Part I

Numbers

Chapter 2

0 and 1

’0 and 1’ may seem like an uninteresting title for this first proper chapter, but most readers probably know that at the most fundamental level computers always deal with 0s and 1s. Here we will first learn about some of the advantages of this, and then consider some of the mathematics of 0 and 1.

2.1 Robust communication

Suppose you are standing at one side of a river and a friend is standing at the other side, 500 meters away; how can you best communicate with your friend in this kind of situation, assuming you have no aids at your disposal? One possibility would be to try and draw the letters of the alphabet in the air, but at this distance it would be impossible to differentiate between the different letters as long as you only draw with your hands. What is needed is a more robust way to communicate where you are not dependent on being able to decipher so many different symbols. As far as robustness is concerned, the best would be to only use two symbols, say ’horizontal arm’ and ’vertical arm’ or ’h’ and ’v’ for short. You can then represent the different letters in terms of these symbols. We could for example use the coding shown in table 2.1 which is built up in a way that will become evident in chapter 3. You would obviously have to agree on your coding system in advance.

The advantage of using only two symbols is of course that there is little danger of misunderstanding as long as you remember the coding. You only have to differentiate between two arm positions, so you have generous error margins for how you actually hold your arm. The disadvantage is that some of the letters require quite long codes. In fact, the letter ’s’ which is among the most frequently used in English, requires a code with five arm symbols, while the two letters ’a’

a	h	**j**	vhhv	**s**	vhhvh
b	v	**k**	vhvh	**t**	vhhvv
c	vh	**l**	vhvv	**u**	vhvhh
d	vv	**m**	vvhh	**v**	vhvhv
e	vhh	**n**	vvhv	**w**	vhvvh
f	vhv	**o**	vvvh	**x**	vhvvv
g	vvh	**p**	vvvv	**y**	vvhhh
h	vvv	**q**	vhhhh	**z**	vvhhv
i	vhhh	**r**	vhhhv		

Table 2.1. Representation of letters in terms of 'horizontal arm' ('h') and 'vertical arm' ('v').

and 'b' which are less common both require one symbol each. If you were to make heavy use of this coding system it would therefore make sense to reorder the letters such that the most frequent ones (in your language) have the shortest codes.

2.2 Why 0 and 1 in computers?

The above example of human communication across a river illustrates why it is not such a bad idea to let computers operate with only two distinct symbols which we may call '0' and '1' just as well as 'h' and 'v'. A computer is built to manipulate various kinds of information and this information must be moved between the different parts of the computer during the processing. By representing the information in terms of 0s and 1s, we have the same advantages as in communication across the river, namely robustness and the simplicity of having just two symbols.

In a computer, the 0s and 1s are represented by voltages, magnetic charges, light or other physical quantities. For example 0 may be represented by a voltage in the interval 1.5 V to 3 V and 1 by a voltage in the interval 5 V to 6.5 V. The robustness is reflected in the fact that there is no need to measure the voltage accurately, we just need to be able to decide whether it lies in one of the two intervals. This is also a big advantage when information is stored on an external medium like a DVD or hard disk, since we only need to be able to store 0s and 1s. A '0' may be stored as 'no reflection' and a '1' as 'reflection', and when light is shone on the appropriate area we just need to detect whether the light is reflected or not.

A disadvantage of representing information in terms of 0s and 1s is that we may need a large amount of such symbols to encode the information we are interested in. If we go back to table 2.1, we see that the 'word' hello requires 18 symbols ('h's and 'v's), and in addition we have to also keep track of the boundaries between the different letters. The cost of using just a few symbols is therefore that we must be prepared to process large numbers of them.

Although representation of information in terms of 0s and 1s is very robust, it is not foolproof. Small errors in for example a voltage that represents a 0 or 1 do not matter, but as the voltage becomes more and more polluted by noise, its value will eventually go outside the permitted interval. It will then be impossible to tell which symbol the value was meant to represent. This means that increasing noise levels will not be noticed at first, but eventually the noise will break the threshold which will make it impossible to recognise the symbol and therefore the information represented by the symbol.

When we think of the wide variety of information that can be handled by computers, it may seem quite unbelievable that it is all comprised of 0s and 1s. In chapter 1 we saw that information commonly processed by computers can be represented by numbers, and in the next chapter we shall see that all numbers may be expressed in terms of 0s and 1s. The conclusion is therefore that a wide variety of information can be represented in terms of 0s and 1s.

