Recursive Functions and Computability

Tom Davis
tomrdavis@earthlink.net
http://www.geometer.org/mathcircles
November 7, 2005
DRAFT!

1 What Is a Computation?

There are many ways to define what is meant by the term "computation", but most people would agree that a computation should satisfy the following characteristics:

1. A computation should be mechanical in the sense that there is a unique way to proceed at any point. (Or, at least if there is not a unique approach, then all approaches should lead to the same result. For example, to calculate the sum $x + y + z$ it doesn't matter whether $x + y$ is calculated first, or $y + z$. The best situation, of course, is if there is no ambiguity whatsoever.)
2. It must be possible to state the rules for a computation in a finite manner. We should not be able to list an infinte number of possible options.
3. Similarly, the size of the input and size of the output must be finite.
4. It should be clear when the process has terminated. (It may be that some computations never terminate, but those are usually not particularly useful, except, possibly, in a theoretical sense.)[1]

There are a number of different very precisely-defined mathematical schemes that satisfy the rules above. We can define a Turing machine that is a sort of mathematical model of a computer and see what it can compute. We can define a set of primitve functions that are obviously computable and some rules for combining the functions that can obviously be performed to produce new functions and then examine what sorts of functions can be generated. We can define a formal language with syntax and grammar and add rules for manipulation of sentences in the language and look to see what sorts of sentences can be "proved" from a set of sentences that are "obviously" true. Or even more simply, we can examine the sets of sentences that can be generated from

[1]This brings up the following consideration: suppose there is a computational method that satisfies all of the conditions above, but may not terminate in all cases. In other words, when you begin the computation, you know that if it terminates, it will terminate with the correct answer, but you do not know whether it will terminate, or have any upper bound on the amount of time it may take to complete the calculation. Just because you have been calculating for a million years does not mean that the computation will not terminate; it may terminate on the next step. A surprising number of very useful calculations that are performed all the time have this form.

various mathematically-defined grammars that generate valid sentences from a primitive starting point. There are other methods to approach the idea of what is meant by a computation as well.

What is incredible is that all the methods above result in *exactly* the same set of computations, if you interpret the results of each as encodings of the others. Although at first glance, these computational schemes may seem wildly different, each results in exactly the same set of answers, indicating that there is something incredibly special about the resultant set of computations.

There is no space in this article to examine all the ideas and approaches mentioned in the paragraphs above, but we will be able to take a look at one sort of computation; namely, computaions that can be performed by the so-called recursive functions.

The first restriction we will make is that we will consider only mathematical functions that take natural numbers (one or more) as inputs and yield a natural number as the output. In what follows, we will define the set of natural numbers[2] as follows:

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\}.$$

We want the term "computation" to include all of the obvious standard calculations: addition, subtraction, multiplication, division, factorization, integer powers, et cetera, as well as combinations of those. You may never have thought about operations like addition and multiplication as functions in the same way as you did $f(x)$ in your algebra class, but they are. As an example, let us show how one might indicate arithmetic operations like $x \cdot y + z \cdot (x + w)$.

One way is to replace the infix operators "$+$" amd "$\cdot$" by functions of two variables: $\mathcal{A}(x, y)$ represents $x + y$ and $\mathcal{M}(x, y)$ represents $x \cdot y$. Then the expression $x \cdot y + z \cdot (x + w)$ in the previous paragraph can be represented with the following functional notation:

$$\mathcal{A}(\mathcal{M}(x, y), \mathcal{M}(z, \mathcal{A}(x, w))).$$

We are of course interested in more complex operations as well. For example, suppose the function $f(n)$ is defined to be 0 if $n < 2$ and the largest prime factor of n otherwise. Thus $f(0) = f(1) = 0$, $f(2) = 2$, $f(3) = 3$, $f(4) = 2$, $f(5) = 5$, $f(6) = 3$, and so on. In normal circumstances, the definition above is sufficient, but here we are also interested in how $f(n)$ might be computed, given n. The expression "largest prime factor of n" really doesn't tell us a mechanical way to calculate that number.

The fact that we restrict ourselves to the consideration of only functions of the natural numbers is hardly a restriction at all. For example, every computer program falls into this category, since all the inputs are basically converted to binary (strings of zeros and ones) before the program operates on them, and the outputs are binary strings before the final conversion to characters or colors on the screen or whatever else is produced. We can just consider these binary input and output strings to be natural numbers so the computer program is just a function mapping natural numbers to natural numbers.

[2]Some people do not include zero as a natural number, but we will do so here.

Similarly, the floating-point numbers used in computer computation are only finite approximations of the "real" real numbers. They are saved as 32, 64, or occasionally 128 bit patterns, but these could equally well be considered to be 32, 64, or 128 bit natural numbers. To be sure, the calculation of $2.71828 + 3.14159$, interpreted as a purely integer operation is a *very* strange one, but it *can* be considered to be one.

2 Some Concrete Examples

Here are some examples that will be recursive functions, although to show them to be so will require some work.

2.1 The standard arithmetic functions

- The successor function: $\mathcal{S}(x) = x + 1$.
- The constant functions: (0, 1, 2, 3, ...).
- Addition and multiplication: $\mathcal{A}(x, y) = x + y$ and $\mathcal{M}(x, y) = x \cdot y$.
- Decrement, Subtraction, Division and Remainder: $x-1$, $x-y$, x/y and $x \bmod y$. Obviously subtraction cannot go negative, so $x - y$ is defined to be zero if $y \geq x$. Similarly, division yields the whole-number divisor and the modulus function yields the remainder, so $y \cdot (x/y) + x \bmod y = x$.
- Integer power: x^y.

Obviously we want to be able to define combinations of the functions above so that we can perform calculations like $(x + y)^{(zx-w)} - y$.

2.2 Recursive Functions

> Recursion: If you know what recursion is, just remember the answer. Otherwise, locate someone who is standing closer to Douglas Hofstadter than you are, and ask him/her what recursion is.
> *Andrew Plotkin*

The quotation above provides a surprisingly nice image of how recursion works: A recursive calculation is one that can be done for the smallest natural number, zero, and the calculation for larger numbers always depends on calculations for smaller ones.

2.2.1 The Factorial Function

Probably the first recursive defintion that most people see is for the factorial function: $f(n) = n! = n \cdot (n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1$:

$$f(n) = \begin{cases} 1 & : \text{ if } n = 0, \\ n \cdot f(n-1) & : \text{ otherwise.} \end{cases}$$

The recursive definition above tells us that $f(0) = 1$, and that to compute the value of $f(n)$ if $n > 0$ all you need to do is look up $f(n-1)$ and multiply that by n.

So to calculate 3! we see that the result is 3 multiplied by 2!, but then to get 2! you need to multiply 2 by 1!, and so on. The net result is $3 \cdot 2 \cdot 1 \cdot 1 = 6$. More formally:

$$\begin{aligned} 3! = f(3) &= 3 \cdot f(2) \\ &= 3 \cdot 2 \cdot f(1) \\ &= 3 \cdot 2 \cdot 1 \cdot f(0) \\ &= 3 \cdot 2 \cdot 1 \cdot 1 \\ &= 6. \end{aligned}$$

2.2.2 The Fibonacci Sequence

An interesting function generates the Fibonacci sequence, $1, 1, 2, 3, 5, 8, 13, 21, \ldots$, where the first two entries are 1 and after that, each is obtained by adding the previous two:

$$f(n) = \begin{cases} 1 & : \text{ if } n = 0 \text{ or } n = 1, \\ f(n-1) + f(n-2) & : \text{ otherwise.} \end{cases}$$

Here is the calculation for $f(5)$:

$$\begin{aligned} f(5) &= f(4) + f(3) \\ &= (f(3) + f(2)) + (f(2) + f(1)) \\ &= ((f(2) + f(1)) + (f(1) + f(0))) + ((f(1) + f(0)) + 1) \\ &= (((f(1) + f(0)) + 1)) + (1 + 1) + ((1 + 1) + 1) \\ &= (((1 + 1) + 1)) + (1 + 1) + ((1 + 1) + 1) \\ &= 8. \end{aligned}$$

The example above deserves a couple of comments. First, the calculation as presented is very inefficient: the values of $f(n)$ for the small values of n are calculated over and over again. Second, in the calculation above a number of substitutions of $f(n)$ by $f(n-1) + f(n-2)$ were made in each line. One of the ways we characterized a computation in the first section of this article was as a sequence of operations where there was no question about which one was to be done first. In this case, it makes no difference, and in addition, it would be easy to specify exactly the order in which the substitutions should be made. For example, we could just say that at any stage of the calculation, the leftmost possible expansion/substitution would be done.

