Chapter 3

Propositional Formulas

It is amazing that people manage to cope with all the ambiguities in the English language. Here are some sentences that illustrate the issue:

1. "You may have cake, or you may have ice cream."

2. "If pigs can fly, then you can understand the Chebyshev bound."

3. "If you can solve any problem we come up with, then you get an A for the course."

4. "Every American has a dream."

What *precisely* do these sentences mean? Can you have both cake and ice cream or must you choose just one dessert? If the second sentence is true, then is the Chebyshev bound incomprehensible? If you can solve some problems we come up with but not all, then do you get an A for the course? And can you still get an A even if you can't solve any of the problems? Does the last sentence imply that all Americans have the same dream or might some of them have different dreams?

Some uncertainty is tolerable in normal conversation. But when we need to formulate ideas precisely —as in mathematics and programming —the ambiguities inherent in everyday language can be a real problem. We can't hope to make an exact argument if we're not sure exactly what the statements mean. So before we start into mathematics, we need to investigate the problem of how to talk about mathematics.

To get around the ambiguity of English, mathematicians have devised a special mini-language for talking about logical relationships. This language mostly uses ordinary English words and phrases such as "or", "implies", and "for all". But mathematicians endow these words with definitions more precise than those found in an ordinary dictionary. Without knowing these definitions, you might sometimes get the gist of statements in this language, but you would regularly get misled about what they really meant.

Surprisingly, in the midst of learning the language of logic, we'll come across the most important open problem in computer science —a problem whose solution could change the world.

3.1 Propositions from Propositions

In English, we can modify, combine, and relate propositions with words such as "not", "and", "or", "implies", and "if-then". For example, we can combine three propositions into one like this:

If all humans are mortal **and** all Greeks are human, **then** all Greeks are mortal.

For the next while, we won't be much concerned with the internals of propositions —whether they involve mathematics or Greek mortality —but rather with how propositions are combined and related. So we'll frequently use variables such as P and Q in place of specific propositions such as "All humans are mortal" and "$2 + 3 = 5$". The understanding is that these variables, like propositions, can take on only the values **T** (true) and **F** (false). Such true/false variables are sometimes called *Boolean variables* after their inventor, George —you guessed it —Boole.

3.1.1 "Not", "And", and "Or"

We can precisely define these special words using *truth tables*. For example, if P denotes an arbitrary proposition, then the truth of the proposition "NOT P" is defined by the following truth table:

P	NOT P
T	**F**
F	**T**

The first row of the table indicates that when proposition P is true, the proposition "NOT P" is false. The second line indicates that when P is false, "NOT P" is true. This is probably what you would expect.

In general, a truth table indicates the true/false value of a proposition for each possible setting of the variables. For example, the truth table for the proposition "P AND Q" has four lines, since the two variables can be set in four different ways:

P	Q	P AND Q
T	**T**	**T**
T	**F**	**F**
F	**T**	**F**
F	**F**	**F**

According to this table, the proposition "P AND Q" is true only when P and Q are both true. This is probably the way you think about the word "and."

There is a subtlety in the truth table for "P OR Q":

P	Q	P OR Q
T	**T**	**T**
T	**F**	**T**
F	**T**	**T**
F	**F**	**F**

The third row of this table says that "P OR Q" is true when even if *both* P and Q are true. This isn't always the intended meaning of "or" in everyday speech, but this is the standard definition in mathematical writing. So if a mathematician says, "You may have cake, or you may have ice cream," he means that you *could* have both.

If you want to exclude the possibility of having both having and eating, you should use "exclusive-or" (XOR):

P	Q	P XOR Q
T	**T**	**F**
T	**F**	**T**
F	**T**	**T**
F	**F**	**F**

3.1.2 "Implies"

The least intuitive connecting word is "implies." Here is its truth table, with the lines labeled so we can refer to them later.

P	Q	P IMPLIES Q	
T	**T**	**T**	(tt)
T	**F**	**F**	(tf)
F	**T**	**T**	(ft)
F	**F**	**T**	(ff)

Let's experiment with this definition. For example, is the following proposition true or false?

