JUNCTION SIX

The footsteps fade behind you, the ticking of massed arrays of grandfather clocks disappears below the threshold of human hearing. The sounds behind the walls cease. You stop to catch your breath, and look down at your feet. They no longer leave prints on the floor, but in your mind you can see the pattern: left, right, left, right, a double frieze with glide reflection symmetry, laid down in space by the passage of time – and the passage of your feet.

You will always *be able to see those patterns, now, whenever you so choose. You will never again watch an animal loping past without looking at the order in which it moves its feet. Everything that moves will leave a luminous trail in your imagination.*

But you are not yet out of the maze. Where next?

You have a choice: pass through a low archway to your left, or continue straight on to a corner that turns right.

You continue straight ahead, and turn the corner.

You are standing on a tiny railway platform. The rails run off along the passage, disappearing round the next bend.

At the platform is a train, a single engine. There is no driver, but there is a seat. It looks extremely comfortable. The only controls are a button labelled START and another labelled STOP. You feel that operating such a machine is within your competence, and you are tired from running.

You climb aboard, sit down, and push START.

The train chugs slowly away, down the track, and round the first bend. It comes to a set of points, and takes the left fork. Another track joins from the right, and the points seem set against you! You brace yourself for the crash, but as your train passes through the points they reset themselves automatically to let it pass unharmed. There are more points – hundreds, thousands ... The train trundles on through unmarked tunnels, continually passing through sets of points. Some remain unchanged by its passage, some switch to another setting.

You never meet another train, never pass through a station. You think about pushing the STOP button, but you have no idea how you would ever get out of the maze of tracks. There is no REVERSE button – and even if there were, the points have been reset by your passing.

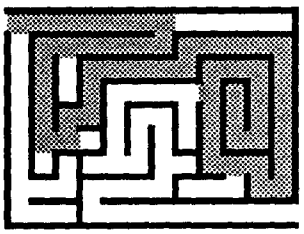
After a time you wonder whether you're going round in circles. You strain to see some distinguishing feature, a mark on the wall of the tunnel, a chip out of a sleeper. Anything to reassure you that the train isn't caught in an endless loop.

You desperately want to STOP, and think what to do next. You promise yourself that never again will you leap before you've looked.

You wonder whether the train will ever reach a station.

Passage Six

TURING'S TRAIN SET



We are living in an age when computers perpetually seem to get faster, their memories get bigger, their graphics get better, and their price gets smaller. The rate of progress is so great that it is difficult to imagine that there could be any limitations to computers. Well, there might be limitations such as intelligence or consciousness, but it would be surprising if there were limits to their ability to calculate.

However, such limits do exist.

Ironically, these limits were discovered as a result of an attempt to prove, once and for all, that there are absolutely no limits to mathematics.

In 1900 there was a big international mathematical conference, the International Congress of Mathematicians. The event continues to be held every four years, and nowadays attracts about five thousand participants. In those days the numbers were smaller, but it was still *the* major event in the mathematician's calendar. At the 1900 congress, to mark the new century☛, David Hilbert – a German mathematician generally regarded as the leading mathematician of his time – gave a talk in which he outlined twenty-three major unsolved problems. He also gave a talk on the radio, to a general audience of non-mathematicians.

At that time, by the way – indeed until quite recently – it was entirely normal for the great mathematicians and scientists to popularise their subject. Felix Klein wrote several books on recreational mathematics, and

Hilbert also wrote a popular geometry book with the aid of a co-author. Henri Poincaré wrote popular books on the philosophy and methodology of science. These three were the leading trio of mathematicians at the turn of the nineteenth century. It is only during our present century that science popularisation has somehow become out of bounds for 'serious' academics. Fortunately, that attitude is beginning to disappear again.

Radio was a very new technology in Hilbert's day, and sound recording was in its infancy; nevertheless, a scratchy, noisy recording of Hilbert's talk, made on wire coated in magnetic material, survives. At the end, we can hear Hilbert coming to the climax of his talk, on a ringing note of optimism: '*We must know: we shall know.*' Hilbert, having assembled the great unsolved mathematical questions of his time, was affirming his belief that eventually they would all be solved. Indeed, he believed much more: he thought that every question in mathematics must have an answer, and that there ought to be a uniform method for finding it.

He even laid down a programme for discovering such a method – a so-called decision procedure for mathematics. Given such a procedure, Hilbert would then be ready to proceed to the second phase of his programme, which was to prove, beyond any shadow of doubt, that mathematics can never contradict itself. That is, valid mathematical deductions will never lead to a proof that some statement is true, and also to a proof that the same statement is false.

Nobody was really worried that this could happen, but it was a serious philosophical problem. Mathematics is a construct of the human mind: it may *model* reality, but it's not the *same* as reality. Mathematics makes use of idealisations, such as 'infinity', that do not have evident counterparts in the real world. If those idealisations had overreached themselves, then mathematics might contain a hidden, fatal flaw. If such a contradiction were ever to be found, mathematics would collapse in ruins. The reason is that, according to the rules of logic, such a contradiction would imply that *all* statements are true (and also false). For instance, $2+2=5$ would be true, and so would $2+2=6$. Fermat's Last Theorem☛ would be true, and Fermat's Last Theorem would be false. The angles of a triangle would not add up to 180º, even in Euclidean geometry. Pythagoras's theorem would be valid for cubes instead of squares. The circumference of a circle would equal its radius ...

The whole exercise would cease to have any meaning.

Probably something could, eventually, be salvaged. A huge effort would be needed to clamber through the rubble, trying to find out what went wrong and how much of the structure could be saved by more careful rebuilding.

If that sounds far-fetched, you should bear in mind that in the mid-1900s there was a major crisis in mathematics, when certain theorems, apparently correctly proved, contradicted other equally plausible theorems. The problems arose in an area known as Fourier analysis☛, in which complicated waveforms are represented as combinations of much simpler trigonometric 'sine' and 'cosine' curves. In those cases it eventually turned out that the mathematicians had been a bit too sloppy, jumping to unwarranted conclusions in various of the more subtle parts of their topic. It taught them to be very cautious indeed – just one of the several reasons why the true mathematician is never happy with 'experimental' evidence, but requires watertight logical proofs.

Extremely watertight.

Hilbert, in fact, was after the biggest piece of watertight logic of them all: a guarantee that there would *never* be another crisis – as long as mathematicians didn't make silly mistakes.

It seemed a reasonable idea at the time. After all, mathematics *feels* perfectly consistent, and all of the evidence is that it has never yet contradicted itself. The equation $2+2=5$ doesn't exactly ring true. If you've got two pigs in a sty, and you put two more in, you don't – at least not immediately – have *five* fat porkers in the sty.

However, as I've just said, 'experimental' evidence is not the same as proof. So Hilbert devised a line of attack in which mathematics was treated as a purely formal game in which symbols are manipulated according to a system of precisely stated rules. All you had to do then was show that no amount of valid manipulation could ever produce the string of symbols $0=1$. It's not so different from proving that in a game of chess there can never be eleven knights of the same colour on the board at once. There *can* be ten: pawns that reach the far side of the board can be promoted to any piece. Queens are usual, but knights are legal. You start with two knights and eight pawns, so if you trade all the pawns for knights ... It will never happen in a sensible game, but it's possible. Eleven knights isn't – but can you prove that?

