15

Computability

However unapproachable these problems may seem to us and however helpless we stand before them, we have, nevertheless, the firm conviction that their solution must follow by a finite number of purely logical processes. . .This conviction of the solvability of every mathematical problem is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason; for in mathematics these is no ignorabimus.
David Hilbert, *Address to the Second International Congress of Mathematicians*, 8 August 1900

Gödel's paper has reached me at last. I am very suspicious of it now but will have to swot up the Zermelo-van Neumann system a bit before I can put objections down in black & white.
Alan Turing, letter to Max Newman, 1940

The small subset of Scheme introduced in Chapter 3 is sufficient to define a procedure that produces any possible computation. What remains to be considered, however, is what problems can and cannot be solved by mechanical computation. This is the *computability* question: a problem is computable if it can be solved by some algorithm.

computability

Before getting to the question of computability, we consider a similar question for declarative knowledge: are there true statements that cannot be proven by *any* proof?

Section 15.2 introduces the *Halting Problem*, a problem that cannot be solved by any algorithm. Section 15.3 sketches Alan Turing's proof that the Halting Problem is noncomputable. Section 15.4 explores how to use the Halting Problem to recognize other problems that are noncomputable.

15.1 Mechanizing Reasoning

Humans have been attempting to mechanize reasoning for thousands of years. An early attempt was Aristotleï£¡s *Organon* in approximately 350 BC. Aristotle developed *syllogisms*, rules of inference that codify logical deductions.

Euclid went beyond Aristotle by developing a formal axiomatic system. An

axiomatic system *axiomatic system* is a formal system consisting of a set of axioms and a set of inference rules. The goal of an axiomatic system is to codify knowledge in some domain.

The axiomatic system Euclid developed in *The Elements* concerned constructions that could be drawn using just a straightedge and a compass.

Euclid started with five axioms (more commonly known as *postulates*); an example axiom is: *A straight line segment can be drawn joining any two points.* In addition to the postulates, Euclid states five *common notions*, which could be considered inference rules. An example of a common notion is: *The whole is greater than the part.*

Starting from the axioms and common notions, along with a set of definitions (e.g., defining a *circle*), Euclid proved 468 propositions mostly about geometry and number theory. A *proposition* is a statement that is stated precisely enough to be either true or false. Euclid's first proposition is: given any line, an equilateral triangle can be constructed whose edges are the length of that line.

proposition

proof A *proof* of a proposition in an axiomatic system is a sequence of steps that ends with the proposition. Each step must follow from the axioms using the inference rules. Nearly all of Euclid's proofs are constructive: most of the propositions state that a particular thing exists, and the proof is a series of steps that construct one of the target things. The steps follow from the postulates and the inference rules to prove that the thing resulting at the end satisfies the requirements of the proposition.

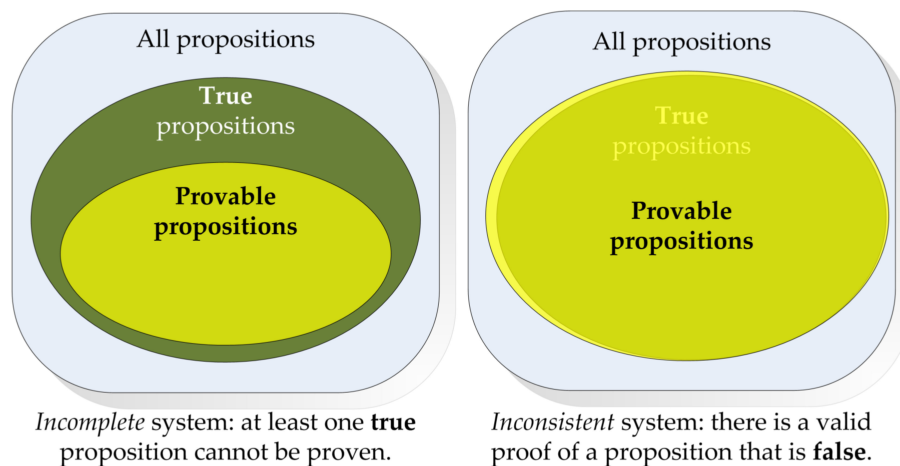
consistent A *consistent* axiomatic system is one that can never derive contradictory statements by starting from the axioms and following the inference rules. If a system can generate both A and *not* A for any proposition A, the system is inconsistent. If the system cannot generate any contradictory pairs of statements it is consistent. Euclid's system is consistent; there is no way to derive contradictory statements in the system.

