11 Primitive Recursive Functions

While the most convincing definition of mechanical procedures is by means of Turing's concept of abstract machines, the equivalent concept of recursive functions first appeared historically as more or less a culmination of extensions of the simple recursive definitions of addition and multiplication.

Wang, p. 87

A. Definition by Induction

When you first learned about exponentiation you probably were told that

$$x^n = \underbrace{x \cdot x \cdot \cdots \cdot x}_{n \text{ times}}$$

That was suggestive and probably convinced you that you could compute the function. Later you learned a proper definition by induction: $x^0 = 1$ and $x^{n+1} = x^n \cdot x$.

Similarly, the factorial function is usually introduced as $n! = n \cdot (n-1) \cdot \cdots \cdot 2 \cdot 1$. An inductive definition of it would be: $0! = 1$ and $(n+1)! = (n+1) \cdot (n!)$.

In its simplest general form, a definition of a function f by induction from another function g looks like $f(0) = m$ and $f(n+1) = g(f(n))$. We have confidence that this method of definition really gives us a function because we can convince ourselves that the generation of values of f can be matched to the generation of natural numbers and is completely determined at each stage:

To	0	1	2	3	...
Assign	$f(0) = m$	$f(1) = g(m)$	$f(2) = g(f(1))$	$f(3) = g(f(2))$	...

This convincing ourselves cannot be reduced to a proof by induction, for to apply that method here we'd already need to have f in hand.

Moreover, since generating the natural number series is effective (computable), if g is computable then without doubt f will also be computable. So let's consider the functions we can obtain using induction and composition, starting with a few simple indisputably computable functions.

B. The Definition of the Primitive Recursive Functions

The class of functions we described in intuitive terms in Section A is composed entirely of computable functions. But for a function to be computable, there must be an algorithm or procedure for computing it. So in our formal definition of this class of functions we must replace the intuitive, semantic ideas of Section A with precise descriptions of the functions, exactly as we did in Chapter 9.

To begin, we take as variables the letters $n, x_1, x_2, \ldots$ though we will continue to use x, y, and z informally. We'll write $\vec{x}$ for $(x_1, \ldots, x_k)$.

Next we list the basic, incontrovertibly computable functions that we will use as building blocks for all others.

1. Basic (initial) functions

zero $\quad Z(n) = 0$ for all n

successor $\quad S(n) = \begin{cases} \text{that number which follows } n \\ \text{in the natural number series} \end{cases}$

projections $\quad P_k^i(x_1, \ldots, x_k) = x_i \quad$ for $1 \le i \le k$

We sometimes call the projections the *pick-out* functions, and P_1^1 the *identity* function, written $id(x) = x$. We don't say that $S(x) = x + 1$ because addition is a more complicated function which we intend to define.

Next, we specify the ways we allow new functions to be defined from ones we already have.

2. Basic operations

Composition

If g is a function of m-variables and $h_1, \ldots, h_m$ are functions of k variables, which are already defined, then composition yields the function

$$f(\vec{x}) = g(h_1(\vec{x}), \ldots, h_m(\vec{x}))$$

Primitive recursion

For functions of one variable the schema is:

$$f(0) = d$$
$$f(n+1) = h(f(n), n)$$

where d is a number and h is a function already defined.

For functions of two or more variables, if g and h are already defined then f is given by *primitive recursion on h with basis g* as:

$$f(0, \vec{x}) = g(\vec{x})$$
$$f(n+1, \vec{x}) = h(f(n, \vec{x}), n, \vec{x})$$

[The reason we allow n and $\vec{x}$ as well as $f(n, \vec{x})$ to appear in h is that we may wish to keep track of both the stage we're at and the input, so that we can have, for example, $f(5,47) = f(10,47)$, but $f(6,47) \neq f(11,47)$.]

3. An inductive definition of the class of functions

Finally, we complete the definition by stipulating that the *primitive recursive functions* are exactly those which are either basic or can be obtained from the basic ones by a finite number of applications of the basic operations. This is an *inductive definition* of the class of functions. To see that, think of assigning:

0	to all the basic functions
1	to all those functions which can be obtained by one or no application of a basic operation to functions which have been assigned 0 (so the basic functions are also assigned 1)
⋮	
$n+1$	to all those functions which can be obtained by at most one application of a basic operation to functions which have been assigned a number less than $n+1$.

Then a function is primitive recursive if and only if it is assigned some number n.

