3 Computability and Incomputability

Computer Science is no more about computers than astronomy is about telescopes.
— E. W. Dijkstra

To use: Apply shampoo to wet hair. Massage to lather, then rinse. Repeat.
— A typical hair-washing algorithm that fails to halt

But let your communication be Yea, yea; Nay, nay: for whatsoever is more than these cometh of evil.
— An early proposal for binary code, Matthew 5:37

FOR MOST PEOPLE, the notion of what is computable is closely related to what types of programs exist. A typical computer has an operating system that acts as an interface between the computer hardware and the software. Window systems that provide graphical user interfaces to other programs are at an even higher level of abstraction. A typical program that could operate in this environment is a word-processing program that allows the user to type in keystrokes and to perform mouse actions such as pointing, clicking, dragging, and using menus. Many things appear to be happening at once, but in fact each action by the user is processed by the computer program during a discrete time interval. When you type a key to insert text into your document, the program must update the graphical representation of your document by drawing the raw pixels that make up the letters. The program will usually have to update some internal representation of the text of your document as well. Similarly, when you use a mouse, the program translates the mouse clicks to a coordinate system and performs whatever action is required to make the graphical representation of your document look the way it is supposed to. Each action is carefully coordinated by the computer program, which can be a daunting task, considering the thousands of details that go into writing a program such as a word processor.

The example above illustrates an interactive session that a human could have with a program; however, there is no reason why the session would have to be interactive at all. As a thought experiment, imagine that we could record each keystroke and mouse action. The recording could be saved into a file that has strings[1] of the form `keystroke 'k'`, `mouse-down (113, 156)`, or `mouse-up (115, 234)`. It is always possible to take a program like a word processor and convert it into a similar program that processes a file that has the recorded actions. If one of those actions corresponded to pulling down a menu and selecting a "save file" option, then one form of the output of the program can be saved as well. Moreover, each graphical action taken by the word processor could be saved into another file with entries indicating which pixels were drawn and in what color. In this way it is possible to convert any program into a form that takes one long string as input and produces one long string as output.

As a historical footnote, in the early days of computers, both programs and input data had to be submitted to a computer via an antiquated device known as a punch-card reader. The output of the program could then be sent to a printer or some other device. Graphical user interfaces are a very convenient way of using a computer, but in no way does the user interface change what is fundamentally possible for a computer to compute. If a mad computer programmer had the desire to, there is no reason why he couldn't have simulated an entire interactive computer session back in the 1950s with punch cards and printer output.

Therefore, without losing any of the notion of what it means to compute, we can completely disregard the "bells and whistles" of modern computers and only concentrate on the gist of what happens inside of a computer. What is left is a picture not unlike the earliest computer mainframes that handled only punch cards as input. Why bother to do this? If we were to attempt to discuss the notion of computability in the context of user interfaces, sound cards, mice, and laser printers, there would be no clear way of reducing all of the detail into something that looks remotely mathematical. If we could not reduce what happens inside of computers into some mathematical formalism, there would be no way of proving or disproving the properties that computers have.

Scientists like to boil things down to the simplest terms possible, and it turns out that this picture of having a string of input and a string of output is still overly verbose for a theoretical computer scientist's tastes. Just as we were able to dispense with most of the details of how modern computers operate, so it is possible to dispense with the idea that programs have "strings" of input and output. It is possible (and also theoretically useful) to convert a program, input string, or output string into a single natural number.

[1]The term *string* will be used to denote any sequence of letters, numbers, or digits, or any other type of list.

3.1 Gödelization

Kurt Gödel, one of the greatest mathematicians of the twentieth century, literally shocked his contemporaries with some of his mathematical results. We will talk more about Gödel and his contributions to mathematics toward the end of this book part; however, in this section we will concentrate on the process for converting many numbers into one number that bears his name.

A *Gödelization* is a method for mapping many natural numbers into a single natural number. The details of how the mapping is performed are not very interesting, but the fact that it can be done is extremely important. Recall that the input or output to a program can be represented as some finite-length string. Also note that a program can be represented in the same way. Don't be bothered by the fact that one form of the input string may look like `keystroke 'k'` or some other nonnumerical form. We can always adopt some convention whereby we agree to represent actions such as `keystroke 'k'` by two natural numbers, one for the action, `keystroke`, and the other for the key that was pressed, `'k'`. It is always possible to perform some mapping such that any string is unambiguously coded into a sequence of integers. The interesting question is: How can many numbers be encoded into one?

The key to the whole process is the fact that every number has a unique prime factorization. If you pick any natural number, x, then there is exactly one sequence of prime numbers, p_{x_1}, p_{x_2}, $\ldots$, p_{x_n}, such that the product of the n prime numbers is equal to x. Now, let's go back to looking at a program's input, which we earlier agreed to think of as a sequence of numbers. If there are n numbers in the sequence, then let every number in the sequence be denoted by x_1, x_2, $\ldots$, x_n. To calculate the Gödel number of the input string, we use the first n prime numbers and calculate

$$\prod_{i=1}^{n} p_i^{x_i} = p_1^{x_1} p_2^{x_2} \cdots p_n^{x_n},$$

which forms a unique natural number. Granted, Gödel numbers will tend to be huge in size, but who cares? Given a Gödel number, we can reconstruct the original string by taking the prime factorization of the Gödel number. If there are thirteen 2s in the prime factorization of the Gödel number, then that means that the first number in the original string was 13. If there are eighty-seven 3s in the prime factorization, then the second number in the original string was 87. And if there is a single 5 in the prime factorization, then the third number was 1.

Gödelization adds another simplification to studying the nature of computation. Instead of worrying about programs with multiple input and output sequences we can now ignore most of the details and just concentrate on functions that take a single number as input and produce a single number as output. Even with this restriction, a computer program that operates in this manner is still doing all of

How Many Prime Numbers Exist? **Digression 3.1**

Constructing a Gödel number from a string depends on there existing an infinite number of prime numbers. If there were only a finite number of prime numbers, say n of them, then it would be impossible to encode strings of length greater than n. Here is an extremely elegant proof discovered by Euclid around the third century B.C. that shows that there are an infinite number of primes.

If there are a finite number of prime numbers, then we could list all of them as $p_1, p_2, \ldots, p_n$, where p_n is the largest prime number. We will now construct a new number from these n prime numbers by taking the product of all n prime numbers and adding one

$$p' = \prod_{i=1}^{n} p_i + 1 = (p_1 p_2 \cdots p_n) + 1.$$

What does the prime factorization of p' look like? Before you answer, take note of the fact that none of the n prime numbers, $p_1, \ldots, p_n$, evenly divides p'. Try it and you will see that you always get a remainder of 1. But if no prime number evenly divides p', then one of two things must be true: Either p' is prime or it is not and there exists some other prime number greater than p_n that does evenly divide p'. We don't care which is the case, since either implies that there is another prime number greater than p_n. Therefore, there must be an infinite number of primes.

the "hard" part of computing. Therefore, without loss of generalization, when we speak of computation, we will sometimes refer to the computation as manipulating strings, numbers, or even bits. It really doesn't matter. What does matter is that the representation of a computer's input and output can always be converted from one form to another. We will simply use whatever form is most convenient at the time.

