3

Computers and thought

So far, I have tried to explain the philosophical problem of the nature of representation, and how it is linked with our understanding of other minds. What people say and do is caused by what they think – what they believe, hope, wish, desire and so on – that is, by their representational states of mind or *thoughts*. What people do is caused by the ways they represent the world to be. If we are going to explain thought, then we have to explain how there can be states which can at the same time be representations of the world and causes of behaviour.

To understand how anything can have these two features it is useful to introduce the idea of the mind as a computer. Many psychologists and philosophers think that the mind is a kind of computer. There are many reasons why they think this, but the link with our present theme is this: a computer is a causal mechanism which contains representations. In this chapter and the next I shall explain this idea, and show its bearing on the problems surrounding thought and representation.

The very idea that the mind is a computer, or that computers might think, inspires strong feelings. Some people find it exciting, others find it preposterous, or even degrading to human nature. I will try and address this controversial issue in as fair-minded a way as possible, by assessing some of the main arguments for and against the claims that computers can think, and that the mind is a computer. But first we need to understand these claims.

Asking the right questions

It is crucial to begin by asking the right questions. For example, sometimes the question is posed as: can the human mind be modelled on a computer? But, even if the answer to this question is

'Yes', how could that show that the mind is a computer? The British Treasury produces computer models of the economy – but no-one thinks that this shows that the economy is a computer. This chapter will explain how this confusion can arise. One of this chapter's main aims is to distinguish between two questions:

1 Can a computer think? Or, more precisely, can anything think simply by being a computer?
2 Is the human mind a computer? Or, more precisely, are any actual mental states and processes computational?

This chapter will be concerned mainly with question 1, and Chapter 4 with question 2. The distinction between the two questions may not be clear yet, but, by the end of the chapter, it should be. To understand these two questions, we need to know at least two things: first, what a computer is; and, second, what it is about the mind that leads people to think that a computer could have a mind, or that the human mind could be a computer.

What is a computer? We are all familiar with computers – many of us use them every day. To many they are a mystery, and explaining how they work might seem a very difficult task. However, though the details of modern computers are amazingly complex, the basic concepts behind them are actually beautifully simple. The difficulty in understanding computers is not so much in grasping the concepts involved, but in seeing *why* these concepts are so useful.

If you are familiar with the basic concepts of computers, you may wish to skip the next five sections, and move directly to the section of this chapter called 'Thinking computers?' on p. 109. If you are not familiar with these concepts, then some of the terminology that follows may be a little daunting. You may want to read through the next few sections quite quickly, and the point of them will become clearer after you have then read the rest of this chapter and Chapter 4.

To prepare yourself for understanding computers, it's best to abandon most of the presuppositions that you may have about them. The personal computers we use in our everyday lives normally

have a typewriter-style keyboard and a screen. Computers are usually made out a combination of metal and plastic, and most of us know that they have things inside them called 'silicon chips', which somehow make them work. Put all these ideas to one side for the moment – none of these features of computers is essential to them. It's not even essential to computers that they are electronic.

So what is essential to a computer? The rough definition I will eventually arrive at is: *a computer is a device which processes representations in a systematic way*. This is a little vague until we understand 'processes', 'representations' and 'systematic' more precisely. In order to understand these ideas, there are two further ideas that we need to understand. The first is the rather abstract mathematical idea of a *computation*. The second is how computations can be *automated*. I shall take these ideas in turn.

Computation, functions and algorithms

The first idea we need is the idea of a mathematical *function*. We are all familiar with this idea from elementary arithmetic. Some of the first things we learn in school are the basic arithmetical functions: addition, subtraction, multiplication and division. We then normally learn about other functions such as the square function (by which we produce the square of a number, x^2, by multiplying the number, x, by itself), logarithms and so on.

As we learn them at school, arithmetical functions are not numbers, but things that are 'done' to numbers. What we learn to do in basic arithmetic is to take some numbers and apply certain functions to them. Take the addition of two numbers, 7 and 5. In effect, we take these two numbers as the 'input' to the addition function and get another number, 12, as the 'output'. This addition sum we represent by writing: 7 + 5 = 12. Of course, we can put any two numbers in the places occupied by 7 and 5 (the input places) and the addition function will determine a unique number as the output. It takes training to figure out what the output will be for any number whatsoever – but the point is that, according to the addition function, there is exactly one number that is the output of the function for any given group of input numbers.

If we take the calculation 7 + 5 = 12, and remove the numerals 7, 5 and 12 from it, we get a complex symbol with three 'gaps' in it: _ + _ = _. In the first two gaps, we write the inputs to the addition function, and in the third gap we write the output. The function itself could then be represented as _ + _, with the two blanks indicating where the input numbers should be entered. These blanks are standardly indicated by italic letters, *x*, *y*, *z* and so on – so the function would therefore be written *x* + *y*. These letters, called 'variables' are a useful way of marking the different gaps or *places* of the function.

Now for some terminology. The inputs to the function are called the *arguments* of the function, and the output is called the *value* of the function. The arguments in the equation *x* + *y* = *z* are pairs of numbers *x* and *y* such that *z* is their value. That is, the value of the addition function is the sum of the arguments of that function. The value of the subtraction function is the result of subtracting one number from another (the arguments). And so on.

Though the mathematical theory of functions is very complex in its details, the basic idea of a function can be explained using simple examples such as addition. And, though I introduced it with a mathematical example, the notion of a function is extremely general and can be extended to things other than numbers. For example, because everyone has only one natural father, we can think of the expression 'the natural father of *x*' as describing a function, which takes people as its arguments and gives you their fathers as values. (Those familiar with elementary logic will also know that expressions such as 'and' and 'or' are known as *truth*-functions, e.g. the complex proposition P&Q involves a function that yields the value True when both its arguments are true, and the value False otherwise.)

The idea of a function, then, is a very general one, and one that we implicitly rely on in our everyday life (every time we add up the prices of something in a supermarket, for example). But it is one thing to say what a function is, in the abstract, and another to say how we use them. To know how to employ a function, we need a method for getting the value of the function for a given argument

or arguments. Remember what happens when you learn elementary arithmetic. Suppose you want to calculate the product of two numbers, 127 and 21. The standard way of calculating this is the method of long multiplication:

$$\begin{array}{r} 127 \\ \times 21 \\ \hline 127 \\ +2540 \\ \hline 2667 \end{array}$$

What you are doing when you perform long multiplication is so obvious that it would be banal to spell it out. But, in fact, what you know when you know how to do this is something incredibly powerful. What you have is a method for calculating the product of *any* two numbers – that is, of calculating the value of the multiplication function for any two arguments. This method is entirely general: it does not apply to some numbers and not to others. And it is entirely unambiguous: if you know the method, you know at every stage what to do next to produce the answer.

(Compare a method like this with the methods we use for getting on with people we have met for the first time. We have certain rough-and-ready rules we apply: perhaps we introduce ourselves, smile, shake hands, ask them about themselves, etc. But obviously these methods do not yield definite 'answers'; sometimes our social niceties backfire.)

A method, such as long multiplication, for calculating the value of a function is known as an *algorithm*. Algorithms are also called 'effective procedures' as they are procedures which, if applied correctly, are entirely effective in bringing about their results (unlike the procedures we use for getting on with people). They are also called 'mechanical procedures', but I would rather not use this term, as in this book I am using the term 'mechanical' in a less precise sense.

It is very important to distinguish between algorithms and functions. An algorithm is a *method* for finding the *value* of a function.

A function may have more than one algorithm for finding its values for any given arguments. For example, we multiplied 127 by 21 by using the method of long multiplication. But we could have multiplied it by adding 127 to itself 20 times. That is, we could have used a different algorithm.

To say that there is an algorithm for a certain arithmetical function is not to say that an application of the algorithm will always give you a *number* as an answer. For example, you may want to see whether a certain number divides *exactly* into another number without remainder. When you apply your algorithm for division, you may find out that it doesn't. So, the point is not that the algorithm gives you a number as an answer, but that it always gives you a procedure for finding out whether there is an answer.

When there is an algorithm that gives the value of a function for any argument, then mathematicians say that the function is *computable*. The mathematical theory of computation is, in its most general terms, the theory of computable functions, i.e. functions for which there are algorithms.

Like the notion of a function, the notion of an algorithm is extremely general. Any effective procedure for finding the solution to a problem can be called an algorithm, so long as it satisfies the following conditions:

1 At each stage of the procedure, there is a definite thing to do next. Moving from step to step does not require any special guesswork, insight or inspiration.
2 The procedure can be specified in a finite number of steps.

So we can think of an algorithm as a rule, or a bunch of rules, for giving the solution to a given problem. These rules can then be represented as a 'flow chart'. Consider, for example, a very simple algorithm for multiplying two whole numbers, x and y, which works by adding y to itself. It will help if you imagine the procedure being performed on three pieces of paper, one for the first number (call this piece of paper X), one for the second number (call this piece of paper Y) and one for the answer (call this piece of paper the ANSWER).

Figure 3.1 shows the flow chart; it represents the calculation by the following series of steps:

Step (i): Write '0' on the ANSWER, and go to step (ii).
Step (ii): Does the number written on $X = 0$?
If YES, then go to step (v)
If NO, then go to step (iii)
Step (iii): Subtract 1 from the number written on *X*, write the result on *X*, and go to step (iv)
Step (iv): Add the number written on *Y* to the ANSWER, and go to step (ii)
Step (v): STOP

Let's apply this to a particular calculation, say 4 times 5. (If you are familiar with this sort of procedure, you can skip this example and move on to the next paragraph.)