Another way the class of primitive recursive functions is sometimes described is by saying that it is the *smallest* class containing the basic functions and *closed under* the basic operations, where "smallest" is understood to mean the set-theoretic intersection and "closed under" means that whenever one of the operations is applied to elements of the set, the resulting object is also in the set. That way of talking presupposes that the entire completed infinity of the class of functions exists as an intersection of other infinite classes of functions, whereas the inductive definition is nothing more than a constructive way of handing out the label "primitive recursive" to various functions. Since we wish to avoid the use of infinities in our analysis of computability, when we speak sometimes of a class closed under an operation we will understand that as shorthand for an inductive definition.

Thus, to demonstrate that a function is primitive recursive we need to show that it has a description, a definition that precisely fits the criteria above. Though here, as for Turing machine computable functions, if a function has one definition then it will have arbitrarily many others (Exercise 9).

C. Examples

1. The constants

For any natural number n, the function $\lambda x\, f(x) = n$ can be defined as

$$\lambda x\ \underbrace{S(S(\ldots S(Z(x))\ldots))}_{n\ S\text{'s}}$$

But the use of "..." is precisely what we are trying to avoid. We define inductively a sequence of functions: $C_0 = Z$; $C_{n+1} = S \circ C_n$, so that $\lambda x_1\, C_n(x_1) = n$.

Here again we have defined the natural numbers by a unary representation, reflecting that "zero and the idea of one more", rather than "whole number and zero", is our primitive concept.

2. Addition

We can define $x + n$ by viewing it as a function of one variable, n, with the other variable held fixed as parameter. That is, we define addition by x, $\lambda n\,(x + n)$, as:

$$x + 0 = x$$
$$x + (n + 1) = (x + n) + 1$$

But that's not a proper definition according to our description of primitive recursive functions. So let's try again:

$$+(0, x) = x$$
$$+(n + 1, x) = S(+(n, x))$$

This seems like a careful formal definition, but it still doesn't have the required form. A definition that exactly fits the criteria given in Section B for a function to be classified as primitive recursive begins by first defining $S(P_3^1(x_1, x_2, x_3))$, which is primitive recursive since it's a composition of initial functions. Then,

$$+(0, x_1) = P_1^1(x_1)$$
$$+(n + 1, x_1) = S(P_3^1(+(n, x_1), n, x_1))$$

3. Multiplication

Now that we have addition, we can give an inductive definition of multiplication. We use x as a parameter to define $x \cdot n$ as multiplication by x, so $x \cdot 0 = 0$ and $x \cdot (n + 1) = (x \cdot n) + x$. Or, to write it in functional notation,

$$\cdot(0, x) = 0 \quad \text{and} \quad \cdot(n + 1, x) = +(x, \cdot(n, x))$$

This definition looks formal enough, but again it isn't in a form specified in Section B necessary to justify that multiplication is primitive recursive. The first homework exercise below is for you to give such a definition.

4. Exponentiation

Formally in our system of primitive recursive functions we define:

$$Exp(0, x_1) = 1$$
$$Exp(n + 1, x_1) = h(Exp(n, x_1), n, x_1)$$

where

$$h(x_1, x_2, x_3) = \cdot\,(P_3^1(x_1, x_2, x_3), P_3^3(x_1, x_2, x_3))$$

5. Signature and zero test

The signature function is:

$$sg(0) = 0$$
$$sg(n+1) = 1$$

The zero test function is:

$$\overline{sg}(0) = 1$$
$$\overline{sg}(n+1) = 0$$

Exercise 2 asks for definitions of these functions that fulfill the criteria of Section B.

6. Half

We can't divide odd numbers by 2, but we can find the largest natural number less than or equal to one-half of n:

$$half(n) = \begin{cases} \frac{n}{2} & \text{if } n \text{ is even} \\ \frac{n-1}{2} & \text{if } n \text{ is odd} \end{cases}$$

To give a primitive recursive definition of this function, we first need to be able to separate out the case where n is odd:

$$Odd(n) = \begin{cases} 1 & \textit{if } n \text{ is odd} \\ 0 & \textit{if } n \text{ is even} \end{cases}$$

We ask you to show that *Odd* is primitive recursive in Exercise 3. Then

$$half(0) = 0 \quad \text{and} \quad half(n+1) = h\,(half(n)\,,\, n\,)$$

where

$$h(x_1, x_2) = +\,(P_2^1(x_1, x_2),\ Odd\ (P_2^2(x_1, x_2)))$$

7. Predecessor and limited subtraction

In order to define addition, we started with the successor function which adds 1. To define subtraction, we start with the predecessor function which subtracts 1, namely, $P(0) = 0$; $P(n+1) = n$. Exercise 4 below asks you to show that this function is primitive recursive.

