ', and the like. This practice, which continued down through the centuries, embodied the crucial insight that logic deals with the *form* of propositions and how they are combined, and not their *content*.

It was widely believed up until at least the 19th century that Aristotle's system of logic was complete, and that it was not possible to make any further discoveries in the subject.[4] However, this view was challenged by a number of writers, beginning with Leibniz (late 17th century), who tried to work out the details of 'logical calculus', in which logical reasoning could be carried out by performing algebra-like computations.

[4]Kant (late 18th c.) for example, wrote, 'There are but few sciences that can come into a permanent state, which admits of no further alteration. To these belong Logic and Metaphysics. Aristotle has omitted no essential point of the understanding; we have only become more accurate, methodical, and orderly.' (Kant's 'Introduction to Logic').

The watershed development was the work of the British mathematician George Boole in the first half of the nineteenth century, who presented a detailed algebraic system of logic. Boole called his work *An Investigation of the Laws of Thought* and described its purpose as

> to investigate the fundamental laws of those operations of the mind by which reasoning is performed; to give expression to them in the symbolical language of a Calculus, and upon this foundation to establish the science of Logic and construct its method...

and, even more ambitiously,

> to collect from the various elements of truth brought to view in the course of these inquiries some probable intimations concerning the nature and constitution of the human mind.

It is from his name, of course, that we get 'boolean'.

Boole's methods were expanded upon and refined by the American C. S. Peirce, whose system ('The Algebra of Logic', 1880) begins to resemble the propositional logic described in this text. Peirce also was the first to observe that every propositional formula is equivalent to one using the single connective NAND.

We have said very little about modeling *deduction*. Much of the research in logic, beginning with Frege in 1879, was aimed at giving a precise foundation for mathematical reasoning, and extended far beyond propositional logic. *Logicomix: An Epic Search for Truth* (2009) is entertaining account of this quest, centered around the figure of Bertrand Russell, in the form of, of all things, a graphic novel. When restricted to propositional logic, the system of Frege gives a collection of axioms and rules of deduction from which all the tautologies can be deduced. The result of Exercise 31, which outlines how to derive every propositional identity from a small number of basic identities, is in this spirit.

Much of this history closely parallels that of the developing interest in automatic computation: Leibniz also invented a calculating machine. Boole and DeMorgan, who also wrote extensively on symbolic logic, were contemporaries and acquaintances of Charles Babbage, who designed a programmable general-purpose computer in the 1840's. William Stanley Jevons, beginning in the 1860's, both wrote on symbolic logic and designed a 'logic piano', a mechanical device for carrying out logical calculations. C. S. Peirce, in an 1886 letter to a former student, described how a logic machine could be built using electrical circuits.

We will see this theme reappear in the last chapter, when the history of logic and computation converge in the work of Turing.

1.8 Exercises

1.8.1 Syntax of propositional logic

1. Which of the following is a propositional formula? For purposes of this problem, you should *strictly* follow the rules given in the definition in 1.2.1, rather than the relaxed

rules discussed in Exercise 3 below and in 1.2.5. The relaxed rules allow you to leave off the outermost pair of parentheses in a formula, and to put an extra pair of parentheses around any subformula. How would these new rules change your answer?

(a) $p_1 \wedge \neg q$

(b) $((p_1 \wedge \neg q))$

(c) $(\neg(p_1 \wedge \neg q))$

(d) $(p_1 \wedge (\neg q \vee r))$

(e) $(p_1 \wedge \neg(q \vee r))$

(f) $(p_1 \vee \neg q \wedge r)$

(g) $(p_1(\vee)((\neg)q \wedge r)$

(h) $(p_1 \wedge ((\neg q) \vee r)$

2. For those parts of Exercise 1 that are legal formulas, show a derivation of the formula using the grammar in 1.2.4.

3. Our definition of propositional formula does not allow $(\neg p)$ or $(((p \wedge q)))$ as formulas. While the extra parentheses in these strings are unnecessary, one really ought to be allowed to place parentheses around any subformula occurring within a formula. On the other hand, we should not allow things like $(p(\wedge)q)$.

   (a) Add another rule to the grammar so as to allow these extra parentheses.

   (b) With this new rule, several of the illegal formulas in Exercise 1 become legal. Show the derivations of these formulas using the new grammar.

4. Another way to exhibit the syntactic structure of a formula is by use of a *parse tree*. In such a tree, the leaves are labeled by atomic formulas, internal nodes with one child by $\neg$, and internal nodes with two children by $\vee$ or $\wedge$.

   (a) You can probably figure out how the tree encodes a formula. Write down the formula represented by the parse tree in Figure 4. Include whatever parentheses are required by the definition.

   (b) Draw parse trees for the parts of Exercise 1 that represent legal formulas.

5. The following rules are part of a grammar for generating algebraic expressions.

$$\begin{aligned} E &\longrightarrow x|y|z \\ E &\longrightarrow E + E \\ E &\longrightarrow E \times E \end{aligned}$$

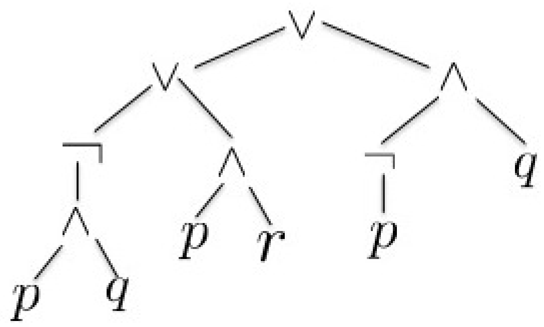


Figure 1.4: *Parse tree for Problem 4*

Find two different derivations of the expression $x + y \times z$ using this grammar. Which of these corresponds to the interpretation 'multiply first, then add', and which to the interpretation 'add first, then multiply'?

6. Even with our more relaxed rules, we still disallow ambiguous formulas like $p \vee q \wedge r$. Does Python allow boolean formulas of the form `p or q and r`? If so, how does it resolve the ambiguity? You can search the online documentation for the answer, but it's not easy to find. (See Exercise 7.) Alternatively, you can perform a simple experiment.

7.* How can we revise the grammar for propositional formulas so that it allows us to leave off the outermost pair of parentheses, or to add a new pair of parentheses around any formula? Thus it should be possible to derive $p \wedge (q \vee r)$ with this new grammar, but not $p \wedge q \vee r$ or, for that matter, $p \vee q \vee r$. (HINT: Our present grammar has just the single symbol `formula` to represent a grammatical category. Consider adding a new symbol representing a special kind of formula, those to which connectives can be applied...)

8.* The on-line Python documentation includes within it a grammar for boolean expressions. (In fact, it includes a grammar for all of Python!) Find where this grammar is hidden, and explain how it works. Show the derivation of `x and not y or z` in this grammar.

1.8.2 Semantics of propositional logic

9. In 1.3, we saw that the assignment $p \mapsto$ **true**, $q \mapsto$ **true**, and $r \mapsto$ **false** results in the formula $r \vee (\neg p \vee \neg(q \wedge p))$ having the value **false**. Is there a different assignment of truth values to the variables that gives the same result?

The next several problems ask you to translate a variety of rules, expressed in some other manner, into the language of propositional logic. These can all be done using only the standard connectives $\wedge, \vee$ and $\neg$, but you might find it helpful to use an occasional $\rightarrow$ in your formulas.

10. (Can you get in?) The following is a verbatim extract from academic regulations posted a few years back on the University of Missouri website. (The posting has since been modified, removing the interesting ambiguity in the version quoted here.)

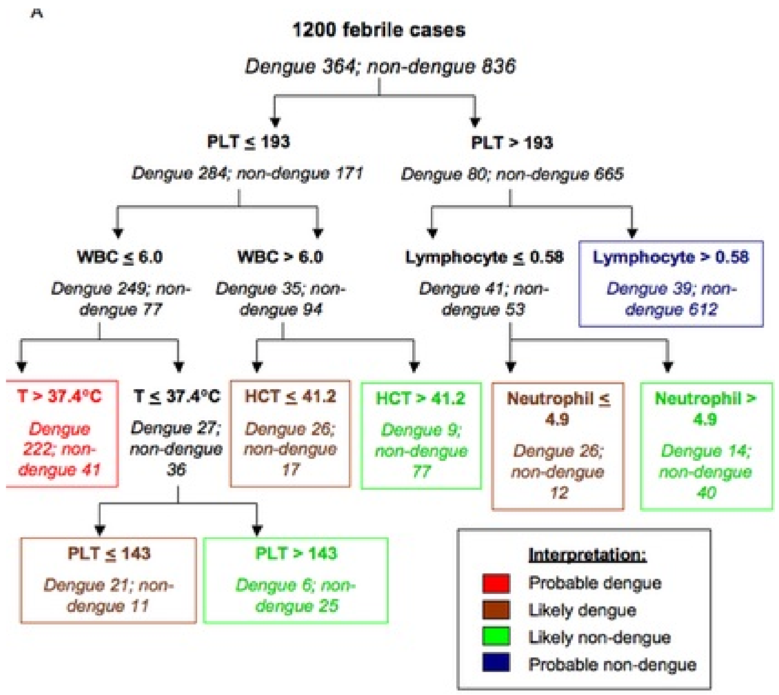
    > Incoming freshmen are eligible for automatic admission to the Honors College upon submission of an application, if they have 29 or higher on the ACT or 1280 on the SAT and are in the top 10 percent of their high school graduating class. Students from high schools that do not rank will be automatically eligible if their core GPA is greater than 3.70.