2.2.3 The Greatest Common Divisor

An extremely important function yields the greatest common divisor of two numbers. For example, $\gcd(12, 18) = 6$, since 6 is the largest number that divides both 12 and 18 evenly. There is a very nice recursive definition for the gcd function:

$$\gcd(m, n) = \begin{cases} n & : \text{ if } m = 0, \\ \gcd(n \bmod m, m) & : \text{ otherwise.} \end{cases}$$

For example, to calculate $\gcd(29680, 17360)$, we proceed as follows:

$$\begin{aligned} \gcd(29680, 17360) &= \gcd(17360 \bmod 29680, 29680) = \gcd(17360, 29680) \\ &= \gcd(29680 \bmod 17360, 17360) = \gcd(12320, 17360) \\ &= \gcd(17360 \bmod 12320, 12320) = \gcd(5040, 12320) \\ &= \gcd(12320 \bmod 5040, 5040) = \gcd(2240, 5040) \\ &= \gcd(5040 \bmod 2240, 2240) = \gcd(560, 2240) \\ &= \gcd(2240 \bmod 560, 560) = \gcd(0, 560) \\ &= 560. \end{aligned}$$

The result, 560 is correct, since $29680 = 53 \cdot 560$ and $17360 = 31 \cdot 560$, and 53 and 31 are obviously relatively prime. Notice that the first application of recursion simply reverses the inputs which guarantees that afterwards the first parameter will be smaller than the second. After that, the recursion guarantees that each successive application has smaller values for the first parameter, and so it will eventually get to zero. Finally, it is not hard to show that at every stage of the calculation, the remaining numbers have the same greatest common divisor as the previous pair of numbers.

2.2.4 Some Simple Recursive Functions

The examples above are all fairly famous; let us now consider some very simple recursive functions. These examples show that even the arithmetic operations like addition and multiplication that we consider to be "primitive" can in fact be defined in terms of the even more primitive operation, successor.

The successor function: $\mathcal{S}(n) = n + 1$ yields the next larger number above n. With the successor function defined, we can define the addition function $\mathcal{A}(m, n) = m + n$ recursively as follows:

$$\mathcal{A}(m, n) = \begin{cases} n & : \text{ if } m = 0, \\ \mathcal{A}(k, \mathcal{S}(n)) & : \text{ if } m = \mathcal{S}(k). \end{cases}$$

The definition above basically amounts to saying that if you add zero to a number, it remains the same, and if you add m and n, that's the same as adding $m - 1$ and $n + 1$. Eventually the repeated subtractions of 1 from m will get it down to zero, and we know how to add zero to any number.

A slightly different method can be used to define multiplication: $\mathcal{M}(m,n) = m \cdot n$, in terms of addition: It's easy to multiply by zero, and to multiply m by n amounts to the same thing as multiplying $m-1$ by n and adding n:

$$\mathcal{M}(m,n) = \begin{cases} 0 & : \text{ if } m = 0, \\ \mathcal{A}(n, \mathcal{M}(k,n)) & : \text{ if } m = \mathcal{S}(k). \end{cases}$$

Exponentiation follows the same pattern as multiplication.

These examples show that we really need nothing more than the successor function to define the rest of the standard arithmetic functions. We will now go back to the beginning, and give a very formal defintion of what is meant by a recursive function.

3 Recursive Functions: Informal and Formal Definitions

One way to build up a class of functions is to begin with a tiny set of relatively simple functions and to build additional functions based on combinations of the simpler ones. Then more functions are built upon those and so on. As a simple, concrete example, let's consider the primitive recursive functions.

The general strategy will be this: we will define the set by giving a set of functions that belong to it, and then we will give two rules that will allow us to build more functions from functions that are already in the set. Of course once you add those functions to the set, it may be possible to apply the rules for building more functions to those and thereby obtain even more functions, and so on. If this process: beginning with a few functions and then applying the function-building rules to all of them and adding the newly-constructed functions to the set, is repeated forever, the resulting set will contain all the primitive recursive functions.

We will choose the initial functions to be functions that are "obviously" easy to compute: it should be obvious that any human or computer should be able to make those computations. The rules for building functions should also produce functions that are obviously computable. In other words, they should have the form so that *if* you are convinced that the component functions are computable, it should be obvious to you that the resulting functions will also be.

Later, when we build the set of so-called "general recursive functions", we will use exactly the same strategy. The only difference will be that in addition to the initial functions and the rules for primitive recursion, we will add a single additional function-building rule.

3.1 An Informal Description

In the following section (3.2) we will give a totally formal description of the primitive recursive functions, but it's easy to get lost in the forest of subscripts. In this section, we'll give an intuitive description of what we're trying to do, and then when you read the formal definitions in the next section, it should be easy to understand them, based on the simpler examples we present here.

Every function will take some number of natural numbers as inputs and will return a natural number as an output.

3.1.1 The Initial Functions

There will be three types of functions that we'll initally add to the set, before we try to make more functions from our function-generating rules. The first type consists of only a single function, the successor function: $\mathcal{S}(n) = n + 1$. It takes a single natural number as an input, adds 1 to it, and returns the result. Hopefully, most people will agree that the successor function is computable for any natural number input.

The other two types consist of an infinite number of functions, but again, hopefully most people would agree that all of them are computable. The first type consists of all the functions that return zero, no matter what the inputs are. This is an infinite set of functions, since we would like to include functions of 1, 2, 3, ... variables. Now in practical use, it's hard to imagine a function that takes a billion (or a google) of natural numbers as inputs, but just to be on the safe side, we'll include all of them. But the computation is quite simple: ignore all the inputs and return zero! Here's what the list would look like:

$$\begin{array}{l} z_1(x) = 0 \\ z_2(x, y) = 0 \\ z_3(x, y, z) = 0 \\ z_4(x, y, z, w) = 0 \\ \cdots \end{array}$$

The second type is essentially a doubly-infinite list of functions, but again, they are all quite simple. Each one returns a particular input with no computation done on it. So what we add is a series of functions like this:

$$\begin{array}{llll} p_{11}(x) = x & & & \\ p_{12}(x, y) = x & p_{22}(x, y) = y & & \\ p_{13}(x, y, z) = x & p_{23}(x, y, z) = y & p_{33}(x, y, z) = z & \\ p_{14}(x, y, z, w) = x & p_{24}(x, y, z, w) = y & p_{34}(x, y, z, w) = z & p_{44}(x, y, z, w) = w \\ \cdots & & & \end{array}$$

The idea is simple: we need a set of functions for every possible number of inputs that simply return each of the possible inputs. Thus we will need a million different functions that take one million inputs: one that returns the first, one that returns the second, and so on. Again, most people would agree that, in principle, all of these functions are computable. Note that we only talk about functions with a finite number of inputs, but of *every* finite size. The second subscript that is part of the function names indicates how many parameters it takes, and the first one tells which of those parameters will be returned.

These are called the “projection functions”.

What is truly amazing, is that the three function types above: the successor function, the functions that return zero, no matter what, and the functions that simply return one of their inputs, is all that we need to construct almost any function taking natural numbers as input and returning a natural number that you can imagine.

