14 Primitive recursive functions

The formal theories of arithmetic that we've looked at so far have (at most) the successor function, addition and multiplication built in. But why stop there? Even high school arithmetic acknowledges many more numerical functions. This chapter describes a very wide class of such functions, the so-called primitive recursive ones. But in Chapter 15 we will be able to show that in fact L_A can already express all of them. Then in Chapter 17 we prove that Q, and hence PA, already has the resources to capture all these functions too.

14.1 Introducing the primitive recursive functions

We start by considering two more functions that are very familiar from elementary arithmetic. First, take the factorial function $y!$, where e.g. $4! = 1 \times 2 \times 3 \times 4$. This can be defined by the following two equations:

$$0! = S0 = 1$$
$$(Sy)! = y! \times Sy$$

The first clause tells us the value of the function for the argument $y = 0$; the second clause tells us how to work out the value of the function for Sy once we know its value for y (assuming we already know about multiplication). So by applying and reapplying the second clause, we can successively calculate 1!, 2!, 3!, 4!, ... Our two-clause definition therefore fixes the value of '$y!$' for all numbers y.

For our second example – this time a two-place function – consider the exponential, standardly written in the form 'x^y'. This can be defined by a similar pair of equations:

$$x^0 = S0$$
$$x^{Sy} = (x^y \times x)$$

Again, the first clause gives the function's value for any given value of x and for $y = 0$; and – keeping x fixed – the second clause gives the function's value for the argument Sy in terms of its value for y.

We've seen this two-clause pattern before, of course, in our formal Axioms for the addition and multiplication functions. Informally, and now presented in the style of everyday mathematics (leaving quantifiers to be understood), we have:

$$x + 0 = x$$
$$x + Sy = S(x + y)$$

$x \times 0 = 0$
$x \times Sy = (x \times y) + x$

Three comments about our examples so far:

i. In each definition, the second clause fixes the value of a function for argument Sn by invoking the value of the *same* function for argument n. This kind of procedure is standardly termed 'recursive' – or more precisely, 'primitive recursive'. And our two-clause definitions are examples of *definition by primitive recursion*.[1]

ii. Note, for example, that $(Sn)!$ is defined as $n! \times Sn$, so it is evaluated by evaluating $n!$ and Sn and then feeding the results of these computations into the multiplication function. This involves, in a word, the *composition* of functions, where evaluating a composite function involves taking the output(s) from one or more functions, and treating these as inputs to another function.

iii. Our series of examples, then, illustrates two short *chains* of definitions by recursion and functional composition. Working from the bottom up, addition is defined in terms of the successor function; multiplication is then defined in terms of successor and addition; then the factorial (or, on the other chain, exponentiation) is defined in terms of multiplication and successor.

Here's another little definitional chain:

$P(0) = 0$
$P(Sx) = x$

$x \dot{-} 0 = x$
$x \dot{-} Sy = P(x \dot{-} y)$

$|x - y| = (x \dot{-} y) + (y \dot{-} x)$

'P' signifies the predecessor function (with zero being treated as its own predecessor); '$\dot{-}$' signifies 'subtraction with cut-off', i.e. subtraction restricted to the non-negative integers (so $m \dot{-} n$ is zero if $m < n$). And $|m - n|$ is of course the absolute difference between m and n. This time, our third definition doesn't involve recursion, only a simple composition of functions.

[1] Strictly speaking, we need a proof of the claim that primitive recursive definitions really *do* well-define functions: such a proof was first given by Richard Dedekind (1888, §126) – for a modern version see, e.g., Moschovakis (2006, pp. 53–56).

There are also other, more complex, kinds of recursive definition – i.e. other ways of defining a function's value for a given argument in terms of its values for smaller arguments. Some of these kinds of definition turn out in fact to be equivalent to definitions by a simple, primitive, recursion: but others, such as the double recursion we meet in defining the Ackermann-Péter function in Section 34.3, are not. For a classic treatment see Péter (1951).

These simple examples motivate the following initial gesture towards a definition:

> A *primitive recursive function* is one that can be similarly characterized using a chain of definitions by recursion and composition.[2]

That is a quick-and-dirty characterization, though it should be enough to get across the basic idea. Still, we really need to pause to do better. In particular, we need to nail down more carefully the 'starter pack' of functions that we are allowed to take for granted in building a definitional chain.