Another conclusion that can be reached from the ideas of this section is that there are as many programs as there are natural numbers, since we can Gödelize programs as well. This fact will be expanded on toward the end of this chapter. In the next section we will get into the details of what it means to compute by studying some models of computation.

3.2 Models of Computation

There is a subtle difference between computations and models of computation that we should examine. It is fair to think of a computation as a "method" for producing one number from another. Computer textbooks often speak of algorithms and instructions. If it makes you feel more comfortable with the subject, you can think of a computation, method, algorithm, or instruction as merely a recipe like one you would find in any cookbook. We are, after all, cooking here, but with numbers instead of food.

What is a model of computation? Generally speaking, a model of computation describes how to build recipes or, if you like, a recipe for recipes. Actually, computations describe how to map numbers to other numbers, and models of computation describe how to construct the mappings.

In the past century, many mathematicians have grappled with the problem of how to describe all of the infinitely many computations that are possible. The problem, put more concisely, is: What is the minimal set of rules that we can use to construct computations such that every possible computation can be realized by the rules? Many models of computation have been proposed in the past century. What follows is a brief description of some of the better-known ones. What they have in common is that each model of computation operates on numbers, strings, and symbols by manipulating them at discrete time steps. Moreover, each model has a well-defined "program," "input," and "output." If the presentation of the models is too formal for your tastes, feel free to skim the descriptions and jump ahead to the end of this section.

General Recursive Functions General recursive functions are constructed by composing a small number of rules together. The idea is that one can take a few simple functions and construct more complex functions by applying the following rules repeatedly. The rules will either specify a base general recursive function, in which case a name and an example are given, or a rule for composing a new general recursive function.

- **Zero:** The zero function returns zero for any argument, e.g., $Z(x) = 0$.
- **Successor:** The successor function adds one to its argument, e.g., $S(x) = x + 1$.
- **Projection:** The projection rule simply states that a general recursive function is allowed to return any one of its arguments as the result, e.g., $P_i(x_1, \cdots, x_n) = x_i$.
- **Composition:** The composition rule allows for a new function to be constructed as the composition of two or more functions. Thus, if $g(x)$ and $f(x)$ are general recursive functions, then so is $g(f(x))$.
- **Recursion:** General recursive functions can have recursive definitions. For example, if $g(x)$ and $h(x)$ are general recursive, then so is $f(x, y)$ defined as $f(x, 0) = g(x)$, for $y = 0$, and $f(x, y + 1) = f(h(x), y)$, for all other y.

- **Minimization:** A general recursive function can be expressed as the minimization of another general recursive function. For example, if $g(x, y)$ is general recursive, then so is the function $f(x) = \mu y[g(x, y) = 100]$, where μ is the minimization operator. We can interpret $f(x)$ as being "the smallest value of y such that $g(x, y) = 100$." Note that there may be no y that satisfies the constraint for the supplied x, in which case f is undefined for that x.

Notice that the definition of general recursive functions is closely coupled to natural numbers, in that the functions clearly operate on the natural numbers. If we wanted to construct a general recursive function to add two numbers, the definition would look like:

$$\begin{aligned} f(x, 0) &= P_1(x) = x \\ f(x, y+1) &= S(P_3(x, y, f(x, y))) = f(x, y) + 1. \end{aligned}$$

Here we can think of a "program" as being a general recursive function that is constructed, of the program's "input" as the natural number that we plug into the function, and of the program "output" as the natural number that we get as a result.

Turing Machines A Turing machine is a hypothetical device proposed by Alan Turing in 1936. The machine has a read/write head mounted to a tape of infinite length. The tape consists of an infinite number of discrete cells in which the Turing machine can read or write symbols. At every discrete point in time a Turing machine exists in one and only one state. The "program" for a Turing machine consists of a state transition table with entries that contain: the current state, the symbol underneath the head, the next state that the machine should enter, the new symbol that should be written, and the direction that the tape should move (left, right, or none). There is also a unique state known as the starting state and one or more halting states. When a Turing machine starts up, the "input" to the program consists of the symbols already written on the tape. At each time step, the Turing machine performs an action, determined by the current state and the symbol

State	**Input**	**Action/Output**	**Next State**
Start	$\star$	get in cruise lane	cruise
Cruise	cruise lane clear	drive at cruise speed	cruise
Cruise	slow driver ahead	get in pass lane	pass
Pass	cruise lane clear	get in cruise lane	cruise
Pass	slow driver in cruise lane	accelerate	pass
$\star$	desired exit ahead	exit highway	halt

Table 3.1 A driving "program" for a Turing machine

Alan Turing: A Mini-Biography **Digression 3.2**

Alan Turing was born in 1912. As one of the pioneers in the theory of computation, his importance to the field is staggering. He participated in many public debates concerning the future of computers and artificial intelligence, and as part of this activity he invented what is now popularly known as the *Turing test*, a method to determine if a computer has "intelligence." During World War II, Turing was part of a British research team that cracked the Germans' most secret encryption device. The success of the project allowed Winston Churchill to listen in on many of the Axis powers' most classified command decisions.

Alan Turing was also a homosexual. While filing a complaint to the police department about a burglary of his house, he implied that he had more than a casual relationship with a possible suspect. The police interrogated Turing regarding this, and he confessed to being a homosexual. Subsequently, he was forced to undergo hormone therapy that led to depression. In 1954, Turing was simultaneously experimenting with chemicals and making candied apples. In what may or may have not been suicide, he ate a poisoned apple and died (Hodges, 1983).

underneath the head. This action may involve writing a new symbol, moving the head, and/or moving into a new internal state. The Turing machine continues this process until one of the halting states is reached. The "output" of the program consists of the remaining symbols on the tape.

At first glance, Turing machines seem very alien, but in actuality most people are familiar with how they work on an intuitive level. Table 3.1 illustrates a sort of Turing machine algorithm (without reference to the tape) for driving on a highway. To simplify things, we will consider only four types of states while driving: entering the highway (the starting state), cruising at a steady speed, passing a slow driver, and exiting the highway. For passing and cruising, we consider two special cases: when we should maintain the current state and when we should switch to another state.

In the table the $\star$ character is a wild card, meaning that it matches any possibility for that entry. Thus, according to the table, whenever you see the desired exit, you should exit the highway and go into the halting state, regardless of what state you are currently in.

Lambda Calculus The λ-calculus is a model of computation proposed by Alonzo Church in 1941. In it, computations are defined in terms of λ-expressions that consist of either a symbol or a list of λ-expressions, or have the form:

$$(\lambda \textit{bound-variable}(\lambda\text{-expression})).\lambda\text{-expression}.$$

The last form describes a way of rewriting the leftmost λ-expression. The result is to take the leftmost λ-expression and replace every occurrence of the bound variable with the lambda expression on the right-hand side. For example $(\lambda x(fx)).a = (fa)$, since we replace every occurrence of x in (fx) by a.

In λ-calculus the "program" corresponds to the left-hand λ-expression, and the "input" is the right-hand λ-expression. The "output" corresponds to the λ-expression resulting from continuously expanding the input applied to the program. You may be inclined to think that expanding a λ-expression is a one-step process but, in fact, λ-expressions can consist of λ-expressions nested within more λ-expressions, which means that expanding a single λ-expression may take many steps. The λ-calculus is very similar to the programming language Lisp, which is discussed in further detail shortly.

