Interlude

Let's pause to draw breath and take stock.

1. In Chapter 1 we met the First Incompleteness Theorem is this rough form: a nice enough theory T (which contains the language of basic arithmetic) will always be negation incomplete – there will always be sentences of basic arithmetic it can neither prove nor disprove.

2. We then noted in Chapter 2 that we can cash out the idea of being a 'nice enough' theory in two ways. We can assume T to be *sound*. Or, retreating from that semantic assumption, we can require T to be a *consistent theory which proves a modest amount of arithmetic*. Gödel himself highlights the second version.

3. Of course, we didn't *prove* the Theorem in either version, there at the very outset. However, in Chapter 3, we waved an arm rather airily at the basic strategy that Gödel uses to establish the Theorem – namely we 'arithmetize syntax' (i.e. numerically code up facts about provability in ways that we can express in formal arithmetic) and then construct a Gödel sentence that is (provably) true if and only if is isn't provable.

4. In Chapter 4 we did a bit better, in the sense that we actually gave a *proof* that a consistent, effectively axiomatized, sufficiently strong, formal theory cannot be negation complete.

   The argument was revealing, as it shows that we can get incompleteness results without calling on the arithmetization of syntax and the construction of Gödel sentences. However, the argument depends on the notion of 'sufficient strength' which is defined in terms of the informal notion of a 'decidable property' (a theory, remember, is sufficiently strong if it captures every decidable property of the natural numbers). And the discussion in Chapter 4 doesn't explain how we can sharpen up that informal notion of a decidable property, nor does it explain what a sufficiently strong theory might look like.

5. We need to get less abstract, and start thinking about specific theories of arithmetic. In Chapter 5, as a warm-up exercise, we first looked at BA, the quantifier-free arithmetic of the addition and multiplication of particular

numbers. This is a negation-complete and decidable theory – but of course it is only complete, i.e. is only able to decide every sentence constructible in its language, because its quantifier-free language is indeed so very limited. However, if we augment the language of BA by allowing ourselves the usual apparatus of first-order quantification, and replace the schematically presented axioms of BA with their obvious universally quantified correlates (and add in the axiom that every number bar zero is a successor) we get the much more interesting Robinson Arithmetic Q.

Since we are considerably enriching what can be expressed in our arithmetic language while not greatly increasing the power of our axioms, it is no surprise that Q is negation incomplete. And we can prove this without any fancy Gödelian considerations. We can easily show, for example, that Q can't prove either $\forall x(0 + x = x)$ or its negation. Q, then, is a very weak arithmetic. Still, it will turn out to be the 'modest amount of arithmetic' needed to get a syntactic version of the First Theorem to fly. We announced (but of course haven't proved) that Q is in fact sufficiently strong: which explains why Q turns out to be so interesting despite its weakness.

6. In Chapter 6, we then moved on to introduce first-order Peano Arithmetic PA, which adds to Q a whole suite of induction axioms (every instance of the Induction Schema). Exploration reveals that this theory, in contrast to Q, is very rich and powerful. We might, pre-Gödel, have very reasonably supposed that it is a negation-complete theory of the arithmetic of addition and multiplication. But the theory is still effectively axiomatized, and the First Theory is going to apply (assuming PA is sound, or is at least consistent). So PA too will turn out to be negation incomplete.

7. There are theories intermediate in strength between Q and PA, theories which have induction axioms but only for wffs up to some degree of quantificational complexity. We will be interested in one such intermediate theory later in these notes (Chapter 16). But the task of Chapter 7 is just to explain this notion of quantificational complexity, and in particular explain what Π_1 wffs are.

Which brings us up to the current point in these notes. To give a sense of direction, let's next outline where we are going in the next five chapters. (Skip if you don't want spoilers!)

8. 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, like the factorial and the exponential.

   Chapter 8 describes a very wide class of numerical functions, the so-called primitive recursive (p.r.) ones. They are a major subclass of the effectively computable functions.

   We also define the primitive recursive properties and relations – a nu-

merical property/relation is p.r. when some p.r. function can effectively decide when it holds.