14.2 Defining the p.r. functions more carefully

(a) Consider the recursive definition of the factorial again:

$$0! = 1$$
$$(Sy)! = y! \times Sy$$

This is an example of the following general scheme for defining a one-place function f:

$$f(0) = g$$
$$f(Sy) = h(y, f(y))$$

Here, g is just a number, while h is – crucially – a function we are assumed already to know about (i.e. know about prior to the definition of f). Maybe that's because h is an 'initial' function like the successor function that we are allowed just to take for granted. Or perhaps it is because we've already given recursion clauses to define h. Or perhaps h is a composite function constructed by plugging one known function into another – as in the case of the factorial, where $h(y, u) = u \times Sy$.

Likewise, with a bit of massaging, the recursive definitions of addition, multiplication and the exponential can all be treated as examples of the following general scheme for defining two-place functions:

$$f(x, 0) = g(x)$$
$$f(x, Sy) = h(x, y, f(x, y))$$

where now g and h are both functions that we already know about. Three points about this:

i. To get the definition of addition to fit this pattern, we have to take $g(x)$ to be the trivial identity function $I(x) = x$.

[2]The basic idea is there in Dedekind and highlighted by Skolem (1923). But the modern terminology 'primitive recursion' seems to be due to Rósza Péter (1934); and 'primitive recursive function' was first used in Stephen Kleene's classic (1936a).

ii. To get the definition of multiplication to fit the pattern, $g(x)$ has to be treated as the even more trivial zero function $Z(x) = 0$.

iii. Again, to get the definition of addition to fit the pattern, we have to take $h(x, y, u)$ to be the function Su. As this illustrates, we must allow h not to care what happens to some of its arguments. One neat way of doing this is to help ourselves to some further trivial identity functions that serve to select out particular arguments. Suppose, for example, we have the three-place function $I^3_3(x, y, u) = u$ to hand. Then, in the definition of addition, we can put $h(x, y, u) = SI^3_3(x, y, u)$, so h is defined by composition from previously available functions.

So with these points in mind, we now give the official 'starter pack' of functions we are allowed to take for granted:

> The *initial functions* are
>
> i. the successor function S,
> ii. the zero function $Z(x) = 0$, and
> iii. all the k-place identity functions, $I^k_i(x_1, x_2, \ldots, x_k) = x_i$ for each k, and for each i, $1 \leq i \leq k$.[3]

(b) We next want to generalize the idea of recursion from the case of one-place and two-place functions. There's a standard notational device that helps to put things snappily: we write $\vec{x}$ as short for the array of k variables $x_1, x_2, \ldots, x_k$. Then we can generalize as follows:

> Suppose that the following holds:
>
> $$f(\vec{x}, 0) = g(\vec{x})$$
> $$f(\vec{x}, Sy) = h(\vec{x}, y, f(\vec{x}, y))$$
>
> Then f *is defined from* g *and* h *by primitive recursion.*

This covers the case of one-place functions $f(y)$ like the factorial if we allow $\vec{x}$ to be empty, in which case $g(\vec{x})$ is a 'zero-place function', i.e. a constant.

(c) Finally, we need to tidy up the idea of definition by composition. The basic idea, to repeat, is that we form a composite function f by treating the output value(s) of one or more given functions g, g', g'', ... as the input argument(s) to another function h. For example, we set $f(x) = h(g(x))$. Or, to take a slightly more complex case, we could set $f(x, y, z) = h(g'(x, y), g''(y, z))$.

There's a number of equivalent ways of covering the manifold possibilities of compounding multi-place functions. But one standard way is to define what we might call one-at-a-time composition (where we just plug *one* function g into another function h), thus:

[3] The identity functions are also often called *projection* functions. They 'project' the k-vector $\vec{x}$ with components $x_1, x_2, \ldots, x_k$ onto the i-th axis.

> If $g(\vec{y})$ and $h(\vec{x}, u, \vec{z})$ are functions – with $\vec{x}$ and $\vec{z}$ possibly empty – then f *is defined by composition by substituting* g *into* h just if $f(\vec{x}, \vec{y}, \vec{z}) = h(\vec{x}, g(\vec{y}), \vec{z})$.

We can then think of generalized composition – where we plug more than one function into another function – as just iterated one-at-a-time composition. For example, we can substitute the function $g'(x, y)$ into $h(u, v)$ to define the function $h(g'(x, y), v)$ by composition. Then we can substitute $g''(y, z)$ into the defined function $h(g'(x, y), v)$ to get the composite function $h(g'(x, y), g''(y, z))$.