Since we can't define subtraction on the natural numbers, we define *limited subtraction*:

$$x \dot{-} n = \begin{cases} x - n & \textit{if } n \le x \\ 0 & \textit{if } n > x \end{cases}$$

Keeping x fixed, as n increases the value of $x \dot{-} n$ goes down until 0 is reached. So we can define: $x \dot{-} 0 = x$; $x \dot{-} (n+1) = P(x \dot{-} \mathrm{n})$, which you can convert into a correct formal definition (Exercise 4).

Exercises Part 1

1. Give a definition of multiplication as a primitive recursive function that precisely fits the specifications of Section B. Compare that definition to the Turing machine definition in Chapter 9.C.
2. Demonstrate that $\overline{sg}$ and sg are primitive recursive.
3. Show that *Odd* is primitive recursive.
4. Show that the predecessor and limited subtraction functions are primitive recursive.
5. Give a primitive recursive definition of the factorial function of Section A.
6. Demonstrate that the following functions are primitive recursive:

$$<(x,y) = \begin{cases} 1 & \text{if } x<y \\ 0 & \text{if } x \geq y \end{cases} \qquad E(x,y) = \begin{cases} 1 & \text{if } x=y \\ 0 & \text{if } x \neq y \end{cases}$$

7. Show that the function f "defined" by $f(n) = 0 + 1 + \cdots + n$ is primitive recursive.
8. Denote the *maximum* of $x_1, \ldots, x_n$ by $max(x_1, \ldots, x_n)$. Show that this is primitive recursive. (*Hint*: See Section C.1; there is one function for each n.)
9. Show that if f has a primitive recursive definition, then there are arbitrarily (countably) many other primitive recursive definitions that give rise to f.

†10. A famous function defined by induction is the Fibonacci series:

$$1, 1, 2, 3, 5, 8, 13, \ldots, \quad u_{n+2} = u_{n+1} + u_n$$

To calculate u_n we need to know what's been calculated in the previous two steps, which, backtracking, we can do once we get to the first two terms. Devise a definition of $f(n) = u_n$ as a primitive recursive function.

D. Other Operations That Are Primitive Recursive

We wouldn't be surprised if you had difficulty showing that the Fibonacci series (Exercise 10) is primitive recursive. It's clearly computable, but primitive recursion allows us to use only the last value of the function at the induction stage, not the previous two values. Rather than tackle that function, it would be much more useful to show that any definition that begins with primitive recursive functions and uses any of the previous values of the function at the induction step in a primitive recursive fashion always results in a primitive recursive function.

We call an operation *primitive recursive* if whenever it is applied to primitive recursive functions it yields a primitive recursive function. In that case it can be simulated by using composition, primitive recursion, and auxiliary primitive recursive functions. In this section we're going to show that the operation described above, and others, are legitimate ways to form primitive recursive functions from primitive recursive functions.

1. Addition and multiplication of functions

If f and g are primitive recursive, then $f+g$ is primitive recursive, where $(f+g)(x) = f(x) + g(x)$ (composition of primitive recursive functions). Similarly $(f \cdot g)(x) = f(x) \cdot g(x)$ is primitive recursive if f and g are. Generally, we define

$$\sum_{i=1}^{n} f_i(\vec{x}) \equiv_{\text{Def}} f_1(\vec{x}) + \cdots + f_n(\vec{x})$$

$$\prod_{i=1}^{n} f_i(\vec{x}) \equiv_{\text{Def}} f_1(\vec{x}) \cdot \cdots \cdot f_n(\vec{x})$$

In Exercise 11 we ask you to give a correct inductive definition of these that does not use "..." and to show that for each n, if $f_1, \ldots, f_n$ are primitive recursive, so are $\sum_{i=1}^{n} f_i(\vec{x})$ and $\prod_{i=1}^{n} f_i(\vec{x})$.

2. Functions defined according to conditions

As an example consider

$$f(n) = \begin{cases} 2n & \text{if } n \text{ is even} \\ 3n & \text{if } n \text{ is odd} \end{cases}$$

Here we are thinking of all numbers as divided into two sets: A = evens, and B = odds. We need only an informal notion of set here, for we don't need to be given all numbers at once to divide them up. All we need is that the following functions are primitive recursive:

$$Odd(n) = \begin{cases} 1 & \text{if } n \text{ is odd} \\ 0 & \text{if } n \text{ is even} \end{cases}$$

and

$$Even(n) = \overline{sg}\,[\text{Odd}(n)] = \begin{cases} 0 & \text{if } n \text{ is odd} \\ 1 & \text{if } n \text{ is even} \end{cases}$$

(See Exercises 2 and 3).