Mathematics is a *much* more subtle game than chess. So subtle that, within just a few years, it was Hilbert's programme that collapsed in ruins. Not because somebody found a contradiction within mathematics, though. Because a young logician called Kurt Gödel showed that Hilbert's programme could never achieve its aims. And, fatally, neither could any similar programme. You could tinker with Hilbert's approach for as long as you liked, but you'd never be able to fix it up.

Even if mathematics *is* consistent, said Gödel, we can never be certain

that it is. Indeed, if anyone ever managed to write down a proof that mathematics is consistent, then we would immediately know that it isn't☛.

Gödel's dramatic discovery led to a new understanding of the limits to mathematics, and to formal logic. About thirty years later, in the early days of the computer, before much of the hardware existed and when most computational aids were made from cogwheels or electrical relays, very similar ideas turned out to be fundamental to the theory of computation. In fact, the whole area is much more easily understood if it is viewed from the standpoint of computation.

LOGICAL PARADOXES

At the heart of all mathematics, and of all computation, lies logic. Logic is the glue that holds mathematics together, letting us start from known facts and end up with new ones. Computers are just huge logic boxes, slavishly following prescribed rules to reach desired ends.

Logic is the study of valid deductions. Its raw materials are not numbers, but *statements*. Statements are assertions that are *either* true or false – statements like

dogs bark

or

the United States is a continent.

Here the first statement is true, the second false. However, logic does not concern itself with the factual truth or falsity of specific statements – those fall into other domains, here science and geography. Logic is about the correctness, or not, of deductions of some statements from others. Such as

dogs bark
barking is annoying
therefore dogs are annoying.

According to conventional logic, and leaving aside a few quibbles (such as 'well, *some* dogs don't bark'), this particular deduction is considered to be correct. Indeed, as a *deduction* it is correct even if the individual statements involved might be false – as the second is for many humans. To a logician, the deduction

$$2+2=3,$$
$$3=5,$$
$$\textit{therefore } 2+2=5$$

is entirely unobjectionable; and so is

$$2+2=3,$$
$$3=4,$$
$$\textit{therefore } 2+2=4.$$

In both cases, the logic is impeccable. However, various of the component statements are false, so we are not entitled to deduce that the final conclusion is *true*. As it happens, it is not true in the first case, but it is in the second; however, none of this is related to the validity of the deduction.

Most of the statements that arise in logic are relatively harmless things. One category, however, causes endless problems. This is the category of *self-referential* statements, ones that refer to themselves. The simplest example is

This statement is false.

Let's see: is it true, or is it false? Well, if it's true, then we immediately deduce that it's false. On the other hand, if it's false, then of course it's true. It is a self-contradictory statement, otherwise known as a *paradox*.

The great classical instance of this statement is the alleged claim, by Epimenides the Cretan, that 'all Cretans are liars'. If 'liar' means '*always* tells untruths' then Epimenides is lying when he says he's a liar, so he's truthful, so he *is* a liar after all, so ... However, if 'liar' means '*sometimes* tells untruths', as it is usually interpreted, then there is no contradiction. On this occasion he happens to be telling the truth, but maybe on another occasion he won't.

Curiously, no such paradox arises from the statement.

This statement is true.

If it's true, then it's true; if it's false, then it's false. So being self-referential *alone* is not enough to cause trouble.

There are many variants on this 'liar paradox'. One is

There are three mistokes in this semtence.

OK, there are two obvious *spelling* mistakes: 'mist*o*kes' and 'se*m*tence'. But what is the third mistake? Could it be (aha!) that there are actually only

two mistakes? Yes, but if that's so, then there *are* three mistakes after all, so there are only two, so ...

Another variant is a card, on one side of which is the message

> THE STATEMENT
> ON THE OTHER SIDE OF THIS CARD
> IS TRUE

and on the back is another message:

> THE STATEMENT
> ON THE OTHER SIDE OF THIS CARD
> IS FALSE

(Think about it.)

A slightly different kind of logical paradox is the so-called Richard paradox. Some English sentences define numbers – examples are

> The number of planets in the solar system.
> The sum of two and two.
> The number of words in this statement.

Some don't – for instance

> The species to which Eeyore belongs.

Presumably, every statement either defines a number, or it doesn't. So does

> The smallest number that cannot be defined in a sentence of less than sixteen words

define a number, or not? Obviously it does: there are only finitely many words in the English language, there are finitely many sentences with less than sixteen words, so among the numbers they define (when they do) there is a largest one. Add one to that, and you get the number defined by the above sentence.

Fine. So let's count how many words the above sentence has.

Fifteen.

Hmm. We have just defined the smallest number that *cannot* be defined in a sentence of less than sixteen words, using a sentence of less than sixteen words.

Oops!

The classical logicians were forced to grapple with these paradoxes, and find ways to rule them out, otherwise the whole edifice of logic would have come tumbling down.

One way out is simple – and in a sense, is forced upon us. I said that a statement is an assertion that is neither true nor false. For these self-referential assertions, truth or falsity seems self-contradictory. Conclusion: *they are not statements* – not within the meaning of the act. The only problem with this approach is that we can't easily (if at all) tell which sentences *are* statements. Is '2 + 2 = 4' a statement? Well, if Hilbert is right and mathematics doesn't contradict itself, then the answer is 'yes' – but if he's wrong, then the answer is 'no'.

You can't build a sound logical foundation for mathematics if your criterion for what is admissible involves making that kind of choice.

More recently, several unorthodox logical systems have been devised. In one of them, due to Patrick Grim and Gary Mar, the truth of a statement is *dynamic* – it varies over time. So 'this statement is false' is alternately true and false, since each implies the other. In another system, called fuzzy logic, the statement is a half-truth. Literally: it is half true and half false. Despite its name, fuzzy logic is a perfectly precise area of mathematics, and it is the basis of a billion-dollar industry, used in things like washing machines, air conditioners, camcorders, and so forth. Precision is still present because the degree of truth is *precisely* 50%, not 51% or 49%. But rather than developing these esoteric areas of modern logic, we shall make use of the paradoxical nature of certain classical statements to discover limits to mathematical proof and limits to computation.

ALGORITHMS

Nowadays there are thousands of different designs of computer, but they all operate along the same basic lines. They can be thought of as machines that manipulate symbols according to specific rules. The symbols in effect reduce to two, the 'binary' digits 0 and 1. In effect, a computer is a lot of switches with 1 meaning 'on' and 0 'off'. It works by manipulating these switches according to fixed rules.

Symbols more complex than 0 and 1 can be created by stringing lots of 0's and 1's together. The same string of 0's and 1's can have many different interpretations, according to which rules are currently in operation. In arithmetical calculations with whole numbers, a string such as

1001011 is interpreted as the number

$$64+8+2+1=75.$$

When manipulating text, it may instead be interpreted as 'ASCII code', in which case it stands for the letter K. The computer doesn't 'know' any of this: it just follows its rules – so it does no serious harm to head towards more familiar ground and think of the symbols manipulated by the computer as being the ordinary numbers 0, 1, 2, 3, 4, and so on. For simplicity, I'll do just that from now on.

The rules take the form of a *program*, a list of instructions that tells you what manipulations to make on the symbols. Real programs are stored in the computer as sequences of symbols 0 and 1 as well, with yet another interpretation – as instructions. Here we can be less formal, and just write down the instructions in words, perhaps like this:

1. Let FRED equal 7.
2. Let ALICE equal 3.
3. Let TOTAL = 0.
4. Add FRED to TOTAL.
5. Subtract 1 from ALICE.
6. If ALICE is bigger than zero, go to step 4.
7. Stop.