Observation 2.1 (0 and 1 in computers)**.** *In a computer, all information is usually represented in terms of two symbols, '0' and '1'. This has the advantage that the representation is robust with respect to noise, and the electronics necessary to process one symbol is simple. The disadvantage is that the code needed to represent a piece of information becomes longer than what would be the case if more symbols were used.*

Whether we call the two possible values 0 and 1, 'v' and 'h' or 'yes' and 'no' does not matter. What is important is that there are only two symbols, and what these symbols are called usually depends on the context. An important area of mathematics that depends on only two values is logic.

2.3 True and False

In everyday speech we make all kinds of statements and some of them are objective and precise enough that we can check whether or not they are true. Most people would for example agree that the statements 'Oslo is the capital of Norway' and 'Red is a colour' are true, while there is less agreement about the statement 'Norway is a beautiful country'. In normal speech we also routinely link such logical statements together with words like 'and' and 'or', and we negate a statement with 'not'.

Mathematics is built by strict logical statements that are either true or false. Certain statements which are called axioms, are just taken for granted and form the foundation of mathematics (something cannot be created from nothing). Mathematical proofs use logical operations like 'and', 'or', and 'not' to combine existing statements and obtain new ones that are again either true or false. For example we can combine the two true statements 'π is greater than 3' and 'π is smaller than 4' with 'and' and obtain the statement 'π is greater than 3 and π is smaller than 4' which we would usually state as 'π lies between 3 and 4'. Likewise the statement 'π is greater than 3' can be negated to the opposite statement 'π is less than or equal to 3' which is false.

Even though this description is true, doing mathematics is much more fun than it sounds. Not all new statements are interesting even though they may be true. To arrive at interesting new truths we use intuition, computation and any other aids at our disposal. When we feel quite certain that we have arrived at an interesting statement comes the job of constructing the formal proof, i.e., combining known truths in such a way that we arrive at the new statement. If this sounds vague you should get a good understanding of this process as you work your way through any university maths course.

2.3.1 Logical variables and logical operators

When we introduced the syntax for algorithms in section 1.4, we noted the possibility of confusion between assignment and test for equality. This distinction is going to be important in what follows since we are going to discuss logical expressions which may involve tests for equality.

In this section we are going to introduce the standard logical operators in more detail, and to do this, logical variables will be useful. From elementary mathematics we are familiar with using x and y as symbols that typically denote real numbers. Logical variables are similar except that they may only take the values 'true' or 'false' which we now denote by $\mathtt{T}$ and $\mathtt{F}$. So if p is a logical variable, it may denote any logical statement. As an example, we may set

$$p = (4 > 3)$$

which is the same as setting $p = \mathtt{T}$. More interestingly, if a is any real number we may set

$$p = (a > 3).$$

The value of p will then be either $\mathtt{T}$ or $\mathtt{F}$, depending on the value of a so we may think of $p = p(a)$ as a function of a. We then clearly have $p(2) = \mathtt{F}$ and $p(4) = \mathtt{T}$. All the usual relational operators like $<$, $>$, $\leq$ and $\geq$ can be used in this way.

The function

$$p(a) = (a == 2)$$

has the value `T` if a is 2 and the value `F` otherwise. Without the special notation for comparison this would become $p(a) = (a = b)$ which certainly looks rather confusing.

Definition 2.2. *In the context of logic, the values true and false are denoted* `T` *and* `F`*, and assignment is denoted by the operator* $=$*. A logical statement is an expression that is either* `T` *or* `F` *and a logical function* $p(a)$ *is a function that is either* `T` *or* `F`*, depending on the value of* a*.*

Suppose now that we have two logical variables p and q. We have already mentioned that these can be combined with the operators 'and', 'or' and 'not' for which we now introduce the notation $\wedge$, $\vee$ and $\neg$. Let us consider each of these in turn.

The expression $\neg p$ is the opposite of p, i.e., it is `T` if p is `F` and `F` if p is `T`, see column three in table 2.2. The only way for $p \wedge q$ to be `T`, is for both p and q to be `T`; in all other cases it is `F`, see column four in the table. Logical or is the opposite: The expression $p \vee q$ is only `F` if both p and q are `F`; otherwise it is `T`; see column five in table 2.2.

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

Table 2.2. Behaviour of the logical operators $\neg$ (not), $\wedge$ (and), $\vee$ (or), and $\oplus$ (exclusive or).

This use of 'not' and 'and' is just like in everyday language. The definition of 'or', however, does not always agree with how it is used in speech. Suppose someone says 'The apple was red or green', is it then possible that the apple was both red and green? Many would probably say no, but to be more explicit we would often say 'The apple was either red or green (but not both)'.

This example shows that there are in fact two kinds of 'or', an inclusive or ($\vee$) which is `T` when p and q are both `T`, and an exclusive or ($\oplus$) which is `F` when both p and q are `T`, see columns five and six of Table 2.2.