Begin by writing the numbers to be multiplied, 4 and 5, on the X and Y pieces of paper respectively. Apply step (i) and write 0 on the ANSWER. Then apply step (ii) and ask whether the number written on *X* is 0. It isn't – it's 4. So move to step (iii), and subtract 1 from the number written on X. This leaves you with 3, so you

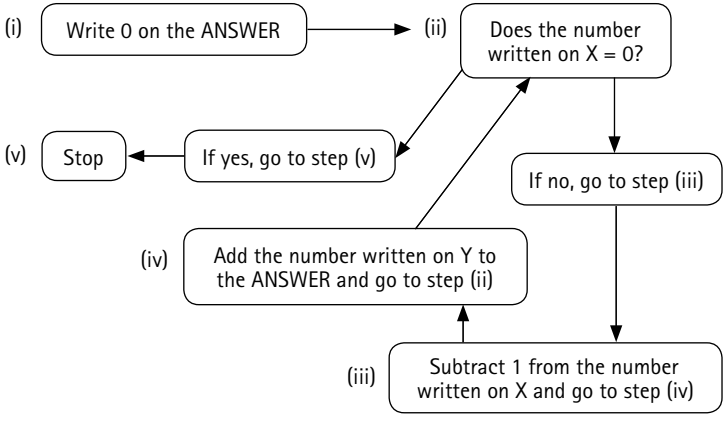

Figure 3.1 Flow chart for the multiplication algorithm.

should write this down on X, and move to step (iv). Add the number written on Y (i.e. 5) to the ANSWER, which makes the ANSWER read 5. Move to step (ii), and ask again whether the number on *X* is 0. It isn't – it's 3. So move to step (iii), subtract 1 from the number written on X, write down 2 on X and move to step (iv). Add the number written on Y to the ANSWER, which makes the ANSWER read 10. Ask again whether the number written on X is 0. It isn't – it's 2. So move to step (iii), subtract 1 from the number written on *X*, write down 1 on *X* and move to step (iv). Add the number written on Y to the ANSWER, which makes the ANSWER read 15. Ask again whether the number written on X is 0; it isn't, it's 1. So move to step (iii), subtract 1 from the number written on X, write down 0 on X and move to step (iv). Add the number written on *Y* to the ANSWER, which makes the ANSWER read 20. Move to step (ii) and ask whether the number written on *X* is 0. This time it is, so move to step (v), and stop the procedure. The number written on the ANSWER is 20, which is the result of multiplying 4 by 5.[1]

This is a pretty laborious way of multiplying 4 by 5. But the point of the illustration is not that this is a *good* procedure for us to use. The point is rather that it is an entirely *effective* procedure: at each stage, it is completely clear what to do next, and the procedure terminates in a finite number of steps. The number of steps could be very large; but for any pair of finite numbers, this will still be a finite number of steps.

Steps (iii) and (iv) of the example illustrate an important feature of algorithms. In applying this algorithm for multiplication, we employ other arithmetical operations: subtraction in step (iii), addition in step (iv). There is nothing wrong with doing this, so long as there are algorithms for the operations of subtraction and addition too – which of course there are. In fact, most algorithms will use other algorithms at some stage. Think of long multiplication: it uses addition to add up the results of the 'short' multiplications. Therefore, you will use some algorithm for addition when doing long multiplication. So our laborious multiplication algorithm can be broken down into steps which depend only on other (perhaps simpler) algorithms and simple 'movements' from step to step. This idea is very important in understanding computers, as we shall see.

The fact that algorithms can be represented by flow charts indicates the generality of the concept of an algorithm. As we can write flow charts for all sorts of procedures, so we can write algorithms for all sorts of things. Certain recipes, for example, can be represented as flow charts. Consider this algorithm for boiling an egg.

1 Turn on the stove
2 Fill the pan with water
3 Place the pan on the stove
4 When the water boils, add one egg, and set the timer
5 When the timer rings, turn off the gas
6 Remove the egg from the water
7 Result: one boiled egg.

This is a process that can be completed in a finite number of steps, and at each step there is a definite, unambiguous, thing to do next. No inspiration or guesswork is required. So, in a sense, boiling an egg can be described as an algorithmic procedure (see Figure 3.2).

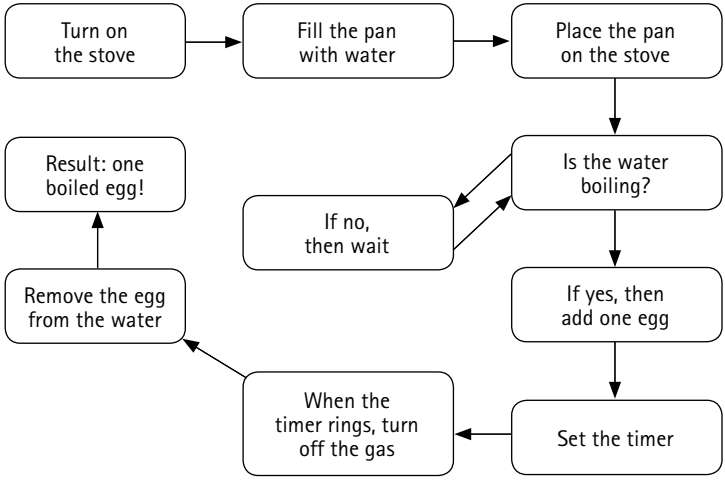

Figure 3.2 A flow chart for boiling an egg.

Turing machines

The use of algorithms to compute the values of functions is at least as old as Ancient Greek mathematics. But it was only relatively recently (in fact, in the 1930s) that the idea came under scrutiny, and mathematicians tried to give a precise meaning to the concept of an algorithm. From the end of the nineteenth century, there had been intense interest in the *foundations* of mathematics. What makes mathematical statements true? How can mathematics be placed on a firm foundation? One question which became particularly pressing was: what *determines* whether a certain method of calculation is adequate for the task in hand? We know in particular cases whether an algorithm is adequate, but is there a general method that will tell us, for any proposed method of calculation, whether or not it is an algorithm?

This question is of deep theoretical importance for mathematics, because algorithms lie at the heart of mathematical practice – but if we cannot say what they are, we cannot really say what mathematics is. An answer to the question was given by the brilliant English mathematician Alan Turing in 1937. As well as being a mathematical genius, Turing (1912–1954) was arguably one of the most influential people of the twentieth century, in an indirect way. As we shall see, he developed the fundamental concepts from which flowed modern digital computers and all their consequences. But he is also famous for cracking the Nazis' Enigma code during the Second World War. This code was used to communicate with U-boats, which at the time were decimating the British Navy, and it is arguable that cracking the code was one of the major factors that prevented Britain from defeat at that point in the war.[2]

Turing answered the question about the nature of computation in a vivid and original way. In effect, he asked: what is the simplest possible device that could perform any computation whatsoever, no matter how complicated? He then proceeded to describe such a device, which is now called (naturally enough) a 'Turing machine'.

A Turing machine is not a machine in the ordinary sense of the word. That is, it is not a physical machine, but rather an abstract, theoretical specification of a possible machine. Though people have

built machines to these specifications, the point of them is not (in the first place) to be built, but to illustrate some very general properties of algorithms and computations.

There can be many kinds of Turing machines for different kinds of computation. But they all have the following features in common: a tape divided into squares and a device that can write symbols on the tape and then read those symbols.[3] The device is also in certain 'internal states' (more on these later), and it can move the tape to the right or to the left, one square at a time. Let us suppose for simplicity that there are only two kinds of symbol that can be written on the tape: '1' and '0'. Each symbol occupies just one square of the tape - so the machine can only read one square at a time. (We don't have to worry yet what these symbols 'mean' – just consider them as *marks* on the tape.)

So the device can only do four things:

1 It can move the tape one square at a time, from left to right or from right to left.
2 It can read a symbol on the tape.
3 It can write a symbol on the tape, either by writing onto a blank square or by overwriting another symbol.
4 It can change its 'internal state'.

The possible operations of a particular machine can be represented by the machine's 'machine table'. The machine table is, in effect, a set of instructions of the form 'if the machine is in state X and reading symbol S, then it will perform a certain operation (e.g. writing or erasing a symbol, moving the tape) and change to state Y (or stay in the same state) and move the tape to the right/left'. If you like, you can think of the machine table as the machine's 'program': it tells the machine what to do. In specifying a particular position in the machine table, we need to know two things: the current *input* to the machine and its current *state*. What the machine does is *entirely fixed* by these two things.

This will all seem pretty abstract, so let's consider a specific example of a Turing machine, one that performs a simple

mathematical operation, that of adding 1 to a number.[4] In order to get a machine to perform a particular operation, we need to *interpret* the symbols on the tape, i.e. take them to represent something. Let's suppose that our 1s on the tape represent numbers: 1 represents the number 1, obviously enough. But we need ways of representing numbers other than 1, so let's use a simple method: rather as a prisoner might represent the days of his imprisonment by rows of scratches on the wall, a line or 'string' of *n* 1s represents the number *n*. So, 111 represents 3, 11111 represents 5, and so on.

To enable two or more numbers to be written on a tape, we can separate numbers by using one or more 0s. The 0s simply function to mark spaces between the numbers – they are the only 'punctuation' in this simple notation. So for example, the tape,

. . . 000011100111111000100 . . .

represents the sequence of numbers 3, 6, 1. In this notation, the number of 0s is irrelevant to which number is written down. The marks . . . indicate that the blank tape continues indefinitely in both directions.

We also need a specification of the machine's 'internal states'; it turns out that the simple machine we are dealing with only needs two internal states, which we might as well call state A (the initial state) and state B. The particular Turing machine we are considering has its behaviour specified by the following instructions:

1 If the machine is in state A, and reads a 0, then it stays in state A, writes a 0, and moves one square to the right.
2 If the machine is in state A, and reads a 1, then it changes to state B, writes a 1, and moves one square to the right.
3 If the machine is in state B, and reads a 0, then it changes to state A, writes a 1 and stops.
4 If the machine is in state B, and reads a 1, then it stays in state B, writes a 1, and moves one square to the right.