Here FRED, ALICE, and TOTAL are names for *variables*, things that can take on numerical values. The idea is to follow the instructions in numerical order, except when an instruction such as number 6 re-routes you.

What does this program do? Let's work through the whole thing, step by step:

1. FRED becomes 7.
2. ALICE becomes 3.
3. TOTAL becomes 0.
4. TOTAL becomes $0+7=7$.
5. ALICE becomes $3-1=2$.
6. ALICE is bigger than zero, so go to step 4.
4. TOTAL becomes $7+7=14$.
5. ALICE becomes $2-1=1$.
6. ALICE is bigger than zero, so go to step 4.
4. TOTAL becomes $14+7=21$.
5. ALICE becomes $1-1=0$.

6. ALICE is not bigger than zero, so continue to step 7.
7. *Stop.*

We've ended up with TOTAL at 21, ALICE reduced to 0, and FRED still 7. So what this program does is multiply FRED by ALICE and put the result into TOTAL. You can easily convince yourself that with other positive whole numbers for FRED and ALICE the result is again to put the product FRED × ALICE in TOTAL.

This is a typical, albeit very simple and crude, program. Its main feature is a 'loop' from instruction 6 back to step 4, which is carried out when some logical condition (here 'ALICE is bigger than zero') is true, and ignored if that condition is false. It also has the very important instruction *stop*. You can't be sure what a program calculates unless it signals to you that the calculation has *finished.*

Programs don't have to stop. In fact, if you change step 5 to

5a. Add 1 to ALICE,

then the program never escapes from the loop. It keeps going round the steps 4, 5, 6 in turn for ever, with TOTAL increasing by FRED every time, ALICE by 1 every time, and FRED staying exactly the same throughout.

It is relatively easy to spot this particular 'infinite loop' and see that the amended program with instruction 5a never stops. But in the early days of computing it became clear that it's not always so easy to decide whether a program will eventually terminate. Then Alan Turing, a British mathematician with a penchant for logic and one of the two founding fathers of the science of computation (the other being John Von Neumann) discovered that there is no foolproof method of finding out. Not just that nobody had yet found one: that nobody could *ever* find one.

TURING MACHINES

Turing's first step was to find a simple, conceptually 'clean' model of the computational process. Real computers are too complicated; their design involves all sorts of engineering considerations that get in the way of understanding what they do. Indeed, there are thousands of different makes of computers, and dozens, perhaps hundreds, of different programming languages. However, the differences are not fundamental: they are tactical, so to speak, rather than strategic. In particular, all computers can in principle compute the same things. Indeed, were this not so, some

computers would be obviously 'better' than others – able to compute things that others could not – and these would come to dominate the market.

'All computers can in principle compute the same things.' It's an easy thing to say, and it sounds plausible – but is it *true*? How can we prove that two computers can compute the same things? Not by running lots of programs – that's experimental evidence, not proof. No, what we need to do is show that each computer can *simulate* the other (or *emulate*, another term used for the same idea). That is, computer A can run a program which effectively turns it into computer B – in the sense that any program that runs on B can be run in the simulation and yields the same answers – and in the same manner computer B can run a program which effectively turns it into computer A. Simulations normally run more slowly than the thing they are simulating, so for practical purposes A may well be superior to B, say. But if speed is unimportant, then anything A can do, so can B, and conversely.

The way to write a simulation program is to start from the rules of operation of computer B, and program A to answer the question 'in the current circumstances, what would B do, and what would be the result?' Then A can follow, step by step, everything that B would do when running a complex program with large numbers of instructions.

Turing employed this idea, but he took it one stage further. The more complicated your computer's instruction list, the more difficult it is for a mathematical method to analyse what it does. There are just more possibilities to worry about. Mathematicians like to keep things simple – honest they do, even though it may not always appear that way. So Turing developed an extremely simple model of the computational process, known as a Turing machine. It is a 'machine' only in conceptual terms – you *could* build one, with today's technology, but it would be far too cumbersome to use, so nobody bothers to. A Turing machine, slow and cumbersome as it is, is good enough to simulate any existing computer. And the way a Turing machine 'works' is so simple that any computer can easily simulate a Turing machine. In short, a Turing machine and a computer can compute exactly the same things. This is another way to see that all computers can compute the same things: since they all compute exactly the same things as a Turing machine, they obviously all compute the same things as one another☛.

The things they compute, by the way, are called *computable functions*. A *function* is a rule for turning input data (a sequence of 0's and 1's) into output data (another sequence of 0's and 1's); an example is shown in Table 11. The rule is computable if it can be carried out by a Turing

Table 11

input	rule	output
1001110100	Change 0 to 1 and 1 to 0	0110001011
1001110100	Select every second digit	10100
1001110100	Copy the sequence backwards	0010111001
1001110100	Collapse every sequence of successive 0's to a single 0	10111010

machine – or any other computer – by obeying an *algorithm*: a precisely defined sequence of instructions. In short, a program. An algorithm must also be equipped with a guarantee that it will eventually finish the calculation and report its answer.

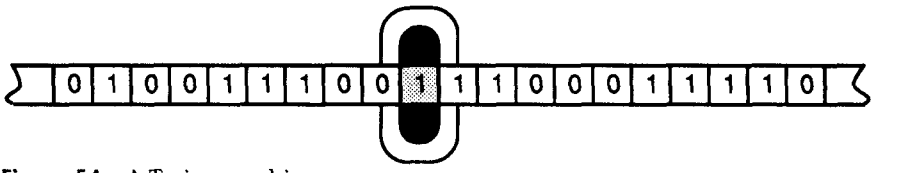


Figure 54 A Turing machine.

For a mental image of a Turing machine, imagine a long tape divided into consecutive square cells (Figure 54). Each cell contains a digit, 0 or 1. You can either use an infinite tape, or if you don't like infinities you just have to be prepared to add some more cells to the tape whenever you need them. The tape travels through a box containing apparatus that can read the digit in a given cell, write another digit in its place if necessary, and move the tape in accordance with various rules. The box itself can be in any of a finite set of internal *states*. For each combination of its state and the digit on the tape immediately beneath it, the box must obey a small list of instructions, like this:

- Leave the current tape digit alone/change it.
- Then move one space left/right.
- Then go into some specified internal state ready for the next step.

Alternatively, the instruction can be just

- Stop

and the computation then finishes.

Here's an example of a typical list of rules, with three internal states, which I'll refer to as states 1, 2, and 3.

- *State 1, Digit 0*: Change digit, move left, go to state 2.

- *State 1, Digit 1*: Stop.
- *State 2, Digit 0*: Leave digit, move right, go to state 3.
- *State 2, Digit 1*: Change digit, move right, go to state 2.
- *State 3, Digit 0*: Change digit, move right, go to state 1.
- *State 3, Digit 1*: Leave digit, move left, go to state 2.

Here, 'move left' and 'move right' refer to the cell of the tape that is in the box. The tape itself moves the other way. In practice, it often helps to think of the box as moving along a fixed tape, in the specified direction.

Suppose we start this machine in state 2, poised over a tape with the digits ... 1001 ... which I'll represent like this:

tape	1	0	0	1
cell	↑			
state	2.			

What does it do?

To find out, we just work through the rules. Tape digits are in bold when they get changed.

- *State 2, Digit 1*: Change digit, move right, go to state 2.

tape	**0**	0	0	1
cell		↑		
state		2.		

- *State 2, Digit 0*: Leave digit, move right, go to state 3.

tape	0	0	0	1
cell			↑	
state			3.	