The *characteristic function of a condition* (or, informally, of a set) A is:

$$C_A(x) = \begin{cases} 1 & \text{if } x \text{ satisfies the condition} \\ 0 & \text{if } x \text{ does not satisfy the condition} \end{cases}$$

We say a *condition* (*set*) is *primitive recursive* if its characteristic function is primitive recursive.

Suppose we have n primitive recursive conditions $A_1, \ldots, A_n$ such that every number x satisfies one and only one of these (e.g., odd/even). (Informally, we have a disjoint (nonoverlap) partition (dividing up) of all natural numbers into sets $A_1, \ldots, A_n$.) Suppose further that we have n primitive recursive functions $h_1, \ldots, h_n$. We may define:

$$f(x) = \begin{cases} h_1(x) & \text{if } x \text{ satisfies } A_1 \\ \vdots & \vdots \\ h_n(x) & \text{if } x \text{ satisfies } A_n \end{cases}$$

which is then primitive recursive: f is $\sum_{i=1}^{n} h_i \cdot C_{A_i}$

Often we need nonconstructive proofs to *demonstrate* that every x satisfies exactly one of $A_1, \ldots, A_n$. But that's *outside* the system and does not affect whether a function is primitive recursive or not. Remember, we are viewing functions extensionally.

As an example, we can show that given a primitive recursive function g, the following function is primitive recursive:

$f(0) = x_0$
$f(1) = x_1$
$\vdots$
$f(n) = x_n$
and for $x > n$, $f(x) = g(x)$.

We may always specify the value of a function at some arbitrary number of places before we give a general procedure: That's like providing an accompanying table of values. From our extensional point of view:

FINITE = *TRIVIAL*
GENERAL METHOD = *for all but finitely many*

3. Predicates and logical operations

We can have conditions involving more than one number; for instance "$x < y$" or "$max(x, y)$ is divisible by z". We call a condition which is either satisfied or is not satisfied by every k-tuple of numbers a *predicate* or *relation* of k variables. For instance $R(x, y)$ defined as $x < y$ is satisfied by $(2,5)$ and is not satisfied by $(5,2)$. We say that $R(2,5)$ *holds* (or is true), or we simply write "$R(2,5)$", and "not $R(5,2)$". Another example is the predicate $Q(x,y,z)$ defined as $x + y = z$. Then $Q(2,3,5)$ but not $Q(5,2,3)$. We usually let capital letters stand for predicates.

As for sets, we define *the characteristic function of predicate R* as:

$$C_R(\vec{x}) = \begin{cases} 1 & \text{if } R(\vec{x}) \\ 0 & \text{if not } R(\vec{x}) \end{cases}$$

W say that a predicate is *primitive recursive* if its characteristic function is. We can view sets as predicates of one variable.

Given two predicates we can form new ones; for example, from "x is odd" and "x is divisible by 7" we can form:

"x is odd *and* x is divisible by 7"
"x is odd *or* x is divisible by 7"
"x is *not* divisible by 7"

Given two predicates P, Q we write

$$P(\vec{x}) \wedge Q(\vec{x}) \equiv_{\text{Def}} \vec{x} \text{ satisfies } P \text{ and } \vec{x} \text{ satisfies } Q$$

$$P(\vec{x}) \vee Q(\vec{x}) \equiv_{\text{Def}} \vec{x} \text{ satisfies } P \text{ or } \vec{x} \text{ satisfies } Q, \text{ or } \vec{x} \text{ satisfies both } P \text{ and } Q$$

$$\neg P(\vec{x}) \equiv_{\text{Def}} \vec{x} \text{ does not satisfy } P$$

$$P(\vec{x}) \rightarrow Q(\vec{x}) \equiv_{\text{Def}} \vec{x} \text{ does not satisfy } P \text{ or } \vec{x} \text{ satisfies } Q$$

(In Chapter 19.C we will suggest that we can read $P \rightarrow Q$ as "if P, then Q".) We needn't require that P and Q use the same number of variables. For example, "x is odd and $x < y$" will be viewed as a predicate of 2 variables: (x, y) will be said to satisfy "x is odd" if x does.

In Exercise 14 we ask you to show that if P, Q are primitive recursive then so are all the above.

We may recast these ideas in terms of sets. Given A and B, define

$$A \cap B = \{ x : x \in A \wedge x \in B \}$$
$$A \cup B = \{ x : x \in A \vee x \in B \}$$
$$\overline{A} = \{ x : x \notin A \}$$

If A and B are primitive recursive, so are these sets.