3.1.2 Composition: The First Construction Method

The idea is simple: if you know how to compute some functions, you should be able to use the outputs of those functions as inputs to other functions. In real analysis, you’ve seen expressions like $\sin(x^2)$ which essentially takes the value x and feeds it to the squaring function to obtain x^2. Then that value is provided as input to the $\sin$ function which returns its value.

The formal definition in section 3.2 may seem like an overly-complicated way to express the idea of composition, but it results from the fact that we need to be able to do this sort of composition of functions with different numbers of inputs. In other words, if $f(x, y)$, $g(x, y)$ and $h(x, y)$ are three functions that we know how to compute, and $F(x, y, z)$ is another, then we should be able to compute a new function $G(x, y)$ defined as follows:

$$G(x, y) = F(f(x, y), g(x, y), h(x, y)).$$

In other words, G takes two input values and inserts those two into f, g and h, which yields three output values. These are then used as the three inputs that F requires.

In the definitons, we will require that each of the functions like f, g and h have the same number of parameters. This might appear to be a problem, since you can certainly imagine wanting to do something like this: $f(x, y)$ is a function with two inputs, but $g(x)$ and $h(x)$ only take a single parameter, but we’d like to define G as follows:

$$G(x, y) = F(f(x, y), g(x), h(y)).$$

This is not really a problem, since if we need to do something like this, we can define new functions $g'(x, y) = g(x)$ and $h'(x, y) = h(y)$. This kind of definiton can easily be made with the projection functions illustrated in section 3.1.1. In fact, using the notation in that section, we can define:

$$h'(x, y) = h(p_{22}(x, y)),$$

where the projection function p_{22} snags the second parameter of a two-parameter function and returns it. Note that the expression above has the correct form for function composition that we’ve allowed.

Note that the projection functions will allow us to do things like permute or collapse the order of parameters. For example, suppose that we’ve got an interesting function already defined: $f(x, y, z)$ and we would like to define a new function g as follows:

$$g(x, y) = f(y, x, x).$$

With the projection functions, it is easy:

$$g(x, y) = f(p_{22}(x, y), p_{12}(x, y), p_{12}(x, y)).$$

3.1.3 Recursion: The Second Construction Method

The examples of recursion we've seen in earlier sections essentially tell how a function behaves for the smallest input value (zero) and if the input is not zero, it tells how to obtain the result based on the value of the function with a smaller input. This process is guaranteed to terminate, since the input value cannot be smaller than zero, and when it hits zero, we've given an effective way to complete the computation.

The examples are almost all simple in that they take only a single parameter, and surely we'd like to allow for multiple parameters. To keep things simple, though, we'll only allow recursion on one parameter, and this is not a real restriction. Thus we'd like our definition to be something like this:

If $n = 0$, then $f(n, x, y, \ldots, z)$ can be computed as the value of some function $g(x, y, \ldots, z)$, where g is some function that we know we can compute. But if $n > 0$, we want to have a function h that may depend not only on the values $x, y, \ldots, z$, but also possibly on n and on $f(n-1, x, y, \ldots, z$. Thus the definition of f will depend on two functions: one, g, that uses one fewer parameter than f, and tells what to do if $n = 0$, and on another function h that has one more parameter than f and tells how to do the computation based on the values that f provided, but also the value of f when called with $n - 1$ as its first parameter.

Note that there's no penalty for requiring that h have all these parameters: it can ignore as many as it wants.

With all of the above in mind, take a look again at section 2.2.4, and see how the definitions of the initial functions and constrution methods could be applied to obtain functions like the addition and multiplication of natural numbers beginning only with the successor function.

Now we'll do essentially the same thing we did over again, but in a totally formal way.

3.2 Primitive Recursive Functions

The set of primitive recursive functions is the smallest set of functions that map the natural numbers into the natural numbers and include the following functions. The first three entries below name specific functions that must be included; the last two are basically "recipes" for creating new primitve recursive functions from those that have already been shown to be primitive recursive.

1. **The successor function.** $\mathcal{S}(n) = n + 1$.
2. **The constant zero functions.** $\mathcal{Z}_k(n_1, n_2, \ldots, n_k) = 0$.

3. **The projection functions.** $\mathcal{P}_i^{(k)}(n_1, n_2, \ldots, n_k) = n_i$, for every $k \geq i > 0$.
4. **Composition of functions.** If $f(n_1, \ldots, n_k)$ is in the set and if $g_i(n_1, \ldots, n_l)$ are in the set for $1 \leq i \leq k$, then

$$h(n_1, \ldots, n_l) = f(g_1(n_1, \ldots, n_l), \ldots, g_k(n_1, \ldots, n_l))$$

is in the set.

5. **Primitive recursion.** If $f(n_1, \ldots, n_k)$ and $g(n_1, \ldots, n_{k+2})$ are in the set, then the function h defined as follows is in the set:

$$\begin{aligned} h(0, n_1, \ldots, n_k) &= f(n_1, \ldots, n_k) \\ h(S(n), n_1, \ldots, n_k) &= g(h(n, n_1, \ldots, n_k), n, n_1, \ldots, n_k) \end{aligned}$$

The final recursion condition is very safe in the sense that it will always be possible to evaluate functions so defined. The value is defined at zero, and based on that, the values at $1, 2, 3, \ldots$, are also defined, each depending upon the result of a calculation with a smaller input.

All the so-called primitive recursive functions are thus defined for all possible input values. Such functions are called “total functions”.

3.3 Examples of Primitive Recursive Functions

The vast majority of commonly-used arithmetic functions are primitive recursive. In this section we’ll build up a few of them. In a sense, it is easier to work with recursive functions in this way than with Turing machines, since once we have constructed a recursive function and studied its properties, we can just use it as it stands in the definition of a more complex recursive function. Embedding working Turing machines inside other ones requires, at the least, a little more bookkeeping.

We will list a number of examples below, all in the same format. First, we’ll describe exactly the function to be implemented. Next, we will show the formal derivation using either the functions listed above or functions that we have derived in previous sections. Finally, if warranted, we will include some discussion of the method. Since the definitions tend to build one on the other, in this section we have been careful to give each function a unique name that may be used in a later definition.[3].

3.3.1 The constant functions

For any integers m and k, the function $\mathcal{C}_k^{(m)}(n_1, \ldots, n_k) = m$ is primitive recursive.

[3]Although the defintions that follow may seem fairly trivial, it is a little bit tricky to get them exactly right. They may, in fact, not be exactly right in this text. The author was a bit paranoid about getting them right, so after he wrote the first draft of this article, he wrote a computer simulator for his definitions of primitive recursive functions. *Every one of the original definitions had at least a minor error, and some had major errors!*

$$\begin{array}{rcl}
\mathcal{C}_k^{(0)}(n_1,\ldots,n_k) & = & \mathcal{Z}_k(n_1,\ldots,n_k) \\
\mathcal{C}_k^{(1)}(n_1,\ldots,n_k) & = & \mathcal{S}(\mathcal{C}_k^{(0)}(n_1,\ldots,n_k)) \\
\mathcal{C}_k^{(2)}(n_1,\ldots,n_k) & = & \mathcal{S}(\mathcal{C}_k^{(1)}(n_1,\ldots,n_k)) \\
\mathcal{C}_k^{(3)}(n_1,\ldots,n_k) & = & \mathcal{S}(\mathcal{C}_k^{(2)}(n_1,\ldots,n_k)) \\
\ldots & = & \ldots
\end{array}$$

In other words, the successor of the functions that generate the constant output zero is the one that has constant output 0, and so on.

3.3.2 Addition of two natural numbers

The function $\mathcal{A}(m,n) = m+n$ is primitive recursive.

$$\begin{array}{rcl}
\mathcal{A}_1(n_1,n_2,n_3) & = & \mathcal{S}(\mathcal{P}_1^{(3)}(n_1,n_2,n_3)) \\
\mathcal{A}(0,n_1) & = & \mathcal{P}_1^{(1)}(n_1) \\
\mathcal{A}(\mathcal{S}(n),n_1) & = & \mathcal{A}_1(\mathcal{A}(n,n_1),n,n_1)
\end{array}$$

Addition is based on the idea that if you add zero to something, the result is the something. To add $n+1$ to something, you can just add 1 to what you get when you add n to that same something.