General recursive functions, Turing machines, and λ-calculus are not the only formal models of computation, just the best known. There are actually several more. You probably noticed that each model was very different from the others. An interesting question at this time is: Are there any functions that can be computed by one model of computation but not by another? In 1941, Church proved that λ-calculus was capable of representing exactly the same functions as general recursive functions. Later Turing proved that Turing machines could compute exactly the same functions as λ-calculus, which proved that all three models of computation are equivalent. This is a truly remarkable result, considering how different the three models of computation are. In Church's 1941 paper he made a statement that is now known as the Church-Turing thesis: Any function that can be called *computable* can be computed by λ-calculus, a Turing machine, or a general recursive function.

Recall the point that was made about functions describing relationships between numbers and models of computation describing functions. Well, the Church-Turing thesis is yet another level more fundamental than a model of computation. As a statement about models of computation, it is not subject to proof in the usual sense; thus, it is impossible to prove that the thesis is correct. One could disprove it by coming up with a model of computation over discrete elements that could calculate things that one of the other models could not; however, this has not happened. The fact that every posed model of computation has always been exactly equivalent to (or weaker than) one of the others lends strong support to the Church-Turing thesis.

3.3 Lisp and Stutter

So far we have really discussed computation only in very broad terms. It would be nice to demonstrate exactly how one could compute some useful functions with one of the three mentioned models. As we shall see, there is a certain beauty in deriving higher mathematical functions from such primitive beginnings. To illustrate this point, we will now examine a simple computer language that is as powerful as the other models but is a bit more understandable. In the late 1950s John McCarthy created a computer language known as Lisp (which stands for list processing). Lisp was inspired by λ-calculus, but I like to think of it as a close cousin of general recursive functions as well. A testament to the elegance of Lisp is that it is one of the few "old" languages still in common use today.

Modern Lisp (Common Lisp) is a very rich language, with hundreds of defined functions, macros, and operators. What follows is a description of a subset of Lisp that I will refer to as Stutter.[2] Stutter is an interpreted language which means that all expressions are evaluated during the runtime of a program, unlike compilers, which translate modern computer languages into the native machine language of the computer or an intermediate language (as in the case of Java and its bytecode compilation). The heart of the Stutter interpreter is the read-eval-print loop, which does exactly what its name describes. More specifically, when using Stutter, the user is prompted by the '`>`' prompt. After the user types in a Stutter expression and presses the "Enter" key, the computer evaluates the expression and prints the result, which brings us back to the "read" portion of the loop.

But what is a Stutter expression? In Stutter everything is either a *list* or an *atom*. An atom is simply a sequence of characters, such as `bob`, `xyz`, or `256`. There is nothing significant about any of these atoms, including `256` since the Stutter interpreter has no hardwired notion of what the value of 256 means to you and me. There are also four reserved punctuation characters that have special meaning: '`(`', '`)`', '`'`', and '`;`'. Other than these four characters an atom can consist of any sequence of nonblank characters in any order. The parentheses are used to contain lists, and the quotation character is used to quote atoms and lists. The reason why the quotation character is necessary will be explained in the examples below. The semicolon is used as a comment delimiter, that is, any text in a single line of a Stutter program following a semicolon is ignored by the Stutter interpreter and serves only to add comments within a Stutter program.

A list can have any number of members of any type, including other lists. Function definitions in Stutter are also lists, which lends to the beauty of the language, since functions can operate on other functions. As a special case, the empty list is denoted as either `()` or `nil`, and can be considered both an atom and a list. A function call is yet another list of the form `(f a b c)`, which means that the named function, `f`, will be called with the supplied arguments, `a`, `b`, and `c`. We must also make a distinction between the name of an atom and the value of an atom. Because it is useful to store things in variables, we are allowed to treat each atom as a variable. In this case, we say that the atom and variable are "bound" to each other. Later we will see how one can extract the value of an atom, but for now note the distinction that an atom unevaluated is itself, while an atom evaluated results in its value.

[2]Motivated readers can consult the C source code to Stutter. All of the examples from this chapter were produced from the supplied Stutter interpreter.

In the examples that follow, we will see the input to the Stutter interpreter as the text immediately following the '>' prompt. The line immediately following the input is the Stutter interpreter's output. Sometimes the output will be omitted if it is not interesting. To start, let's examine how things look when quoted:

```
> 'testing
testing
> 'testing-1-2-3
testing-1-2-3
> '(this is one way of writing a string of text)
(this is one way of writing a string of text)
> '(here is a list (with a list and another (list)))
(here is a list (with a list and another (list)))
> this-is-an-unquoted-undefined-atom
Error: unbound atom "this-is-an-unquoted-undefined-atom"
```

In the last example, when we typed in the unquoted atom, the Stutter interpreter responded with an error message because the unquoted atom was not bound to any value. We can set the value of an atom with the `set` function. Because this is our first use of a Stutter function, a brief digression is in order. In general, all Stutter functions either are built-in primitives or are user-defined. We will see how to define a user-defined function a little later, but for now let's examine how a function call is evaluated. When the Stutter interpreter is asked to call a function such as `(f a b c)`, the Stutter interpreter will first evaluate the first element in the list, `f`, which must be bound to a function or an error will occur. If `f` is a user-defined function, then the Stutter interpreter will immediately evaluate the arguments, `a`, `b`, and `c`.

For built-in functions, things are a little more subtle. The built-in functions are either *value functions* or *special functions*. Value functions behave just like user-defined functions but are primitives in the Stutter language because there is no way of defining them as user functions. Some value functions that we will encounter later on are `car`, `cdr`, and `cons`. The `set` function is also a primitive value function. Special functions behave similarly to the other functions except that their arguments are not evaluated initially. This allows the special functions to evaluate arguments only if it is appropriate. For example, the quote, character ‘'’ is really equivalent to the special function `quote`. Instead of using the single quote we could also type `(quote a)`, which evaluates to `a`—not the value of `a`. Later we will use another special function, `if`.

To use the `set` function, we supply it with two arguments. The first argument should evaluate to the atom that we want to bind a value to. The second argument is also evaluated, and the result is bound to the atom the first argument evaluated to. Most uses of `set` will look like `(set 'a b)`, with the first atom being quoted. Requiring the quote may seem like an unnecessary inconvenience, but it actually gives us some latitude in how we define things. Consider the examples:

```
> (set 'name 'call-me-Ishmael)
> name
call-me-Ishmael
> (set 'ten '10)
> ten
10
> (set ten '5-plus-5)  ;;; Change the value of '10 indirectly.
> ten
10
> 10
5-plus-5
```

Let's now look at how **car**, **cdr**, and **cons** work. All three functions are used to manipulate lists in some way. Computer science has had some strange effects on spoken language, and one such oddity is that Lisp programmers often say things like "the car of the list" or "the cdr of something." What they really mean in the first example is "the result of calling the car function with the given list as an argument." That's too much of a mouthful for my tastes, so you should not be confused when I use the programmer's verbal shortcut.