4. Bounded minimization

If we have a computable function, we ought to be able to check what we know about it up to some given bound. We say that *f is obtained from h by the operation of bounded minimization* if

$$f(\vec{x}) = \mathit{min}\ y \leq n\ [\, h(\vec{x}, y) = 0 \,]$$

where this means:

the least $y \leq n$ such that $h(\vec{x}, y) = 0$ if there is one; n otherwise

Note that n is fixed for all $\vec{x}$; that is, we have a different function for each n.

There are two ways we can show that this is a primitive recursive operation. We could define a different function for each n, and then show by induction on n that each is primitive recursive. But in general we want to show something more, namely that there is one primitive recursive function which calculates them all. We say that the functions $h_1, \ldots, h_n, \ldots$ are *uniformly* primitive recursive if there is a primitive recursive function q such that for all n, $h_n(\vec{x}) = \lambda \vec{x}\, q(n, \vec{x})$ (cf. Exercise 12a).

In this case we define $min\, y \le n\, [\, h(\vec{x}, y) = 0\,]$ as $q(n, \vec{x})$ where

$$q(0, \vec{x}) = 0$$
$$q(n+1, \vec{x}) = q(n, \vec{x}) + sg(h(\vec{x}, q(n, \vec{x})))$$

Although the bound needs to be fixed for each $\vec{x}$, it needn't be the same for all $\vec{x}$: if h and g are primitive recursive, then so is f where $f(\vec{x}) = min\, y \le g(\vec{x})\, [\, h(\vec{x}, y) = 0\,]$. Moreover, we can check more than just whether an output of h equals 0. If the function g and the predicate Q are primitive recursive, then so is f, where $f(\vec{x}) = min\, y \le g(\vec{x})\, [Q(\vec{x}, y)]$, usually written as $f(\vec{x}) = min\, y_{\, y \le g(\vec{x})}\, [Q(\vec{x}, y)]$ (Exercise 15). We also ask you to show the same when "$\le$" is replaced by "$<$".

5. Existence and universality below a bound

We can view bounded minimization as a way to deal with questions of existence below a bound. For a predicate P we define:

$\exists y \le n\ P(\vec{x}, y) \equiv_{\text{Def}}$ there is a $y \le n$ such that $P(\vec{x}, y)$

and

$\forall y \le n\ P(\vec{x}, y) \equiv_{\text{Def}}$ for all $y \le n$, $P(\vec{x}, y)$

In Exercise 15 you're asked to show that these are primitive recursive predicates if P is.

6. Iteration

Iteration is the simplest form of definition by induction. Informally, the iteration of the function h is

$$f(n, x) = h^{(n)}(x) = \underbrace{h(h(\ldots h(x)\ldots))}_{n \text{ times}}$$

This way of describing f is only suggestive. Including $n = 0$, we define:

$$h^{(0)}(x) = x$$
$$h^{(n+1)}(x) = h(h^{(n)}(x))$$

Then we say that *f arises by iteration from h* if $f(0, x) = id(x)$ and $f(n+1, x) = h^{(n+1)}(x) = h(P_3^{\,1}(f(n, x), n, x))$
which is in a correct form to demonstrate that it is primitive recursive.

7. Simultaneously defined functions

Sometimes we define two functions f and g together, so that at stage $n+1$ the value of each depends on the previous values of both of them (cf. the prices on the stock exchanges in Chicago and New York). More precisely, let k, q, h, and t

be primitive recursive. We define:

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

We ask you to show that these are primitive recursive in Exercise 18.

8. Course-of-values induction

Up to this point we have only used the single previously calculated value $f(n)$ in calculating $f(n+1)$. This corresponds to simple induction. Using any of the previously calculated values corresponds to a proof by *course-of-values induction*: Given a statement $A(n)$,

if $A(0)$, and for all n, if all $y \leq n$ $A(y)$, then $A(n+1)$;
then for all n, $A(n)$

Course-of-values induction can be reduced to simple induction by applying simple induction to $\forall y \leq n\ A(y)$.

Similarly, we wish to show that a definition of a function that can use all of its previously calculated values, which we call a *course-of-values recursion*, can be reduced to primitive recursion. To do that we code up the previous values of the function into one function, since we can't have a varying number of variables in the inductive step. Let p_n be the nth prime: $p_0 = 2$, $p_1 = 3$, $p_2 = 5$, ... Let f be the function that has a course-of-values definition. Define f^* by:

$$\begin{aligned} f^*(0,\vec{x}) &= 1 \\ f^*(n+1,\vec{x}) &= p_n^{f(n,\vec{x})+1} \cdot \dots \cdot p_1^{f(1,\vec{x})+1} \cdot p_0^{f(0,\vec{x})+1} \\ &= p_n^{f(n,\vec{x})+1} \cdot f^*(n,\vec{x}) \end{aligned}$$

Any definition of $f(n+1,\vec{x})$ that uses the values $f(0,\vec{x})$, $f(1,\vec{x})$, ... , $f(n,\vec{x})$ in some primitive recursive auxiliary function can be defined by extracting those values from $f^*(n+1,\vec{x})$. And we can simultaneously define f and f^*, so long as our coding and uncoding procedure on the primes is primitive recursive (see Exercise 18).