To define the addition function in a completely rigorous way, we need first to define the "helper" function $\mathcal{A}_1$ as above. This is because in the formal definition of recursion, when we define a function with k parameters, we need a $k+1$ parameter function to provide the recursive part of the definition.

This example is quite simple: if the first parameter is zero, simply return the second parameter. If the first parameter is the successor of n, work out the sum with n as the first parameter and then pass that sum, n, and the second parameter to the recursion-defining function. In this case, the recursion-defining function has no use for its second and third parameters; it simply selects its first parameter (using $\mathcal{P}_1^{(3)}$) and returns the successor of that.

Because the form of definition of primitve recursive functions is so rigid, we will often need to use the trick above of defining one or more helper functions.

3.3.3 Doubling a natural number

The function $f(x) = 2x$ is primitive recursive.

$$f(n_1) = \mathcal{A}(\mathcal{P}_1^{(1)}(n_1), \mathcal{P}_1^{(1)}(n_1))$$

Obviously, since we have defined addition in the previous section, multiplying by 2 is equivalent to adding a number to itself. The only thing that is slighty tricky in the defintion above is that to exactly satisfy the form for the substitution construction, we must pass functions of the parameter n_1 rather that the parameter itself. This is trivial to do: $\mathcal{P}_1^{(1)}$ is basically the identity function.

3.3.4 Multiplication of two natural numbers

The function $\mathcal{M}(x, y) = x \cdot y$ is primitive recursive.

$$\begin{aligned}
\mathcal{M}_1(n_1, n_2, n_3) &= \mathcal{A}(\mathcal{P}_1^{(3)}(n_1, n_2, n_3), \mathcal{P}_3^{(3)}(n_1, n_2, n_3)) \\
\mathcal{M}(0, n_1) &= \mathcal{Z}_1(n_1) \\
\mathcal{M}(\mathcal{S}(n), n_1) &= \mathcal{M}_1(\mathcal{M}(n, n_1), n, n_1)
\end{aligned}$$

Multiplication is defined recursively in terms of addition in much the same way that addition is defined recursively in terms of the successor function. Multiplication by zero yields zero, and to multiply by $n + 1$, you simply multiply by n and then add another copy. The structure of the helper function is similar to what we used for addition. Exponentiation, which we'll do next, seems almost identical, but we'll add one small trick.

3.3.5 Exponentiation of natural numbers

The function $\mathcal{E}(x, y) = x^y$, where 0^0 is defined to be 1 is primitive recursive.

$$\begin{aligned}
\mathcal{E}_2(n_1, n_2, n_3) &= \mathcal{M}(\mathcal{P}_1^{(3)}(n_1, n_2, n_3), \mathcal{P}_3^{(3)}(n_1, n_2, n_3)) \\
\mathcal{E}_1(0, n_1) &= \mathcal{C}_1^{(1)}(n_1) \\
\mathcal{E}_1(\mathcal{S}(n), n_1) &= \mathcal{E}_2(\mathcal{E}_1(n, n_1), n, n_1) \\
\mathcal{E}(n_1, n_2) &= \mathcal{E}_1(\mathcal{P}_2^{(2)}(n_1, n_2), \mathcal{P}_1^{(2)}(n_1, n_2))
\end{aligned}$$

Primitive recursion is defined for all inputs, so we cannot simply make the expression 0^0 be "undefined". Here, we've arbitrarily defined it to the 1. The construction the the function $\mathcal{E}$ is very similar to multiplication except that an exponent of 0 yields 1. The small trick here is that the function $\mathcal{E}_1(x, y) = y^x$, not x^y. To obtain x^y we need to swap the parameters, and that's what is done in the final line of the definition.

3.3.6 Decrement a natural number

The function $\Delta(x) = \max(0, x - 1)$ is primitive recursive. (We use the Greek letter delta (Δ) since we're reserving the symbol $\mathcal{D}$ to stand for subtraction. Think of it as standing for "difference".)

$$
\begin{aligned}
\Delta_1(0, n_1) &= \mathcal{Z}_1(n_1) \\
\Delta_1(S(n), n_1) &= \mathcal{P}_2^{(3)}(\Delta_1(n, n_1), n, n_1) \\
\Delta(n_1) &= \Delta_1(\mathcal{P}_1^{(1)}(n_1), \mathcal{P}_1^{(1)}(n_1))
\end{aligned}
$$

Here we would like to define the decrement function directly as a recursive function, but it only has one parameter, and every function defined using recursion according to our scheme must have at least two. We get around the problem by defining a helper function that has an unused parameter, and then we define decrement by plugging a "garbage" value into that second slot.

3.3.7 Subtraction of natural numbers

The function $\mathcal{D}(x, y) = \max(0, x - y)$ is primitive recursive.

$$
\begin{aligned}
\mathcal{D}_2(n_1, n_2, n_3) &= \Delta(\mathcal{P}_1^{(3)}(n_1, n_2, n_3)) \\
\mathcal{D}_1(0, n_1) &= \mathcal{P}_1^{(1)}(n_1) \\
\mathcal{D}_1(\mathcal{S}(n), n_1) &= \mathcal{D}_2(\mathcal{D}_1(n, n_1), n, n_1) \\
\mathcal{D}(n_1, n_2) &= \mathcal{D}_1(\mathcal{P}_2^{(}2)(n_1, n_2), \mathcal{P}_1^{(2)}(n_1, n_2))
\end{aligned}
$$

The idea here is that if y is zero, then $x - y$ is x. Otherwise, subtract $y - 1$ from $x - 1$. It's a little tricker that it seems, and to make it work nicely, it's easier to reverse the parameters.

3.3.8 Maximum of two natural numbers

The function $\max(x, y)$ is primitive recursive.

$$
\begin{aligned}
\mathcal{H}_1(0, n_1, n_2) &= \mathcal{P}_2^{(2)}(n_1, n_2) \\
\mathcal{H}_1(\mathcal{S}(n), n_1, n_2) &= \mathcal{P}_3^{(4)}(\mathcal{H}_1(n, n_1, n_2), n, n_1, n_2) \\
\max(n_1, n_2) &= \mathcal{H}_1(\mathcal{D}(n_1, n_2), \mathcal{P}_1^{(2)}(n_1, n_2))
\end{aligned}
$$

The $\max$ function works by subtracting the second parameter from the first, where the subtraction is the usual version restricted to natural numbers. If the result is zero, then the second element is larger (or the same size). To perform the "if" statement in the previous sentence, we need a helper function that we've called $\mathcal{H}_1(n, n_1, n_2)$ that basically looks at n_1 to see if it is zero. If $n_1 = 0$, it returns n_2; otherwise, n_1.

3.3.9 Parity of a natural number

The parity function $\Pi(x) = x \bmod 2$ is primitive recursive.

$$\begin{aligned}
\alpha(0, n_1, n_2) &= \mathcal{C}_2^{(1)}(n_1, n_2) \\
\alpha(\mathcal{S}(n), n_1, n_2) &= \mathcal{Z}_4(\alpha(n, n_1, n_2), n, n_1, n_2) \\
\Pi_1(0, n_1) &= \mathcal{Z}_1(n_1) \\
\Pi_1(\mathcal{S}(n), n_1) &= \alpha(\Pi_1(n, n_1), n, n_1) \\
\Pi(n) &= \Pi_1(\mathcal{P}_1^{(1)}(n), \mathcal{P}_1^{(1)})
\end{aligned}$$

A couple of helper functions make this work. The helper function α returns 1 if the first input is zero and zero otherwise. The helper function $\Pi_1(n, n_1)$ returns zero for a $n = 0$, and α of itself applied to $n - 1$ otherwise. It is a helper function since it is recursive and hence requires at least two parameters. The final definiton of the parity function Π simply invents a garbage second parameter for Π_1.