9. Chapter 9 then shows that L_A, the language of basic arithmetic, can *express* all p.r. functions and relations. Moreover Q and hence PA can *capture* all those functions and relations too (i.e. case-by-case prove wffs that assign the right values to the functions for particular numerical arguments). So Q and PA, despite having only successor, addition and multiplication 'built in', can actually deal with a vast range of functions (at least in so far as they can 'calculate' the value of the functions for arbitrary numerical inputs).

   Note the link with our earlier talk about 'sufficiently strong theories' (Defn. 19). Those, recall, are theories that can capture all effectively decidable properties of numbers. Well, now we are going to show that PA (indeed, even Q) can capture at least all those effectively decidable properties of numbers which are primitive recursive (a very important class). And we'll find that that's enough for the core Gödelian argument to go through.

10. In Chapter 10 we then introduce again the key idea of the 'arithmetization of syntax' by Gödel-numbering which we first met in §§3.3 and 3.4. Focus on PA for the moment, and fix on a suitable Gödel-numbering. Then we can define various numerical properties/relations such as:

    Wff(n) iff n is the code number of a PA-wff;
    Sent(n) iff n is the code number of a PA-sentence;
    Prf(m, n) iff m is the code number of a PA-proof of the sentence with code number n.

    Moreover – the crucial result – these properties/relations are primitive recursive. Similar results obtain for any sensibly axiomatized formal theory.

11. Since *Prf* is p.r., and the theory PA can capture all p.r. relations, there is a wff $\mathsf{Prf}(\mathsf{x}, \mathsf{y})$ which captures the relation *Prf* in the theory. In chapter 11 we use this fact in constructing a Gödel sentence which is true if and only if it is not provable in PA. We can thereby prove the semantic version of Gödel first incompleteness theorem for PA in something close to Gödel's way, assuming PA is sound. The result generalizes to other sensibly axiomatized sound arithmetics that include Q.

12. Then Chapter 12, at last, proves a crucial syntactic version of the First Incompleteness Theorem, again in something close to Gödel's way.

Now read on ...

8 Primitive recursive functions

As we have just noted in the Interlude, the primitive recursive functions form a large subclass of the effectively computable functions. This chapter explains what they are, and proves some elementary results about them.

8.1 Introducing the primitive recursive functions

Let's start by revisiting the basic axioms for *addition* and *multiplication* which we adopted formally in Q and PA. Here again is what they say, but now presented in the style of everyday informal mathematics – for everything in this chapter belongs to informal mathematics. So, leaving quantifiers to be understood in the familiar way, and taking the variables to be running over the natural numbers, the principles are:

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

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

The first of the pair of equations for addition tells us the result of adding zero. The second tells us the result of adding Sy (i.e. adding the successor of y) in terms of the result of adding y. Hence these equations – as we pointed out before – together tell us how to add any of $0, S0, SS0, SSS0, \ldots$, i.e. they tell us how to add *any* number. Similarly, the first of the pair of equations for multiplication tells us the result of multiplying by zero. The second equation tells us the result of multiplying by Sy in terms of the result of multiplying by y. Hence these equations together tell us how to multiply by any of $0, S0, SS0, SSS0, \ldots$, i.e. they tell us how to multiply by *any* number.

Here are two more functions that are familiar from elementary arithmetic. Take the *factorial* function $y!$, where e.g. $4! = 1 \times 2 \times 3 \times 4$. Then the factorial function can be defined by the following two equations:

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

The first equation tells us the conventional value of the factorial function for the argument 0; the second equation 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 equation, we can indeed successively calculate 1!, 2!, 3!, 4! ..., as follows:

$$1! = 0! \times 1 = 1$$
$$2! = 1! \times 2 = 2$$
$$3! = 2! \times 3 = 6$$
$$4! = 3! \times 4 = 24$$

And so on and on it goes. Our two-equation definition is properly called a definition because it fixes the value of '$y!$' for all numbers y.

