8 Noncomputability and Undecidability

this thing is too heavy for thee
EXODUS 18: 18

though a man labour to seek it out, yet he shall not find it
ECCLESIASTES 8: 17

OR
Sometimes You Can't Get It Done At All!

In April 1984, *Time* magazine ran a cover story on computer software. In the otherwise excellent article, there was a paragraph that quoted the editor of a software magazine as saying:

> Put the right kind of software into a computer, and it will do whatever you want it to. There may be limits on what you can do with the machines themselves, but there are no limits on what you can do with software.

In a way, the results of Chapter 7 already contradict this claim, by exhibiting problems that are provably intractable. We might argue, however, that intractability is really a consequence of insufficient resources. Given enough time and memory space (albeit, unreasonably large amounts), perhaps any algorithmic problem can, in principle, be solved by the right software. Indeed, the reasons that people often fail in getting their computers to do what they want seem to fall roughly into three categories: insufficient money, insufficient time, and insufficient brains. With more money one could buy a larger and more sophisticated computer, supported by better software, and perhaps then get the job done. With more time one could wait longer for time-consuming algorithms to terminate, and with more brains one could perhaps invent algorithms for problems that seem to defy solution.

The algorithmic problems we wish to discuss in this chapter are such that no amount of money, time or brains will suffice to yield solutions. We still require, of course, that algorithms terminate for each legal input within some finite amount of time, but we now allow that time to be unlimited. The algorithm can take as long as it wishes on each input, but it must eventually stop and produce the desired output. Similarly, while working on an input, the algorithm will be given any amount of memory it asks for. Even so, we shall see interesting and important problems, for which there simply are no algorithms, and it doesn't matter how clever we are, or how sophisticated and powerful our computers are.

Such facts have deep philosophical implications, not only on the limits of man-made machines, but also on our own limits as mortals with finite mass.

Even if we were given unlimited pencil and paper, and an unlimited life span, there would be precisely defined problems we could not solve. There are people who are opposed to drawing such extended conclusions from mere algorithmic results for various reasons. In consideration of the fact that the issue in this extended form definitely deserves a much broader treatment, we shall stick to pure algorithmics here, and leave the deeper implications to philosophers and neurobiologists. However, the reader should keep the existence of such implications in mind.

The Rules of the Game

Just to reclarify things, it should be emphasized that questions regarding the computer's ability to run companies, make good decisions, or love, are not relevant to our present discussions, since they do not involve precisely defined algorithmic problems.

Another fact worth recalling is the requirement that an algorithmic problem be associated with a set of legal inputs, and that a proposed solution apply to all inputs in the set. As a consequence, if the set of inputs is *finite*, the problem always admits a solution. For a decision problem whose sole legal inputs are the items $I_1, I_2, \ldots, I_K$, there is an algorithm that "contains" a table with the K answers. The algorithm might read:

(1) if the input is I_1 then output "yes" and stop;
(2) if the input is I_2 then output "yes" and stop;
(3) if the input is I_3 then output "no" and stop;
...
(K) if the input is I_K then output "yes" and stop.

This works, of course, because the finiteness of the set of inputs makes it possible to tabulate all input/output pairs and "hardwire" them into the algorithm. It might be difficult to carry out the tabulation (that is, to *construct* such a table-driven algorithm), but we are not interested in this "meta-difficulty" here. For our present purposes it suffices that finite problems always have solutions. It is the problems with infinitely many inputs that are really interesting. In such cases, a finite algorithm must be able to cope with infinitely many cases, prompting one to question the very existence of such algorithms for all problems.

To be able to state our claims as precisely, yet as generally, as possible, the reader will have to put up with a certain kind of terminological looseness in this chapter, to be fully justified in the next. Specifically, an arbitrary, but fixed, high-level programming language L is hereby implicitly assumed to be the medium for expressing algorithms, and the word "algorithm" will be used as a synonym for "program in L." In particular, when we say "no algorithm exists," we really mean that no program can be written in the

language L. This convention might look a little pretentious here, apparently weakening our claims considerably. Not so. In the next chapter we shall see that, under the unlimited resources assumption, all programming languages are equivalent. Thus, if no program can be written in L, no program can be written in *any* effectively implementable language, running on any computer of any size or shape, now or at any time in the future.

