7 Recursive Functions

In this chapter, we define the class of recursive functions as the functions that can be built from certain base functions using some simple operations. We show that a wide range of functions are recursive, and that the recursive functions are exactly the Turing-computable functions.

7.1 Primitive recursive functions

In Chapter 5, I said that a function is *computable* if there are precise instructions, an *algorithm*, for determining its output for any given input, without drawing on external resources or creativity. Let's concentrate on functions with natural numbers as inputs and outputs. (I've explained in Section 5.5 why this is not a serious restriction.)

Consider the "counting on" algorithm for addition that you may have learned as a child. To compute $x + y$, you start with the number x; then you "count on" from x, adding 1 y times. The algorithm effectively reduces addition to repeated application of the successor function. Simple algorithms for multiplication similarly reduce multiplication to repeated addition.

Generalizing, algorithms for computing arithmetical functions usually invoke subroutines for computing simpler functions. These subroutines may in turn invoke other subroutines, until we reach functions that are so simple that they can be computed in a single step, without any subroutines. This suggests that the computable functions might be defined as the functions that can be built up from certain base functions using certain modes of construction. This is how Gödel defined the class of *primitive recursive* functions in his 1931 paper on the incompleteness theorems.

Following Gödel, we start with three kinds of base functions.

1. The *successor function* s returns the next larger number for any input number.
2. The *zero function* z that returns 0 for any input number.
3. For each n, i, the *projection function* π_i^n takes n numbers as input and return the i-th of them. (For example $\pi_2^3(5, 9, 2) = 9$.)

These functions are trivially computable, without needing any subroutines.

We now define two operations for constructing new functions from old ones. The first is *composition*. Given some functions f and g, we can define a new function h by applying one to the output of the other:

$$h(x) = f(g(x)).$$

If f and g are computable, then h is also computable. To compute $h(x)$, we only need to compute $g(x)$ and feed the output into f.

I've assumed that f and g both take one number as input. For the general case, assume that f is a function of m arguments, and each of $g_1, \ldots, g_m$ is a function of n arguments. We define the *composition* of f and $g_1, \ldots, g_m$ as:

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

Instead of introducing a new name 'h' for the composed function, we can also write the composition as $\mathrm{Cn}[f, g_1, \ldots, g_m]$. For example, $\mathrm{Cn}[s, z]$ is the function that takes a number as input, passes it to the zero function and passes the output to the successor function: $\mathrm{Cn}[s, z](x) = s(z(x))$. This is the constant function that always outputs 1.

Exercise 7.1 What is (a) $\mathrm{Cn}[s, s]$? (b) $\mathrm{Cn}[s, \mathrm{Cn}[s, z]]$? (c) $\mathrm{Cn}[\pi_2^2, z, z]$?

Exercise 7.2 Using Cn and the base functions, define the 1-place function that always returns 4.

Our second method for constructing functions is primitive recursion. We've met this in Section 4.1 when we talked about axioms of arithmetic. Addition, for example, can be reduced to the successor function by the following definition:

$$x + 0 = x$$
$$x + s(y) = s(x + y)$$

The definition effectively tells us how to compute $x + y$ starting from $x + 0$, then working our way up through $x + 1$, $x + 2$, and so on, until we reach $x + y$. (The first line gives us $x + 0$; the second line tells us how to get from $x + y$ to $x + s(y)$.) In imperative pseudocode, the algorithm, which resembles the counting-on strategy, could be stated as follows:

```
function add(x, y):
    let z = x
    for i from 1 to y:
        z = s(z)
    return z
```

Some more examples. First, the *factorial function* that takes a number n as input and returns the product of all numbers from 1 to n:

$$\begin{aligned} \text{fact}(0) &= 1 \\ \text{fact}(s(y)) &= s(y) \cdot \text{fact}(y) \end{aligned}$$

Next, we can define the *truncated predecessor* function that maps any positive number to its predecessor, and 0 to 0:

$$\begin{aligned} \text{pred}(0) &= 0 \\ \text{pred}(s(y)) &= y \end{aligned}$$

This is a somewhat unusual case of primitive recursion because the value of pred for $s(y)$ doesn't depend on the value of pred for y; it only depends on y itself. But we allow for that.

Using pred, we can define a *truncated subtraction* function:

$$\begin{aligned} x \dot{-} 0 &= x \\ x \dot{-} s(y) &= \text{pred}(x \dot{-} y) \end{aligned}$$

$x \dot{-} y$ is $x - y$ if $x \geq y$ and 0 otherwise.

Another useful function is the *switcheroo* function δ ("delta") that takes every positive integer to 0 and 0 to 1:

$$\begin{aligned} \delta(0) &= 1 \\ \delta(s(y)) &= 0 \end{aligned}$$

Here, the value for $s(y)$ depends on neither the value for y nor on y itself. That, too, is allowed.

Exercise 7.3 Define multiplication and exponentiation using primitive recursion.

Exercise 7.4 Use primitive recursion to define a function h that maps 0 to 0 and every positive number to 1.