complete A *complete* axiomatic system can derive all true statements by starting from the axioms and following the inference rules. This means if a given proposition is true, some proof for that proposition can be found in the system. Since we do not have a clear definition of *true* (if we defined true as something that can be derived in the system, all axiomatic systems would automatically be complete by definition), we state this more clearly by saying that the system can decide any statement. This means, for a given statement A, a complete axiomatic system would be able to derive either A or *not* A. A system that cannot decide all statements in the system is *incomplete*.

An ideal axiomatic system would be complete and consistent; it would be able to generate all true statements but never generate any false statements.

Euclid's system is consistent but not complete. There are statements that concern simple properties in geometry (a famous example is *any angle can be divided into three equal sub-angles*) that cannot be derived in the system; trisecting an angle requires more powerful tools than the straightedge and compass provided by Euclid's postulates.

Figure 15.1 depicts two axiomatic systems. The left one is *incomplete* since there are some propositions that can be stated in the system that are true for which no valid proof exists in the system. The right one is *inconsistent*: it is possible to construct a proof that starts from the axioms and follows the inference rules but ends up with a proposition that is false. Once a single contradictory proposition can be proven the system becomes completely useless. The contradictory propositions amount to a proof that *true* = *false*, so once a single pair of contradictory propositions can be proven every other false proposition can also be proven in the system. Hence, we focus on systems that are consistent and consider whether it is possible for them to also be complete.



Incomplete system: at least one **true** proposition cannot be proven.

Inconsistent system: there is a valid proof of a proposition that is **false**.

Figure 15.1. Incomplete and inconsistent axiomatic systems.

15.1.1 Russell's Paradox

Towards the end of the 19th century, many mathematicians sought to systematize mathematics. The goal was to develop an axiomatic system that was complete and consistent for all of some area of mathematics. One notable attempt was Gottlob Frege's *Grundgestze der Arithmetik* (1893) which attempted to develop an axiomatic system for all of mathematics built from simple logic.

Bertrand Russell discovered a problem with Frege's system, which is now known as *Russell's paradox*. Suppose S is defined as the set containing all sets that do not contain themselves as members. For example, the set of all prime numbers does not contain itself as a member (since all its members are numbers), so it is a member of S. On the other hand, the set of all entities that are not prime numbers is a member of S. This set contains all sets, since a set is not a prime number, so it must contain itself.

The paradoxical question is: *is the set S a member of S?* Like all binary questions, there are two possible answers to consider: "yes" and "no":

- Yes: suppose S is a member of S. Then, the set S contains itself. But, we defined the set S as the set of all sets that do not contain themselves as member. Hence, S cannot be a member of itself, and the statement that S is a member of S must be false.
- No: suppose S is not a member of S. Then, the set S does not contain itself. But, we defined the set S as the set of all sets that do not contain themselves as a member. So, if S is not a member of S, it does not contain itself, and it must be a member of set S. This is a contradiction, so the statement that S is not a member of S must be false.

This is a paradox! The question is a perfectly clear and precise binary question, but neither the "yes" answer or the "no" answer makes any sense.

Symbolically, we summarize the paradox: For any set A, $A \in S$ if and only if $A \notin A$. For any item and any set, the item is either in the set or not. So, selecting S for the item and S for the set, either (1) $S \in S$ or (2) $S \notin S$. But, both of these possibilities lead to contradictions because of the property above. Substituting S for A, we have $S \in S$ if and only if $S \notin S$. Thus, if $S \in S$ the property implies $S \notin S$, a contradiction. But, if $S \notin S$ the property implies $S \in S$, also a contradiction.