"If Goldbach's Conjecture is true, then $x^2 \geq 0$ for every real number x."

Now, we told you before that no one knows whether Goldbach's Conjecture is true or false. But that doesn't prevent you from answering the question! This proposition has the form $P \longrightarrow Q$ where the *hypothesis*, P, is "Goldbach's Conjecture is true" and the *conclusion*, Q, is "$x^2 \geq 0$ for every real number x". Since the conclusion is definitely true, we're on either line (tt) or line (ft) of the truth table. Either way, the proposition as a whole is *true*!

One of our original examples demonstrates an even stranger side of implications.

"If pigs fly, then you can understand the Chebyshev bound."

Don't take this as an insult; we just need to figure out whether this proposition is true or false. Curiously, the answer has *nothing* to do with whether or not you can understand the Chebyshev bound. Pigs do not fly, so we're on either line (ft) or line (ff) of the truth table. In both cases, the proposition is *true*!

In contrast, here's an example of a false implication:

> "If the moon shines white, then the moon is made of white cheddar."

Yes, the moon shines white. But, no, the moon is not made of white cheddar cheese. So we're on line (tf) of the truth table, and the proposition is false.

The truth table for implications can be summarized in words as follows:

> *An implication is true exactly when the if-part is false or the then-part is true.*

This sentence is worth remembering; a large fraction of all mathematical statements are of the if-then form!

3.1.3 "If and Only If"

Mathematicians commonly join propositions in one additional way that doesn't arise in ordinary speech. The proposition "P if and only if Q" asserts that P and Q are logically equivalent; that is, either both are true or both are false.

P	Q	P IFF Q
T	**T**	**T**
T	**F**	**F**
F	**T**	**F**
F	**F**	**T**

The following if-and-only-if statement is true for every real number x:

$$x^2 - 4 \geq 0 \quad \text{iff} \quad |x| \geq 2$$

For some values of x, *both* inequalities are true. For other values of x, *neither* inequality is true . In every case, however, the proposition as a whole is true.

3.1.4 Problems

Class Problems

Problem 3.1.

When the mathematician says to his student, "If a function is not continuous, then it is not differentiable," then letting D stand for "differentiable" and C for continuous, the only proper translation of the mathematician's statement would be

$$\text{NOT}(C) \text{ IMPLIES } \text{NOT}(D),$$

or equivalently,

$$D \text{ IMPLIES } C.$$

But when a mother says to her son, "If you don't do your homework, then you can't watch TV," then letting T stand for "watch TV" and H for "do your homework," a reasonable translation of the mother's statement would be

$$\text{NOT}(H) \ \text{IFF} \ \text{NOT}(T),$$

or equivalently,

$$H \ \text{IFF} \ T.$$

Explain why it is reasonable to translate these two IF-THEN statements in different ways into propositional formulas.

Problem 3.2.
Prove by truth table that OR distributes over AND:

$$[P \ \text{OR} \ (Q \ \text{AND} \ R)] \quad \text{is equivalent to} \quad [(P \ \text{OR} \ Q) \ \text{AND} \ (P \ \text{OR} \ R)] \tag{3.1}$$

Homework Problems

Problem 3.3.
Describe a simple recursive procedure which, given a positive integer argument, n, produces a truth table whose rows are all the assignments of truth values to n propositional variables. For example, for $n = 2$, the table might look like:

T	**T**
T	**F**
F	**T**
F	**F**

Your description can be in English, or a simple program in some familiar language (say Scheme or Java), but if you do write a program, be sure to include some sample output.