    The most natural interpretation of the language of the rule is that a student with a GPA higher than 3.70, graduating from a high school that does not rank its graduates, is automatically eligible, regardless of their standardized test scores. That is probably not what was intended, because why should test scores matter only in high schools that compute a class rank?

    Select variables to associate with the atomic propositions in these criteria. (One of the variables should denote 'the student's high school ranks its graduates', and another should be 'the student's ACT score is at least 29'. There should be five variables in all.) Then write two propositional formulas using these variables that are **true** if the student is eligible under these criteria and **false** otherwise. One of the formulas should capture the 'natural interpretation' described above, and the other the intended interpretation. [5]

11. (Do you have Dengue Fever?) The figure below is reproduced from a research article 'Decision tree algorithms predict the diagnosis and outcome of Dengue Fever in the early stage of illness'.

[5]Students to whom I assigned this problem in the past have pointed out other ambiguities to me: For example, you have to have '29 or higher on the ACT' or '1280 on the SAT'. The rule does not say '1280 or higher', which allows for the interpretation that you have to hit 1280 right on the nose. Should the first condition '29 or higher on the ACT or 1280 on the SAT and are in the top 10 percent of their high school graduating class' be read as $(p \vee q) \wedge r$ or as $p \vee (q \wedge r)$? One of these makes much more sense than the other. For these additional ambiguities, assume the most plausible interpretation.



A positive diagnosis ('Probable dengue' or 'Likely dengue' in the legend) is made based on the platelet count (PLT), hematocrit (CT), white blood cell count (WBC) and body temperature (T). Choose appropriate interpretations for the propositional variables (one of them, for example, should be '$WBC \leq 6.0$') and write a propositional formula using these variables that is **true** if and only if the diagnosis is positive.

12. (Are you kosher?) The following is an extract from the Bible (Leviticus 11):

> The Lord spoke again to Moses and to Aaron, saying to them, "Speak to the sons of Israel, saying, 'These are the creatures which you may eat from all the animals that are on the earth. Whatever divides a hoof, thus making split hoofs, and chews the cud, among the animals, that you may eat. Nevertheless, you are not to eat of these, among those which chew the cud, or among those which divide the hoof: the camel, for though it chews cud, it does not divide the hoof, it is unclean to you. Likewise...the pig, for though it divides the hoof, thus making a split hoof, it does not chew cud, it is unclean to you.
>
> 'These you may eat, whatever is in the water: all that have fins and scales, those in the water, in the seas or in the rivers, you may eat. But whatever is in the seas and in the rivers that does not have fins and scales among all the teeming life of the water, and among all the living creatures that are in the water, they are detestable things to you, and they shall be abhorrent to you; you may not eat of their flesh, and their carcasses you shall detest. Whatever in the water does not have fins and scales is abhorrent to you.' "

Introduce propositional variables encoding 'this animal lives on land', '...lives in the water', '..has scales', '...is okay to eat', *etc.* There should be seven variables in all. Write a propositional formula using these variables that encodes this commandment. Observe that God has helpfully provided Moses and Aaron with some examples illustrating the rules, but your answer should not refer to camels or pigs. By the way, this is not an exhaustive list of what is and isn't okay to eat; the actual biblical commandment goes on to discuss birds, bugs, and reptiles. As a result, your answer will admit some assignments that aren't consistent with the complete rule.

13. (Are you being served?) Figure 1.5 shows another sort of commandment.

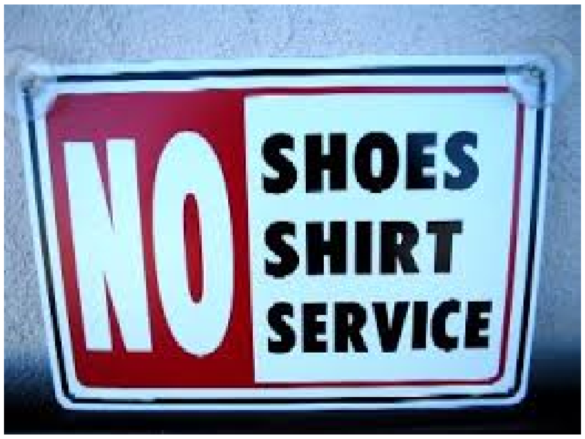


Figure 1.5: No shoes

Once again, identify the atoms in the proposition to encode by variables, and write a propositional formula equivalent to the rule. Be careful—it doesn't say that if you ARE wearing shoes and a shirt, then you get service. You might not be wearing pants.

14. Construct the truth table for the formula $r \vee (\neg p \vee \neg(q \wedge p))$ that we considered earlier. Is this formula satisfiable? Is it a tautology?

15. How many rows does the truth table you constructed in the last problem have? The examples in Exercises 10 and 11 each have five variables. Without actually constructing the truth table, find how many rows the truth tables for these formulas have. How did you figure this out? What is the general rule?

16. Consider again the college admission problem in Exercise 10. Without constructing the truth table, tell how many satisfying assignments there are—that is, how many rows of the truth table have **T** in the final column? There are two different answers for this question, depending on which of the two interpretations of the rule you use.

17. In each part of the problem below, tell whether the formula is satisfiable, or a tautology, or a contradiction. You can always solve these problems by explicit construction of a

truth table, but it is often more efficient to talk your way through it, reasoning from the formula itself ('...what would it take to make this formula true?') to constraints on the values of the variables.

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

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

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

(d) $(p \wedge q) \to (\neg p \vee r)$

(e) $(p \wedge q) \oplus (\neg p \vee r)$

(f) $(p \wedge q) \to (p \vee q)$

(g) $(p \vee q) \to (p \wedge q)$

18. Consider the two formulas

$$p \to ((p \to q) \to q), (p \to (p \to q)) \to q$$

(a) One of these is a tautology, and the other is not. Determine which is the tautology. Is the formula that is not a tautology satisfiable? If so, give a satisfying assignment; if not, explain why not.

(b) Write a formula that is equivalent to the one that is not a tautology, and that is as simple as possible.

1.8.3 Equivalence

19. Prove the last identity (the distributive law) in Table 1.6, using either a truth table or a 'talk it through' argument.

20. Prove the following identities involving the connective $\to$.

(a) $p \vee q \equiv \neg p \to q.$

(b) $p \wedge q \equiv \neg(p \to \neg q).$

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

(d) $(p \to r) \vee (q \to r) \equiv (p \wedge q) \to r.$

21. We saw that the connectives $\vee$ and $\wedge$ obey the associative and commutative laws. What about the connectives $\to$, $\leftrightarrow$ and $\oplus$? (There are six different questions here: for each of the three connectives, you have to determine whether it is associative, and whether it is commutative.)

22. Let ϕ, ρ be formulas, and let p be a variable appearing in ϕ. We denote by $\phi[p \mapsto \rho]$ the formulas that results when every occurrence of p in ϕ is replaced by ρ. As we remarked in the text, if

$$\phi \equiv \psi$$

then

$$\phi[p \mapsto \rho] \equiv \psi[p \mapsto \rho].$$

(a) Let ϕ be a tautology. Is $\phi[p \mapsto \rho]$ necessarily a tautology?

(b) Same question, with 'tautology' replaced by 'satisfiable formula'.

(c) Same question, with 'tautology' replaced by 'contradiction'.

23. Find the duals of the formulas in parts (a)-(c) of Exercise 17

24. Suppose you are given the truth table for a formula ϕ. Describe general procedure for obtaining the truth table for the dual ϕ'. It's possible to figure out such an algorithm by looking at the way we 'proved' the duality theorem Theorem 1.5.1. If you don't see it, try a few particular cases, constructing the truth tables for formulas together with their duals, and seeing if you can discern the rule.

25. Find DNF formulas equivalent to those of Exercise 17. Then find equivalent CNF formulas.

26. Find a CNF formula for the soup-and-salad example that is even simpler than the one given in the example in the text.

27. Can a formula be simultaneously a DNF and a CNF formula? The answer is yes! Give at least three different examples.

28.* Describe an algorithm for determining if a formula in DNF is satisfiable. First describe how to determine if a conjunction of literals is satisfiable. Then show how to use this to determine if the disjunction of such conjunctions is satisfiable. The algorithm that you produce should be *fast* in the sense that it only requires a single scan of the formula, so that the time required to determine satisfiability of a DNF formula with n literals is proportional to n.

29.* The preceding problem implies that there is an algorithm for determining if *any* formula is satisfiable: Convert it to an equivalent DNF formula, and apply the algorithm for satisfiability of DNFs. How fast is this algorithm?

30.* Our algorithm form converting a formula to an equivalent formula in CNF proceeded by using truth tables, first extracting a DNF formula for the either the dual or the negation of the original formula. The purpose of this problem is to outline another method for putting a formula in CNF that proceeds by directly transforming the formula through basic identities.