Let's officially define the operation of primitive recursion. We assume that an $n+1$-place function h is defined by primitive recursion from an n-place function f and an $n+2$-place function g. The function f specifies the starting point, $h(x_1, \dots, x_n, 0)$; The function g specifies $h(x_1, \dots, x_n, s(y))$ in terms of $h(x_1, \dots, x_n, y)$. We allow g to also depend on $x_1, \dots, x_n$, and y. So our general format for primitive recursion looks like this:

$$\begin{aligned} h(x_1, \dots, x_n, 0) &= f(x_1, \dots, x_n) \\ h(x_1, \dots, x_n, s(y)) &= g(x_1, \dots, x_n, y, h(x_1, \dots, x_n, y)). \end{aligned}$$

Again, this effectively defines an algorithm for computing h:

```
function h(x1, ..., xn, y):
    let z = f(x1, ..., xn)
    for i from 0 to y-1:
        z = g(x1, ..., xn, i, z)
    return z
```

If h is defined by primitive recursion from f and g, we write $h = \mathrm{Pr}[f, g]$. For example, addition is $\mathrm{Pr}[\pi^1_1, \mathrm{Cn}[s, \pi^3_3]]$. That's because

$$\begin{aligned} x + 0 &= \pi^1_1(x) = x \\ x + s(y) &= \mathrm{Cn}[s, \pi^3_3](x, y, x + y) = s(x + y). \end{aligned}$$

Exercise 7.5 Can you express your definition of multiplication using the Pr notation?

A downside of the above format is that it doesn't directly account for our definitions of pred, fact, and δ, which are one-place functions. Recall the definition of pred:

$$\begin{aligned} \mathrm{pred}(0) &= 0 \\ \mathrm{pred}(s(y)) &= y \end{aligned}$$

Here there are no extra arguments $x_1, \dots, x_n$. So the function f would be a zero-place "function" that returns 0. We don't have such a function, and we can't define it using our present resources.

But there's a workaround. We can first define a two-place function dpred with a dummy first argument:

$$\begin{aligned} \text{dpred}(x, 0) &= 0 \\ \text{dpred}(x, s(y)) &= y \end{aligned}$$

This fits our official format: $\text{dpred} = \text{Pr}[z, \pi^3_2]$. Since $\text{pred}(y) = \text{dpred}(x, y)$ for any x, we can now define pred by composition and projection from dpred:

$$\text{pred} = \text{Cn}[\text{dpred}, \pi^1_1, \pi^1_1]$$

This trick also works for fact and δ, and any other case of "one-place primitive recursion".

We now define the class of primitive recursive functions.

Definition 7.1

A function is *primitive recursive* if it can be defined from the base functions s, z, and π^n_i by finitely many applications of composition and primitive recursion.

Proposition 7.1

Every primitive recursive function is total.

Proof. By induction on a function's construction. The base functions are evidently total. The composition of total functions is total. For primitive recursion, note that the algorithm implicitly defined by Pr always terminates after y steps when computing $\text{Pr}[f, g](x_1, \ldots, x_n, y)$, returning the desired output. □

We can extend the concept of primitive recursiveness to sets and relations:

Definition 7.2

A set is primitive recursive if its characteristic function is primitive recursive. A relation is primitive recursive if its extension is primitive recursive.

Remember that the characteristic function of a set is the function that maps every member of the set to 1 and every non-member to 0. At the moment, we are interested in

sets whose members are numbers or tuples of numbers. For example, the set of odd numbers is primitive recursive, as its characteristic function χ_{odd} can be defined by primitive recursion, as follows:

$$\begin{aligned} \chi_{\text{odd}}(0) &= 0 \\ \chi_{\text{odd}}(s(y)) &= \delta(\chi_{\text{odd}}(y)) \end{aligned}$$

Definition 7.2 also covers properties, because a property is a 1-place relation. A property of numbers is primitive recursive iff there is a primitive recursive function that maps every number that has the property to 1 and every other number to 0. As we've just shown, the property of being odd is primitive recursive.

An example of a two-place primitive recursive relation is the less-than-or-equal relation on $\mathbb{N}$. Its characteristic function can be defined by composition from $\dot{-}$ and δ:

$$\text{Leq}(x, y) \;=\; \delta(x \dot{-} y).$$

Another example is the identity relation on $\mathbb{N}$, with the following characteristic function:

$$\text{Eq}(x, y) \;=\; \delta\big((x \dot{-} y) + (y \dot{-} x)\big).$$

To see why this works, note that if $x = y$ then $x \dot{-} y$ and $y \dot{-} x$ are both 0; if $x \neq y$ then at least one of them is positive.

Exercise 7.6 Show that the set of even numbers is primitive recursive.

Exercise 7.7 Show that the less-than relation on $\mathbb{N}$ is primitive recursive. You may, if you want, use the functions Leq and Eq.

Exercise 7.8 Show that if a relation R is primitive recursive then so is its negation $\neg R$ (which holds of exactly those tuples that do not satisfy R).

Exercise 7.9 Show that if two relations R and S are primitive recursive then so is their conjunction $R \wedge S$ (which holds of exactly those tuples that satisfy both R and S).

7.2 Primitive recursive operations

In the last two exercises, you showed that the primitive recursive relations are closed under negation and conjunction. Since all truth-functional combinations can be built up from these two operations, it follows that the primitive recursive relations are closed under all truth-functional operations.

