6. THE UNSOLVABLE

1. Are Computers Omnipotent?
   Is there anything a computer can’t do? Certainly we’ve witnessed some amazing developments during the seventy 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 do anything that is impossible. 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 omnipotence 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 am 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.

2. The Halting Problem
   There is 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 unforseen 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.

   You’d have to be pretty stupid to write such an obvious infinite loop into your program, but the problem is that infinite loops can be very subtle and hard to find.

   Take the following program.

   READ X
   READ K
   ADD K TO X UNTIL X IS GREATER THAN 10 OR LESS THAN 1
   PRINT X
Everything is fine if K is given any value other than zero. Even if K is very small, repeatedly adding it to X will cause X to eventually satisfy the “exit condition”. But if X starts somewhere between 1 and 10, and the input to K is 0, the program will be locked into an infinite loop.

While not being extremely subtle, this is an example of the more insidious bugs in a program — those that only show up in certain cases.

Incidentally, what’s actually happening when a computer “locks up” or “freezes” is far from what it appears to be doing. At such times the poor computer hasn’t gone to sleep. No, it’s furiously running around some infinite loop. The operating system of a computer is designed to look at the keyboard at very frequent intervals. But because it can only do one thing at a time, it can’t be always looking to see if you’ve pressed a key. And if it gets into an infinite loop it says to itself “I really must get back and see if someone has pressed a key — just as soon as I get out of this loop”.

Now wouldn’t it be wonderful if some piece of software could examine my program, and the data 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 tie 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.”

Not so! This dream will be forever a dream. Very clever programmers may be able to design something like this 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.

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.

What you do need to know is that there are three ingredients in the computing process — input, the program and output. We shall use a couple of notational conventions.

We shall write program\input = output to indicate the way we link the input and output via the program. So if toaster is a list of instructions for using a domestic toaster, we will write toaster\bread = 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 multiply a number by 2, then double\3 = 6.

Often the output of one program becomes the input of another. We write expressions such as prog1\prog2\input to mean prog1(prog2(input)). This means that the input is
processed by means of the instructions in program prog2. The output then becomes the input to prog1. Just remember to read from right to left.

If bake is a set of instructions for baking bread we might write \( \text{bake} \{ \text{dough} = \text{bread} \} \). Then since \( \text{toaster} \{ \text{bread} = \text{toast} \} \) we could write \( \text{toaster} \{ \text{bake} \{ \text{dough} = \text{toast} \} \} \) to indicate the whole process. And \( \text{double} \{ \text{double} \} \{ \text{3} = 12 \} \).

4. Some Sample Programs

The input to a program could be a physical object, such as a slice of bread, or a number. But 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.

Whenever our programs write something they write it on the same line as the input, coming immediately after it and so the output includes the input. Often the program will explicitly instruct you to erase the input.

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

<table>
<thead>
<tr>
<th>REVERSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Write input backwards.</td>
</tr>
<tr>
<td>2. Erase input.</td>
</tr>
<tr>
<td>3. Halt.</td>
</tr>
</tbody>
</table>

So \( \text{REVERSE} \{ \text{MESSAGE} = \text{EGASSEM} \} \). A palindrome is a string which reads the same forward as backwards, like PUP so it is a string which \( \text{REVERSE} \) 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:

\( \text{REVERSE} \{ \text{REVERSE} \{ \text{MESSAGE} = \text{MESSAGE} \} \) \)

\( \text{COUNT} \) is a program which counts the number of symbols in a string.

<table>
<thead>
<tr>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Count the symbols in the input.</td>
</tr>
<tr>
<td>2. Write this number in words.</td>
</tr>
<tr>
<td>3. Erase input.</td>
</tr>
<tr>
<td>4. Halt.</td>
</tr>
</tbody>
</table>

So \( \text{COUNT} \{ \text{MESSAGE} = \text{SEVEN} \} \). Let’s combine \( \text{COUNT} \) with \( \text{REVERSE} \). There are two ways we can do this since they can be applied in either order. The important thing to notice is that the output is different in each case. \( \text{COUNT} \{ \text{REVERSE} \{ \text{MESSAGE} = \text{SEVEN} \} \) while \( \text{REVERSE} \{ \text{COUNT} \{ \text{MESSAGE} = \text{NEVES} \} \) \)