Tiling Problem: An Example

The following example is reminiscent of the monkey puzzle problem of Chapter 7. The problem involves covering large areas using square tiles, or cards, with colored edges, such that adjacent edges are monochromatic. A tile is a 1-by-1 square, divided into four by the two diagonals, each quarter colored with some color (see Figure 8.1). As with monkey cards, we assume that the tiles have fixed orientation and cannot be rotated. (In this case the assumption is, in fact, necessary. Can you see why?)

The algorithmic problem inputs some finite set T of tile descriptions, and asks whether any finite area, of any size, can be covered using only tiles of the kinds described in T, such that the colors on any two touching edges are the same. It is assumed that an unlimited number of tiles of each type is available, but that the number of types of tiles is finite.

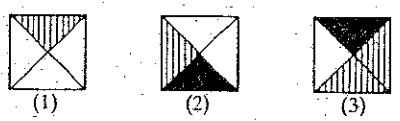
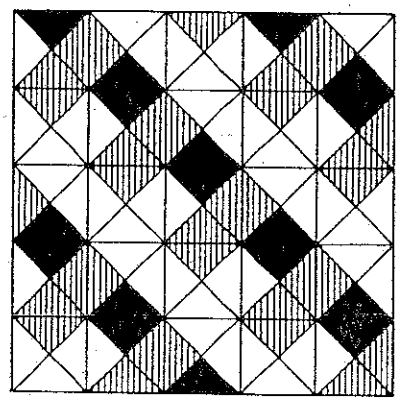


Figure 8.1

Tile types that can tile any area.



Think of tiling a house. The input T is a description of the various types of tiles available, and the color-matching restriction reflects a rule enforced by an interior designer for aesthetic reasons. The question we would like to ask ahead of time is this: can a room of any size be tiled using only the available tile types, while adhering to the restriction?

This algorithmic problem and its variants are commonly known as **tiling problems**, but are sometimes called **domino problems**, the reason being the domino-like restriction on touching edges.

Figure 8.1 shows three tile types and a 5 by 5 tiling, and the reader will have no difficulty verifying that the pattern in the lower portion of the figure can be repeated to yield a tiling of any area whatsoever. In contrast, if we exchange the bottom colors of tiles (2) and (3) it can be shown quite easily that even very small rooms cannot be tiled at all. Figure 8.2 is meant as an illustration of this fact. An algorithm for the tiling problem, then, should answer "yes" on the inputs of Figure 8.1 and "no" on those of Figure 8.2.

The problem is to somehow mechanize or "algorithmicize" the reasoning employed in generating these answers. And here comes the interesting fact: this reasoning is impossible to mechanize. There is no algorithm, and there never will be, for solving the tiling problem! More specifically, for any algorithm we might design for the problem, there will always be input sets T (there will actually be infinitely many such sets) upon which the algorithm will either run forever and never terminate, or terminate with the wrong answer.

How can we make such a general claim, without restricting the basic operations allowed in our algorithms? Surely, if *anything* is allowed, then the following two-step procedure solves the problem:

(1) if the types in T can tile any area, output "yes" and stop;
(2) otherwise, output "no" and stop.

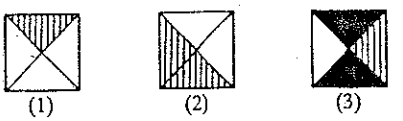
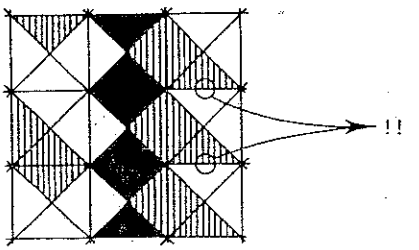


Figure 8.2

Tile types that cannot tile even small areas.



The answer lies in our use of "algorithm" here to stand for a program in a conventional programming language *L*. No program in any effectively executable language can correctly implement the test in line (1) of the procedure, and hence, for our purposes, such a "procedure" will not be considered an algorithm at all.

An algorithmic problem that admits no algorithm is termed **noncomputable**; if it is a decision problem, as is the case here and with most of the examples that follow, it is termed **undecidable**. The tiling, or domino, problem is therefore undecidable. There is no way we can construct an algorithm, to be run on a computer, any computer, regardless of the amount of time and memory space required, that will have the ability to decide whether arbitrary finite sets of tile types can tile areas of any size.