(d) To summarize. We informally defined the primitive recursive functions as those that can be defined by a chain of definitions by recursion and composition. Working backwards down a definitional chain, it must bottom out with members of an initial 'starter pack' of trivially simple functions. At the outset, we highlighted the successor function among the given simple functions. But we have since noted that, to get our examples to fit our official account of definition by primitive recursion, we need to acknowledge some other, even more trivial, initial functions.

So putting everything together, let's now offer this more formal characterization of the primitive recursive function – or the *p.r. functions* as we'll henceforth call them for short):[4]

1. The initial functions S, Z, and I_i^k are p.r.;
2. if f can be defined from the p.r. functions g and h by composition, substituting g into h, then f is p.r.;
3. if f can be defined from the p.r. functions g and h by primitive recursion, then f is p.r.;
4. nothing else is a p.r. function.

We allow g in clauses (2) and (3) to be zero-place, i.e. be simply a constant.

Note, by the way, that the initial functions are total functions of numbers, defined for every numerical argument; also, primitive recursion and composition both build total functions out of total functions. Which means that *all p.r. functions are total functions*, defined for all natural number arguments.

14.3 An aside about extensionality

(a) We should pause here for a clarificatory remark about the identity conditions for functions, which we will then apply to p.r. functions in particular. The general remark is this (taking the case of one-place numerical functions):

> If f and g are one-place total numerical functions, we count them as being the *same* function iff, for each n, $f(n) = g(n)$.

[4] Careful! Many books use 'p.r.' to abbreviate 'partial recursive', which is a quite different idea. Our abbreviatory usage is, however, also a common one.

Recall that in Section 5.4(d) we defined the extension of a function f to be the set of pairs $\langle m, n\rangle$ such that $f(m) = n$. So we can also put it this way: we count f and g as the same function if they have the same extension. The point generalizes to many-place numerical functions: we count them as the same if they match up however-many arguments to values in the same way.

In a word, then, we are construing talk of functions *extensionally*. Of course, one and the same function can be presented in different ways, e.g. in ways that reflect different rules for calculating it. For a trivial example, the function $f(n) = 2n + 1$ is the same function as $g(n) = (n + 1)^2 - n^2$; but the two different 'modes of presentation' indicate different routines for evaluating the function.

(b) Now, a p.r. function is by definition one that *can* be specified by a certain sort of chain of definitions. And so the natural way of presenting such a function will be by giving a definitional chain for it (which makes it transparent that the function *is* p.r.). But the same function can be presented in other ways; and some modes of presentation can completely disguise the fact that the given function is recursive. For a dramatic example, consider the function

> $fermat(n) = n$ if there are solutions to $x^{n+3} + y^{n+3} = z^{n+3}$ (with x, y, z positive integers);
> $fermat(n) = 0$ otherwise.

This definition certainly doesn't reveal on its face whether the function is primitive recursive. But we know now – thanks to Andrew Wiles's proof of Fermat's Last Theorem – that *fermat* is in fact p.r., for it is none other than (i.e. has the same extension as) the trivially p.r. function $Z(n) = 0$.

Note too that other modes of presentation may make it clear that a function is p.r., but still not tell us *which* p.r. function is in question. Consider, for example, the function defined by

> $julius(n) = n$ if Julius Caesar ate grapes on his third birthday;
> $julius(n) = 0$ otherwise.

There is no way (algorithmic or otherwise) of settling what Caesar ate on his third birthday! But despite that, the function $julius(n)$ is plainly primitive recursive. Why so? Well, either it is the trivial identity function $I(n) = n$, or it is the zero function $Z(n) = 0$. So we know that $julius(n)$ must be a p.r. function, though we can't determine *which* function it is from our style of definition.

The key observation is this: *primitive recursiveness is a feature of a function itself, irrespective of how it happens to be presented to us.*

(c) We count numerical *functions* as the same or different, depending on whether their extensions are the same. In the same vein, it is rather natural to treat numerical *properties* as the same or different, depending on whether *their* extensions are the same. And this will be our official line too.

Indeed, if you accept the thesis of Frege (1891), then we indeed have to treat properties and functions in the same way here. For Frege urges us to regard

properties as just a special kind of function – so a numerical property, in particular, is a function that maps a number to the truth-value *true* (if the number has the property) or *false* (otherwise). Which comes very close to identifying a property with its characteristic function – see Section 2.2.