\( \text{COUNT} \{ \text{COUNT} \{ \text{MESSAGE} = \text{SEVEN} \) \} \) is the number of letters in \( \text{COUNT} \{ \text{MESSAGE} \), that is the number of letters in \( \text{SEVEN} \), which is \( \text{FIVE} \). And since \( \text{FIVE} \) is a four letter word, \( \text{COUNT} \{ \text{COUNT} \{ \text{COUNT} \{ \text{MESSAGE} = \text{FOUR} \} \) \} \) In fact, if you start with any string and repeatedly apply the program \( \text{COUNT} \), eventually you will reach \( \text{FOUR} \). Why?
The next program doesn’t erase the input. Instead it makes a second copy of the input.

<table>
<thead>
<tr>
<th>REPEAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Write “+”.</td>
</tr>
<tr>
<td>2. Copy input.</td>
</tr>
<tr>
<td>3. Halt.</td>
</tr>
</tbody>
</table>

So, \( \text{REPEAT} \text{MESSAGE} = \text{MESSAGE}+\text{MESSAGE} \). What is the difference between \( \text{REPEAT} \text{COUNT} \text{MESSAGE} \) and \( \text{COUNT} \text{REPEAT} \text{MESSAGE} \)? The first gives \( \text{SEVEN}+\text{SEVEN} \) while the second gives \( \text{COUNT} \text{MESSAGE}+\text{MESSAGE} = \text{FIFTEEN} \).

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

<table>
<thead>
<tr>
<th>HALT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Write “!”</td>
</tr>
<tr>
<td>2. Halt.</td>
</tr>
</tbody>
</table>

So \( \text{HALT} \text{HELP} = \text{HELP}! \)

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

<table>
<thead>
<tr>
<th>LOOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Copy the last letter of the input.</td>
</tr>
<tr>
<td>2. Go to step 1.</td>
</tr>
</tbody>
</table>

So \( \text{LOOP} \text{AGH} = \text{AGH} \text{HHHHHHHHHHHHHHH} \ldots \). There is no real output because the program never halts. This program will loop whatever the input. The next one is more discriminating. In fact it will loop, but only if it is told to halt.

<table>
<thead>
<tr>
<th>DISOBEY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. If input = ( \text{LOOP} ) then ( \text{HALT} ).</td>
</tr>
<tr>
<td>2. If input = ( \text{HALT} ) then ( \text{LOOP} ).</td>
</tr>
<tr>
<td>3. Otherwise just halt.</td>
</tr>
</tbody>
</table>

Of course \( \text{DISOBEY} \) doesn’t really disobey its instructions. It only appears to be disobedient. Now \( \text{DISOBEY} \text{LOOP} = \text{LOOP}! \) (the machine actually halts after printing the exclamation mark). But \( \text{DISOBEY} \text{HALT} = \text{HALTTTTTTTTTTTTTTTTTTTT} \ldots \), and so the machine does not halt. For anything else, nothing happens, except for halting. Thus \( \text{DISOBEY} \text{STAY} = \text{STAY} \).

The last of our examples here combines \( \text{HALT} \) and \( \text{LOOP} \) with \( \text{COUNT} \).

<table>
<thead>
<tr>
<th>MAYBE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. If ( \text{COUNT} \text{input} ) is even then ( \text{HALT} ).</td>
</tr>
<tr>
<td>2. Otherwise ( \text{LOOP} ).</td>
</tr>
</tbody>
</table>
So $\text{MAYBE}\text{\textbackslash NO} = \text{NO}$! while $\text{MAYBE}\text{\textbackslash YES} = \text{YESSSSSSSSSSSSSSSSSSSSSSSS}$........... What is $\text{MAYBE}\text{\textbackslash REPEAT}\text{\textbackslash ANYTHING}$? Since $\text{ANYTHING}+\text{ANYTHING}$ has odd length, $\text{MAYBE}$ sends it into $\text{LOOP}$ and so the output is $\text{ANYTHING}+\text{ANYTHINGGGGGGGG}$.... And finally $\text{MAYBE}\text{\textbackslash MAYBE}\text{\textbackslash NO} = \text{NO}!!!!!!!!!!!!!!!!!...........