Whitehead and Russell attempted to resolve this paradox by constructing their system to make it impossible to define the set S. Their solution was to introduce types. Each set has an associated type, and a set can only contain members of type below the type of the set. The set types are defined recursively:

- A type zero set is defined as a set that contains only objects (that is, it cannot contain any sets as members).
- A type n set is a set that contains only objects and sets of type $n - 1$ and below.

With this definition, the paradox is avoided: the definition of S must now define S as a set of type k set containing all sets of type $k - 1$ and below

that do not contain themselves as members. Since S is a type k set, it cannot contain itself, since it cannot contain any type k sets.

The type restriction eliminates the paradox regarding set self-inclusion. In 1913, Alfred North Whitehead and Bertrand Russell published *Principia Mathematica*, a bold attempt to mechanize mathematical reasoning that stretched to over 2000 pages. Whitehead and Russell attempted to derive all true mathematical statements about numbers and sets starting from a set of axioms and formal inference rules. They employed the type restriction to eliminate the particular paradox caused by set inclusion, but it does not eliminate all self-referential paradoxes.

For example, consider this paradox named for the Cretan philosopher Epimenides who was purported to have said "All Cretans are liars.". If the statement is true, than Epimenides, a Cretan, is not a liar and the statement that all Cretans are liars is false. Another version is the self-referential sentence: *This statement is false.* If the statement is true, then it is true that the statement is false (a contradiction). If the statement is false, then it is a true statement (also a contradiction).

It was not clear until Gödel, however, if such statements could be stated in the *Principia Mathematica* system.

15.1.2 Gödel's Incompleteness Theorem

Gödel with Einstein, Princeton 1950

Institute for Advanced Study Archives

Kurt Gödel was born in Brno (then part of Austria-Hungary, now in the Czech Republic) in 1906. Gödel proved that the axiomatic system in *Principia Mathematica* could not be complete and consistent, but more generally that *no* powerful axiomatic system could be both complete and consistent. He proved that no matter what this axiomatic system is, if it is powerful enough to express certain things, it must also be the case that there exist statements that can be expressed in the system by cannot be proven either true or false within the system.

Gödel's proof used construction: to prove that *Principia Mathematica* contains statements which cannot be proven either true or false, it is enough to find one such statement. Gödel's statement is:

> G_{PM}: Statement G_{PM} does not have any proof in the system of *Principia Mathematica*.

If statement G_{PM} is provable in the system, then the system is inconsistent: it can be used to prove a statement that is not true. If G_{PM} is proven, then it means G_{PM} does have a proof, but G_{PM} stated that G_{PM} has no proof. On the other hand, if G_{PM} is not provable in the system, then the system

is incomplete. Since G_{PM} cannot be proven in the system, G_{PM} is a true statement. But, the premise is that G_{PM} is not provable. So, we have a true statement that is not provable in the system.

The proof generalizes to *any* axiomatic system, powerful enough to express a corresponding statement G:

> G: Statement G does not have any proof in the system.

For the proof to be valid, it is necessary to show that statement G can be expressed in the system.

To express G formally, we need to consider what it means for a statement to not have any proof in the system. A proof of the statement G is a sequence of steps, T_0, T_1, T_2, ..., T_N. Each step is the set of all statements that have been proven true so far. Initially, T_0 is the set of axioms in the system. To be a proof of G, T_N must contain G. To be a valid proof, each step should be producible from the previous step by applying one of the inference rules to statements from the previous step.

To express statement G an axiomatic system needs to be powerful enough to express the notion that a valid proof does not exist. Gödel showed that such a statement could be constructed using the *Principia Mathematica* system, and using any system powerful enough to be able to express interesting properties. That is, in order for an axiomatic system to be complete and consistent, it must be so weak that it is not possible to express *this statement has no proof* in the system.

15.2 The Halting Problem

Gödel established that no interesting and consistent axiomatic system is capable of proving all true statements in the system. Now we consider the analogous question for computing: *are there problems for which no algorithm exists?*

Chapter 4 provided these definitions:

- A *problem* is a description of an input and a desired output.
- A *procedure* is a specification of a series of actions.
- An *algorithm* is a procedure that is guaranteed to always terminate.