(a) Show how, by use of the distributivity of $\vee$ over $\wedge$, a formula in DNF can be converted to CNF. (First practice on $(p_1 \wedge \neg p_2) \vee (q_1 \wedge q_2))$.)

(b) Use part *(a)* to show how, given, a formula ϕ in CNF, we can find a CNF representation of $\neg\phi$.

(c) Given formulas ϕ and ψ in CNF, show how to find a CNF representation of $\phi \wedge \psi$. Why is this almost a silly problem?

(d) Given formulas ϕ and ψ in CNF, show how to find a CNF representation of $\phi \vee \psi$. (Use *(b)*, *(c)*, and DeMorgan's Laws for this.)

(e) Now, put it all together and describe an algorithm for converting a formula into an equivalent CNF formula. Demonstrate your algorithm with the formula $\neg((p \wedge q) \vee \neg(p \wedge r)) \vee (q \wedge r)$.

31.* The purpose of this exercise is to show that the identities in Tables 1.6 and 1.7 are *complete,* in the sense that every propositional identity can be derived from them. That is, there are no 'missing' identities.

To set the stage, we have to be a little bit more precise about what it means to derive an identity from other identities. Propositional identities follow some basic familiar rules of algebra:

- If $\phi \equiv \psi$ is an identity, then so is $\psi \equiv \phi$.
- If $\phi \equiv \psi$ and $\psi \equiv \rho$ are identities, then so is $\phi \equiv \rho$.
- If $\phi \equiv \psi$ is an identity, and ρ is any formula, $\phi \vee \rho \equiv \psi \vee \rho$, $\phi \wedge \rho \equiv \psi \wedge \rho$, and $\neg\phi \equiv \neg\psi$ are identities.
- Suppose $\phi \equiv \psi$ is an identity. Let p be a variable, ρ a formula, and $\phi[p \mapsto \rho]$, $\psi[p \mapsto \rho]$ the formulas that result when every occurrence of p in ϕ and ψ is replaced by ρ. Then
$$\phi[p \mapsto \rho] \equiv \psi[p \mapsto \rho]$$
is an identity. (See Exercise 22).

(a) Show that a formula ϕ in CNF is a tautology if and only if every clause of literals
$$L_1 \vee \cdots \vee L_k$$
that it contains has $L_i = p$ and $L_j = \neg p$ for some variable p and indices i, j.

(b) Now use the results of the preceding problem along with part *(a)* to show that if ϕ is a tautology, then $\phi \equiv \mathbf{T}$ can be derived from the identities in Tables 1.6 and 1.7, along with the above rules.

(c) Now suppose $\phi \equiv \psi$ is an identity. Then $\phi \leftrightarrow \psi$ is a tautology (why?). From part *(b)* above, we can derive the identity
$$(\phi \leftrightarrow \psi) \equiv \mathbf{T}.$$

Show that from this we can derive the identity

$$\phi \wedge \psi \equiv \phi.$$

(HINT: AND both sides of the identity with ϕ.)

(d) Show similarly that we can derive

$$\phi \wedge \psi \equiv \psi$$

and conclude that we can derive $\phi \equiv \psi$.

32. The if-then-else statement in the Python code in Figure 1.1 at the beginning of the chapter can be represented as a *binary decision tree* (Figure 1.6). Each interior node of such a tree is labeled by a boolean variable (or, as in this example, an atomic boolean formula) and branches to another node, depending on whether the variable has the value **true** or **false**. The leaves of the tree are labeled True or False. The tree in Exercise 11 provides another example, although one with more than two possible outcomes at the leaves.

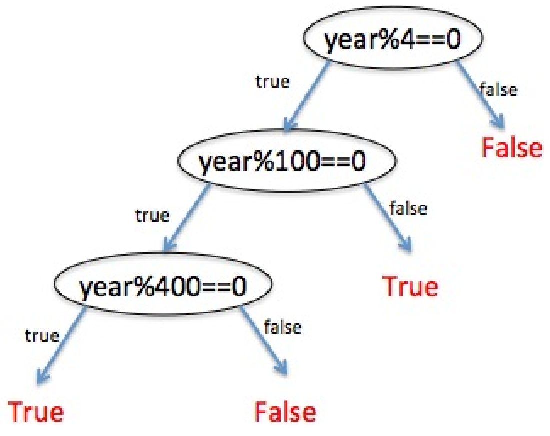


Figure 1.6: A binary decision tree derived from the Python code in Figure 1.1.

(a) The essence of the example at the start of the chapter is to show that the binary decision tree represented by this Python code can be replaced by a single propositional formula. Describe how to turn *any* binary decision tree into an equivalent propositional formula. Illustrate your method with the tree shown in Figure 1.7

(b) Conversely, describe how to convert any propositional formula into an equivalent binary decision tree. Illustrate your method with any of the formulas from Exercise 17.

(c) Describe how to quickly determine from a binary decision tree whether it represents a tautology, or a satisfiable formula.

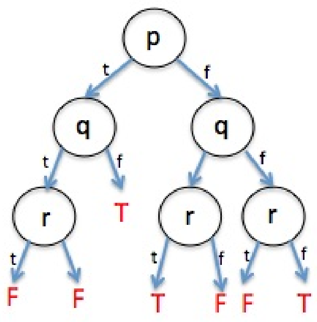


Figure 1.7: Another binary decision tree.

1.8.4 Complete sets of connectives

33. Find formulas, using only $\neg$ and $\wedge$, equivalent to the formulas in Exercise 17. Find equivalent formulas using only $\neg$ and $\vee$.

34. Find formulas, using only NAND, equivalent to $p \to q$, $p \oplus q$, and $p \leftrightarrow q$.

35. Argue that the two connectives $\neg$ and $\to$ form a complete set of connectives. Write formulas using these two connectives equivalent to those in Exercise 17.

36. Suppose we define

$$p \text{ NOR } q \equiv \neg(p \vee q).$$

Argue that every propositional formula can be expressed using NOR alone. Then find formulas, using only NOR, equivalent to $p \to q$, $p \oplus q$, and $p \leftrightarrow q$.

37.* Give a convincing argument that neither the pair $\vee, \wedge$, nor the pair $\neg, \oplus$ is a complete set of connectives. This is a hard problem to solve at this stage of the course, but it is worth thinking about. Somehow you will have to identify properties shared by all formulas built using only $\vee$ and $\wedge$ that are not true of every propositional formula, and similarly for the pair $\neg, \oplus$. For starters, think about what happens to the value of such formulas when we change the value assigned to one variable from **F** to **T**, and vice-versa.

See Exercise 16 of Chapter 6 for a real proof of these claims.

Chapter 2

Applications of Propositional Logic

2.1 Digital Logic

2.1.1 Computation as Bit-manipulation

Attorney: So, let's get you prepared for your testimony tomorrow. Do you know what color my dress is?

Client: It's green.

Attorney: Wrong! My dress **is** *green, but the right answer is 'yes'.*

The smallest amount of information you can give, and still give *some* information, is the answer to a single yes-or-no question. It is a common convention to denote the **yes** answer by 1 and the **no** answer by 0, and to call each such unit of information a *bit,* both because it is a tiny little bit of information, and also because it is a **b**inary dig**it**.

A critical insight of Computer Science is that *all* information can be encoded as sequences of bits.

An example (which long predates computers) is the Braille alphabet for the blind, which represents each text character as a grid of 6 dots, some of which are raised from the surface of the page. (See Figure 2.1.) If we think of the raised dots as 1s and the unraised dots as 0s, and read from left to right starting at the top row, then the letter 'a' is encoded by the bit sequence 100000, the letter 'z' by 100111. The ubiquitous seven-segment displays that you see on every sort of electronic device (Figure 2.2) encode each of the ten decimal digits by a sequence of seven bits: If we denote a lighted segment by 1 and an unlighted segment by 0, and read the segments from left to right starting at the top, then, for example, '1' is encoded by 0010010 and '4' by 0111010.

But even very complex information that you do not ordinarily think of as textual or numeric can be encoded by sequences of bits. A video (without the sound) consists of a sequence of frames, each frame is a grid of pixels, each pixel a collection of three numbers giving the relative quantities of red, green and blue in the pixel, and each number in turn encoded by a sequence of bits.