Proposition 7.2

If R and S are primitive recursive relations, then so are $\neg R$, $R \wedge S$, $R \vee S$, $R \to S$, and $R \leftrightarrow S$.

Proof. Immediate from exercises 7.8 and 7.9. □

I'll define three more operations for defining primitive recursive functions and relations.

First, bounded quantification. Consider the Divides relation that holds between numbers x and y iff y is a multiple of x. (For example, 3 divides 12, but 3 doesn't divide 5.) We might define this as follows:

$$\text{Divides}(x, y) \text{ iff } \exists z\, (y = x \cdot z).$$

But we don't need to quantify over all numbers z. If x and y are natural numbers, $y = x \cdot z$ can only hold if z is less than or equal to y. So we can also use a *bounded quantifier* in the definition:

$$\text{Divides}(x, y) \text{ iff } \exists z \leq y\, (y = x \cdot z).$$

This says that x divides y iff there is some number z less than or equal to y such that y equals x times z. Since multiplication and equality are primitive recursive, this relation is primitive recursive:

Proposition 7.3

If $R(x_1, \ldots, x_n, y)$ is a primitive recursive relation, then so are $\forall y \leq k\, R(x_1, \ldots, x_n, y)$ and $\exists y \leq k\, R(x_1, \ldots, x_n, y)$.

Proof. To simplify notation, I assume that R is a two-place relation $R(x, y)$. Let χ be

the characteristic function of R. Define χ' by primitive recursion as follows:

$$\begin{aligned}\chi'(x,0) &= \chi(x,0)\\ \chi'(x,s(k)) &= \chi'(x,k)\cdot\chi(x,s(k))\end{aligned}$$

This function returns 1 for input x,k iff $R(x,0), R(x,1), \dots, R(x,k)$ all hold, otherwise it returns 0. So $\chi'(x,k)$ is the characteristic function of $\forall y \leq k\, R(x,y)$.

From bounded universal quantification, we obtain bounded existential quantification by truth-functional operations: $\exists y \leq k\, R(x,y)$ is equivalent to $\neg\, \forall y \leq k\, \neg R(x,y)$. □

So Divides is primitive recursive. The same is true for the property of being a prime number, as the following definition shows:

$$\text{Prime}(x) \text{ iff } 1 < x \wedge \forall y \leq x\, (\text{Divides}(y,x) \to (y = 1 \vee y = x)).$$

Don't get confused by the fact that this looks vaguely like a formula of first-order logic. We're not trying to define Prime in some first-order language. Everything is in the metalanguage. 'Divides' and '=' are metalinguistic names for relations on $\mathbb{N}$ that we've shown to be primitive recursive; '$\wedge$', '$\to$', '$\vee$', and '$\forall y \leq x$' denote operations on such relations.

Next, definition by cases. Suppose we want to define the function $\max(x,y)$ that returns the larger of two numbers x and y:

$$\max(x,y) = \begin{cases} x & \text{if } y \leq x,\\ y & \text{otherwise.}\end{cases}$$

We can use switcheroo and addition to distinguish the two cases:

$$\begin{aligned}\max(x,y) &= x\cdot \text{Leq}(y,x) + y\cdot\delta(\text{Leq}(y,x))\\ &= x\cdot\delta(y \dot{-} x) + y\cdot\delta(\delta(y \dot{-} x)).\end{aligned}$$

This trick can be generalized:

Proposition 7.4

If f and g are primitive recursive functions and R is a primitive recursive relation

then the function h defined by

$$h(x_1, \ldots, x_n) = \begin{cases} f(x_1, \ldots, x_n) & \text{if } R(x_1, \ldots, x_n), \\ g(x_1, \ldots, x_n) & \text{otherwise} \end{cases}$$

is primitive recursive.

Proof. Let χ be the characteristic function of R. Then

$$f(x_1, \ldots, x_n) \cdot \chi(x_1, \ldots, x_n) + g(x_1, \ldots, x_n) \cdot \delta(\chi(x_1, \ldots, x_n))$$

is primitive recursive and defines h.

Exercise 7.10 Suppose h is defined by distinguishing three cases:

$$h(x) = \begin{cases} f(x) & \text{if } P_1(x), \\ g_1(x) & \text{if } P_2(x), \\ g_2(x) & \text{otherwise,} \end{cases}$$

where f, g_1, and g_2 are primitive recursive functions. Explain why it follows from proposition 7.4 that h is primitive recursive

The final operation I want to mention is bounded minimization. Suppose we want to define the function $\text{spf}(x)$ that returns the smallest prime number y that divides x. We can write this as follows:

$$\text{spf}(x) = \mu y \, (\text{Prime}(y) \wedge \text{Divides}(y, x)),$$

where 'μy' ("mu y") means "the least y such that …". So $\mu y \, (\text{Prime}(y) \wedge \text{Divides}(y, x))$ is the least number y such that y is prime and y divides x. That number y will never be greater than x. So we can equivalently define $\text{spf}(x)$ as the least number y less than or equal to x that is prime and divides x:

$$\text{spf}(x) = \mu y \leq x \, (\text{Prime}(y) \wedge \text{Divides}(y, x)).$$

The $\mu y \leq x$ operator expresses a bounded search. To compute $\mu y \leq x \, R(y)$, we check $R(0)$, then $R(1)$, then $R(2)$, and so on, up to $R(x)$, until we find a number y of which $R(y)$

holds; then we return that number. To ensure that the operation defines a total function, let's stipulate that $\mu y \leq x\, R(y)$ is 0 if there is no number $y \leq x$ for which $R(y)$ holds.