3.3.10 Sum of values of a primitive recursive function

If $f(n)$ is a primitive recursive function, then so is the following function:

$$\sigma_f(n) = \sum_{i=0}^{n} f(i).$$

$$\begin{aligned}
\sigma_1(n) &= f(\mathcal{S}(n)) \\
\sigma_2(n_1, n_2, n_3) &= \sigma_1(\mathcal{P}_2^{(3)}(n_1, n_2, n_3)) \\
\sigma_3(n_1, n_2, n_3) &= \mathcal{A}(\mathcal{P}_1^{(3)}(n_1, n_2, n_3), \sigma_2(n_1, n_2, n_3)) \\
\sigma_4(0, n_1) &= \mathcal{Z}_1(n_1) \\
\sigma_4(\mathcal{S}(n), n_1) &= \sigma_3(\sigma_4(n, n_1), n, n_1) \\
\sigma_f(n) &= \sigma_4(\mathcal{P}_1^{(1)}(n), \mathcal{P}_1^{(1)}(n))
\end{aligned}$$

There is nothing particularly tricky going on above—just a lot of technical problems with numbers of parameters.

Notice that with a couple of trivial changes we can also show that finite products of the same form of primitive recursive functions are primitive recursive:

$$\pi_f(n) = \prod_{i=0}^{n} f(i).$$

3.4 Characteristic Functions (or Predicates)

If $\mathbb{S} \subset \mathbb{N}$, then the characteristic function of $\mathbb{S}$, indicated by $\chi_{\mathbb{S}}(n)$, is defined to be 1 if $n \in \mathbb{S}$ and 0 if $n \notin \mathbb{S}$.

A characteristic function can also be thought of as a predicate (a special function that returns either "true" or "false") that takes inputs from the natural numbers. The usual convention is the 1 corresponds to "true" and 0 to "false". In this way you can imagine a function like $prime(x)$ (or $x > 7$) that returns 1 (true) if and only if x is prime (or $x > 7$).

We will be interested, of course, in which characteristic functions (or predicates) are primitive recursive (and later, general recursive) functions.

If characteristic functions are interpreted as predicates, then we can also investigate logical combinations of those functions. In this way we will be able to build predicates like $prime(x) \wedge (x > 7)$. (The $\wedge$ symbol signifies a logical "and" operation. In what follows, $\vee$ will indicate logical "or" and $\neg$ will indicate logical "not".)

It turns out that if two predicates p and q are primitive recursive (recursive) then the predicates $p \wedge q$, $p \vee q$ and $\neg p$ will also be primitive recursive (recursive). This is easy to see, since if p is a predicate, then $1 - p$ corresponds to $\neg p$. Similarly, if p and q are predicates, then pq (the product of p and q) corresponds to $p \wedge q$. From these two facts and from De Morgan's law: $p \vee q$ = $\neg((\neg p) \wedge (\neg q))$ we can infer that $p \vee q$ is also primitive recursive. (All the other common logical operations like implication, exclusive-or, nand, nor, et cetera, can be similarly constructed from $\wedge$ and $\neg$).

(Notice that if we consider the functions to be characteristic functions of subsets of $\mathbb{N}$ instead of predicates, then the constructions above yield new characteristic functions corresponding the set operations: union ($\cup \leftrightarrow \vee$), intersection ($\cap \leftrightarrow \wedge$), set complement ($' \leftrightarrow \neg$), et cetera.)

In this section we will show how to define a few interesting primtive recursive predicates. The definitons below will not be so formal as they were in the previous section. Hopefully the examples there indicate how one might go about constructing the absolutely formal definitions. In every case, this formalization can be accomplished.

3.4.1 True and False are primitive recursive

$$\begin{aligned} True(x) &= \mathcal{C}_1^{(1)}(x) \\ False(x) &= \mathcal{Z}_1(x) \end{aligned}$$

3.4.2 Finite sets

We will show that the characteristic function of any finite set is primtive recursive. This will be done by induction. The characteristic function of the empty set ϕ is: $\chi_\phi(x) = \mathcal{Z}_1(x)$. We will be done if we can show that if every characteristic function of a set of $n-1$ elements is primitive recursive then so is the characteristic function of any set with n elements.

We will begin by showing that the characteristic function of any singleton set is primitive recursive. Here is the definition of $\chi_{\{0\}}(n)$:

$$\begin{aligned} f(0, n_1) &= \mathcal{C}_1^{(1)} \\ f(\mathcal{S}(n), n_1) &= \mathcal{Z}_3(f(n, n_1), n, n_1) \\ \chi_{\{0\}}(n) &= f(\mathcal{P}_1^{(1)}(n), \mathcal{P}_1^{(1)}(n)) \end{aligned}$$

Next, suppose that we have defined $\chi_{\{0\}}(n), \chi_{\{1\}}(n), \chi_{\{2\}}(n), \ldots, \chi_{\{k\}}(n)$ and we wish to define $\chi_{\{k+1\}}(n)$. Here is how to proceed:

$$\begin{aligned} \chi_{h_1}^{(k+1)}(n) &= \chi_{\{k\}}(\mathcal{S}(n)) \\ \chi_{h}^{(k+1)}(n_1, n_2, n_3) &= \chi_{h_1}^{(k+1)}(\mathcal{P}_2^{(3)}(n_1, n_2, n_3)) \\ f(0, n_1) &= \mathcal{Z}_1(n_1) \\ f(\mathcal{S}(n), n_1) &= \chi_{h}^{(k+1)}(f(n, n_1), n, n_1) \\ \chi_{\{k+1\}}(n) &= f(\mathcal{P}_1^{(1)}(n), \mathcal{P}_1^{(1)}(n)) \end{aligned}$$

Now we have defined (by induction) the characteristic function of every singleton set. But we know from our previous discussion that if f and g are two predicates that are primitive recursive, then $f \vee g$ is also primitive recursive. But $f \vee g$ corresponds to the predicate of the union of the values for which both f and g are true, so we can build up a characteristic function for any finite collection of natural numbers.

3.4.3 Bounded Quantification

If $P(n, n_1, \ldots, n_k)$ is a primitive recursive predicate, then so is:

$$E_P^F(y, n_1, \ldots n_k) = \exists x\left((x < y) \wedge P(x, n_1, \ldots n_k)\right).$$

In English, this new predicate $E_P^F(y, n_1, \ldots, n_k)$ is true if there exists some value of $x < y$ such that $P(x, n_1, \ldots, n_k)$ is true. The "F" superscript indicates a finite search. Notice that this is just a finite product:

$$E_P^F(y, n_1, \ldots, n_k) = \prod_{i=0}^{y-1} P(i, n_1, \ldots, n_k).$$

We can similarly test to see if a predicate is true for all input values less than some fixed number since:

$$\forall x\left((x < y) \wedge P(x, n_1, \ldots, n_k)\right) = \neg\exists x\left((x < y) \wedge \neg P(x, n_1, \ldots, n_k)\right).$$

Based on this sort of operator, we can construct predicates that tell us about divisibility. To see if x divides y, just form a function like this:

$$x|y = \exists n \left((n < y) \wedge (\mathcal{M}(n, x) = y) \right).$$

3.4.4 Bounded Minimization

If we have a predicate $P(x, n_1, \ldots, n_k)$ it is very useful to find the *least* value of $x < y$ such that $P(x, n_1, \ldots n_k)$ is true. Using the results from the previous section, that is not too hard to do.

Consider

$$f(y, n_1, \ldots n_k) = \sum_{x<y} \left(\forall v((v < x) \wedge \neg P(v, n_1, \ldots, n_k)) \right).$$

This sum will basically add 1 each time there is a value of v where the predicate is false for it and for all smaller values. Once the predicate is true for the first time, all the other terms in the sum will add zero. Thus the sum will represent the smallest value where the predicate is true for the first time.