The machine table for this machine will look like Figure 3.3.

		INPUT 1	INPUT 0
MACHINE STATE	A	Change to B; Write a 1; Move tape to right	Stay in A; Write a 0; Move tape to right
	B	Stay in B; Write a 1; Move tape to right	Change to A; Write a 1; STOP

Figure 3.3 A machine table for a simple Turing machine.

Let's now imagine presenting the machine with part of a tape that looks like this:

0	0	0	1	1	0	0	0

This tape represents the number 2. (Remember, the 0s merely serve as 'punctuation', they don't represent any number in this notation.) What we want the machine to do is add 1 to this number, by applying the rules in the machine table.

This is how it does it. Suppose it starts off in the initial state, state A, reading the square of tape at the extreme right. Then it follows the instructions in the table. The tape will 'look' like this during this process (the square of the tape currently being read by the machine is underlined):

(i) 0 0 0 1 1 0 0 0 . . .
(ii) . . 0 0 1 1 0 0 0 . . .
(iii) . . 0 0 0 1 1 0 0 0 . . .
(iv) . . . 0 0 0 1 1 0 0 0 . . .
(v) . . . . 0 0 0 1 1 0 0 0 . . .
(vi) . . . . . 0 0 0 1 1 0 0 0 . . .
(vii) . . . . . 0 0 1 1 1 0 0 0 . . .

At line (vi), the machine is in state B, it reads a 0, so it writes a 1, changes to state A, and stops. The 'output' is on line (vii): this represents the number 3, so the machine has succeeded in its task of adding 1 to its input.

But what, you may ask, has this machine really done? What is the *point* of all this tedious shuffling around along an imaginary tape? Like our example of an algorithm for multiplication above, it seems a laborious way of doing something utterly trivial. But, as with our algorithm, the point is not trivial. What the machine has done is *compute a function*. It has computed the function $x + 1$ for the argument 2. It has computed this function by using only the simplest possible 'actions', the 'actions' represented by the four squares of the machine table. And these are only combinations of the very simple steps that were part of the definition of all a Turing machine can do (read, write, change state, move the tape). I shall explain the lesson of this in a moment.

You may be wondering about the role of the 'internal states' in all this. Isn't something being smuggled into the description of this very simple device by talking of its 'internal' states? Perhaps *they* are what is doing the calculation? I think this worry is a very natural one; but it is misplaced. The internal states of the machine are nothing over and above what the machine table says they are. The internal state, B, is, by definition, the state such that if the machine gets a 1 as input, the machine does so-and-so; and such that, if it gets a 0 as input, the machine does such-and-such. That's all there is to these states.[5] ('Internal' may therefore be misleading, as it suggests the states have a 'hidden nature'.)

To design a Turing machine that will perform more complex operations (such as our multiplication algorithm of the previous section), we need a more complex machine table, more internal states, more tape and a more complex notation. *But we do not need any more sophisticated basic operations*. There is no need for us to go into the details of more complex Turing machines, as the basic points can be illustrated by our simple adder. However, it is important to dwell on the issue of notation.

Our prisoner's tally notation for numbers has a number of obvious drawbacks. One is that it can't represent 0 – a big drawback. Another is that very large numbers will take ages to compute, as the machine can only read one square at a time. (Adding 1 to the number 7,000,000 would require a tape with more squares than

there are inhabitants of London.) A more efficient system is the binary system, or base ', where all natural numbers are represented by combinations of 1s and 0s. Recall that, in binary notation, the column occupied by multiples of 10 in the standard 'denary' system (base 10) is occupied by multiples of 2. This gives us the following translation from denary into binary:

1 = 1

2 = 10

3 = 11

4 = 100

5 = 101

6 = 110

7 = 111

8 = 1000

And so on. Obviously, coding numbers in binary gives us the ability to represent much larger numbers more efficiently than our prisoner's tally does.

An advantage of using binary notation is that we can design Turing machines of great complexity without having to add more symbols to the basic repertoire. We started off with two kinds of symbols, 0 and 1. In our prisoner's tally notation, the 0s merely served to divide the numbers from each other. In base 2, the 0s serve as numerals, enabling us to write any number as a string of 1s and 0s. But notice that the machine still only needs the same number of basic operations: read a 1, write a 1, read a 0, write a 0, move the tape. So using base 2 gives us the potential of representing many more numbers much more efficiently without having to add more basic operations to the machine. (Obviously we need punctuation too, to show where one instruction or piece of input stops and another one starts. But, with sufficient ingenuity, we can code these as 1s and 0s too.)

We are now on the brink of a very exciting discovery. With an adequate notation, such as binary, not only the *input* to a Turing machine (the initial tape) but the *machine table itself* can be coded as numbers in the notation. To do this, we need a way of labelling the distinct operations of the machine (read, write, etc.), and the 'internal states' of the machine, with numbers. We used the labels 'A' and 'B' for the internal states of our machine. But this was purely arbitrary: we could have used any symbols whatsoever for these states: %, @, *, or whatever. So we could also use numbers to represent these states. And if we use base 2, we can code these internal states and 'actions' as 1s and 0s on a Turing machine tape.

Because any Turing machine is completely defined by its machine table, and any Turing machine table can be numerically coded, it obviously follows that any Turing machine can be numerically coded. So the machine can be coded in binary, and written on the tape of another Turing machine. So the other Turing machine can take the tape of the first Turing machine as its input: it can *read* the first Turing machine. All it needs is a method of converting the operations described on the tape of the first Turing machine – the program – into its own operations. But this will only be another machine table, which itself can be coded. For example, suppose we code our 'add 1' machine into binary. Then it could be represented on a tape as a string of 1s and 0s. If we add some 1s and 0s representing a number (say 127) to the tape, then these, plus the coding of our 'add 1' machine, can be the input to another Turing machine. This machine would itself have a program which interprets our 'add 1' machine. It can then do exactly what our 'add 1' machine does: it can add 1 to the number fed in, 127. It would do this by 'mimicking' the behaviour of our original 'add 1' machine.

Now, the exciting discovery is this: there is a Turing machine which can mimic the behaviour of any other Turing machine. Because any Turing machine can be numerically coded, it can be fed in as the input to another Turing machine, so long as that machine has a way of reading its tape. Turing proved from this that, to perform all the operations that Turing machines can perform, we don't need a separate machine for each operation. We need only

one machine that is capable of mimicking every other machine. This machine is called a *universal Turing machine*. And it is the idea of a universal Turing machine that lies behind modern general purpose digital computers. In fact, it is not an exaggeration to say that the idea of a universal Turing machine has probably affected the character of all our lives.

However, to say that a universal Turing machine can do anything that any particular Turing machine can do only raises the question: what *can* particular Turing machines do? What sorts of operations can they perform, apart from the utterly trivial one I illustrated?

Turing claimed that any computable function can in principle be computed on a Turing machine, given enough tape and enough time. That is, any algorithm could be executed by a Turing machine. Most logicians and mathematicians now accept the claim that to be an algorithm *is simply* to be capable of execution on some Turing machine, i.e. *being capable of execution on a Turing machine* in some sense tells us what an algorithm is. This claim is called Church's thesis after the American logician Alonzo Church (b. 1903), who independently came to conclusions very similar to those of Turing. (It is sometimes called the Church–Turing thesis.)[6] The basic idea of the thesis is, in effect, to give a precise sense to the notion of an algorithm, to tell us what an algorithm is.

You may still want to ask: *how* has the idea of a Turing machine told us what an algorithm is? How has it helped to appeal to these interminable 'tapes' and the tedious strings of 1s and 0s written on them? Turing's answer could be put as follows: what we have done is reduced anything which we naturally recognise as an effective procedure to a series of simple steps performed by a very simple device. These steps are so simple that it is not possible for anyone to think of them as mysterious. What we have done, then, is to make the idea of an effective procedure unmysterious.

Coding and symbols

A Turing machine is a certain kind of *input–output* device. You put a certain thing 'into' the machine – a tape containing a string of 1s

and 0s – and you get another thing out – a tape containing another string of 1s and 0s. In between, the machine does certain things to the input – the things determined by its machine table or instructions – to turn it into the output.

One thing that might have been worrying you, however, is not the definition of the Turing machine, but the idea that such a machine can perform *any* algorithm whatsoever. It's easy to see how it performs the 'add 1' algorithm, and with a little imagination we can see how it could perform the multiplication algorithm described earlier. But I also said that you could write an algorithm for a simple recipe, such as boiling an egg, or for figuring out which key opens a certain lock. How can a Turing machine do that? Surely a Turing machine can only calculate with numbers, as that is all that can be written on its tape?

Of course, a Turing machine cannot boil an egg, or unlock a door. But the algorithm I mentioned is a *description* of how to boil an egg. And these descriptions can be coded into a Turing machine, given the right notation.

How? Here's one simple way to do it. Our algorithms were written in English, so first we need a way of coding instructions in English text into numbers. We could do this simply by associating each letter of the English alphabet and each significant piece of punctuation with a number, as follows:

A – 1, B – 2, C –3, D – 4, and so on.

So my name would read:

20 9 13

3 18 1 14 5

Obviously, punctuation is crucial. We need a way of saying when one letter stops and another starts, and another way of saying when one word stops and another starts, and yet another way of knowing when one whole piece of text (e.g. a machine table) stops and another starts. But this presents no problem of principle. (Think how old-fashioned telegrams used words for punctuation, e.g. separat-

ing sentences with 'STOP'.) Once we've coded a piece of text into numbers, we can rewrite these numbers in binary.