Definition 2.3. *The logical operators 'not', 'and', 'or', and 'exclusive or' are denoted by the symbols $\neg$, $\wedge$, $\vee$, and $\oplus$, respectively and are defined in table 2.2.*

So far we have only considered expressions that involve two logical variables. If p, q, r and s are all logical variables, it is quite natural to consider longer expressions like

$$(p \wedge q) \wedge r, \tag{2.1}$$
$$(p \vee q) \vee (r \vee s), \tag{2.2}$$

(we will consider mixed expressions later). The brackets have been inserted to indicate the order in which the expressions are to be evaluated since we only know how to combine two logical variables at a time. However, it is quite easy to see that both expressions remain the same regardless of how we insert brackets. The expression in (2.1) is `T` only when all of p, q and r are `T`, while the expression in (2.2) is always true except in the case when all the variables are `F`. This means that we can in fact remove the brackets and simply write

$$p \wedge q \wedge r,$$
$$p \vee q \vee r \vee s,$$

without any risk of misunderstanding since it does not matter in which order we evaluate the expressions.

Many other mathematical operations, like for example addition and multiplication of numbers, also have this property, and it therefore has its own name; we say that the operators $\wedge$ and $\vee$ are *associative*. The associativity also holds when we have longer expressions: If the operators are either all $\wedge$ or all $\vee$, the result is independent of the order in which we apply the operators.

What about the third operator, $\oplus$ (exculsive or), is this also associative? If we consider the two expressions

$$(p \oplus q) \oplus r, \qquad p \oplus (q \oplus r),$$

the question is whether they are always equal. If we check all the combinations and write down a truth table similar to Table 2.2, we do find that the two expressions are the same so the $\oplus$ operator is also associative. A general description of such expressions is a bit more complicated than for $\wedge$ and $\vee$. It turns out that if we have a long sequence of logical variables linked together with $\oplus$, then the result is true if the number of variables that is `T` is an odd number and `F` otherwise.

The logical operator $\wedge$ has the important property that $p \wedge q = q \wedge p$ and the same is true for $\vee$ and $\oplus$. This is also a property of addition and multiplication of numbers and is usually referred to as commutativity.

For easy reference we sum all of this up in a theorem.

Proposition 2.4. *The logical operators $\wedge$, $\vee$ and $\oplus$ defined in Table 2.2 are all commutative and associative, i.e.,*

$$\begin{aligned} p \wedge q = q \wedge p, &\quad (p \wedge q) \wedge r = p \wedge (q \wedge r), \\ p \vee q = q \vee p, &\quad (p \vee q) \vee r = p \vee (q \vee r), \\ p \oplus q = q \oplus p, &\quad (p \oplus q) \oplus r = p \oplus (q \oplus r). \end{aligned}$$

where p, q and r are logical variables.

2.3.2 Combinations of logical operators

The logical expressions we have considered so far only involve one logical operator at a time, but in many situations we need to evaluate more complex expressions that involve several logical operators. Two commonly occurring expressions are $\neg(p \wedge q)$ and $\neg(p \vee q)$. These can be expanded by *De Morgan's laws* which are easily proved by considering truth tables for the two sides of the equations.

Lemma 2.5 (De Morgan's laws)**.** *Let p and q be logical variables. Then*

$$\begin{aligned} \neg(p \wedge q) &= (\neg p) \vee (\neg q), \\ \neg(p \vee q) &= (\neg p) \wedge (\neg q). \end{aligned}$$

De Morgan's laws can be generalised to expressions with more than two operators, for example

$$\neg(p \wedge q \wedge r \wedge s) = (\neg p) \vee (\neg q) \vee (\neg r) \vee (\neg s),$$

see exercise 3.

We are going to consider two more laws of logic, namely the two distributive laws.

Theorem 2.6 (Distributive laws). *If p, q and r are logical variables, then*

$$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r),$$
$$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r).$$

As usual, these rules can be proved setting up truth tables for the two sides.

Exercises

1 Use a truth table to prove that the exclusive or operator $\oplus$ is associative, i.e., show that if p, q and r are logical operators then $(p \oplus q) \oplus q = p \oplus (q \oplus r)$.

2 Prove de Morgan's laws.

3 Generalise De Morgan's laws to expressions with any finite number of $\wedge$- or $\vee$-operators, i.e., expressions on the form

$$\neg(p_1 \wedge p_2 \wedge \cdots \wedge p_n) \quad \text{and} \quad \neg(p_1 \vee p_2 \vee \cdots \vee p_n).$$

Hint: Use Lemma 2.5.

4 Use truth tables to check that

a) $(p \wedge q) \vee r = p \wedge (q \vee r)$.

b) $(p \vee q) \wedge (q \vee r) = (p \wedge r) \vee q$.