Exercise 7.11 Let $h(x) = \mu y \leq x\, (y + y = x)$. What are $h(0)$, $h(1)$, and $h(2)$?

Proposition 7.5

If $R(x_1, \ldots, x_n, y)$ is a primitive recursive relation, then the function f defined by

$$f(x) = \mu y \leq x\, R(x_1, \ldots, x_n, y)$$

is primitive recursive.

Proof. I'll assume that $n = 1$, to simplify the notation. Let χ be the characteristic function of $R(x, y)$. Let f be defined as follows:

$$f(x, 0) = \begin{cases} 0 & \text{if } \chi(x, 0) = 1, \\ 1 & \text{otherwise.} \end{cases}$$

$$f(x, s(k)) = \begin{cases} f(x, k) & \text{if } f(x, k) \leq k, \\ k + 1 & \text{if } \chi(x, k + 1) = 1, \\ k + 2 & \text{otherwise.} \end{cases}$$

Think of this as defining a sequence $f(x, 0), f(x, 1), f(x, 2), \ldots$. At each step $k = 0, 1, 2, \ldots$, $f(x, k)$ is the first y with $R(x, y)$ if we've already found one, otherwise it is $k + 1$. So $f(x, k)$ is the least number $y \leq k$ such that $R(x, y)$ holds, or $k + 1$ if there is no such number. Finally,

$$\mu y \leq x\, R(x, y) = \begin{cases} f(x, x) & \text{if } f(x, x) \leq x, \\ 0 & \text{otherwise.} \end{cases}$$

□

These operations give us a useful toolkit for defining primitive recursive functions and relations. I'll give three examples, related to coding sequences of (non-zero) numbers by prime powers, as introduced in section 5.5.

First, consider the function pri that takes a number n as input and returns the n-th

prime number (counting from zero). This function is primitive recursive:

$$\begin{aligned} \mathrm{pri}(0) &= 2, \\ \mathrm{pri}(s(y)) &= \mu x \leq 2 \cdot \mathrm{pri}(y)\ (\mathrm{Prime}(x) \wedge x > \mathrm{pri}(y)). \end{aligned}$$

Here I use Chebyshev's theorem, which states that for any $n \geq 1$ there is a prime between n and $2n$.

Second, we can define a primitive recursive function $\mathrm{entry}(x, y)$ that returns the exponent of the y-th prime in the prime factorization of x:

$$\mathrm{entry}(x, y) = \mu z \leq x\ (\mathrm{Divides}(\mathrm{pri}(y)^z, x) \wedge \neg \mathrm{Divides}(\mathrm{pri}(y)^{z+1}, x)).$$

I call this 'entry' because it returns the y-th entry in the sequence coded by x when we use Gödel's scheme to code a sequence of numbers $n_1, n_2, \ldots, n_k$ as $2^{n_1} \cdot 3^{n_2} \cdots p_k^{n_k}$. For example, $\mathrm{entry}(2^3 \cdot 3^2 \cdot 5^6 \cdot 7^4, 3) = 6$.

Finally, we can define a primitive recursive function $\mathrm{len}(x)$ that returns the length of the sequence coded by x:

$$\mathrm{len}(x) = \mu y \leq x\ \forall z \leq x\ (z \geq y \to \mathrm{entry}(x, z) = 0).$$

Thus $\mathrm{len}(2^3 \cdot 3^2 \cdot 5^6 \cdot 7^4) = 4$.

7.3 Unbounded search

Any arithmetical function you can think of is almost certainly primitive recursive. But not all computable functions on the natural numbers are primitive recursive. A concrete counterexample is the Goodstein function.

To explain this function, I need the fact that any number x can be expressed as a sum of powers of n, for any choice of $n > 1$. For example, choosing $n = 2$, we can express 266 as $2^8 + 2^3 + 2^1$. Here, the exponents are 8, 3, and 1. If we write these as powers of 2 as well, we get the "hereditary base-2 representation" of 266:

$$266 = 2^{2^{2+1}} + 2^{2+1} + 2^1.$$

Starting with any number n, we can now define a sequence of numbers, called the *Goodstein sequence for n*. The first item in the sequence is n. For the second item, we replace each 2 in the hereditary base-2 representation of n by 3, and subtract 1. So the second

item in the Goodstein sequence for 266 is

$$3^{3^{3+1}} + 3^{3+1} + 3^1 - 1 = 7,625,597,484,987.$$

For the third item, we replace each 3 in the hereditary base-3 representation of the second item by 4, and subtract 1. And so on. While Goodstein sequences initially grow large very quickly, Reuben Goodstein proved that their growth eventually stalls and reverses, until it reaches 0. (This is *Goodstein's Theorem*.) The *Goodstein function* now maps any number n to the length of the Goodstein sequence for n before it reaches 0. It can be shown that this function isn't primitive recursive. But it is clearly computable: from each item in the sequence, one can mechanically compute the next item. To compute the Goodstein function for n, we therefore simply need to compute all items in the sequence, one by one, until we reach 0, keeping count of how many items we've computed.