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

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

Again, the first equation gives the function's value for a given value of x when $y = 0$, and – keeping x fixed – the second equation gives the function's value for the argument Sy in terms of its value for y. The equations determine, e.g., that $3^4 = 3 \times 3 \times 3 \times 3 = 81$.

Three comments about our examples so far. (1) Note that in each definition, the second equation fixes the value of a function for argument Sy by invoking the value of the *same* function for argument y. A procedure where we evaluate a function for one input by calling the *same* function for a smaller input or inputs is standardly termed 'recursive' – and the particularly simple pattern we've illustrated is called, more precisely, 'primitive recursive'. So our two-clause definitions are examples of *definition by primitive recursion*.[1]

(2) Next note, for example, that $(Sy)!$ is defined as $y! \times Sy$, so it is evaluated by evaluating $y!$ and Sy 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.

(3) Our four examples can be arranged into 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, in the second chain, exponentiation) is defined in terms of multiplication and successor.

Here's another short chain of definitions:

[1] "Surely, defining a function in terms of that very same function is circular!" But of course, that isn't quite what's happening. We are fixing the value of the function for one input in terms of its already-settled value for a *smaller* input: and *that* is not circular. Still, strictly speaking, we can ask for a proof confirming that primitive recursive definitions really do well-define functions: such a proof was first given by Richard Dedekind in 1888.

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

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

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

'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 the absolute difference between m and n. This time, our third definition doesn't involve recursion, only a simple composition of functions.

These chains of definitions motivate the following initial way of specifying the p.r. functions:

Defn. 28. *Roughly: a primitive recursive function is one that can be similarly characterized using a chain of definitions by recursion and composition, starting from trivial 'initial functions' like the successor function.*[2]

That is a quick-and-dirty characterization, but it is enough to get across the basic idea we need.

8.2 Defining the p.r. functions more carefully

On the one hand, I suppose you really ought to read this section! On the other hand, *don't* get lost in the details. All we are trying to do here is to give a more careful presentation of the ideas we've just been sketching, and to elaborate that rough Defn. 28.

We have three things to explain more carefully. (a) We need to tidy up the idea of defining a function by primitive recursion. (b) We need to tidy up the idea of defining a new function by composing old functions. And (c) we need to say more about the 'starter pack' of initial functions which we can use in building up a chain of definitions by primitive recursion and/or composition.

We'll take these steps in turn.

(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:

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

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

Here, g is just a number, while h is a two-place function which – crucially – *we are assumed already to know about* prior to the definition of f. Maybe that's because h is an 'initial' function that we are allowed to take for granted; or maybe it's because we've already given recursion clauses to define h; or maybe h is a composite function constructed by plugging one known function into another – as in the case of the factorial, where $h(y, z) = z \times Sy$ (where we take the output from the successor function as one input into the multiplication function).

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 is a one-place function, and h is a three-place function, again functions that we already know about. Three points about this:

i. To get the definition of addition to fit this pattern, with a unary function on the right of the first equation, we have to take $g(x)$ to be the trivial *identity function* $I(x) = x$.

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 $h(x, y, z)$ to be the function Sz. As this illustrates, we must allow h not to care what happens to some of its arguments, while operating on some other argument(s). The conventional way of doing this is to help ourselves to some further trivial identity functions that serve to select out particular arguments. For example, the function I^3_3 takes three arguments, and just returns the third of them, so $I^3_3(x, y, z) = z$. Then, in the definition of addition, we can put $h(x, y, z) = SI^3_3(x, y, z)$, so h is defined by composition from initial functions which we can take for granted.

We can now generalize the idea of a definition by recursion from the case of one-place and two-place functions to cover the case of many-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 $x_1, x_2, \ldots, x_k$ (taking the relevant k to be fixed by context). Then:

Defn. 29. *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.

(b) Now 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:

Defn. 30. *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 $h(g(x, y), g'(y, z))$. (No one promised that these details were going to be exciting!)