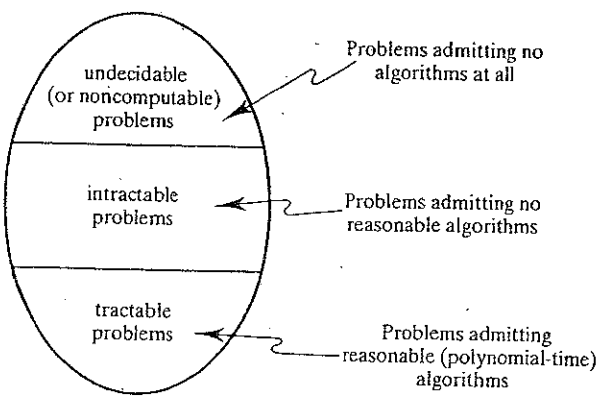
We can now refine the sphere of algorithmic problems appearing in Chapter 7 (see Figure 7.6), taking noncomputable problems into account. Figure 8.3 is the current version.

It is interesting to observe that the following, slightly different-looking problem, is actually equivalent to the one just described. In this version, instead of requiring that T be able to tile finite areas of any size, we require that T be able to tile the entire integer grid; that is, the entire infinite plane. One direction of the equivalence (if we can tile the entire plane then we can tile any finite area) is trivial, but the argument that establishes the other direction is quite delicate, and the reader is encouraged to try to find it for himself.

Interestingly, the undecidability of this version means that there must be tile sets T that can be used to tile the entire grid, but not periodically. That is, while such a T admits a complete tiling of the grid, the tiling, unlike that of Figure 8.1, does *not* consist of a finite portion that repeats indefinitely in all directions. The reason for this is that otherwise we could decide the problem by an algorithm that would proceed to check all finite areas exhaustively, searching either for a finite area that cannot be tiled at all or for one that admits a repeat in all directions.

Figure 8.3

The sphere of algorithmic problems: Version II.



Unboundedness Can Be Misleading

Some people react to results like this by saying: "Well, obviously the problem is undecidable, because a single input can give rise to a potentially infinite number of cases to check, and there is no way you can get an infinite job done by an algorithm that must terminate after finitely many steps." And indeed, here a single input T apparently requires all areas of all sizes to be checked (or, equivalently, as mentioned above, a single area of infinite size), and there seems to be no way to *bound* the number of cases to be checked.

This unboundedness-implies-undecidability principle is quite wrong, and can be very misleading. It is just like saying that any problem that seems to require exponentially many checks is necessarily intractable. In Chapter 7 we saw the two problems of Hamiltonian paths and Eulerian paths, both of which seemed to require a search through all of the exponentially many paths in the input graph. The second problem, however, was shown to admit an easy polynomial-time algorithm. With undecidability it is also possible to exhibit two very similar variants of a problem, both of seemingly unbounded nature, which contrast in a rather surprising way to violate the principle.

The inputs in both cases contain a finite set T of tile types and two locations V and W on the infinite integer grid. Both problems ask whether it is possible to connect V to W by a "domino snake" consisting of tiles from T, with every two adjacent tiles having monochromatic touching edges (see Figure 8.4). Notice that a snake originating from V might twist and turn erratically, reaching unboundedly distant points before converging on W. Hence, on the face of it, the problem requires a potentially infinite search, prompting us to conjecture that it, too, is undecidable.

It is interesting, therefore, that the decidability of the domino snake problem depends on the portion of the plane available for laying down the connecting tiles. Clearly, if this portion is finite the problem is trivially decidable, as there are only finitely many possible snakes that can be positioned in a given finite area. The distinction we wish to make is between

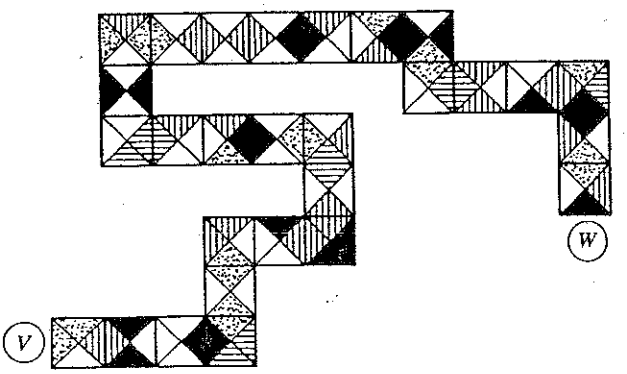


Figure 8.4

A domino snake connecting V to W.

two *infinite* portions, and is quite counter-intuitive. If snakes are allowed to go anywhere (that is, if the allowed portion is the entire plane), the problem is decidable, but if the allowed area is only a half of the plane (say, the upper half), the problem becomes undecidable! The latter case seems to be "more bounded" than the former, and therefore perhaps "more decidable." The facts, however, are quite different.