So we could then convert any algorithm written in English (or any other language) into binary code. And this could then be written on a Turing machine's tape, and serve as input to the universal Turing machine.

Of course, actual computer programmers don't use this system of notation for text. But I'm not interested in the real details at the moment: the point I'm trying to get across is just that once you realise that any piece of text can be coded in terms of numbers, then it is obvious that any algorithm that can be written in English (or in any other language) can be run on a Turing machine.

This way of representing is wholly *digital*, in the sense that each represented element (a letter, or word) is represented in an entirely 'on–off' way. Any square on a Turing machine's tape has either a 1 on it or a 0. There are no 'in-between' stages. The opposite of digital form of representation is the *analogue* form. The distinction is best illustrated by the familiar example of analogue and digital clocks. Digital clocks represent the passage of time in a step-by-step way, with distinct numbers for each second (say), and nothing in between these numbers. Analogue clocks, by contrast, mark the passage of time by the smooth movement of a hand across the face. Analogue computers are not directly relevant to the issues raised here – the computers discussed in the context of computers and thought are all digital computers.[7]

We are now, finally, getting close to our characterisation of computers. Remember that I said that a computer is a device that processes representations in a systematic way. To understand this, we needed to give a clear sense to two ideas: (i) 'processes in a systematic way' and (ii) 'representation'. The first idea has been explained in terms of the idea of an algorithm, which has in turn been illuminated by the idea of a Turing machine. The second idea is implicit in the idea of the Turing machine: for the machine to be understood as actually computing a function, the numbers on its tape have to be taken as *standing for* or *representing* something. Other representations – e.g. English sentences – can then be coded into these numbers.

Sometimes computers are called information processors. Sometimes they are called symbol manipulators. In my terminology, this is the same as saying that computers process representations. Representations carry information in the sense that they 'say' something, or are interpretable as 'saying' something. That is *what* computers process or manipulate. *How* they process or manipulate is by carrying out effective procedures.

Instantiating a function and computing a function

This talk of representations now enables us to make a very important distinction that is crucial for understanding how the idea of computation applies to the mind.[8]

Remember that the idea of a function can be extended beyond mathematics. In scientific theorising, for example, scientists often describe the world in terms of functions. Consider a famous simple example: Newton's second law of motion, which says that the acceleration of a body is determined by its mass and the forces applied to it. This can be represented as $F = ma$, which reads 'Force = mass × acceleration'. The details of this don't matter: the point is that the force or forces acting on a certain body will equal the mass times the acceleration. A mathematical function – multiplication – whose arguments and values are numbers can represent the relationship in nature between masses, forces and accelerations. This relationship in nature is a function too: the acceleration of a body is a function of its mass and the forces exerted upon it. Let's call this 'Newton's function' for simplicity.

But, when a particular mass has a particular force exerted upon it, and accelerates at a certain rate, it does not *compute* the value of Newton's function. If it did, then every force–mass–acceleration relationship in nature would be a computation, and every physical object a computer. Rather, as I shall say, a particular interaction *instantiates* the function: that is, it is an *instance* of Newton's function. Likewise, when the planets in the solar system orbit the sun, they do so in a way that is a function of gravitational and inertial 'input'. Kepler's laws are a way of describing this function. But the

solar system is not a computer. The planets do not 'compute' their orbits from the input they receive: they just move.

So the crucial distinction we need is between a system's *instantiating* a function and a system's *computing* a function. By 'instantiating' I mean 'being an instance of' (if you prefer, you could substitute 'being describable by'). Compare the solar system with a real computer, say a simple adding machine. (I mean an actual physical adding machine, not an abstract Turing 'machine'.) It's natural to say that an adding machine computes the addition function by taking two or more numbers as input (arguments) and giving you their sum as output (value). But, strictly speaking, this is not what an adding machine does. For, whatever numbers are, they aren't the sort of thing that can be fed into machines, manipulated or transformed. (For example, you don't destroy the number 3 by destroying all the 3s written in the world; that doesn't make sense.) What the adding machine really does is take *numerals* – that is, representations of numbers – as input, and gives you numerals as output. This is the difference between the adding machine and the planets: although they instantiate a function, the planets do not employ representations of their gravitational and other input to form representations of their output.

Computing a function, then, requires representations: representations as the input and representations as the output. This is a perfectly natural way of understanding 'computing a function': when we compute with pen and paper, for example, or with an abacus, we use representations of numbers. As Jerry Fodor has said: 'No computation without representation!'.[9]

How does this point relate to Turing machines and algorithms? A Turing machine table specifies transitions between the states of the machine. According to Church's thesis, any procedure that is step-by-step algorithmic can be modelled on a Turing machine. So, any process in nature which can be represented in a step-by-step fashion can be represented by a Turing machine. The machine merely specifies the transitions between the states involved in the process. But this doesn't mean that these natural processes are *computations*, any more than the fact that physical quantities such as

my body temperature can be represented by numbers means that my body temperature actually *is* a number. If a theory of some natural phenomenon can be represented algorithmically, then the theory is said to be *computable* – but this is a fact about theories, not about the phenomena themselves. The idea that theories may or may not be computable will not concern us any further in this book.[10]

Without wishing to labour the point, let me emphasise that this is why we needed to distinguish at the beginning of this chapter between the idea that some systems can be *modelled* on a computer and the idea that some systems actually perform computations. A system can be modelled on a computer when a *theory* of that system is *computable*. A system performs computations, however, when it processes representations by using an effective procedure.

Automatic algorithms

If you have followed the discussion so far, then a very natural question will occur to you. Turing machines describe the abstract structure of computation. But, in the description of Turing machines, we have appealed to ideas like 'moving the tape', 'reading the tape', 'writing a symbol' and so on. We have taken these ideas for granted, but how are they supposed to work? How is it that any effective procedure gets off the ground at all, without the intervention of a human being at each stage in the procedure?

The answer is that the computers with which we are familiar use *automated* algorithms. They use algorithms, and input and output representations, that are in some way 'embodied' in the physical structure of the computer. The last part of our account of computers will be a very brief description of how this can be done. This brief discussion cannot, of course, deal with all the major features of how actual computers work, but I hope it will be enough to give you the general idea.

Consider a very simple machine (not a computer) that is used for trapping mice. We can think of this mousetrap in terms of input and output: the trap takes live mice as input, and gives dead (or

perhaps just trapped) mice as output. A simple way of representing the mousetrap is shown in Figure 3.4.

From the point of view of the simple description of the mousetrap, it doesn't really matter what's in the MOUSETRAP 'box': what's 'in the box' is whatever is it that traps the mice. Boxes like this are known to engineers as 'black boxes': we can treat something as a black box when we are not really interested in how it works internally, but are interested only in the input–output tasks it performs. But, of course, we can 'break into' the black box of our mousetrap and represent its innards as in Figure 3.5.

The two internal components of the black box are the bait and the device that actually traps the mice (the arrow is meant to indicate that the mouse will move from the bait into the trapping device, not vice versa). In Figure 3.4, we are, in effect, treating the BAIT and TRAPPING DEVICE as black boxes. All we are interested in is what they do: the BAIT is whatever it is that attracts the mouse, and the TRAPPING DEVICE is whatever it is that traps the mouse.

But we can of course break into *these* black boxes too, and find out how they work. Suppose that our mousetrap is of the old-fashioned comic-book kind, with a metal bar held in place by a spring, which is released when the bait is taken. We can then

Figure 3.4 Mousetrap 'black box'.

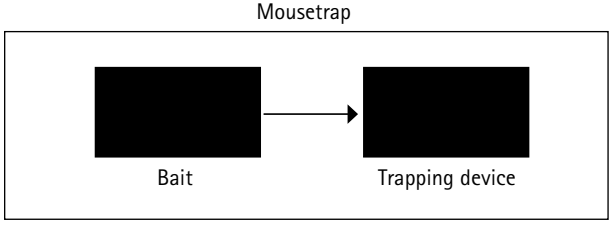

Figure 3.5 The mousetrap's innards.

describe the trapping device in terms of its component parts. And its component parts too – SPRING, BAR etc. – can be thought of as black boxes. It doesn't matter exactly what they are; what matters is what they are *doing* in the mousetrap. But, these boxes too can be broken into, and we can specify in more detail how they work. What is treated as one black box at one level can be broken down into other black boxes at other levels, until we come to understand the workings of the mousetrap.

This kind of analysis of machines is sometimes known as 'functional analysis': the analysis of the working of the machine into the functions of its component parts. (It is also sometimes called 'functional boxology'.) Notice, though, that the word 'function' is being used in a different sense than in our earlier discussion: here, the function of a part of a system is the causal role it plays in the system. This use of 'function' corresponds more closely to the everyday use of the term, as in 'what's the function of this bit?'.

Now back to computers. Remember our simple algorithm for multiplication. This involved a number of tasks, such as writing symbols on the X and Y pieces of paper, and adding and subtracting. Now think of a machine that carries out this algorithm, and let's think of how to functionally analyse it. At the most general level, of course, it is a multiplier. It takes numerals as input and gives you their products as output. At this level, it may be thought of as a black box (see Figure 3.6).

But this doesn't tell us much. When we 'look' inside the black box, what is going on is what is represented by the flow chart (Figure 3.7). Each box in the flow chart represents a step performed by the machine. But some of these steps can be broken down into simpler steps. For example, step (iv) involves *adding* the number written on Y to the ANSWER. But adding is also a step-by-step procedure,

Figure 3.6 Multiplier black box.