- *State 3, Digit 0*: Change digit, move right, go to state 1

tape	0	1	**1**	1
cell				↑
state				1.

- *State 1, Digit 1*: Stop.

So here the program takes the *input* string 1001, turns it into the *output* string 0111, and stops.

This is a short and not terribly interesting computation – but the point is that the machine carries out a definite series of actions on any input string. Sensible programs that compute useful things generally need a lot more states than three. Now, if there are T internal states then, in addition to 'stop' there are $4T$ possible instructions – two choices for the first, times two for the second, times T for the third. You get one Turing machine for each assignment of rules to states.

The digits on the tape provide the computer's *input*; the list of which instructions to perform for which internal state of the box form the *program*, and the list of digits on the tape when the computation stops is the *output*.

The program itself can be written as a sequence of digits 0 and 1, by using some appropriate code – say

000 print 0
001 print 1
010 move right
011 move left
100 stop.

and so on. This is interesting, because it opens up the possibility of *interpreting* a program as data, so that another program can operate on it.

Programs can operate on programs.

THE HALTING PROBLEM

Amazingly, Turing established that these apparently simple devices can carry out any algorithm whatsoever. More than that: there exists a *universal Turing machine*, which can carry out any computation that any other Turing machine can carry out. This one machine can do everything – given the right input.

It has a program, which we'll symbolise as U.

For any program P, call the corresponding encoded sequence of 0's and 1's code(P). The input data for U take the form code(P) + d – that is, the string of 0's and 1's that makes up code(P), followed by the string d. Here P is a program and d is the string of input data for P. The program U scans the first part of its data, the string of the form code(P), to find out what P would do if it were given data d. Then U carries out the same action that P would on the 'd' bit of its own input.

In other words, U *simulates* the action of P on data d.

An important feature of any practical algorithm is that it should eventually *stop* – otherwise you don't know whether it's finished working out the 'answer', the output string. The computation in the previous section does stop, but it's also easy to find programs that don't stop. The easiest way to do this is not to have any 'stop' instruction in the list, but there are more subtle ways. In particular, some programs stop for some inputs but not for others.

Is there any way to tell whether a given program will eventually stop

with a given input? Suppose you've sat watching the Turing machine churn away, and after ten million steps it hasn't stopped yet. Is there some way you can check whether it's going to stop at all – or must you just keep waiting? Turing named this question the halting problem: for a given machine, can you test which inputs lead to a computation that stops, and which don't? In the spirit of the enterprise, the test ought itself to be an algorithm – so it should be possible to carry it out on some appropriate Turing machine.

Is there a halting algorithm?

In 1936, Turing was thinking about this question, and he wondered what would happen if you applied a putative halting algorithm to a universal Turing machine. More precisely, let U be a universal Turing machine, let its input data be d, and consider the following list of instructions:

- Check whether d is equal to code(D), where D is a program. If not, go back to the start and repeat.
- If d = code(D), double up the data string to give $d+d$.
- If U stops with input data $d+d$, repeat this step indefinitely.
- If U does *not* stop with input data $d+d$, then stop.

If it is possible to solve the halting problem algorithmically, then the third and fourth steps can be carried out algorithmically, so this entire computation can be turned into a program H for a Turing machine.

The construction of H includes exactly two places where the machine may get 'hung up' for ever and not stop. It can do so at the first stage, if d is not the code of a program. It can get hung up at the third stage, if U *does* stop with input $d+d$. These are the only places it can get hung up.

OK, fine. Since H is a program, it has a code, say h = code(H). Now comes Turing's really sneaky question:

Does H halt with input data h?

It certainly doesn't get hung up at the first stage, because the input data h *is* the code of a program – namely, H. The only other place it can get hung up for ever is the third step, and it gets past this provided U does not halt with input $h+h$. That is:

H halts with input data h if and only if U does not halt with input data $h+h$.

So far so good. Now think about how U simulates a program P. It starts with input code(P) + d, and then behaves just like P with input data d. That is:

P halts with input data d if and only if U halts with input data code(P) + d.

This also looks entirely reasonable. But suppose we now let $P = H$, so that code(P) = code(H) = h; and let's also put $d = h$. Then the previous statement becomes

> H halts with input data h if and only if U halts with input data $h+h$.

But this exactly contradicts the previous statement but one.

Turing concluded that it is *not* possible to solve the halting problem algorithmically. In brief, his argument is this: if we could solve the halting problem algorithmically, then we could construct a Turing machine that halts if and only if it doesn't.

Turing's proof has a very similar logical structure to the card that I mentioned, whose two sides read

> THE STATEMENT
> ON THE OTHER SIDE OF THIS CARD
> IS TRUE

> THE STATEMENT
> ON THE OTHER SIDE OF THIS CARD
> IS FALSE

In fact, we can summarise Turing's proof as a card that says on one side

> THIS MACHINE HALTS
> IF AND ONLY IF
> THE MACHINE ON THE OTHER SIDE
> HALTS

and on the other side

> THIS MACHINE HALTS
> IF AND ONLY IF
> THE MACHINE ON THE OTHER SIDE
> DOES NOT HALT

The construction of such self-contradictory machines depends on being able to solve the halting problem algorithmically – so Turing deduced that such a solution cannot exist.

THE INTELLIGENT SUBWAY

Turing went on to consider the vexed question of machine intelligence, inventing his famous 'Turing test'. Put a computer in a room and ask it questions from a remote terminal. If you can't tell the difference between its responses and those of an intelligent human being, said Turing, then to all intents and purposes the machine is intelligent.

I'm not going to discuss the merits or otherwise of this proposal, which is often misunderstood. Instead, let me phrase the idea differently. Suppose that the operation of the universe is algorithmic – that is, the 'laws of nature' are a system of mathematical rules, and the universe simply follows those rules from given initial conditions. Then, in effect, the universe is a Turing machine. Its program is the laws of nature, its data are the initial conditions. If an intelligent creature inhabits that universe, then it, too, obeys the laws – so it, too, is (more accurately, can be simulated precisely by) a Turing machine.

In a universe that operates algorithmically, 'strong artificial intelligence' – the construction of truly intelligent machines – *must* be possible in principle. Not necessarily in practice, though, because it may not be possible to encapsulate the rules and the initial data into a sufficiently compact device. For instance, if the 'machine' were the entire universe, we wouldn't accept it as the solution. Indeed, *any* universal Turing machine would be (potentially) intelligent, needing only the right 'intelligence program' and 'intelligence data'.

The real interest in this idea is that is pushes the question of artificial intelligence away from the machine's hardware, and into its program and data.

It also shows that, if we equate intelligence with carrying out some (no doubt complex) algorithm, then a sufficiently complex subway system could become intelligent. It would 'think' rather *s-l-o-w-l-y* ... but it would still be able to 'think' – carry out the intelligence algorithm.

Be that as it may, a subway can certainly compute. Compute what? Anything that a computer can.

It sounds like science fiction. Offhand, I can only think of two science-fictional subway stories. One is A.J. Deutsch's *A Subway Named Möbius*, in which a subway system becomes so complicated that the trains on it cease to have well-defined locations – and so disappear. The other is Colin Kapp's *The Subways of Tazoo*, in his 'unorthodox engineers' series, which chronicles the demise of an alien race that becomes over-reliant on a single kind of power source. But what I have in mind is more like David

Brin's *Earth*, in which the global computer network becomes so interconnected that it acquires enough intelligence to save humanity from a rogue black hole.