Since all the information a computer manipulates is represented by sequences of bits, you can view any computation as a process that takes as its input a sequence $a_1a_2\cdots a_m$ of bits, and produces as its output another sequence $b_1\cdots b_n$ of bits. Let us look at a very simple example, called a *full adder*: here we take a sequence $a_1a_2a_3$ of bits as an input, and compute the sum

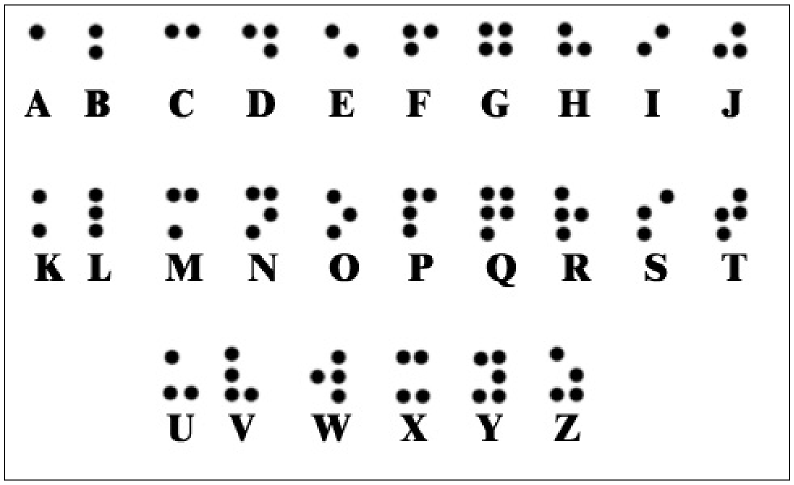


Figure 2.1: *The Braille Alphabet for the blind. Each letter is encoded as a sequence of six bits.*

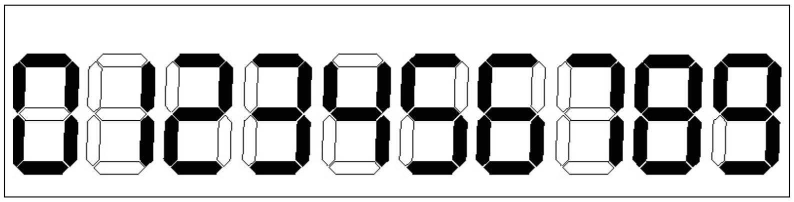


Figure 2.2: *Decimal digits on a seven-segment display. Each digit is encoded as a sequence of seven bits.*

$a_1 + a_2 + a_3$. The possible values for the sum are 0,1,2,3. We will encode each of these possibilities as a sequence of two bits:[1]

$$\begin{array}{ccc} 0 & \leftrightarrow & 00 \\ 1 & \leftrightarrow & 01 \\ 2 & \leftrightarrow & 10 \\ 3 & \leftrightarrow & 11 \end{array}$$

We can represent the process by a table giving the complete input-output relation. (Table 2.1.1.) The table itself depends on how we choose to encode the inputs and outputs by bits; here we have used the standard encoding of numbers by bits.

This, of course, is a truth table—just think of the 1s as **T**s and the 0s as **F**s. As we noted in the previous chapter, the output columns in such a table can be represented by propositional formulas whose variables represent the inputs, and we can read off these formulas in disjunctive normal form.

[1]This is the standard binary encoding of integers by bits, which we will discuss in detail in Chapter 7: The right-hand digit is the 1s position, the left-hand digit is the 2s position; if we had another digit further to the left it would be the 4s position, *etc.*

input 1	input 2	input 3	output 1	output 2
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 2.1: Input-output table for the full adder

In this case, we'll represent the three inputs by p, q, r respectively. The output formulas are:

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

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

These formulas can be simplified: you can show that the first of these formulas is equivalent to

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

and the second to

$$p \oplus q \oplus r.$$

(In connection with this last formula, see Exercise 21 of Chapter 1.)

2.1.2 Truth tables realized by networks of logic gates

We saw above that any computation can be realized by a truth table, and thus by a collection of propositional formulas. Each propositional formula can in turn be realized in hardware. The idea is that each logical connective is simulated by a simple device, called a *logic gate.* The gates, in turn, are connected together in a network whose inputs are the values of the variables and whose output is the resulting value of the formula. The exact implementation of the gates in hardware is irrelevant. In principle, the gate could be a mechanical spring-driven device, like an old-fashioned clock, or run on steam, but typically the gates are small electrical circuits, and the logical values **true** and **false** are represented by high and low voltage levels. Thus, for example, the connective NAND is implemented by a circuit with two inputs and a single output. Setting the voltage at both of the inputs high switches the voltage at the output low, while setting either input voltage low results switches the output voltage high. Standard graphical symbols for logic gates are shown in Figure 2.3.

A full adder circuit is thus constructed by building the two formulas from these basic gates. (Figure 2.4)

The existence of small, functionally complete, sets of connectives means that we can construct all such circuits using only a few basic gate types—in fact, using only a single gate type. As we

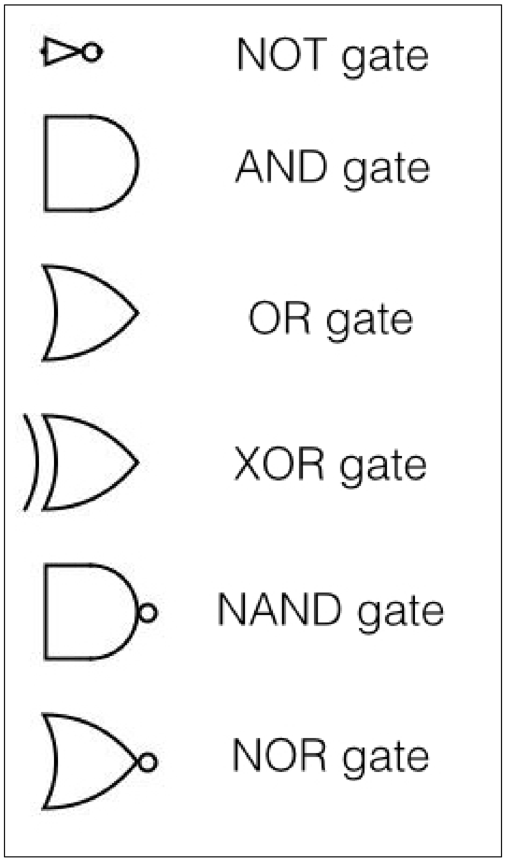


Figure 2.3: Standard Gate Symbols

saw before, the NAND connective suffices all by itself, and in fact a NAND gate is a particularly simple hardware component to construct. Figure 2.5 shows the construction of an exclusive-or gate from NAND gates, using the formula we computed in 1.6.

In the same manner *any* input-output table can be realized by a network consisting of many copies of this one type of gate. So in a very real sense, any computation at all can be carried out by repeatedly performing this single simple computation. We will return to this point in the exercises for this chapter, and later in the book.

2.2 Liar Puzzles

There is a large class of puzzles about an island in which there are two groups of inhabitants: Members of one group always lie, members of the other always tell the truth. The rules of this world are such that the liars are just as reliable as the truth-tellers: They are not trying to deceive you; they are not trying to do *anything*. They just can't help themselves; every statement they make is false.

The puzzles below ask you to determine which of the two groups the characters belong to, based on the statements they make, or to devise a question that will extract accurate information

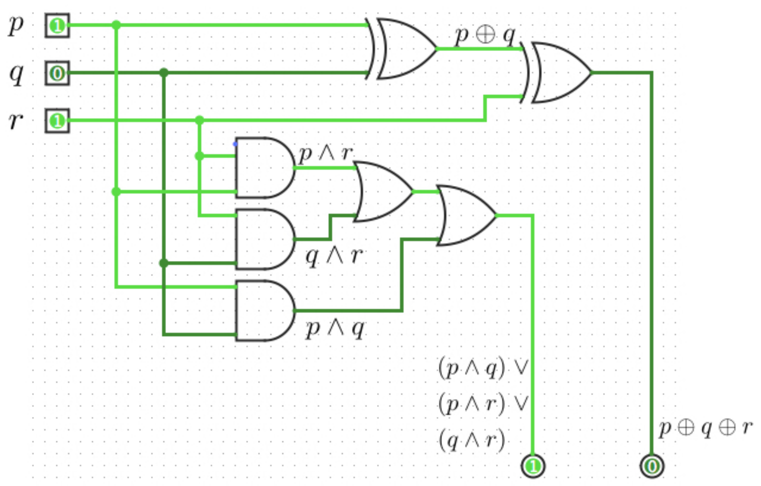


Figure 2.4: The Full Adder Circuit

from an inhabitant, whether he is a truth-teller or a liar. The puzzles themselves, along with the Knights-and -Knaves/Humans-and-Vampires back story, are all due to the mathematician Raymond Smullyan.

This is not really a practical application; but questions about statements that assert their own falsehood play a serious role in logic and Computer Science, as we will see later.

2.2.1 Some Knights-and-Knaves Puzzles

Knights only make true statements. Knaves only make false statements.

Puzzle 1.

A says, 'B and I are both knights'. B says 'A is a knave'. Determine, if possible, which groups A and B belong to.

Puzzle 2.

You meet an islander. Ask her a single question to determine whether she is a knight or a knave. (Observe that 'Are you a knight?' doesn't work!)

Puzzle 3.

You come to a fork in the road. An inhabitant of the island stands at the fork. You know that one of the roads leads to the village, where there is a decent restaurant and a comfortable hotel, and that the other road leads to a crocodile-infested swamp. You have to ask *one* question whose answer will tell you which road leads to the village. What do you ask?