For another example of a computable function that isn't primitive recursive, note that we can effectively enumerate the primitive recursive functions: we start with the base functions, then we list all functions that can be obtained from these by one application of composition or primitive recursion, followed by all functions that require two applications of these operations, and so on. Let $f_1, f_2, f_3, \ldots$ be this enumeration, but omitting any functions with more than one argument. We can now define an antidiagonal function d by setting

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

Since all primitive recursive functions are total, this function is well-defined. It is evidently computable. But it can't be primitive recursive, since it differs from each primitive recursive function f_n at input n.

How would we compute $d(n)$? We would first identify the n-th one-place primitive recursive function f_n. Then we would compute $f_n(n)$ until we get the output, to which we would add 1. Like the computation of the Goodstein function, this computation involves an unbounded loop: we simply have to wait until $f_n(n)$ returns an output; we can't tell in advance how long this will take.

If we want to capture all computable functions, we need to add an operation that allows for this kind of unbounded search. This operation will search through all numbers $0, 1, 2, \ldots$ until it finds a number x for which a given condition $P(x)$ is satisfied.

We've already met such an operation above, in the form of the unbounded minimization operation μ: $\mu x\, P(x)$ is the least number x for which $P(x)$ holds. Given a two-place function $f(x, y)$, we can use μ to define a 1-place function h that maps any number x to

the least number y for which $f(x, y)$ equals a desired value k:

$$h(x) = \mu y\,(f(x, y) = k).$$

Without loss of generality, we can assume that the desired value is always 0: if we want to find the least y such that $f(x, y) = k$, we can equivalently look for the least y such that $g(x, y) = 0$ where $g(x, y)$ is defined as $f(x, y) \dot{-} k$.

So assume that f is a total function of $n + 1$ arguments. (We'll deal with non-total functions in a moment.) Then the n-place function h defined by

$$h(x_1, \ldots, x_n) = \mu x f(x_1, \ldots, x_n, x) = 0$$

is called the *minimization* of f. We write $h = \mathrm{Mn}[f]$.

If f is computable then so is $\mathrm{Mn}[f]$. We simply need to compute $f(x_1, \ldots, x_n, i)$ for each $i = 0, 1, 2, \ldots$ until we find an i for which $f(x_1, \ldots, x_n, i) = 0$:

```
function Mn_f(x1, ..., xn):
    let i = 0
    while f(x1, ..., xn, i) != 0:
        i = i + 1
    return i
```

If there is no i for which $f(x_1, \ldots, x_n, i) = 0$, this algorithm runs forever. Thus $\mathrm{Mn}[f]$ may fail to be total, even if f is total. For example, $\mathrm{Mn}[+]$, the minimization of the addition function, returns 0 for input 0, but is undefined for every other input: if $x > 0$, there is no y such that $x + y = 0$.

A function $f(x_1, \ldots, x_n, y)$ is called *regular* if it is total and for all $x_1, \ldots, x_n$ there is some y such that $f(x_1, \ldots, x_n, y) = 0$. When minimization is applied to a regular function f, the result is always total. In that case, we say that $\mathrm{Mn}[f]$ is defined by *regular minimization* from f.

So far, I've assumed that Mn is applied to a total function. We can also apply minimization to non-total functions. But we need a further constraint to ensure that $\mathrm{Mn}[f]$ is computable. Suppose $f(x, y)$ is 0 for some x, y, and undefined for the same x and some $z < y$. Then we may not be able to effectively search for the least y with $f(x, y) = 0$ by checking $f(x, 0), f(x, 1), f(x, 2), \ldots$: if the computation of, say, $f(x, 2)$ never halts, the search doesn't proceed beyond 2. We therefore stipulate that if f is an $n + 1$-place function, then $\mathrm{Mn}[f]$ is the function that takes n numbers $x_1, \ldots, x_n$ as input and returns the least number y for which

(i) $f(x_1, \ldots, x_n, y) = 0$, and
(ii) $f(x_1, \ldots, x_n, z)$ is defined for all $z < y$.

If there is no such y, $\mathrm{Mn}[f](x_1, \ldots, x_n)$ is undefined.

Exercise 7.12 Let $f(x, y) = x \cdot y$. What is $\mathrm{Mn}[f]$? Is it total? Is it regular?

Exercise 7.13 Assume that g is a total recursive two-place function, and f is a total recursive one-place function. Show that we can construct a recursive function h such that $h(x)$ is the least y for which $g(x, y) = f(y)$, assuming such a y always exists.

Exercise 7.14 Assume that R is primitive recursive two-place relation. Show that we can construct a recursive function h such that $h(x)$ is the least number y for which $R(x, y)$ holds. (If there is no such y, $h(x)$ is undefined.)

Exercise 7.15 Consider the function $h(x) = \mu y\,(2y = x)$. What does this function do? Can you define h with the Mn notation?

Exercise 7.16 Use minimization to define a one-place function $h(x)$ that is undefined for every input x.