Word Correspondence and Syntactical Equivalence

Here are two additional undecidable problems. The first, the **word correspondence problem**, involves forming a word in two different ways. Its inputs are two groups of words over some finite alphabet. Call them the *X*s and the *Y*s. The problem asks whether it is possible to concatenate words from the *X* group, forming a new word, call it *Z*, so that concatenating the corresponding words from among the *Y*s form the very same compound word *Z*. Figure 8.5(a) shows an example consisting of five words in each group, where the answer is "yes," since choosing the words for concatenation according to the sequence 2, 1, 1, 4, 1, 5 from either the *X*s or the *Y*s yields the same word, "*aabbabbbabaabbaba*." On the other hand, the input described in Figure 8.5(b), which is obtained from Figure 8.5(a) by simply removing the first letter from the first word of each group, does not admit any such choice, as the reader can verify. Its answer is therefore "no."

The word correspondence problem is undecidable. There is no algorithm that can distinguish, in general, between the likes of Figures 8.5(a) and 8.5(b). The unbounded nature of the problem stems from the fact that the number of words that need to be chosen to yield the common compound word is not

gure 8.5

An instance of the word correspondence problem.

	1	2	3	4	5
X	*abb*	*a*	*bab*	*baba*	*aba*
Y	*bbab*	*aa*	*ab*	*aa*	*a*

(a) Admits a correspondence: 2, 1, 1, 4, 1, 5

	1	2	3	4	5
X	*bb*	*a*	*bab*	*baba*	*aba*
Y	*bab*	*aa*	*ab*	*aa*	*a*

(b) Admits no correspondence

bounded. However, here too, we can point to a seemingly "less bounded" variant, in which it seems that there are more cases to check for, but which nevertheless is decidable. In it, the inputs are as before, but there is no restriction on the way choices are made from the *X*s and *Y*s; even the number of words selected need not be the same. We are asking simply if it is possible to form a common compound word by concatenating some words from *X* and some words from *Y*. In Figure 8.5(b), which gave rise to a "no" for the standard version of the problem, the word "*babaa*," for example, can be obtained from the *X*s by the sequence 3, 2, 2, and from the *Y*s by 1, 2, and hence is a "yes" in the new version. This more liberal problem actually admits a fast polynomial-time algorithm!

The second problem concerns the syntax of programming languages. Suppose someone provides us with the syntax rules of some language, say in the diagrammatic form of Figure 3.1. If someone else comes along with a different set of rules, we might be interested in knowing whether the two definitions are equivalent, in the sense that they define the same language; that is, the same syntactic class of statements (or programs). This problem is of relevance to the construction of compilers, since compilers, among their other chores, are also responsible for recognizing the syntactic validity of their input programs. In order to do so they have a set of syntactic rules built in. It is quite conceivable that in the interest of making a compiler more efficient its designer would want to replace the set of rules with a more compact set. Clearly, it is important to know in advance that the two sets are interchangeable.

This problem is also undecidable. No algorithm exists, which, upon reading from the input two sets of syntax rules, will be able to decide in finite time whether they define precisely the same language.

Problems with Outputs Are No Better

We should perhaps reemphasize the fact that technical convenience is the only reason we limit ourselves to decision problems here. Each of the undecidable problems we describe has variants that ask for outputs, and which are also noncomputable. Trivial variants are those that are basically decision problems in (rather transparent) disguise. An example is the problem that asks, for a given set of colored-tile types *T*, to output the size of the smallest area not tileable by *T*, and to output 0 if every finite area is tileable. It is clear that this problem cannot be computable, since the distinction in the output between 0 and all the other numbers is precisely the distinction between "yes" and "no" in the original problem.

- More sophisticated problems hide the ability to make an undecidable decision far better. The following problem is noncomputable too. In order to define it, let us say that a finite portion of the grid is a **limited area** for a set *T* of tiles, if it can be

tiled legally by T, but cannot be extended in any way by more tiles without violating the tiling rules. Now, the problem involves finding particular sets T that have large limited areas. Specifically, the algorithmic problem is given a number N (which should be at least 2), and is asked to output the size of the largest limited area, for *any* set of tiles, that involves no more than N colors. The reader should convince himself that for any $N > 1$ this number is well defined.