3.2 Propositional Logic in Computer Programs

Propositions and logical connectives arise all the time in computer programs. For example, consider the following snippet, which could be either C, C++, or Java:

```
if ( x > 0 || (x <= 0 && y > 100) )
      ⋮
(further instructions)
```

The symbol `||` denotes "or", and the symbol `&&` denotes "and". The *further instructions* are carried out only if the proposition following the word `if` is true. On closer inspection, this big expression is built from two simpler propositions. Let A

be the proposition that `x > 0`, and let B be the proposition that `y > 100`. Then we can rewrite the condition this way:

$$A \text{ or } ((\text{not } A) \text{ and } B) \tag{3.2}$$

A truth table reveals that this complicated expression is logically equivalent to

$$A \text{ or } B. \tag{3.3}$$

A	B	A or ((not A) and B)	A or B
T	**T**	**T**	**T**
T	**F**	**T**	**T**
F	**T**	**T**	**T**
F	**F**	**F**	**F**

This means that we can simplify the code snippet without changing the program's behavior:

```
if ( x > 0 || y > 100 )
    ⋮
(further instructions)
```

The equivalence of (3.2) and (3.3) can also be confirmed reasoning by cases:

A is **T**. Then an expression of the form (A or anything) will have truth value **T**. Since both expressions are of this form, both have the same truth value in this case, namely, **T**.

A is **F**. Then (A or P) will have the same truth value as P for any proposition, P. So (3.3) has the same truth value as B. Similarly, (3.2) has the same truth value as ((not **F**) and B), which also has the same value as B. So in this case, both expressions will have the same truth value, namely, the value of B.

Rewriting a logical expression involving many variables in the simplest form is both difficult and important. Simplifying expressions in software might slightly increase the speed of your program. But, more significantly, chip designers face essentially the same challenge. However, instead of minimizing `&&` and `||` symbols in a program, their job is to minimize the number of analogous physical devices on a chip. The payoff is potentially enormous: a chip with fewer devices is smaller, consumes less power, has a lower defect rate, and is cheaper to manufacture.

3.2.1 Cryptic Notation

Programming languages use symbols like && and ! in place of words like "and" and "not". Mathematicians have devised their own cryptic symbols to represent these words, which are summarized in the table below.

English	**Cryptic Notation**
not P	$\neg P$ (alternatively, $\overline{P}$)
P and Q	$P \wedge Q$
P or Q	$P \vee Q$
P implies Q	$P \longrightarrow Q$
if P then Q	$P \longrightarrow Q$
P iff Q	$P \longleftrightarrow Q$

For example, using this notation, "If P and not Q, then R" would be written:

$$(P \wedge \overline{Q}) \longrightarrow R$$

This symbolic language is helpful for writing complicated logical expressions compactly. But words such as "OR" and "IMPLIES," generally serve just as well as the cryptic symbols $\vee$ and $\longrightarrow$, and their meaning is easy to remember. So we'll use the cryptic notation sparingly, and we advise you to do the same.

3.2.2 Logically Equivalent Implications

Do these two sentences say the same thing?

If I am hungry, then I am grumpy.
If I am not grumpy, then I am not hungry.

We can settle the issue by recasting both sentences in terms of propositional logic. Let P be the proposition "I am hungry", and let Q be "I am grumpy". The first sentence says "P implies Q" and the second says "(not Q) implies (not P)". We can compare these two statements in a truth table:

P	Q	P IMPLIES Q	$\overline{Q}$ IMPLIES $\overline{P}$
T	**T**	**T**	**T**
T	**F**	**F**	**F**
F	**T**	**T**	**T**
F	**F**	**T**	**T**

Sure enough, the columns of truth values under these two statements are the same, which precisely means they are equivalent. In general, "(NOT Q) IMPLIES (NOT P)" is called the *contrapositive* of the implication "P IMPLIES Q." And, as we've just shown, the two are just different ways of saying the same thing.

In contrast, the *converse* of "P IMPLIES Q" is the statement "Q IMPLIES P". In terms of our example, the converse is:

If I am grumpy, then I am hungry.

This sounds like a rather different contention, and a truth table confirms this suspicion:

P	Q	P IMPLIES Q	Q IMPLIES P
T	**T**	**T**	**T**
T	**F**	**F**	**T**
F	**T**	**T**	**F**
F	**F**	**T**	**T**

Thus, an implication *is* logically equivalent to its contrapositive but is *not* equivalent to its converse.

One final relationship: an implication and its converse together are equivalent to an iff statement, specifically, to these two statements together. For example,

If I am grumpy, then I am hungry.
If I am hungry, then I am grumpy.

are equivalent to the single statement:

I am grumpy iff I am hungry.