(c) So far, so good. Now, the quick-and-dirty Defn. 28 tells us that the primitive recursive functions are built up by recursion and composition, beginning from some 'starter pack' of trivial basic functions. But which functions are they? In fact, we've met all the ones we need:

Defn. 31. *The initial functions are the successor function S, the zero function $Z(x) = 0$ and all the k-place identity functions, $I_i^k(x_1, x_2, \ldots, x_k) = x_i$ for each k, and for each i, $1 \leq i \leq k$.*

These identity functions are also often called *projection* functions (they 'project' the vector with components $x_1, x_2, \ldots, x_k$ onto the i-th axis).

Let's now put everything together. We informally defined the primitive recursive (henceforth, p.r.[3]) 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. Or, working in the opposite direction, from the bottom up, we can give this more formal characterization of the p.r. functions:

Defn. 32. *The p.r. functions are the following:*

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]Terminology alert: some authors writing in this area use 'p.r.' as short for *partial* recursive – a quite different notion!

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 a constant.)

So 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. Let's say:

Defn. 33. *A* definition chain *for the p.r. function f is a sequence of functions $f_0, f_1, f_2, \ldots, f_k$ where each f_j is either an initial function or is defined from previous functions in the sequence by composition or recursion, and $f_k = f$.*

Then every p.r. function is required to have a definition chain in this sense (the chain need not be unique, but the function must have at least one to be p.r.) – which sharpens the informal characterization Defn. 28 which we gave at the end of the previous section.

8.3 How to prove a result about all p.r. functions

The point that every p.r. function has a definition chain means that there is a simple method of proving that every p.r. function shares some feature. Suppose that, for some given property P, we can show the following:

P1. The initial functions have property P.

P2. If the functions g and h have property P, and f is defined by composition from g and h, then f also has property P.

P3. If the functions g and h have property P, and f is defined by primitive recursion from g and h, then f also has property P.

Then P1, P2, and P3 together suffice to establish that all primitive recursive functions have property P.

Why? Well, trek along a definitional chain for a p.r. function f. Each initial function we encounter has property P by P1. By P2 and P3, each definition by recursion or composition which is used in the chain takes us from functions which have property P to another function with property P. So, every function we define as we go along has property P, including the final target function f.

In sum, then: to prove that all p.r. functions have some property P, it suffices to prove the relevant versions of P1, P2 and P3.

For a simple first example, take the property of being a *total function* of the natural numbers, i.e. being a function which outputs a natural number value for any given numerical input. (Example: the function x^2 defined over the natural numbers is total – give it a natural number, and it outputs a natural number; but the function $\sqrt{x}$ defined over the natural numbers is only partial – for input 4 it outputs 2; but 5 has no square root in the natural numbers, and so $\sqrt{5}$ has

no value.) Now, the initial functions are, trivially, total functions of numbers, defined for every numerical argument; also, primitive recursion and composition evidently both build total functions out of total functions. Which means that all p.r. functions are total functions, defined for all natural number arguments.

8.4 The p.r. functions are computable

We now show that every p.r. function is effectively computable. Given the general strategy just described, it is enough to show these:

C1. The initial functions are computable.

C2. If f is defined by composition from computable functions g and h, then f is also computable.

C3. If f is defined by primitive recursion from the computable functions g and h, then f is also computable.

But C1 is trivial: the initial functions S, Z, and I_i^k are all effectively computable by utterly trivial algorithms. And C2, the composition of two computable functions g and h is computable (you just feed the output from whatever algorithmic routine evaluates g as input into the routine that evaluates h).

To illustrate C3, 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. And obviously the argument generalizes to establish C3.

It is worth comparing the informal algorithm that reflects the second, recursive, part definition of the factorial with the following schematic program which takes a number $n > 0$ as input.

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

Here, *fact* is a register that we initially prime with the value of 0!. Then the program enters a loop, updating the contents of *fact* on each iteration. And the crucial thing about executing this kind of *'for' loop* is that the total number of iterations to be run through is bounded in advance: you number the loops from 0, and in executing the loop, you increment the counter by one on each

cycle until you hit the bound set by $n - 1$. So in this case, on loop number k the program replaces the value in the register with Sk times the previous value (we'll assume the computer already knows how to find the successor of k and can do 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 the second clause of

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

with the corresponding program which takes input $n > 0$:

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

Given that h is computable, the value of $f(n)$ will be computable using this bounded 'for' loop that terminates with the required value in the register *func*.