Based upon this, we can show that division is primitive recursive:

$$x \text{ div } y = \min_{z<y} \mathcal{M}(y, z+1) > x.$$

The mod function can be defined in terms of this:

$$x \text{ mod } y = y - x(x \text{ div } y).$$

Also from the fact that we can represent divisibility as a primitive recursive function, the least common multiple ($\mathrm{lcm}(m, n)$), greatest common divisior ($\mathrm{gcd}(m, n)$), et cetera can be constructed:

$$\begin{aligned} \mathrm{lcm}(m, n) &= \min_{p<mn} (m|p \wedge n|p) \\ \mathrm{gcd}(m, n) &= mn \text{ div } (\mathrm{lcm}(m, n)) \end{aligned}$$

It is also easy to define the $prime(x)$ predicate, then a predicate that identifies the i^{th} prime, the number of times the i^{th} prime divides a given number, and so on. In fact if we enumerate the prime numbers as $p_0 = 2, p_1 = 3, p_2 = 5$ and so on, then given any nunber n, we can write primitive recursive functions that will yield $k_0, k_1, \ldots$ such that $n = p_0^{k_0} p_1^{k_1} \ldots$.

We can then define functions that yield the prime factorization of numbers, allowing us to encode multiple numbers or even strings as single numbers. See 3.5.2.

3.5 Functions that are not Primitive Recursive

The examples above begin to show that the number of functions that are primitve recursive is vast—almost every function we use in day-to-day calcutions seems to fit into that category. But in fact, there are functions that are obviously computable that are *not* primitive recursive. In this section we will show how to construct one.

The proof that there are effectively computable total functions that are not primitive recursive is based on two ideas:

- It is possible to enumerate *all* of the primitive recursive functions in a logical way such that we can say, $\mathcal{F}_i$ is the i^{th} primitve recursive function, and for every primitive recursive function g, there exists a k such that $\mathcal{F}_k = g$.
- Once we have enumerated all the primitve recursive functions, we use a diagonalization argument to produce a function that is *not* in the list. Since the list includes all the primitive recursive functions, this new function cannot be primitive recursive. On the other hand, we can calculate the value of this new function for any input, and hence it is effectively computable.

The description above glosses over some of the difficulties, and we will eventually make the argument completely rigorous, but first let's see the general idea.

Suppose that the list were not all primitive recursive functions, but rather just the primitve recursive functions of a single variable. (We will see how to produce this from the original list later on.)

Thus $\mathcal{F}_0(n)$ is the zeroth primitive recursive function of a single variable; $\mathcal{F}_1(n)$ is the first, $\mathcal{F}_2(n)$ is the second, and so on. (We could have begun numbering the functions at 1, but it is slightly simpler to see what is going on if we start numbering from 0.

In principle we can now fill in a table of all the values of all the primitive recursive functions that looks like this:

$\boxed{\mathcal{F}_0(0)}$	$\mathcal{F}_0(1)$	$\mathcal{F}_0(2)$	$\mathcal{F}_0(3)$	$\mathcal{F}_0(4)$	$\cdots$
$\mathcal{F}_1(0)$	$\boxed{\mathcal{F}_1(1)}$	$\mathcal{F}_1(2)$	$\mathcal{F}_1(3)$	$\mathcal{F}_1(4)$	$\cdots$
$\mathcal{F}_2(0)$	$\mathcal{F}_2(1)$	$\boxed{\mathcal{F}_2(2)}$	$\mathcal{F}_2(3)$	$\mathcal{F}_2(4)$	$\cdots$
$\mathcal{F}_3(0)$	$\mathcal{F}_3(1)$	$\mathcal{F}_3(2)$	$\boxed{\mathcal{F}_3(3)}$	$\mathcal{F}_3(4)$	$\cdots$
$\mathcal{F}_4(0)$	$\mathcal{F}_4(1)$	$\mathcal{F}_4(2)$	$\mathcal{F}_4(3)$	$\boxed{\mathcal{F}_4(4)}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Next, we will define a new function $\mathcal{G}(n)$ that disagrees with the function $\mathcal{F}_i(n)$ when $n = i$. This is easy to do: $\mathcal{G}(n) = \mathcal{F}_n(n) + 1$. This $\mathcal{G}(n)$ disagrees with the function $\mathcal{F}_i(n)$ at least at the boxed values of $\mathcal{F}_i$ in the table above. Since $\mathcal{G}$ disagrees with every primitve recursive function for at least one input value, $\mathcal{G}$ must not be in the list, so $\mathcal{G}$ must not be primitve recursive.

On the other hand, this $\mathcal{G}$ which we have constructed is obviously computable. If you want to find the value for $\mathcal{G}(17)$, simply figure out what the 17^{th} primitive recursive function is, evaluate that for the input value of 17, and add one to the result—a perfectly mechanical procedure.

3.5.1 Enumerating the Primitive Recursive Functions

Now for the more interesting part: how can the primitive recursive functions be enumerated? We have to be at least a little bit careful, since if we begin with the most obvious approach, numbering the constant functions, we run out of natural numbers before we get to any other functions.

In what follows, we will be trying to generate a "grand list" of all primitve recursive functions:

$${}_0, \mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3, \ldots$$

Specifically, if we say $\mathcal{F}_1$ is the constant zero function with one argument, $\mathcal{F}_2$ is the constant zero function with two arguments, and in general, $\mathcal{F}_n$ is the constant zero function with n arguments, then we have enumerated all the constants, but there are no more natural numbers to assign to other functions; they are all used up.

And what should be done for the projections? For every $i > 0$ and $j > 0$, there is a $\mathcal{P}_i^{(j)}$. If we're not careful, we could also use up all the natural numbers on the subset $\mathcal{P}_1^{(j)}$.

Here is the basic idea: the first function, $\mathcal{F}_0$, will be the successor function, $\mathcal{S}(n)$. Next, we will make four lists of all the functions generated by the following four conditions, and will add functions to our list of all primitve recursive functions by cycling through the lists: first item from the first list, first item from the second, first from the third, first from the fourth, second from the first list, and so on.

The list of constant functions is easy:

$$\mathcal{Z}_1(n_1), \mathcal{Z}_2(n_1, n_2), \mathcal{Z}_3(n_1, n_2, n_3), \mathcal{Z}_4(n_1, n_2, n_3, n_4), \ldots$$

The list of projections is a little trickier, but not difficult. We simply list all of the $\mathcal{P}_i^{(j)}$ having $i + j$ equal to 2, then to 3, then to 4, et cetera. Omitting the parameters, this is what the list of the $\mathcal{P}_i^{(j)}$ will look like:

$$\mathcal{P}_1^{(1)}, \mathcal{P}_1^{(2)}, \mathcal{P}_2^{(1)}, \mathcal{P}_1^{(3)}, \mathcal{P}_2^{(2)}, \mathcal{P}_3^{(1)}, \mathcal{P}_1^{(4)}, \mathcal{P}_2^{(3)}, \mathcal{P}_3^{(2)}, \mathcal{P}_4^{(1)}, \mathcal{P}_1^{(5)}, \ldots$$

The list of functions made by composition is the trickiest, so we will discuss that last. Next, let's look at the list of functions defined by primitive recursion.

Any pair of primitive recursive functions f and g will define a new primitve recursive function h as long as g has two more parameters than f. By the time we need to add the first function of this sort to our grand list, the grand list will already contain at least three functions: $\mathcal{S}(n)$, $\mathcal{Z}_1(n_1)$ and $\mathcal{P}_1^{(1)}$. Each successive time we are faced with adding functions generated by the third and fourth rule, there will be more functions on the grand list.

