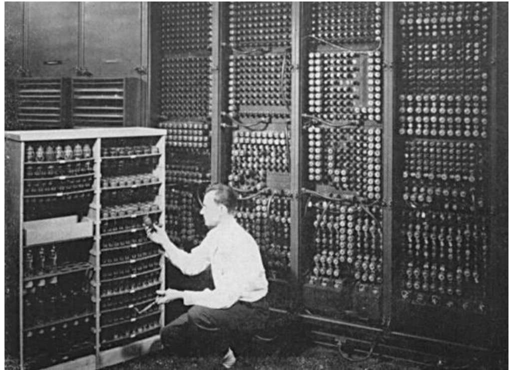
6. THE UNSOLVABLE

§6.1. Are Computers Omnipotent?

Is there anything a computer can't do? Certainly we've witnessed some amazing developments during the eighty or so years computers have been around. Of course we can think of some things that computers can't do – yet. But sooner or later ...

Of course computers can't solve any problem that has no solution. They can't come up with a proof that 1 + 1 = 3, or a procedure which can trisect any angle exactly by ruler and compass. But surely if a solution to a problem exists a computer program can be written to find it. perhaps not today, or tomorrow, but at some time in the future.



In fact our popular belief in the intellectual omniscience of the computer is misplaced. There *are* problems which have solutions but which no computer *has* ever solved, *will* never solve and *can* never solve.

But wait! Aren't we limiting the ingenuity of man? People once said that man will never fly in heavier-than air machines, that we will never be able to reach the moon, that smallpox will never be eradicated. How short-sighted is the person who declares that so and so will never happen. Yet that’s what I’m saying. Problems exist, problems which have a solution, which man can never solve. And not just human man. No being whose thought processes are based on the same logic as ours can possibly solve these unsolvable problems.

§6.2. The Halting Problem

There’s a dream that every novice programmer has. When a computer program is ‘compiled’ (this just means translating it into a form that the computer more readily understands) the compiler program generates error messages to say that you appear to have left out a comma here or you’ve misspelt the name of a variable there.

But despite this, usually the first time a novice writes his or her first really complex program the computer ‘freezes’. Stupid machine – the keyboard doesn’t work, the screen goes on strike. The program has to be aborted by using some emergency key-stroke combination or switching off the power. Even experienced programmers, like the ones who wrote the operating system of your computer, can’t avoid having bugs that emerge from time to time – hopefully not too frequently.

When our own program ‘crashes’ our first thought may be to blame the operating system, or the hardware. Perhaps my computer has a virus. But soon the novice discovers that it wasn’t the computer that was at fault, but their program. There was an unforeseen infinite loop in the program.

A very obvious case of such an infinite loop is:

```
10: GO TO 10
```

which in line 10 sends the machine back to the same instruction all the time.

An equally obvious case of a program failing to halt is:

```
10: N = 2
20: LET N = N + 1
30: IF N < 2 THEN GO TO 10
```

You might argue that we haven’t got into the sort of loop whereby the computer returns to a previous state. The value of N never repeats. Nevertheless we include such infinite paths as an infinite ‘loop’.

You’d have to be pretty stupid to write such programs, but the problem is that infinite loops can be very subtle and hard to find. Take the following program.

```
1: LET T = 0
2: LET N = 0
3: ADD 1 TO N
4: ADD 1/N^2 TO T
5: IF T < 2 GO BACK TO STEP 3
6: OTHERWISE PRINT N AND HALT
```

We start with T = 0 and N = 0. Then we add 1 to N, giving N = 1, and add $1/N^2$ to T, giving T = 1.
Since T is less than 2 we go back to step 3.
Then we add 1 to N, so that now N = 2 and add ¼ to T, giving T = 1.25.
Again T is less than 2 and so we go back to step 3.
After 10 steps we will have:
$T = 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \frac{1}{25} + \frac{1}{36} + \frac{1}{49} + \frac{1}{64} + \frac{1}{81} + \frac{1}{100}$ which is about 1.55.

Although T is increasing, it will never be bigger than 2. In fact it's possible to show that T will never be bigger than 1.645. So the program will continue running forever. It will never halt. The Swiss mathematician, Euler, proved that the sum to infinity of this series is $\pi^2/6$ which is 1.644934067 …

Now consider the following similar program.

```
1: LET T = 0
2: LET N = 1
3: ADD 1/N TO T
4: ADD 1 TO N
5: IF T < 10 GO BACK TO STEP 3
6: OTHERWISE PRINT THE VALUE OF N AND
HALT
```

As with the previous program we're adding smaller and smaller numbers to the total. It will still be running after 10000 steps and it would be easy to think that it will go on forever. However, this time the program will eventually halt. When N = 12367, the value of T will become 10.00004301.