The **car** function always returns the first element in a list. If you try to take the **car** of something that is not a list, an error will occur. As a special case, the **car** of `nil` is also `nil`. Complementary to **car**, the **cdr** function returns the supplied list with everything but the first element. If you like, you can think of **cdr** as being synonymous with "everything but the car." Thus, the **cdr** of a list with a single element is `nil`, and as another special case the **cdr** of `nil` is also `nil`

The **cons** function is used to construct a new list from two arguments. The first argument will be the **car** of the result list and the second argument will be the **cdr** of the result. Thus, for any list `l` except `nil`, `(cons (car l) (cdr l))` is always equal to `l`. Here are some example uses of **car**, **cdr**, and **cons**.

```
> (car '((a b c) x y))
(a b c)
> (cdr '((a b c) x y))
(x y)
> (car (car '((a b c) x y)))
a
> (cdr (cdr '((a b c) x y)))
(y)
> (car (cdr (cdr (car '((a b c) x y)))))
c
> (cons 'a nil)
(a)
> (cons 'a '(b))
```

```
(a b)
> (cons '(a b c) '(x y))
((a b c) x y)
> (cons '(a b c) nil)
((a b c))
```

During the lifetime of a program, it is often necessary to ask questions. Stutter is no exception to this, so to facilitate this need, Stutter has a built-in special function known as `if`, which takes three arguments. The first argument is a condition. The `if` function evaluates the condition, and if it is true, then `if` will return the second argument evaluated but not evaluate the third argument at all. If the condition is false, then `if` will not evaluate the second argument but return the third argument evaluated instead. Any missing arguments are presumed to be `nil`. But what is "true" and "false" in Stutter? We will take "false" to be synonymous with `nil` and "true" to mean anything that is not `nil`. Because it useful to have a consistent name for the concept of "true," we will use `t` to mean just that. Moreover, `t` evaluates to itself because it is defined by `(set 't 't)`. Yet, in Stutter there is nothing special about the symbol `t`; it is just an atom like any other atom. By themselves, `if` expressions are not very interesting, but here are two that illustrate how they work:

```
> (if nil '(it was true) '(it was false))
(it was false)
> (if 'blah-blah-blah '(it was true) '(it was false))
(it was true)
```

In general, `if` statements usually take the form: `(if (condition-expression) (then-expression) (else-expression))`. The real power of an `if` statement is when the "then" or "else" portions of the statement contain even more expressions.

The last type of Stutter expression that will be highlighted is a special type of expression known as a *lambda expression*. Lambda expressions in Stutter are similar to λ-expressions in λ-calculus, in that they allow the user to define new functions. The symbol `lambda` is not a function per se, but a special atom. In general, a lambda expression will look like `(lambda (arg1 arg2) (function body ...))`. You should read the last expression as "This is a function with two arguments. When the function is called, the function body is evaluated, with the supplied arguments replacing the arguments that appear in the body."

That's all there is to Stutter. Nothing else. Your first reaction to Stutter may be that it is a rather weak programming language. For example, how would one go about adding numbers? The concept of numbers doesn't even appear in the language definition, let alone addition. Yet Stutter is as powerful as any other programming language. It is universal in that it can do anything that the other three models of computation can do (as well as your home computer[3]). How so? To

[3]Technically speaking, your home PC is weaker than any of the other models of computation because it has only finite memory.

illustrate this, we will reinvent the basic mathematical operations in Stutter. Doing so will also illustrate how to use lambda expressions.

To begin with, we need a representation for the numbers. For a start, let's define zero:

```
> (set '0 nil)
```

That's fine, but there is an infinite number of other numbers to deal with. Instead of giving a unique definition for each number, we will define what it means to be a number. More specifically, for every number there is always another number that is one greater than the first. In this spirit, let's define an increment function:

```
> (set '1+ (lambda (x) (cons t x)))
```

In English the `1+` function definition reads as "take the argument (which is presumably a list) and append the symbol `t` to the front of it." Now, if we wanted to, we could define other numbers:

```
> (set '1 (1+ 0))
> (set '2 (1+ 1))
> (set '3 (1+ 2))
  ...
> (set '10 (1+ 9))
```

But these definitions are not strictly necessary, since Stutter now understands that `'(t t t)` means the same thing as `(1+ (1+ (1+ 0)))`, which means the same thing as 3 does to us. This may seem like a cumbersome way of representing numbers, but not for Stutter.

It would also be useful to have a notion of "one less" than some number. The only difficulty is that all natural numbers are positive. Thus, the following decrement function will do just fine for positive numbers:

```
> (set '1- (lambda (x) (cdr x)))
```

Now that we have numbers, how do we do useful things with them? Let's start with addition:

```
> (set '+ (lambda (x y) (if y (1+ (+ x (1- y))) x)))
```

This definition is clearly recursive since it refers to itself. The definition in English reads "The sum of two numbers is defined as the first number if the second number is 0. If the second number is not 0 then the result is equal to 1 plus the sum of the first number and 1 less than the second number." In other words, if you ask Stutter to compute $(5 + 2)$, it will roughly carry out the expansion: $(5+2) = (1+(5+1)) = (1+(1+(5+0))) = 7$. Let's try it out:

```
> (+ 5 2)
(t t t t t t t)
> (+ 9 3)
(t t t t t t t t t t t t)
```

Multiplication and exponentiation are just as easy to define as addition since they have their own elegant recursive definitions:

```
> (set '* (lambda (x y) (if y (+ (* x (1- y)) x) 0)))
> (set '^ (lambda (x y) (if y (* x (^ x (1- y))) 1)))
```

With these definitions, we can now do some fancy calculating:

```
> (* 3 5)
(t t t t t t t t t t t t t t t)
> (^ 2 4) ;;;  2 raised to the 4th power.
(t t t t t t t t t t t t t t t t)
> (^ (+ 1 2) (* 2 2)) ;;; 3 raised to the 4th power.
(t t t t t t t t t t t t t t t t t t t t t t t t t t t
 t t t t t t t t t t t t t t t t t t t t t t t t t t t
 t t t t t t t t t t t t t t t t t t t t t t t t t t t)
```

Included with the Stutter source code are Stutter statements that define many more useful operations and predicates, such as subtraction, division, logarithm, an equality test, and greater than and less than. Moreover, if you are still troubled by the fact that numbers are represented by very long lists, there is a simple function that converts lists of the form `'(1 2 3)` into unary lists (which is the form that we have been using) and back. It is also possible to define a representation for floating-point numbers and to define more complex operations, such as the square root.

3.4 Equivalence and Time Complexity

Since all of the mentioned models of computation are equivalent, in that each of them can compute exactly what all of the others can compute, what can we say about the relative efficiency of each model? Is one type of model more efficient than another, in the sense that it can do exactly what another model can do but faster? There are some differences in speed between the different models, but not a significant amount. Why this is so is the topic of this section.

