2 General Recursive Functions

In the preceding chapter, we saw an overview of several possible formalizations of the concept of effective calculability. In this chapter, we focus on *one* of those: primitive recursiveness and search, which give us the class of general recursive partial functions. In particular, we develop tools for showing that certain functions are in this class. These tools will be used in Chapter 3, where we study computability by register-machine programs.

2.1 Primitive Recursive Functions

The primitive recursive functions have been defined in the preceding chapter as the functions on $\mathbb{N}$ that can be built up from zero functions

$$f(x_1, \ldots, x_k) = 0,$$

the successor function

$$S(x) = x + 1,$$

and the projection functions

$$I_n^k(x_1, \ldots, x_k) = x_n$$

by using (zero or more times) composition

$$h(\vec{x}) = f(g_1(\vec{x}), \ldots, g_n(\vec{x}))$$

and primitive recursion

$$\begin{aligned} h(\vec{x}, 0) &= f(\vec{x}) \\ h(\vec{x}, y + 1) &= g(h(\vec{x}, y), \vec{x}, y), \end{aligned}$$

where $\vec{x}$ can be empty:

$$\begin{aligned} h(0) &= m \\ h(y + 1) &= g(h(y), y). \end{aligned}$$

Example: Suppose we are given the number $m = 1$ and the function $g(w, y) = w \cdot (y + 1)$. Then the function h obtained by primitive recursion from g by using m is the

Computability Theory. DOI: 10.1016/B978-0-12-384958-8.00002-8

function given by the pair of equations

$$\begin{aligned} h(0) &= m = 1 \\ h(y+1) &= g(h(y), y) = h(y) \cdot (y+1). \end{aligned}$$

Using this pair of equations, we can proceed to calculate the values of the function h:

$$\begin{aligned} h(0) &= m = 1 \\ h(1) &= g(h(0), 0) = g(1, 0) = 1 \\ h(2) &= g(h(1), 1) = g(1, 1) = 2 \\ h(3) &= g(h(2), 2) = g(2, 2) = 6 \\ h(4) &= g(h(3), 3) = g(6, 3) = 24 \end{aligned}$$

And so forth. In order to calculate $h(4)$, we first need to know $h(3)$, and to find that we need $h(2)$, and so on. The function h in this example is, of course, better known as the factorial function, $h(x) = x!$.

It should be pretty clear that given any number m and any two-place function g, *there exists* a unique function h obtained by primitive recursion from g by using m. It is the function h that we calculate as in the preceding example. Similarly, given a k-place function f and a $(k+2)$-place function g, there exists a unique $(k+1)$-place function h that is obtained by primitive recursion from f and g. That is, h is the function given by the pair of equations

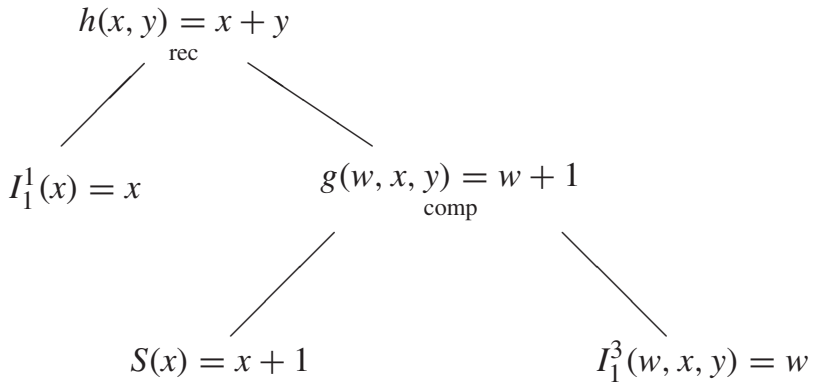
$$\begin{aligned} h(\vec{x}, 0) &= f(\vec{x}) \\ h(\vec{x}, y+1) &= g(h(\vec{x}, y), \vec{x}, y). \end{aligned}$$

Moreover, if f and g are total functions, then h will also be total.

Example: Consider the addition function $h(x, y) = x + y$. For any fixed x, its value at $y+1$ (i.e., $x+y+1$) is obtainable from its value at y (i.e., $x+y$) by the simple step of adding one:

$$\begin{aligned} x + 0 &= x \\ x + (y+1) &= (x+y) + 1. \end{aligned}$$

This pair of equations shows that addition is obtained by primitive recursion from the functions $f(x) = x$ and $g(w, x, y) = w + 1$. These functions f and g are primitive recursive; f is the projection function I_1^1, and g is obtained by composition from successor and I_1^3. Putting these observations together, we can form a tree showing how addition is built up from the initial functions by composition and primitive recursion:

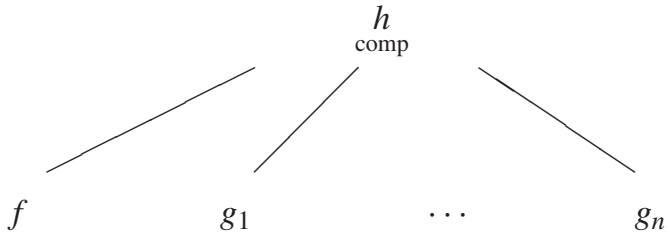


More generally, for any primitive recursive function h, we can use a labeled tree ("construction tree") to illustrate exactly how h is built up, as in the example of addition. At the top (root) vertex, we put h. At each minimal vertex (a leaf), we have an initial function: the successor function, a zero function, or a projection function. At each other vertex, we display either an application of composition or an application of primitive recursion.

An application of composition

$$h(\vec{x}) = f(g_1(\vec{x}), \ldots, g_n(\vec{x}))$$

can be illustrated in the tree by a vertex with $(n + 1)$-ary branching:

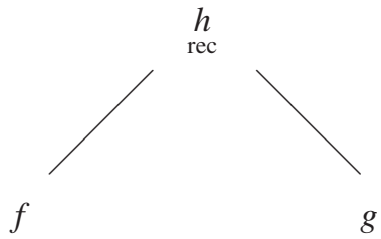


Here f must be an n-place function, and $g_1, \ldots, g_n$ must all have the same number of places as h.

An application of primitive recursion to obtain a $(k + 1)$-place function h

$$\begin{cases} h(\vec{x}, 0) = f(\vec{x}) \\ h(\vec{x}, y + 1) = g(h(\vec{x}, y), \vec{x}, y) \end{cases}$$

can be illustrated by a vertex with binary branching:



Note that g must have two more places than f, and one more place than h (e.g., if h is a two-place function, then g must be a three-place function and f must be a one-place function).