Merely deciding that if a program hasn't halted after a certain number of steps is no guarantee that it will never halt. If we changed the '10' in the above program to 1000 the program would, in principle, still halt. But if you ran this program on the fastest computer in the world it would still be running after a year. You might decide that surely it's going to run forever, but in principle it would halt. The only problem is that the universe, and all computers within it, would have decayed long before the program halted.

Now wouldn't it be wonderful if some piece of software could examine a program, and the data that I plan to use as input, to see if it will get into any infinite loops *before* I actually run it. Such a program would examine the logical structure of my program and very cleverly predict whether or not my program would get itself in an infinite loop.

Such are the very simple specifications for a halting predictor program. "I can see how it could easily detect obvious bugs like 10: GO TO 10 but I'm not sure how it would detect the more subtle loops. But still I'm sure it could be done by some very clever programmer."

Alan designed the perfect computer

Not so! This dream will be forever a dream. Very clever programmers may be able to design something that picks up the more obvious loops. But no programmer will ever be able to write something that can pick up *all* of them. The reason is that doing so is a logical contradiction.

§6.3. Programs

A computer program is simply a list of instructions which the computer follows to solve a problem. Humans are often given instructions and there's nothing fundamentally different between a computer and the human brain in this sense.

Recipes are simply programs for cooking. Knitting instructions use a set of symbolic abbreviations which one has to learn. In principle a human being armed with unlimited supplies of paper and pencils can do anything that a computer can do. It's just that the computer does it very much more quickly and accurately.

We can prove that the halting problem is unsolvable using any suitable programming language. One can even use the English language, provided we make our meaning sufficiently precise. This means you can follow this argument without knowing much about computers. Basically, all you need to know is that there are three ingredients in the computing process – input, the program and output.

INPUT ⟶ **PROGRAM** ⟶ **OUTPUT**

I'll use a shorthand to represent a diagram such as the above. I'll write

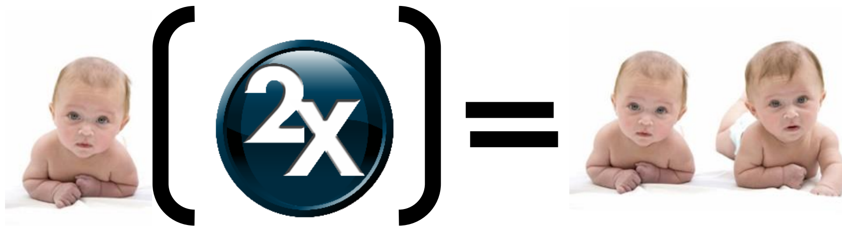
INPUT[PROGRAM] = OUTPUT.

So if **TOASTER** is a program (list of instructions) for using a domestic toaster, I'll write:

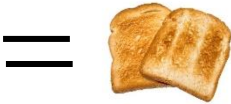
BREAD[TOASTER] = TOAST

to indicate that if the toaster instructions are applied to the input **BREAD**, the output is **TOAST**.

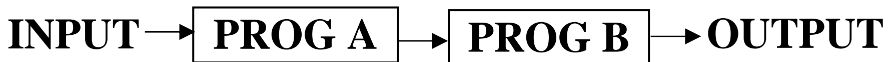
If **DOUBLE** is the program which describes how to double a number then **3[DOUBLE] = 6**.



Often the output of one program becomes the input of another.



So if **PROG A** and **PROG B** are two programs we'll write the program that consists of doing **PROG A** and then **PROG B** as **PROG A + PROG B**.



I'll use a shorthand to represent a diagram such as the above. I'll write

INPUT[PROG A + PROG B]
= INPUT[PROG A][PROG B]
= OUTPUT.

In the above example we could write the process as:

DOUGH[BAKE + CUT + TOASTER]
= DOUGH [BAKE] [CUT] [TOASTER]
= BREAD [CUT] [TOASTER]

= SLICE [TOASTER]
= TOAST

But beware. PROG A + PROG B is not usually the same as PROG B + PROG A. The order in which you do things usually matters. Try following the steps in a recipe in random order!

If PROG A is the operation of putting on your socks and PROG B is the operation of putting on your shoes then PROG A + PROG B is the normal way of getting dressed. PROG B + PROG A, where you put the shoes on first and then the socks will be quite different.