Once again, we can verify this with a truth table:

P	Q	$(P$ IMPLIES $Q)$ <u>AND</u> $(Q$ IMPLIES $P)$			Q <u>IFF</u> P
T	**T**	**T**	**T**	**T**	**T**
T	**F**	**F**	**F**	**T**	**F**
F	**T**	**T**	**F**	**F**	**F**
F	**F**	**T**	**T**	**T**	**T**

The underlined operators have the same column of truth values, proving that the corresponding formulas are equivalent.

SAT

A proposition is **satisfiable** if some setting of the variables makes the proposition true. For example, P AND $\overline{Q}$ is satisfiable because the expression is true when P is true and Q is false. On the other hand, P AND $\overline{P}$ is not satisfiable because the expression as a whole is false for both settings of P. But determining whether or not a more complicated proposition is satisfiable is not so easy. How about this one?

$$(P \text{ OR } Q \text{ OR } R) \text{ AND } (\overline{P} \text{ OR } \overline{Q}) \text{ AND } (\overline{P} \text{ OR } \overline{R}) \text{ AND } (\overline{R} \text{ OR } \overline{Q})$$

The general problem of deciding whether a proposition is satisfiable is called *SAT*. One approach to SAT is to construct a truth table and check whether or not a **T** ever appears. But this approach is not very efficient; a proposition with n variables has a truth table with 2^n lines, so the effort required to decide about a proposition grows exponentially with the number of variables. For a proposition with just 30 variables, that's already over a billion!

Is there a more *efficient* solution to SAT? In particular, is there some, presumably very ingenious, procedure that determines in a number of steps that grows *polynomially* —like n^2 of n^{14} —instead of exponentially, whether any given proposition is satifiable or not? No one knows. And an awful lot hangs on the answer. An efficient solution to SAT would immediately imply efficient solutions to many, many other important problems involving packing, scheduling, routing, and circuit verification, among other things. This would be wonderful, but there would also be worldwide chaos. Decrypting coded messages would also become an easy task (for most codes). Online financial transactions would be insecure and secret communications could be read by everyone.

Recently there has been exciting progress on *sat-solvers* for practical applications like digital circuit verification. These programs find satisfying assignments with amazing efficiency even for formulas with millions of variables. Unfortunately, it's hard to predict which kind of formulas are amenable to sat-solver methods, and for formulas that are NOT satisfiable, sat-solvers generally take exponential time to verify that.

So no one has a good idea how to solve SAT more efficiently or else to prove that no efficient solution exists —researchers are completely stuck. This is the outstanding unanswered question in theoretical computer science.

3.2.3 Problems

Class Problems

Problem 3.4.
This problem[1] examines whether the following specifications are *satisfiable*:

1. If the file system is not locked, then
   (a) new messages will be queued.
   (b) new messages will be sent to the messages buffer.
   (c) the system is functioning normally, and conversely, if the system is functioning normally, then the file system is not locked.
2. If new messages are not queued, then they will be sent to the messages buffer.
3. New messages will not be sent to the message buffer.

(a) Begin by translating the five specifications into propositional formulas using four propositional variables:

$$\begin{aligned} L &::= \text{file system locked}, \\ Q &::= \text{new messages are queued}, \\ B &::= \text{new messages are sent to the message buffer}, \\ N &::= \text{system functioning normally}. \end{aligned}$$

(b) Demonstrate that this set of specifications is satisfiable by describing a single truth assignment for the variables L, Q, B, N and verifying that under this assignment, all the specifications are true.

(c) Argue that the assignment determined in part (b) is the only one that does the job.

Problem 3.5.
Propositional logic comes up in digital circuit design using the convention that **T** corresponds to 1 and **F** to 0. A simple example is a 2-bit *half-adder* circuit. This circuit has 3 binary inputs, a_1, a_0 and b, and 3 binary outputs, c, o_1, o_0. The 2-bit word $a_1 a_0$ gives the binary representation of an integer, k, between 0 and 3. The 3-bit word $c s_1 s_0$ gives the binary representation of $k + b$. The third output bit, c, is called the final *carry bit*.