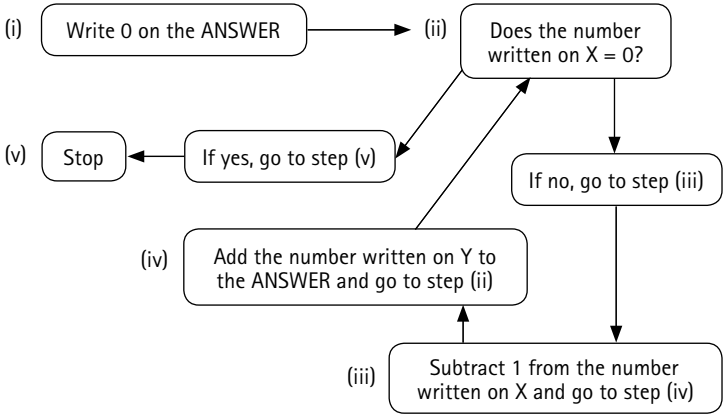

Figure 3.7 Flow chart for the multiplication algorithm again.

and so we can write a flow chart for this too. Likewise with the other steps: subtracting, 'reading' and so on. When we functionally analyse the multiplier, we find out that its tasks become simpler and simpler, until we get down to the simplest tasks it can perform.

Daniel Dennett has suggested a vivid way of thinking of the architecture of computers. Imagine each task in the flow chart's boxes being performed by a little man, or 'homunculus'. The biggest box (labelled Multiplier in Figure 3.6) contains a fairly intelligent homunculus, who, say, multiplies numbers expressed in denary notation. But inside this homunculus are other, less intelligent, homunculi who can do only addition and subtraction, and writing denary symbols on the paper. Inside these other homunculi are even more stupid homunculi who can translate denary notation into binary. And inside these are really stupid homunculi who can only read, write or erase binary numerals. Thus, the behaviour of the intelligent multiplier is functionally explained by postulating progressively more and more stupid homunculi.[11]

If we have a way of making a real physical device that functions as a simple device – a stupid homunculus – we can build up combinations of these simple devices into complex devices that can perform the task of the multiplier. After all, the multiplier is nothing

more than these simple devices arranged in the way specified by the flow chart. Now, remember that Turing's great insight was to show that any algorithm could be broken down into tasks simple enough to be performed by a Turing machine. So let's think of the simplest devices as the devices which can perform these simple Turing machine operations: move from left or right, read, write, etc. All we need to do now is make some devices that can perform these simple operations.

And, of course, we have many ways of making them. For vividness, think of the tape of some Turing machine represented by an array of switches: the switch being on represents 1 and the switch being off represents 0. Then any computation can be performed by a machine that can move along the switches one by one, register which position they are in ('reading') and turn them on or off ('writing'). So long as we have some way of *programming* the machine (i.e. telling it which Turing machine it is mimicking), then we have built a computer out of switches.

Real computers are, in a sense, built out of 'switches', although not in the simple way just described. One of the earliest computers (built in 1944) used telephone relays, while the Americans' famous war effort ENIAC (used for calculating missile trajectories) was built using valves; and valves and relays are, in effect, just switches. The real advances came when the simplest processors (the 'switches') could be built out of semi-conductors, and computations could be performed faster than Turing ever dreamed of. Other major advances came with high-level 'programming languages': systems of coding that can make the basic operations of the machine perform all sorts of other more complex operations. But, for the purposes of this book, the basic principle behind even these very complex machines can be understood in the way I have outlined. (For more information about the history of the computer, see the chronology at the end of this book.)

One important consequence of this is that it doesn't really matter what the computer is made of. What matters to its being a computer is *what it does* – that is, what computational tasks it performs, or what *program* it is running. The computers we use

today perform these tasks using microscopic electronic circuits etched on tiny pieces of silicon. But, although this technology is incredibly efficient, the tasks performed are, in principle, capable of being performed by arrays of switches, beads, matchsticks and tin cans, and even perhaps by the neurochemistry of the brain. This idea is known as the 'variable realisation' (or 'multiple realisation') of program (or software) by physical mechanism (hardware), i.e. the same program can be variably or multiply 'realised' by different pieces of hardware.

I should add one final point about some real computers. It is a simplification to say that all computers work entirely algorithmically. When people build computer programs to play chess, for example, the rules of chess tell the machine, entirely unambiguously, what counts as a legal move. At any point in the game only certain moves are allowed by the rules. But how does the machine know *which* move to make, out of all the possible moves? As a game of chess will come to an end in a finite – though possibly very large – number of moves, it is possible in principle for the machine to scan ahead, figuring out every consequence of every permitted move. However, this would take even the most powerful computer an enormous (to put it mildly) amount of time. (John Haugeland estimates that the computer would have to look ahead 10^{120} moves – which is a larger number than the number of quantum states in the whole history of the universe.[12]) So, designers of chess-playing programs add to their machines certain rules of thumb (called *heuristics*) that suggest good courses of action, though, unlike algorithms, they do not guarantee a particular outcome. A heuristic for a chess-playing machine might be something like, 'Try and castle as early in the game as possible'. Heuristics have been very influential in artificial intelligence research. It is time now to introduce the leading idea behind artificial intelligence: the idea of a thinking computer.

Thinking computers?

Equipped with a basic understanding of what computers are, the question we now need to ask is: why would anyone think that

being a computer – processing representations systematically – can constitute thinking?

At the beginning of this chapter, I said that to answer the question, 'Can a computer think?', we need to know three things: what a computer is, what thinking is and what it is about thought and computers that supports the idea that computers might think. We now have something of an idea of what a computer is, and in Chapters 1 and 2 we discussed some aspects of the common-sense conception of thought. Can we bring these things together?

There are a number of obvious connections between what we have learned about the mind and what we have learned about computers. One is that the notion of *representation* seems to crop up in both areas. One of the essential features of certain states of mind is that they represent. And in this chapter we have seen that one of the essential features of computers is that they process representations. Also, your thoughts cause you to do what you do because of how they represent the world to be. And it is arguable that computers are caused to produce the output they do because of what they represent: my adding machine is caused to produce the output 5 in response to the inputs 2, +, 3 and =, partly because those input symbols represent what they do.

However, we should not get too carried away by these similarities. The fact that the notion of representation can be used to define both thought and computers does not imply anything about whether computers can think. Consider this analogy: the notion of representation can be used to define both thought and books. It is one of the essential features of books that they contain representations. But books can't think! Analogously, it would be foolish to argue that computers can think simply because the notion of representation can be employed in defining thought and computers.

Another way of getting carried away is to take the notion of 'information processing' too loosely. In a sense, thinking obviously does involve processing information – we take information in from our environments, do things to it and use it in acting in the world. But it would be wrong to move from this plus the fact that computers are known as 'information processors' to the conclusion

that what goes on in computers must be a kind of thinking. This relies on taking 'information processing' in a very loose way when applying it to human thought, whereas, in the theory of computing, 'information processing' has a precise definition. The question about thinking computers is (in part) about whether the information processing that *computers* do can have anything to do with the 'information processing' involved in *thought*. And this question cannot be answered by pointing out that the words 'information processing' can be applied to both computers and thought: this is known as a 'fallacy of equivocation'.

Another bad way to argue, as we have already seen, is to say that computers can think because there must be a Turing machine table for thinking. To say that there is a Turing machine table for thinking is to say that the *theory* of thinking is computable. This may be true; or it may not. But, even if it were true, it obviously would not imply that thinkers are computers. Suppose astronomy were computable: this would not imply that the universe is a computer. Once again, it is crucial to emphasise the distinction between computing a function and instantiating a function.

On the other hand, we must not be too quick to dismiss the idea of thinking computers. One familiar debunking criticism is that people have always thought of the mind or brain along the lines of the latest technology; and the present infatuation with thinking computers is no exception. This is how John Searle puts the point:

> Because we do not understand the brain very well we are constantly tempted to use the latest technology as a model for trying to understand it. In my childhood we always assured that the brain was a telephone switchboard . . . Sherrington, the great British neuroscientist, thought that the brain worked like a telegraph system. Freud often compared the brain to hydraulic and electro-magnetic systems. Leibniz compared it to a mill, and I am told that some of the ancient Greeks thought the brain functions like a catapult. At present, obviously, the metaphor is the digital computer.[13]

Looked at in this way, it seems bizarre that anyone should think that the human brain (or mind), which has been evolving for millions

of years, should have its mysteries explained in terms of ideas that arose some sixty or seventy years ago in rarified speculation about the foundations of mathematics.

But, in itself, the point proves nothing. The fact that an idea evolved in a specific historical context – and which idea didn't? – doesn't tell us anything about the *correctness* of the idea. However, there's also a more interesting specific response to Searle's criticism. It may be true that people have always thought of the mind by analogy with the latest technology. But the case of computers is very different from the other cases that Searle mentions. Historically, the various stages in the invention of the computer have always gone hand in hand with attempts to systematise aspects of human knowledge and intellectual skills – so it is hardly surprising that the former came to be used to model (or even explain) the latter. This is not so with hydraulics, or with mills or telephone exchanges. It's worth dwelling on a few examples.

Along with many of his contemporaries, the great philosopher and mathematician G.W. Leibniz (1646–1716) proposed the idea of a 'universal character' (*characteristica universalis*): a mathematically precise, unambiguous language into which ideas could be translated, and by means of which the solutions to intellectual disputes could be resolved by 'calculation'. In a famous passage, Leibniz envisages the advantages that such a language would bring:

> Once the characteristic numbers are established for most concepts, mankind will then possess a new instrument which will enhance the capabilities of the mind to a far greater extent than optical instruments strengthen the eyes, and will supersede the microscope and telescope to the same extent that reason is superior to eyesight.[14]

Leibniz did not get as far as actually designing the universal character (though it is interesting that he did invent binary notation). But with the striking image of this concept-calculating device we see the combination of interests which have preoccupied many computer pioneers: on the one hand, there is a desire to strip human thought of all ambiguity and unclarity; while, on the other, there is the idea of a calculus or machine that could process these skeletal thoughts.