The $k = 0$ case, where a one-place function h is obtained by primitive recursion from a two-place function g by using the number m

$$\begin{cases} h(0) = m \\ h(x+1) = g(h(x), x), \end{cases}$$

can be illustrated by a vertex with unary branching:

In both forms of primitive recursion ($k > 0$ and $k = 0$), the key feature is that the value of the function at a number $t + 1$ is somehow obtainable from its value at t. The role of g is to explain how.

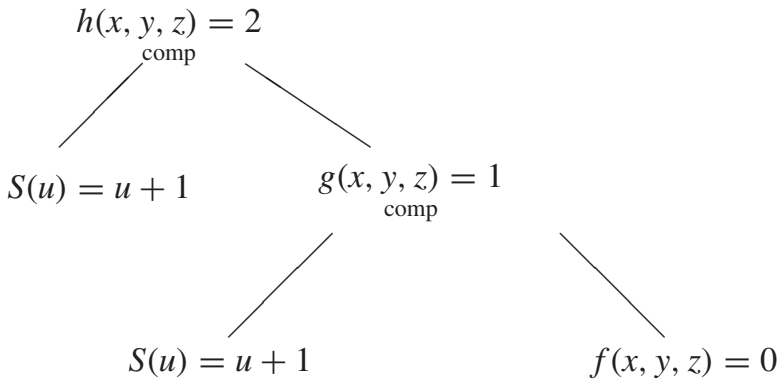
Every primitive recursive function is total. We can see this by "structural induction." For the basis, all of the initial functions (the zero functions, the successor function, and the projections functions) are total. For the two inductive steps, we observe that composition of total functions yields a total function, and primitive recursion applied to total functions yields a total function. So for any primitive recursive function, we can work our way up its construction tree. At the leaves of the tree, we have total functions. And each time we move to a higher vertex, we still have a total function. Eventually, we come to the root at the top, and conclude that the function being constructed is total.

Next we want to build up a catalog of basic primitive recursive functions. These items in the catalog can then be used as "off the shelf" parts for later building up of other primitive recursive functions.

1. Addition $\langle x, y\rangle \mapsto x + y$ has already been shown to be primitive recursive.

The symbol "$\mapsto$" is read "maps to." The symbol gives us a very convenient way to name functions. For example, the squaring function can be named by the lengthy phrase "the function that given a number, squares it," which uses the pronoun "it" for the number. It is mathematically convenient to use a letter (such as x or t) in place of this pronoun. This leads us to the names "the function whose value at x is x^2" or "the function whose value at t is t^2." More compactly, these names can be written in symbols as "$x \mapsto x^2$" or "$t \mapsto t^2$." The letter x or t is a dummy variable; we can use any letter here.

2. Any constant function $\vec{x} \mapsto k$ can be obtained by applying composition k times to the successor function and the zero function $\vec{x} \mapsto 0$. For example, the three-place function that constantly takes the value 2 can be constructed by the following tree:



3. For multiplication $\langle x, y\rangle \mapsto x \times y$, we first observe that

$$\begin{aligned} x \times 0 &= 0 \\ x \times (y + 1) &= (x \times y) + x. \end{aligned}$$

This shows that multiplication is obtained by primitive recursion from the functions $x \mapsto 0$ and $\langle w, x, y\rangle \mapsto w + x$. The latter function is obtained by composition applied to addition and projection functions.

We can now conclude that any polynomial function with positive coefficients is primitive recursive. For example, we can see that the function $p(x, y) = x^2y + 5xy + 3y^3$ is primitive recursive by repeatedly applying **1**, **2**, and **3**.

4. Exponentiation $\langle x, y\rangle \mapsto x^y$ is similar:

$$\begin{aligned} x^0 &= 1 \\ x^{y+1} &= x^y \times x. \end{aligned}$$

5. Exponentiation $\langle x, y\rangle \mapsto y^x$ is obtained from the preceding function by composition with projection functions. (The functions in items **4** and **5** are different functions; they assign different values to $\langle 2, 3\rangle$. The fact that they coincide at $\langle 2, 4\rangle$ is an accident.)

We should generalize this observation. For example, if f is primitive recursive, and g is defined by the equation

$$g(x, y, z) = f(y, 3, x, x)$$

then g is also primitive recursive, being obtained by composition from f and projection and constant functions. We will say in this situation that g is obtained from f by *explicit transformation*. Explicit transformation permits scrambling variables, repeating variables, omitting variables, and substituting constants.

6. The factorial function $x!$ satisfies the pair of recursion equations

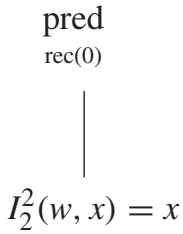
$$\begin{aligned} 0! &= 1 \\ (x + 1)! &= x! \times (x + 1). \end{aligned}$$

From this pair of equations, it follows that the factorial function is obtained by primitive recursion (by using 1) from the function $g(w, x) = w \cdot (x + 1)$. (See the example at the beginning of this chapter.)

7. The predecessor function $\text{pred}(x) = x - 1$ (except that $\text{pred}(0) = 0$) is obtained by primitive recursion from I_2^2:

$$\begin{aligned} \text{pred}\ (0) &= 0 \\ \text{pred}\ (x+1) &= x. \end{aligned}$$

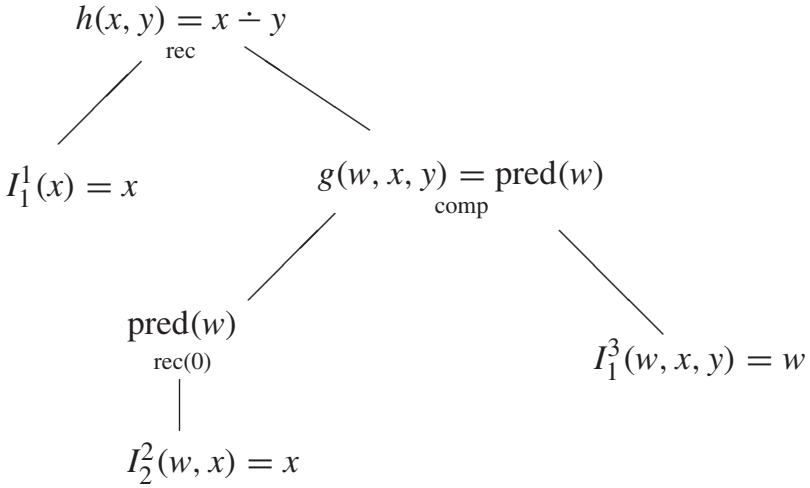
This pair of equations leads to the tree:



8. Define the *proper subtraction* function $x \dot{-} y$ by the equation $x \dot{-} y = \max(x - y, 0)$. This function is primitive recursive:

$$\begin{aligned} x \dot{-} 0 &= x \\ x \dot{-} (y+1) &= \text{pred}(x \dot{-} y) \end{aligned}$$

This pair of recursion equations yields the following construction tree:



By the way, the symbol $\dot{-}$ is sometimes read as "monus."