But I'm straying from the point.

In 1994, the journal *Eureka*, published by Cambridge University's student mathematical society the Archimedeans, printed a fascinating article by Adam Chalcraft and Michael Greene. It was about the computational abilities of train sets – toy train sets, with rails and points and little men in out-of-date railroad uniforms. They showed that a train set can compute. And whatever a train set can do, a subway surely can.

Agreed, a train set is not quite your Massively Parallel SuperProcessor. Not in speed. But in theoretical computing ability, the two are equally powerful. The theory of Turing machines tells us that any programmable digital computer can simulate any other, given enough memory. Now a computer is just a huge switching circuit with adaptable switches – and trains can switch tracks using points. Suppose, said Chalcraft and Greene, you've got a big enough stock of straight and curved track, bridges, and various kinds of points. However, you've got only one engine and *no* rolling stock. What computations can you perform if you set up the right track layout?

At first it's hard to see how a train can compute at all: it's just a thing on wheels that moves along the rails. But electrons are just things that move along wires, and computers compute using them. The computational aspect is a matter of interpretation in both cases; the underlying mechanisms are much the same. The idea is to encode an input as a lot of 0's and 1's corresponding to the settings of various points in the train layout. Then you run the train along the tracks, and of course those setting change, which in turn alters the path of the train. Eventually the train is side-tracked into a line leading to a terminal – the program 'stops' – and you read off the output from the settings of the same collection of points.

Let's start with the simplest switching unit, known as *lazy points*. They're a Y-shaped piece of track. A train entering the Y from below runs up the upright and out of whichever arm of the Y the points are set for, leaving them set as they were to begin with. However, a train that enters from one of the arms will – if necessary – reset the points so that they connect that arm to the upright, and exit via the upright. But no trains ever come in down one arm and go out via the other (Figure 55(a)). Lazy points have two states, depending on which arm is connected to the upright: let me call them *left* (*L*) and *right* (*R*).

Layouts whose only active components are lazy points have very limited computational ability. Assume that you use a finite number of bits of track,

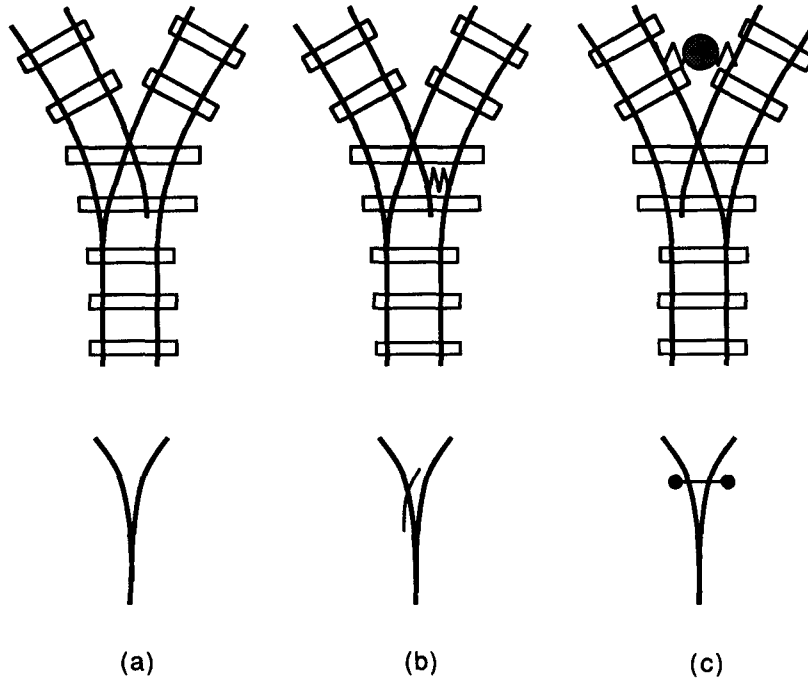


Figure 55 Three types of points (above) and their 'circuit symbols' (below): (a) lazy points, (b) sprung points, (c) flip-flop.

fix some layout, and let the train run through it. The only things that change are the points, and these can be in only two states, *L* or *R*. If there are *n* lazy points, then the layout has at most 2^n states. So eventually the motion of the train must fall into a repeating cycle, when it starts to run through a state of the layout that it has run through in the past. Discard any bits of track that aren't traversed during the cycle. Amazingly, there are only two topologically distinct types of layout that remain. One is just a closed loop with no points, and thus computes absolutely nothing at all. The other is a figure-of-eight arrangements with two points, in which the arms of each point are joined in a loop and the uprights are run together.

Now, suppose that in the second case the lazy points both start out in state *R*, with the train in between them. If you trace its path and the effect on the points you'll find that it cycles the settings in the sequence ***RLRLLRLR***, repeated for ever, where the bold symbols indicate one set of points and the plain letters indicate the other one. So it's like a computer that can compute only the terms of the sequence 10100101, repeated for ever.

For simplicity, I'm letting each *R* represent 1 and each *L* represent 0. I'll clarify how to interpret the train's motion as a computation later, when we get to a more representative layout.

We can now see that we need some more complicated switching gear if we're going to build a computer. The next type of point is a *sprung* one (Figure 55(b)). It's like a lazy point, except that any train entering along the upright of the Y always leaves by the same arm – say the left arm. It doesn't matter which arm you use, because by using bridges you can connect the rest of the layout to either arm, so the right-armed sprung point doesn't do anything new. However, it does simplify layouts to some extent, so I'll allow either arm to be sprung.

The third kind of point is a *flip-flop* (Figure 55(c)) – not standard railroad jargon, but it'll do. With a flip-flop, the train always enters along the upright of the Y, and exits through the left and right arms alternately. Chalcraft and Greene showed that, given these components, we can build a computer – a universal Turing machine.

Here's how.

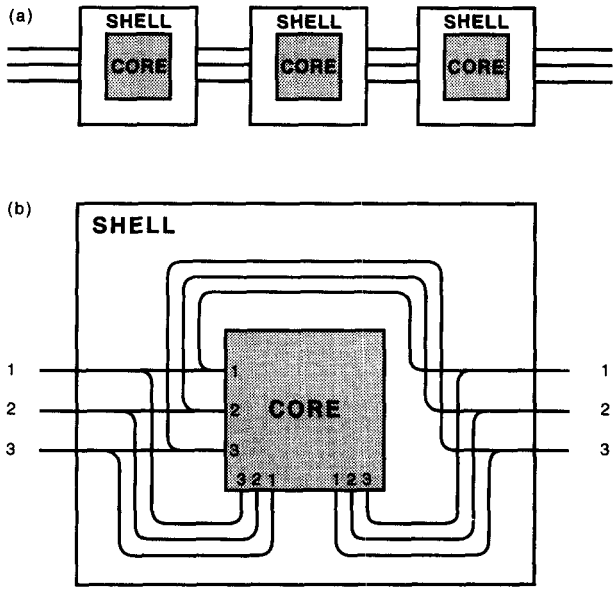


Figure 56 (a) Replacing the tape of a Turing machine by a series of identical track layouts. (b) Each square of the tape consist of a shell, which ensures that trains entering from either direction are treated alike, and a core, which simulates the rules for the Turing machine.

It helps to break the problem down into a series of stages. Recall that a Turing machine employs a box, through which its tape travels. We need to find a train layout that can play the role of the box. Then we just plug an enormous number of these boxes into each other, side by side, to represent the whole tape (Figure 56(a)). Each box will have T tracks coming in from the left and T going out at the right – one for each internal state. Instead of the tape moving through the box, the train moves along the row of boxes. You can tell which 'square' of the 'tape' is being worked on by which box the train is in. But what do we put in the box?