A procedure solves a problem if that procedure produces a correct output for every possible input. If that procedure always terminates, it is an algo-

rithm. So, the question can be stated as: *is there a problem P such that there exists no procedure that produces the correct output for problem P for all inputs in a finite amount of time?*

A problem is *computable* (the term *decidable* means the same thing) if there exists an algorithm that solves the problem. It is important to remember that in order for an algorithm to be a solution for a problem P, it must always terminate (otherwise it is not an algorithm) and must always produce the correct output for *all* possible inputs to P. If no such algorithm exists, the problem is *noncomputable* (also known as *undecidable*).

computable

noncomputable

Alan Turing proved that there exist noncomputable problems. Similarly to Gödel's proof, the way to show that uncomputable problems exist is to find one the way Gödel show unprovable true statements exist by finding an unprovable true statement. The problem Turing found is known as the *Halting Problem*:[1]

Halting Problem

Input: A String representing a Scheme expression.

Output: If evaluating the input expression in the current execution environment would ever finish, output *true*. Otherwise, output *false*.

Suppose we had a procedure *halts?* that solves the Halting Problem. The input to *halts?* is a Scheme expression expressed as a String. We could not use the Scheme expression directly as the input, since that would mean that it is evaluated before *halts?* is applied; if this were done, the *halts?* procedure would never be applied if the input expression would not halt.

The hypothetical *halts?* procedure should work like this:

```
> (halts? "(+ 2 3)")
true
> (define (factorial n) (if (= n 0) 1 (* n (factorial (- n 1)))))
> (halts? "(factorial 10)")
true
> (halts? "(factorial -1)")
false
> (define (fibo n) (if (or (= n 1) (= n 2)) 1
```

[1]This problem is a variation on Turing's original problem, which assumed a procedure that takes one input. Of course, Turing did not define the problem using a Scheme expression since Scheme had not yet been invented when Turing proved the Halting Problem was noncomputable in 1936. In fact, nothing resembling a programmable digital computer would emerge until several years later.

```
(+ (fibo (− n 1)) (fibo (− n 2)))))
> (halts? "(fibo 60))")
true
```

Note that it is not possible to implement *halts?* by evaluating the expression and outputting true if it terminates. The problem is it is not clear when to give up and output *false*. As we analyzed in Chapter 7, evaluating (*fibo* 60) would take millions of years; in theory, though, it eventually finishes so *halts?* should output *true*.

This argument is not sufficient to prove that *halts?* is noncomputable. It just shows that one particular way of implementing *halts?* would not work. To show that *halts?* is noncomputable, we need to show that there is no possible way to implement a *halts?* procedure that would produce the correct output for all inputs in a finite amount of time.

Proofs of non-existence are usually much tougher than proofs of existence. One way to prove non-existence of an X, is to show that if an X exists it leads to a contradiction. We will prove that if a *halts?* algorithm exists, a contradiction results, hence no *halts?* algorithm exists. We do this by showing one input for which the *halts?* procedure could not possibly work correctly.

Consider this input procedure:

```
(define (paradox)
  (if (halts? "(paradox)")
      (loop-forever)
      true))
```

where the procedure *loop-forever* is defined as:

```
(define (loop-forever) (loop-forever))
```

The body of the *paradox* procedure is an if-expression. The predicate expression is (*halts?* "(paradox)"). This could evaluate to either true or false.

If the predicate expression evaluates to true it means evaluating (*paradox*) finished. When the predicate expression evaluates to true, the consequent expression, (*loop-forever*), is evaluated. This applies the *loop-forever* procedure, whose body expression is an application of *loop-forever*. Evaluating this never terminates since it is a circular definition with no base case. Thus, if the predicate evaluates to true, the evaluation of an application of *paradox* never halts. But, this means the result of the (*halts?* "(paradox)") predicate was incorrect. Hence, it is not sensible for (*halts?* "(paradox)") to evaluate

to a true value, since this would cause the application of *paradox* to never terminate.

The other option is that the predicate expression evaluates to false. If this is the case, the alternate expression is evaluated. It is the primitive expression *true*. Thus, the evaluation of *paradox* terminates after the if-expression is evaluated. But, this option assumed the predicate (*halts?* "(paradox)") evaluates to false. This means the evaluation of *paradox* does not terminate. Hence, it is not sensible for (*halts?* 'paradox) to evaluate to a false value, since this would cause the application of *paradox* to terminate.