5. Cannibalism

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 would 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.

What is $\text{REVERSE}\text{\textbackslash REVERSE}$? Clearly it’s $\text{ESREVER}$. And $\text{COUNT}\text{\textbackslash COUNT} = \text{FIVE}$. And $\text{REPEAT}\text{\textbackslash REPEAT} = \text{REPEAT}+\text{REPEAT}$. Feeding $\text{HALT}$ to itself produces $\text{HALT}!$ while $\text{LOOP}\text{\textbackslash LOOP} = \text{LOOPPPPPPPPPPPPPP}$............ And does $\text{DISOBEY}$ do the right thing when told to $\text{DISOBEY}$? No! $\text{DISOBEY}\text{\textbackslash DISOBEY} = \text{DISOBEY}$. Finally, $\text{MAYBE}\text{\textbackslash MAYBE} = \text{MAYBBBBBBBBBBBBBBBBEEE}$...........

6. Predicting Loopiness

We now come to a program which doesn’t exist.

<table>
<thead>
<tr>
<th>PREDICT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. If the input has the form program+data and program&gt;data would halt, write “HALT”.</td>
</tr>
<tr>
<td>2. If the input has the form program+data and program&gt;data would not halt write “LOOP”..</td>
</tr>
<tr>
<td>3. If the input doesn’t have the form program+data, write “?” and halt.</td>
</tr>
<tr>
<td>4. Erase the input.</td>
</tr>
<tr>
<td>5. Halt.</td>
</tr>
</tbody>
</table>

Although we’ve listed what we’d like the program to do we haven't said how it should decide whether program>data would halt. Of course the fact that we can’t think how to do it doesn’t of itself make $\text{PREDICT}$ an impossibility. That is something we’ve yet to prove. But just suppose for the moment that such a $\text{PREDICT}$ existed.

What would $\text{PREDICT}$ do to $\text{COUNT}\text{\textbackslash MESSAGE}$? This has the required form and since $\text{COUNT}\text{\textbackslash MESSAGE} = \text{SEVEN}$, and so halts, $\text{PREDICT}\text{\textbackslash COUNT}\text{\textbackslash MESSAGE} = \text{HALT}$.

$\text{PREDICT}\text{\textbackslash LOOP} = ?$ simply because the input $\text{LOOP}$ doesn’t have the required form with a “+” separating two parts. What about $\text{PREDICT}\text{\textbackslash LOOP}\text{\textbackslash MESSAGE}$? Since $\text{LOOP}\text{\textbackslash MESSAGE} = \text{MESSAGEEEEEEE}$...., it is going into an infinite loop and so $\text{PREDICT}\text{\textbackslash LOOP}\text{\textbackslash MESSAGE} = \text{LOOP}$.

A couple of other combinations are the following. $\text{PREDICT}\text{\textbackslash MAYBE}\text{\textbackslash YES} = \text{LOOP}$. Why? Because $\text{MAYBE}\text{\textbackslash YES} = \text{YESSSSSSSSSSS}$, which doesn’t halt. $\text{PREDICT}\text{\textbackslash MAYBE}\text{\textbackslash NO} = \text{HALT}$ because $\text{MAYBE}\text{\textbackslash NO} = \text{NO}$! which halts.
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.

7. Cannibal Programs

We shall 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, which halts. So REVERSE is a cannibal. COUNT \* COUNT = FIVE, which halts. It, too, is a cannibal. So are REPEAT and HALT. These programs halt for all inputs and so certainly if fed themselves as input.

DISOBEY sometimes halts and sometimes loops, but fed its own description it halts and so it too is a cannibal.

On the other hand, LOOP and MAYBE are not cannibals because they loop when fed their own description. LOOP \* LOOP = LOOPPPP......., which doesn’t halt, so LOOP is not a cannibal. MAYBE \* MAYBE = MAYBEEEE.....

So some programs are cannibals and others are not. Can we predict whether or not a given program is a cannibal? The answer is yes, and no.