I'll explain how the box is designed in stages. The train tracks are used as both input and output lines, so the box doesn't 'remember' which direction the trains came in from. So it can be set up as an outer shell that feeds trains the same way into an inner core from both input lines, and directs them out again according to the Turing program (Figure 56(b)). Then we can ignore the outer shell, and just concentrate on the design of the core.

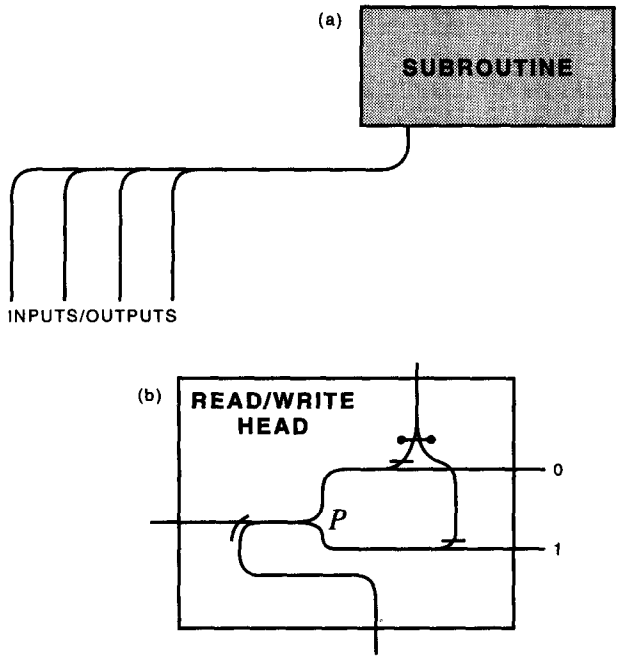


Figure 57 (a) Layout for calling a subroutine: trains enter through lazy points, and exit along the track. (b) Design of a read/write head; note the presence of a flip-flop.

We'll need some gadgets that computer scientists call 'subroutines'. A subroutine is a part of a program that can be used repeatedly by 'calling' it from any other part. You can build complex programs by stringing subroutines together. We can set up a subroutine by hooking up a self-contained sublayout to a whole series of lazy points. Then the train comes in, setting the points as it does so, and wanders round the sublayout until it's carried out whatever subroutine that sublayout computes. Finally it exits by the same track that it came in on, because of the way it set the points on entry. Using one lazy point for each input line, the trains can all be made to enter from the left, carry out the subroutine, and exit to the right along the same track they entered from (Figure 57(a)). We also need one more piece of gadgetry, a *read/write head* (Figure 57(b)). If a train comes in from the left it exits along line 0 or line 1, depending on the

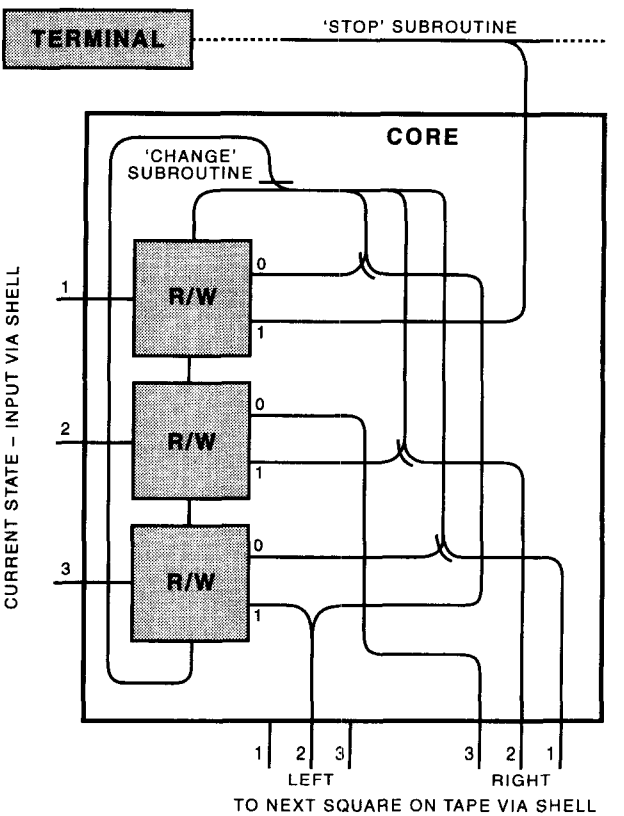


Figure 58 The design of the core circuit: the train enters from the left, obeys the appropriate Turing machine rule, and exits at the bottom.

digit at the 'current' point of the tape; and if a train comes in from above it swaps the 0 and the 1. To achieve this, the lazy point P is set to redirect the train along output lines 0 or 1 according to the digit on the 'tape' at that square. The flip-flop is set so that the first train entering from the top switches P to the other position.

Having got all these bits and pieces, you build the inner core of the box as in Figure 58. You may need some bridges to avoid the tracks crossing, but we can ignore that. The core consists of a parallel set of read/write heads, one for each internal state of the core. The output lines 0 and 1 lead to one of the outputs of the core, or to a lazy point that diverts the train into a subroutine that changes the state of that 'square' on the tape, or to a 'stop' subroutine that guides the train into a single terminal.

In my earlier example, one of the rules is '*State 1, Digit 0*: Change digit, move left, go to state 2'. How does that work?

The square being in state 2 means that the train enters from the side along line 2. This state is set by the output of the *previous* square, which directs the train on to line 2 when it exits. The digit 'written' on that square is 0: that just means that we've set all the lazy points in the read/write heads to 0. So the train comes into the second read/write head, leaves along line 0, and runs into a set of lazy points. These direct it into the 'change' subroutine. It runs vertically downwards, through all the read/write heads, flipping their states from 0 to 1. So the digit 'written' on that square now reads 1, not 0. The train goes back up the vertical track to the left of the heads, exists from the subroutine back on to its original track, and then comes out of the core on the output line *2 left*, which effectively moves the train into the box to the left, in state 2, as required.

Suppose we look at the rule '*State 2, Digit 0*: Leave digit, move right, go to state 3'. The train comes in along line 2 and exits the read/write head along line 0, which leads directly to exit *3 right*. And it never goes anywhere near the subroutine loop, so the state of the square remains unchanged.

And it's equally obvious that the rule '*State 1, Digit 1*: Stop' works properly. Enter along line 1, exit the read/write head along line 1, and you get fed straight into the line that ends at the terminal. By setting up the lines according to the list of rules for the Turing machine, you can make the layout simulate that Turing machine exactly.

These ideas have a fascinating philosophical implication. They show that the future behaviour of a train-set can be undecidable. Turing proved that the halting problem for Turing machines is formally undecidable. Given an arbitrary track layout, you can't predict in advance whether the

train is ever going to reach the terminal.

That's quite startling. Most of us have never been terribly bothered about theoretical questions of formal undecidability in computer science. But it's a bit worrying that you could set up a mechanical system with train tracks, whose workings are totally transparent, and not be able to answer such a simple question as whether a train will ever reach a particular station.

LIFE

Nearly twenty years ago, the British mathematician John Horton Conway invented a model of the computational process that is even simpler than Turing's. His original aim was to produce a game, but all along at the back of his mind was the idea that that game should have simple rules but very complex results – as complex, indeed, as any computation.