Often in mathematics the order of operations is not important. You can add numbers in any order or multiply them in any order. Adding 3 and then adding 2 is the same as adding 2 and then adding 3. But this is the exception. Most operations in mathematics are sensitive to the order in which we carry them out. For example:

7 [DOUBLE] [ADD 1] = 14 [ADD 1] = 15

while

7 [ADD 1] [DOUBLE] = 8 [DOUBLE] = 16.

§6.4. Some Sample Programs

The input to a program could be a physical object, such as a slice of bread, or a number. But programs that can be processed by a computer can't handle input in the form of a slice of bread. When they were first built it was thought that computers could only accept numbers as their input, but this was wrong.

Fundamentally the input to any computer can only be a string of symbols, essentially strings of 0's and 1's. Such strings can represent numbers. But increasingly, as computers developed we began to realise that such strings can represent words, or pictures, or sounds.

In the examples that follow the input and output are strings of symbols. Fundamentally that's all computers can process.

The strings might represent words, or numbers, or pictures but to the computer they're just meaningless strings to be manipulated according to certain rules.

We'll assume that whenever our programs write something as output they write it on the same line as the input, coming immediately after it, and so the output includes the input, though often the program will explicitly instruct you to erase the input. And when the program runs out of things to do, it halts.

Although, for a real computer, the strings are strings of 0's and 1's, we'll use strings of letters of the alphabet. This will make it more interesting, and easier to understand.

Our first program is called **REVERSE**. Very simply, it reverses the order of the letters in a string. The instructions that make up this program are as follows:

REVERSE
1. Write input backwards.
2. Erase input.

So **MESSAGE[REVERSE] = EGASSEM**.

A palindrome is a string which reads the same forward as backwards, like PUP so palindromes are those strings that **REVERSE** doesn't alter.

One of the most famous palindromes of all is what Napoleon is supposed to have said:

ABLE WAS I ERE I SAW ELBA.

Another famous palindrome, this time without the spaces, is:

AMANAPLANACANALPANAMA.

Reversing a message twice of course brings the message back to the way it was. Thus we can write:

MESSAGE[REVERSE][REVERSE] = MESSAGE.

COUNT is a program which counts the number of symbols in a string.

COUNT
1. Count the symbols in the input.
2. Write this number in words.
3. Erase input.

Carefully examine the following examples and convince yourself that the output claimed is the correct one.

MESSAGE[COUNT] = SEVEN because MESSAGE has 7 letters.

MESSAGE[REVERSE][COUNT]
= EGASSEM[COUNT]
= SEVEN

MESSAGE[COUNT][REVERSE]
= SEVEN[REVERSE]
= NEVES

MESSAGE[COUNT][COUNT][COUNT]
= SEVEN[COUNT][COUNT]
= FIVE[COUNT]
= FOUR

The first COUNT counts the number of letters in MESSAGE and the output is SEVEN. The second COUNT counts the number of letters in SEVEN, which is FIVE. The third COUNT counts the number of letters in FIVE and the output is FOUR.

In fact, if you start with any string and repeatedly apply the program COUNT, eventually you will reach FOUR. Why?
The next program doesn't erase the input. Instead it makes a second copy of the input.

REPEAT
1. Write "+".
2. Copy input.

MESSAGE[REPEAT] = MESSAGE+MESSAGE

BOO [REPEAT] [REPEAT]
= BOO+BOO [REPEAT]
= BOO+BOO+BOO+BOO

MESSAGE[COUNT][REPEAT]
= SEVEN[REPEAT]
= SEVEN+SEVEN

MESSAGE[REPEAT][COUNT]
= MESSAGE+MESSAGE[COUNT]
= FIFTEEN.

The next program doesn't do much except halt. It does throw out an exclamation mark just to prove it's been run.

HALT
1. Write "!".

HELP[HALT] = HELP!

Now for a program which deliberately gets into an infinite loop.

LOOP
1. Copy the last letter of the input.
2. Go to step 1.

AGH[LOOP] = AGHHHHHHHHHHHHHHHH.......

There's no real output because the program never halts. This program will loop, no matter what the input is.

The next program is more discriminating. In fact it will loop, but only if it is told to halt.

DISOBEY
1. If input = **LOOP** then **HALT**.
2. If input = **HALT** then **LOOP**.
3. Otherwise do nothing.

Of course **DISOBEY** doesn't really disobey its instructions. It only appears to do so.

LOOP[DISOBEY] = LOOP!

The machine halts after printing the exclamation mark.

HALT[DISOBEY] = HALTTTTTTTTTTTTT........

This time it doesn't halt. For any other input nothing happens, except for halting.

STAY[DISOBEY] = STAY.