9. Assume that f is primitive recursive, and define the functions s and p by the equations

$$s(\vec{x}, y) = \sum_{t<y} f(\vec{x}, t) \quad \text{and} \quad p(\vec{x}, y) = \prod_{t<y} f(\vec{x}, t)$$

(subject to the standard conventions for the empty sum $\sum_{t<0} f(\vec{x}, t) = 0$ and the empty product $\prod_{t<0} f(\vec{x}, t) = 1$). Then both s and p are primitive recursive. For p, we have the pair of equations:

$$\begin{aligned} p(\vec{x}, 0) &= 1 \\ p(\vec{x}, y+1) &= p(\vec{x}, y) \cdot f(\vec{x}, y) \end{aligned}$$

10. Define the function z by the equation

$$z(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x > 0. \end{cases}$$

That is, the function z looks to see if its input is zero, and returns Yes (i.e., 1) if it is zero; otherwise, it returns No (i.e., 0). The function z is primitive recursive. We can see this from the equation $z(x) = 0^x$. More directly, we can see it from the equation $z(x) = 1 \dot{-} x$. And even more directly, we can see it from the recursion equations

$$\begin{aligned} z(0) &= 1 \\ z(x+1) &= 0 \end{aligned}$$

showing that z is obtained by primitive recursion (by using 1) from the function $g(w, x) = 0$.

$$\begin{array}{c} z \\ \text{rec}(1) \\ \Big| \\ g(w, x) = 0 \end{array}$$

11. In a similar vein, the function h that checks its two inputs x and y to see whether or not $x \leq y$

$$h(x, y) = \begin{cases} 1 & \text{if } x \leq y \\ 0 & \text{if otherwise} \end{cases}$$

is primitive recursive because $h(x, y) = z(x \dot{-} y)$.

Items **10** and **11** can be reformulated in terms of relations (instead of functions). Suppose that R is a k-ary relation on the natural numbers, that is, R is some set of k-tuples of natural numbers: $R \subseteq \mathbb{N}^k$. We define R to be a *primitive recursive relation* if its characteristic function

$$C_R(x_1, \ldots, x_k) = \begin{cases} 1 & \text{if } \langle x_1, \ldots, x_k \rangle \in R \\ 0 & \text{if otherwise} \end{cases}$$

is a primitive recursive function. For example, item **11** states that the ordering relation $\{\langle x, y \rangle \mid x \leq y\}$ is a primitive recursive binary relation. And item **10** states that $\{0\}$ is a primitive recursive unary relation.

From composition, we derive the *substitution rule*: If Q is an n-ary primitive recursive relation, and $g_1, \ldots, g_n$ are k-place primitive recursive functions, then the k-ary relation

$$\{\vec{x} \mid \langle g_1(\vec{x}), \ldots, g_n(\vec{x}) \rangle \in Q\}$$

is primitive recursive because its characteristic function is obtained from C_Q and $g_1, \ldots, g_n$ by composition.

From a relation Q, we can form its *complement* $\overline{Q}$:

$$\overline{Q} = \{\vec{x} \mid \vec{x} \text{ is not in } Q\}$$

From two k-ary relations Q and R (for the same k), we can form their *intersection*,

$$Q \cap R = \{\vec{x} \mid \text{both } \vec{x} \in Q \text{ and } \vec{x} \in R\}$$

and their *union*

$$Q \cup R = \{\vec{x} \mid \text{either } \vec{x} \in Q \text{ or } \vec{x} \in R \text{ or both}\}.$$

We can streamline the notation slightly by writing, instead of $\vec{x} \in Q$, simply $Q(\vec{x})$. In this notation,

$$\begin{aligned} \overline{Q} &= \{\vec{x} \mid \text{not } Q(\vec{x})\}, \\ Q \cap R &= \{\vec{x} \mid \text{both } Q(\vec{x}) \text{ and } R(\vec{x})\}, \\ Q \cup R &= \{\vec{x} \mid \text{either } Q(\vec{x}) \text{ or } R(\vec{x}) \text{ or both}\}. \end{aligned}$$

The following theorem assures us that these constructions preserve primitive recursiveness. That is, when applied to primitive recursive relations, they produce primitive recursive relations. This theorem will be useful in extending our supply of primitive recursive relations and functions.

Theorem: *Assume that Q and R are k-ary primitive recursive relations. Then the following relations are also primitive recursive:*

(a) *The complement $\overline{Q}$ of Q:*

$$\overline{Q} = \{\vec{x} \mid \text{not } Q(\vec{x})\}$$

(b) *The intersection $Q \cap R$ of Q and R:*

$$Q \cap R = \{\vec{x} \mid \text{both } Q(\vec{x}) \text{ and } R(\vec{x})\}$$

(c) *The union $Q \cup R$ of Q and R:*

$$Q \cup R = \{\vec{x} \mid \text{either } Q(\vec{x}) \text{ or } R(\vec{x}) \text{ or both}\}$$

Proof.

(a)

$$C_{\overline{Q}}(\vec{x}) = z(C_Q(\vec{x}))$$

where z is the function from **10**. That is, $C_{\overline{Q}}$ is obtained by composition from functions known to be primitive recursive. The other parts are proved similarly; we need to make the characteristic function from primitive recursive parts.

(b)

$$C_{Q\cap R}(\vec{x}) = C_Q(\vec{x}) \cdot C_R(\vec{x})$$

(c)

$$C_{Q\cup R}(\vec{x}) = \text{pos}[C_Q(\vec{x}) + C_R(\vec{x})]$$

where pos is the function from Exercise 5.

⊣

For example, we can apply this theorem to conclude that $>$ and $=$ are primitive recursive relations:

12. The relation $\{\langle x, y\rangle \mid x > y\}$ is primitive recursive because it is the complement of the $\leq$ relation from item **11**.

13. The relation $\{\langle x, y\rangle \mid x = y\}$ is primitive recursive because it is the intersection of the $\leq$ and the $\geq$ relations, and $\geq$ is obtained from $\leq$ by explicit transformation.

It follows from item **13** and the substitution rule that for any primitive recursive function f, its graph

$$\{\langle \vec{x}, y\rangle \mid f(\vec{x}) = y\}$$

is a primitive recursive relation.