If we add minimization to our toolkit for constructing functions, we get the class of partial recursive functions. If we add regular minimization, we get the class of (total) recursive functions.

Definition 7.3

A function is *partial recursive* if it can be defined from the base functions s, z, and π_i^n by finitely many applications of composition, primitive recursion, and minimization.

Definition 7.4

A function is *(total) recursive* (a.k.a. *μ-recursive*) if it can be defined from the base functions s, z, and π_i^n by finitely many applications of composition, primitive recursion, and regular minimization.

As before, we can extend the concept of recursiveness to sets and relations.

Definition 7.5

A set is recursive if its characteristic function is (total) recursive. A relation is recursive if its extension is (total) recursive.

Above, I mentioned two functions that are computable but not primitive recursive: the Goodstein function and the antidiagonal of the primitive recursive functions. Both these functions are recursive. There is no known example of a computable function that is not recursive, and there are good reasons to believe that no such function exists. As we're going to show next, any such function would also be uncomputable by a Turing machine.

7.4 Recursiveness and Turing-computability

We'll now show that the class of partial recursive functions coincides precisely with the class of Turing-computable functions. We take the two directions in turn, starting with the easier direction: every partial recursive function is Turing-computable.

The proof idea is simple. Since every partial recursive function is built up from the base functions by composition, primitive recursion, and minimization, all we need to show is that the base functions are Turing-computable, and that the Turing-computable functions are closed under composition, primitive recursion, and minimization.

Theorem 7.1

Every partial recursive function is Turing-computable.

Proof sketch. The proof is by induction on the construction of partial recursive functions. I assume the same coding convention as in section 6.2, so that a number n is represented by a block of $n + 1$ strokes.

Base functions. You designed a Turing machine for the successor function in exercise 6.4. A machine for the zero function erases the input, writes a stroke, and halts. A machine for the projection functions erases all but one of its input blocks. These machines are trivial to design.

Composition. I assume for readability that $n = 1$ and $m = 2$. Suppose we have Turing machines for computing g_1, g_2 and f. We can design a machine for computing $h = \mathrm{Cn}[f, g_1, g_2]$ on any input x as follows. The machine first calls the g_1 and g_2 machines (as subroutines) on input x, and stores the results next to each other, separated by a blank. It then calls the f machine on this pattern of strokes and blanks. The output is $f(g_1(x), g_2(x))$.

Primitive recursion. Suppose we have Turing machines for computing f and g. Let $h = \mathrm{Pr}[f, g]$. That is, $h(x, 0) = f(x)$ and $h(x, y+1) = g(x, y, h(x, y))$, assuming for readability that f is 1-place. The machine for computing h works as follows. Given input x, y, it first calls the f-machine on x and stores the result in a block that will eventually hold $h(x, y)$; call this the "result block". The input y is kept on the tape in a separate "y block". In yet another block, we initialize a counter to 0 (represented by a single stroke). The machine then enters a loop. If the counter has the same length as the y block, the machine erases everything except the result block and halts. Otherwise it calls the g machine on x, the current counter value, and the current result block, and stores the output in the result block. The machine then increments the counter (by adding one stroke) and enters the next iteration of the loop. The loop will run exactly y times before halting. At that point, the result block will contain $h(x, y)$.

Minimization. Suppose we have a Turing machine for computing f. Let $h = \mathrm{Mn}[f]$. that is, $h(x) = \mu y\,[f(x, y) = 0]$, assuming for readability that h is 1-place. We can construct a machine for computing h as follows. First, the machine initialises a "y block" to 0, represented by a single stroke. It then goes into a loop. In each iteration, it runs the machine for f on x and the current y block. If the output is 0, the machine halts and erases everything except the y block. Otherwise, the machine adds a stroke to the y block and enters the next iteration of the loop. If there is some y such that $f(x, y) = 0$ and $f(x, z)$ is defined for all $z < y$, this machine will output the least such y. □

Now for the other direction: every Turing-computable function (on $\mathbb{N}$) is recursive. Let M be a Turing machine that computes some (possibly partial) function f on $\mathbb{N}$. For simplicity, let's assume that f is a 1-place function. Our task is to find a recursive definition of f. Here's an outline of how this can be done.

Remember that each stage of a Turing machine computation is captured by a *configuration*. A configuration records the current state of the machine, the position of the

head, and the contents of the tape. The initial configuration of our machine M on some input x, for example, specifies that the machine is in state q_0 and that its head is scanning the leftmost stroke of a block of $x+1$ strokes on an otherwise blank tape. We can code any such configuration as a natural number. Let init be a function that takes a number x as input and outputs the code number of M's initial configuration for input x. With a suitable coding scheme, this function will be primitive recursive.

Let $\text{next}(c)$ be a function that takes the code number c of a configuration as input and outputs the code number of the next configuration, according to the rules of M. If there are no applicable rules (i.e., if M halts in configuration c), we let $\text{next}(c)$ equal c. The function next is also primitive recursive.