Either result for (*halts?* "(paradox)") leads to a contradiction! The only sensible thing *halts?* could do for this input is to not produce a value. That means there is no way to define an algorithm that solves the Halting Problem. Any procedure we attempt to define to implement *halts?* must sometimes either produce the wrong result or fail to produce a result at all (that is, run forever without producing a result). Thus, the Halting Problem is noncomputable.

There is one important hold in our proof: we argued that because *paradox* does not make sense, something in the definition of *paradox* must not exist, and identified *halts?* as the component that does not exist. This assumes that everything else we used to define *paradox* does exist.

This seems reasonable enough—we have been using everything else in it already (an if-expression, a string literal, applications, and the primitive *true*) and they seem to exist. But, perhaps the reason *paradox* leads to a contradiction is because *true* does not really exist or because it is not possible to implement an if-expression that adheres to the Scheme evaluation rules. Although we have been using there and they seems to always work fine, we have no formal model in which to argue that evaluating *true* always terminates or that an if-expression means what we think it does.

Our informal proof is also insufficient to prove the stronger claim that no algorithm exists to solve the halting problem. All we have shown is that no Scheme procedure exists that solves *halts?*. Perhaps there is a procedure in some more powerful programming language (in fact, we will see that no more powerful programming language exists).

Overcoming these weaknesses requires a formal model of computing. This is the reason Alan Turing developed the Turing Machine model we introduced in Chapter 6.

15.3 Universality

Turing argued that this simple model corresponds to our intuition about what can be done using systematic computation. Recall this was 1936, so the model for systematic computation was not what a mechanical computer can do, but what a human computer can do. Turing argued that his model corresponded to what a human computer could do by following a systematic procedure: the infinite tape was as powerful as a two-dimensional sheet of paper, the set of symbols must be finite otherwise it would not be possible to correctly distinguish all symbols, and the number of machine states must be finite because there is a limited amount a human can remember.

Next, Turing argued that it is possible to enumerate all Turing Machines. One way to see this is to devise a notation for writing down any Turing Machine. A Turing Machine is completely described by its alphabet, states and transition rules. So, we could write down any Turing Machine by numbering each state and listing each transition rule as a tuple of the current state, alphabet symbol, next state, output symbol, and tape direction. We can map each state and alphabet symbol to a number, and use this encoding to write down a unique number for every possible Turing Machine. Hence, we can enumerate all possible Turing Machines by just enumerating the positive integers. Most positive integers do not correspond to a valid Turing Machine, but if we go through all the numbers we will eventually reach every possible Turing Machine.

TODO: Recall Chapter 1

This is already a strong step towards proving that some problems cannot be solved by any algorithm. The number of Turing Machines is less than the number of real numbers. Both numbers are infinite, so what does it mean to say there are more real numbers than integers? This question was resolved by Georg Cantor. His proof used a technique known as *diagonalization*. Suppose the real numbers are enumerable. This means we could list all the real numbers in order, so we could assign a unique integer to each number. For example, considering just the real numbers between 0 and 1, our enumeration might be:

diagonalization

1	.00000000000000 . . .
2	.25000000000000 . . .
3	.33333333333333 . . .
4	.66666666666666 . . .
. . .	. . .
57236	.141592653589793 . . .
. . .	. . .

Cantor proved by contradiction that there is no way to enumerate all the real numbers. The trick is to produce a new real number that is not part of the enumeration. We can do this by constructing a number whose first digit is different from the first digit of the first number, whose second digit is different from the second digit of the second number, etc. For the example enumeration above, we might choose .1468 . . ..

The kth digit of the constructed number is different from the kth digit of the number k in the enumeration. Since the constructed number differs in at least one digit from every enumerated number, it does not match any of the enumerated numbers exactly. Thus, there is a real number that is not included in the enumeration list, and it is impossible to enumerate all the real numbers. This suggests that there are real numbers that cannot be produced by any Turing Machine: there are fewer Turing Machines than there are real numbers, so there must be some real numbers that cannot be produced by any Turing Machine.