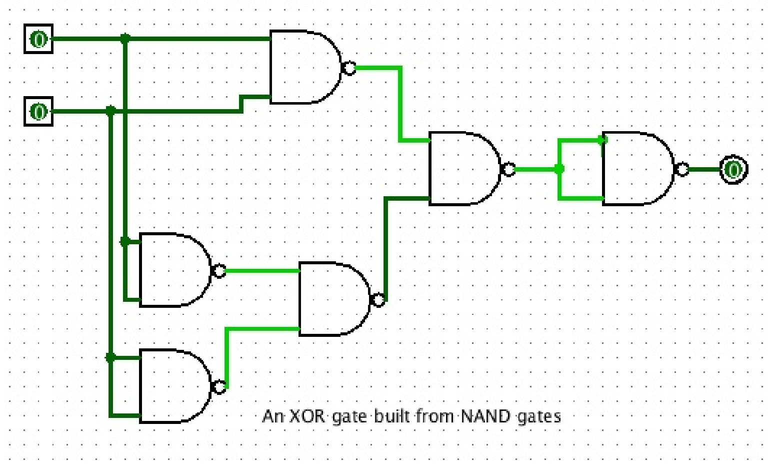


Figure 2.5: An exclusive-or gate built from NAND gates

2.2.2 A Vampire Puzzle

In Transylvania, everyone is a human or a vampire, and either sane or insane. Humans always say what they believe to be true, and vampires always say what they believe to be false. Sane beings only believe what is true, and insane beings only believe what is false. As a result, sane humans and insane vampires always tell the truth, and sane vampires and insane humans always lie.

Exactly one of the sisters Lucy and Minna is a vampire. They have the following exchange with Inspector Craig of Scotland Yard:
Lucy: We are both insane.
Craig: Is that true?
Minna: Of course not!

Which one of the sisters is the vampire?

2.2.3 Modeling the puzzles with propositional logic

Here we will use the methods of the first chapter to solve these puzzles. We begin with Knights-and-Knaves puzzles. For each individual $A, B, C, \ldots$ in the problem, introduce a propositional variable $a, b, c, \ldots$. The variable a is true if A is a knight and false if A is a knave; similarly $b, c, \ldots$.

If A asserts that a proposition x is true (or, what is the same thing, answers 'yes' to the question 'Is x true?') then the proposition $a \leftrightarrow x$ is true. This is because if A is a knight, and A asserts x, then x is true, while if A is a knave and A asserts x, then x is false. Similarly, if A denies x (answers 'no' to 'Is x true') then $a \leftrightarrow x$ is false. So for any proposition x, $a \leftrightarrow x$ is equivalent to 'A asserts x'.

How does this help us solve the puzzles? Look at Puzzle 1 : The data in the problem tells us that both

$$a \leftrightarrow (a \wedge b), b \leftrightarrow \neg a$$

are true. The problem is to determine what values of a and b lead to both these propositions being true. We can solve this with truth tables.

a	b	$a \wedge b$	$a \leftrightarrow (a \wedge b)$	$b \leftrightarrow \neg a$
T	T	T	T	F
T	F	F	F	T
F	T	F	T	T
F	F	F	T	F

Table 2.2: Truth table used to solve Puzzle 1

a	p	$a \leftrightarrow x$	x
T	T	T	T
T	F	F	F
F	T	T	F
F	F	F	T

Table 2.3: Truth table used to solve Puzzle 3.

There is only one row of Table 2.2 where both of the last columns have the entry **T**. This corresponds to A knave and B knight, so that is the solution.

Let's do Puzzle 2. The problem is to find a proposition x so that $a \leftrightarrow x$ has the same truth value as a. This makes the answer to 'Is x true?' the same as the *correct* answer to 'Are you a knight?' We just need a proposition x that is always true regardless of the value of a So we can ask

"Is two plus two equal to four?"

For Puzzle 3,we'll again calculate with truth tables. Let p be the proposition 'the left road leads to the village'. We need to find x so that $a \leftrightarrow x$ has the same value as p. We can then ask 'Is x true?' In Table 2.3 we set the value of $a \leftrightarrow x$ to be identical to p, and deduce what x has to be as a result.

From the table we find that proposition x we need for our question must have the same value as $a \leftrightarrow p$, which is equivalent to 'A asserts p'. So we can ask:

"Would you say that the left road leads to the village?"

In case you don't quite see it: If the islander is a Knave, and the road really leads to the village, she would say that the left road does not lead to the village (since she always lies). Thus she must lie about what she would say, and thus answer the question above "Yes". Note that in our rigid logical world, this is a different question from "Does the left road lead to the village?"

Now let's solve the Vampire puzzle. For each individual $A, B, C \ldots$, we introduce two variables $a, a', b, b', c, c', \ldots,$ where a means 'A is human', and a' means 'A is sane', and similarly for the other pairs of variables. Observe that 'A always tells the truth' is equivalent to A being both sane and human or neither sane nor human, that is, to $a \leftrightarrow a'$. Thus 'A asserts x' is equivalent to

$$(a \leftrightarrow a') \leftrightarrow x.$$

l	l'	m	m'	$l \leftrightarrow l'$	$m \leftrightarrow m'$	$l' \vee m'$	$(l \leftrightarrow l') \leftrightarrow \neg(l' \vee m')$	$(m \leftrightarrow m') \leftrightarrow (l' \vee m')$
T	T	T	T	T	T	T	F	T
T	T	T	F	T	F	T	F	F
T	T	F	T	T	F	T	F	F
T	T	F	F	T	T	T	F	T
T	**F**	**T**	**T**	**F**	**T**	**T**	**T**	**T**
T	F	T	F	F	F	F	F	T
T	F	F	T	F	F	T	T	F
T	F	F	F	F	T	F	F	F
F	**T**	**T**	**T**	**F**	**T**	**T**	**T**	**T**
F	T	T	F	F	F	T	T	F
F	T	F	T	F	F	T	T	F
F	**T**	**F**	**F**	**F**	**T**	**T**	**T**	**T**
F	F	T	T	T	T	T	F	T
F	**F**	**T**	**F**	**T**	**F**	**F**	**T**	**T**
F	F	F	T	T	F	T	F	F
F	F	F	F	T	T	F	T	F

Table 2.4: Truth table for the Vampire puzzle.

In our problem, there are four variables l, l', m, m'. The sisters' statements are

$$(l \leftrightarrow l') \leftrightarrow \neg(l' \vee m')$$

$$(m \leftrightarrow m') \leftrightarrow (l' \vee m').$$

In other words, Minna is asserting that at least one of them is sane, and Lucy the negation of this—that they are both insane. Let's look at the truth table for these two propositions:

There are four rows in which the last two columns are true. The first such row corresponds to an assignment in which both Lucy and Minna are human, which we can rule out, since we know that one of them is a vampire. Another of these rows has them both being vampires, which we similarly rule out. The remaining two solutions have Lucy being a vampire and Minna a human, so that is the solution. We cannot tell from the information given whether or not they are sane, but we know that they are either both sane or both insane.

If you find yourself thinking that constructing a truth table with over one hundred entries is overkill, you are quite right. We could have deduced the correct solution by thinking it through: If Minna is a vampire, she is either sane or insane. If she is insane, then what she asserts — namely that at least one of them is sane – is true. That means that Lucy is a sane vampire, contradicting our assumption that exactly one of them is a vampire. If Minna is a sane vampire, then what she asserts is false, and thus she would have to be insane—a contradiction. So we cannot have a solution in which Minna is a vampire and Lucy is human.

But the question for us, as computer scientists, is whether we can automate this reasoning so that a computer can solve the puzzles, and this is something that the truth tables provide. While the truth tables are easy to program, they quickly become impractically large as the number of variables increases. We will return to this point in the next section.

2.3 Satisfiability Problems

Satisfiability problems have the following form: You're given a propositional formula ϕ, and you have to find an assignment of truth values to the variables that makes ϕ true, or to determine that no such assignment exists. In other words, you have to determine whether ϕ is satisfiable, and if it is, to find a satisfying assignment.

Below we give examples of several such problems. Each example is a toy problem, but each also belongs to a class of problems that arise in practice. We'll discuss how to translate them into the form described, and say something about how to go about solving them.

2.3.1 A Scheduling Problem

An overly eager student wants to sign up for seven courses: English, Math, Chinese, History, Computer Science, Philosophy and Music. Each class meets just once a week, in either a long morning session, or a long afternoon session, Monday through Friday. The student is trying to make a schedule that meets the following constraints:

- The student does not want to schedule any classes on Wednesday, a day he prefers to stay in bed.
- The English professor allows students to drink beer in class, so our student would like to schedule English for Friday afternoon.
- Philosophy is only taught on Monday.
- History is only taught in morning sessions.
- The student would like to schedule Chinese and Music on the same day.