Computer science theory has a branch, known as *time complexity theory*, that deals with the question of how fast something can be computed. In each model of computation, the "computer" has to take some step-by-step actions. For example, in one time step the Turing machine reads the symbol underneath the head, writes

```
(+ 2 3)                                ;;; Expand   (+ 2 3)
  (1+ (+ 2 (1- 3)))                    ;;; Evaluate (1- 3)  -> 2
  (1+ (+ 2 2))                         ;;; Expand   (+ 2 2)
    (1+ (1+ (+ 2 (1- 2))))             ;;; Evaluate (1- 2)  -> 1
    (1+ (1+ (+ 2 1)))                  ;;; Expand   (+ 2 1)
      (1+ (1+ (1+ (+ 2 (1- 1)))))      ;;; Evaluate (1- 1)  -> 0
      (1+ (1+ (1+ (+ 2 0))))           ;;; Evaluate (+ 2 0) -> 2
      (1+ (1+ (1+ 2)))                 ;;; Evaluate (1+ 2)  -> 3
    (1+ (1+ 3))                        ;;; Evaluate (1+ 3)  -> 4
  (1+ 4)                               ;;; Evaluate (1+ 4)  -> 5
5
```

Table 3.2 Stutter execution path of `(+ 2 3)`

the appropriate new symbol, moves the tape left or right, and then makes a virtual jump to the next state. For a general recursive function, λ-calculus expression, or Stutter program there is also an iterative process taking place. You can imagine that each primitive or built-in function takes exactly one time step to execute, and that user-defined functions take as many time steps as the number of primitive functions they ultimately call upon. Modern digital computers also have a fetch-execute cycle that involves retrieving a machine language instruction from memory, decoding it, then executing it.

How do we measure the speed of a computer program? First of all, some programs will always take longer than others, given the same input. As an example, let's simulate how Stutter would add the numbers 2 and 3. Recall that addition was defined as `(lambda (x y) (if y (1+ (+ x (1- y))) x))`. Table 3.2 shows roughly what takes place to compute the final result by indenting each level of recursion. The listing is simplified somewhat, in that most of the details have been omitted of how, say, `(1- 3)` is evaluated, since it is not self-recursive and simply calls `cdr`. Moreover, instead of expanding each recursive function call into the full function body, I have expanded recursive function calls only into the portion of the body that is evaluated by the `if` expression. Looking closely at how the `+` function executes, there appear to be three types of steps. First of all, there are several expansions of the `+` function because it recursively refers to itself. Next, there are three `1+` calls and three `1-` calls. The fact that there are three of each of these calls is a consequence of `3` being the second argument. If `7` been the second argument, then there would have been seven calls to `1+` and `1-`. Likewise, the four expansions are a result of `3` being the second argument. Putting this all together, we find that the total number of steps for performing the operation `(+ a b)` is roughly $3b + 1$.

Obviously, the smart thing to do would be to make sure that the second argument is at least as small as the first. If we amended the + function so that this check is performed initially, then we would have the additional overhead of computing which is the smaller of the two arguments. The supplied Stutter code contains a Stutter definition for the relational operator <, which takes approximately the same number of steps as the smaller of the two arguments. Therefore, combining all of this into a "smart" + operation would yield a total number of steps roughly equal to $4x + 1$, where x is the smaller of the two arguments.

This is still not quite the answer that we are looking for, since we really don't want to estimate the time a function takes to execute on the basis of only one of its arguments. What we really need is a way of expressing the execution time as a function of the length of the input. In the case of the + function, the input length is equal to the length of the two list arguments. Therefore, let's agree to call x the sum of the length of the two arguments, a and b. As a worst case, let's also assume that a and b are really equal to one another. Why? If one of the two arguments was less than the other, then, since the + function executes in time proportional to its smallest argument, this would be faster than if a and b were really equal. With this assumption the smallest argument is equal to $\frac{x}{2}$. Therefore, the execution time of the + function, expressed in terms of the length of the input, is equal to $2x + 1$. This means that if you give the + function any two arguments, you can reasonably expect it to take about $2x + 1$ steps to compute the result.

However, we are not yet finished with simplifying the time measure. Time complexity analysis is one of the few mathematical disciplines in which one is supposed to take shortcuts. More specifically, the expression $2x + 1$ is just an affine linear function, and the 2 is simply a coefficient. Since we really want to know how well the + function scales when we give it really big numbers, the 2 doesn't tell us anything special, since one computer can easily be twice as fast as another. Moreover, if instead of $2x + 1$, the number of steps that a function takes was something more complicated, like $\frac{1}{2}x^4 - 2x^2 + 82x + 13$, under time complexity analysis we would simplify the whole expression down to x^4, which is the most significant term in the polynomial. The reason we are allowed to do this is not as superficial as I have made it seem, and is in fact mathematically sound. As an exercise you could compute $\frac{1}{2}x^4 - 2x^2 + 82x + 13$ divided by x^4 on a calculator with some very large values for x. As you increase the size of x, the ratio of the numbers will eventually approach a constant factor of $\frac{1}{2}$. We are allowed to simplify time complexity expressions in this manner because the ratio approaches a constant value. A computer scientist would express the x^4 time complexity measure with what is known as "big-Oh" notation, or $O(x^4)$, which is just a formal way of saying that a function or program takes about x^4 time steps to execute, given an input of length x.

Now back to the issue of how fast the + function is. Since we are now in agreement that the addition operation takes time proportional to x, we will denote this fact by saying that + has a time complexity of $O(x)$. Intuitively, this analysis makes sense,

```
(* 2 3)                                ;;; Expand    (* 2 3)
  (+ (* 2 (1- 3)) 2)                   ;;; Evaluate (1- 3)  -> 2
  (+ (* 2 2) 2)                        ;;; Expand    (* 2 2)
    (+ (+ (* 2 (1- 2)) 2) 2)           ;;; Evaluate (1- 2)  -> 1
    (+ (+ (* 2 1) 2) 2)                ;;; Expand    (* 2 1)
      (+ (+ (+ (* 2 (1- 1)) 2) 2) 2)   ;;; Evaluate (1- 1)  -> 0
      (+ (+ (+ (* 2 0) 2) 2) 2)        ;;; Evaluate (* 2 0) -> 0
      (+ (+ (+ 0 2) 2) 2)              ;;; Evaluate (+ 0 2) -> 2
    (+ (+ 2 2) 2)                      ;;; Evaluate (+ 2 2) -> 4
  (+ 4 2)                              ;;; Evaluate (+ 4 2) -> 6
6
```

Table 3.3 Stutter execution path of (* 2 3).

since our smart + operation is very similar to the way you would "add" two piles of stones. You would take one stone from the smaller of the two piles and place it into the larger pile, repeating the process until the smaller pile was gone. If you doubled the size of your original pile of stones, then the whole task would take you twice as long as before. This is exactly what it means to have a time complexity of $O(x)$: If you multiply the input size by n, then the task will take roughly n times as long as before. Isn't it nice to see that mathematics agrees with intuition?