The next step is to define a *Universal Turing Machine* as depicted in Figure 15.2.

Universal Turing Machine

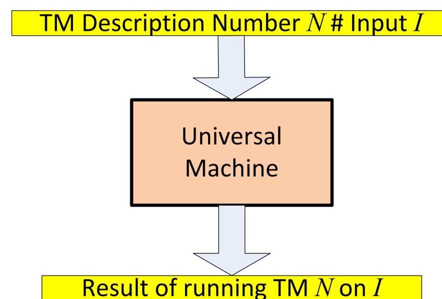


Figure 15.2. Universal Turing Machine.

A Universal Turing Machine is a machine that takes as input a number that identifies a Turing Machine and a description of an input and simulates the specified Turing Machine on the given input. The universal machine can simulate any Turing Machine on any input. In his proof, Turing describes the transition rules for a universal machine. A universal machine simulates the Turing Machine encoded by the input number. One can imagine doing this by using the tape to keep track of the state of the simulated machine. For each step, the universal machine needs to search the description of the input machine to find the appropriate rule. This is the rule for the current state of the simulated machine on the current input symbol of the simulated machine. The universal machine keeps track of the machine and tape state of the simulated machine, and simulates each step. Thus, there is a single Turing Machine that can simulate every Turing Machine.

Using the universal machine and a diagonalization argument similar to the one above for the real numbers, Turing was able to reach a similar contradiction for a problem analogous to the Halting Problem described above for Scheme expressions but for Turing Machines instead.

Since a Universal Turing Machine can simulate every Turing Machine, and a Turing Machine can perform any computation according to our intuitive notion of computation, this means a Universal Turing Machine can perform all computations. If we can simulate a Universal Turing Machine in a programming language, that language is a *universal programming language*. There is some program that can be written in that language to perform every possible computation. It is sufficient to show that it can simulate a Universal Turing Machine, since a Universal Turing Machine can perform every possible computation. To simulate a Universal Turing Machine, we need some way to keep track of the state of the tape (for example, a List in Scheme would be adequate), a way to keep track of the internal machine state (a Number in Scheme can do this), and a way to execute the transition rules (we could define a Scheme procedure that does this, using an if-expression to make the decision about which transition rule to follow). Thus, Scheme is a universal programming language: one can write a Scheme program to simulate a Universal Turing Machine, and thus, perform any mechanical computation.

universal programming language

15.4 Proving Non-Computability

We can show that a problem is computable by describing a procedure and proving that the procedure always terminates and always produces the correct answer. It is enough to provide a convincing argument that such a procedure exists; finding the actual procedure is not necessary (but often helps to make the argument more convincing). To show that a problem is not computable, we need to show that *no* algorithm exists that solves the problem. Since there are an infinite number of possible procedures, we cannot just list all possible procedures and show why each one does not solve the problem. Instead, we need to construct an argument showing that if there were such an algorithm it would lead to a contradiction.

The core of our argument is based on knowing the Halting Problem is noncomputable. If a solution to some new problem P could be used to solve the Halting Problem, then we know that P is also noncomputable. That is, for a given problem P, no algorithm exists that can solve P since if such an algorithm exists it could be used to also solve the Halting Problem which we already know is impossible.

The proof technique where we show that a solution for some problem P can be used to solve a different problem Q is known as a *reduction*. *reduction*

A problem Q is *reducible* to a problem P if a solution to P could be used to solve Q. This means that problem Q is no harder than problem P, since a solution to problem Q leads directly to a solution to problem P. *reducible*

Example 15.1: Prints-Three Problem. Consider the problem of determining if an application of a procedure would ever print 3:

Prints-Three

Input: A specification of a Scheme expression.

Output: If evaluating the specified expression would ever print 3, output *true*. Otherwise, output *false*.

We show the Prints-Three Problem is noncomputable by showing that it is as hard as the Halting Problem, which we already know is noncomputable.

Suppose we had a procedure *prints-three?* that solves the Prints-Three Problem. Then, we could define *halts?* as:

```
(define (halts? expr)
  (prints-three?
   (string-append "(begin " expr " (print 3))")))
```

We use *string-append* to paste the input string representing a Scheme expression into a begin-expression where it is followed by (*print* 3).