Definition by cases: *Assume that Q is a primitive recursive k-ary relation, and that f and g are primitive recursive k-place functions. Then the function h defined by the equation*

$$h(\vec{x}) = \begin{cases} f(\vec{x}) & \text{if } Q(\vec{x}) \\ g(\vec{x}) & \text{if not } Q(\vec{x}) \end{cases}$$

is also primitive recursive.

Proof. $h(\vec{x}) = f(\vec{x}) \cdot C_Q(\vec{x}) + g(\vec{x}) \cdot C_{\overline{Q}}(\vec{x})$. ⊣

This result can be extended to more than two cases; see Exercise 12. For example, we might want to handle an equation of the form

$$h(\vec{x}) = \begin{cases} f_1(\vec{x}) & \text{if } Q(\vec{x}) \text{ and } R(\vec{x}) \\ f_2(\vec{x}) & \text{if } Q(\vec{x}) \text{ and not } R(\vec{x}) \\ f_3(\vec{x}) & \text{if } R(\vec{x}) \text{ and not } Q(\vec{x}) \\ f_4(\vec{x}) & \text{if neither } Q(\vec{x}) \text{ nor } R(\vec{x}) \end{cases}$$

or one of the form

$$h(\vec{x}) = \begin{cases} f_1(\vec{x}) & \text{if } Q_1(\vec{x}) \\ f_2(\vec{x}) & \text{if } Q_2(\vec{x}) \\ \cdots & \cdots \\ f_9(\vec{x}) & \text{if } Q_9(\vec{x}) \\ f_{10}(\vec{x}) & \text{if none of the above} \end{cases}$$

in a situation in which it is known that no two of $Q_1, \ldots, Q_9$ can hold simultaneously.

Moreover, from a k-ary relation Q, we can form

$$\{\langle x_1, \ldots, x_{k-1}, y\rangle \mid \text{ for every } t < y, \langle x_1, \ldots, x_{k-1}, t\rangle \in Q\},$$

which can be written in better notation as

$$\{\langle \vec{x}, y\rangle \mid (\forall t < y)\, Q(\vec{x}, t)\},$$

where the symbol $\forall$ is read "for all." In the same spirit, we can form

$$\{\langle x_1, \ldots, x_{k-1}, y\rangle \mid \text{ for some } t < y, \langle x_1, \ldots, x_{k-1}, t\rangle \in Q\},$$

which is better written as

$$\{\langle \vec{x}, y\rangle \mid (\exists t < y)\, Q(\vec{x}, t)\},$$

where the symbol $\exists$ is read as "there exists ... such that."

Again, these constructions preserve primitive recursiveness:

Theorem: *Assume that Q is a $(k+1)$-ary primitive recursive relation. Then the following relations are also primitive recursive:*

(a)

$$\{\langle \vec{x}, y\rangle \mid (\forall t < y)\, Q(\vec{x}, t)\}$$

(b)

$$\{\langle \vec{x}, y\rangle \mid (\exists t < y)\, Q(\vec{x}, t)\}$$

Proof.

(a) The value of the characteristic function at $\langle \vec{x}, y\rangle$ is

$$\prod_{t<y} C_Q(\vec{x}, t).$$

Apply item **9**.

(b) The value of the characteristic function at $\langle \vec{x}, y \rangle$ is

$$\text{pos}\left[\sum_{t<y} C_Q(\vec{x}, t)\right]$$

where pos is the function from Exercise 5. This is primitive recursive by item **9** and Exercise 5.

⊣

For example, we can apply these results to show that the relation

$$\{\langle x, y \rangle \mid (\exists q < y + 1)[x \cdot q = y]\}$$

is primitive recursive. We do this by looking at the way the above line is written, and then filling in the details. First of all, the ternary relation

$$R_1(x, y, q) \iff x \cdot q = y$$

is obtained from the equality relation by substituting the functions $\langle x, y, q \rangle \mapsto x \cdot q$ and I_2^3. Secondly, an application of the preceding theorem then shows that the ternary relation

$$R_2(x, y, z) \iff (\exists q < z)[x \cdot q = y]$$

is primitive recursive. Finally, we apply substitution:

$$(\exists q < y + 1)[x \cdot q = y] \iff R_2(x, y, y + 1).$$

In short, we can show that this relation is primitive recursive by examining the syntactical form of its definition and verifying that it has been built up by using only pieces that are known to be primitive recursive.

14. The divisibility relation $x \mid y$, that is, the relation

$$\{\langle x, y \rangle \mid x \text{ divides } y \text{ with 0 remainder}\},$$

is primitive recursive. (Here we adopt the convention that 0 divides itself, but it does not divide any positive integer.) This is because

$$\begin{aligned} x \mid y &\iff \text{for some quotient } q, \text{ we have } x \cdot q = y \\ &\iff (\exists q \leq y)[x \cdot q = y] \\ &\iff (\exists q < y + 1)[x \cdot q = y]. \end{aligned}$$

That is, the relation we examined in the foregoing example is nothing but the divisibility relation!

In effect, we are building up a certain *language* such that any function or relation definable in the language is guaranteed to be primitive recursive. (For divisibility, the crucial fact was that the expression "$(\exists q < y + 1)[x \cdot q = y]$" belonged to this language.) This language includes the following:

- Variables: The projection functions are primitive recursive.
- Constants (numerals): The constant functions are primitive recursive.
- Function symbols: We can use symbols for any primitive recursive function in the list we are building up ($+$, $\times$, $\dot{-}$, ..., with more to come).
- Combinations: $\sum_{x<y}$, $\prod_{x<y}$, with more to come.
- Relation symbols: We can use symbols for any primitive recursive relation in the list we are building up ($\leq$, $=$, $|$, ..., with more to come).
- More combinations: "not," "and," "or" can be applied to relations.
- Bounded "quantifiers": $\forall x < y$ and $\exists x < y$. (The upper bound y is needed here.)

We have theorems assuring us that functions or relations expressible in this language are certain to be primitive recursive.

For example, we next add the set of primes (as a unary relation) to our list:

15. The set $\{2, 3, 5, \ldots\}$ of prime natural numbers (as a unary relation on $\mathbb{N}$) is primitive recursive. To see this, observe that

$$x \text{ is prime} \iff 1 < x \text{ and } (\forall u < x)(\forall v < x)[uv \neq x],$$

and the right-hand side is written within the language available to us.

2.1.1 Bounded Search

The search operator (often called minimalization or the μ-operator) provides a useful way of defining a function in terms of a "search" for the first time a given condition is satisfied.