So if k and b were both 1, then the value of $a_1 a_0$ would be `01` and the value of the output $c s_1 s_0$ would `010`, namely, the 3-bit binary representation of $1 + 1$.

[1] From Rosen, 5th edition, Exercise 1.1.36

In fact, the final carry bit equals 1 only when all three binary inputs are 1, that is, when $k = 3$ and $b = 1$. In that case, the value of cs_1s_0 is `100`, namely, the binary representation of $3 + 1$.

This 2-bit half-adder could be described by the following formulas:

$$
\begin{aligned}
c_0 &= b \\
s_0 &= a_0 \text{ XOR } c_0 \\
c_1 &= a_0 \text{ AND } c_0 && \text{the carry into column 1} \\
s_1 &= a_1 \text{ XOR } c_1 \\
c_2 &= a_1 \text{ AND } c_1 && \text{the carry into column 2} \\
c &= c_2.
\end{aligned}
$$

(a) Generalize the above construction of a 2-bit half-adder to an $n + 1$ bit half-adder with inputs $a_n, \ldots, a_1, a_0$ and b for arbitrary $n \geq 0$. That is, give simple formulas for s_i and c_i for $0 \leq i \leq n + 1$, where c_i is the carry into column i and $c = c_{n+1}$.

(b) Write similar definitions for the digits and carries in the sum of two $n + 1$-bit binary numbers $a_n \ldots a_1a_0$ and $b_n \ldots b_1b_0$.

Visualized as digital circuits, the above adders consist of a sequence of single-digit half-adders or adders strung together in series. These circuits mimic ordinary pencil-and-paper addition, where a carry into a column is calculated directly from the carry into the previous column, and the carries have to ripple across all the columns before the carry into the final column is determined. Circuits with this design are called *ripple-carry* adders. Ripple-carry adders are easy to understand and remember and require a nearly minimal number of operations. But the higher-order output bits and the final carry take time proportional to n to reach their final values.

(c) How many of each of the propositional operations does your adder from part (b) use to calculate the sum?

Problem 3.6. **(a)** A propositional formula is *valid* iff it is equivalent to **T**. Verify by truth table that

$$(P \text{ IMPLIES } Q) \text{ OR } (Q \text{ IMPLIES } P)$$

is valid.

(b) Let P and Q be propositional formulas. Describe a single propositional formula, R, involving P and Q such that R is valid iff P and Q are equivalent.

(c) A propositional formula is *satisfiable* iff there is an assignment of truth values to its variables —an *environment* —which makes it true. Explain why

P is valid iff $\text{NOT}(P)$ is *not* satisfiable.

(d) A set of propositional formulas $P_1, \ldots, P_k$ is *consistent* iff there is an environment in which they are all true. Write a formula, S, so that the set $P_1, \ldots, P_k$ is *not* consistent iff S is valid.

Homework Problems

Problem 3.7.
Considerably faster adder circuits work by computing the values in later columns for both a carry of 0 and a carry of 1, *in parallel*. Then, when the carry from the earlier columns finally arrives, the pre-computed answer can be quickly selected. We'll illustrate this idea by working out the equations for an $n+1$-bit parallel half-adder.

Parallel half-adders are built out of parallel "add1" modules. An $n+1$-bit add1 module takes as input the $n+1$-bit binary representation, $a_n \ldots a_1 a_0$, of an integer, s, and produces as output the binary representation, $c\, p_n \ldots p_1\, p_0$, of $s+1$.

(a) A 1-bit add1 module just has input a_0. Write propositional formulas for its outputs c and p_0.

(b) Explain how to build an $n+1$-bit parallel half-adder from an $n+1$-bit add1 module by writing a propositional formula for the half-adder output, o_i, using only the variables a_i, p_i, and b.

We can build a double-size add1 module with $2(n+1)$ inputs using two single-size add1 modules with $n+1$ inputs. Suppose the inputs of the double-size module are $a_{2n+1}, \ldots, a_1, a_0$ and the outputs are $c, p_{2n+1}, \ldots, p_1, p_0$. The setup is illustrated in Figure 3.1.