These two interests coincide in the issues surrounding another major figure in the computer's history, the Irish logician and mathematician George Boole (1815–1864). In his book *The Laws of Thought* (1854), Boole formulated an algebra to express logical relations between statements (or propositions). Just as ordinary algebra represents mathematical relations between numbers, Boole proposed that we think of the elementary logical relations between statements or propositions – expressed by words such as 'and', 'or', etc. – as expressible in algebraic terms. Boole's idea was to use a binary notation (1 and 0) to represent the arguments and values of the functions expressed by 'and', 'or', etc. For example, take the binary operations $1 \times 0 = 0$ and $1 + 0 = 1$. Now, suppose that 1 and 0 represent *true* and *false* respectively. Then we can think of $1 \times 0 = 0$ as saying something like, 'If you have a truth and a falsehood, then you get a falsehood' and $1 + 0 = 1$ as saying 'If you have a truth or a falsehood, then you get a truth'. That is, we can think of $\times$ as representing the 'truth-function' *and,* and think of + as representing the truth-function *or*. (Boole's ideas will be familiar to students of elementary logic. A sentence 'P and Q' is true just in case P and Q are both true, and 'P or Q' is true just in case P is true or Q is true.)

Boole claimed that, by building up patterns of reasoning out of these simple algebraic forms, we can discover the 'fundamental laws of those operations of the mind by which reason is performed'.[15] That is, he aimed to systematise or codify the principles of human thought. The interesting fact is that Boole's algebra came to play a central role in the design of modern digital computers. The behaviour of the function $\times$ in Boole's system can be coded by a simple device known as an 'and-gate' (see Figure 3.8). An and-gate is a mechanism taking electric currents from two sources (X and Y) as inputs, and giving one electric current as output (Z). The device is designed in such a way that it will output a current at Z when, and

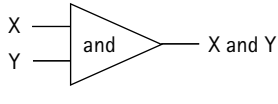

Figure 3.8 An 'and-gate'.

only when, it is receiving a current from *both* X and Y. In effect, this device represents the truth function 'and'. Similar gates are constructed for the other Boolean operations: in general, these devices are called 'logic gates' and are central to the design of today's digital computers.

Eventually, the ideas of Boole and Leibniz, and other great innovators, such as the English mathematician Charles Babbage (1792–1871), gave birth to the idea of the general-purpose programmable digital computer. The idea then became reality in the theoretical discoveries of Turing and Church, and in the technological advances in electronics of the post-war years (see the chronology at the end of the book for some more details). But, as the cases of Boole and Leibniz illustrate, the ideas behind the computer, however vague, were often tied up with the general project of understanding human thought by systematising or codifying it. It was only natural, then, when the general public became aware of computers, that they were hailed as 'electronic brains'.[16]

These points do not, of course, *justify* the claim that computers can think. But they do help us see what is wrong with some hasty reactions to this claim. In a moment we will look at some of the detailed arguments for and against it. But first we need to take a brief look at the idea of artificial intelligence itself.

Artificial intelligence

What is artificial intelligence? It is sometimes hard to get a straight answer to this question, as the term is applied to a number of different intellectual projects. Some people call artificial intelligence (or AI) the 'science of thinking machines', while others, e.g. Margaret Boden, are more ambitious, calling it 'the science of intelligence in general'.[17] To the newcomer, the word 'intelligence' can be a bit misleading here, because it suggests that AI is interested only in tasks which we would ordinarily classify as requiring intelligence – e.g. reading difficult books or proving theorems in mathematics. In fact, a lot of AI research concentrates on matters which we wouldn't ordinarily think of as requiring intelligence, such as seeing three-dimensional objects or understanding simple text.

Some of the projects that go under the name of AI have little to do with thought or thinking computers. For example, there are the so-called 'expert systems', which are designed to give advice on specialised areas of knowledge – e.g. drug diagnosis. Sophisticated as they are, expert systems are not (and are not intended to be) thinking computers. From the philosophical point of view, they are simply souped-up encyclopaedias.

The philosophically interesting idea behind AI is the idea of building a thinking computer (or any other machine, for that matter). Obviously, this is an interesting question in itself; but, if Boden and others are right, then the project of building a thinking computer should help us understand what intelligence (or thought) is in general. That is, by building a thinking computer, we can learn about thought.

It may not be obvious how this is supposed to work. How can building a thinking computer tell us about how we think? Consider an analogy: building a flying machine. Birds fly, and so do aeroplanes; but building aeroplanes does not tell us very much about how birds manage to fly. Just as aeroplanes fly in a different way from the way birds do, so a thinking computer might think in a different way from the way we do. So how can building a thinking computer in itself tell us much about human thought?

On the other hand, this argument might strike you as odd. After all, thinking is what *we* do – the essence of thinking is human thinking. So how could anything think without thinking in the way we do? This is a good question. What it suggests is that, instead of starting off by building a thinking computer and *then* asking what this tells us about thought, we should first figure out what thinking is, and then see if we can build a machine which does this. However, once we had figured out what thinking is, building the machine wouldn't then tell us anything we didn't already know!

If the only kind of thinking were human thinking (whatever this means exactly) then it would only be possible to build a thinking computer if human thinking actually *were* computational. To establish this, we would obviously have to investigate in detail what thinking and other mental processes are. So this approach will need

a psychological theory behind it: for it will need to figure out what the processes are before finding out what sort of computational mechanisms carry out these processes. The approach will then involve a collaboration between psychology and AI, to provide the full theory of human mental processing. I'll follow recent terminology in calling this collaboration 'cognitive science' – this will be topic of Chapter 4.[18]

On the other hand, if something could think, but *not* in the way we do, then AI should not be constrained by finding out about how human psychology works. Rather, it should just go ahead and make a machine that performs a task with thought or intelligence, regardless of the way we do it. This was, in fact, the way that the earliest AI research proceeded after its inception in the 1950s. The aim was to produce a machine that would do things that *would* require thought if done by people. They thought that doing this would not require detailed knowledge of human psychology or physiology.[19]

One natural reaction to this is that this approach can only ever produce a *simulation* of thought, not the real thing. For some, this is not a problem: if the machine could do the job in a intelligent-seeming way, then why should we worry about whether it is the 'real thing' or not? However, this response is not very helpful if AI really is supposed to be the 'science of intelligence in general', as, by blurring the distinction between real thought and simulation, it won't be able to tell us very much about how our (presumably real) thought works. So how could anyone think that it was acceptable to blur the distinction between real thought and its simulation?

The answer, I believe, lies in the early history of AI. In 1950, Turing published an influential paper called 'Computing Machinery and Intelligence', which provided something of the philosophical basis of AI. In this paper, Turing addressed the question, 'Can a machine think?'. Finding this question too vague, he proposed replacing it with the question: 'Under what circumstances would a machine be mistaken for a real thinking person?'. Turing devised a test in which a person is communicating at a distance with a machine and another person. Very roughly, this 'Turing test' amounts to this: if the first person cannot tell the difference between the conversation

with the other person and the conversation with the machine, then we can say that the machine is thinking.

There are many ramifications of this test, and spelling out in detail what it involves is rather complicated.[20] My own view is that the assumptions behind the test are behaviouristic (see Chapter 2, 'Understanding other minds', p. 47) and that the test is therefore inadequate. But the only point I want to make here is that accepting the Turing test as a decisive test of intelligence makes it possible to separate the idea of something *thinking* from the idea of something *thinking in the way humans do*. If the Turing test is an adequate test of thought, then all that is relevant is how the machine performs in the test. It is not relevant whether the machine passes the test in the way that humans do. Turing's redefinition of the question 'Can a machine think?' enabled AI to blur the distinction between real thought and its mere simulation.

This puts us in a position to distinguish between the two questions I raised at the beginning of this chapter:

1 Can a computer think? That is, can something think simply by being a computer?
2 Is the human mind a computer? That is, do we think (in whole or in part) by computing?

These questions are distinct, because someone taking the latter kind of AI approach could answer 'Yes' to 1 while remaining agnostic on 2 ('I don't know how *we* manage to think, but here's a computer that can!'). Likewise, someone could answer 'Yes' to question 2 while denying that a mere computer could think. ('Nothing could think *simply* by computing; but computing is part of the story about how we think.')

Chapter 4 will deal with question 2, while the rest of this chapter will deal with some of the most interesting philosophical reasons for saying 'No' to question 1. For the sake of clarity, I will use the terms 'AI' and 'artificial intelligence' for the view that computers can think – but it should be borne in mind that these term are also used in other ways.

How has philosophy responded to the claims of AI, so defined? Two philosophical objections stand out:

1 Computers cannot think because thinking requires abilities that computers by their very nature can never have. Computers have to obey rules (whether algorithms or heuristics), but thinking can never be captured in a system of rules, no matter how complex. Thinking requires rather an active engagement with life, participation in a culture and 'know-how' of the sort that can never be formalised by rules. This is the approach taken by Hubert Dreyfus in his blistering critique of AI, *What Computers Can't Do*.
2 Computers cannot think because they only manipulate symbols according to their *formal* features; they are not sensitive to the *meanings* of those symbols. This is the theme of a well-known argument by John Searle: the 'Chinese room'.

In the final two sections of this chapter, I shall assess these objections.[21]

Can thinking be captured by rules and representations?