The *prints-three?* application will evaluate to *true* if evaluating *expr* would halt, since that means the second expression in the begin expression would be evaluated and that expression prints 3. On the other hand, if evaluating *expr* would not halt, the second expression in the begin expression is never evaluated. As long as the procedure never prints 3, the application of *prints-three?* should evaluate to *false*. Hence, the output would correctly solve the Halting Problem.

The one wrinkle is that the input procedure might print 3 itself. We can avoid this problem by transforming the procedure in a way that it would never print 3 itself, without otherwise altering its halting or non-halting behavior. One way to do this would be to take advantage of the evaluation rule for names to hide the built-in *print* procedure. We can do this by using a let-expression to define a new *print* place containing a procedure that does nothing. Since this name will be found in a new environment, the definition of *print* in the global environment will not be used.

```
(define (halts? expr)
  (prints-three?
   (string-append "(begin (let ((print (lambda (x) (void)))) "
                  expr
                  ") (print 3))")))
```

Note that the let-expression that hides the built-in *print* procedure when *expr* would be evaluated is closed by the right parentheses before (*print* 3), so if the second expression in the begin-expression is evaluated, the built-in print is used and 3 is indeed printed.

Thus, if there exists a *prints-three?* procedure that correctly solves the Prints-Three Problem — that is, it always terminates and always produces the correct true or false value indicating if the input procedure specification would ever print 3 — we could also define a *halts?* procedure that solves the Halting Problem. But, we know that the Halting Problem is noncomputable. Hence, the *prints-three?* procedure we used to define *halts?* cannot exist, and the Prints-Three Problem must also be noncomputable.

The Halting Problem and Prints-Three Problem are noncomputable, but do seem to be obviously important problems. It is useful to know if a procedure application will terminate in a reasonable amount of time, but the Halting Problem does not answer that question. It concerns the question of whether the procedure application will terminate in any finite amount of time, no matter how long it is. Next, we consider a problem that it would be very useful to have a solution for it one existed.

Example 15.2: Is-Virus Problem. A virus is a program that infects other programs. A virus spreads by copying its own code into the code of other programs, so when those programs are executed the virus will execute. In this manner, the virus spreads to infect more and more programs. A typical virus also includes a malicious payload so when it executes in addition to infecting other programs it also performs some damaging (corrupting data files) or annoying (popping up messages) behavior. The Is-Virus Problem is to determine if a procedure specification contains a virus:

Is-Virus

Input: A specification of an expression.

Output: If the expression contains a virus (a code fragment that will infect other files) output *true*. Otherwise, output *false*.

We demonstrate the Is-Virus Problem is noncomputable using a similar strategy to the one we used for the Prints-Three Problem: we show how to define a *halts?* algorithm given a hypothetical *is-virus?* algorithm. Since we know *halts?* is noncomputable, this shows there is no *is-virus?* algorithm.

Assume *infect-files* is a procedure that infects files, so the result of evaluating (*is-virus?* "(infect-files)") is true. Then, we can define *halts?* as:

```
(define (halts? expr)
  (is-virus?
   (string-append
    "(begin " expr " (infect-files))")))
```

This works as long as the input *expr* does not exhibit the file-infecting behavior. If it does, *expr* could infect a file and never terminate, and *halts?* would produce the wrong output.

To solve this we need to do something like we did in the previous example to hide the printing behavior of the original program. In this case it is a bit trickier, since we do not have a clear notion of what it means to infect a file. A rough definition of file-infecting behavior would be to consider any write to an executable file to be an infection. Then, we would use a let-expression to hide the definition of *write*, the library function that writes to a file, and other procedures that write to files. We could replace these procedures with procedures that check if the object being written to is an executable file, and ignore the write in these cases.

```
(define (halts? expr)
  (is-virus?
   (string-append
    "(begin (let ((write (lambda (val f) (if (file-stream-port? f) (void) ...)))) "
    expr
    " (infect-files)))")))
```

The actual let-expression needed to hide all file infections without otherwise changing the behavior of *expr* would be more complex, but the important point is that such an expression could be written, and then we could use *is-virus?* to define *halts?*. Since we know there is no algorithm that solves the Halting Problem, this proves that there is no algorithm that solves the *Is-Virus* problem.