If we list all pairs of functions on the grand list as we did for the projection functions above, then each time we need to look at a composition or recursive definition, we can use sets of functions previously added to the grand list. Here is the list of pairs we will consider in the case of functions defined by recursion:

$$(\mathcal{F}_1, \mathcal{F}_1), (\mathcal{F}_1, \mathcal{F}_2), (\mathcal{F}_2, \mathcal{F}_1), (\mathcal{F}_1, \mathcal{F}_3), (\mathcal{F}_2, \mathcal{F}_2), (\mathcal{F}_3, \mathcal{F}_1), (\mathcal{F}_1, \mathcal{F}_4), \ldots$$

For each pair, we check to see if the number of parameters of the two functions works out to form a valid function defined by recursion. If not, we add nothing to the grand list and continue in our cycle of the four lists. If we do have a valid pair, we add the resulting function defined by recursion to the grand list.

Finally, the trickiest list to construct: we need to generate a linear list of all functions defined by substitution. We will use the same general approach that we did for the recursively defined functions, but this time we have to look at strings of previously-defined primitively recursive functions.

This is because to make a valid substitution, we need a function f of k variables followed by k functions $g_i, 1 \leq i \leq k$, each with the same number l of variables.

What we will do is to generate all possible finite strings of function names from the grand list and each time we have the opportunity to add a function defined by substitution, we will look at the next string in the set. The first function in the string will have some number k of variables. If the list does not contain $k + 1$ items, it is invalid and we skip it. If it does contain $k + 1$ items, then we examine the last k of them to see if each has the same number l of parameters. If that is also true, then we can generate a valid function for the grand list defined by substitution; if it's invalid, we skip the list and continue our grand cycle of the primitive recursive function lists.

OK, how do we make a list of all strings of functions that includes duplications and eventually lists all finite strings from an infinite "alphabet"?

3.5.2 Gödel Numbering

To make the problem precise, our "alphabet" will be the list of primitive recursive functions on our grand list: $\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \ldots$.

We want to output a list of all finite strings of the $\mathcal{F}_i$ where each string is assigned to a unique natural number. The list should include all of the following in some order:

$$\begin{array}{l}
\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3, \mathcal{F}_4, \ldots \\
\mathcal{F}_0\mathcal{F}_0, \mathcal{F}_0\mathcal{F}_1, \mathcal{F}_1\mathcal{F}_0, \mathcal{F}_1\mathcal{F}_1, \mathcal{F}_0\mathcal{F}_2, \ldots \\
\mathcal{F}_0\mathcal{F}_0\mathcal{F}_0, \mathcal{F}_0\mathcal{F}_0\mathcal{F}_1, \mathcal{F}_0\mathcal{F}_1\mathcal{F}_0, \mathcal{F}_0\mathcal{F}_1\mathcal{F}_1, \mathcal{F}_1\mathcal{F}_0\mathcal{F}_0, \ldots
\end{array}$$

$$\mathcal{F}_0\mathcal{F}_0\mathcal{F}_0\mathcal{F}_0, \mathcal{F}_0\mathcal{F}_0\mathcal{F}_0\mathcal{F}_1, \mathcal{F}_0\mathcal{F}_0\mathcal{F}_1\mathcal{F}_0, \mathcal{F}_0\mathcal{F}_0\mathcal{F}_1\mathcal{F}_1, \ldots$$
$$\ldots$$

The trick we will use, called "Gödel numbering", is based on the fact that every natural number has a unique prime factorization. First, list the prime numbers as follows: $p_0 = 2, p_1 = 3, p_2 = 5, p_3 = 7, p_4 = 11, p_5 = 13$, and so on.

Every natural number n can be written as the following infinite product:

$$n = \prod_{i=0}^{\infty} p_i^{k_i} = p_0^{k_0} p_1^{k_1} p_2^{k_2} p_3^{k_3} p_4^{k_4} \cdots.$$

In every such representation, all but a finite number of the k_i are equal to zero.

An easy way to encode a string is using the exponents of the prime factors of a natural number. For example, suppose we wish to encode the string:

$$\mathcal{F}_{i_0}\mathcal{F}_{i_1}\mathcal{F}_{i_2}\cdots\mathcal{F}_{i_k}$$

The following number does the trick:

$$p_0^{i_0+1} p_1^{i_1+1} p_2^{i_2+1} \cdots p_k^{i_k+1}.$$

We add one to each exponent, since there will be an infinte number of exponents equal to zero in the prime factorization of any number.

This will assign a unique number to every string, but there will be some numbers that do not correspond to any string.

Here are the first dozen valid numbers, their prime factorizations, and the corresponding strings:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
2 & = & p_1^1 \longrightarrow \mathcal{F}_0 & 18 & = & p_1^1 p_2^2 \longrightarrow \mathcal{F}_0\mathcal{F}_1 \\
4 & = & p_1^2 \longrightarrow \mathcal{F}_1 & 24 & = & p_1^3 p_2^1 \longrightarrow \mathcal{F}_2\mathcal{F}_0 \\
6 & = & p_1^1 p_2^1 \longrightarrow \mathcal{F}_0\mathcal{F}_0 & 30 & = & p_1^1 p_2^1 p_3^1 \longrightarrow \mathcal{F}_0\mathcal{F}_0\mathcal{F}_0 \\
8 & = & p_1^3 \longrightarrow \mathcal{F}_2 & 32 & = & p_1^5 \longrightarrow \mathcal{F}_4 \\
12 & = & p_1^2 p_2^1 \longrightarrow \mathcal{F}_1\mathcal{F}_0 & 36 & = & p_1^2 p_2^2 \longrightarrow \mathcal{F}_1\mathcal{F}_1 \\
16 & = & p_1^4 \longrightarrow \mathcal{F}_3 & 48 & = & p_1^4 p_2^1 \longrightarrow \mathcal{F}_3\mathcal{F}_0
\end{array}$$

There is no unique method of Gödel numbering—the method above is just an example. The term "Gödel numbering" just refers to the idea of encoding strings of information of arbitrary length using the fact that natural numbers can be factored uniquely into prime numbers.

3.5.3 The Fibonacci sequence revisited

One of the first examples of a recursive function that we presented in section 2.2.2 was that of the Fibonacci numbers. It seems that it should be trivial to demonstrate that this sequence is primitive recursive (and it is) but it turns out to be surprisingly complicated to show.

In fact, the usual way to do so uses something akin to Gödel numbering. The construction method for recursion in defining primitive recursive functions only allows the use of $f(n-1)$ in the calculation of $f(n)$. Since the basic recursive definition of Fibonacci numbers is: $f(n) = f(n-1) + f(n-2)$, there is no simple way to access $f(n-2)$ during the calculation of $f(n)$.

The easiest way to get around this problem is to define a set of helper functions that encodes the previous two values as exponents of primes. The number that is passed along looks like this: $2^{f(n-1)}3^{f(n-2)}$. This can be "decoded" by primitive recursive functions, then the appropriate calculation is performed, and the result is re-encoded in the same form. The actual function that generates the numbers in the Fibonacci sequence executes the recursion as described above and then decodes and returns the exponent of 2.

In a similar way, this method can be used to define primitive recursive functions that make use of any number of previous values. In fact, an arbitrary set of such values can be stored as the exponents of prime numbers in longer and longer Gödel numbered encodings of those sequences of values.

3.6 General Recursive Functions

In this section we will introduce one additional construction mechanism to those we already have for the primitve recursive functions and the resulting set of functions will be, in a sense, complete—we will not be able to use diagonalization arguments to produce addtional functions that are clearly effectively computable.

The set of general recursive functions includes all of the primitive recursive functions, but provides one additional construction operation:

6. **Unbounded search function.** If $f(n, n_1, \ldots, n_k)$ is in the set, then so is the function $\mu_f(n_1, \ldots, n_k)$ that returns the smallest n such that $f(i, n_1, \ldots, n_k)$ is defined for $i < n$ and $f(n, n_1, \ldots, n_k) = 0$. If no such n exists, then $\mu_f(n_1, \ldots, n_k)$ itself is not defined.