Similarly, of course, for many-place functions. For just one example, take the recursive clause of the definition of the addition function: $x + Sy = S(x + y)$. There is a corresponding program which takes as input a number m and a number $n > 0$ and terminates with the sum $m + n$ in the register *add*:

1. $add := m$
2. For $y = 0$ to $n - 1$
3. $\quad add := S(add)$
4. Loop

In other words, the effect of the definition by recursion can computed by a 'for' loop.

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 as we have just seen can be computed by another 'for' loop (invoking the successor). So reflecting the downward chain of recursive definitions

$$\text{factorial} \Rightarrow \text{multiplication} \Rightarrow \text{addition} \Rightarrow \text{successor}$$

there's 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, giving us

Theorem 22. *Primitive recursive functions are effectively computable by a series of (possibly nested) bounded 'for' loops.*

The converse is also true. Suppose we have a program which sets a value for $f(0)$, and then goes into a 'for' loop which computes the value of a one-place function $f(n)$ (for $n > 0$), a loop which calls on an already-known function which is used on the $k+1$-th loop to fix the value of $f(Sk)$ in terms of the value of $f(k)$. This plainly corresponds to a definition by recursion of f. And generalizing,

Theorem 23. *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 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.

For example, take the two-place function $gcd(x, y)$ which outputs the greatest common divisor of the two inputs. Evidently, a bounded search through cases is enough to do the trick: at its crudest and most inefficient, we can look in turn at all the numbers up and including to the smaller of x and y and see if it divides both. That assures us that $gcd(x, y)$ is p.r. without going through the palaver of actually writing down a suitable definition chain.

8.5 Not all computable numerical functions are p.r.

We have seen that any p.r. function is effectively computable. And most of the ordinary computable numerical functions you already know about from elementary maths are in fact primitive recursive. *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'll cook up an example.

First, then, some plausibility considerations. We've 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. Each loop goes through a specified number of iterations. However, we do allow procedures involving *open-ended searches* to count as effective computations, even if there is no prior bound on the length of search (so long as we know the search will terminate in a finite number of steps). We made essential use of this permission 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.

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 the number of iterations to be executed.*[4]

[4] A complication: I'm told that in a language in the syntactic tradition of C (the likes of

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

True, that is as yet only a plausibility consideration. Our remarks so far leave open the possibility that computations can always somehow be turned into procedures using 'for' loops with a bounded limit on the number of steps. But in fact we can now show that isn't the case:

Theorem 24. *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, a p.r. function 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 such chains of definitions. 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 definition chain for a p.r. function. That generates a list which effectively enumerates the p.r. functions, repetitions allowed.

Now consider the following table:

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

Down the table we list off the p.r. functions f_0, f_1, f_2, ... An individual row then gives the values of a particular f_n for each argument. Let's define the corresponding *diagonal* function, by putting $\delta(n) = f_n(n)+1$. To compute $\delta(n)$, we just run our effective enumeration of the recipes for p.r. functions until we get to

Javascript, C#, PHP, ...), a 'for' loop is treated as a case of a 'while' loop. So I guess I've been thinking all along of Old School languages like Basic or Pascal which explicitly mark the difference between the two kinds of loops. Maybe showing my age here! But the point of principle remains: there is a difference between cases where the bound to a looping procedure is set in advance, and cases where the procedure is allowed to carry on for an indefinite number of iterations.