14.4 The p.r. functions are computable

To repeat, a p.r. function f is one that *can* be specified by a chain of definitions by recursion and composition, leading back ultimately to initial functions. But (a) it is trivial that the initial functions S, Z, and I_i^k are effectively computable. (b) The composition of two effectively computable functions g and h is also computable (you just feed the output from whatever algorithmic routine evaluates g as input into the routine that evaluates h). And (c) – the key observation – if g and h are effectively computable, and f is defined by primitive recursion from g and h, then f is effectively computable too. So as we build up longer and longer chains of definitions for p.r. functions, we always stay within the class of effectively computable functions.

To illustrate (c), return once more to our example of the factorial. Here is its p.r. definition again:

$$0! = 1$$
$$(Sy)! = y! \times Sy$$

The first clause gives the value of the function for the argument 0; then – as we said – you can repeatedly use the second recursion clause to calculate the function's value for $S0$, then for $SS0$, $SSS0$, etc. So the definition encapsulates an algorithm for calculating the function's value for any number, and corresponds exactly to a certain simple kind of computer routine.

Thus compare our definition with the following schematic program:

1. $fact := 1$
2. For $y = 0$ to $n - 1$
3. $\quad fact := (fact \times Sy)$
4. Loop

Here *fact* indicates a memory register that we initially prime with the value of 0!. Then the program enters a loop: and the crucial thing about executing a 'for' loop is that the total number of iterations to be run through is fixed in advance. The program numbers the loops from 0, and on loop number k the program replaces the value in the register *fact* with Sk times the previous value (we'll assume the computer already knows how to find the successor of k and do the multiplication). When the program exits the loop after a total of n iterations, the value in the register *fact* will be $n!$.

More generally, for any one-place function f defined by recursion in terms of g and the computable function h, the same program structure always does the trick for calculating $f(n)$. Thus compare

$$f(0) = g$$
$$f(Sy) = h(y, f(y))$$

with the corresponding program telling us what to put in a register *func*:

1. *func* := g
2. For $y = 0$ to $n - 1$
3. *func* := $h(y, func)$
4. Loop

So long as h is already computable, the value of $f(n)$ will be computable using this 'for' loop, which terminates with the required value now in the register *func*.

Exactly similarly, of course, for many-place functions. For example, the value of the two-place function defined by

$$f(x, 0) = g(x)$$
$$f(x, Sy) = h(x, y, f(x, y))$$

is calculated, for given first argument m, by the procedure

1. *func* := $g(m)$
2. For $y = 0$ to $n - 1$
3. *func* := $h(m, y, func)$
4. Loop

which is algorithmic so long as g and h are computable, and which terminates with the value for $f(m, n)$ in *func*.

Now, our mini-program for the factorial calls the multiplication function which can itself be computed by a similar 'for' loop (invoking addition). And addition can in turn be computed by another 'for' loop (invoking the successor). So reflecting the downward chain of recursive definitions

factorial ⇒ multiplication ⇒ addition ⇒ successor

there is a program for the factorial containing *nested* 'for' loops, which ultimately calls the primitive operation of incrementing the contents of a register by one (or other operations like setting a register to zero, corresponding to the zero function, or copying the contents of a register, corresponding to an identity function).

The point obviously generalizes, establishing

Theorem 14.1 *Primitive recursive functions are effectively computable, and computable by a series of (possibly nested) 'for' loops.*

The converse is also true. Take a 'for' loop which computes the value of a one-place function f for argument n by going round a loop n times, on each circuit calling the function h and applying h again to the output of the previous loop. Then if h is p.r., f will be p.r. too. And generalizing, if a function can be computed by a program using just 'for' loops as its main programming structure – with the program's 'built in' functions all being p.r. – then the newly defined function will also be primitive recursive.

This gives us a quick-and-dirty (but reliable!) way of convincing ourselves that a new function is p.r.: *sketch out a routine for computing it and check that it can all be done with a succession of (possibly nested) 'for' loops which only invoke already known p.r. functions: then the new function will be primitive recursive.*[5]

14.5 Not all computable numerical functions are p.r.

(a) We have seen that any p.r. function is mechanically computable. *But not all effectively computable numerical functions are primitive recursive.* In this section, we first make the claim that there are computable-but-not-p.r. numerical functions look plausible. Then we will cook up an example.[6]

We start, then, with some plausibility considerations. We have just seen that the values of a given primitive recursive function can be computed by a program involving 'for' loops as its main programming structure, where each such loop goes through a specified number of iterations. However, back in Section 3.1 we allowed procedures to count as computational even when they don't have nice upper bounds on the number of steps involved. In other words, we allowed computations to involve *open-ended searches*, with no prior bound on the length of search. We made essential use of this permission in Section 4.6, when we showed that negation-complete theories are decidable – for we allowed the process 'enumerate the theorems and wait to see which of φ or $\neg\varphi$ turns up' to count as a computational decision procedure.