The last of our examples here combines **HALT** and **LOOP** with **COUNT**.

MAYBE
1. If the number of symbols in the input is even then **HALT**. 2. Otherwise **LOOP**.

NO[MAYBE] = NO!

YES[MAYBE] = YESSSSSSSS...........

ANYTHING[REPEAT][MAYBE]
= ANYTHING+ANYTHING[MAYBE]
= ANYTHING+ANYTHINGGGGGG....

(ANYTHING+ANYTHING has odd length.)

NO[MAYBE][MAYBE]
= NO![MAYBE]
= NO!!!!!!!!!!!!.........
Since NO has even length, **NO[MAYBE] = NO!**
Since NO! has odd length:
NO![MAYBE] = NO!!!!!!!!!!!!.........

§6.5. Cannibalism

Let's assemble all the programs we've discussed.

REVERSE	**COUNT**	**REPEAT** / **HALT**
1. Write input backwards. 2. Erase input.	1. Count the symbols in the input. 2. Write this number in words. 3. Erase input.	**REPEAT** 1. Write "+". 2. Copy input. **HALT** 1. Write "!".

LOOP	**DISOBEY**	**MAYBE**
1. Copy the last letter of the input. 2. Go to step 1.	1. If input = **LOOP** then **HALT**. 2. If input = **HALT** then **LOOP**. 3. Otherwise just halt.	1. If input has even length then **HALT**. 2. Otherwise **LOOP**.

Perhaps you may be thinking that it's confusing writing both programs and their input/output data with capital letters. Wouldn't it be better to use lower case for data and capitals for programs? The reason is that programs can be considered as data for other programs. A compiler for a programming language is a very complicated program into which you feed a program to convert it to a form which is convenient for the computer. It's not uncommon for compilers to be written in the same language as the programs they're designed to compile. So you could feed a compiler into a second copy of itself!

Normally when feeding a program to itself we'd do this with the complete list of all the instructions – not just the name of the program. But for simplicity in this discussion let's just work with the names. Let's take each of our seven programs in turn and work out what would happen if their name was used as their own input.

REVERSE[REVERSE] = ESREVER

COUNT[COUNT] = FIVE

REPEAT[REPEAT] = REPEAT+REPEAT

HALT[HALT] = HALT!

LOOP[LOOP] = LOOPPPPPPPPPP.............

DISOBEY[DISOBEY] = DISOBEY

MAYBE[MAYBE] = MAYBEEEEEEEE..........

§6.6. Predicting Loopiness

We now come to a program which doesn't exist, even though the following description suggests that it might.

PREDICT
1. If the input has the form data+program and the program would halt given that data as input, then erase the input, write **HALT** and halt. 2. If the input has the form data+program and the program would never halt given that data as input, then erase the input, write **LOOP** and halt. 3. If the input doesn't have the form data+program, then erase everything and write **???**.

Although I've listed what we'd like the program to do we haven't said how it should decide whether the program would halt given that data as input. Of course the fact that we can't think

how to do it doesn't of itself make **PREDICT** an impossibility. That is something we've yet to prove. But just *suppose* for the moment that such a **PREDICT** existed.

MESSAGE+COUNT[PREDICT] = HALT
because MESSAGE[COUNT] = SEVEN, and so stops.

LOOP[PREDICT] = ???
because the input doesn't have the required form with a "+" separating two parts.

MESSAGE+LOOP[PREDICT] = LOOP
because MESSAGE[LOOP] = MESSAGEEEEEE...., going into an infinite loop.

YES+MAYBE[PREDICT] = LOOP
because YES[MAYBE] = YESSSSSSSS........, which doesn't stop.

NO+MAYBE[PREDICT] = HALT
because NO[MAYBE] = NO! which stops.

Notice that in all these cases our human brain was ingenious enough to work out what would happen — halt or loop. How did we do it? Did we have a systematic procedure? If so, we're well on the way to bringing **PREDICT** into existence. But no, we predicted the

behaviour of our programs on an *ad hoc* basis. As we shall see this is the best we can ever hope for.

§6.7. Cannibal Programs

We'll call a program a **cannibal** if it halts when fed a copy of itself as input. Let's see how many cannibals we've bred.

REVERSE[REVERSE] = ESREVER
This halts, so **REVERSE** is a cannibal.

COUNT[COUNT] = FIVE
This halts. It, too, is a cannibal.

REPEAT[REPEAT] = REPEAT+REPEAT

HALT[HALT] = HALT!

Both REPEAT and HALT are cannibals.