We are now going to take a quicker look at multiplication in Stutter. The execution of * is illustrated in Table 3.3. The multiplication listing is very similar to the listing for addition. Note, however, that we did not expand each of the + function calls. A quick look of the listing shows that to perform (* a b) requires $b+1$ expansions, b 1- operations, and b + operations, with a always being the second argument to the + function. As before, the worst-case scenario for the * function is for a and b to be equal to one another. Thus, denoting the input length by x, we know that a and b are equal to $\frac{x}{2}$. A quick estimate of the running time reveals that the costliest portion of the function is the $\frac{x}{2}$ additions that we have to make with $\frac{x}{2}$ being the second argument. Since we are performing an $O(x)$ operation $\frac{x}{2}$ times with input length $\frac{x}{2}$, the time complexity of the * function is equal to $O(x^2)$, which means that if you double the size of the arguments to the * function, you can reasonably expect the function to take four times as long. Similarly, if you increased the size of the arguments by a factor of n, it would take n^2 times as long to compute the product. This means that in some ways multiplication is "harder" than addition, but you knew this already. Once again, mathematics confirms intuition.

We are now ready to go back to the question posed at the beginning of this section, "Which model of computation is more efficient than the others?" If you were hoping for a definitive answer, then I am afraid you are going to be disappointed, since some models of computation are ideally suited for certain problems that are difficult for other models. However, what is truly interesting is that no matter what the problem is and no matter what model you use to solve it, any of the other models can compute the same result in time proportional to some polynomial of what it took the first model to compute it. In other words, if it takes Stutter $O(f(x))$ time steps to compute some function, then in the worst case a Turing machine can do the same thing in $O(g(f(x)))$ time, where g is a polynomial function.

Polynomial functions always have the form

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0,$$

where the a_i terms are coefficients and n is the largest power. Under time complexity analysis we would simplify the above function to $O(x^n)$. Computer scientists like functions that take polynomial time because of all the possible functions, polynomials are relatively well-behaved. On the other extreme, many well-known problems have no known solution that takes less than exponential time. To see the difference in how these functions can grow, you can take a moderate polynomial like x^4 and a small exponential function like 2^x. For small values of x (less than 16), the polynomial will be larger than the exponential. For slightly larger values of x, such as 20 or 30, the exponential will explode in size relative to the polynomial. Another redeeming feature of polynomials is that they are closed under composition. This means that if you take a polynomial of a polynomial, you will still have a polynomial. Therefore, if a problem can be solved in a "reasonable" amount of time under one model of computation, it can be solved in a "reasonable" amount of time with any of the other models. If the problem takes exponential time to solve, then it really isn't all that solvable to begin with, and taking a polynomial of an exponential makes it marginally worse. Thus, relatively speaking, all of the models of computation are roughly equivalent in speed. Nevertheless, if you try to use the Stutter exponentiation function, you'll be in for a long wait, so these facts should be taken with a grain of salt.

3.5 Universal Computation and Decision Problems

One of the nice things about Lisp and Stutter is that function definitions are also lists. This property not only is cosmetically appealing but also lends the languages a certain degree of power, in that functions can operate on function definitions very easily. For example, in Stutter (or Lisp) one could theoretically write a Stutter program with only about a hundred lines of code that is actually a Stutter interpreter running on top of the original Stutter interpreter. You may have seen commercial software that allows one type of computer to run software from another type. Such programs are known as emulators, and they are normally very complicated. Moreover, to write a program on a computer that emulates the computer that the program is running on is normally very difficult. But Stutter's simple and compact

```
• Input $x$ and $y$.
• If $x = 0$, then output 0 and halt.
• If $y = 0$, then output 1 and halt.
• (We can assume that $x \geq 1$ and $y \geq 1$.)
• Set $h$ to 1.
• Repeat until $x^y < h$ is true.
        • Set $l$ to $h$.
        • Set $h$ to $2 * h$.
• (We now know that $x^y$ is between $l$ and $h$.)
• Set $m$ to $(h + l)/2$.
• Repeat until $x^y < m$ is false and $x^y < m + 1$ is true.
        • If $x^y < m$ is true, then set $h$ to $m$.
        • If $x^y < m$ is false, then set $l$ to $m$.
        • Set $m$ to $(h + l)/2$.
• Output $m$.
```

Table 3.4 An algorithm that computes x^y from queries.

representation makes doing the same thing under Stutter relatively easy. However, having a universal computer emulate itself is always possible, no matter what the underlying model of computation is.

Recall that we were able to reduce any program's input into a single natural number via Gödelization. Using the same technique as before, it is possible to represent all of the Stutter primitive function names and punctuation characters as a list of integers. For example, we could represent `car`, `cdr`, `cons`, `if`, `lambda`, `quote`, `set`, '`)`', and '`(`' by the numbers 1 through 9. Any additional variables or atom names that we need for a program could be mapped into the numbers 10, 11, 12, and so on. Now that the program is represented as a list of natural numbers, one could code the entire program into a single Gödel number.

The important thing to realize is that any of our models of computation can convert a program representation into a Gödel number. Moreover, they also can invert the process to retrieve the original program. Combining the facts that computers can invert Gödel numbers and emulate themselves means that for any computational model there theoretically exists a very special program that takes two numbers as input and performs the following computation. The first number is interpreted as the Gödel number of a program, and the second number is interpreted as the Gödel number for the input that one would want to supply to the program represented by the first input number. This special program can emulate the Gödelized program on the Gödelized input as if the real program were being executed. Such a program is known as a universal computer, and I will use the notation $U(x, y)$ to mean that "the universal computer is executed with the Gödel number x of some program on the Gödel number y as input."

We are also used to thinking of programs as producing some sort of meaningful output. Yet it is possible to take any program and convert it to another program that performs a computation similar to that of the first but outputs only either a 1 or a 0. The new program is referred to as solving a *decision problem*. Using decision problems will provide us with another mathematical shortcut later on, but for now let's see what this idea really means.

Consider a program that takes two inputs, x and y (or a single Gödel number for the two inputs), and outputs x^y. We could use this program to write another program that takes three inputs, x, y, and z, and outputs a 1 if $x^y < z$ and 0 otherwise. Now suppose that you never really have access to the first program, and you are allowed to only use the second program. How would you find out what x^y really is? Table 3.4 gives an algorithm that computes x^y by querying the second program. The basic idea behind the algorithm is to perform what is known as a binary search. It works in two stages. In the first stage the algorithm figures out an upper bound for x^y by doubling an estimate until the estimate exceeds x^y. Since the previous estimate was less than x^y, we can assume that x^y must be between l (for low) and h (for high). In the second stage the algorithm computes a middle point, m, between l and h and checks to see which half of the range (l to h) x^y is in. The value of l or h is updated to reflect the in which half x^y was found, and the process is repeated until x^y is finally isolated.

How significant is the extra overhead in computing x^y in such a manner? Surprisingly, not very. As a worst case, exponentiation in Stutter takes $O(2^x)$ time (where x is the input length). However, the binary search performs the exponentiation operation only $\log_2(x)$ times in the worst case, which is actually better than a polynomial. Therefore, computing an exponent via a decision problem is just as "hard" as the original exponentiation program but not significantly harder.

In general, any program can be converted into a similar program that solves a decision problem, that is, set membership determination for some predetermined set. Moreover, using the decision program instead of the original program increases the original complexity only by an amount polynomial in the original running time. Why would you want to compute something in such an awkward manner? You wouldn't, but theoretically this gives us a simplification in how computers work that we will exploit in the next section.