It is far from obvious that in order to solve the limited area problem we need the ability to decide the likes of the tiling problem. And so, although the problem gives rise to a well-defined, non-yes/no function of N, this function simply cannot be computed algorithmically.

Algorithmic Program Verification

In Chapter 5 we asked whether computers can verify our programs for us. That is, we were after an **automatic verifier** (see Figure 5.4). Specifically, we are interested in the decision problem whose inputs are the description of an algorithmic problem and the text of an algorithm, or program, that is believed to solve the given problem. We are interested in determining algorithmically whether the given algorithm solves the given problem or not. In other words, we want a "yes" if for each of the problem's legal inputs the algorithm will terminate and its outputs will be precisely as specified by the problem, and we want a "no" if there is even one input for which the algorithm either fails to terminate or terminates with the wrong results. Note that the problem calls for an algorithm that works for *every* choice of a problem-algorithm pair.

Clearly, the **verification problem** cannot be discussed without being more specific about the allowed inputs. Which programming language is to be used for coding the input algorithms? Which specification language is to be used for describing the input algorithmic problems?

As it happens, even very humble choices of these languages render the verification problem undecidable. Even if the allowed programs can manipulate only integers or strings of symbols, and can employ only the basic operations of addition or attachment of symbols to strings, they cannot be verified algorithmically. Candidate algorithmic verifiers might work nicely for many sample inputs, but the general problem is undecidable, meaning that there will always be algorithms that the verifier will not be able to verify.

As discussed in Chapter 5, this implies the futility of hoping for a software system that would be capable of automatic program verification. It also reduces the hope for optimizing compilers capable of transforming programs into optimally efficient ones. Such a compiler might not even be able to tell, in general, whether a new candidate version even solves the same problem as the original program, let alone whether it is more efficient.

However, not only full verification is undecidable, we cannot even decide whether a given algorithm merely *terminates* on all its legal inputs. Moreover, it is not even decidable whether the algorithm terminates on *one* given input! These problems of termination deserve special attention.

The Halting Problem

Consider the following algorithm *A*:

(1) while $X \neq 1$ do the following: $X \leftarrow X - 2$;
(2) stop.

In other words, *A* repeatedly reduces *X* by 2 until *X* becomes equal to 1. Assuming that its legal inputs consist of the positive integers ⟨1, 2, 3, . . .⟩, it is quite obvious that *A* halts precisely for *odd* inputs. An even number will be decreased repeatedly by 2 until it reaches 2, and will then "miss" the 1, running forever through 0, −2, −4, −6, and so on. Hence, for this particular algorithm, deciding whether a legal input will cause it to terminate is trivial: just check whether the input is odd or even and answer accordingly.

Here is another, similar looking, algorithm *B*:

(1) while $X \neq 1$ do the following:
  (1.1) if *X* is even do $X \leftarrow X/2$;
  (1.2) otherwise (*X* is odd) do $X \leftarrow 3X + 1$;
(2) stop.

The algorithm *B* repeatedly halves *X* if it is even, but *increases* it more than threefold if it is odd. For example, if *B* is run on 7, the sequence of values is: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, eventually resulting in termination. Actually, if we try running the algorithm on some arbitrary positive integer we will either find that it terminates or we will see an erratic sequence of values that provides no evidence of eventually converging or diverging. The sequence of values is often quite unusual, reaching surprisingly high values, and fluctuating unpredictably before it reaches 1. Indeed, over the years *B* has been tested on numerous inputs and, whenever there was sufficient time to wait, it has always terminated. Nevertheless, no one has been able to prove that it terminates for *all* positive integers, although most people believe that it does. The question of whether or not this is the case is actually a difficult open problem in the branch of mathematics known as **number theory**. Now, if indeed *B* (or any other program) terminates for all its inputs, there *is* a proof of this fact, as discussed in Chapter 5, but for *B* no one has found such a proof yet. These examples illustrate just how difficult it is to analyze the termination properties of even very simple algorithms.

Let us now define a specific version of the problem of algorithmic termination, called the **halting problem**. We define it here in terms of our agreed on high-level programming language *L*.