From init and next, we can define (by primitive recursion) another function conf that takes an input x and a step number y, and outputs the code number of M's configuration after y steps on input x:

$$\begin{aligned} \text{conf}(x, 0) &= \text{init}(x) \\ \text{conf}(x, s(y)) &= \text{next}(\text{conf}(x, y)). \end{aligned}$$

We need two more functions. Let runs map the code number of any halting configuration of M to 0 and any other number to 1. Let out take the code number of a halting configuration as input and extract the content of the tape as output. Both of these are primitive recursive. We can now define the function f computed by M:

$$f(x) = \text{out}(\text{conf}(x, \mu y[\text{runs}(\text{conf}(x, y)) = 0])).$$

This says that $f(x)$ is the output extracted from the configuration at the first step at which M halts on input x.

The following proof sketch fills in a few more details.

Theorem 7.2

Every Turing-computable function is partial recursive.

Proof sketch. Let M be a Turing machine computing a (partial) function f on $\mathbb{N}$.

Each configuration of M can be coded as a quadruple $\langle q,\ L,\ s,\ R\rangle$, where q is the current state, $s \in \{0, 1\}$ is the scanned symbol (0 for blank, 1 for a stroke), and R is a finite sequence of 0s and 1s giving the contents of the tape to the right of the head (0 for blank, 1 for stroke) up to the last non-blank symbol, and L is a finite sequence

of 0s and 1s giving the contents of the tape to the left of the head, in reverse order, up to the last non-blank symbol. For example, if the tape is $\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{0}$ and the head is at the shaded cell, s would be 1, R would be 1, and L would be 1101. We can read s, R, and L as numbers in binary notation. (In the example, $s = 1$, $R = 1$, and $L = 2^3 + 2^2 + 2^0 = 13$.) If we code the state q as a number (using i for q_i), the entire configuration becomes a quadruple of natural numbers. We can code this quadruple as a single natural number using Gödel's prime-exponent coding (section 5.5). To extract the components of a coded configuration, we can use the primitive recursive functions len and entry from section 7.2.

The initial configuration for input x has state q_0, scanned symbol 1 (since the input is a block of $x + 1$ strokes), an empty left sequence, and a right sequence consisting of x many 1s (the input block minus the scanned stroke). The corresponding quadruple of numbers is

$$\langle 0,\ 0,\ 1,\ 2^x - 1 \rangle.$$

The init function therefore takes x as input and outputs the code number of this quadruple:

$$\text{init}(x) = 2^0 \cdot 3^0 \cdot 5^1 \cdot 7^{2^x - 1}.$$

This function is evidently primitive recursive.

To define next, we need a predicate $\text{HasRule}(x, y)$ that tests whether the machine table of M has a rule for state x and symbol y. (So $\text{HasRule}(2, 1)$ is true iff the machine has a rule for what to do in state q_2 when scanning a stroke.) Since the machine table is finite, this is a finite boolean combination of equalities, hence primitive recursive.

We can similarly define functions $\text{nextState}(x, y)$, $\text{write}(x, y)$, and $\text{move}(x, y)$ that extract the relevant components of the rule for state x and symbol y in the machine table (and return some arbitrary default value if no such rule exists).

With these in place, we can define next by cases. Given a coded configuration c as input, from which we can extract the quadruple $\langle q, L, s, R \rangle$ using entry, the next function first checks if $\text{HasRule}(q, s)$. If no, it returns c. If yes, it computes $\text{move}(q, s)$, $\text{write}(q, s)$, and $\text{nextState}(q, s)$. If the move is to the right, the new L becomes the old L with the written symbol appended at the front (in binary), the new scanned symbol becomes the first symbol of R, and the new R becomes the rest of R. If the move is to the left, the new R becomes the old R with the written symbol appended at the front, the new scanned symbol becomes the first symbol of L, and the new L becomes the rest of L. These operations on binary numbers (appending/deleting/extracting the first symbol) are primitive recursive. So next is primitive recursive.

We define conf as described above:

$$\begin{aligned} \text{conf}(x, 0) &= \text{init}(x) \\ \text{conf}(x, s(y)) &= \text{next}(\text{conf}(x, y)). \end{aligned}$$

The function runs is easily defined from HasRule:

$$\text{runs}(c) = \begin{cases} 1 & \text{if HasRule}(\text{entry}(c, 0), \text{entry}(c, 2)), \\ 0 & \text{otherwise.} \end{cases}$$

It remains to define the out function that extracts the output number represented by the tape contents in a halting configuration c. This simply needs to add the number of 1s in the left sequence, the scanned symbol, and the right sequence, and subtract 1.

Finally, we can define the function f computed by M, as announced above:

$$f(x) = \text{out}(\text{conf}(x, \mu y[\text{runs}(\text{conf}(x, y)) = 0])).$$

Equivalently,

$$f = \text{Cn}[\text{Cn}[\text{out}, \text{conf}], \pi_1^1, \text{Mn}[\text{Cn}[\text{runs}, \text{conf}]]].$$

□

Notice that the entire construction only uses a single unbounded μ, at the very end. We've therefore discovered an interesting corollary:

Theorem 7.3: (Kleene's Normal Form Theorem)

Every partial recursive function can be defined using a single instance of Mn.

Proof. Immediate from the proof of Theorem 7.2.

What about total recursive functions? We can show that the total recursive functions are precisely the Turing-computable total functions. It follows that a function is recursive iff it is partial recursive and total.