The *Arizona Daily Star* for 31 May 1986 reported this unfortunate story:

> A rookie bus driver, suspended for failing to do the right thing when a girl suffered a heart attack on his bus, was following overly strict rules that prohibit drivers from leaving their routes without permission, a union official said yesterday. 'If the blame has to be put anywhere, put it on the rules that those people have to follow' [said the official]. [A spokesman for the bus company defended the rules]: 'You give them a little leeway, and where does it end up?'[22]

The hapless driver's behaviour can be used to illustrate a perennial problem for AI. By sticking to the strict rule – 'only leave your route if you have permission' – the driver was unable to deal with the emergency in an intelligent, thinking way. But computers must, by

their very nature, stick to (at least some) strict rules – and, therefore, will never be able to behave with the kind of flexible, spontaneous responses that real thinkers have. The objection concludes that thinking cannot be a matter of using strict rules; so computers cannot think.

This objection is a bit quick. Why doesn't the problem lie with the *particular* rules chosen, rather than the idea of following a rule as such? The problem with the rule in the example – 'Only leave your route if you have permission' – is just that it is too simple, not that it is a *rule*. The bus company should have given the driver a rule more like: 'Only leave your route if you have permission, unless a medical emergency occurs on board, in which case you should drive to the nearest hospital'. This rule would deal with the heart attack case – but what if driver knows that the nearest hospital is under siege from terrorists? Or what if he knows that there is a doctor on board? Should he obey the rule telling him to go to a hospital? Probably not – but, if he shouldn't, then should he obey some other rule? But which rule is this?

It is absurd to suppose that the bus company should present the driver with a rule like, 'Only leave your route if you have permission, unless a medical emergency occurs on board, in which case you should drive to the nearest hospital, unless the hospital is under siege from international terrorists, or unless there is a doctor on board, or . . . in which case you should . . . ' – we don't even know how to fill in the dots. How can we get a rule that is *specific* enough to give the person following it precise directions about what to do (e.g. 'Drive to the nearest hospital' rather than 'Do something sensible') but *general* enough to apply to all eventualities (e.g. not just to heart attacks, but to emergencies in general)?

In his essay, 'Politics and the English language', George Orwell gives a number of rules for good writing (e.g. 'Never use a long word where a short one will do'), ending with the rule: 'Break any of these rules sooner than say anything outright barbarous'.[23] We could add an analogous rule to the bunch of rules given to the bus driver: 'Break any of these rules sooner than do anything stupid'. Or, more politely, 'Use your common sense!'.

With human beings, we can generally rely on them to use their common sense, and it's hard to know how we could understand problems like the bus driver's without appealing (at some stage) to something like common sense, or 'what it's reasonable to do'. If a computer were to cope with a simple problem like this, it will have to use common sense too. But computers work by manipulating representations according to rules (algorithms or heuristics). So, for a computer to deal with the problem, common sense will have to be stored in the computer in terms of rules and representations. What AI needs, then, is a way of programming computers with explicit representations of common-sense knowledge.

This is what Dreyfus says can't be done. He argues that human intelligence requires 'the background of common-sense that adult human beings have by virtue of having bodies, interacting skilfully with the material world, and being trained in a culture'.[24] And, according to Dreyfus, this common-sense knowledge cannot be represented as 'a vast base of propositional knowledge', i.e. as a bunch of rules and representations of facts.[25]

The chief reason why common-sense knowledge can't be represented as a bunch of rules and representations is that common-sense knowledge is, or depends on, a kind of *know-how*. Philosophers distinguish between knowing *that* something is the case and knowing *how* to do something. The first kind of knowledge is a matter of knowing facts (the sorts of things that can be written in books: e.g. knowing that Sofia is the capital of Bulgaria), while the second is a matter of having skills or abilities (e.g. being able to ride a bicycle).[26] Many philosophers believe that an ability such as knowing how to ride a bicycle is not something that can be entirely reduced to knowledge of certain rules or principles. What you need to have when you know how to ride a bicycle is not 'book-learning': you don't employ a rules such as 'when turning a corner to the right, then lean slightly to the right with the bicycle'. You just *get the hang of it*, through a method of trial and error.

And, according to Dreyfus, getting the hang of it is what you do when you have general intelligence too. Knowing *what a chair is* is not just a matter of knowing the definition of the word 'chair'. It also

essentially involves knowing what to do with chairs, how to sit on them, get up from them, being able to tell which objects in the room are chairs, or what sorts of things can be used as chairs if there are no chairs around – that is, the knowledge presupposes a 'repertoire of bodily skills which may well be indefinitely large, because there seems to be an indefinitely large variety of chairs and of successful (graceful, comfortable, secure, poised, etc.) ways to sit in them'.[27] The sort of knowledge that underlies our everyday way of living in the world either is – or rests on – practical know-how of this kind.

A computer is a device that processes representations according to rules. And representations and rules are obviously not skills. A book contains representations, and it can contain representations of rules too – but a book has no skills. If the computer has knowledge, it must be 'knowledge that so-and-so is the case' rather than 'knowledge of how to do so-and-so'. So, if Dreyfus is right, and general intelligence requires common sense, and common sense is a kind of know-how, then computers cannot have common sense, and AI cannot succeed in creating a computer which has general intelligence. The two obvious ways for the defenders of AI to respond are *either* to reject the idea that general intelligence requires common sense *or* to reject the idea that common sense is know-how.

The first option is unpromising – how could there be general intelligence which did not employ common sense? – and is not popular among AI researchers.[28] The second option is a more usual response. Defenders of this option can say that it requires hard work to make explicit the assumptions implicit in the common-sense view of the world; but this doesn't mean that it can't be done. In fact, it has been tried. In 1984, the Microelectronics and Computer Technology Corporation of Texas set up the CYC project, whose aim was to build up a knowledge base of a large amount of common-sense knowledge. (The name 'CYC' derives from 'encyclopaedia'.) Those working on CYC attempt to enter common-sense assumptions about reality, assumptions so fundamental and obvious that they are normally overlooked (e.g. that solid objects are not generally penetrable by other solid objects etc.). The aim is to express a large percentage of common-sense knowledge in terms of about 100 million

propositions, coded into a computer. In the first six years of the project, one million propositions were in place. The director of the CYC project, Doug Lenat, once claimed that, by 1994, they would have stored between thirty and fifty per cent of common-sense knowledge (or, as they call it, 'consensus reality').[29]

The ambitions behind schemes like CYC have been heavily criticised by Dreyfus and others. However, even if all common-sense knowledge could be stored as a bunch of rules and representations, this would only be the beginning of AI's problems. For it is not enough for the computer merely to have the information stored; it must be able to retrieve it and use it in a way that is intelligent. It's not enough to have an encyclopaedia – one must be able to know how to look things up in it.

Crucial here is the idea of *relevance*. If the computer cannot know which facts are relevant to which other facts, it will not perform well in using the common sense it has stored to solve problems. But whether one thing is relevant to another thing varies as conceptions of the world vary. The sex of a person is no longer thought to be relevant to whether they have a right to vote; but two hundred years ago it was.

Relevance goes hand in hand with a sense of what is out of place or what is exceptional or unusual. Here is what Dreyfus says about a program intended for understanding stories about restaurants:

> [T]he program has not understood a restaurant story the way people in our culture do, until it can answer such simple questions as: When the waiter came to the table did he wear clothes? Did he walk forward or backward? Did the customer eat his food with his mouth or his ear? If the program answers 'I don't know', we feel that all its right answers were tricks or lucky guesses and that it has not understood anything of our everyday restaurant behaviour.[30]

Dreyfus argues that it is only because we have a way of living in the world that is based on skills and interaction with things (rather than the representation of propositional knowledge or 'knowledge that so-and-so') that we are able to know what sorts of things are out of place, and what is relevant to what.

There is much more to Dreyfus's critique of AI than this brief summary suggests – but I hope this gives an idea of the general line of attack. The problems raised by Dreyfus are sometimes grouped under the heading of the 'frame problem',[31] and they raise some of the most difficult issues for the traditional approach to AI, the kind of AI described in this chapter. There are a number of ways of responding to Dreyfus. One response is that of the CYC project: to try and meet Dreyfus's challenge by itemising 'consensus reality'. Another response is to concede that 'classical' AI, based on rules and representations, has failed to capture the abilities fundamental to thought – AI needs a radically different approach. In Chapter 4, I shall outline an example of this approach, known as 'connectionism'. Another response, of course, is to throw up one's hands in despair, and give up the whole project of making a thinking machine. At the very least, Dreyfus's arguments present a challenge to the research programme of AI: the challenge is to represent common-sense knowledge in terms of rules and representations. And, at most, the arguments signal the ultimate breakdown of the idea that the essence of thought is manipulating symbols according to rules. Whichever view one takes, I think that the case made by Dreyfus licenses a certain amount of scepticism about the idea of building a thinking computer.

The Chinese room

Dreyfus argues that conventional AI programs don't stand a chance of producing anything that will succeed in passing for general intelligence – e.g. plausibly passing the Turing test. John Searle takes a different approach. He allows, for the sake of argument, that an AI program could pass the Turing test. But he then argues that, even if it did, it would only be a *simulation* of thinking, not the real thing.[32]

To establish his conclusion, Searle uses a thought experiment which he calls the 'Chinese room'. He imagines himself to be inside a room with two windows – let's label them I and O respectively. Through the I window come pieces of paper with complex markings

on them. In the room is a huge book written in English, in which is written instructions of the form, 'Whenever you get a piece of paper through the I window with *these* kinds of markings on it, do certain things to it, and pass a piece of paper with *those* kind of markings on it through the O window'. There is also a pile of pieces of paper with markings inside the room.

Now suppose the markings are in fact Chinese characters – those coming through the I window are questions, and those going through the O window are sensible answers to the questions. The situation now resembles the set-up inside a computer: a bunch of rules (the program) operates on symbols, giving out certain symbols through the output window in response to other symbols through the input window.