Namely, the first single size add1 module handles the first $n+1$ inputs. The inputs to this module are the low-order $n+1$ input bits $a_n, \ldots, a_1, a_0$, and its outputs will serve as the first $n+1$ outputs $p_n, \ldots, p_1, p_0$ of the double-size module. Let $c_{(1)}$ be the remaining carry output from this module.

The inputs to the second single-size module are the higher-order $n+1$ input bits $a_{2n+1}, \ldots, a_{n+2}, a_{n+1}$. Call its first $n+1$ outputs $r_n, \ldots, r_1, r_0$ and let $c_{(2)}$ be its carry.

(c) Write a formula for the carry, c, in terms of $c_{(1)}$ and $c_{(2)}$.

(d) Complete the specification of the double-size module by writing propositional formulas for the remaining outputs, p_i, for $n+1 \leq i \leq 2n+1$. The formula for p_i should only involve the variables a_i, $r_{i-(n+1)}$, and $c_{(1)}$.

(e) Parallel half-adders are exponentially faster than ripple-carry half-adders. Confirm this by determining the largest number of propositional operations required to compute any one output bit of an n-bit add module. (You may assume n is a power of 2.)

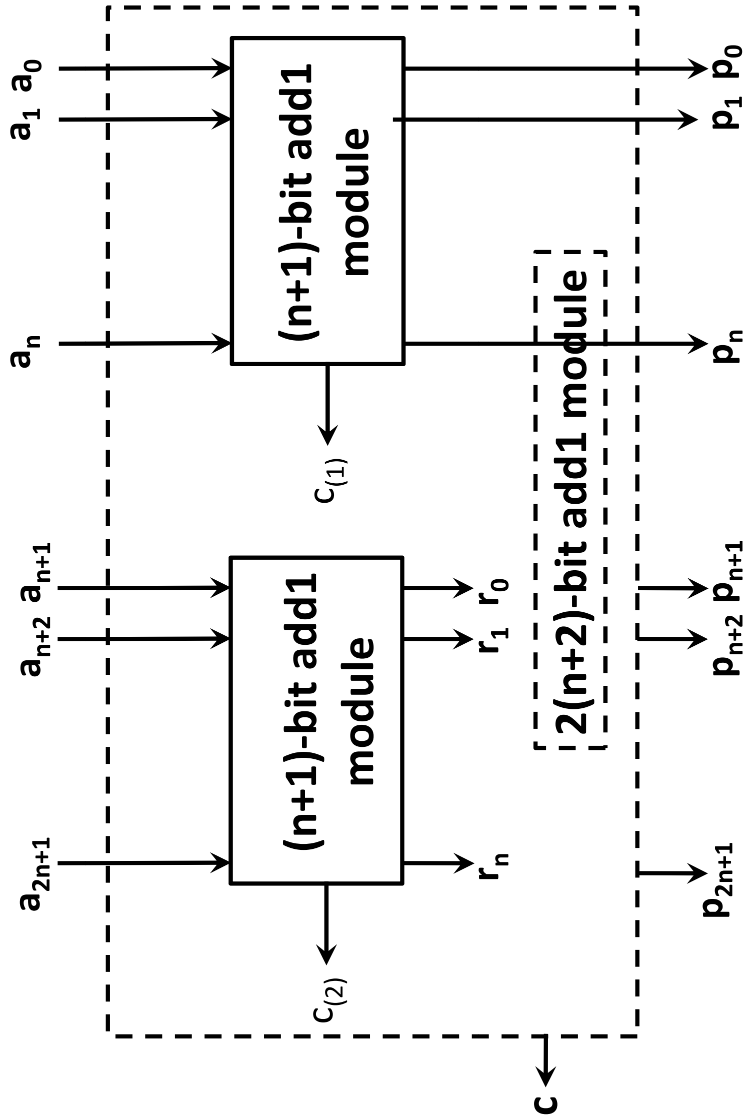

Figure 3.1: Structure of a Double-size Add1 Module.

MIT OpenCourseWare
http://ocw.mit.edu

6.042J / 18.062J Mathematics for Computer Science
Spring 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.