If PREDICT exists we can couple it with REPEAT to create a program we’ll call HANNIBAL. HANNIBAL takes any program as input and prints out YES if that program is a cannibal and NO if it isn’t.

In a sense HANNIBAL exists, but only conditionally. If PREDICT exists then so must HANNIBAL. But we’ll shortly show that HANNIBAL cannot possibly exist. This will prove that PREDICT doesn’t exist either. We will have shown that the halting problem is unsolvable. So how would HANNIBAL be defined, if it existed?

<table>
<thead>
<tr>
<th>HANNIBAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. REPEAT.</td>
</tr>
<tr>
<td>2. PREDICT.</td>
</tr>
<tr>
<td>3. Halt.</td>
</tr>
</tbody>
</table>

So HANNIBAL \* COUNT = YES since COUNT is a cannibal and HANNIBAL \* LOOP = NO because LOOP isn't a cannibal. We’re now ready for the final showdown!

7. High Noon

We have one final program to build. I call it MONSTER.

<table>
<thead>
<tr>
<th>MONSTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. HANNIBAL.</td>
</tr>
<tr>
<td>2. DISOBEY.</td>
</tr>
<tr>
<td>3. Halt.</td>
</tr>
</tbody>
</table>

Now if PREDICT exists then so does HANNIBAL and if HANNIBAL exists so does MONSTER. So we have a chain of possibilities. If we can show that MONSTER can’t exist then HANNIBAL can’t exist. And if HANNIBAL doesn’t exist then PREDICT can’t exist.

Here then comes the final question that will clinch it all.

Is MONSTER a cannibal?
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 **MONSTER** is a cannibal.
What does that mean? It means that **MONSTER** will halt if it feeds upon itself, that is, **MONSTER)**MONSTER halts.
Now **MONSTER)**MONSTER

\[ = DISOBEY)HANNIBAL)MONSTER \]
\[ = DISOBEY)PREDICT)REPEAT)MONSTER. \]
\[ = DISOBEY)PREDICT)MONSTER+MONSTER \]
\[ = DISOBEY)HALT \]

(remember that in this case we’re assuming that **MONSTER)**MONSTER halts)

\[ = HALT!!!!!! \ldots \]
But this says that **MONSTER** doesn’t halt when fed its own description, contradicting our assumption.

**Case 2:** Suppose **MONSTER** is not a cannibal.
Now **MONSTER)**MONSTER

\[ = DISOBEY)HANNIBAL)MONSTER \]
\[ = DISOBEY)PREDICT)REPEAT)MONSTER. \]
\[ = DISOBEY)PREDICT)MONSTER+MONSTER \]
\[ = DISOBEY)LOOP \]

(in this case we’re assuming that **MONSTER)**MONSTER doesn’t halt)

\[ = LOOP! \]
But this says that **MONSTER** does halt when fed its own description. Again this contradicts our assumption.

Two possibilities — 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 the data

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 ever the data 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]

Copy the data

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 whatever instructions you have on whatever appears on the board. 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.

Now REVERSE.

Noel rubs out the message RAHRAH and replaces it with HAR HAR.

And DOUBLE again.

The message now becomes HARHARHARHAR.

And finally REPEAT.

Peter was about to start tacking a row of T's on the end of the data but the Prof caught him just in time.

No Pete. Your instructions are to add T's only when it 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.
Noel replaces the **TIXETIXE** with **EXITEXIT**.

And now REPEAT.

*Peter excitedly writes T after T, getting EXIT EXITTTTTTTTTTTTTT...... 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 like 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 data is HALT then REPEAT
else REVERSE
```
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 “else” 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: DOUBLEDUPLICATE I suppose.

Prof: And, Leon 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 REPEATTTTTTTTTTTTTTTT......

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 is not. 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 “else”. 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:

<table>
<thead>
<tr>
<th>PREDICT</th>
</tr>
</thead>
</table>
| 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 a long time.

Prof: How 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!

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 TESTTEST.

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 the word REPEAT to the right of HALT.

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 are at the moment assuming that it is 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 HALTTTTTTTTTTTTTTTTTTTTTT...

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.