Conway called his game Life☛, and it will give you some very useful intuition about how simple rules can lead to complex behaviour. Human beings have a habit of assuming that the complexity of behaviour must somehow be present in the causes of that behaviour – that is, in the rules that generate it. Life is one of a number of mathematical gadgets that demonstrate that this intuition is false. Complexity can be generated *spontaneously* by very simple rules. Not by all simple rules, though: the rule 'do nothing' is simple, but what it generates is totally static and boring. Some simple rules lead to complex behaviour; others do not. One of the morals of this chapter is that we should not expect to find an *easy* test for which rules do or do not generate complexity.

Life is a cellular automaton, a concept we've already met. A huge chessboard, with cells that change colour according to some fixed system of rules. Life uses only two colours, black and white, and there are only three rules:

- A cell that is white at one instant becomes black at the next if it has precisely three black neighbours.
- A cell that is black at one instant becomes white at the next if it has four or more black neighbours.
- A cell that is black at one instant becomes white at the next if it has one or no black neighbours.

In all other cases, cells maintain their colour. The neighbours of a given square are the eight cells adjacent to it vertically, horizontally, or

diagonally. All changes are deemed to be made simultaneously.

The idea is to start with an object made up of black cells, and the rest of the board white. Then you follow the rules and watch how that object changes. For example, a 2×1 block dies out at the first move. A 2×2 block doesn't do anything, so it survives indefinitely. More interesting is a simple shape called a glider: it moves. It changes shape in a cycle of length four, and by the end of the cycle it has moved one cell diagonally. More complicated shapes, spaceships, move horizontally or vertically. A 'glider gun' which changes through a fixed cycle of 30 shapes fires an endless stream of gliders (Figure 59).

Life is a rule-based universe. The future of any initial state is completely determined by the three rules – Life's 'Theory of Everything'. But in practice it may be very hard to predict what will happen, even though it's all implicit in the rules. To make it explicit, you have to follow the rules and see what happens: there don't seem to be any short cuts. Big shapes can collapse, small ones grow, and there are always surprises. Conway proved that the outcome of the game is inherently unpredictable, in the sense that there is no way to decide in advance whether a given object will survive indefinitely, or disappear entirely. He did this by constructing a 'programmable computer' that uses pulses of gliders instead of electrical

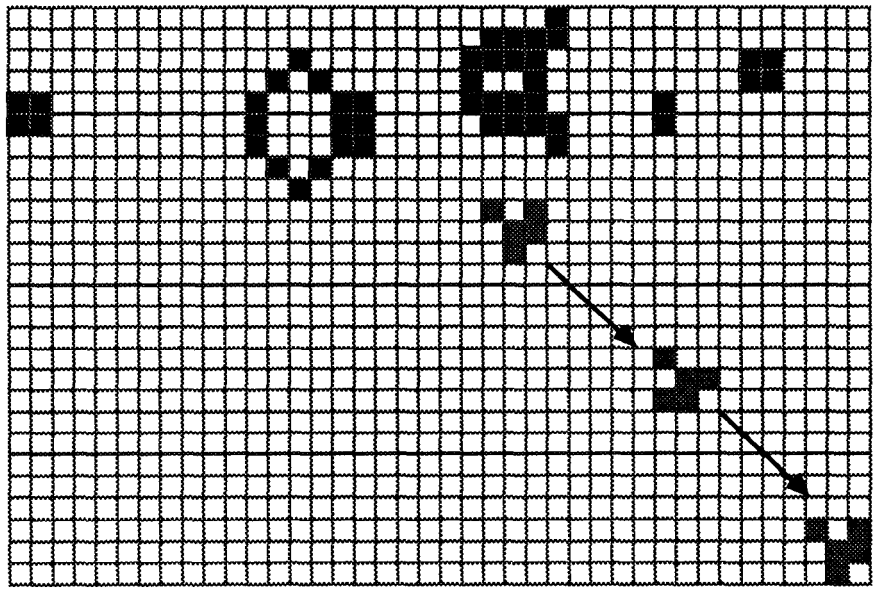


Figure 59 Glider gun (black) fires a stream of gliders (grey).

impulses to carry and manipulate information. Turing's work shows that there cannot exist a computer program that can decide in advance whether any given program, when run on a given machine, will go on for ever, or will stop. The only way to find out is to run the program and watch. If it stops, you know; if it keeps going, you have no idea whether it will continue to keep going, or whether it's just about to stop as soon as you give up and go away.

You can run such an undecidable 'program' on the Life computer. You can even arrange matters so that *if* the program stops then the entire structure annihilates itself by shooting itself down with gliders. So either it goes on for ever, or it wipes itself out. The problem is that there's no way you can tell which. The long-term fate of this particular object, in Conway's simple rule-based game, cannot be determined in advance by any finite computation. We can only let the system run, and watch what it does.

The universe may also be like this, deep down. Even if we knew the rules by which the universe operates – *its* Theory of Everything – we still might not be able to decide what the implications of those rules were.

GÖDEL'S THEOREM

A problem like the halting problem, for which there exists no algorithmic solution, is said to be *undecidable*. Turing's problem was not the first to be proved undecidable, but it is in many ways the simplest example of such a problem. The first problem to be proved undecidable was a mathematical one. It had a rather similar 'feel' to it, but instead of programs and input data it worked with mathematical theorems and proofs. This was the aforementioned work of Kurt Gödel, which demolished Hilbert's programme to put all of mathematics on irrefutably sound foundations.

Gödel must at some point have convinced himself that the Hilbert programme was too ambitious ever to succeed. Indeed, when you come to think about it, a mathematical proof of the infallibility of mathematics looks like circular logic. For example, if mathematics is logically flawed, then even if you could write down such a 'proof', it could well be fallacious – because the language in which it is written is logically flawed. Hilbert was aware of this difficulty, but he convinced himself that it was possible to get round it by viewing mathematics in two distinct ways: as a meaningful series of logically related statements, or as a manipulative game played with meaningless symbols. Gödel found a way to pin down

the feeling that the logic of Hilbert's programme was circular. At its heart – just as with Turing's work – there was a logical paradox.

Turing's proof of undecidability for the halting problem is modelled on the card whose two sides bear contradictory messages. Gödel's theorem about the undecidability of arithmetic is similarly modelled on the even simpler 'liar paradox':

This sentence is false.

Instead of truth and falsity, however, Gödel worked with provability. A mathematical statement is said to be *provable* if there exists a proof – starting from some formal system of logical axioms for mathematics, and deduced using valid logic. If no such proof exists, then the statement is said to be *unprovable*. Provability is a concept well suited to the metaphor of the magical maze. The axioms for mathematics are the entrance to the maze. The rules of logic take us along passages to new junctions, new logical statements – new theorems. A theorem is provable if we can find a path through the maze that leads to it, starting from the entrance. It is unprovable if there is no such path. The unprovable theorems lie in inaccessible regions of the magical maze.

The natural assumption for Hilbert and his predecessors was that unprovable is the same as disprovable. A theorem is disprovable if its negation is provable: for instance, by proving that $2+2=4$ we disprove $2+2\neq4$. A theorem is disprovable if we can find a path through the maze that leads to the theorem's *negation*, starting from the entrance. If mathematics is consistent, as Hilbert was sure it must be, then it is impossible to find a path through the maze that leads to a theorem, and another path through the maze that leads to its negation. The provable theorems are true, the disprovable ones are false, and nothing can be both. This implies that all disprovable theorems lie in inaccessible regions of the magical maze: anything disprovable is certainly unprovable.