The problem has two inputs, the text of a legal program *R* in the language *L* and a potential input *X* to *R*.* The halting problem asks whether *R* would

*We can assume that *R* expects just one input, since a number of inputs can be encoded into a single string of symbols, with the various parts separated by some special symbol like "#." This point is treated in more detail in Chapter 9.

have terminated had we run it on the input X, a fact we denote by $R(X)\downarrow$. The case of R not terminating, or *diverging*, on X is denoted $R(X)\uparrow$ (see Figure 8.6).

As already stated, the halting problem is undecidable, meaning that there is no way to tell, in a finite amount of time, whether a given R will terminate on a given X. In the interest of solving this problem, it is tempting to propose an algorithm that will simply run R on X and see what happens. Well, if and when execution terminates, we can justly conclude that the answer is "yes." The difficulty is in deciding when to stop waiting and say "no." We cannot simply give up at some point and conclude that since R has not terminated until now it never will. Perhaps if we had left R just a little longer it *would* have terminated. Running R on X, therefore, does not do the job, and, as stated, nothing can do the job, since the problem is undecidable.

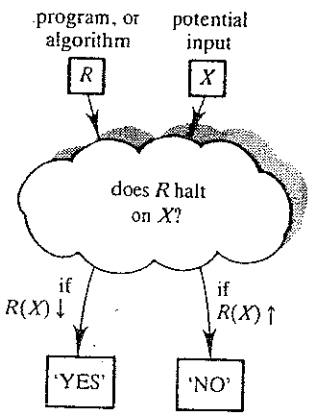
Proving Undecidability

How do we prove that a problem P is undecidable? How do we establish the fact that no algorithm exists for solving P, no matter how clever the algorithm designer is?

This situation is similar to that described in Chapter 7 for the NP-complete problems. First there has to be one initial problem, whose undecidability is established using some direct method. In the case of undecidability this role is played by the halting problem, which we shall prove undecidable later. Once such a first undecidable problem exists, the undecidability of other problems is established by exhibiting **reductions** from problems already known to be undecidable to the problems in question. The difference is that here a reduction from problem P to problem Q need not necessarily be bounded; it can take any amount of time or memory space. All that is required is that there is an algorithmic way of transforming a P-input into a

Figure 8.6

The halting problem.



Q-input, in such a way that P's yes/no answer to an input is precisely Q's answer to the transformed input. In this way, if P is already known to be undecidable Q must be undecidable too. The reason is that otherwise we could have solved P by an algorithm that would take any input, transform it into an input for Q and ask the Q algorithm for the answer. Such a hypothetical algorithm for Q is called an oracle, and the reduction can be thought of as showing that P is decidable given an oracle for deciding Q. In terms of decidability, this shows that P cannot be any worse than Q.

- For example, it is relatively easy to reduce the halting problem to the problem of verification. Assume that the halting problem is undecidable (we will actually prove this directly in the next section). To show that the verification problem is undecidable too we have to show that a verification oracle would enable us to decide the halting problem. Well, given an algorithm R and one of its potential inputs X, we transform the pair $\langle R, X\rangle$, which is an input to the halting problem, into the pair $\langle P, R\rangle$ which is an input to the verification problem. The algorithm R remains the same, and the algorithmic problem P is described by specifying that X is the sole legal input to R, and that the output for this one input is unimportant. Now, to say that R is (totally) correct with respect to this rather simplistic problem P is just to say that R terminates on all legal inputs (i.e., on X) and produces some output, which is really just to say that R terminates on X. In other words, the verification problem says "yes" to $\langle P, R\rangle$ if, and only if, the halting problem says "yes" to $\langle R, X\rangle$. Consequently, the verification problem is undecidable, since otherwise we could have solved the halting problem by constructing an algorithm that would first transform any $\langle R, X\rangle$ into the corresponding $\langle P, R\rangle$, and then use the verification oracle for deciding the correctness of $\langle P, R\rangle$.

Other reductions are far more subtle. What on earth have tiling problems got to do with algorithmic termination? How do we reduce domino snakes to two-way word formations? Despite the apparent differences, these problems are all intimately related, by being interreducible, and in Chapter 9 we shall discuss one of these reductions in some detail.

Proving the Undecidability of the Halting Problem

We shall now prove that the halting problem, as described in Figure 8.6, is undecidable. This, as explained earlier, is carried out directly—not by a reduction from some other problem. Now, faithful to our terminological convention, what we really have to show is that there is no program in the agreed on high-level programming language L that solves the halting problem for programs in L. (As mentioned earlier, this language-dependent fact will be extended into a far more general statement in Chapter 9.)