Definition: For a $(k+1)$-ary relation P, the number $(\mu t < y)P(\vec{x}, t)$ is defined by the equation:

$$(\mu t < y)P(\vec{x}, t) = \begin{cases} \text{the least } t \text{ such that } t < y \text{ and } P(\vec{x}, t), \text{ if any} \\ y \quad \text{if there is no such } t \end{cases}$$

For example, if we let

$$f(x, y) = \mu t < y[t \text{ is prime and } x < t]$$

then $f(6, 4) = 4$ and $f(6, 8) = f(6, 800) = 7$.

Theorem: *If P is a primitive recursive relation, then the function*

$$f(\vec{x}, y) = (\mu t < y)P(\vec{x}, t)$$

is a primitive recursive function.

Proof. We will apply primitive recursion. Trivially $f(\vec{x}, 0) = 0$, so there is no problem here. The problem to see how $f(\vec{x}, y+1)$ (call this b) depends on $f(\vec{x}, y)$ (call this a):

- If $a < y$, then $b = a$. (The search below y succeeded.)
- If $a = y$ and $P(\vec{x}, y)$, then $b = y$.
- Otherwise, $b = y + 1$.

Thus, if we define

$$g(a, \vec{x}, y) = \begin{cases} a & \text{if } a < y \\ y & \text{if } a \not< y \text{ and } P(\vec{x}, y) \\ y+1 & \text{if } a \not< y \text{ and not } P(\vec{x}, y), \end{cases}$$

then f is obtained by primitive recursion from the functions $\vec{x} \mapsto 0$ and g. Because P is a primitive recursive relation, it follows that g is primitive recursive (by definition-by-cases), and hence f is primitive recursive. ⊣

There is another proof of this theorem, which relies on the following remarkable equation:

$$(\mu t < y)P(\vec{x}, t) = \sum_{u<y} \prod_{t \le u} C_{\overline{P}}(\vec{x}, t)$$

A related search operator is bounded *maximalization*. Define the $\overline{\mu}$-operator as follows:

$$(\overline{\mu} t \le y)P(\vec{x}, t) = \begin{cases} \text{the largest number } t \text{ such that } t \le y \text{ and } P(\vec{x}, t), & \text{if any} \\ 0 \quad \text{if there is no such } t \end{cases}$$

Theorem: *If P is a primitive recursive relation, then the function*

$$f(\vec{x}, y) = (\overline{\mu} t \le y)P(\vec{x}, t)$$

is a primitive recursive function.

Proof.

$$(\overline{\mu} t \le y)P(\vec{x}, t) = y \dot{-} (\mu s < y)P(\vec{x}, y \dot{-} s).$$

This equation captures the idea of searching down from y. ⊣

Euclid observed that the set of prime numbers is unbounded. Hence the function

$$h(x) = \text{the smallest prime number larger than } x$$

is total. It is also primitive recursive because

$$h(x) = \mu t < (x! + 2)[t \text{ is prime and } x < t].$$

The upper bound $x! + 2$ suffices because for any prime factor p of $x! + 1$, we have $x < p \le x! + 1$. So any search for a prime larger than x need go no further than $x! + 1$.

Digression: There is an interesting result in number theory here. "Bertrand's postulate" states that for any $x > 3$, there will always be a prime number p with $x < p < 2x - 2$. (Bertrand's postulate implies that in the previous paragraph, it suffices to use simply $h(x) = \mu t < (2x+3)[t \text{ is prime and } x < t]$.) In 1845, the French mathematician Joseph Bertrand, using prime number tables, verified this statement for x below three million. Then in 1850, the Russian P. L. Chebyshev (Tchebychef) proved the result in general. In 1932, the Hungarian Paul Erdős gave a better proof, which can now be found in undergraduate number theory textbooks. The origin of

Chebyshev said it
So I'll say it again
There's always a prime
Between N and $2N$

is unknown, which may be just as well.

Define p_x to be the $(x + 1)$st prime number, so that

$$p_0 = 2, \quad p_1 = 3, \quad p_2 = 5, \quad p_3 = 7, \quad p_4 = 11,$$

and so forth. In other words, p_x is the xth odd prime, except that $p_0 = 2$. (The prime number theorem tells us that p_x grows at a rate something like $x \ln x$, but that is beside the point.)

16. The function $x \mapsto p_x$ is primitive recursive because we have the recursion equations

$$\begin{aligned} p_0 &= 2 \\ p_{x+1} &= h(p_x), \end{aligned}$$

where h is the above function that finds the next prime.

It is easy to see that we always have $x+1 < p_x$; a formal proof can use induction.

We will need methods for encoding a string of numbers by a single number. One method that is conceptually simple uses powers of primes. We define the "bracket notation" as follows.

$$\begin{aligned} [\,] &= 1 \\ [x] &= 2^{x+1} \\ [x, y] &= 2^{x+1}3^{y+1} \\ [x, y, z] &= 2^{x+1}3^{y+1}5^{z+1} \\ &\cdots \\ [x_0, x_1, \ldots, x_k] &= 2^{x_0+1}3^{x_1+1}\cdots p_k^{x_k+1} \end{aligned}$$

For example, $[2, 1] = 72$ and $[2, 1, 0] = 360$. Clearly, for any one value of k, the function

$$\langle x_0, x_1, \ldots, x_k\rangle \mapsto [x_0, x_1, \ldots, x_k]$$

is a primitive recursive $(k+1)$-place function. But encoding is useless, unless we can *decode*. (The "fundamental theorem of arithmetic" is the statement that every positive integer has a factorization into primes, unique up to order. For decoding, we are implicitly exploiting the uniqueness of prime factorization.) Item **17** will give a primitive recursive decoding function.

Digression: Using powers of primes is by no means the only way to encode a string of numbers. It is a very convenient method for our present purposes, but there are a number of other methods. Here is a very different approach:

$$\langle x_0, x_1, \ldots, x_k\rangle \mapsto 1\underbrace{00\cdots0}_{x_0}1\underbrace{00\cdots0}_{x_1}1\cdots\cdots1\underbrace{00\cdots0}_{x_k}\ \text{two}$$

That is, a sequence of length n can be coded by the number whose binary representation has n 1's. The number of 0's that follow the ith 1 in the representation corresponds to the ith component in the sequence.

Here are some examples:

$$\begin{aligned}
\langle 0,3,2\rangle &\mapsto 11000100_{\text{two}} = 188\\
\langle 2,1,0\rangle &\mapsto 100101_{\text{two}} = 37\\
\langle\ \rangle &\mapsto 0_{\text{two}} = 0\\
\langle 7\rangle &\mapsto 10000000_{\text{two}} = 128\\
\langle 0,0,0,0\rangle &\mapsto 1111_{\text{two}} = 31
\end{aligned}$$

These values can be compared with the values yielded by the bracket notation: $[0,3,2] = 2^1\cdot3^4\cdot5^3 = 20,250$, $[2,1,0] = 2^3\cdot3^2\cdot5^1 = 360$, $[\ ] = 1$, $[7] = 2^8 = 256$, $[0,0,0,0] = 2\cdot3\cdot5\cdot7 = 210$.