Hilbert thought that the two concepts ought to be identical.

Suppose, for the sake of argument, that Hilbert is right. Then every mathematical statement is either provable or unprovable – and not both. (In the jargon, mathematics is both *complete* and *consistent*.) Then a statement will be provable if and only if it is true, and unprovable if and only if it is false. We here assume the axioms to be true, hence any valid logical consequence of them is also assumed to be true. If a statement S is true, then not-S is false; if S if false, then not-S is true. The magical maze of all possible mathematical statements divides into two 'mirror image' pieces – a white piece of true and provable statements, and a black piece of false,

unprovable, and inaccessible statements☛. The mirror that maps one piece to the other is the logical operation 'not'. The pieces do not overlap (consistency), and there are no 'grey areas' that are neither black nor white (completeness).

It's a comfortable picture, but Gödel showed that it is an incorrect one. He proved that either black and white areas must overlap (and so mathematics is inconsistent) or there have to be some grey areas (and some statements can neither be proved nor disproved). His idea was to make use of the simplest logical paradox, by constructing a mathematical theorem that states:

This theorem cannot be proved.

He couldn't do this directly – 'this theorem' has no obvious interpretation as a statement in formal mathematics. So he played Hilbert's game of interpreting mathematics in two different ways; and he played it to Hilbert's detriment. The first step, like Hilbert's, was to think of any mathematical statement as a string of symbols, so that '2+2=4' is no longer considered as a statement about arithmetic – one that must be either true or false, and is actually true. Instead it is just a string of five meaningless symbols:

'2' '+' '2' '=' '4'.

Not only statements, but entire proofs, are strings of symbols.

Next, Gödel found a way to encode any symbol string as a number – and conversely to decode numbers into symbol strings. This is a bit more tricky, but here's one way to do it. First, assign a fixed numerical value to each symbol – perhaps as in Table 12 (the sequence can be continued indefinitely). It doesn't matter how you do this, but the Gödel code has

Table 12

symbol	value
–	1
=	2
1	3
2	4
3	5
4	6

to be set up at the start and never changed thereafter. Then you take a string of symbols, such as 2, +, 2, =, 4, and replace it by the corresponding sequence of code values 4, 1, 4, 2, 6. This is a whole series of numbers,

not just one, but now you calculate the single number

$$2^4 3^1 5^4 7^2 11^6,$$

where 2, 3, 5, 7, 11, ... are the primes, and the powers 4, 1, 4, 2, 6 are the code values for the symbols in the original string, in order.

The result is a pretty big number, even for a short string of symbols, but that doesn't matter. What is important is that each symbol string determines a single number, and conversely each number determines a symbol string. Going from symbol strings to numbers is straightforward and algorithmic. Going the other way – decoding the number to get its symbol string – relies on the fact that a number can be represented as a product of primes in exactly one way. So if we start with a number such as 4800, we find its symbol string like this:

- Start with 4800.
- Write it as a product of primes: $4800 = 2^6 3^1 5^2$.
- Look at the sequence of powers: 6, 1, 2.
- Write down the corresponding symbols: 4, +, =.
- Remove commas to get the symbol string: 4+=.

It may not be a meaningful string – here it isn't – but we know which string it is.

What is a proof? It is a series of logical manipulations on symbol strings. By way of the Gödel code, it can be converted into a series of manipulations on *numbers*. Those manipulations can be represented in terms of ordinary arithmetic – not easily or obviously, but in a straightforward way once you adopt the right viewpoint. So the existence of a proof of some statement can be rephrased in arithmetical terms. Instead of

> Starting from the axioms, there exists a series of logical deductions that leads to statement X,

we have the equivalent statement

> Starting with this list of numbers and applying one or more of the following arithmetical processes, there exists a series of logical deductions that leads to the number that is the code for X.

This second statement provides an arithmetical interpretation for 'there exists a proof of'. In particular, Gödel could give an arithmetical interpretation of statements such as

There exists a proof of the symbol string with Gödel code 1066,

or

There does not exist a proof of the symbol string with Gödel code 1066.

These statements, too, could be represented by single numbers.

Finally, he arranged matters so that the Gödel code for 'There does not exist a proof of symbol string 1066' was the *same* number, 1066, that was referred to in the statement itself. Of course he didn't use 1066 as such, but for simplicity we can pretend that that was the number. The actual number was *huge* – too big to write down, but not too big to specify indirectly.

In effect, symbol string 1066 said (in a suitable interpretation) that 'this statement has no proof'. It was – via its code – self-referential.

Now, suppose that we are in the situation that Hilbert considered desirable – and achievable – in which 'provable' is the same as 'true', and 'unprovable' is the same as 'false'. Then is statement 1066 true or false?

If statement 1066 is true, then since it asserts that statement 1066 is unprovable, it follows that statement 1066 is false.

If statement 1066 is false, then it is unprovable, so the statement 'statement 1066 is unprovable' is true. But this is statement 1066, so statement 1066 is true.

If statement 1066 is true, then it is false; similarly if statement 1066 is false, then it is true.

Notice that Gödel is *not* asserting that such a statement actually exists. What he is asserting is more subtle: *if* Hilbert is right that 'true' and 'provable' are the same thing, *then* such a self-contradictory statement exists. Since self-contradictory statements cannot occur in a logically consistent mathematics, we then deduce that *either*

Hilbert was wrong

or

Mathematics is logically inconsistent.

The second statement is just another way of saying that Hilbert was wrong, but for a different reason. So either way, Hilbert was wrong.

He wasn't very pleased to be told this. But he was a good enough mathematician, and an honest enough person, to recognise that Gödel was right.

'We must know, we shall know,' said Hilbert in his radio broadcast. Gödel knew better. There are some things that we *cannot* know.

JUNCTION SEVEN

The train pulls into a station, and halts. You do not even have to push the STOP button. You sag into the driver's seat and draw a deep breath. Ahead are buffers. The station is a dead-end. You took the wrong path, wandered over to the wrong side of the railway tracks, and discovered only a blank wall, blocking any further progress through the magical maze.

The STOP button has disappeared. In its place is one marked REVERSE. You hesitate, but only for a moment. What other choice is there? You push the button. The train chugs gently backwards, out of the station. After a worryingly long sequence of twists and turns, and much resetting of points, as always, you return to where you went astray.

Nervously you jump out of the engine, but it just sits there – waiting for the next foolish passenger.

You know where to go, now. You turn sharply to your right, through an archway, into the seventh passage of the magical maze. Over the arch is engraved a motto:

THINK FIRST, CALCULATE LATER

The archway leads to an irregular, zigzag passage, with several blind alleys branching off ... you almost become trapped again, but you realise your error and backtrack – fast. A dozen burly workmen are digging trenches. Some are feeding lengths of cable into the trenches from large reels. Others are pulling it out again – and tempers are becoming frayed. A foreman, consulting a large plan, scratches his head in bewilderment. You tiptoe quietly past them.

A tiny bubble drifts past on the breeze. Then another, and another. Soon they are coming thick and fast, shimmering in all the colours of the rainbow. There are bubbles as big as your head, as big as your body – one bubble so huge that you could imagine living in it.

There are bubbles joined to bubbles, bubbles inside bubbles, long strings of bubbles like plump, round sausages ...

You follow the trail of bubbles towards its source.

Now the passage is beginning to fill with foam. It rises to your ankles, your knees, your waist ...

You decide it's time to get out of here.
By whichever route is the quickest.