[5] We can put all that just a bit more carefully by considering a simple programming language LOOP. A particular LOOP program operates on a set of registers. At the most basic level, the language has instructions for setting the contents of a register to zero, copying contents from one register to another, and incrementing the contents of a register by one. And the *only* important programming structure is the 'for' loop. Such a loop involves setting a register with some initial contents (at the zero-th stage of the loop) and then iterating a LOOP-defined process n times (where on each loop, the process is applied to the result of its own previous application), which has just the effect of a definition by recursion. Such loops can be nested. And sets of nested LOOP commands can be concatenated so that e.g. a loop for evaluating a function g is followed by a loop for evaluating h: concatenation evidently corresponds to composition of functions. Even without going into any more details, it is very easy to see that every LOOP program will define a p.r. function, and every p.r. function is defined by a LOOP program. For a proper specification of LOOP and proofs see Tourlakis (2002); the idea of such programs goes back to Meyer and Ritchie (1967).

[6] Our cooked-up example, however, isn't one that might be encountered in ordinary mathematical practice: it requires a bit of ingenuity to come up with a 'natural' example – see Section 34.3 where we introduce so-called Ackermann functions.

Standard computer languages of course have programming structures which implement just this kind of unbounded search. Because as well as 'for' loops, they allow 'do until' loops (or equivalently, 'do while' loops). In other words, they allow some process to be iterated until a given condition is satisfied – *where no prior limit is put on the number of iterations to be executed.*

Given that we count what are presented as unbounded searches as computations, then it looks very plausible that not everything computable will be primitive recursive.

(b) But we can do better than a mere plausibility argument. We will now *prove*

> **Theorem 14.2** *There are effectively computable numerical functions which aren't primitive recursive.*

Proof The set of p.r. functions is effectively enumerable. That is to say, there is an effective way of numbering off functions f_0, f_1, f_2, ..., such that each of the f_i is p.r., and each p.r. function appears somewhere on the list.

This holds because, by definition, every p.r. function has a full 'recipe' in which it is defined by recursion or composition from other functions which are defined by recursion or composition from other functions which are defined ... ultimately in terms of some primitive starter functions. So choose some standard formal specification language for representing these recipes. Then we can effectively generate 'in alphabetical order' all possible strings of symbols from this language; and as we go along, we select the strings that obey the rules for being a full recipe for a p.r. function (that's a mechanical procedure). That generates a list of recipes which effectively enumerates the p.r. functions f_i, repetitions allowed.

	0	1	2	3	...
f_0	$\underline{f_0(0)}$	$f_0(1)$	$f_0(2)$	$f_0(3)$	...
f_1	$f_1(0)$	$\underline{f_1(1)}$	$f_1(2)$	$f_1(3)$	...
f_2	$f_2(0)$	$f_2(1)$	$\underline{f_2(2)}$	$f_2(3)$	...
f_3	$f_3(0)$	$f_3(1)$	$f_3(2)$	$\underline{f_3(3)}$	...
...	...	...	...	...	↘

Now consider our table. Down the table we list off the p.r. functions f_0, f_1, f_2, ... An individual row then gives the values of f_n for each argument. Let's define the corresponding *diagonal* function, by putting 'going down the diagonal and adding one', i.e. we put $\delta(n) = f_n(n) + 1$ (cf. Section 2.5). To compute $\delta(n)$, we just run our effective enumeration of the recipes for p.r. functions until we get to the recipe for f_n. We follow the instructions in that recipe to evaluate that function for the argument n. We then add one. Each step is entirely mechanical. So our diagonal function is effectively computable, using a step-by-step algorithmic procedure.

By construction, however, the function δ can't be primitive recursive. For suppose otherwise. Then δ must appear somewhere in the enumeration of p.r. functions, i.e. be the function f_d for some index number d. But now ask what the value of $\delta(d)$ is. By hypothesis, the function δ is none other than the function f_d, so $\delta(d) = f_d(d)$. But by the initial definition of the diagonal function, $\delta(d) = f_d(d) + 1$. Contradiction.

So we have 'diagonalized out' of the class of p.r. functions to get a new function δ which is effectively computable but not primitive recursive. ⊠

'But hold on! *Why* is the diagonal function not a p.r. function?' Well, consider evaluating $d(n)$ for increasing values of n. For each new argument, we will have to evaluate a *different* function f_n for that argument (and then add 1). We have no reason to expect there will be a nice pattern in the successive computations of all the different functions f_n which enables them to be wrapped up into a single p.r. definition. And our diagonal argument in effect shows that this can't be done.