DISOBEY[DISOBEY] = DISOBEY
Although DISOBEY will sometimes loop forever (if the input is HALT), when fed its own description it halts and so it too is a cannibal.

LOOP[LOOP] = LOOPPPPPPP …………

MAYBE[MAYBE] = MAYBEEEEE ………
These are not cannibals because they loop when fed their own description.

§6.8. The Final Showdown

We're now going to build our final program. I call it **MEPHISTOPHELES** because this is a being that most people believe doesn't exist. In the description that follows, **THIS** represents any possible input and **THAT** represents any valid program.

Now to do this we'll need to assume that a program called **PREDICT**, as described above, *does* exist. If it's fed some data of the form **THIS+THAT** it assumes that **THIS** is data and **THAT** is a program. It then works in some clever way whether the program that we are calling **THAT** would halt, if given **THIS** as input, and prints out **HALT** or **LOOP** accordingly.

If the input doesn't have the right format, or if **THAT** is not in our list of programs, then it halts with **?** as output.

Now of course it can't do this by running the program **THAT**, starting with input **THIS**, because if **THAT** would never halt if given this input then the **PREDICT** program would never reach a final answer.

In a world where some clocks could go forever you could never predict that a given clock will never stop just by waiting for it to stop.

If some humans were immortal, and others weren't, you couldn't decide who was which by reading the death notices in the newspaper. The fact that a name doesn't appear for over a thousand years (of course we're assuming that all deaths are reported in the death notices) might mean that the person is really an immortal or simply that he's a very long living mortal. Computer programs are the same.

If **PREDICT** is going to exist it will have to be exceedingly cunning and examine the structure of the program whose behaviour it is trying to predict. And is it impossible that a clever programmer might one day be able to do it? Frankly, yes, it is impossible. We will *prove* that it is impossible.

MEPHISTOPHELES is built up from the programs that we've constructed, which certainly do exist, together with the one we have merely described, **PREDICT**. If things go wrong, as they will, it will mean that **PREDICT** doesn't really exist.

MEPHISTOPHELES
1. REPEAT.
2. PREDICT.
3. DISOBEY

Let's check out MEPHISTOPHELES with certain inputs.

THIS [MEPHISTOPHELES]
= THIS [REPEAT] [PREDICT] [DISOBEY]
= THIS+THIS [PREDICT] [DISOBEY]
= ? [DISOBEY]
= ???

Here **THIS** is not representing arbitrary input, but the specific four letter word. Since we have not defined a program called **THIS**, **PREDICT** simply prints **???** and halts. So unless the input to MEPHISTOPHELES is a valid program the output will simply be ???

DOUBLE [MEPHISTOPHELES]
= DOUBLE [REPEAT] [PREDICT] [DISOBEY]
= DOUBLE+DOUBLE [PREDICT] [DISOBEY]
= HALT [DISOBEY]
= HALT [LOOP]
= HALTTTTTT …..

LOOP [MEPHISTOPHELES]
= LOOP [REPEAT] [PREDICT] [DISOBEY]

= LOOP+LOOP [PREDICT] [DISOBEY]
= LOOP [DISOBEY]
= LOOP [HALT]
= LOOP!

Since **LOOP [LOOP]**
= **LOOPPPPPPP ………………..**
PREDICT will detect this and merely print out **LOOP**. Then **DISOBEY** will run the **HALT** program.

The big question we are now going to ask is this:

QUESTION: Is MEPHISTOPHELES a cannibal?

Of course the answer has to be either "yes" or "no". Let's examine each possibility in turn. The logic of the argument requires a little tenacity to follow. Just hang in there and follow it slowly, step by step.

CASE 1: Suppose **MEPHISTOPHELES** is a cannibal. What does that mean?
It means that **MEPHISTOPHELES** will halt if it feeds upon itself, that is:
MEPHISTOPHELES[MEPHISTOPHELES] will halt and so

MEPHISTOPHELES+MEPHISTOPHELES
[PREDICT] = HALT.

Now **MEPHISTOPHELES [MEPHISTOPHELES]**
= MEPHISTOPHELES [REPEAT] [PREDICT] [DISOBEY]
= MEPHISTOPHELES+MEPHISTOPHELES [PREDICT] [DISOBEY]
= HALT [DISOBEY]
= HALT!!!!!! …

But this says that **MEPHISTOPHELES** *doesn't* halt when fed its own description, contradicting our assumption.

CASE 2: Suppose **MEPHISTOPHELES** is not a cannibal, that is
MEPHISTOPHELES+MEPHISTOPHELES [PREDICT] = LOOP.