In particular, suppose we want to encode a sequence of *two* numbers. This method yields

$$\langle m,n\rangle \mapsto 1\underbrace{00\cdots0}_{m}1\underbrace{00\cdots0}_{n}\ \text{two} = 2^{m+n+1} + 2^n = 2^n(2^{m+1}+1).$$

The bracket notation yields simply $[m,n] = 2^{m+1}\cdot3^{n+1}$. Both of these "pairing functions" have the feature that they grow exponentially as m and n increase.

Interestingly, there are *polynomial* pairing functions, and here is one:

$$J(m,n) = \frac{1}{2}((m+n)^2 + 3m + n).$$

The function J is one-to-one, so the pair $\langle m,n\rangle$ is recoverable from the value $J(m,n)$. In fact the function J maps $\mathbb{N}\times\mathbb{N}$ one-to-one *onto* $\mathbb{N}$.

And where does J come from? Here is a clue. Calculate $J(m,n)$ for all small values of m and n, say $m+n\le4$. Then make a chart in the plane, by placing the number $J(m,n)$ at the point in the plane with coordinates $\langle m,n\rangle$. Check if a pattern is emerging.

17. There is a primitive recursive two-place "decoding" function, whose value at $\langle x, y\rangle$ is written $(x)_y$, with the property that whenever $y \le k$,

$$([x_0, x_1, \ldots, x_k])_y = x_y.$$

That is,

$$(\text{code for a sequence})_y = \text{ the } (y+1)\text{st term of the sequence.}$$

For example, $(72)_0 = 2$ and $(72)_1 = 1$ because $72 = [2, 1]$.

First, observe that the exponent of a prime q in the factorization of a positive integer x is

$$\mu e \left(q^{e+1} \nmid x\right),$$

the smallest e for which $e+1$ would be too much. We can bound the search at x because if $q^e \mid x$, then $e < q^e \le x$. That is, the exponent of q in the factorization of x is

$$(\mu e < x) \left(q^{e+1} \nmid x\right).$$

Now suppose that prime q is p_y. We define

$$(x)_y^* = (\mu e < x) \left(p_y^{e+1} \nmid x\right)$$

so that $(x)_y^*$ is the exponent of p_y in the prime factorization of x.

Secondly, for our decoding function, we need one *less* than the exponent of the prime p_y in the factorization of the sequence code. Accordingly, we define

$$(x)_y = (x)_y^* \dot{-} 1 = (\mu e < x) \left(p_y^{e+1} \nmid x\right) \dot{-} 1.$$

The right-hand side of this equation is written in our language, so the function is primitive recursive. The function tests powers of p_y until it finds the largest one in the factorization of x, and then it backs down by 1. If p_y does not divide x, then $(x)_y = 0$, harmlessly enough. Also $(0)_y = 0$, but for a different reason.

18. Say that y is a *sequence number* if either $y = [\,]$ or $y = [x_0, x_1, \ldots, x_k]$ for some k and some $x_0, x_1, \ldots, x_k$. For example, 1 is a sequence number but 50 is not. The set of sequence numbers is primitive recursive; see Exercise 14. The set of sequence numbers starts off as $\{1, 2, 4, 6, 8, 12, \ldots\}$.

19. There is a primitive recursive function lh such that

$$\text{lh}[x_0, x_1, \ldots, x_k] = k + 1.$$

For example, $\text{lh}(360) = 3$. Here "lh" stands for "length." We define

$$\text{lh}(x) = (\mu k < x) \left(p_k \nmid x\right).$$

Thus, for example, $\text{lh}(50) = 1$.

It is apparent from its definition that this function is primitive recursive. The upper bound on the μ search is adequate because if $p_{k-1} \mid x$, then $(k-1)+1 < p_{k-1} \leq x$.

If s is a sequence number of positive length, then

$$(s)_{\mathrm{lh}(s) \dot{-} 1}$$

will be the *last* component of the sequence.

20. There is a two-place primitive recursive function whose value at $\langle x, y\rangle$ is called the *restriction* of x to y, written $x \upharpoonright y$, with the property that whenever $y \leq k+1$ then

$$[x_0, x_1, \ldots, x_k] \upharpoonright y = [x_0, x_1, \ldots, x_{y-1}].$$

That is, the restriction of x to y gives us the first y components of the sequence. We define

$$x \upharpoonright y = \prod_{i<y} p_i^{(x)_i^*}.$$

For example, if s is a sequence number, then $s \upharpoonright (\mathrm{lh}(s) \dot{-} 1)$ will encode the result of deleting the *last* item in the sequence, if any.

21. There is a two-place primitive recursive function whose value at $\langle x, y\rangle$ is called the *concatenation* of x and y, written $x * y$, with the property that whenever x and y are sequence numbers, then $x * y$ is the sequence number of length $\mathrm{lh}(x) + \mathrm{lh}(y)$ whose components are first the components of x and then the components of y. We define

$$x * y = x \cdot \prod_{i<\mathrm{lh}(y)} p_{i+\mathrm{lh}(x)}^{(y)_i^*}.$$

For example, $72 * 72 = [2, 1, 2, 1] = 441,000$. If s is a sequence number, then $s * [x]$ will encode the result of adjoining x to the *end* of the sequence.

22. We can also define a "capital asterisk" operation. Let

$$*_{t<y} a_t = a_0 * a_1 * \cdots * a_{y-1}$$

(grouped to the left). If f is a primitive recursive $(k+1)$-place function, then so is the function whose value at $\langle \vec{x}, y\rangle$ is $*_{t<y} f(\vec{x}, t)$, as can be seen from the pair of recursion equations:

$$*_{t<0} f(\vec{x}, t) = 1$$

$$*_{t<y+1} f(\vec{x}, t) = *_{t<y} f(\vec{x}, t) * f(\vec{x}, y)$$

For any $(k+1)$-place function f, we define $\overline{f}$ by the equation

$$\overline{f}(\vec{x}, y) = [f(\vec{x}, 0), f(\vec{x}, 1), \ldots, f(\vec{x}, y-1)]$$

so that the number $f(\vec{x}, y)$ encodes y values of f, namely the values $f(\vec{x}, t)$ for all $t < y$. For example, $\overline{f}(\vec{x}, 0) = [\,] = 1$, encoding 0 values. And $\overline{f}(\vec{x}, 2) = [f(\vec{x}, 0), f(\vec{x}, 1)]$. Clearly $\overline{f}(\vec{x}, y)$ is always a sequence number of length y.