Theorem 7.4

A total function is recursive iff it is Turing-computable.

Proof sketch. The left-to-right direction is immediate from Theorem 7.1. For the right-to-left direction, we show that whenever minimization yields a total function, it can be replaced by regular minimization.

Let f be a partial recursive function, and $h = \mathrm{Mn}[f]$. By Theorem 7.1, there is a Turing machine M that computes f. Assume h is total. This means that for any x there is a y such that M halts on input x, y with output 0, and M halts on input x, z with some nonzero output for all $z < y$. (I assume without loss of generality that f has one argument.) It follows that for any x there is a y and a bound t such that

(i) M halts within t steps on input x, y with output 0, and
(ii) for all $z < y$, M halts within t steps on input x, z with some nonzero output.

We can express (i) and (ii) in terms of conf, out, and out from the proof of Theorem 7.2. That is, we can define a primitive recursive 3-ary predicate H so that $H(x, y, t)$ holds iff x, y, and t satisfy conditions (i) and (ii). Define

$$g(x, w) = \begin{cases} 0 & \text{if } H(x, \mathrm{entry}(w, 0), \mathrm{entry}(w, 1)), \\ 1 & \text{otherwise.} \end{cases}$$

So $g(x, w)$ returns 0 iff w encodes a pair $\langle y, t \rangle$ such that (i) and (ii) hold of x, y, and t. The function g is regular. We can define h by regular minimization from g:

$$h = \mathrm{Cn}[\mathrm{entry}, \pi^1_1, \mathrm{Mn}[g]].$$

□

Since the Turing-computable functions and the partial recursive functions coincide, it doesn't matter if we state the Church-Turing thesis (Section 5.2) as the claim that the computable functions are precisely the partial recursive functions or as the claim that the computable functions are precisely the Turing-computable functions. The two claims are equivalent.

I can now also explain – if it isn't clear already – why the Church-Turing Thesis is true. Take any partial recursive function: a function that can be defined from zero, successor, and projection by composition, primitive recursion, and minimization. The definition effectively *gives us* an algorithm for computing the function. That such an algorithm exists, or that one could in principle follow it mechanically, is not a speculative conjecture. This direction of the Church-Turing thesis is beyond doubt.

The other direction, that every computable function is recursive, requires a little more

argument. Here, we may draw on the fact that any mechanical computation must be representable as a rule-based, step-by-step manipulation of symbols, where the manipulations at each step are determined by a finite set of predefined rules. This isn't fully precise, but every way of making it precise leads to the same conclusion: the computation can be simulated by a Turing machine, and so it computes a partial recursive function.

Exercise 7.17 Give a diagonal argument to show that the set of total computable functions is not computably enumerable. Using the Church-Turing thesis, show that (a) the set of total recursive functions is not computably enumerable, and therefore (b) the set of regular recursive functions is not decidable.

7.5 Feasible computation

I mentioned in section 7.1 that the base functions s, z, and π_i^n are computable "in one step", without subroutines or loops. We can also count the number of steps needed to compute more complex functions. For example, if h is defined by composition from f and g, so that $h(x) = f(g(x))$, then one can compute h by first computing $g(x)$, then feeding the result into f; the total number of steps is the sum of the number of steps needed to compute f and g, for the given input x. We may also count the steps in a Turing machine computation that executes a given algorithm. Again, the number of steps will generally depend on the input.

Either way, the "step count" gives us a way to measure the *computational complexity* of an algorithm. The field of computational complexity theory studies different types of complexity. For example, in the class of *linear-time* algorithms, the number of steps it takes to compute an output is (at most) proportional to the size of the input. In the broader class of *polynomial-time* algorithms, the number of steps is bound by a polynomial function of the input size. For example, if the input has size n, the number of steps might be bound by n^2, or by $10n^{10} + 3n^7$.

The class of polynomial-time algorithms turns out to be very natural. It doesn't depend on details of how we count steps or how we measure the size of the input; it is also closed under composition and "subroutine insertion", wherein an arbitrary part of an algorithm is replaced by another algorithm. In analogy to the Church-Turing theses, the polynomial-time algorithms have been suggested to formalize the informal concept of a *feasible* algorithm.

Consider, for example, the task of checking whether a given sentence S of propositional logic is true in a given model σ, which assigns a truth-value to every sentence

letter. It's easy to show that this can be achieved by a polynomial-time algorithm, using the truth-table method. This algorithm is feasible. By contrast, consider the task of checking whether an $\mathfrak{L}_0$-sentence S is true relative to *some* assignment of truth-values – that is, whether it is satisfiable. How could we do this? The obvious "brute-force" algorithm is to try out all possible assignments of truth-values to the sentence letters in S. This algorithm is not polynomial, but *exponential* in the number of sentence letters. For 100 sentence letters, it requires $2^{100} \approx 10^{30}$ steps. This is clearly not a feasible algorithm.

Oddly, it is not known whether there is also a feasible, polynomial-time algorithm for deciding whether an $\mathfrak{L}_0$-sentence is satisfiable. Nobody has yet found such an algorithm, and it is generally believed that none exists. But we don't know for sure. This is an instance of the notorious *P vs NP problem*, which has yet to be resolved.