Now **MEPHISTOPHELES [MEPHISTOPHELES]**
= MEPHISTOPHELES [REPEAT][PREDICT] [DISOBEY]
= MEPHISTOPHELES+MEPHISTOPHELES [PREDICT] [DISOBEY]
= LOOP [DISOBEY]
= LOOP!

But this says that **MEPHISTOPHELES** *does* halt when fed its own description. Again this contradicts our assumption.

Only two possibilities and neither of them true. Each alternative leads to a contradiction. We're in a maze and there's no way out except the door by which we came in. Everything we did was conditional on our assumption that a program satisfying the specifications of **PREDICT** can exist. Therefore it cannot! The halting problem is unsolvable!

INTERLUDE: PLAY
"It's Got To Stop Sometime"

Scene: *A classroom with a long, wide blackboard at the front. The professor is standing at the front, asking for volunteers.*

Prof: Come on now, I need five volunteers to be "people programs". All you need to do is to hold up one of these cards and, when I say, you just perform the instructions on the card to whatever is written on the board.

Noel: I'll have a go but I'm not very good at this sort of thing. I'm sure I'll get it all back-to-front.

Prof: That's exactly what I want you to do. Your program is called REVERSE.

He hands Noel a card on which is written the words:

REVERSE
Reverse what's on the board.

Now whenever I call on you, all you have to do is to rewrite whatever is on the board backwards.

Peter: If it's as easy as that then I'm your man and as my mum always says if you want someone to do a job

properly and not give up half-way through then ask me because I’m your man and as my mum always says ...

Prof: I’m sure you are, Peter. Your program is called REPEAT.

He hands him a second card bearing the instruction:

REPEAT
While what is written on the board ends in “T” put another “T” at the end of it.

The Bubble Twins: *(in chorus)* We’d like to help too, but only if we can do it together.

Prof: Oh, then you'll like your job.

He gives June Bubble a card on which is written:

DOUBLE
Make a second copy of what’s on the board, separated by a space

When I call on you, all you have to do is to make a second copy of whatever appears on the blackboard.

Jane: *(to her sister)* Ooh, I'll do the copying because I've got the steadier hand. You can hold up the instructions in case I forget them.

Prof: Right, let's practise those three programs.

Mary: What about me? I knew you'd forget me. It's just not fair!

Prof: You'll get your chance, Miss Contrary, I've got just the job for you. But we'll just practice these first three. Now when I call out the name of your program you have to perform the instructions on your card to whatever is on the blackboard. If I say REVERSE that's your cue, Leon.

Noel: Do you mean me?

Prof: Sorry, Noel, yes it's you I mean. And if I say REPEAT its over to you Peter. And your cue girls is DOUBLE.

He writes the letters RAH on the board.

OK it's DOUBLE first.

*The Bubble sisters write a second **RAH** next to the first to get RAH RAH.*

Now REVERSE.

*Noel rubs out the message **RAH RAH** and replaces it with **HAR HAR**.*

And DOUBLE again.

*The message now becomes **HAR HAR HAR HAR**.*

And finally REPEAT.

*Peter was about to start tacking a row of **R**'s on the end of the data but the Prof caught him just in time.*

No Pete. Your instructions are to add T's and only when what is already there ends in T. When it ends in anything else you do nothing.

Peter, somewhat disappointed, sits down again.

Now we’ll try another one.

*He cleans the board and writes the word **EXIT**.*

REVERSE.

*Noel changes **EXIT** into **TIXE**.*

Peter: Isn’t ENTRANCE the reverse of EXIT?

Prof: No Pete, Noel's right. I said REVERSE, not OPPOSITE. OK, now DOUBLE.

*Jane Bubble adds a second **TIXE**.*

REVERSE

*Noel replaces the **TIXE TIXE** with **EXIT EXIT**.*

And now REPEAT.

*Peter excitedly writes **T** after **T**, getting **EXIT EXITTTTTTTTTTTTTTTT......** until he runs out of blackboard. The Prof has to restrain him from continuing across the wall.*

Mary: That's stupid! Whenever Pete takes off nobody else can follow him.

Prof: No, Mary, its not stupid. It's just what happens when a computer program crashes because it gets into a loop.

Mary: Well it's stupid ever to get into a loop. The computer should be clever enough to know that it's being told to get into a never-ending loop and spit out the offending program.

Prof: But Mary, it’s not always so easy to ensure that a program will go on forever.

Mary: ‘Course it is! Any fool could see what was going to happen when Pete took over. A clever computer would be able to examine any programs it had to run and refuse any which would make it crash.

Prof: But that would need another program to work out what would happen.