More precisely, we want to prove the following claim:

> There is no program in L which, upon accepting any pair $\langle R, X\rangle$, consisting of the text of a legal program R in L and a string of symbols X, terminates after some finite

amount of time, and outputs "yes" if R halts when run on input X and "no" if R does not halt when run on input X.

Such a program, if it exists, is itself just some legal program in L; it can use as much memory space and as much time as it requests, but it must work, as described, for *every* pair $\langle R, X \rangle$.

We shall prove that a program satisfying these requirements is nonexistent, *by contradiction*. In other words, we shall assume that such a program does exist, call it Q, and shall derive a downright contradiction from that assumption. Throughout, the reader should be wary, making sure that anything we do is legal and according to the rules, so that when the contradiction becomes apparent we shall be justified in pointing to the assumption about Q's existence as the culprit.

Let us now construct a new program in L, call it S, as illustrated schematically in Figure 8.7. This program has a single input, which is a legal program W in L. Upon reading its input, S makes another copy of it. This copying is obviously possible in any high-level programming language, given sufficient memory. Recalling that the (assumed-to-exist) program Q expects

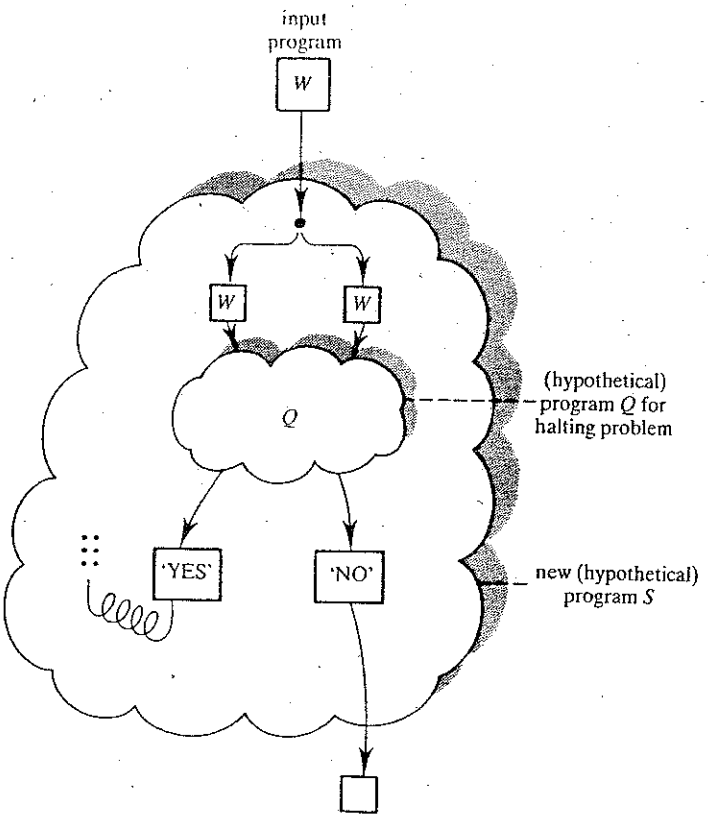


Figure 8.7

Proving undecidability of the halting problem: The program S.

two inputs, the first of which is a program, the next thing that S does is to activate Q on the input pair consisting of the two copies of W. The first of these is indeed a program, as expected by Q, and the other is considered to be an input string, though that string just happens to be the text of the same program, W. This activation of Q can be carried out by calling Q as a subroutine with parameters W and W, or by inserting the (assumed-to-exist) text of Q into the right place, and assigning the values W and W to its expected input variables R and X, respectively.

The program S now waits for this activation of Q to terminate, or, in the metaphorical terms used earlier in the book, S's processor Runaround_S waits for Q's processor Runaround_Q to report back to headquarters. The point is that by our assumption Q must terminate, since, as explained, its first input is a legal program in L, and its second, when considered as a string of symbols, is a perfectly acceptable potential input to W.* And so, by our hypothesis, Q must eventually terminate, saying "yes" or "no." We now instruct the new program S to react to Q's termination as follows. If Q says "yes," S is to promptly enter a self-imposed infinite loop, and if it says "no," S is to promptly terminate (the output being unimportant). This can also be achieved in any high-level language, by the likes of:

```
. . .
(17) if OUT="yes" then goto (17), otherwise stop;
```

where OUT is the variable containing Q's output. This completes the construction of the (strange-looking) program S, which, it should be emphasized, is a legal program in the language L, assuming, of course, that Q is.