3.6 Incomputability

Do there exist problems that are unsolvable by any computer? "Unsolvable" should be understood in the strictest sense of the word; that is, if all of the computers in the world worked in conjunction on one specific problem and they theoretically re-

	Natural Numbers						
Subsets	1	2	3	4	5	6	⋯
Even Numbers	$\boxed{\text{no}}$	yes	no	yes	no	yes	⋯
Odd Numbers	yes	$\boxed{\text{no}}$	yes	no	yes	no	⋯
Primes	no	yes	$\boxed{\text{yes}}$	no	yes	no	⋯
Squares	yes	no	no	$\boxed{\text{yes}}$	no	no	⋯
Powers of 2	yes	yes	no	yes	$\boxed{\text{no}}$	no	⋯
Multiples of 3	no	no	yes	no	no	$\boxed{\text{yes}}$	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱
Diagonal Set	yes	yes	no	no	yes	no	⋯

Table 3.5 Listing out simple sets to derive a diagonal set.

quired a billion times the age of the universe to finally compute the correct answer, we would still consider such a problem "solvable." This definition may seem unreasonable, but the fact is that there are many problems that can't be solved even with such loose criteria.

Recall once again that every program can be reduced to another program that computes set membership. Some computable sets for sophisticated programs are very complex, but others should be familiar to all of us. Listed in Table 3.5 are some simple sets with the membership for a particular number being given in the entries.

Since the number of programs and computable sets is countable, we can enumerate all of the computable sets (or programs) in one long list, just as we did with the natural numbers in Chapter 2. Once again, the entries along the diagonal are boxed for emphasis. With the diagonal entries we can construct a diagonal set such that each member of that set is exactly opposite the entry in the main diagonal. Amazingly, the full infinite set represented by the complement of the diagonal is not computable because it differs by at least one entry from every computable set; thus no computer program could ever exist that could tell you for any natural number whether it was a member or not. Our diagonal set is a nice illustration and serves as an existence proof, but it really doesn't tell us what a truly impossible problem looks like.

Alan Turing was the first to demonstrate that there are many problems that are not computable. For the rest of this section, we will concentrate on a single noteworthy incomputable problem that Turing discovered. Suppose you had a program and some data that you wanted to run the program on. A reasonable question to ask is: Will the program ever halt with a solution? Let's assume that the program you want to check looks for the solution to a problem and halts only when it finds one. Therefore, the program may not ever stop if a solution doesn't exist (or if it is simply too dumb to find one). Wouldn't it be nice if there existed a special program that could tell us if another program would not halt? We could save a lot of time by seeing if the program we are really interested in would halt, and only then would we go to the next step of running it to find the solution. If such a program existed, programmers could use it to see if other programs had bugs in them that caused infinite loops. You could also use this special program on your home computer's operating system (which is just another program) to eliminate those annoying software crashes.

Let's assume that this special program exists. We will denote an instance of this program by the notation $M(x, y)$, where x is the Gödel number of the program that we wish to check and y is the Gödel number of the input that we want to feed to program x. Note that this is very similar to the universal computer that we constructed in the last section, but instead of producing the normal output of program x on y, it outputs

$$M(x,y) = \begin{cases} 1 \text{ if } U(x,y) \text{ halts} \\ 0 \text{ if } U(x,y) \text{ does not halt} \end{cases} .$$

If the program $M(x, y)$ exists, then we can easily construct another program, $M'(x)$, based on it that gives the output

$$M'(x) = \begin{cases} \text{runs forever in an infinite loop if } M(x,x) = 1 \\ \text{halts with any output if } M(x,x) = 0 \end{cases} .$$

Don't be bothered by the fact that we have purposely designed M' so that it can conceivably run forever (that is, diverge). Doing so is actually quite easy; for example, the Stutter function defined by `(set 'f (lambda () (f)))` will run forever if called. Since M' is just another program, it has its own unique Gödel number, which we will call m'. Let's see what happens if we try to run $M'(x)$ with an input of m'. What do you think will happen? Will it halt or run forever?

Let's assume for a moment that $M'(m')$ halts. If this is so, we know that $M(m', m') = 0$ must be true. This means that program m' with input m' runs forever, which contradicts our original assumption that $M'(m')$ halts. Similarly, assuming that $M'(m')$ runs forever, we can conclude that $M(m', m') = 1$, which further implies that program m' on input m' halts. Again, a contradiction! We are faced with contradictions no matter what assumption we make because the original program, $M(x, y)$, simply cannot exist since it pretends to solve an unsolvable problem.

Notice that the real difficulty in constructing a program similar to $M(x, y)$ is in determining when a program fails to halt. If a program will halt, then plugging the program and input into the universal computer will tell us this is the case because the universal computer will also halt. Determining if a program will not halt is impossible because you would have to wait forever to see that it did not halt.

One tantalizing facet of our proof above is that it relies on running the program M' on its own Gödel number, m'. Think about this: We are feeding a program itself as input. In other words, the program is examining itself and trying to perform some sort of self-analysis: Do I halt—or not? "Know thyself" seems to be an impossible command for a computer program. If a model of computation is so weak that it cannot ask questions about programs, then it will skirt the abyss, but at the cost of being too stupid to solve interesting problems. Once a model of computation has the ability to look within itself, it pays the price of not being able to halt under the right (or wrong) set of circumstances.

3.7 Number Sets Revisited

Let's put a few of the ideas from the last two sections together. Since every program has a unique Gödel number, let's think about what those numbers look like. Obviously they will be very large numbers. Moreover, not every natural number will represent a legal program; for example, we could compute a Gödel number for the character sequence `(xy))(z''()`, but such a string of characters does not represent a legal Stutter program because it violates the syntax of the language. However, syntax checking is not too difficult a problem, so it would be theoretically possible to write a program that takes a single natural number as input and decides if it represents a Gödelization of a legal program. With this program, we could write another program that also takes a single natural number, x, as input, but this time the new program will output the xth Gödel number that represents a legal Stutter program.

This new program maps the natural numbers to programs, just as we did in Chapter 2 with simpler sets. Since we now know how to do a one-to-one mapping of natural numbers to programs, consider the set of numbers that map to a program that halts. This set is known as the halting set and has the property that it is recursively enumerable (RE for short) but not computable.

A *recursive set* is a set of numbers such that some program can decide if a number is a member or not. Recursive sets obviously can be finite in size, since a program to decide set membership for a finite set can always be written as a simple lookup table. There are also recursive sets that are infinite in size, such as the set of even numbers, or primes, or the infinite set of numbers that represent the Gödelization of three numbers, x, y, and z, such that $x^y < z$ is true.

The really important attribute that recursive sets have is that a program can determine if a number is or is not a member. On the other hand, for strictly RE sets (i.e., a set that is RE but not recursive) a program can decide only if a number is a member of the set. If a number is not a member of an RE set, there is no general way of determining that this is the case.