Mary: So what! It might be a complicated program but I'm sure someone smart like Tim could come up with one. You just get Tim's program to look at the one you're going to run and if it’s OK it rings a bell and if it would loop forever it rings a buzzer. Then you'd know not to let the computer run any program that sets off the buzzer.

Prof: But this program would have to be able to work on every possible program.

Mary: Sure, and what’s wrong with that?

Prof: Well, it would even have to be able to work on itself.

Mary: Well any dum dum can see that Tim's program would always halt so if you ran it on itself you'd get the

bell, of course. Now when are you going to give me my program, or had you forgotten?

Prof: OK Mary Contrary, I’ve got just the program for you. It’s called DISOBEY.

He gives her a card with the following instructions:

DISOBEY
If what’s written on the board is
HALT then REPEAT.
If it is LOOP then REVERSE.
Otherwise print “?”

Mary: But that’s silly. If I’m told to HALT I go on forever writing HALTTTTTTT.......
and if, for example, I’m told LOOP, I write POOL and then halt. I’ll always be doing the opposite to what I’m told.

Prof: That's why it’s called DISOBEY, Miss Contrary! Let’s try it out.

He writes ***POTS*** *on the board.*

Now REVERSE.

Noel changes it to STOP.

And now DISOBEY.

Mary: Well the data isn't HALT so I do the "otherwise" bit. That means getting POTS again.

She picks up the duster but the professor gently restrains her.

What's the matter, I've got to do a REVERSE, don't I?

Prof: Not you, your job is to activate Leon as a subroutine. He does the actual reversing.

Mary: Oh, all right then. Go on Noel. (I suppose that's who you meant.)

*Noel reverses **LOOP** and once again the word **POOL** is written on the board.*

Prof: Now again.

*He cleans the board and writes **HALT**.*

OK Mary DISOBEY.

*Mary gives Peter a hard thump and Peter starts writing dozens of **T**'s until the Professor gives Peter a nudge to break him out of his infinite loop.*

Now has it ever occurred to you that a program can be made to operate on itself?

Tim: Well I suppose I could write a program called COUNT which counts the number of words in a piece of text and I could run it on a copy of the COUNT program itself.

Prof: Exactly. So June, if DOUBLE acted upon itself, what would happen?

June: DOUBLE DOUBLE toil and trouble – well just DOUBLE DOUBLE I suppose.
Prof: And, Leon, what if you REVERSE REVERSE?

Noel: You'd get ESREVER.

Prof: Pete, would you mind doing REPEAT on REPEAT.

Peter: What do you mean?

Prof: I mean write REPEAT on the board as your data and carry out the REPEAT program on it.

Peter writes REPEAT on the board and then, after scratching his head for a minute, he turns it into REPEATTTTTTTTTTTTTTTTTT......

Prof: So if DOUBLE acts upon itself it will halt. The same is true of REVERSE. But if REPEAT acts on its own description as data it will never halt.

Jane: It's just like it gets indigestion. It can't digest a copy of itself.

Mary: Sounds like a cannibal. What a positively disgusting idea!

Prof: That's a good analogy. How about if we call a program a "cannibal" if it halts when it feeds on itself. So DOUBLE and REVERSE are cannibals. But REPEAT isn't. As Jane says, it gets indigestion if it tries to eat a copy of itself. What about DISOBEY Mary?

Mary: DISOBEY isn't HALT so once again I do the "otherwise". Go on Noel, REVERSE.

And Noel proceeds to turn DISOBEY into YEBOSID.

Prof: So DISOBEY is a cannibal program.
Now Tim, the last program is yours. It's called PREDICT.

Tim: I knew you'd say something like that. You're going to tell me that my program predicts whether or not any program will halt, or whether it will go into an infinite loop.

Prof: Exactly, and because the answer will depend on what data it's given it needs to be given the program plus the data.

He hands Tim the last card with the program:

PREDICT If the program will halt when given the data, print out HALT but if the program will loop, print out LOOP

Noel: That's not very difficult. All Tim's program has to do is just run the given program and if it halts then it prints out HALT and if it doesn't halt ...

Prof: ... then you'd never be able to break into it to print out the message LOOP.

Peter: Well can't you just break it out of its loop if it seems to be going on too long?

Prof: How long is too long? A program might take a very long time and still halt. Even if you waited a hundred years you wouldn't know for certain that it's not going to halt some time in the future.

Noel: Well how's Tim going to do it?

Prof: He can’t. It’s impossible.

Mary: That’s rubbish. Tim’s a computer whiz. And even if Tim can’t, someone will one day. It makes me mad when people say that something is impossible just because they're not clever enough to do it themselves! Someone clever can examine the program and work out whether it will halt, without actually running it.