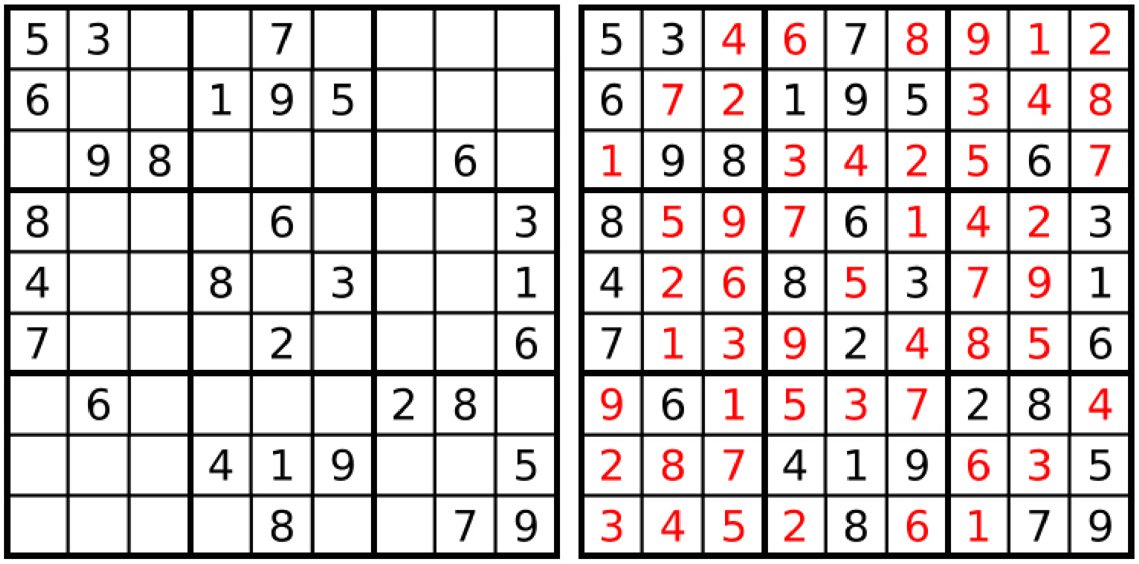
The problem is to find a schedule that meets these constraints.

2.3.2 Sudoku: A Combinatorial Design Problem

You have probably Sudoku puzzles. An example puzzle, together with its solution, is shown in Figure 2.3.2. [2]

The problem is to fill in the boxes in the grid with the integers 1 through 9 in such a manner that no number appears twice in the same row, nor twice in the same column, nor twice in any of the outlined 3x3 subgrids.

[2]Source for these images: "Sudoku-by-L2G-20050714 solution" by en:User:Cburnett - Added red numbers based on en:Image:Sudoku-by-L2G-20050714.svg. Licensed under CC BY-SA 3.0 via Commons -https://commons.wikimedia.org/wiki/File:Sudoku-by-L2G-20050714_solution.svg#/media/File:Sudoku-by-L2G-20050714_solution.svg.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 2.6: A Sudoku puzzle and its solution

2.3.3 Modeling the examples as Satisfiability Problems

The input to a satisfiability problem is a boolean formula ϕ, and the solution is either a satisfying assignment to the variables in the formula, or a determination that no such assignment exists.

Both of the problems given above can be modeled as satisfiability problems. In each case, we can express the required condition as a collection of separate constraints that must simultaneously be satisfied. This suggests that the condition ought to be represented in conjunctive normal form.

Let's start with the scheduling problem. There are seven courses and eight time slots. Our formula will have 56 different variables, one for each combination of course and time slot. We'll denote the variables in this manner:

$$\mathrm{Mu}_{m,Th}, \mathrm{Ph}_{a,F},$$

etc., where the first of these is true if Music is scheduled for Thursday morning, and the second is true if Philosophy is scheduled for Friday afternoon. We construct the clauses of ϕ as follows:

- Each of the seven subjects must be scheduled at one of the available times. This gives a clause

$$\mathrm{Mu}_{m,M} \vee \mathrm{Mu}_{a,M} \vee \mathrm{Mu}_{m,T} \vee \mathrm{Mu}_{a,T} \vee \mathrm{Mu}_{m,Th} \vee \mathrm{Mu}_{a,Th} \vee \mathrm{Mu}_{m,F} \vee \mathrm{Mu}_{a,F},$$

  for Music, and six others just like it for the remaining subjects.

- Different classes cannot be scheduled in the same time slot. This gives, for instance, a clause

$$\neg\mathrm{Mu}_{a,T} \vee \neg\mathrm{Ch}_{a,T},$$

  which says that Music and Chinese cannot both be scheduled for Tuesday afternoon. Altogether, there are $21 \times 8 = 168$ such clauses, since there are 21 different pairs of subjects and 8 time slots.

- A single class cannot be scheduled at two different time slots. This gives rise to clauses like

$$\neg \mathrm{Mu}_{a,T} \vee \neg \mathrm{Mu}_{m,F}.$$

  There are 196 such clauses in all.

- English has to be taken Friday afternoon:

$$\mathrm{En}_{a,F}$$

- Philosophy has to be taken Monday:

$$\mathrm{Ph}_{m,M} \vee \mathrm{Ph}_{a,M}$$

- The constraint about Chinese and Music can be expressed by the conjunction of eight formulas, the first two of which are

$$\mathrm{Ch}_{m,M} \to \mathrm{Mu}_{a,M}, \mathrm{Ch}_{a,M} \to \mathrm{Mu}_{m,M},$$

  and the remaining six of which use the three other available days. Furthermore, an implication $a \to b$ can be rewritten as $\neg a \vee b$.

So the CNF for our formula has 56 variables and 382 clauses, the majority of which just specify that there is a unique spot in the schedule for each class. The large size of the CNF, compared to the relatively succinct original specification of the problem, seems to have made things harder, rather than easier. In fact, as we'll see below, it is the key to automating the solution.

For Sudoku, we have $9 \times 9 \times 9 = 729$ boolean variables, which we'll denote $p_{i,j,k}$ where i, j, k are integers between 1 and 9 inclusive. $p_{i,j,k}$ is true if the cell in row i and column j contains the number k. The constraints can be formulated as CNF clauses in the following way:

- Every cell contains a label. This gives 81 clauses

$$p_{i,j,1} \vee p_{i,j,2} \vee \cdots \vee p_{i,j,9}$$

  one for each choice of i, j. We can denote each of these with the more compact notation

$$\bigvee_{k=1}^{9} p_{i,j,k}.$$

- No cell contains two labels. This gives a clause

$$\neg p_{i,j,k} \vee \neg p_{i,j,k'}$$

  for all i, j, k, k' with $k \neq k'$. There $81 \times 36 = 2916$ such clauses.

- Every row contains every value. For example, to say row 3 contains 7, we have the clause

$$\bigvee_{j=1}^{9} p_{3,j,7}.$$

  There are 81 such clauses, one for each row and each value .

- Similarly we get 81 clauses saying that each column contains every value. For instance

$$\bigvee_{i=1}^{9} p_{i,5,4}$$

  says column 5 contains a 4.

- And, similarly, there are 81 clauses saying that each subgrid contains every value. The following clause says that the central subgrid contains 2:

$$\bigvee_{i=4}^{7} \bigvee_{j=4}^{7} p_{i,j,2}.$$

- Finally, there is a small number of additional clauses that give the initial configuration of the puzzle. For instance, $p_{3,3,3}$ says that the cell in the third row of the third column contains three.

2.3.4 Satisfiability Solvers

A *satisfiability solver,* also called a *SAT solver,* is a program for solving satisfiability problems. The input formula is usually given in conjunctive normal form, and the output is either a satisfying assignment, or a message that the formula is unsatisfiable.

Most SAT solvers in use today share a standard format for encoding CNF formulas as text files, called the DIMACS format. The file starts with optional comment lines, each of which begins with the letter `c`. The comment lines are followed by a line of the form

`p cnf` *num1 num2*

where *num1* is the number of variables and *num2* is the number of *clauses,* or conjuncts. The remaining lines of the file contain the conjuncts themselves. Let us denote the variables $p_1, p_2, \ldots$. If the literal p_i appears in the clause, then the number i appears in the line. If the literal $\neg p_i$ appears in the clause, then $-i$ appears in the line. Each such line is terminated by 0.

For example, recall that our soup-and-salad format in CNF is

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

We'll treat p as p_1, q as p_2, and r as p_3.

There are three variables and three clauses. The complete text file is given by:

```
c CNF file for soup and salad
c variables: order soup, order salad, it's freezing
p cnf 3 3
1 2 0
-1 -2 0
2 3 0
```

This example was given as input to a SAT solver called `sat4j.` The program produced dozens of lines of output reporting on what the program was doing, but the real result is contained in the last three lines:

```
s SATISFIABLE
v -1 2 -3 0
c Total wall clock time (in seconds) : 0.0070
```

This tells us that the formula is satisfiable, and that a satisfying assignment is $p \mapsto$ **false** (don't order the soup), $q \mapsto$ **true** (order the salad), and $r \mapsto$ **false** (it's not freezing).

If we wanted to find a different satisfying assignment, we would have to add a clause saying that the satisfying assignment found in this first run is *not* made. We do this by adding the following line to our specification, and running the program on the new specification.

```
1 -2 3 0
```

The result is

```
s SATISFIABLE
v -1 2 3 0
c Total wall clock time (in seconds) : 0.0090
```

This corresponds to the solution where we don't order the soup, order the salad, and it's freezing outside. If we want to see yet another satisfying assignment, we repeat the process, adding to our specification the negation of the assignment we just found:

```
1 -2 -3 0
```

and get the result

```
s SATISFIABLE
v 1 -2 3 0
c Total wall clock time (in seconds) : 0.0090
```

which corresponds to ordering soup, not ordering the salad, and it being freezing outside.