E. Prime Numbers for Codings

1. We first want to show that the function $p(n)$ = the nth prime p_n is primitive recursive. We begin by noting that

m divides n (written "$m \mid n$") iff $\exists i \leq n\ (m \cdot i = n)$

is a primitive recursive predicate by Section D.5; we denote its characteristic function as $d(m,n)$. Then

n is a prime iff $(1 < n) \wedge [\forall x < n\,(x = 1 \vee \neg (x \mid n))]$

is a primitive recursive predicate (via Section D.5). Denote its characteristic function:

$$prime\,(n) = \begin{cases} 1 & \text{if } n \text{ is prime} \\ 0 & \text{if } n \text{ is not prime} \end{cases}$$

By Euclid's theorem (Exercise 6 of Chapter 5) we know that if p is prime then there is another prime between p and $p! + 1$. We define the auxiliary function $h(z) = min\, y_{\; y \leq z!+1}\, [\, z < y \wedge prime(y) = 1\,]$. Then we can define the function p by $p(0) = 2$, $p(n+1) = h(p(n))$.

2. If we code along primes and are given a number, say 270, we need to know the exponents of the primes in its decomposition: $270 = 2^1 \cdot 3^3 \cdot 5^1$. Let

$[\,x\,]_n =$ the exponent of the n^{th} prime in the prime decomposition of x

That this is a well-defined function depends on the fact that every natural number has a unique decomposition into primes (Exercise 4 of Chapter 5). To show that $\lambda x\,[\,x\,]_n$ is primitive recursive we note that $p_n^{[x]_n}$ divides x, but $p_n^{[x]_n + 1}$ does not. So $[\,x\,]_n = \; min\, y < x\;\; [d(p(n)^{y+1}, x) = 0]$.

3. We define the *length* of x to be

$lh(x) \; = \; min\, y < x \;\; (\,[\,x\,]_y = 0\,)$

This measures the number of different primes in a row beginning with 2 that have non-zero exponents in the prime decomposition of x. For example,

$$\begin{aligned} lh(6) &= lh(2 \cdot 3) = lh(p_0 \cdot p_1) = 2 \\ lh(21) &= lh(3 \cdot 7) = lh(p_1 \cdot p_3) = 0 \\ lh(42) &= lh(2 \cdot 3 \cdot 7) = lh(p_0 \cdot p_1 \cdot p_3) = 2 \end{aligned}$$

4. We need to be able to code 0, but $p_n^0 = 1$ and we can't tell if p_n is "there" or not. So we code y into a number via p_n^{y+1}. To uncode we then need the function

$$(x)_n \; = \; [x]_n \dot{-} 1 \; = \begin{cases} \text{1 less than the exponent of the } n \text{ th prime} \\ \text{in the prime decomposition of } x \end{cases}$$

Note that for $x > 0$, $(x)_n < x$. We write $(x)_{n,m}$ for $((x)_n)_m$.

5. Now we have a way to code finite sequences of numbers into single numbers. We code $(a_0, a_1, \ldots, a_n)$ by:

$$\langle\, a_0, a_1, \ldots, a_n \,\rangle = p_0^{a_0+1} \cdot p_1^{a_1+1} \cdot \cdots \cdot p_n^{a_n+1}$$

For each n this is primitive recursive (Exercise 19).

With the convention that every number codes only up to its length, we also have a unique sequence assigned to each natural number:

x codes the sequence $((x)_0, (x)_1, \dots, (x)_{lh(x)-1})$,
where if $lh(x) = 0$, x codes the empty sequence

This is the coding we will use throughout this text.

We needn't have that $x = \langle (x)_0, (x)_1, \dots, (x)_{lh(x)-1} \rangle$, for example, $756 = 2^2 \cdot 3^3 \cdot 7$ which codes the sequence $(1,2)$, but $\langle 1,2 \rangle = 108$. Different numbers can code the same sequence.