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, as they say, '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 δ not a p.r. function?' Well, consider evaluating $\delta(n)$ 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.[5]

Finally in this section, it is worth noting that our diagonal argument generalizes to get us a result about computable functions more generally (not just the p.r. ones)

Suppose we can effectively list the recipes for some class C of computable total (i.e. everywhere-defined) functions from natural numbers to natural numbers, where the algorithms compute in turn the functions f_0^C, f_1^C, f_2^C, ... Then again we can define $\delta^C(n) = f_n^C(n) + 1$. This will be a total function, by assumption because f_n^C is everywhere defined, so defined for input n in particular. Again $\delta^C(n) + 1$ is computable — just go along our effectively generated list of algorithms until to yet to the n-th one, and then run that algorithm on input n. But, as before δ^C cannot be one of the f_j^C. In a slogan, we can 'diagonalize out' of class C, and get another computable function.

So C can't contain all the computable (one-place, numerical) total functions. That gives us a theorem:[6]

Theorem 25. *No effective listing of algorithms can include algorithms for* all *the intuitively computable total (one place, numerical) functions.*

[5] To expand that thought a bit, note that in the algorithm to compute a p.r. function, the nesting depth of the for-loops is fixed. But in order to compute the diagonal function δ we have to be able to evaluate in turn the n-th p.r. function for the input n, and as we go down the list we get functions whose algorithms will have loops of varying depths – so our computation of $\delta(n)$ will involve going through a nest of loops of varying depth depending on the input n. (I owe that observation to Henning Makholm.)

[6] Note that the restriction to total functions is doing essential work here. Consider algorithms for *partial* computable functions (the idea is that when the algorithm for the partial function φ_i 'crashes' on input n, $\varphi_i(n)$ is undefined). And consider a listing of algorithms for partial functions. Then the diagonal function $\delta(n) = \varphi_n(n) + 1$ could then consistently appear on the list e.g. as φ_d, if $\varphi_d(d)$ and hence $\varphi_d(d) + 1$ are both undefined.

8.6 Defining p.r. properties and relations

We have defined the class of p.r. *functions*. Finally in this chapter, we extend the scope of the idea of primitive recursiveness and introduce the ideas of *p.r. decidable (numerical) properties* and *relations*.

Now, quite generally, we can tie together talk of functions and talk of properties and relations by using the notion of a *characteristic function*:

Defn. 34. *The characteristic function of the numerical property P is the one-place function c_P such that if m is P, then $c_P(m) = 0$, and if m isn't P, then $c_P(m) = 1$.*

The characteristic function of the two-place numerical relation R is the two-place function c_R such that if m is R to n, then $c_R(m, n) = 0$, and if m isn't R to n, then $c_R(m, n) = 1$.

And similarly for many-place relations. The choice of values for the characteristic function is, of course, entirely arbitrary: any pair of distinct numbers would do. Our choice is supposed to be reminiscent of the familiar use of 0 and 1, one way round or the other, to stand in for *true* and *false*. And our selection of 0 rather than 1 for *true* follows Gödel.

The numerical property P partitions the numbers into two sets, the set of numbers that have the property and the set of numbers that don't. Its corresponding characteristic function c_P also partitions the numbers into two sets, the set of numbers the function maps to the value 0, and the set of numbers the function maps to the value 1. And these are the *same* partition. So in a good sense, P and its characteristic function c_P contain exactly the same information about a partition of the numbers: hence we can move between talk of a property and talk of its characteristic function without loss of information. Similarly, of course, for relations (which partition pairs of numbers, etc.). And we can use this link between properties and relations and their characteristic functions in order to carry over ideas defined for functions and apply them to properties/relations.

For example, without further ado, we now extend the idea of primitive recursiveness to cover properties and relations:

Defn. 35. *A p.r. decidable property is a property with a p.r. characteristic function, and likewise a p.r. decidable relation is a relation with a p.r. characteristic function.*

By way of casual abbreviation, we'll fall into saying that p.r. decidable properties and relations are themselves (simply) p.r.

For a quick example, consider the property of being a *prime* number. Take the characteristic function $pr(n)$ which has the value 0 when n is prime, and 1 otherwise. Now just note that we can evidently compute $pr(n)$ just using 'for' loops (we just do a bounded search through numbers less than n – indeed, no greater than $\sqrt{n}$ – and if we find a divisor of n other than 1, return the value 1, and otherwise return the value 0). So the property of being prime is p.r.