Virus scanners such as Symantec's Norton AntiVirus attempt to solve the *Is-Virus* Problem, but its non-computability means they are doomed to always fail. Virus scanners detect known viruses by scanning files for strings that match signatures in a database of known viruses. As long as the signature database is frequently updated they may be able to detect currently spreading viruses, but this approach cannot detect a new virus that will not match the signature of a previously known virus.

Sophisticated virus scanners employ more advanced techniques than signature scanning to attempt to detect complex viruses such as metamorphic

viruses that alter their own code as they propagate to avoid detection. But, because the general *Is-Virus* Problem is not computable, we know that it is impossible to create a program that always terminates and that always correctly determines if an input procedure specification is a virus.

Exercise 15.1. Is the Launches-Missiles Problem described below computable? Provide a convincing argument supporting your answer.

Launches-Missiles

Input: A specification of a procedure.

Output: If an application of the procedure would lead to the missiles being launched, outputs *true*. Otherwise, outputs *false*.

You may assume that the only thing that causes the missiles to be launched is an application of the *launch-missiles* procedure.

Exercise 15.2. Is the Same-Result Problem described below computable? Provide a convincing argument supporting your answer.

Same-Result

Input: Specifications of two procedures, P and Q.

Output: If an application of P terminates and produces the same value as applying Q, outputs true. If an application of P does not terminate, and an application of Q also does not terminate, outputs true. Otherwise, outputs false.

Exercise 15.3. Is the Check-Proof Problem described below computable? Provide a convincing argument supporting your answer.

Check-Proof

Input: A specification of an axiomatic system, a statement (the theorem), and a proof (a sequence of steps, each identifying the axiom that is applied).

Output: Outputs true if the proof is a valid proof of the theorem in the system, or false if it is not a valid proof.

Exercise 15.4. Is the Find-Finite-Proof Problem described below computable? Provide a convincing argument supporting your answer.

Find-Finite-Proof

Input: A specification of an axiomatic system, a statement (the theorem), and a maximum number of steps (max-steps).

Output: If there is a proof in the axiomatic system of the theorem that uses max-steps or fewer steps, outputs true. Otherwise, outputs false.

I am rather puzzled why you draw this distinction between proof finders and proof checkers. It seems to me rather unimportant as one can always get a proof finder from a proof checker, and the converse is almost true: the converse false if for instance one allows the proof finder to go through a proof in the ordinary way, and then, rejecting the steps, to write down the final formula as a 'proof' of itself. One can easily think up suitable restrictions on the idea of proof which will make this converse true and which agree well with our ideas of what a proof should be like. I am afraid this may be more confusing to you than enlightening.
Alan Turing, letter to Max Newman, 1940

Exercise 15.5. [⋆] Is the Find-Proof Problem described below computable? Provide a convincing argument why it is or why it is not computable.

Find-Proof

Input: A specification of an axiomatic system, and a statement (the theorem).

Output: If there is a proof in the axiomatic system of the theorem, outputs true. Otherwise, outputs false.

15.5 Summary

Although today's computers can do amazing things, many of which could not even be imagined twenty years ago, there are some problems that cannot be solved by computing. The Halting Problem is the most famous example: it is impossible to define a mechanical procedure that always terminates and correctly determines if the computation specified by its input would terminate. Once we know the Halting Problem is noncomputable, we can show that other problems are also noncomputable by illustrating how a solution to the other problem could be used to solve the Halting Problem, which we know to be impossible.

Noncomputable problems frequently arise in practice. For example, identifying viruses, analyzing program paths, and constructing proofs, are all noncomputable problems. Just because a problem is noncomputable does not mean we cannot produce useful programs that address the problem. These programs provide approximate solutions — they produce the correct results on many inputs, but on some inputs either do not produce any result, or produce an incorrect result. Approximate solutions are often useful

in practice, however, even if they are not correct solutions to the problem.