14.6 Defining p.r. properties and relations

We have defined the class of p.r. *functions*. Next, without further ado, we extend the scope of the idea of primitive recursiveness and introduce the ideas of *p.r. (numerical) properties* and *relations*.

> A *p.r. property* is a property with a p.r. characteristic function, and likewise a *p.r. relation* is a relation with a p.r. characteristic function.

Given that any p.r. function is effectively computable, p.r. properties and relations are among the effectively decidable ones.

14.7 Building more p.r. functions and relations

(a) The last two sections of this chapter give some general principles for building new p.r. functions and relations out of old ones, and then give examples of some of these principles at work.

These are more 'trivial but tiresome' details which you could in fact cheerfully skip, since we only pick up their details in other sections that you can also skip. True, in proving Gödel's Theorems, we will need to claim that a variety of key functions and relations are p.r.; and our claims will seem more evidently plausible if you have already worked through some simpler cases. It is therefore probably worth skimming through these sections: but if you have no taste for this sort of detail, don't worry. *Don't get bogged down!*

(b) A couple more definitions before the real business gets under way. First, we introduce the *minimization* operator 'μx', to be read: 'the least x such that ...'.

Much later, in Section 34.1, we'll be considering the general use of this operator: but here we will be concerned with *bounded minimization*. So we write

$$f(n) = (\mu x \leq n)P(x)$$

when f takes the number n as argument and returns as value the least number $x \leq n$ such that $P(x)$ if such an x exists, or returns n otherwise. Generalizing,

$$f(n) = (\mu x \leq g(n))P(x)$$

returns as value the least number $x \leq g(n)$ such that $P(x)$ if such an x exists, or returns $g(n)$ otherwise.

Second, suppose that the function f is defined in terms of $k+1$ other p.r. functions f_i as follows

$$\begin{array}{l} f(n) = f_0(n) \text{ if } C_0(n) \\ f(n) = f_1(n) \text{ if } C_1(n) \\ \quad \vdots \\ f(n) = f_k(n) \text{ if } C_k(n) \\ f(n) = a \text{ otherwise} \end{array}$$

where the conditions C_i are mutually exclusive and express p.r. properties (i.e. have p.r. characteristic functions c_i), and a is a constant. Then f is said to be *defined by cases* from other p.r. functions.

(c) Consider the following claims:

A. If $f(\vec{x})$ is an n-place p.r. function, then the corresponding relation expressed by $f(\vec{x}) = y$ is an $n+1$-place p.r. relation.

B. Any truth-functional combination of p.r. properties and relations is p.r.

C. Any property or relation defined from a p.r. property or relation by bounded quantifications is also p.r.

D. If P is a p.r. property, then the function $f(n) = (\mu x \leq n)P(x)$ is p.r. And generalizing, suppose that $g(n)$ is a p.r. function, and P is a p.r. property; then $f'(n) = (\mu x \leq g(n))P(x)$ is also p.r.

E. Any function defined by cases from other p.r. functions is also p.r.

In each case, we have a pretty obvious claim about what can be done using 'for' loops. For example, claim (A) comes to this (applied to one-place functions): if you can evaluate $f(m)$ by an algorithmic routine using 'for' loops, then you can check whether $f(m) = n$ by an algorithmic routine using 'for' loops. The other claims should all look similarly evident. But let's make this an official theorem:

Theorem 14.3 *The claims (A) to (E) are all true.*

Proof for (A) Start with a preliminary result. Put $sg(n) = 0$ for $n = 0$, and $sg(n) = 1$ otherwise. Then sg is primitive recursive. For we just note that

$$sg(0) = 0$$
$$sg(Sy) = SZ(sg(y))$$

where $SZ(u)$ is p.r. by composition, and $SZ(sg(y)) = S0 = 1$. Also, let $\overline{sg}(n) = 1$ for $n = 0$, and $\overline{sg}(n) = 0$ otherwise. Then $\overline{sg}$ is similarly shown to be p.r.