23. If f is primitive recursive, then so is $\overline{f}$ because

$$\overline{f}(\vec{x}, y) = \prod_{i<y} p_i^{f(\vec{x},i)+1}.$$

Now suppose we have a $(k+2)$-place function g. Then there exists a unique function $(k+1)$-place f satisfying the equation

$$f(\vec{x}, y) = g(\overline{f}(\vec{x}, y), \vec{x}, y)$$

for all $\vec{x}$ and y. For example,

$$\begin{aligned} f(\vec{x}, 0) &= g([\,], \vec{x}, 0) = g(1, \vec{x}, 0) \\ f(\vec{x}, 1) &= g([f(\vec{x}, 0)], \vec{x}, 1) \\ f(\vec{x}, 2) &= g([f(\vec{x}, 0), f(\vec{x}, 1)], \vec{x}, 2) \end{aligned}$$

and so forth. The function f is determined recursively; we can find $f(\vec{x}, y)$ after we know $f(\vec{x}, t)$ for all $t < y$.

24. Assume that g is a primitive recursive $(k+2)$-place function, and let f be the unique $(k+1)$-place function for which

$$f(\vec{x}, y) = g(\overline{f}(\vec{x}, y), \vec{x}, y)$$

for all $\vec{x}$ and y. Then f is also primitive recursive.

To see that f is primitive recursive, we first examine $\overline{f}$. We have the pair of recursion equations

$$\begin{aligned} \overline{f}(\vec{x}, 0) &= 1 \\ \overline{f}(\vec{x}, y+1) &= \overline{f}(\vec{x}, y) * [g(\overline{f}(\vec{x}, y), \vec{x}, y)] \end{aligned}$$

from which we see that $\overline{f}$ is primitive recursive. Secondly, the primitive recursiveness of f itself follows from the equation

$$f(\vec{x}, y) = g(\overline{f}(\vec{x}, y), \vec{x}, y)$$

once we know that $\overline{f}$ is primitive recursive.

The definition of primitive recursion involved defining the value of a function in terms of its immediately preceding value. Item **24** shows that we get an added bonus: the value of a function can be defined in terms of *all* its preceding values.

At this point, we have seen that many of the everyday functions on the natural numbers are primitive recursive. But the class of primitive recursive functions does not include *all* of the functions on $\mathbb{N}$ that one would regard as effectively calculable. W. Ackermann showed how to construct an effectively calculable function that grows faster than any primitive recursive function. Also, we can "diagonalize out" of the primitive recursive functions. In rough outline, here is how that would go: Any primitive recursive function is determined by tree, showing how it is built up from initial

functions by the use of composition and primitive recursion. We can, with some effort, code such trees by natural numbers. The "universal" function

$$\Psi(x, y) = \begin{cases} f(x) & \text{if } y \text{ codes a tree for a one-place primitive recursive function } f \\ 0 & \text{otherwise} \end{cases}$$

is effectively calculable (and total). But $\Psi(x, x)+1$ and $1 \dot{-} \Psi(x, x)$ are total effectively calculable functions that cannot be primitive recursive. (See also page 19.)

2.2 Search Operation

We obtain the class of general recursive partial functions by allowing functions to be built up by use of search (in addition to composition and primitive recursion). Search (also called minimalization) corresponds to an unbounded μ-operator. For a $(k + 1)$-place partial function g, we define

$$\mu y[g(\vec{x}, y) = 0] = \begin{cases} \text{the least number } y \text{ such that both } g(\vec{x}, y) = 0 \text{ and} \\ \quad \text{for all } t \text{ less than } y, \text{ the value } g(\vec{x}, t) \text{ is defined} \\ \quad \text{and is nonzero, if there is any such } y \\ \text{undefined, if there is no such } y. \end{cases}$$

This quantity may be undefined for some (or all) values of $\vec{x}$, even if g happens to be a total function.

Example: Assume that we know the following pieces of information about the function g:

$$\begin{array}{ll} g(0, 0) = 7 & g(0, 1) = 0 \\ g(1, 0) \uparrow & g(1, 1) = 0 \end{array}$$

Then $\mu y[g(0, y) = 0]$ is 1, and $\mu y[g(1, y) = 0]$ is undefined.

A k-place partial function h is said to be obtained from g by *search* if the equation

$$h(\vec{x}) = \mu y[g(\vec{x}, y) = 0]$$

holds for all $\vec{x}$, with the usual understanding that for an equation to hold, either both sides are undefined, or both sides are defined and are equal.

Then we say that a partial function is *general recursive* if it can be built up from the zero, successor, and projection functions, where we are allowed to use composition, primitive recursion, and search.

The collection of general recursive partial functions includes all of the primitive recursive functions (which are all total), and more. As an extreme example, the one-place empty function (i.e., the function with empty domain) is a general recursive partial function; it is obtained by search from the constant function $g(x, y) = 3$.

9A. If f is a general recursive partial function, then so are the functions s and p:

$$s(\vec{x}, y) = \sum_{t<y} f(\vec{x}, t) \quad \text{and} \quad p(\vec{x}, y) = \prod_{t<y} f(\vec{x}, t)$$

For any particular $\vec{x}$, these functions are defined either for all y, or for a finite initial segment of the natural numbers.

We define a relation R to be a *general recursive relation* if its characteristic function C_R (which by definition is always total) is a general recursive function. As a special case of search, whenever R is a $(k+1)$-ary general recursive relation, then the k-place function h defined by the equation

$$h(\vec{x}) = \mu y\, R(\vec{x}, y)$$

is a general recursive partial function.

We again have a *substitution rule*: Whenever Q is an n-ary general recursive relation, and $g_1, \ldots, g_n$ are k-place *total* general recursive functions, then the k-ary relation

$$\{\vec{x} \mid \langle g_1(\vec{x}), \ldots, g_n(\vec{x})\rangle \in Q\}$$

is general recursive because its characteristic function is obtained from C_Q and $g_1, \ldots, g_n$ by composition. But this does not necessarily hold if the g_i functions are nontotal. In that case, composition does not give us the full C_Q, but a nontotal subfunction of it.

For example, for any *total* general recursive function f, its graph

$$\{\langle \vec{x}, y\rangle \mid f(\vec{x}) = y\}$$

is a general recursive relation. (Similarly, the graph of any primitive recursive function will be a primitive recursive relation.) We will see later that this can fail in the case of a nontotal function.