What if we keep going? We will add one more line to our specification, the negation of this last satisfying assignment:

```
-1 2 -3 0
```

The result is, just as you might expect:

```
s UNSATISFIABLE
c Total wall clock time (in seconds) : 0.01
```

Even in our toy problems of the student schedule and Sudoku, the number of variables and clauses is so large that it is not practical to prepare the text file that encodes the CNF formula by hand. However, it is not hard in either case to generate the CNF encoding with a program, using the original succinct problem descriptions as a guide. We will say more about formalizing such succinct descriptions in Chapter 4.

2.3.5 *How do SAT solvers work?

We already know how to test if a formula is satisfiable: construct the truth table for the formula, and see if there is a **T** in the rightmost column. Consider what this method implies for the Sudoku problem: There are 789 variables, and thus the truth table would have $2^{789} \approx 10^{237}$ rows. There is not enough space in the universe to store such a table.

Of course, we do not have to store the entire table—we can just compute one row at a time, and reuse the same storage for each row. But there are still 2^{789} rows to compute, and if the result is that the puzzle has no solution, we would have to compute *all* of them. The sun will burn out before this calculation can be completed.

One trick that can be used to speed things up is to tentatively assign truth values to variables one at a time. After each assignment is made, we can write a simplified formula involving fewer variables. If, as a result of this process, we wind up with an unsatisfiable formula, we backtrack to the last assignment we made, and make the opposite assignment. Let us illustrate how this works with CNF formula

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

This is the soup-and-salad problem with additional constraints. Let's begin by choosing a variable and assigning it a value. Say we begin with q and give it the value **T**. Our formula now becomes

$$(p \vee \mathbf{T}) \wedge (\neg p \vee \mathbf{F}) \wedge (\mathbf{T} \vee r) \wedge (p \vee \mathbf{F} \vee r) \wedge (p \vee \mathbf{F} \vee \neg r).$$

Now a clause of the form $\mathbf{T} \vee \psi$ always has the value **true**, so we can just eliminate this clause. A clause of the form $\mathbf{F} \vee \psi$ is equivalent to ψ, so we can just erase the **F** from the clause. The resulting simplified formula is

$$\neg p \wedge (p \vee r) \wedge (p \vee \neg r).$$

The literal $\neg p$ all by itself in a clause means that we are forced to assign **F** to p in order to get a satisfying assignment. This gives

$$\mathbf{T} \wedge (\mathbf{F} \vee r) \wedge (\mathbf{F} \vee \neg r),$$

which simplifies to

$$r \wedge \neg r.$$

Of course, this contradiction cannot be satisfied. So we backtrack to the only place where we had a choice to make, when we assigned **T** to q, and now change this to **F**. The result is

$$(p \vee \mathbf{F}) \wedge (\neg p \vee \mathbf{T}) \wedge (\mathbf{F} \vee r) \wedge (p \vee \mathbf{T} \vee r) \wedge (p \vee \mathbf{T} \vee \neg r).$$

This simplifies to

$$p \wedge r \wedge (p \vee r) \wedge (p \vee \neg r).$$

The variables p and r now appear as separate clauses, and this forces the assignment of **T** to both p and r, so we have found our satisfying assignment of **T** to both p and r, and **F** to q.

Even though we started with a formula that has only a single satisfying assignment, we did not have to try out too many possibilities to find it. To speed things up, we used the trick of simplifying the formula at each step, eliminating both clauses and variables. This leads to clauses that have only a single literal, in which the assignment is forced, with the result that there are many possible assignments that we never have to test.

This method, called the DPLL algorithm (for its creators Davis, Putnam, Logemann and Loveland) dates back to the early 1960s, and is at the core of most SAT solvers in wide use today. Much of the speedup that has been achieved in such programs is due to improved strategies for choosing the next unassigned variable to work with, and improved data structures for representing the formulas.

SAT solvers show an interesting gap between theory and practice. In principle, a backtracking solver like this might be forced into testing a large proportion of the 2^n possible assignments to a formula with n variables. For formulas with a few hundred or a few thousand variables, this would throw us right back into the 'not enough time in the universe' dilemma. There is no algorithm that is known to solve *every* possible satisfiability problem more efficiently than this, and there is some theoretical support for the belief that no such algorithm exists. Indeed, one of the most important unsolved problems in computer science and mathematics—the *P vs. NP Conjecture*—is whether one can prove that, in a sense that can be made precise, there are no efficient algorithms that can solve every instance of the satisfiability problem. If, as most experts believe, this conjecture is true, it would seem to rule out the possibility of creating practical SAT solvers that run on problems with thousands of clauses and variables.

Nonetheless, practical SAT solvers that give fast answers to enormous problems really do exist—the hard cases seem to rarely arise in practice. We will have more to say about the question of what computers can and cannot do, and what they can do efficiently, at the end of the book.

2.4 Historical and Bibliographic Notes

Claude E. Shannon, in his 1940 MIT masters thesis, described how to use propositional logic in the analysis and design of electrical switching circuits, essentially inventing the idea of logic gates described in the text. Shannon's gates were implemented as electromechanical relays, but when electronic computers began to appear, vacuum tubes and, later, transistors, replaced them. At around the same time, V. I. Shestakov in the Soviet Union made essentially the same discovery, also in a graduate dissertation. Independently of both Shannon and Shestakov, in 1937, George R. Stibitz, a scientist at Bell Laboratories, built a two-bit binary adder from relays, a 'play project' that was at the origin of a series of successively more sophisticated relay-based computers.

As mentioned in the text, you don't need electricity to implement logic gates. The Web harbors many zany and earnest projects displaying mechanical implementations of logic circuits, of which this two-bit adder built from Legos is a fairly typical example.

The word 'bit' (in the computer science sense!) as well as the notion of a bit as the smallest unit of information, appeared in print for the first time in 1948 in Claude Shannon's most famous work, laying the foundations of Information Theory. [3]

The logic puzzles of Raymond Smullyan used in the text and in the exercises in Section 2.5.2 are from three collections: *What is the Name of This Book?*, *Forever Undecided*, and *The Lady or the Tiger? And Other Logic Puzzles*. The original story of 'The Lady, or the Tiger?', by Frank Stockton, was published in 1882. Notwithstanding the cheesy over-the-top language of the story,

[3]Shannon attributed this brilliant coinage to John Tukey, who must have had a gift for this sort of thing: He also gave us *software*.

the themes of paradox and unsolvable problem that it evokes will be concerns of ours throughout this book.

The DPLL algorithm was originally described by Davis and Putnam, and then improved by Davis, Logemann and Loveland. See Gomes, *et. al.,* for a modern survey on satisfiability solvers. There is an annual conference at which a competition to find the fastest SAT Solver takes place.

2.5 Exercises

2.5.1 Digital Logic

The exercises in this section ask you to design some circuits by wiring together logic gates, along the lines of those displayed in Figures 2.4 and 2.5. If you wish, you can simply draw these by hand, but it is more instructive, and a good deal more fun, to use a *logic simulator* program that allows you to create the circuit diagrams on the computer, test their behavior on various inputs, and save them as modules that can be incorporated into larger designs.

1. Design a circuit with four inputs and a single output that has the value 1 if exactly two of the inputs are 1, and that has the value 0 otherwise. You can use any gate types that you wish.

2. Design a circuit with four inputs and two outputs. The first output should have the value 1 if at least two of the inputs are 1, and the second output should have the value 1 if at least three of the inputs are 1. (Note that there are only three possible output configurations: 00, 10, and 11.)

3. A three-input AND gate gives the output 1 if all inputs are 1 and the output 0 otherwise. It is a simple matter to construct a three-input AND gate from the standard two-input AND gates, because $AND(x, y, z) = AND(AND(x, y), z)$. A three input NAND gate computes the negation of a three-input AND gate: that is, it gives output 0 if all the inputs are 1, and the output 1 otherwise.

   (a) Show that you cannot build a 3-input NAND gate from a pair of 2-input NAND gates in this manner. That is, show that $NAND(x, y, z)$ is not the same as $NAND(NAND(x, y), z)$.

   (b) Build a 3-input NAND gate using only 2-input NAND gates. How many of the 2-input gates did you need?

4. Design a circuit that adds two 2-bit integers and gives the 3-bit sum. Recall from the discussion in the text that the 3-bit sequence $b_2b_1b_0$ represents the integer $4b_2 + 2b_1 + b_0$. So for example, if the four input bits are, in order, $1, 0, 1, 1$, then this will be treated as adding 10 (two) and 11 (three) to give 101 (five). A second version of this problem is given below. Here you are asked to try to solve this problem by creating a propositional formula corresponding to each of the outputs. You can use any gate types that you wish.

5. Another approach to building the 2-bit adder of the preceding problem is to think of it as two one-bit adders chained together: Connect one of the outputs of the first stage to one of the inputs of the second. If you see how this problem works, you should be able to assemble

a k-bit adder for any value of k. (This problem works best if you use a logic simulator that lets you save circuits as modules that can be incorporated into other circuits.)

6. The circuits we have constructed in the text and the preceding exercises contain no loops—that is, there is never a path from the output of a gate back to its input. As a result, the value at the output of any gate in the circuit can be computed from the inputs. Here you will show that this property may fail to hold if we allow such loops, but that other interesting properties may emerge.