General recursive functions are not always total functions. For any particular function f, there may not be any combination of input values that cause it to evaluate to zero. Some general recursive functions that are not primitve recursive functions are total and some are not. We can speak about the set of total general recursive functions, although, as we shall show, it is impossible to determine in general for any specific function, whether it is total or not.

There are total general recursive functions that are not primitive recursive. Perhaps the most well-known example is Ackermann's function, $\mathbb{A}(x, y, z)$, defined as follows:

$$\begin{aligned}
\mathbb{A}(0,0,y) &= y \\
\mathbb{A}(0,x+1,y) &= \mathbb{A}(0,x,y)+1 \\
\mathbb{A}(1,0,y) &= 0 \\
\mathbb{A}(z+2,0,y) &= 1 \\
\mathbb{A}(z+1,x+1,y) &= \mathbb{A}(z,\mathbb{A}(z+1,x,y),y)
\end{aligned}$$

This amounts to:

$$\begin{aligned}
\mathbb{A}(0,x,y) &= x+y \\
\mathbb{A}(1,x,y) &= x\cdot y \\
\mathbb{A}(2,x,y) &= x^y \\
\mathbb{A}(3,x,y) &= \left. x^{x^{\cdot^{\cdot^{\cdot^{x}}}}} \right\} \quad (y \text{ copies})
\end{aligned}$$

and so on. At each successive stage the previous operation is applied y times to x.

The standard proof that this function is not primitive recursive shows that Ackermann's function increases more rapidly than any primitive recursive function can. It is also clear from the definiton that Ackermann's function is total.

We will not provide a formal proof of that here, but the basic reasoning goes as follows. All of the primitive recursive functions are built upon the successor function $\mathcal{S}(n)$. This is the only function that can increase an input value.

In each round of primitive recursion, we can only take a function that has been previously applied, and apply it repeatedly. Thus in one stage we can get to addition, then, in the next stage, we can get to multiplication, then to exponentiation, then to towers of exponents. But at any fixed stage, there is an upper limit to how fast the outputs can increase relative to the inputs. In Ackermann's function, there is no limit. It is a sort of a diagonalization process applied to the fastest-growing primitive recursive functions available at each stage.

In other words, if Ackermann's function were primitive recursive, it would appear at stage n above the successor function for some n, but $\mathbb{A}(n+1)$ is much bigger than it could be for a function at stage n.

It turns out that an intuitive way to look at primitive recursive and general recursive functions in terms of modern computer languages is that primitive recursive functions can all be coded up using only "for" loops; general recursive functions require "while" loops. The "for" loops are guaranteed to terminate (assuming that the index value is left unchanged inside the loop), but "while" loops are not.

4 Computability and General Recursive Functions

The huge difference between primitive recursive and general recrusive functions is that when we begin a general recursive calculation, we do not know if it will terminate. Thus it is possible that for some input values to a general recursive function, the applications of substitutions may never end.

When the function has the form $\mu_f(n_1, \ldots, n_k)$, there may be no way to tell whether there are *any* values of n such that $f(n, n_1, \ldots n_k) = 0$. One can, of course, begin to plug values of n into the function to test it, but even if you've tested a million input values with no success, you still have tested zero percent of the possible inputs. And worse, your tests may never terminate.

We are basically faced with the following situation: If $\mu_f(n_1, \ldots, n_k)$ exists, we can find its value in finite time. If not, we may not even be able to prove, in any finite amount of time, that it does not exist.

This may seem like a terrible situation in which to find ourselves, but it is really not as bad as it seems. Exactly the same situation holds when you run a computer program on a given input. Some programs can be proved always to halt with an output, but for a general program, it is impossible to know. Just because there are some bad programs does not mean that we avoid them altogether, and just because there are "bad" general recursive functions does not mean that general recursive functions should be avoided altogether.

5 Effective Computations

Perhaps the easiest way to indicate what is meant by an effective computation is to illustrate a function for which no effective computation is known.

Every natural number larger than 1 can be written as the sum of prime numbers, often in many different ways. For example: $2 = 2, 3 = 3, 4 = 2+2, 5 = 2+3, 6 = 3+3 = 2+2+2, 7 = 7 = 2+2+3$, et cetera. Let $f(n)$ be the smallest natural number that requires at least $n+1$ primes in such a summation. If every number can be expressed as a sum of fewer than $n+1$ primes, then $f(n) = 0$.

Thus $f(0) = 2$ since $2 = 2$ requires a single prime. We have $f(1) = 4$, since $4 = 2+2$ and 4 is not prime, so at least two primes must add to give 4. Similarly, $f(2) = 27$. What is $f(3)$? Nobody knows (at least at the time this was written). If the Goldbach conjecture is true, then $f(3) = f(4) = \cdots = 0$. The Goldbach conjecture states that *every* integer greater than 1 can be expressed as the sum of three or fewer primes. The Goldbach conjecture has been tested for all integers smaller than 6×10^{16} at the time of this writing.

It is easy to imagine writing a program that attempts to calculate $f(n)$ as follows:

For each m, look at all possible sums of sets of n or fewer prime numbers less than m and see if they sum to m. If not, return m; otherwise repeat this operation with the next larger value of m.

The problem is that if Goldbach's conjecture is true, the program will never terminate for $n > 2$. This is hardly an effective computation method, since although you may have waited a million years for the program to terminate and it is still running, that does not mean that it might not return with the answer in the next second.

6 Most Functions Are *Not* Effectively Computable

It may seem surprising at first, but *almost all* functions that map $\mathbb{N} \rightarrow \mathbb{N}$ are not effectively computable. This is because there are only a countable number of computer programs of finite length and there are an uncountable number of functions mapping $\mathbb{N} \rightarrow \mathbb{N}$.

One way to see that is that every computer program can be written as a finite sequence of characters. If we assume, say, that there are 100 valid characters that can be used in any program, then there are at most 100^n sequences of characters of length n, and only a tiny number of those sequences will be valid computer programs. Thus the total number of valid programs of length n characters or less is smaller than:

$$100^1 + 100^2 + 100^3 + \cdots + 100^n,$$

which is just a (rather large but finite) natural number.

If we let S_n be the set of all valid programs of length n, then the set S of valid programs is just

$$S = \bigcup_{n=1}^{\infty} S_n.$$

Since S is a countable union of countable sets, it is clearly countable.

The number of functions mapping $\mathbb{N}$ to itself is, however, uncountable. This can easily be shown with a variant of Cantor's diagonalization argument. Suppose that there were only a countable number of such functions. Then we could enumerate them as $f_0(n)$, $f_1(n)$, $f_2(n)$, et cetera. Now define a new function $g(n)$ as follows:

$$g(n) = f_n(n) + 1.$$

We know that for every i, g cannot be equal to f_i, since they differ when evaluated at i: $g(i) = f_i(i) + 1 \neq f_i(i)$. Thus the set of functions cannot be enumerated with the natural numbers, so it must be uncountable.

Thus, *almost every* function f mapping $\mathbb{N}$ to $\mathbb{N}$ is *not* effectively computable.

7 Interesting Functions

$$f_1(x) = \begin{cases} 1 & : \text{ if a run of exactly } x \text{ 5's occurs in the decimal expansion of } \pi \\ 0 & : \text{ otherwise} \end{cases}$$

$$f_2(x) = \begin{cases} 1 & : \text{ if a run of at least } x \text{ 5's occurs in the decimal expansion of } \pi \\ 0 & : \text{ otherwise} \end{cases}$$

$$f_3(x) = \begin{cases} 1 & : \text{ if the number } x \text{ occurs in the decimal expansion of } \pi \\ 0 & : \text{ otherwise} \end{cases}$$

$$f_4(x) = \begin{cases} 1 & : \text{ if Goldbach's conjecture is true} \\ 0 & : \text{ otherwise} \end{cases}$$

Since $\pi = 3.1415926535897\ldots$, we know that $f_3(1) = f_3(141) = f_3(159265) = 1$. It is unknown whether there exists an x such that $f_3(x) = 0$.