We now argue for (A) in the case where f is a one-place function (generalizing can be left as an exercise). The characteristic function of the relation expressed by $f(x) = y$ – i.e. the function $c(x, y)$ whose value is 0 when $f(x) = y$ and is 1 otherwise[7] – is given by

$$c(x, y) = sg(|f(x) - y|).$$

The right-hand side is a composition of p.r. functions, so c is p.r. ⊠

Proof for (B) Suppose $p(x)$ is the characteristic function of the property P. It follows that $\overline{sg}(p(x))$ is the characteristic function of the property *not-P*, since $\overline{sg}$ simply flips the two values 0 and 1. But by simple composition of functions, $\overline{sg}(p(x))$ is p.r. if $p(x)$ is. Hence if P is a p.r. property, so is *not-P*.

Similarly, suppose that $p(x)$ and $q(x)$ are the characteristic functions of the properties P and Q respectively. $p(n) \times q(n)$ takes the value 0 so long as either n is P or n is Q, and takes the value 1 otherwise. So $p(x) \times q(x)$ is the characteristic function of the disjunctive property of being either P or Q; and by composition, $p(x) \times q(x)$ is p.r. if both $p(x)$ and $q(x)$ are. Hence the disjunction of p.r. properties is another p.r. property.

But any truth-functional combination of properties is definable in terms of negation and disjunction. Which completes the proof. ⊠

Proof for (C) Just reflect that checking to see whether e.g. $(\exists x \leq n)Px$ involves using a 'for' loop to check through the cases from 0 to n to see whether any satisfy Px. Likewise, if f is p.r., checking to see whether $(\exists x \leq f(n))Px$ involves calculating $f(n)$ and then using a 'for' loop to check through the cases from 0 to $f(n)$ to see whether Px holds. It follows that, if f is p.r., then so are both of

$$K(n) =_{\text{def}} (\exists x \leq n)Px$$
$$K'(n) =_{\text{def}} (\exists x \leq f(n))Px.$$

More carefully, suppose that $p(x)$ is P's p.r. characteristic function. And by composition define the p.r. function $h(u, v) = (p(Su) \times v)$. Put

$$k(0) = p(0)$$
$$k(Sy) = h(y, k(y))$$

[7] For those who are forgetful or have been skipping, this isn't a misprint: we are following the minority line of taking '0' to be the positive value for a characteristic function, indicating that the relation in question holds. See Section 2.2.

so we have

$$k(n) = p(n) \times p(n-1) \times \ldots \times p(1) \times p(0).$$

Then k is K's characteristic function – i.e. the function such that $k(n) = 1$ until we get to an n such that n is P, when $k(n)$ goes to zero, and thereafter stays zero. Since k is p.r., K is p.r. by definition.

And to get the generalized result, we just note that $K'(n) = K(f(n))$ so is p.r. by composition.

We also have further similar results for bounded universal quantifiers (exercise!). Note too that we can apply the bounded quantifiers to relations as well as monadic properties; and in the bounded quantifiers we could equally use '$<$' rather than '$\leq$'. ⊠

Proof for (D) Again suppose p is the characteristic function of P, and define k as in the last proof. Then consider the function defined by

$$f(0) = 0$$
$$f(n) = k(n-1) + k(n-2) + \ldots + k(1) + k(0), \text{ for } n > 0.$$

Now, $k(j) = 1$ for each j that isn't P, and $k(j)$ goes to zero and stays zero as soon as soon as we hit a j that is P. So $f(n) = (\mu x \leq n)P(x)$, i.e. $f(n)$ returns either the least number that is P, or n, whichever is smaller. So we just need to show that f so defined is primitive recursive. Well, use composition to define the p.r. function $h'(u, v) = (k(u) + v)$, and then put

$$f(0) = 0$$
$$f(Sy) = h'(y, f(y)).$$

Which proves the first, simpler, part of Fact D. For the generalization, just note that by the same argument we have $f(g(n)) = (\mu x \leq g(n))P(x)$ is p.r. if g is, so we can put $f'(n) = f(g(n))$ and we are done. ⊠

Proof for (E) Just note that

$$f(n) = \overline{sg}(c_0(n))f_0(n) + \overline{sg}(c_1(n))f_1(n) + \ldots + \overline{sg}(c_k(n))f_k(n) + c_0(n)c_1(n)\ldots c_k(n)a$$

since $\overline{sg}(c_i(n)) = 1$ when $C_i(n)$ and is otherwise zero, and the product of all the $c_i(n)$ is 1 just in case none of $C_i(n)$ are true, and is zero otherwise. ⊠

14.8 Further examples

With our shiny new tools to hand, we can finish the chapter by giving a few more examples of p.r. functions, properties and relations:

1. The relations $m = n$, $m < n$ and $m \leq n$ are primitive recursive.

2. The relation $m|n$ that holds when m is a factor of n is primitive recursive.

3. Let $Prime(n)$ be true just when n is a prime number. Then $Prime$ is a p.r. property.[8]

4. List the primes as $\pi_0, \pi_1, \pi_2, \ldots$. Then the function $\pi(n)$ whose value is π_n is p.r.

5. Let $exp(n, i)$ be the – possibly zero – exponent of the prime number π_i in the factorization of n. Then exp is a p.r. function.