   The circuit shown in Figure 2.7 is called a *latch.* In the illustration, the output is 1, but this fact cannot be determined just by looking at the inputs.

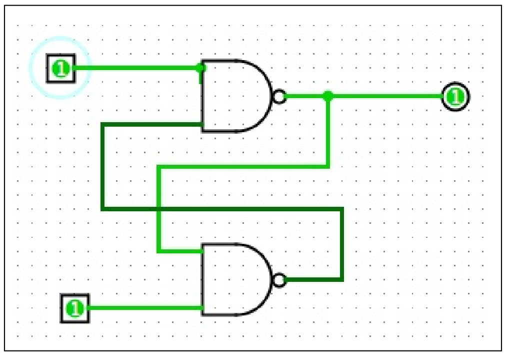


Figure 2.7: A circuit that remembers

   (a) What happens to the output when we switch the *bottom* input from 1 to 0? What happens if we then switch this input back to 1?

   (b) And now, what happens to the output when we switch the *top* input from 1 to 0, and then back again?

   (The latch with both inputs 1 functions as a 1-bit *memory,* recording the which of the two inputs was most recently changed.)

2.5.2 Puzzles

The puzzles in this section, like those in Section 2.2, are all taken from the collections published by Raymond Smullyan. All of these can be approached as illustrated in the examples in the text, by first translating the puzzle into propositional logic.

7. *(More Knights and Knaves)*

   (a) You approach three inhabitants– A, B and C–of the island of Knights and Knaves. A and B tell you:

      A: B and C are both Knights.

      B: A is a Knave and C is a Knight.

Determine, if possible, which group each of the three islanders belongs to.

(b) The setup here is the same as in part *(a).* The statements are:

A: B is a Knave.

B: A and C are of different types.

(c) Surprisingly, Knights and Knaves sometimes intermarry, so you cannot always assume, when you encounter a married couple, that both husband and wife are of the same type.You meet a couple, and the husband says, 'If I am a Knight, then so is my wife.' Can you determine what types this man and his wife belong to?

(d) This is a continuation of part *(c)*: If an islander says 'If I am a Knight, then p', where p is some proposition, what (if anything) can you conclude about the type of the speaker and the truth value of p?

8. *(Lady and Tiger Puzzles)* Generations of American schoolchildren were made to read a 19th century short story called 'The Lady, or the Tiger?'. In it, a rather sadistic ancient ruler administers justice by putting the prisoner in an arena and asking him to choose one of two doors. Behind one door is a woman (whom he is supposed to marry on the spot); behind the other is a hungry tiger, who will devour him instantly.

   Smullyan used the story as the basis for a series of logic puzzles. In the puzzles, each door has either a lady or a tiger behind it, but there is not necessarily one of each: there could be two tigers or two ladies. In addition, there are signs on each door, but the messages on the signs are not necessarily true. Your task in these puzzles is to determine, based on the information given, which door the prisoner should open. (Assume that the prisoner prefers instant marriage to a stranger to being eaten by a tiger!)

   To solve these puzzles, model each sign as a propositional formula in two variables: the first variable is interpreted to mean, 'there is a lady behind the first door', and the second variable, 'there is a lady behind the second door'. Then find an expression for the additional condition in the problem about the truth of the signs, and find a satisfying assignment for the variables.

   (a) The prisoner knows that both signs are true or both are false. First sign: THERE IS A LADY BEHIND AT LEAST ONE OF THESE DOORS. Second sign: THERE IS A TIGER BEHIND THE FIRST DOOR.

   (b) This time the prisoner knows that one of the signs is true, and the other false. First sign: THERE IS A LADY BEHIND THE SECOND DOOR. Second sign: THERE IS A LADY BEHIND ONE OF THESE DOORS AND A TIGER BEHIND THE OTHER.

9.* *The Island of Zombies.* The humans on this island always tell the truth, but the zombies always lie. The words for 'Yes' and 'No' in the language of the islanders are 'Bal' and 'Da', but you do not know which one means 'yes' and which 'no'. In each of these problems you only get to ask one question—you are not, for example, allowed to use the solution of part *(a)* to determine whether 'Bal' means 'yes', to solve the other parts!

   HINT: To get started, let a denote the proposition, 'the islander is a human', let b denote the proposition 'Bal' means 'yes', and let p be an arbitrary proposition. Try to find a formula using these three variables that is equivalent to 'The islander answers "Bal" when asked if p is True'.

(a) You are allowed to ask an islander a single question, which will tell you whether 'Bal' means 'yes' or 'no'. What do you ask?

(b) You and a friend approach an islander. Your friend dares you: I will bet that you cannot make him say 'Bal'. Prove your friend wrong by asking the islander a single question, to which she must answer 'Bal'.

(c) You heard a rumor that there is gold in the hills on the island. You approach an islander and ask him a single question, whose answer will tell you whether or not there is gold in the hills.

2.5.3 Satisfiability Solvers

All the problems in this section, except for the first, are programming problems. To solve them, you should install one of numerous freely available SAT solvers that read input in the DIMACS format (for example `minisat` or `sat4j`).

10. In the example of the DPLL algorithm presented in Section 2.3.5, we began by selecting the variable q and initially setting it to $\mathbf{T}$. The time this algorithm takes to find a satisfying assignment is highly dependent on the order in which we assign truth values to variables. What happens in this example if we choose to work with p first and assign it the value $\mathbf{F}$? What if we choose r first and assign it $\mathbf{T}$?

11. *(a)* Produce a specification file for the CNF formula that encodes the student scheduling problem. There are, of course, too many clauses to create this file by hand, but there are only six types of clauses, and it is not difficult to write a computer program to generate the file.

    To get you started, use the integers 1 through 56 to denote the variables, where variables 1-8 correspond to the 8 successive possible time slots for English, 9-16 for the Math time slots, *etc.* The clause that English has to be scheduled in at least one of the 8 time slots is then

    ```
    1  2  3  4  5  6  7  8  0
    ```

    The 28 clauses that say that English cannot be scheduled in two different time slots begin

    ```
    -1 -2 0
    -1 -3 0
    -1 -4 0
    ```

    and so on up through

    ```
    -7 -8 0
    ```

(b) Now, run the SAT solver on the specification you produced in part *(a)*. What is the schedule that the result corresponds to? How would you use the SAT solver to find a different schedule that satisfies the specifications?

(c) The output produced by the SAT solver for this problem is a bit hard to read, since it gives us the satisfying assignment for *every* variable, where we are really only interested in knowing the variables that are set to **T**. (That is, if `2` appears in the output, then we know that English is scheduled for Monday afternoon, and the output values `-1,-3,-4,...,-8` are simply distracting.) Design a smart back end for this problem that reads the output of the SAT solver and prints the student's schedule in human-readable form.

12. Repeat the preceding problem for Sudoku puzzles. Here you will have to use the integers 1-789 to encode the underlying variables of the problem. The first 3240 clauses of the CNF specification, as described in the text, encode the rules of the puzzle, and the last few lines give the particular puzzle configuration. Write a smart front end that allows you to enter the specification of the puzzle position in a simple intuitive form, and generates the CNF specification file, as well as a smart back end that displays the solution as a square array of integers.

13.* One application of SAT solvers in Artificial Intelligence is the solution of planning problems. We'll illustrate this with classic river-crossing puzzles. Your problem is to model the puzzles as satisfiability problems, and then use the resulting specification as input to a SAT solver. (It is a bit time-consuming to produce the specifications, and the first puzzle is very easy to solve without any special tools. So this should be considered a kind of 'proof-of-concept' problem.)

> A farmer has a fox, a goose, and a sack of grain. He wants to cross a river with these items. His boat is only big enough to hold one of the items along with the farmer. He cannot leave the fox alone with the goose (the fox will eat the goose), and he cannot leave the goose alone with the grain (same reason). Describe how he can get all the items across the river in no more than seven crossings.
>
> The jealous husbands problem: Three married couples want to cross a river, but their boat only holds two people. Each husband is an insanely jealous man, and will not allow his wife to be on one of the river banks with another man unless he himself is present as well. Describe how to get all six people across the river with no more than twelve crossings.

HINT: How can we model these problems as satisfiability problems? The key here is the bound on the number of time steps. Consider the first problem. Number the time steps 0,1,...,7. Introduce variables to encode the propositions:

- the boat is on the left bank at time t
- the fox is on the left bank at time t
- the goose is on the left bank at time t
- the grain is on the left bank at time t

- the fox is placed in the boat at time t
- the goose is placed in the boat at time t
- the grain is placed in the boat at time t

We do not require variables to say that an item or the boat is on the right bank, since these propositions can be encoded by negations of ones described above.

The clauses in the CNF then express the rules of the puzzle: For example, if the fox and the goose are both on the left bank at time t, then the boat must be there at time t; if the goose boards the boat on the left bank at time t then both the boat and the goose will be on the right bank at time $t + 1$, *etc.* These will also include the initial condition (everything on the left bank at time 0) and the goal condition (everything on the right bank at time 7). This can be done with about 200 clauses in all. Like our other puzzles, there are only a few basic kinds of clauses, each repeated a number of times for each time step and each bank of the river.

Here is another take on the first problem.