6. We can, if we wish, give a 1-1 coding, though it's harder to construct and use. Recall the pairing function $J(x,y) = {}^1/_2 [(x+y)(x+y+1)] + x$ of Chapter 6.B.2, which you were asked to show is 1-1 and onto (Exercise 4 of Chapter 6). We can code $(a_0, a_1, \dots, a_n)$ as $J(a_0, J(a_1, \dots, J(a_{n-1}, a_n)) \dots)$. That is, given the coding of n-tuples, J_n, the coding of $n+1$-tuples is $J_{n+1}(a_0, a_1, \dots, a_n) = J(a_0, J_n(a_1, \dots, a_n))$. To uncode we define the unpairing functions

$$K(z) = \mathit{min}\ x \le z\ [\exists y \le z\ (J(x,y) = z)]$$
$$L(z) = \mathit{min}\ y \le z\ [\exists x \le z\ (J(x,y) = z)]$$

These are primitive recursive by Sections D.4 and D.5.

$$K(J(x,y)) = x, \quad L(J(x,y)) = y, \quad J(K(z), L(z)) = z$$

Uncoding longer sequences is left for you as Exercise 21.

F. Numbering the Primitive Recursive Functions

Here is a sketch of how we can computably number the primitive recursive functions.

First we give every initial function a number: $\#(Z) = 11$, $\#(S) = 13$, $\#(P_n^i) = (p_{n+5})^{i+1}$. Then to each operation under which the class is closed we will associate an arithmetical operation. If $\#(g) = a$ and $\#(h) = b$, then the composition $g \circ h$ will have number $2^a \cdot 3^b$. And if $\#(h_1) = a_1$, $\#(h_2) = a_2$, $\dots$, $\#(h_m) = a_m$ and each is a function of k variables and g is a function of m variables, and $\#(g) = b$, then the function $g(h_1(\vec{x}), \dots, h_m(\vec{x}))$ will have number $\#(f) = 2^b \cdot 3^{<a_1, \dots, a_m>}$. Lastly, if $\#(g) = a$ and $\#(h) = b$, and these are functions of the appropriate number of variables, then f defined by primitive recursion on h with basis g will have number $5^a \cdot 7^b$.

Given any primitive recursive definition, we can follow the steps above and obtain a number for the function, which we call an *index*. And given any number, we can decompose it into primes, further decompose the exponents into primes, and so on, until we have an expression consisting only of primes; then we can determine if it corresponds to a definition of a primitive recursive function. Thus, the conditions for a Gödel numbering are satisfied (Chapter 8.C). Moreover, we can check if the definition corresponds to a function of one variable. So we can make a computable list of the primitive recursive functions of one variable: $f_0, f_1, \dots, f_n, \dots$

where f_n is the function which has the n^{th} index. Our list will have repetitions since every primitive recursive function has arbitrarily many different definitions (Exercise 9), and we are really numbering definitions.

G. Why Primitive Recursive ≠ Computable

Consider the function $g(x) = f_x(x) + 1$.

This is computable since our numbering is. Yet it can't be primitive recursive: if it were it would be f_n for some n, and then we would have $g(n) = f_n(n) + 1 = f_n(n)$. We have *diagonalized*.

$$\begin{array}{lllll}
f_0(0) + 1 & f_0(1) & f_0(2) & f_0(3) & \cdots \\
f_1(0) & f_1(1) + 1 & f_1(2) & f_1(3) & \cdots \\
f_2(0) & f_2(1) & f_2(2) + 1 & f_2(3) & \cdots \\
\vdots & & & & \\
f_n(0) & f_n(1) & f_n(2) & \cdots & f_n(n) + 1 \quad \cdots
\end{array}$$

We've made g disagree with every primitive recursive function of one variable by making g disagree with each f_n on the diagonal. Hence, we have found a computable function which is not primitive recursive.

Here is another way to produce a computable function that isn't primitive recursive. Define

$$\begin{aligned}
h(0) &= f_0(0) + 1 \\
h(1) &= f_0(1) + f_1(1) + 1 \\
&\ \vdots \\
h(n) &= f_0(n) + f_1(n) + \cdots + f_n(n) + 1 \\
&\ \vdots
\end{aligned}$$

Again, h is computable since our numbering is. Yet h *dominates* all primitive recursive functions of one variable; that is, if f is a primitive recursive function of one variable, then $f = f_n$ for some n, so for all $x \geq n$, $h(x) > f(x)$. Thus h cannot be primitive recursive.

If we do not yet have all the computable functions, how can we obtain all of them? What further operations or initial functions do we need?