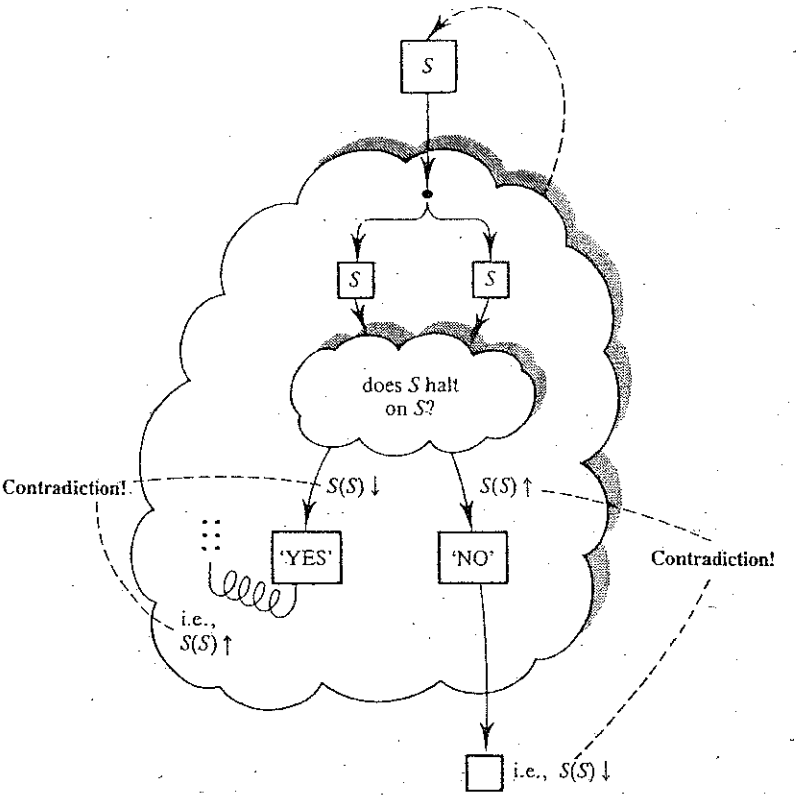
We now want to show that there is something quite wrong with S. There is a logical impossibility in the very assumption that S can be constructed. In exposing this impossibility we shall rely on the obvious fact that, for every choice of a legal input program W, the new program S must either terminate or not. We shall show, however, that there is a certain input program for which S cannot terminate, but it also cannot *not* terminate! This is clearly a logical contradiction, and we shall use it to conclude that our assumption about Q's existence is faulty, thus proving that the halting problem is indeed undecidable.

■ The input program W that causes this impossibility is S itself. To see why S as an input to itself causes a contradiction, assume for the moment that S, when given its own text as an input, terminates. Let us now work through the details of what really happens to S when given its own text as an input. First, two copies are made of the input S (see Figure 8.8), and these are then fed into the (assumed-to-exist) program Q. Now, by our hypothesis, Q must terminate after some finite time, with

*The fact that the very string W itself is considered an input to the program W should not bother the reader too much; it is perhaps a little strange, but not impossible. Any compiler written in the language it compiles can compile itself.

Figure 8.8

Proving undecidability of the halting problem: S swallowing S.



an answer to the question of whether its first input terminates on its second. Now, since Q is presently working on inputs S and S, and since we assumed that S indeed terminates on S, it must halt eventually and say "yes." However, once it has terminated and said "yes," execution enters the self-imposed infinite loop and never terminates. But this means that on the assumption that S applied to S terminates (an assumption that caused Q to say "yes"), we have discovered that S applied to S does *not* terminate! We thus conclude that it is impossible that S terminates on S. This leaves us with one remaining possibility, namely, that S applied to S does not terminate. However, as can be easily verified with the help of Figure 8.8, this assumption leads in a very similar way to the conclusion that S applied to S *does* terminate, since when Q says "no" (and it *will* say "no" because of our assumption) the execution of S applied to S promptly terminates. Thus, it is also impossible that S does not terminate when run on S.

In other words, the program S cannot terminate when run on itself and it cannot not terminate! Consequently, something is very wrong with S itself. However, since all other parts of S were constructed quite legally, the only part that can be held responsible is the program Q, whose assumed existence enabled us to construct S the way we did. The conclusion is that a program Q, solving the halting problem as required, simply cannot exist.