6. Let $len(0) = len(1) = 0$; and when $n > 1$, let $len(n)$ be the 'length' of n's factorization, i.e. the number of distinct prime factors of n. Then len is again a p.r. function.

You should pause here to convince yourself that all these claims are true by the quick-and-dirty method of sketching out how you can compute the relevant (characteristic) functions without doing any unbounded searches, just by using 'for' loops.

But – if you insist – we can also do this the hard way:

> **Theorem 14.4** *The properties, relations and functions listed in (1) to (6) are indeed all primitive recursive*

Proof for (1) The characteristic function of $m = n$ is $sg(|m-n|)$, where $|m-n|$ is the absolute difference function we showed to be p.r. in Section 14.1. The characteristic functions of $m < n$ and $m \leq n$ are $sg(Sm \dot{-} n)$ and $sg(m \dot{-} n)$ respectively. These are all compositions of p.r. functions, and hence themselves primitive recursive. ⊠

Proof for (2) We have

$$m|n \leftrightarrow (\exists y \leq n)(0 < y \ \wedge \ 0 < m \ \wedge \ m \times y = n).$$

The relation expressed by the subformula after the quantifier is a truth-functional combination of p.r. relations (multiplication is p.r., so the last conjunct is p.r. by Fact A of the last section). So that relation is p.r. by Fact B. Hence $m|n$ is a p.r. relation by Fact C. ⊠

Proof for (3) The property of being $Prime$ is p.r. because

$$Prime(n) \leftrightarrow n \neq 1 \ \wedge \ (\forall u \leq n)(\forall v \leq n)(u \times v = n \rightarrow \ (u = 1 \vee v = 1))$$

and the r.h.s. is built up from p.r. components by truth-functional combination and restricted quantifiers. (Here we rely on the trivial fact that the factors of n cannot be greater than n.) ⊠

[8] Remember the useful convention: capital letters for the names of predicates and relations, small letters for the names of functions.

Proof for (4) The function π_n, whose value is the n-th prime (counting from zero), is p.r. – for consider the definition

$$\pi_0 = 2$$
$$\pi_{Sn} = (\mu x \leq n! + 1)(\pi_n < x \wedge \mathit{Prime}(x))$$

where we rely on the familiar fact that the next prime after n is no greater than $n! + 1$ and use the generalized version of Fact D. ⊠

Proof for (5) By the Fundamental Theorem of Arithmetic, which says that numbers have a unique factorization into primes, this function is well-defined. And no exponent in the prime factorization of n is larger than n itself, so

$$\mathit{exp}(n, i) = (\mu x \leq n)\{(\pi_i^x | n) \ \wedge \ \neg(\pi_i^{x+1} | n)\}.$$

That is to say, the desired exponent of π_i is the number x such that π_i^x divides n but π_i^{x+1} doesn't: note that $\mathit{exp}(n, k) = 0$ when π_k isn't a factor of n. Again, our definition of *exp* is built out of p.r. components by operations that yield another p.r. function. ⊠

Proof for (6) $(\mathit{Prime}(m) \wedge m|n)$ holds when m is a prime factor of n. This is a p.r. relation (being a conjunction of p.r. properties/relations). So it has a p.r. characteristic function which we'll abbreviate $\mathit{pf}(m, n)$. Now consider the function

$$p(m, n) = \overline{sg}(\mathit{pf}(m, n)).$$

Then $p(m, n) = 1$ just when m is a prime factor of n and is zero otherwise. So

$$\mathit{len}(n) = p(0, n) + p(1, n) + \ldots + p(n - 1, n) + p(n, n).$$

So to give a p.r. definition of *len*, we can first put

$$l(x, 0) = p(0, x)$$
$$l(x, Sy) = (p(Sy, x) + l(x, y))$$

and then finally put $\mathit{len}(n) = l(n, n)$. ⊠

And that's *quite* enough to be going on with. All good clean fun if you like that kind of thing. But as I said before, don't worry if you don't. For having shown that these kinds of results *can* be proved, you can now cheerfully forget the tricksy little details of how to do it.