Since for every strictly RE set there is a program that halts only if its input is a member of the RE set, there are as many RE sets as programs (and natural numbers); that is, there is a countably infinite number of RE sets. There is a close relationship between RE sets and the computable irrational numbers. Recall from Chapter 2 that two examples of irrational computable numbers are π and $\sqrt{2}$. These numbers are called "computable" because you can write a program that enumerates each digit, one after the other. All modern operating systems have the ability to multitask, which gives the impression that the computer is running many programs simultaneously. All of our formal models of computation are also capable of emulating an arbitrary number of "virtual" computers. With such a scheme, it is possible for a master program to spawn another virtual program that checks the set membership of one particular number. At a future time step, the master program can spawn another virtual program that checks a second potential member, and so on. If any of these virtual programs ever halts with the answer "yes, this element is a member of the RE set we are considering," then the master program can halt that particular virtual program. A virtual program may never halt with an answer of "no" but instead just keeps running forever. Thus, it is theoretically possible for one program to continuously spit out members of an RE set, which is why we say it is recursively enumerable. The master program will never halt, but there also will never be a last element that it gives a positive membership classification to. This master program is similar to a program that would continuously print out digits of π. The π program also will never halt (unless we explicitly tell it to halt at some future time), but always produce another digit, and another, and so on. The main difference between computable numbers and RE sets in this analogy is that we can't compute the nth member of an RE set the way we can compute the nth digit of π, but it is still interesting to see that both RE sets and computable numbers can, in a sense, be perfectly described by an algorithm.

Things get really strange when you start to think about sets that are not recursively enumerable, or "NOT-RE." To start, let's consider a set that is the complement of an RE set that is not recursive. The complement of a set consists of all numbers that are not in the original set. We will give the label "CO-RE" to sets that are constructed in this manner. The complement of the halting set is an example of a CO-RE set. We can think of CO-RE sets as being special cases of the more general NOT-RE set type.

Just as for RE sets, for a CO-RE set you could never write a program that would halt if and only if the input was a member of the CO-RE set. However, in many ways CO-RE sets are more pathological than RE sets because we cannot even determine in a general way if some number is a member. We can answer only the opposite question with any degree of reliability: Is this number not a member? This is reminiscent of trying to draw a picture only by filling in the background—we are not allowed to draw the foreground but can only infer things about it based on what we see in the background. I like to think of CO-RE sets as the "black holes" of recursive mathematics because of this property.

Yet, there are sets that are NOT-RE and also are not CO-RE. What could this possibly mean? Such a set defies algorithmic description, meaning that no

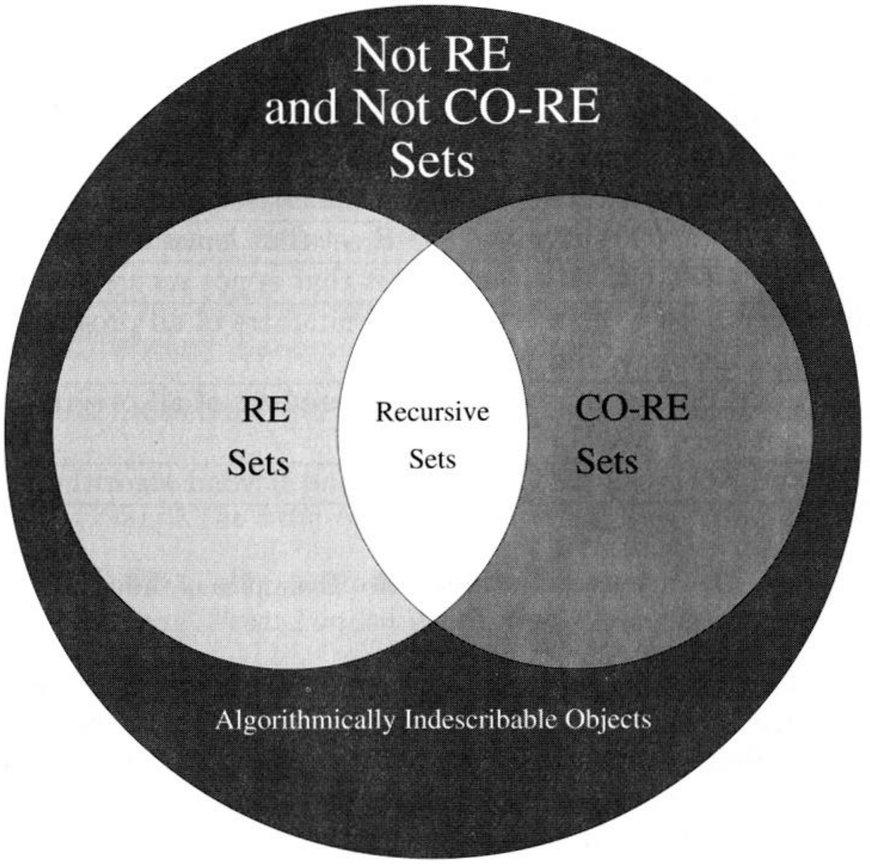

Figure 3.1 Subdivision of numbers and sets

program could tell you anything at all about a potential member with any certainty. The truly disturbing thing about such sets is that there is an uncountably infinite number of them—more than the recursive, RE, and CO-RE combined. These sets are analogous to the uncomputable irrational numbers. You can imagine that the digit expansion of an uncomputable irrational number is effectively random because there is no consistent relationship between any of the digits. (We will talk more about such random numbers in Chapter 9.)

Figure 3.1 gives a stylized representation of numbers and sets, and specific examples of these different set types are given in Table 3.6. Recursive sets are a special case because their complements are also recursive. Recursive sets are trivially RE as well. Sets that are not recursive but are RE are found in the large light-gray section on the left of Figure 3.1. The NOT-RE sets that are also CO-RE are in the darkest nonblack section. The black section contains the uncountably infinite NOT-RE sets that are "very" NOT-RE, in that their complements are also NOT-RE. You can view the lightest regions of the figure as denoting the most "knowable" sets. In the darker regions the sets under consideration become more and more "unknowable."

Set Type	Example
Recursive (finite)	1, 2, and 3
Recursive (infinite)	all even numbers
RE but not Recursive	the Gödel numbers of all programs that halt
CO-RE but not Recursive	the Gödel numbers of all programs that never halt
NOT-RE	a CO-RE set that is not recursive or a "random" set
NOT-CO-RE	an RE set that is not recursive or a "random" set
Recursive Subset of an RE but not Recursive Set	the Gödel numbers of all programs that provably halt
Recursive Subset of a CO-RE but not Recursive Set	the Gödel numbers of all programs that provably never halt
NOT-RE and NOT-CO-RE	"random" and beyond algorithmic description

Table 3.6 Examples of different types of sets

If you are still bothered by the fact that some infinities are larger than others, then I have some bad news for you: It gets worse. Recall that Georg Cantor was the first to realize that there is a difference between countable infinity and uncountable infinity. Cantor also discovered power sets, which is another way of deriving the differences in the infinities. A power set of another set, A, is the set of all subsets of A. For example, if $A = \{1, 2, 3\}$, then the power set of A is $\{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. Notice that we include the original set and the empty set. If A has n members, then its power set will have 2^n members. The set of all natural numbers has a power set. There is also a power set of the power set of all of the natural numbers. The sizes, or cardinalities, of each successive power set are known as the transfinite numbers, with the size of the set of natural numbers being the first transfinite number. It is regarded as true but not provable that the second transfinite number is equal to the cardinality of the real numbers. What is really astounding is that there is an infinite number of transfinite numbers.