

4. THE UNSOLVABLE

1. Are Computers Omnipotent?

Is there anything a computer can't do? Certainly we've witnessed some amazing developments during the fifty 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 which 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 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...

But 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 small-pox will never be eradicated. How short-sighted is the person who declares that so and so will never happen. Yet that is 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 is based on the same logic as ours can possibly solve these unsolvable problems.

This is a different level of impossibility to that of angle-trisection. Our logic is powerful enough to be able to prove that solutions exist, not powerful enough to find these solutions, but powerful enough to prove that we will never be able to find them.

2. The Halting Problem

There is a dream that every novice programmer has. When a computer program is "compiled" (this just means translating it to 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 there first time the 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.

Is it that bug in the CPU chip I've been reading about? Perhaps my computer has a virus? Surely if there was something wrong with my program the compiler would have told me so.

But soon the novice discovers that it wasn't the computer that was at fault, but the program. There was an unforeseen infinite loop in the program.

A very obvious case of 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 finish the 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, much like the way we did for the Turing program above, 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 ever will 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 is 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 instruction 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 *bake>dough = bread*. Then since *toaster>bread = toast* we could write *bake>toaster>dough = toast* to indicate the whole process. And *double>double > 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 is all computers can process. The strings might represent words, or numbers, or pictures but to the computer they are 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 so that only what is written by the program.

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;
3. Halt.

So **REVERSE>MESSAGE = EGASSEM**. A palindrome is a string which reads the same forward as backwards, like PUP so it is a string which **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:

REVERSE>REVERSE>MESSAGE = 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;
4. Halt.

So **COUNT>MESSAGE = SEVEN**. Let's combine **COUNT** with **REVERSE**. There are two ways we can do this since they can be applied in either order. The important thing is to note that the output is different in each case. **COUNT>REVERSE>MESSAGE = SEVEN** while **REVERSE>COUNT>MESSAGE = NEVES**.

COUNT>COUNT>MESSAGE is the number of letters in **COUNT>MESSAGE**, that is the number of letters in **SEVEN**, which is **FIVE**. And since **FIVE** is a four letter word,

COUNT>COUNT>COUNT>MESSAGE = FOUR. In fact if you start with any string and repeatedly apply the program *COUNT*, eventually you will reach *FOUR*.

The next program doesn't erase the input. Instead it makes a second copy of the input.

REPEAT

1. Write "+";
2. Copy input;
3. Halt.

So, *REPEAT>MESSAGE = MESSAGE+MESSAGE*. What is the difference between *REPEAT>COUNT>MESSAGE* and *COUNT>REPEAT>MESSAGE*. The first gives *SEVEN+SEVEN* while the second gives *COUNT>MESSAGE+MESSAGE = FIFTEEN*.

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

HALT

1. Write "!";
2. Halt.

So *HALT>HELP = 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.

So *LOOP>AGH = AGHHHHHHHHHHHHHHHH.....*. 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.

DISOBEY

1. If input = *LOOP* then *HALT*;
2. If input = *HALT* then *LOOP*;
3. Otherwise just halt.

In fact *DISOBEY* doesn't really disobey its instructions. It only appears to be disobedient. Now *DISOBEY>LOOP = LOOP!* (the machine actually halts after printing the exclamation mark). But *DISOBEY>HALT = HALTTTTTTTTTTTTT.....* and so the machine does not halt. For anything else, nothing happens, except for halting. Thus *DISOBEY>STAY = STAY*.

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

MAYBE

1. If *COUNT>input* is even then *HALT*;
2. Otherwise *LOOP*.

So *MAYBE>NO* = *NO!* while *MAYBE>YES* = *YESSSSSSSSS.....* What is *MAYBE>REPEAT>ANYTHING*? Since *ANYTHING+ANYTHING* has odd length, *MAYBE* sends it into *LOOP* and so the output is *ANYTHING+ANYTHINGGGGGG....* And finally *MAYBE>MAYBE>NO* = *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 the programs you write to convert it to a form which is convenient for the computer. It is not uncommon for compilers to be written in the same language as the programs they are 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 the instructions — not just the name of the program. But for simplicity in this discussion let us work with the names only.

What is *REVERSE>REVERSE*? Clearly it is *ESREVER*. And *COUNT>COUNT* = *FIVE*. And *REPEAT>REPEAT* = *REPEAT+REPEAT*. Feeding *HALT* to itself produces *HALT!* while *LOOP>LOOP* = *LOOPPPPPPPP.....* And does *DISOBEY* do the right thing when told to *DISOBEY*? No! *DISOBEY>DISOBEY* = *DISOBEY*. Finally, *MAYBE>MAYBE* = *MAYBEEEEEEEE.....*

6. Predicting Loopiness

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

PREDICT

1. If input does not have the form prog+data, write "?" and halt;
2. Otherwise if prog>data would halt, write "*HALT*".
3. Otherwise write "*LOOP*";
4. Erase the input;
5. Halt.

Although we have listed what we would like the program to do we haven't said how it should decide whether prog>data will halt. Of course the fact that we can't think how to do it doesn't of itself make *PREDICT* an impossibility. That we have yet to prove. But just suppose for the moment that such a *PREDICT* existed.

What would *PREDICT* do to *COUNT+MESSAGE*? This has the required form and since *COUNT>MESSAGE* = *SEVEN*, and so halts, *PREDICT>COUNT+MESSAGE* = *HALT*.

PREDICT>LOOP = ? simply because the input *LOOP* does not have the required form with a + separating two parts. What about *PREDICT>LOOP+MESSAGE*? Since *LOOP>MESSAGE* = *MESSAGEEEEEEE.....*, that is it is going into an infinite loop, *PREDICT>LOOP+MESSAGE* = *LOOP*.

A couple of other combinations are the following. *PREDICT>MAYBE>YES* = *LOOP*. Why? Because *MAYBE>YES* = *YESSSSSSSS.....*, which does not halt. *PREDICT>MAYBE>NO* = *HALT* because *MAYBE>NO* = *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 are 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 have bred.

REVERSE>*REVERSE* = *ESREVER*, which halts. So *REVERSE* is a cannibal. *COUNT*>*COUNT* = *FIVE*, which halts. It, too, is a cannibal. So is *REPEAT* and *HALT*. These programs halted for all inputs and so certainly if fed themselves.

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* = *LOOPPPPPPP*....., which does not 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 is not.

In that sense *HANNIBAL* exists. It exists conditionally. If *PREDICT* exists then so must *HANNIBAL*. But we will shortly show that *HANNIBAL* cannot exist. This will prove that *PREDICT* does not exist and so will show that the halting problem is unsolvable. So how would *HANNIBAL* be defined, if it existed.

<p>HANNIBAL</p> <ol style="list-style-type: none"> 1. <i>REPEAT</i>; 2. <i>PREDICT</i>; 3. If output = <i>HALT</i>, then replace by <i>YES</i>; 4. If output = <i>LOOP</i>, then replace by <i>NO</i>; 5. Halt.

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*.

<p>MONSTER</p> <ol style="list-style-type: none"> 1. <i>HANNIBAL</i>; 2. <i>DISOBEY</i>. 3. Halt.

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* couldn'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 we're assuming that *MONSTER*>*MONSTER* halts)

= *LOOP*

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* (this time we're assuming that *MONSTER*>*MONSTER* doesn't halt)

= *HALT*

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 is 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. Ergo it cannot! The halting problem is unsolvable!