Searle accepts for the sake of argument that, with a suitable program, the set-up could pass the Turing test. From outside the room, Chinese speakers might think that they were having a conversation with the person in the room. But, in fact, the person in the room (Searle) does not understand Chinese. Searle is just manipulating the symbols according to their form (roughly, their shape) – he has no idea what the symbols mean. The Chinese room is therefore supposed to show that running a computer program can never constitute genuine understanding or thought, as all computers can do is manipulate symbols according to their form.

The general structure of Searle's argument is as follows:

1 Computer programs are purely formal or 'syntactic': roughly, they are sensitive only to the 'shapes' of the symbols they process.
2 Genuine understanding (and, by extension, all thought) is sensitive to the meaning (or 'semantics') of symbols.
3 Form (or syntax) can never constitute, or be sufficient for, meaning (or semantics).
4 Therefore, running a computer program can never be sufficient for understanding or thought.

The core of Searle's argument is premise 3. Premises 1 and 2

are supposed to be uncontroversial, and the defence for premise 3 is provided by the Chinese room thought experiment. (The terms 'syntax' and 'semantics' will be explained in more detail in Chapter 4. For the moment, take them as meaning 'form' and 'meaning' respectively.)

The obvious response to Searle's argument is that the analogy does not work. Searle argues that the computer does not understand Chinese because in the Chinese room *he* does not understand Chinese. But his critics respond that this is not what AI should say. Searle-in-the-room is analogous to only a *part* of the computer, not to the computer itself. The computer itself is analogous to Searle + the room + the rules + the other bits of paper (the data). So, the critics say, Searle is proposing that AI claims that a computer understands because a *part* of it understands: but no-one working in AI would say that. Rather, they would say that the whole room (i.e. the whole computer) understands Chinese.

Searle can't resist poking fun at the idea that a room can understand – but, of course, this is philosophically irrelevant. His serious response to this criticism is this: suppose I *memorise* the whole of the rules and the data. I can then do all the things I did inside the room, except that because I have memorised the rules and the data, I can do it outside the room. But I still don't understand Chinese. So the appeal to the room's understanding does not answer the point.

Some critics object to this by saying that memorising the rules and data is not a trivial task – who is to say that once you have done this you wouldn't understand? They argue that it is failure of imagination on Searle's part that makes him rule out this possibility. (I will return to this below.)

Another way of objecting to Searle here is to say that if Searle had not just memorised the rules and the data, but also started *acting* in the world of Chinese people, then it is plausible that he would, before too long, come to realise what these symbols mean. Suppose that the data concerned a restaurant conversation (in the style of some real AI programs), and Searle was actually a waiter in a Chinese restaurant. He would come to see, for example, that a certain symbol was always associated with requests for fried rice,

another one with requests for shark-fin dumplings, and so on. And this would be the beginning (in some way) of coming to see what they mean.

Searle's objection to this is that the defender of AI has now conceded his point: it is not enough for understanding that a program is running, you need interaction with the world for genuine understanding. But the original idea of AI, he claims, was that running a program was enough *on its own* for understanding. So this response effectively concedes that the main idea behind AI is mistaken.

Strictly speaking, Searle is right here. If you say that, in order to think, you need to interact with the world then you have abandoned the idea that a computer can think *simply because* it is a computer. But notice that this does not mean that computation is not involved in thinking at some level. Someone who has performed the (perhaps practically impossible) task of memorising the rules and the data is still manipulating symbols in a rule-governed or algorithmic way. It's just that he or she needs to interact with the world to give these symbols meaning. ('Interact with the world' is, of course, very vague. Something more will be said about it in Chapter 5.) So Searle's argument does not touch the general idea of cognitive science: the idea that thinking might be performing computations, even though that is not all there is to it. Searle is quite aware of this, and has also provided a separate argument against cognitive science, aspects of which I shall look at in Chapter 4.

What conclusion should we draw about Searle's argument? One point on which I think he is quite correct is his premise 3 in the above argument: syntax is not enough for semantics. That is, symbols do not 'interpret themselves'. This is, in effect, a bald statement of the problem of representation itself. If it were false, then in a sense there would be no problem of representation. Does this mean that there can be no explanation of how symbols mean what they do? Not necessarily – some explanations will be examined in Chapter 5. But we must always be careful that, when we are giving such an explanation, we are not surreptitiously introducing what we are trying to explain (understanding, meaning, semantics, etc.). I take this to be one main lesson of Searle's argument against AI.

However, some philosophers have questioned whether Searle is even entitled to this premise. The eliminative materialists Paul and Patricia Churchland use a physical analogy to illustrate this point. Suppose someone accepted (i) that electricity and magnetism were forces and (ii) that the essential property of light is luminance. Then they might argue (iii) that forces cannot be sufficient for, or cannot constitute, luminance. They may support this by the following thought experiment (the 'Luminous room'). Imagine someone in a dark room waving a magnet around. This will generate electromagnetic waves but, no matter how fast she waves the magnet around, the room will stay dark. The conclusion is drawn that light cannot be electromagnetic radiation.

But light *is* electromagnetic radiation, so what has gone wrong? The Churchlands say that the mistake is in the third premise: forces cannot be sufficient for, or cannot constitute, luminance. This premise is false, and the Luminous room thought experiment cannot establish its truth. Likewise, they claim that the fault in Searle's argument lies in its third premise, the claim that syntax is not sufficient for semantics, and that appeal to the Chinese room cannot establish its truth. For the Churchlands, whether syntax is sufficient for semantics is an empirical, scientific question, and not one that can be settled on the basis of imaginative thought experiments like the Chinese room:

> Goethe found it inconceivable that small particles by themselves could constitute or be sufficient for the objective phenomenon of light. Even in this century, there have been people who found it beyond imagining that inanimate matter by itself, and however organised, could ever constitute or be sufficient for life. Plainly, what people can or cannot imagine often has nothing to do with what is or is not the case, even where the people involved are highly intelligent.[33]

This is a version of the objection that Searle is hamstrung by the limits of what he can imagine. In response, Searle has denied that it is, or could be, an empirical question whether syntax is sufficient for semantics – so the Luminous room is not a good analogy. To understand this response, we need to know a little bit more about

the notions of syntax and semantics, and how they might apply to the mind. This will be one of the aims of Chapter 4.

Conclusion: can a computer think?

So what should we make of AI and the idea of thinking computers? In 1965, one of the pioneers of AI, Herbert Simon, predicted that 'machines will be capable, within twenty years, of doing any work that a man can do'.[34] Almost forty years later, there still seems no chance that this prediction will be fulfilled. Is this a problem-in-principle for AI, or is it just a matter of more time and more money?

Dreyfus and Searle think that it is a problem-in-principle. The upshot of Dreyfus's argument was, at the very least, this: if a computer is going to have *general* intelligence – i.e. be capable of reasoning about any kind of subject matter – then it has to have common-sense knowledge. The issue now for AI is whether common-sense knowledge could be represented in terms of rules and representations. So far, all attempts to do this have failed.[35]

The lesson of Searle's argument, it seems to me, is rather different. Searle's argument itself begs the question against AI by (in effect) just denying its central thesis – that thinking is formal symbol manipulation. But Searle's assumption, nonetheless, seems to me to be correct. I argued that the proper response to Searle's argument is: sure, Searle-in-the-room, or the room alone, cannot understand Chinese. But, if you let the outside world have some impact on the room, meaning or 'semantics' might begin to get a foothold. But, of course, this response concedes that thinking cannot be simply symbol manipulation. Nothing can think simply by being a computer.

However, this does not mean that the idea of computation cannot apply in any way to the mind. For it could be true that nothing can think *simply* by being a computer, and also true that the way *we* think is *partly* by computing. This idea will be discussed in the next chapter.

Further reading

A very good (though technical) introduction to artificial intelligence is S.J. Russell and P. Norvig's *Artificial Intelligence: a Modern Approach* (Englewood Cliffs, NJ: Prentice Hall 1995). The two best philosophical books on the topic of this chapter are John Haugeland's *Artificial Intelligence: the Very Idea* (Cambridge, Mass.: MIT Press 1985) and Jack Copeland's *Artificial Intelligence: a Philosophical Introduction* (Oxford: Blackwell 1993). There are a number of good general books which introduce the central concepts of computing in a clear non-technical way. One of the best is Joseph Weizenbaum's *Computer Power and Human Reason* (Harmondsworth: Penguin 1984), Chapters 2 and 3. Chapter 2 of Roger Penrose's *The Emperor's New Mind* (Oxford: Oxford University Press 1989) gives a very clear exposition of the ideas of an algorithm and a Turing machine, with useful examples. A straightforward introduction to the logical and mathematical basis of computation is given by Clark Glymour, in *Thinking Things Through* (Cambridge, Mass.: MIT Press 1992), Chapters 12 and 13. Hubert Dreyfus's book has been reprinted, with a new introduction, as *What Computers* Still *Can't Do* (Cambridge, Mass.: MIT Press 1992). Searle's famous critique of AI can be found in his book *Minds, Brains and Science* (Harmondsworth: Penguin 1984), and also in an article which preceded the book, 'Minds, brains and programs', which is reprinted in Margaret Boden's useful anthology *The Philosophy of Artificial Intelligence* (Oxford: Oxford University Press 1990). This also contains Turing's famous paper 'Computing machinery and intelligence' and an important paper by Dennett on the frame problem. Searle's article, along with some interesting articles by some of the founders of AI, is also reprinted in John Haugeland's anthology *Mind Design* (Cambridge, Mass.: MIT Press 1981; 2nd edn, substantially revised, 1997), which includes a fine introduction by Haugeland.