Exercises Part 2

11. For every $n \geq 2$ give proper primitive recursive definitions of

$$\sum_{i=1}^{n} f_i(\vec{x}) \equiv_{\text{Def}} f_1(\vec{x}) + \cdots + f_n(\vec{x}) \quad \text{and}$$

$$\prod_{i=1}^{n} f_i(\vec{x}) \equiv_{\text{Def}} f_1(\vec{x}) \cdot \cdots \cdot f_n(\vec{x})$$

(*Hint:* It's easy for $n = 2$; then proceed by induction.)

12. a. Show that by holding one variable fixed in a primitive recursive function we obtain a primitive recursive function. That is, given that $\lambda n \lambda \vec{x}\, f(n, \vec{x})$ is primitive recursive, show that for every n, $\lambda \vec{x}\, f(n, \vec{x})$ is primitive recursive. (*Hint:* Use Section C.1.)

    b. Use part (a) and Exercise 11 to show that if f is primitive recursive, so are

    $$\lambda \vec{x} \sum_{i=1}^{n} f(i, \vec{x}) \quad \text{and} \quad \lambda \vec{x} \prod_{i=1}^{n} f(i, \vec{x})$$

13. Suppose we have countably many primitive recursive conditions $A_1, \ldots,$ $A_n, \ldots$ such that every x satisfies exactly one of these. And suppose we also have countably many primitive recursive functions $h_1, \ldots, h_n, \ldots$. Let f be defined by $f(x) = h_n(x)$ if A_n is satisfied by x .

    Is f necessarily primitive recursive? Give a proof or a counterexample with appropriate restrictions.

14. Show that if P and Q are primitive recursive conditions, then so are $P \wedge Q$, $P \vee Q$, $\neg P$, and $P \rightarrow Q$.

15. a. Show that if h and g are primitive recursive, then so is f, where $f(\vec{x}) = \min y_{\, y \leq g(\vec{x})} [\, h(\vec{x}, y) = 0\,]$.

    b. Show that if the function g and predicate Q are primitive recursive, then so is f, where $f(\vec{x}) = \min y_{\, y \leq g(\vec{x})} [Q(\vec{x}, y)]$.

    c. Show that if the predicate P and function g are primitive recursive, then so are the predicates $\exists y \leq g(\vec{x})\, [P(\vec{x}, y)]$ and $\forall y \leq g(\vec{x})\, [P(\vec{x}, y)]$.

    d. Repeat parts (a)–(c) with "$\leq$" replaced by "$<$" .

16. Give one application of definition by conditions and one by bounded minimization which show the utility of knowing that these operations are primitive recursive.

17. Show that the function $e(x) = x^x$ is primitive recursive. Describe the function $f(n, x)$ obtained by iteration of e. Calculate $f(3, 2)$, $f(3, 3)$, $f(10, 10)$. Try to describe in informal mathematical notation the function g that arises by iteration of $\lambda x\, f(x, x)$. Calculate $g(3)$.

†18. a. Show that if h, g, and t are primitive recursive then so is f defined by

    $$f(0, \vec{x}) = g(\vec{x})$$
    $$f(1, \vec{x}) = t(\vec{x})$$

    and for $n \geq 1$, $f(n+1, \vec{x}) = h(f(n-1, \vec{x}), n, \vec{x})$

    Use our codings, not simultaneous recursion.

    b. Show that simultaneous definition by recursion (Section D.7) is a primitive recursive operation.

Hint: Set up a new function $j(n, \vec{x}) = \begin{cases} f\left(\frac{n}{2}, \vec{x}\right) & \text{if } n \text{ is even} \\ g\left(\frac{n-1}{2}, \vec{x}\right) & \text{if } n \text{ is odd} \end{cases}$

19. Show that for each n, $\langle a_0, a_1, \ldots, a_n \rangle$ is primitive recursive.
20. Using our code (Section E.5), find $\langle 3,1,0 \rangle$, $\langle 0,0,2 \rangle$, and $\langle 2,1,0,2,2 \rangle$. What sequences are coded by 900 and by 19,600? By $2^{1047} - 1$?
21. In terms of K and L (Section E.6), give a function which outputs the ith element in the sequence represented by $J_{n+1}(a_0, a_1, \ldots, a_n)$.

†22. Let $f(n) =$ the nth digit in the decimal expansion of π; that is, $f(0) = 3$, $f(1) = 1$, $f(2) = 4$, … . Show that f is primitive recursive.

23. Compare the proof that there is a computable function which is not primitive recursive with:
    a. The proof that the reals are not countable.
    b. The proof that there is no set of all sets.
    c. The Liar paradox.

Further Reading

"Mathematical induction and recursive definitions" by R.C. Buck has a good discussion of definition by induction with lots of examples.