Prof: Well, we’re supposing for the sake of argument that Tim has done it and PREDICT is that program. Let’s try it out.

*He writes the word **TEST** followed by the word **DOUBLE**.*

OK Tim, PREDICT.

Tim: Well it’s obvious that if you ran the program DOUBLE on the TEST data you're just going to get TEST TEST.

Prof: So, carry out your program.

Tim: If I ran DOUBLE on TEST the program would halt so I write the word **HALT**.

*He erases **TEST DOUBLE** and replaces it by **HALT**.*

The Prof now writes ***REPEAT*** *to the right of* ***HALT*** *to get HALT REPEAT.*

Right Tim, here's another example, go ahead and PREDICT.

Tim: Clearly I predict that REPEAT will loop in this case.

He writes the word ***LOOP*** *in place of* ***HALT REPEAT****.*

Prof: Well Tim, is PREDICT a cannibal? Will it halt if it feeds upon its own description?

Tim: I guess so. It is supposed to print either HALT or LOOP, but in either case it, itself, *has* to halt so that you can read its answer.

Prof: Now if I was to attach DOUBLE to PREDICT you'd get a program which tells you whether or not any given program is a cannibal. But I want to give it a twist. Here is a program I've called MONSTER.

The professor holds up the last card displaying the four words:

MONSTER
DOUBLE
PREDICT
DISOBEY

Prof: Do you think MONSTER is a cannibal?

Peter: Well it sounds like a pretty uncivilised, pagan program so I guess it is.

Prof: Guessing isn't good enough. We must have certainty.

Jane: Well, one thing's for certain, either it is a cannibal or it isn't.

Mary: Stupid girl. Where do you think that inane remark will get us?

Prof: Further than you might think. Let's follow up each possibility in turn. Suppose Pete is right and it is a cannibal. Let's feed MONSTER its own description to digest. What happens first?

June: Well first we do DOUBLE and get MONSTER MONSTER.

Tim: Then PREDICT examines the structure of MONSTER and decides whether it will halt when it feeds on MONSTER.

Noel: And because we're at the moment assuming that it's a cannibal it will be able to digest its own description, so PREDICT will spit out HALT.

Mary: Then I come along and upset the applecart, because as soon as I see the word HALT, my instructions in DISOBEY tell me to turn this into HALTTTTTTTTTTTTTT...

Prof: Not quite. You have to ask Peter to run the program REPEAT. But it amounts to the same thing.

Peter: But that will give MONSTER indigestion. It'll never get to the end.

Mary: So MONSTER is not a cannibal after all. That's dumb. We assumed it was.

Prof: So all that means is that that assumption has to be rejected.

Tim: Oh, I see, that contradiction proves that MONSTER is not a cannibal.

Prof: Well, as that seems to be the only possibility remaining, let us assume that MONSTER is not a cannibal, that is, it will go on for ever if it feeds on a copy of itself.

Mary: We don’t need to assume that, we know that.

Prof: So lets follow through MONSTER again as it attempts to digest MONSTER. First step gets us MONSTER MONSTER.

Tim: Then my PREDICT program interprets this as the program MONSTER acting on the data MONSTER and PREDICT must predict whether it will halt.

Noel: And since we know that MONSTER is not a cannibal, the answer LOOP will come out of the PREDICT part of MONSTER.

Mary: And then I come along and DISOBEY, which means that since I don't see the word HALT I simply turn the LOOP into a POOL and halt. But that’s dumb too because that means that MONSTER is a cannibal. It fed upon itself and finished. Didn’t you say that MONSTER couldn’t be a cannibal?

Prof: Well we do appear to be in a bit of a fix. If we suppose that MONSTER is a cannibal we can prove he isn’t and if he isn’t we can prove he is.

Mary: That’s the dumbest thing I ever heard. If he is, he isn’t and if he isn’t he is!

Prof: So we've reached a blank wall again. But remember, we're still making an assumption.

Noel: What's that?

Prof: Well Tim hasn't actually got a PREDICT program.

Peter: So ... ?

Prof: If ever he, or anyone else for that matter, ever came up with a PREDICT program that can decide in advance whether or not any given program will halt, the contradiction we reached a moment ago must inevitably follow. So no such program could ever be written. The Halting Problem is insoluble!

Mary: My "Halting Problem" is the fact that this stupid lesson seems to be going on forever. Tim, do you predict it will ever HALT?

At that moment the end-of-lesson bell was heard.

Tim: Indeed I do.