Theorem:
(d) *If Q and R are k-ary general recursive relations, then so are $\overline{Q}$, $Q \cap R$, and $Q \cup R$.*
(e) *If Q is a $(k+1)$-ary general recursive relation, then so are the relations*

$$\{\langle \vec{x}, y\rangle \mid (\forall t < y)\, Q(\vec{x}, t)\} \quad \text{and} \quad \{\langle \vec{x}, y\rangle \mid (\exists t < y)\, Q(\vec{x}, t)\}.$$

The proof is unchanged.

Definition-by-cases continues to hold, but we need to be more careful with its proof. Suppose that g is a k-place general recursive partial function, and that Q is a k-ary general recursive relation. Define g^Q by the equation

$$g^Q(\vec{x}) = \begin{cases} g(\vec{x}) & \text{if } Q(\vec{x}) \\ 0 & \text{if not } Q(\vec{x}). \end{cases}$$

Then g^Q is also a general recursive partial function. But we cannot write simply $g^Q(\vec{x}) = g(\vec{x}) \cdot C_Q(\vec{x})$ because there may be some $\vec{x}$ that are not in the domain of

g (so the right-hand side will be undefined) and not in Q (so the left-hand side will be 0). Instead, we can first use primitive recursion to construct the function

$$\begin{aligned} G(\vec{x}, 0) &= 0 \\ G(\vec{x}, y+1) &= g(\vec{x}), \end{aligned}$$

which is like g except that it has a "on–off switch." Then we have the equation

$$g^Q(\vec{x}) = G(\vec{x}, C_Q(\vec{x})).$$

showing that g^Q is a general recursive partial function.

Now if we also have another k-place general recursive partial function f, and we define

$$h(\vec{x}) = \begin{cases} f(\vec{x}) & \text{if } Q(\vec{x}) \\ g(\vec{x}) & \text{if not } Q(\vec{x}) \end{cases}$$

then h is a general recursive partial function because $h(\vec{x}) = f^Q(\vec{x}) + g^{\overline{Q}}(\vec{x})$.

24A. Assume that g is a general recursive partial $(k+2)$-place function, and let f be the unique $(k+1)$-place function for which

$$f(\vec{x}, y) = g(\overline{f}(\vec{x}, y), \vec{x}, y)$$

for all $\vec{x}$ and y. (If g is nontotal, then it is possible that for some values of $\vec{x}$, the quantity $f(\vec{x}, y)$ will be defined only for finitely many y's.) Then f is also a general recursive partial function.

The proof is as before.

It was argued earlier that the collection of primitive recursive functions cannot contain all of the effectively calculable total functions. But Church's thesis implies that the collection of general recursive partial functions *does* contain all of them, as well as the effectively calculable nontotal functions. As indicated informally on page 20, it is not possible to "diagonalize out" of the collection of general recursive partial functions.

Exercises

0. Do you understand primitive recursion? Are you positive? If you are positive, go to Exercise 1.

1. Subtract 1. Go to Exercise 0.

2. Give an example of a nontotal function g such that the function h obtained from g by search

$$h(x) = \mu y[g(x, y) = 0]$$

is total.

3. Give a construction tree in full for multiplication (item **3**).

4. Show that the squaring function $f(x) = x^2$ is primitive recursive by giving a construction tree showing in detail how it can be built up from initial functions by the use of composition and primitive recursion. (At the leaves of the tree, you must have only initial functions; e.g., if you want to use addition, you must construct it.)

5. Show that the function

$$\mathrm{pos}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

is primitive recursive by giving a construction tree.

6. Show that the parity function

$$C_{\mathrm{odd}}(x) = \begin{cases} 1 & \text{if } x \text{ is odd} \\ 0 & \text{if } x \text{ is even} \end{cases}$$

is primitive recursive by giving a construction tree.

7. Show that the function $\langle x, y\rangle \mapsto |x - y|$ is primitive recursive.

8. Show that the function $\langle x, y\rangle \mapsto \max(x, y)$ is primitive recursive.

9. Show that the function $\langle x, y\rangle \mapsto \min(x, y)$ is primitive recursive.

10. Show that there is a primitive recursive function div such that whenever $y > 0$, then

$$\mathrm{div}(x, y) = \lfloor x/y \rfloor.$$

(Here $\lfloor z \rfloor$ is the largest natural number that is $\leq z$, i.e., the result of rounding z down to a natural number.)

11. Show that there is a primitive recursive function rm such that whenever $y > 0$, then

$$\mathrm{rm}(x, y) = \text{the remainder when } x \text{ is divided by } y.$$

12. Extend definition by cases (pages 37 and 48) to definition by many (mutually exclusive) cases.

13. Use Bertrand's postulate to show (by induction) that $p_x \leq 2^{x+1}$, and that equality holds only for $x = 0$.

14. Prove item **18**: The set of sequence numbers is primitive recursive.

15. Show that $(x)_y = (\overline{\mu} e \leq x)\, p_y^{e+1} \mid x$.

16. Show that $\mathrm{lh}(x) = (\overline{\mu} t \leq x)\, p_{t \dot{-} 1} \mid x$, for a sequence number x.

17. Assume that R is a finite k-ary relation on $\mathbb{N}$ (i.e., R is a finite subset of $\mathbb{N}^k$). Show that R is primitive recursive.

18. Assume that f is an eventually constant one-place function (i.e., there is some m such that $f(x + 1) = f(x)$ for all $x \leq m$). Show that f is primitive recursive.

19. Show that the function $g(x) = \lceil \sqrt{x} \rceil$ is primitive recursive. (Here $\lceil z \rceil$ is the result of rounding z *up* to a natural number.)

20. **(a)** Assume that f is a primitive recursive one-place function that is strictly increasing (i.e., $f(x+1) > f(x)$ for all x). Show that the range of f is a primitive recursive set.

(b) Assume that g is a primitive recursive one-place function that is nondecreasing (i.e., $g(x+1) \geq g(x)$ for all x) and unbounded. Show that the range of g is a general recursive set.

21. Assume that h is a finite k-place function (i.e., the domain of h consists of only finitely many k-tuples). Show that h is a general recursive partial function.

22. Is 3 a sequence number? What is lh(3)? Find $(1 * 3) * 6$ and $1 * (3 * 6)$.

23. Show that $*$ is associative on sequence numbers. That is, show that if r, s, and t are sequence numbers, then $(r * s) * t = r * (s * t)$.

24. Establish the following facts.

(a) $x + 1 < p_x$.

(b) $(y)_k \leq y$, and equality holds iff $y = 0$.

(c) $\text{lh}\, x \leq x$, and equality holds iff $x = 0$.

(d) $x \upharpoonright i \leq x$ if $x > 0$.

(e) $\text{lh}(x \upharpoonright i)$ is the smaller of i and $\text{lh}\, x$.